AANNS Explained

PREFACE

This document is an informal explanation, in (intelligent) layman's terms, of the principles of the Acolyte Artificial Neural Net System concept. I will try to explain why I think that the AANNS concept is a major breakthrough in the field of Artificial Intelligence (AI), and why I am confident that the specification (contained in "AANNS SPEC.doc") is sufficient for producing an effective product.

My far from trivial task is to convey the "new". If the "new" were easy to explain then it would have already happened! The revolutionary AANNS concept was formed from a collection of coherent ideas honed over many decades. To me, these ideas are now utterly familiar and obvious but to you, the reader, they may very well appear to be counter-intuitive and the relevance of some of the parts, at first sight, may not seem to relate to the whole. I hope not! I will try my very best to explain things in small steps and ask for your patience in building up "the big picture".

If you already have some technical insight into neural nets, please be careful not to assume that the AANNS is going to be some minor variation to the current technology. My assertion is that academia has missed the point! I have had to go back to reinvent the basics in order to make a neural net system work as the original concept intended, i.e. as a *general* applicable learning / problem solving system.

CONTENT

WHAT IS INTELLIGENCE?

The ANNS is an artificial intelligence system designed so as to exhibit many of the features that we commonly refer to as "intelligence" thus being capable of performing many cognitive tasks that at the moment only humans can perform. But the term "intelligence", even in the field of AI, means different things to different people, so, for a coherent logical design, it is an essential first step to come up with a specific practical definition.

In everyday English, the terms "intelligence" and "intelligent" are used comparatively, most often in an implicit context. For instance, the sentence "She is intelligent." expresses an opinion that a specific person is intelligent compared with others. Just which others depends on the context of the sentence including who said or wrote it.

Generally people, other than computer programmers, judge the degree of intelligence of computer programs by the results produced. If the person judging the program feels that they, themselves, could not produce as good a result, they tend to judge favourably. However, the opinion of anyone in receipt of a bill for zero pounds and zero pence is often that the computer program involved is really stupid. This latter evaluation is probably more appropriate.

There is a philosophical expectation in some professions that there exists the same underlying quality called intelligence which largely determines the quality of results in any cognitive system, be it human or machine. Hans Jürgen Eysenck, the British psychologist, even invented the Intelligence Quotient (IQ) to aid uniform measurement of this quality, (albeit at the time without consideration of machine intelligence). In recent times, the idea of one universal quality has fallen out of favour to some extent and has been supplanted with ideas based on "different intelligences".

Avoiding the trap of specifying a general quality of intelligence, I am delighted to have come up with a technical definition which makes sense when comparing similar systems. In informal terms, a high degree of intelligence is demonstrated by "doing a lot with a little knowledge and without much effort". Formally, in my more precise technical definition, intelligence is measured as :

> **The degree of scope for appropriate behaviour of an agent for any given set of knowledge and any given amount of processing used by that agent.**

In the context of giving answers to problems, "appropriate behaviour" refers to producing solutions which are useful rather than perfect (although perfect is best). For example a slightly slow clock is is almost as useful as a perfectly synchronized one.

The scope refers to how wide the arena is in which the agent can perform usefully. Where we often notice severely limited scope is when computer programs fail to match the versatility of our own abilities. We want a Google search to understand the context of our search, but all it can do is match words to web sites without any semantic understanding and thus produce thousands of unwanted pages.

Counter-intuitive as it might seem, the *more* knowledge needed to provide a solution to a problem, the *less* intelligent the agent. Attempts are being made to ameliorate the lack of intelligence when searching the web, by accruing large databases of questions and appropriate answers as judged by the human users, but whatever the advertising might claim this, by my definition, does not amount to intelligent software.

Also counter-intuitively, the *more* processing needed to provide a solution, again the *less* intelligent the agent. For instance, the current crop of highly successful chess programs, by this definition, are not very intelligent because they rely on massive computing power.

The technical definition above informs my quest for truly intelligent software that can compete with and then out-perform the human agent.

THE VISION

The computer revolution, which we are still living through, has largely automated clerical tasks and routine machine control, but has yet to make a major contribution to tackling tasks where human cognitive flexibility is an essential ingredient. Over more than a century, agricultural jobs have been drastically cut, and the number of manual manufacturing jobs is now diminishing, whereas clerical, managerial and other jobs requiring communications with other humans have burgeoned.

When it was realised that not every problem could, in practice, be solved by computer programmers laboriously codifying a response to every possible input, academics in the field of Artificial Intelligence (AI) were charged with finding ways to simulate the human cognitive abilities. The anticipation was that a mechanised robot could be designed to completely supersede the human operator within a few decades. However, compared with early expectations, AI has massively underachieved. Everyday tasks which humans perform with ease proved, on closer inspection, to be highly complex and far more difficult to simulate than at first thought. Nowadays AI academics are less ambitious, seeking minor advances in the expectation of steady overall progress toward a genuine all-dancing all-singing thinking machine. In contrast, my assertion is that AANNS is a big leap forward.

My study of human cognitive processes has convinced me that most of the gubbins lies in regions of the subconscious brain, each region providing a specialized answer service narrowly related to each separate cognitive skill. For example, a professional punter might habitually ask themselves which horse will win the race, and this, if accurately directed, will trigger a response from the relevant specialist part of the brain. Not that this is easy to do - there are a million and one ways for the pathway to go wrong, so why suspect this mechanism? The answer comes from studying people with brain damage. When a hidden mechanism is working well, there are many possible ways in which a mechanism might operate. But when a mechanism goes wrong, then the underlying workings are often revealed. In this case, the proposed model is highly consistent with the capabilities of autistic individuals known as "savants". Without the confusion of competing tasks (many social), common to the ordinary multi-tasking person, in contrast, savants demonstrate a simple single minded and direct access to the part of their brain dedicated to their talent.

So the idea with the AANNS is to simulate the cognitive powers of a savant with the added feature, because it is artificial, of being able to be trained to perform a vast variety of cognitive tasks.

We now come to the question of how to simulate the gubbins in the dedicated brain region. Apart from seeing neurons and axons exhibit some sort of electrical messaging, the technology does not yet exist to examine what is really going on at a detail level, nor to relate brain activity to exhibited behaviours except in the crudest fashion. Instead of trying to copy the brain directly, we must look to the suitability of artificial neural net technology. I think that our brains are effectively an organisation built from biological neural nets - but I cannot prove this. However, I can argue that collections of artificial neural nets each of quite modest size should theoretically have the power to simulate the functions of the biological equivalent.

My starting point is to use evolutionary logic. In evolution, vast tracts of time allow simple components to evolve into structures that exhibit complex behaviours. The crucial idea to grasp is that very simple components can interact in very many ways to give a huge number of effects, and this is sufficient via natural selection in a step by step progression to produce the results we see. Thus we can be confident that underlying the complex behaviour of evolved mechanisms lie very simple components.

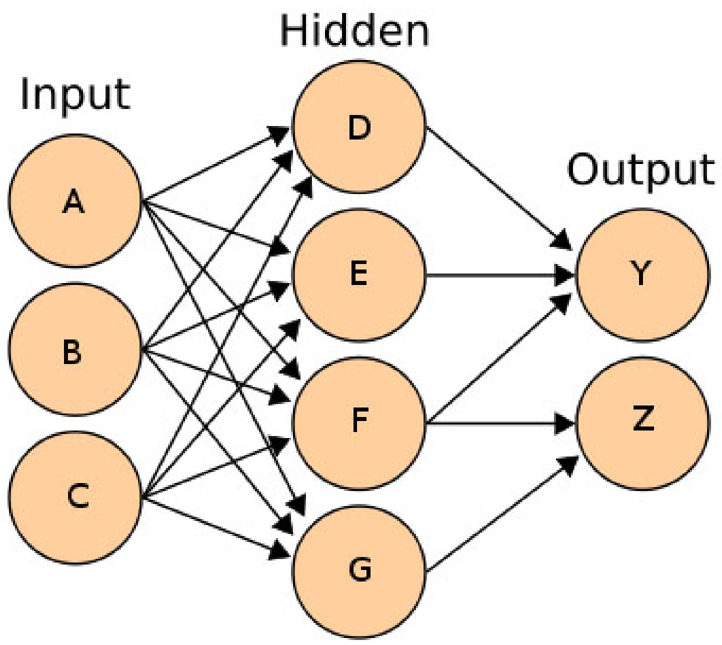
Artificial neural nets are made from nodes and links much in the fashion that neurons and axons make up the main component types in the brain. The number of possible ways of connecting nodes and links are legion. It seems entirely probable that they are theoretically capable of expressing the most complex cognitive behaviour. (Actually this has been formerly proved mathematically.) The not inconsiderable problem that remains is how to train a neural net to form the required specific structure to solve any given type of problem presented to it.

An examination of the commercial neural nets on offer, and a laborious study of current academic papers, does not find systems that are effective except in limited circumstances. The current neural nets seem to do well with some "simple" tasks but fail dismally if presented with a "complex" problem! I shall return to this subject later, but, for now the point is that, in order to have any chance of achieving my goal, I was forced to go back to basics to "reinvent" the neural net afresh.

By making the system generally applicable, in principle, this allows such systems to replace much of conventional computing. More importantly, although not quite stretching to the walking, talking, thinking robot just yet, it greatly expands the scope of potential applications that a computer can handle.

In the next section I will try to explain a simple conventional neural net system, and list the defects found in all publicised current systems that limit their usefulness.

A QUICK INTRO TO NEURAL NETS (with minimal maths)



The above diagram shows the schematic for a very simple artificial neural net. It consists of three **input nodes** A, B and C, connected by **links** to four **intermediate nodes** D, E, F and G, connected by further links to two **output nodes** Y and Z. Each column is referred to as a **layer**, so this schematic shows a three layer net. Very simple neural nets can have just an input and output layer - the minimum to receive and transmit the processed data. However most neural nets need one or more "hidden" intermediate layers in order to express more complex logic.

When fully trained, the resulting neural net (which is dedicated to the one specific application), when presented with data by a computer program called a **neural net interpreter**, produces useful processed numerical output in response to the numerical input in the format dictated by its specified structure (referred to as a **schema**). The input data is presented one record at a time to produce one related output record at a time. This is referred to as the neural net interpreter **expressing** the data.

Input nodes accept number input and transmit each number to each **subordinate link**. Each individual link has a value associated with it called a **weight**. This is used to multiply the incoming number to give a product passed to the next layer of nodes as directed by the links. Each intermediate node in the schematic above will receive the sum of the products of its **superior links**. If this number exceeds the **threshold** for the particular node, that node will transmit the number to its subordinate links. Thus a set of numbers are produced at the output nodes which constitute the answer given to the specific input.

Each neural net starts out life with default values assigned to thresholds and weights. For most conventional systems the number of levels, intermediate nodes and links are manually assigned. The weights and thresholds are then subjected to a training process.

A **neural net trainer** computer program is given a set of **representative data**, records with input values along with their corresponding required output values. Each record is expressed (via the neural net interpreter) to produce output that is very probably in variance to the required output. Very small trial changes are then made to the thresholds and weights of the fledgling net so, that as when the record is expressed again, the new output values are slightly closer to the required answer.

A repetition of this process gradually improves the overall performance of the modified net. Ideally the end result is a net which gives perfect answers to the training data. However, there is no guarantee that "perfection" will be achieved in which case, after giving the process plenty of time, the best net is selected to satisfy the required application.

An added element to training is the automated allocation and organising of intermediate nodes, links and levels. With manual allocation, too few nodes and levels restrict the logic capability of the net, whereas allocating too many nodes and levels allows the net to "remember" each training example by rote and thus failing to capture the more general logic.

How well the resulting net performs on novel data is measured against a reference set of data reserved for the purpose.

From the above account you may discern that training is the key element of a neural net system. As yet, we have barely scratched the surface. We will return to the subject in some considerable detail later on.

Having grasped some of the mechanics, we now turn to neural net applications. We will consider a simple, not very commercial, application as a first example. Suppose we want a computer program to play noughts and crosses (tic tac toe). Technically the most accurate solution would be a computer algorithm using the technique of exhaustive lookahead. However for the purpose of demonstration, we can collect examples of optimal play and use neural net software to train a neural net. With the simple mechanism described above, we could allocate nine input nodes to represent the noughts and crosses board before the move and allocate nine output nodes to indicate the next optional best plays. Input values of zero, one and two could be specified for "empty", cross and nought. Output values of zero and one at the appropriate nodes are sufficient to indicate the optional best moves. There is some non-trivial work involved in collecting the data, putting it into an acceptable format for the neural net trainer, and programming a graphics user interface (a noughts and crosses board). However, at least we do not need to know how to program the game playing algorithm. That bit is magically solved for us!

The important thing to grasp is that the input and output values can be made to represent anything! Albeit some work is needed to translate the numbers into whatever is convenient for the user.

"I think there is a world market for maybe five computers". Attributed to Thomas J. Watson, Chairman of IBM, 1943

"First we thought the PC was a calculator. Then we found out how to turn numbers into letters with ASCII - and we thought it was a typewriter. Then we discovered graphics, and we thought it was a television. With the World Wide Web, we've realized it's a brochure." Douglas Adams

Initially there was a staggering lack of imagination as to the application of the newly invented computer. People failed to realise that numbers can represent almost anything and thus failed to understand that, in principle, a number manipulating/calculating machine can be made to do just about anything. Deja vu! Owing to current limitations, neural net technology is seen as having a rather limited potential in the future. Currently, where they do find a role, it is in the gaps where conventional programming performs badly. Typical applications are face recognition, speech recognition, handwriting and printed text recognition, lip reading, automated surveillance, vehicle control, process control, autonomous robot control, virtual reality interaction, game-playing, medical diagnosis, financial applications, email spam filtering.

The main defects of current neural net systems are :

- owing to the exponential increase in the number of configurations that can be permed from

trial connections as the number of potential connections increases, there is a severe limitation on the capacity to train nets for complex problems.
- an approach to training which, only after much expended effort, eventually discovers that a problem is too complex by failing to generate a useful net.
- a lack of automation in training that requires the developer to experimentally set numbers of nodes, links and levels or, alternatively, use automated systems that perform poorly.
- the propensity of nets to latch on to arbitrary factors whilst missing the main logic required to provide the solution to the problem inputs.
- a general poor level of performance accuracy which severely limits the appropriate use of neural nets to a small selection of types of application.

Academic research is largely aimed at making more and more specialist types of neural nets, and faster training. Both aims promise short-term gains but I have decided to think long-term and row precisely in the opposite direction, i.e. I am seeking more generally capable systems which concentrate on producing neural nets trained to a high degree of accuracy. Within the bounds of practicality, the speed of training is less important.

The following sections describe some of the new AANNS concepts that together overhaul the current neural net technology putting to rights the common defects. I cannot guarantee that there aren't existing neural net systems somewhere that contain some of the invented features. However, having read many current academic papers and articles on the subject, I believe that the AANNS overall concept is probably decades ahead of the field.

THE PARSIMONIOUS STRUCTURE PRINCIPLE

Because the behaviour of complex applications can arguably be characterised as the result of the combination of interactions between simple components, for each individual application, it is mathematically highly probable that there is a single minimal sized configuration of net that forms the perfect solution, i.e. a compact neural net which always gives appropriate output. This structure must be far more compact than one that, although giving perfect answers for the training data, fails to find the generality that would always give the correct answers to novel data. Referring to the definition "**The degree of scope for appropriate behaviour of an agent for any given set of knowledge and any given amount of processing used by that agent**", we can be encouraged that the compact neural net not only exhibits a wider range of appropriate answers, but does so using an efficient structure that uses less processing because the volume of processing is closely proportional to the number of links in the trained neural net.

This consideration leads to two major features in the design of the AANNS which greatly enhance the performance of the resulting trained nets. The first feature is, throughout the training process, to "choke" the dynamic expansion of the net (i.e. when adding links, nodes and levels). At each stage of training, limiting the expansion of the net has the added benefit of limiting the training time which is, very roughly, exponentially proportional to the number of modifiable links. Note that as the training progresses some links and nodes become settled and avoid further trial and error modification. Because expansion can cause the training time to grow exponentially, too much expansion can put effective training beyond feasible time limits, even if using the very latest supercomputer.

The second feature is, in phases, to prune those links and nodes that have the least positive effect on the results from expressing the set of training data. Although this temporarily reduces the training data set performance, it can often actually improve the overall performance if tested against novel data. However, the real point is that it allows alternative trial structural modifications which statistically may have more general applicability. This pruning process is expensive in computer processing terms (although not exponentially so!), but keeping training times low is second to the primary aim of routinely achieving trained nets that perform accurately.

The idea for pruning came out of an understanding of how progressive evolution must work. Evolution can be said to progress when the evolved organisms can prosper in more challenging environments. (Usually that challenge is provided by other competitive evolved organisms.) For this to happen, a level of attrition is necessary favouring the survival of those genes which are generally more useful compared with those which are less so. Note that in long periods of benign environmental stability, via mutations, organisms slowly adapt to niche environments, but then such organisms tend to die out when the environment eventually becomes unstable. There is an optimum rate of development where the environment changes regularly but not by too much (which would result in mass extinctions). Mutations need to be given some time to bed in, but then for long-term survival they benefit from a harsh examination as to their general suitability, thus "phased pruning" is in order. An understanding of this process leads to the realisation that there is a strong association between an evolved organism that can survive in many environments, with a trained neural net that can perform effectively against a wide range of novel data.

The AANNS pruning method can be allowed to be extremely harsh because if it oversteps the mark, unlike in natural evolution where species become irrevocably extinct, the previous trained stages of the net can be resurrected to reverse the detected performance degradation of the over-pruned version. This, in the overall scheme, gives a more rapid rate of progress than a more gentle approach.

THE PARSIMONIOUS USE OF TRAINING DATA PRINCIPLE

There are two main aims when coming to the use of training data. The first is to achieve a high degree of performance, i.e. the resulting net should give a high percentage of correct answers to novel data. The second aim is to train efficiently, i.e. minimise the training processing time necessary to acquire the desired level of performance. How the training records are presented to the neural net trainer can greatly effect the success or otherwise of the training process. Simply cycling around the training data set can merely produce a repetitive cycle of modifications, effectively going nowhere. A rapid training process giving perfect answers for the training data will be likely to perform badly on novel data, and once a training record gets correct results, it no longer has any power to modify. This is all quite tricky stuff!

Conventional training systems present each data record in turn and allow a repetition factor to be manually specified to determine how many modified expressions are tried before turning to the next record. Once all records are processed, the cycle is repeated until satisfactory progress has been made. The manual allocation of a suitable repetition factor requires the user to experiment - a somewhat time consuming and potentially unrewarding exercise. It all smacks of empirical experience without much of an underlying mathematical theory - all a bit of a black art!

Major aims of the AANNS design are to make everything, that can be, completely automated, and to use new training techniques that are both properly understood and also have a high probability of working. Once handed the training data set, the AANNS attempts to progressively drain the maximum training value from that data. Because the process can be lengthy, it can be manually truncated at any point and the extant results used. Alternatively the system can be allowed to continue to make progress until it decides no further progress is mathematically likely.

The first step is for the ANNS to automatically reserve a generous proportion of the training records (referred to as the **internal set**) in a **reference set.** (I am skipping over the tedious detail as to how the AANNS selects which individual records go into which set.) The remaining records in the internal set, which will be used in the actual training process, are referred to as the **load set**. At stages throughout the lengthy training process the reference set is tested to reliably evaluate the performance of the current net. (Obviously data actually used for training is tainted for this purpose.) AANNS goes the extra mile and automatically optimises the selection and size of the reference set :

Each neural net performance evaluation is modified to give a lower bound according to a measure of error dependent on the number of records in the reference set - the more reference set records, the smaller the margin of error. The AANNS can then expand the load set by robbing the reference set, always remembering that the margin allowed for error in the performance calculation will be increased as the reference set decreases. At some point the AANNS can detect the mathematically correct balance between the number of reference and load set records in order to maximise the lower bound performance evaluation figure and thus present the user with the best net possible.

AANNS automatically splits the load set into an **active set** and a **pending set, t**hen further subdivides the active set into a **current active set** and a **recent active set**.

At any one time, AANNS uses only the current active set for training. AANNS strives to keep this set as small as possible commensurate with making progress in improving the performance of the trained net. This wrings the maximum training value from each record whilst fortuitously improving the speed of each training cycle. (Note that when a training record gives completely correct results for the current net structure it can no longer indicate any improvement.)

The recent active set is where training records are temporarily reserved when they provide no immediate training value. This is an important twiddle because not only does it help eke out the training value, but it also allows difficult records to be sidelined avoiding a stall in the upward training progress. This means the net can be effectively trained even given the odd invalid data examples.

The pending set holds all those records not yet used in training. It is heavily relied on as a pseudo reference set in order to frequently evaluate the progress being made. According to results the course of training is altered, and this statistically means there may be, albeit indirect, influence upon the trained net, thus it cannot be said that the pending set is a truly independent untainted reference source. But the pending set is good enough for the purpose of working progress and this allows the proper reference set to be used without feedback influencing the training processing, in order to be certain that the evaluation of the final best net is reliable.

The selection of the records for the current active set is a vital concern because often the amount of training data is somewhat limited and must be used to the maximum, not wasted by a quick training strategy. The aim is to select the lowest number of records, yet still make progress in a trial of what is called a "training expedition" (more later). According to the target, records may be transferred to or from the recent active set to the current active set. Additionally if necessary, and after benefit from numerous contraction and expansion (of the current net structure) is exhausted, new records from the pending set are added until progress is continued. When the current load set is effectively used up, it can be added to by robbing the reference set until further progress is calculated as unlikely.

TRAINING ASCENTS AND EXPEDITIONS

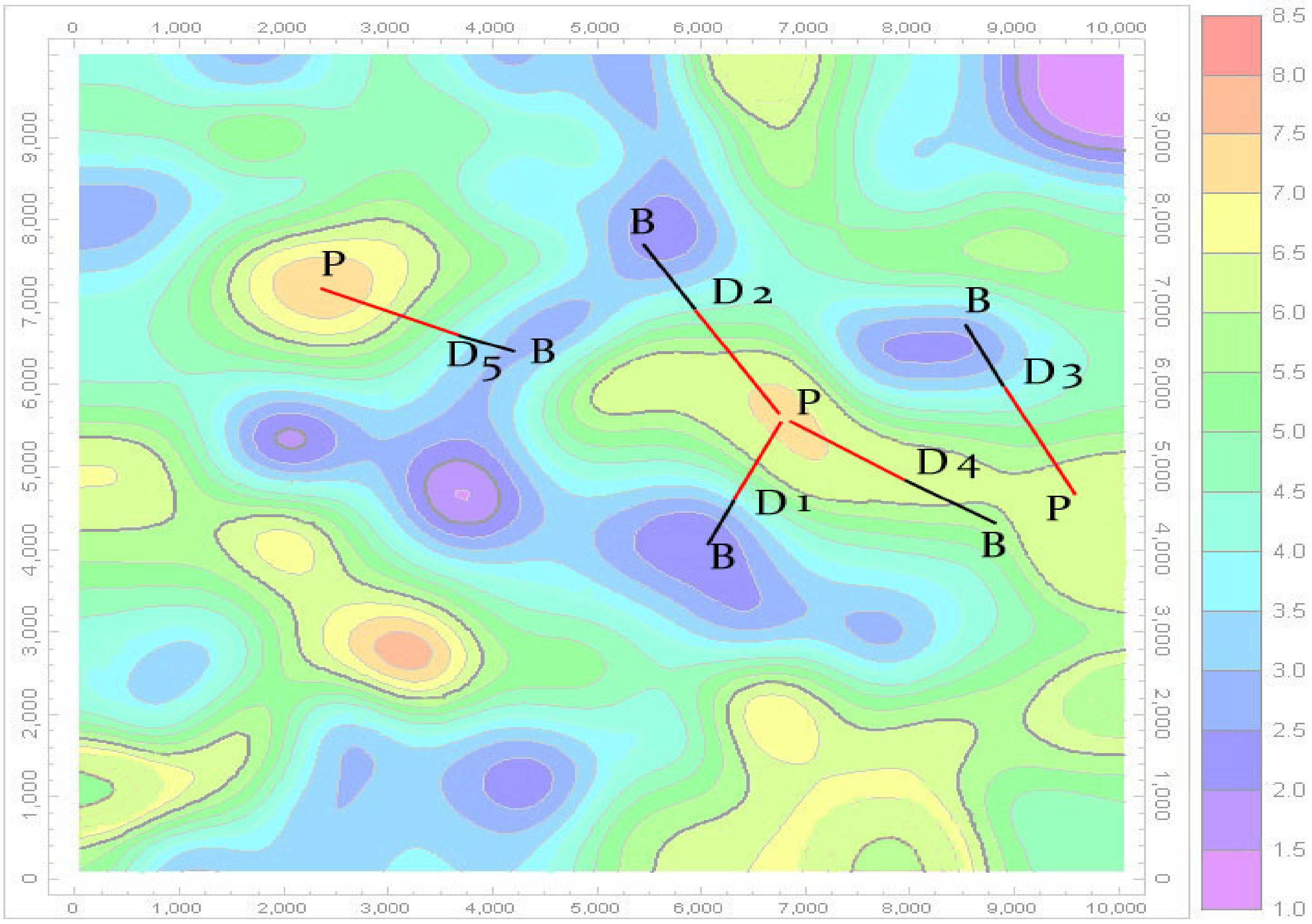
A major feature of AANNS is a logical approach to maximise the chance of finding the peak performance for any one net structure. The concept is an extension of the idea, expounded by Richard Dawkins, of "Climbing Mount Improbable".

If you imagine the peak of Mount Improbable as representing the DNA genome of an organism, the plain from which the mountain rises as representing the DNA genome of simple life forms, and the points on the slopes as contiguous DNA genomes in between, then, because the organism has evolved, there must exist a gentle continuous path from the base to the very top defining the evolutionary history of that organism. (This path may be long and involuted, but evolutionary time-scales are big enough to accommodate such.)

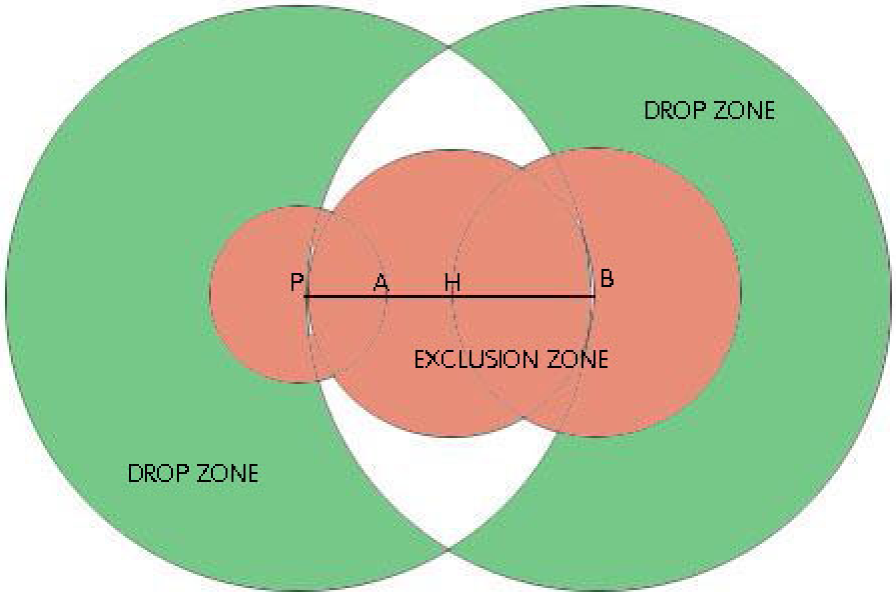
Now replace the DNA genomes with the values of the variables, such as link weights, (usually referred to as **components** but sometimes as **attributes**) in an AANNS neural net plotted against the performance as the height and you have an analogous situation. (The fancy name for this is the **solution attribute topography**.) If this is hard to visualise, imagine just two variables plotted against each other (x and y axes), and against the performance of the net at each point - the z axis representing the height. The fact that there are likely to be many more variables than just two, makes the graph multi-dimensional but the principal of a contiguous landscape still holds mathematically.

In our imagined landscape, there are performance peaks, ridges, troughs, foot hills, plateaus and low lands. Now imagine the whole landscape is covered in cloud and all you have is a limited number of altimeters on parachutes which, when dropped from above (assume perfect vertical descent), can report back both the height and the direction of slope at the landing point. (Note that these limitations reflect the serial processing capabilities of a PC, unlike the parallel processing capabilities of an organic brain.) What is the best strategy to find out where the highest peak is?

Given that the terrain might be a mixture of features, all in many dimensions and to different scales, the search method has a lot to cope with. The secret lies in forming simplified trails, upwards from a drop point to a peak (via a **training ascent**) and extended downward to a base. Then there is a slightly greater statistical likelihood of there being a bigger peak between peaks that are closer to each other than to any other trail path or trail base. Otherwise, there is a lesser chance of there being a bigger peak very near to an existing trail and a slightly greater chance of a fresh upward slope at a commensurate distance away from existing trails. Thus subsequent "modified random" drop points can be chosen, either between peaks or an optimal distance away from existing trails. As the landscape fills up with trails of varying lengths, the chances of finding new peaks diminishes and at some point a statistical judgement can be made to abandon further exploration. The contour diagram below (after five drop points D1 to D5) shows how the base (B) to peak (P) trails progressively occupy the space in a useful fashion.



Each trail contributes a drop zone and an exclusion zone as shown in the above diagram. This pattern arises from a method which greatly simplifies the calculations with the understanding that here approximations do the job just as well. Rather than using complex multi-dimensional geometry, all distances are approximated by summing the absolute differences between the coordinates (expressed in **training step units** - see below) of any two points. For example, the distance between P[3,5,0,9,4] and B[2,6,1,7,8] is 3-2 + 6-5 + 1-0 + 9-7 + 8-4 or PB = 9. To determine whether a potential drop point is within a particular trail's drop zone or an exclusion zone, first distances DB, DP and DH (where B refers to base, P to peak and H to the halfway point between B and P) must be calculated. A further set of coordinates for A, the point on the trail with average height (performance) is needed. As shown in the diagram below, the potential drop point is in the exclusion zone if DP is less than PA, or DH is less than half PB, or DB is less than half PB. Otherwise, the potential drop point is in the trail's drop zone if DP is less than PB and also DB is greater than PB, or DB is less than PB and also DP is greater than PB. However, if BA is less than PA then the B and P points must be transposed. Then the potential drop point is in the trail's exclusion zone if DB is less than BA, or DH is less than half PB, or DP is less than half PB.



In general, drop points are selected randomly from trail drop zones that do not overlap other trails' exclusion zones. The exceptions are the first time (of course!), and when two peaks are closer to each other than any base or halfway point whilst also greater than two training step units apart, in which case the drop point is made midway.

The concept of training step units is needed to unify the disparate scales used by the different neural net variables (components) so the above scheme can work. The primary idea is that each variable must be scaled each by an individual factor such that, on average, the effect of adding or subtracting a fraction of each scaled unit on the change in performance of the net is roughly the same. The only way to achieve this is by trial and error and amassing the necessary statistical data. The method used by AANNS produces this data as a by-product of the training ascents.

The essence of this training step alteration is that occasionally (as determined by a statistical optimisation), a training accent is repeated twice, once with active component training steps increased so as to potentially save a single step, and once with the active components decreased by the same margin. For each trail, basically one of two outcomes are possible: As is likely in early iterations, an entirely new path can be formed, in which case, if the performance level reached is higher than the other two paths, both the path and the new training step sizes are adopted. If however, the original path is repeated (approximately) then the appropriate training step sizes are adopted only if there is an improvement (reduced number of steps) on the previous set of step sizes. Note that reducing the training step size can lead to an improvement by reducing the number of missteps, where the algorithm attempts to re-establish the path assuming that the ascent has overstepped the peak. An important twiddle is that the increase/decrease in training step size is randomly varied by a slight amount so that the correct relationships between component training step sizes are allowed to evolve.

I suppose I'd better explain what an *active* component is. This terminology comes from the consideration of the AANNS back-propagation method. Oops! Now I need to explain back-propagation!

Back-propagation is the process applied to the current net for each training record used in a training ascent. It identifies the adjustment needed for each component (such as link weight) so that the output node target value can be achieved or at least be improved upon. It proceeds from the target output node to its superior intermediate nodes, establishing new targets for the latter, which are then in turn used to propagate back to their superior nodes and so on. Those components that are identified as able to contribute towards an improvement are termed **active** and those identified as incapable are termed **inactive**.

An individual training ascent back-propagates each record in the current active set in order to determine small changes to the current net's link weights (and exponent quantities) which are likely to improve the current active set's performance score. Each "improvement" is applied until no further progress can be made. There is a statistical association such that improvements in the current active set score tends also to improve the pending set score (here used as a surrogate reference set). However, there is usually a point in the sequence of improvements such that the net is over-trained, i.e. the net is trained too specifically for the current active set, away from the generality required. At this point the trail is truncated. By using only the small current active set, the training is efficient and a level of variability is achieved, whilst the inertia of the current net structure along with the testing against the pending set, provides the statistical pressure for upwards evolution.

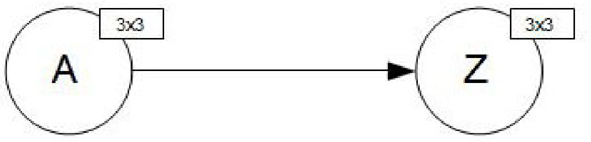
MATCHING NET LOGIC TO DATA TYPES

Many neural net systems concentrate on a limited type of input data, but for a generalised system it is important to cater properly for a wide range. The temptation is to rely on the net to sort out the significance of the numerical data but this can degrade performance, and what is worse, can find spurious logic in data when the magnitude of the numbers is actually arbitrary. The other concern is the wasteful non-use of the information present associated with the topology of the data in arrangements such as grids used for game playing.

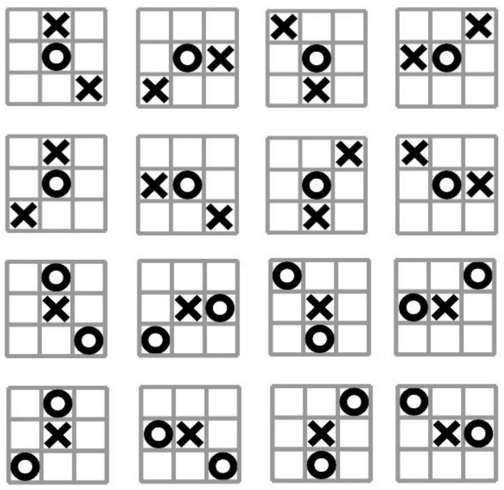
An example of matching structure to input types is, rather than allocate arbitrary numbers to data such as place names, AANNS caters for an **identity** type. Such items of data are actually ignored as not statistically significant if there are too few records present with the specific identity. Otherwise an **indicator** binary node is automatically assigned for the specific value.

It is time to revisit the noughts and crosses (tic tac toe) application example. This is not such a challenging problem for a really good neural net system so how the data is presented to the net and the precise structure of the net may not be crucial. However, it makes for a good illustrative example. Always remember that the facilities described here are for general purpose and not designed specifically for our tic tac toe example.

For a simple neural net system, one would probably have to specify nine numerical input nodes and arbitrarily allocate "0", "1" and "2" values for "empty", "cross" and "nought". In contrast, the AANNS analyst would specify both an input array and an output array each of a 3x3 **square grid** with eight-way spacial symmetry. (The diagram below illustrates the logical net structure before allocation of any intermediate nodes.)

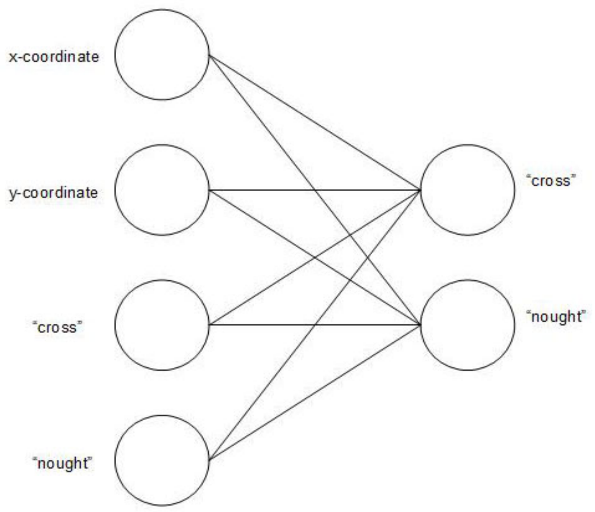


The arrays would be specified as each having two binary elements of the **indicator** data type to represent "cross", "nought" or "empty", the latter by being neither "cross" nor "nought". The binary indicator type is chosen to avoid integers which would otherwise wrongly infer a potential arithmetic relationship. Additionally the analyst would declare the "cross" binary element to be twinned with the "nought" to indicate (two-way) logical symmetry.



The above diagram demonstrates the combined 16 way symmetry for tic tac toe. AANNS detects and rejects symmetrically logical duplicates in the training data. The intermediate node and link structure is automatically configured to give the same logical answer whichever symmetrical version of novel data is presented. Taken in stages this is not so difficult to understand.

First off let us examine the basic logical structure for our example in a little more detail :



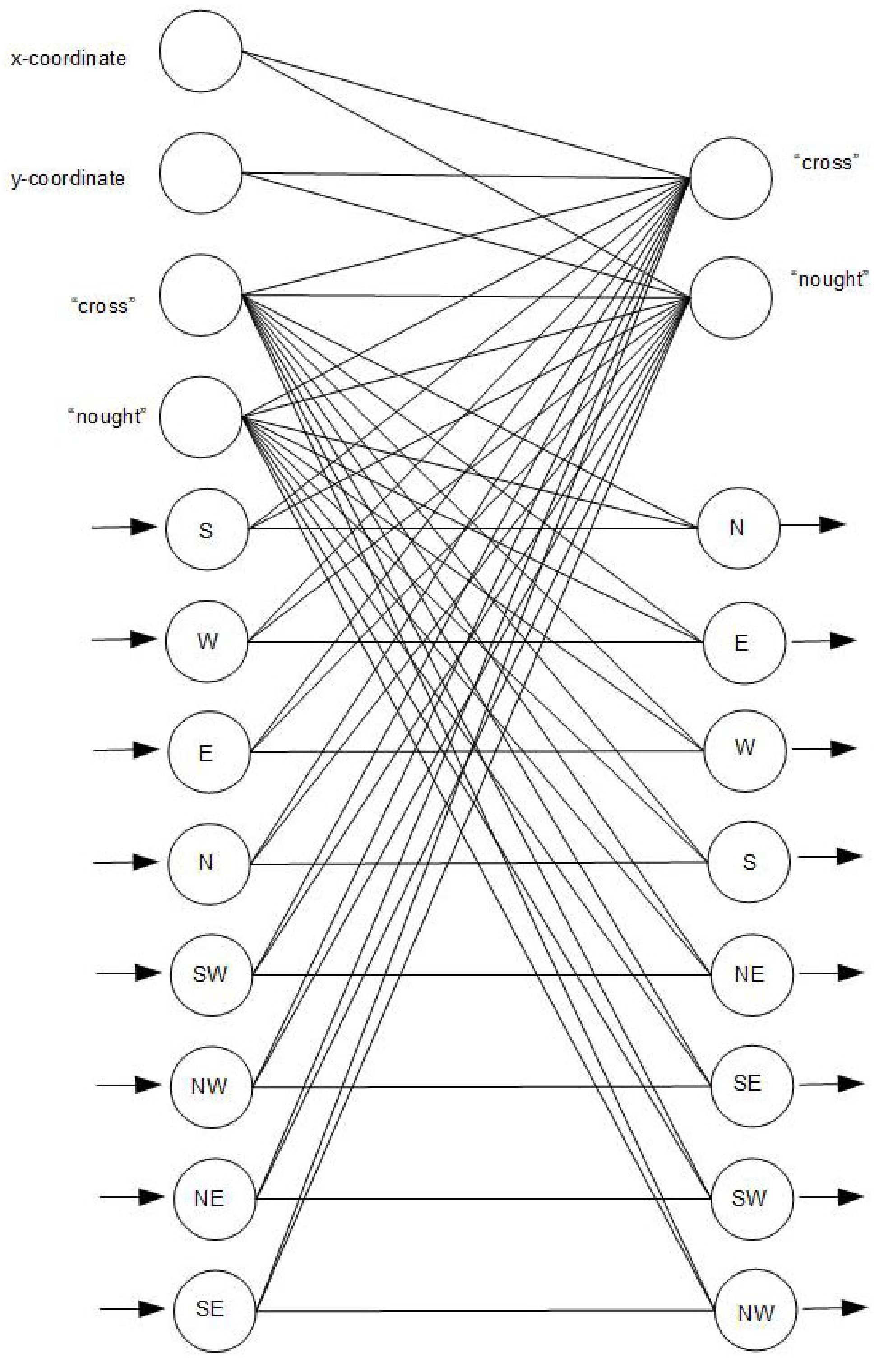
The diagram above shows part of the physical assignment of some of the **secondary nodes** to implement the **primary nodes** in the previous node diagram. The left-hand column represents the A input primary node and the right-hand column the Z output primary node. AANNS uses the same structure for each and every element of the 3x3 array. The content of the x-coordinate and y-coordinate integer nodes varies from element to element. Without symmetry, the content of these nodes would be generated to form conventional column and row numbers. With symmetry, the logical coordinates are as given in the table below :

1,1	1,2	1,1
1,2	2,2	1,2
1,1	1,2	1,1

As regards the "cross" and "nought" nodes, during training, when a new link is added to one of these nodes, then it is automatically added to the other. When a new link and new subordinate node is added, then a new link with a new subordinate node is added to the other. When a link weight is altered on any mirrored link, then the other link is also altered. Similarly, deletions are also automatically managed in tandem.

Whatever the shape of the intermediate structure that evolves, it will be automatically applied to each element of the 3x3 array in the same way. Thus it is impossible to train one symmetrical grid position adequately but not another. But this is not the whole story.

The logical essence of an array is the potential interaction between the elements. To allow the links and intermediate nodes to usefully evolve, the AANNS pre-sets connections between adjacent iterations of the expression of the logical net. The diagram below shows the detail for our tic tac toe example:



Two sets of 4 input and 4 output integer nodes are assigned, the first set for orthogonal connections (N E S W) and the second set for diagonal connections (NE SE SW NW). The easiest way to understand what is going on is to imagine one instance of a specific element expression. AANNS contains iterative logic which connects, for example, the S input node to the N output node from an adjacent intersection. In that adjacent expression, the S input node is connected to another adjacent expression via the adjacent N output node. This continues until the AANNS detects the edge of the array. In other words, the logic is used for each column, row and both diagonals, all in two directions.

Symmetry dictates that whatever happens (regarding links and nodes) connected to one of the direction nodes, it must be equally applied to the others in the set (orthogonal or diagonal), this on top of each link and node update also being equally applied to the "nought" and "cross" nodes owing to their logical twinning.

The full list of topology types it is intended that AANNS covers at a first implementation is :

singleton	single entity - not an array
heap	array with no spatial organisation, each element independent of the others
line	one dimensional array with optional left/right symmetry
square grid	two dimensional square array with up to eight-way symmetry.
rectangular grid	two dimensional rectangular array with up to four-way symmetry

A line is useful, for example, for greyhound race information where the trap position is deemed significant. A rectangular grid would be used for processing any picture information. Additionally variable size arrays can be catered for - AANNS then automatically allocates an array size integer node.

Together, the data types allow a wide range of problems to be efficiently represented. A more general method of topological specification (very difficult!) is deemed not necessary for inclusion in a first implementation.

Please note that the AANNS analyst does not need to know how the nodes and links are organised. This is automatically done. But the analyst does need to understand how to declare the data in the schema. Understanding when to declare an array (rather than say a heap) and with what symmetry is crucial. However, anybody that can grasp the concepts of arrays and of symmetry should have little difficulty in operating the AANNS.

COMPREHENSIVE NET LOGIC

There are many different types of commercial neural net systems, each oriented towards specific types of problem depending on their practical success. If it is to have a chance of coping well across a wide variety of types of problem, a general purpose system must include more types of logical elements than is commonly supplied. We have already some of the relevant AANNS features, but here we will complete the list.

The link weight by modification during training gives the capability of proportional logic. The summation of link products to give node input obviously caters for addition and subtraction. Not mentioned so far, is that then the node input is raised to the power given by a trainable **exponent component**, thus catering for exponential logic. However, one of the most important features is the expression of logic by automatically modifying the link structure. There is complete freedom in allocating levels, because routinely, any node can be modified by addition or removal of links regardless of level. Normally the restriction is that links must flow from input to output without looping, but a backward facing link or loop iteration feature is included for when it is automatically detected to be beneficial. So iterative logic is catered for too!

Another important feature is the initial structure given to the net before training. Singleton input nodes are of course connected directly to each singleton output node. (Intermediate nodes may result from training.) Array input nodes are also connected to all output nodes. For array output nodes the array size is equated to the input array size. For singleton output array nodes connected to array input nodes, the sum of the output from all the element links is taken. Array output nodes are connected directly to all input nodes, but additionally an intermediate singleton node linked to the array output node and all input nodes is created.

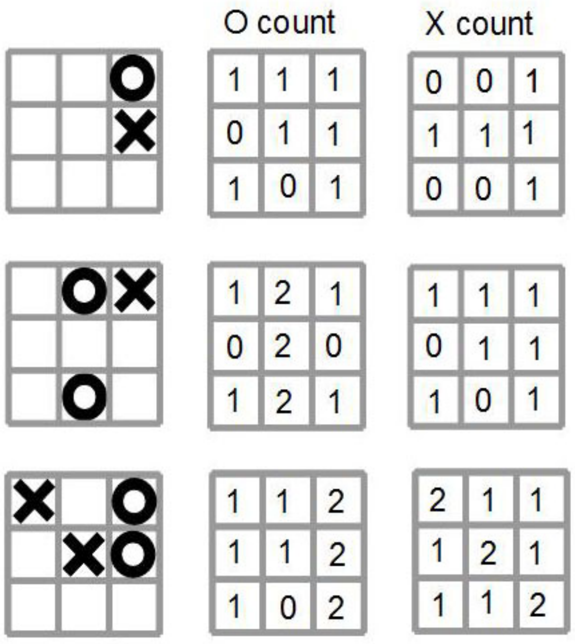
Perhaps the most important feature is that previously trained nets compatible with the input data automatically produce additional input to the net under training. Thus sub-nets effectively act as nodes in a hierarchy of nets. Thus an application can be trained in stages. This is an important concept to grasp. Given realistic time limits for training, the size of individual nets is limited and can solve only relatively easy problems. To solve more complex fare, it is fundamentally necessary to build a net in stages, just as it is necessary to teach a human student by a whole series of lessons, each lesson building on the previous ones. To illustrate this point we shall return to our tic tac toe example:

It is unlikely that our tic tac toe example will require further training, but for the sake of illustration, suppose we are dissatisfied with the achieved performance. What to do?

Assuming there is an adequate amount of training data given an adequate time to train, we must concede that the problem is perhaps too steep for a one hit solution. We must first train our net system to notice things that might help it in its final deliberation. This is best determined by examining the mistakes the net makes. Suppose we find that mistakes occur most often on rows and columns occupied by two symbols (combinations of noughts and crosses). We might guess that the net is only aware of immediate adjacent patterns. Thus we could decide to "pre-train" the net to count the number of each symbol in the best row, column or diagonal. Then on reapplying the original data, there is a much better chance of the net creating the correct logic.

The table below shows a minimal amount of data to create the counting perception :

Note for example, that when the "O" and "X" counts are both equal to 1, the empty position is unsuitable for either side. Also note that a count of 2 for the opposition requires an immediate



block. These observations hint at the possible steps in logic that the net might take.

Having successfully created the sub-net, the original net is again trained using the initial training data. In general, the AANNS analyst need not specify which sub-nets are appropriate for connection to the original net. The AANNS itself automatically detects the suitability of the O and X count output nodes and connects them as **derived input nodes** to the original net. (Matching types of nodes is a complex business which we will skip over here!)

In general it is envisaged that acolyte nets will accrue sub-nets enshrining fundamental perceptions which will be automatically invoked for solving newly presented problems. It is perhaps obvious that our tic tac toe example would be a good sub-net for solving the game "connect five" where the aim is to achieve five symbols in a row on a much larger array (the size of array making the conventional computer technique of brute force lookahead very slow except on a supercomputer). It is less obvious, but actually the case, that the collection of simple "visual logic" nets can underpin very complex problem solving, not dissimilar to human capabilities - for example in deciphering hand writing.

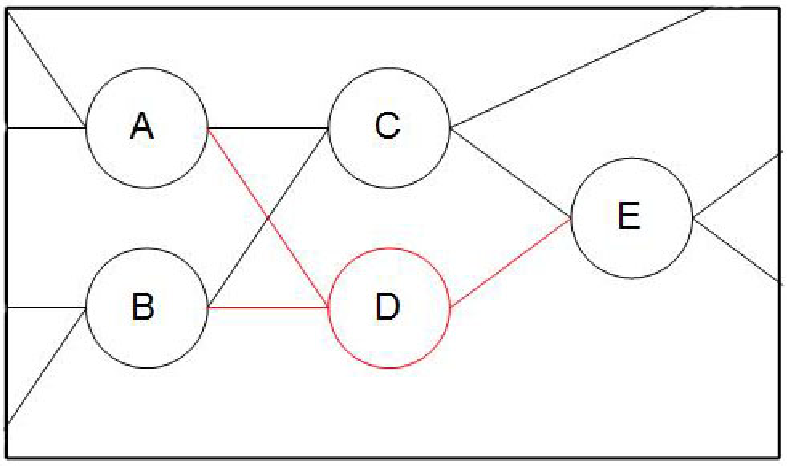
EXPANSION AND CONTRACTION PHASES

The idea of limiting the expansion of the net structure has been introduced in the "THE PARSIMONIOUS STRUCTURE PRINCIPLE" section. The logic for expanding and contracting the neural net during training is quite complex, but to give you, the reader, some flavour of how the AANNS goes about these key tasks, the following simplified overview is offered.

The first thing to understand, is that the AANNS keeps a copy of each **optimal working net structure** as defined by their performance rating and related to their number of links. An optimal working net is one that has a performance rating better than any other net that has a fewer number of links. Thus, if and when a new optimal working net is produced from a training expedition, it may invalidate previously optimal working nets that have insufficient performance ratings, which are then removed from the set of optimal working nets.

After a training expedition, an attempt is made to discover a candidate link for deletion (from any of the set of optimal working nets) that is judged would not degrade performance by too great a margin. If such a link is found, it is removed to form a new net structure which is then trained. However, if such a link is not found, a search is made for a position where a link or a node can be added such that it is deemed potentially to be of benefit. Again, after alteration the new structure is trained. When no new optimal working net is produced and there are no suitable candidate positions for either contraction or expansion, then the active set may be added to from the pending set, or when this is calculated as being unlikely to be beneficial, the load set may be expanded by robbing the reference set. Eventually it will be calculated that robbing the reference set is probably counterproductive, at which point the ANNS ceases training and presents the **best net** for actual use. (Note that the best net may not be one of the current optimal working nets. In order to guarantee validity, best nets use the gold standard of the reference set for evaluation instead of the pending set. They take no part in the selection for the training so that they cannot be compromised. Every newly trained net structure is tested to see if it is the best, i.e. better than the current kept best net, regardless of whether it is optimal or not.)

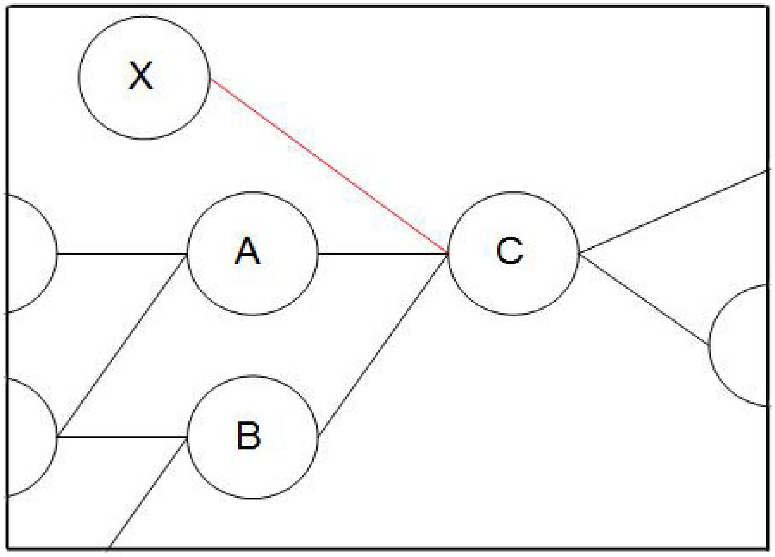
Next we shall turn to a little more detail on how links for trial deletion are chosen :



Consider the above diagram of a subsection of an existing optimal working net. First to notice is that, if the AANNS selects link DE, then links AD and BD along with node D are also made redundant. The fact that three links would be deleted gives a positive factor of three to be included in the calculation as to the desirability of choosing link DE. However the determination of the main criterion for selection candidates is as follows:

A **selection set** is chosen at random from the load set to be expressed and back-propagated. For each record expression, a net score is accumulated as to when the input quantity from the DE link is a help or a hindrance to achieving the back-propagated target input for the E node. The less helpful, the more likely the link will be chosen. The actual odds calculated also depend on the previous success of training for the links chosen according to their helpfulness quotient. Lastly, a randomising factor is included according to how well previous predictions have fared. Thus the more accurate predictions were, the more likely the selection will be rejected if the calculated odds are insufficient. Otherwise more links will tend to be selected.

Not all links are examined. Any selection where removal of links and nodes would produce a net structure that has already been trained, is vetoed. This, of course, includes all links in any one optimal net structure that have previously been chosen but have subsequently failed to produce a new optimal working net. Eventually (or quite soon) AANNS will determine that no further contraction is likely to produce results so attention is turned to expanding the net :

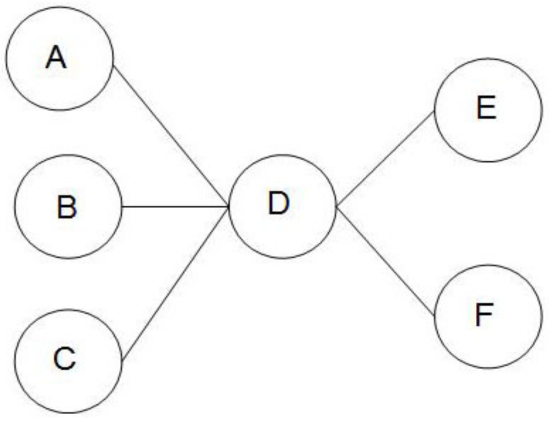


The above diagram shows a subsection of an optimal neural net with a potential candidate additional link XC. The X node might be another intermediate node in the same net (existing input and output links not shown) or any other node in the same net structure (other than A, B or C), or indeed any compatible derived input node (an output node from another already trained net). We will ignore the possibility of creating loops and here confine our explanation to a simple connection.

The suitability of the XC link is checked out in a similar fashion as used for evaluating removal of links. Again use is made of the same selection set, expressed and back-propagated. Two net scores are accumulated, one assuming a very small positive link weight for XC, and one assuming a very small negative link weight. Each result is calculated according to whether the input quantity from the XC link is a help or a hindrance to the back-propagated target input for the C node. The best net score is taken. The more helpful, the more likely the link will be chosen.

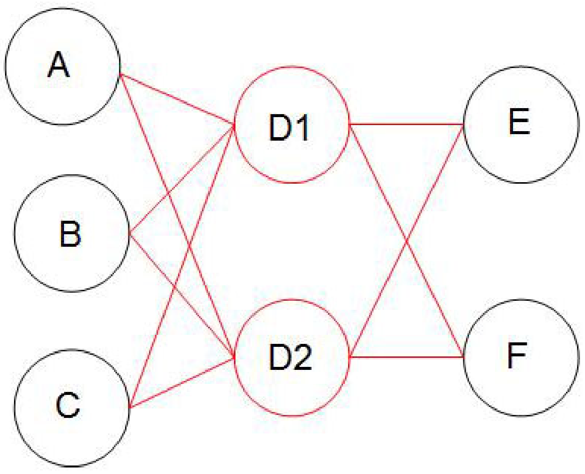
If selected, the XC link is introduced with a zero weight, thus logically giving the same results for the new net structure as before, but when further trained, the link weight will be evolved to potentially improve the performance rating.

Adding a single link is just one way of expanding a net structure. Another way is to clone nodes that are "in conflict" :



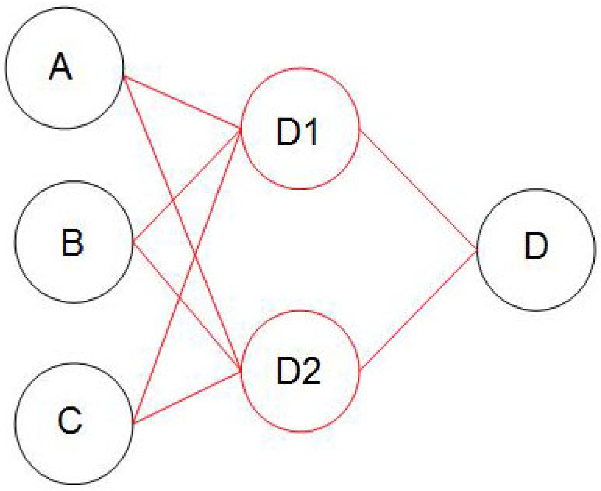
The expressed input quantities (from the selection set) to the intermediate node D (above) may sometimes match the back-propagated target (when active), sometimes be too large, and sometimes be too small. Generally, as a consequence of the previous training, the number of times the input quantity is too large is usually roughly matched by the number of times the input quantity is too small. The method to determine the desirability of cloning node D is to accumulate a benefit/dis-benefit count, for positive or negative input quantity amendment across the expressed and back-propagated selection set. In other words, a trial positive quantity is added to the input quantity to the D node and an accumulation count is incremented if of benefit as regards meeting the target quantity, and decremented if of dis-benefit. This is repeated for a trial negative quantity to form a second accumulation count. If either of these counts is greater than zero, the D node is judged to be "in conflict" and able to benefit from being split up.

If the D node is detected as "in conflict", and also chosen above other candidates the previous net structure is modified thus:



The D node is split into D1 and D2 nodes. The superior and subordinate links are duplicated. The weights of the subordinate links, D1-E, D1-F, D2-E, D2-F are halved. This logically gives the same results for the new structure as before but allows the weights and future structural changes for the superior and subordinate links for D1 and D2 nodes to evolve separately. Note that the new structure has five more links than the old and that this is factored into the selection calculation (the less added links the better).

If the node D (above) were to be an output node, with no subordinate nodes, a different expansion is required :



The original D node is retained but two intermediate node clones are added preceding it and connected to it by links D1-D and D2-D each with a weight of exactly one half. Again, this logically gives the same results as for before, but allows further evolution under training. Notice that this is one way in which extra intermediate levels are evolved.

There are a couple more expansion methods but for the sake of brevity, we shall forego further description. Suffice it to say that from all the expansion candidates the most promising is chosen. Eventually the AANNS runs out of candidates and the next level of control, concerned with expanding the active set, takes over.

AANNS vs CONVENTIONAL PROGRAMMING

Put simply, AANNS automates the coding part of programming and greatly improves the testing of applications. However, there is a lot more to say :

Conventional programming can be split into :

- a requirements specification (what the end user wants).
- a design specification (a high level document to spell out how the programmer should go about his task in satisfying the requirements).
- the coded program (as produced by the programmer to fit the design specification).
- test data (to verify that the program does what it is supposed to do).

However, for complex systems, most of the testing is carried out by the end user (the Beta release and beyond). In my opinion this encourages the developers to skimp. What would be nice is a development regime which guarantees exhaustively tested applications before unleashing them on the public.

AANNS satisfies this aspiration because the training data has to be fulsome for the system to create the required standard of result. If the test data is insufficient then, almost by definition, the product cannot be produced. Although the training may take longer and thus cost more than conventional testing carried out by the developer, the developer saves far more by eliminating program coding costs, and by producing a more reliable product, more directly suited to end users (no compromises necessary to reduce coding complexity), and thus should benefit from more sales.

Conventional programming has an even bigger problem when it comes to adding enhancements to an existing application. The enhancements themselves are not the problem. Making enhancements compatible with what has gone before is! A new programmer has the unenviable task of trying to understand how things work before daring to add to it. All sorts of academic computer language "solutions" have been developed to help (C++ springs to mind) but, if anything, the ability to superficially add yet more complex functions without immediate ructions, whilst still missing deeper semantic inconsistencies, has made the situation even worse.

In contrast, AANNS retains the original test data and together with new data examples for the new functions, reliably "recodes" the whole application. No human is required to understand how the system works. The proof that the application does work is part of what the AANNS does!

Unfortunately, most professional designers wish to understand the neural net logic generated so that they personally can "verify" that logic. Although widespread, in my opinion, this is an unreasonable cultural expectation. Nowadays systems are often too big and too complex for one person, or even a team, to thoroughly understand. But the desire to keep control is a strong one.

(As an aside, despite the difficulty in understanding neural net logic, neural nets used on applications, beyond the capabilities of conventional programming, are often deconstructed with the aim of discovering the logic needed to tackle the problem with the aim of reverting back to conventional techniques. Given the accuracy of current neural net systems perhaps this is justified but this may not be the case in the near future.)

Conventional programs give very accurate answer most of the time, but when they do go wrong the errors can be arbitrarily big - thus the million pound gas bill! Current neural net systems tend not to go wildly wrong but do tend to give approximate answers (all of the time). In some applications this is OK but, for accounting applications and the like, this is unacceptable. However trials of precursors to AANNS show that neural nets *can* be trained to give answers to any desired accuracy.

Unfortunately, there still remains the trust issue. This is a major block to marketing the system and will take a very long time to overcome.

Instead of trying to sell the AANNS lock stock and barrel, the AANNS trainer can be retained in-house. Just the end product neural nets (embedded with the expression algorithm) can be sold on their own functional merits. This has the benefit that the difficult bit, the training algorithm, is safe from plagiarism. But of course, the work to develop new applications is non-trivial, even though it should be quicker than conventional techniques. Commercially, by far the best applications for consideration are those that conventional programs and current neural net systems do badly or not at all.

The practicality of developing novel applications relies on two things. First, the ease of collecting good training data. Second, the ability to teach the AANNS by breaking up complex problems into constituent steps. This capability is mostly satisfied by the combination of the qualities held by good teachers who really understand their subject matter at a fundamental level, and the detective work utilised by good systems analysts to get to the root of how things actually work.

AANNS APPLICATION - HORSE RACING PUNTER BOT

I first became interested in horse racing puntership, when I discovered a report that there are over 2,000 professional punters in the UK making a very healthy living. On investigation, explanations for their success based on race fixing and insider knowledge proved to be unfounded. (I can justify this assertion statistically and at length but for now trust me!) You *do* need to know which horses are being entered for a race just for the outing, but once you understand the basics, this information can easily be determined by anyone. No, what seemed to be the common factor was the striking combination of the professional punter's ability to bet intuitively, whilst being mathematically aware of the which bets were better value and, last but not least, the ability to exercise a lot of discipline as to when and how often to bet. If you could only simulate these characteristics on a computer?

In contrast to the professionals, amateur punters always seem to have reasons for their bets, are often unaware of mathematical value, and change their betting behaviour depending on whether they won or lost their last bet. Having reasons for your bet might seem to be a good thing - until you understand how we humans tick. My speculation is that we can train a part of our organic neural net to come up with high quality answers, but we have no conscious awareness of the neural net logic that achieves this. Because we are social animals, we rationalise in order to justify our actions to others and by reflection ourselves. Unfortunately these rationalisations tend to strangle the more useful intuitions. Professional punters tend to be further along the autism spectrum and mostly do not feel the need to justify themselves. Anyhow, for whatever reason, they do not second guess their intuitive selections.

With the advent of Betfair on the internet, you can usually bet directly against the mass of amateur punters. (There are reports of rigging but only in isolated cases). Betfair charges up to 5% on your winnings so this indicates how much better you need to be than your opponents in order to break even. However, the desired margin is much higher when you factor in overheads, payment for your time, and insurance against having a bad run in the short term. Some years ago I used a crude precursor to AANNS to pretend bet on 1000 races. The results gave a profit of 10%, i.e. on an initial pool of £1000 this would turn into £1100 given bet sizes fixed to one hundredth of the current pool size. However this was using high street bookmaker's odds with a margin to overcome of between 20 and 35 %. AANNS should do a lot better and fully compensate for the development and running costs within a short time period.

The actual initial training is not envisaged to be onerous. But most of the work entailed will be in collecting the training data and updating this on a regular basis. There are prospects of gathering the data from online databases by writing a conventional programming utility. You can find most of the data on racing cards. Adding information on the racing venue e.g. left-handed, uphill at finish etc. should be little trouble. Organising a database to keep the history of each horse (and perhaps jockey and stable) in a suitable form is perhaps the bulk of the programming work.

In summary, the advantage of this scheme is that no marketing is involved. The main drawback is the amount of data that has to be captured and regularly updated.

AANNS APPLICATION - POKER BOT

A main attraction of this potential application is that it too requires no marketing. Furthermore, it requires relatively little training data without the need for constant updating.

The existing poker bots on the internet make a buck or two from detecting and picking on novice players, and by using comprehensive statistical calculations to evaluate each hand. However, strong poker players can work out during the course of play that the bot (of course pretending to be a human player) is very predictable and then have the ability to take advantage. (If the bot tries to behave more randomly, then it can loose its advantage of knowing the statistics associated with each hand.)

I am not a very good poker player. My main fault is that I find it hard to fold hands. I like to call so that I can satisfy my curiosity as to what my opponents hold. However, I do not have to be a good player to teach the AANNS. I (or any other AANNS analyst) just need to take note of those experts who understand and can convey the key concepts of the game, in order to define the AANNS schema, and then train using appropriate data collected mainly from professional games.

OK, that is perhaps a little simplistic! Actually one might start with training to produce the mathematical odds of winning the hand taking into account the number of players in, or potentially in the pot, versus the pot size. This would put the resulting net on a par with existing poker bots. From this base more sophisticated perceptions can be added.

The first of these is to detect, at each hand, to what degree players are likely to be "tight" or "wild". Some of the factors involved are position in betting order, stack size, previous hand success or failure, and response time (the only "tell" available over the internet). This categorisation is a step on the way to trying to guess what range of cards the opponent may be holding. This is best trained from data collected from televised events where the viewer is privy to more of the cards players are holding than in conventional games and can access expert commentators views on tightness of play.

The next step is to collect what the commentators judge the range of hands that it is reasonable for a player to be holding ignoring any viewer privileged information. This can be padded out with guesses of what other players were guessing given their betting patterns but excluding the situations where players are judged to fold solely to protect their stack.

Finally the net can be trained to fold, call, check, raise, or raise all in, in accordance with what happened in professional games, perhaps modified by expert commentary, given that professional do occasionally make obvious mistakes. Please understand that what you should do as a player is quite different from what you should do if you could see your opponent cards face up. Apart from developing statistical perceptions, training a net on the latter case (cards face up) is a mistake. We are trying to mimic professional play which includes amongst other things the ability to bluff!

The success of the training depends mainly on how good the expert exposition of the game is. Identifying the key concepts is key to being able to simulate the sophisticated play that professional players exhibit. Thus the development of this application involves some risk, but if it were to be successful, it would be tantamount to a license to print money with very little ongoing effort expended.

AANNS APPLICATION - GO BOT

If you are unfamiliar with the board game, go, you might want to just skim this section.

A number of very influential people would take a great deal of notice of a genuinely strong go playing algorithm. Because chess has succumbed to the artificial method of brute force lookahead using the massive calculating power of the modern computer, go which is not as vulnerable to this technique, is now held up as the showcase challenge to prove an advance in artificial intelligence.

Recent developments in go programs using the Monte Carlo Tree Search method have been successful in as much as the best is currently holding a 4 dan status for 15 seconds per move on KGS, an online go server. The method has major weaknesses in that very good moves can be followed by unrelated poorer quality moves. This is due to the fact that the search cannot really perceive a "plan" and is relying on randomly finding a good lookahead path, which given the number of possible variations, is not always possible. Nevertheless, a 4 dan level, albeit blitz, is surprisingly good. But with a method relying mainly on brute force lookahead, the question is whether progress will stall at a lookahead horizon. Unfortunately the current success, such as it is, will muddy the argument that go is a benchmark for testing truly intelligent algorithms.

Go is a difficult problem, far too difficult to solve with a single trained net. AANNS gives the prospect of teaching a hierarchy of neural nets a raft of intermediate basic perceptions, the equal of those possessed by strong human go players. To start with there are a lot of very basic perceptions (all on an intersection level) to train for, such as liberties (for empty intersection), proximity to black, proximity to white, single intersection local eye, single intersection local false eye etcetera etcetera. An equally large number of more tricky perceptions awaits :- liberties in general, connectability, half, one, one and half, two eye shapes, damezumari, group identity labelling, territory quality and size, miai, sente, ko evaluation, ko threats, kikashi, etcetera etcetera. New concepts such as sector lines and ideas from Global Connectivity Strategy, can be added to help with fuseki. Training to intelligently encode (rather than rote ingest) joseki might also help.

Potentially, there is one small fly in the ointment - how to mimic human lookahead. First note that humans are generally terrible at serial lookahead (trying out each hierarchical branch and sub-branch) which is why there is a market for tsume go problems which any decent computer algorithm can solve in a millisecond by serial lookahead of the relatively very few possible permutations (- but in real games, most of the time, it is a totally different story!). Neural nets, whether human or artificial, are just not efficient at simulating the serial or "brute force" type of lookahead. This begs the question - so what do humans do with lookahead?

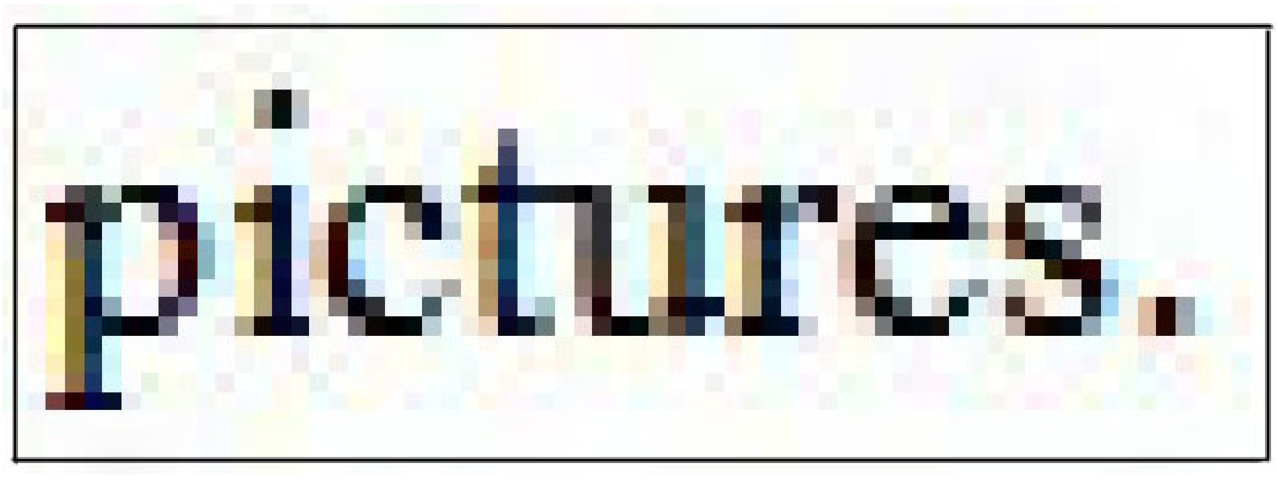
From what I can make out, in a semeai (a race to kill or be killed) the typical amateur 5 dan (relatively a strong player not quite up to professional standard), first intuitively identifies potential key moves in the sequence to win the race. Then the player tries to string these key moves together to make a playable sequence. This is not necessarily done in a specific order but the result is a serial set of moves, or "line". If the end result is a happy one, the player is likely to recheck the key moves for his opponent, taking into account that better moves may be found. Assuming feasible alternatives have been found, the process of stringing these moves together to form a single playable line is repeated. If at any point, the end result is not a happy one, the player will revisit his own key moves, and taking into account the end result, will try to intuit alternative key moves so as to string together a new alternative line.

Thus the human type of lookahead is very limited but very efficient compared with a computer algorithm. The reading strength of the human player lies mainly in their skill of identifying key moves (the stronger the player, the fewer and more discerning key candidate moves are considered) and to a lesser extent, how many lines the player can process in the time available. With the ability to learn candidate key moves, if taught properly, and the ability to apply iterative logic for a multi layered approach to find modified candidate key moves, the AANNS, at least in theory, should be capable of mimicking human lookahead. It is even possible to teach brute force lookahead, but with the downside of potentially very slow execution times (thus running out of time) - the same problem humans have. The big question is whether brute force lookahead needs to be used at all. From observation of the standards achievable playing lightning go, and the errors made by strong players in semeai reading, my judgement is brute force lookahead is probably not needed. However, if that judgement were demonstrated to be wrong, the facility could be artificially added (hard-wired) outside of the AANNS net. Here is potentially one case where conventional programming is better than a neural net. Even with hard-wired brute force lookahead, human lookahead should still be implemented because of its use as a perception on which to base complex group status analysis and other sophisticated perceptions.

Overall there appears to be rather a lot of work to define and teach the necessary number of hierarchically built perceptions, especially since many key perceptions are poorly understood or even totally ignored by even the strongest players (none of whom have an inkling of how they manage to intuit good moves). It requires a lot of work and a lot of aptitude to train a gifted human player up to professional standard, so perhaps we should not underestimate the similar effort needed to train a net using AANNS. It is also a drawback having to explicitly define a large number of intermediate perceptions, where in human training, it appears that the student can, at a subconscious level, find and utilize key perceptions without formal identification, albeit at a very slow rate. (If a teacher can identify these key perceptions either explicitly or by well chosen examples, this is of enormous advantage.) This aspect of the human neural net probably equates to detecting the equivalent of output nodes in the middle of a subnet by connecting preferentially to neurons that exhibit a concentration of data flow. This approach deserves further research, but is not thought vital for achieving the immediate goal of an amateur dan level go playing program.

AANNS APPLICATION - OPTICAL CHARACTER RECOGNITION

OCR utilities take images of text such as held on a digital photo file, e.g. JPEG file, and try to convert them into code, a mere couple of bytes encoding each character, suitable for a document file. For example:



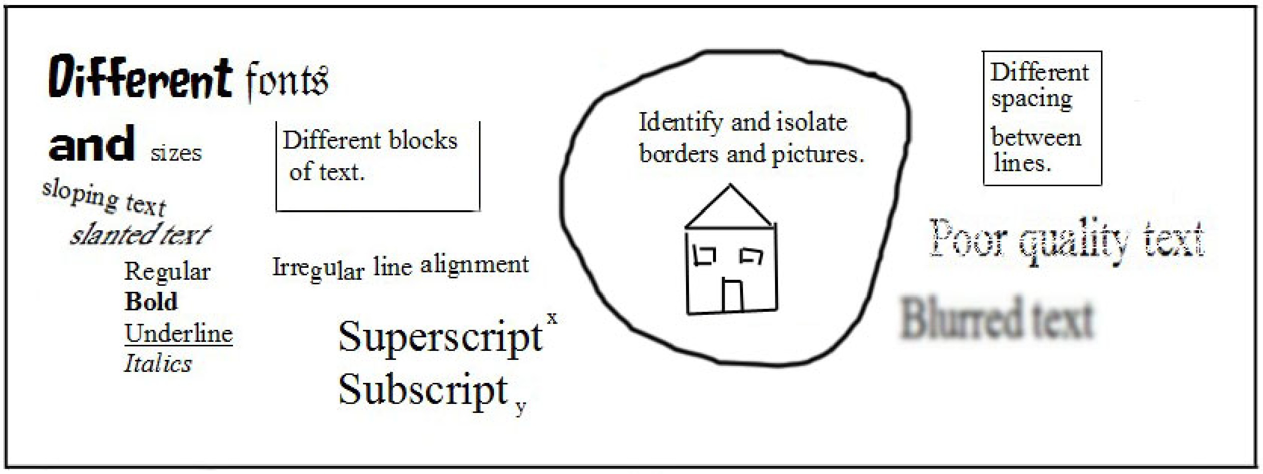
Here is a blow up of an image of a piece of black text. You can clearly see that it is made up of tiny squares (pixels) of differing colours (or if confined to monochrome, shades of grey), but when viewed at the correct scale, it gives the appearance to the human perception of the text "pictures." as viewed on this page. The input to the OCR utility is just a series of numbers defining the colour for each individual pixel.

At the heart of any OCR utility is some form of a neural net. This gives a much better performance than conventional programming, but still can have a significant failure rate. Suppose you want to capture this very document from paper onto your computer so that you can edit it (or whatever). So you decide to use your scanner with its OCR software. On a page of plain text you should expect a failure rate of say half of one percent in character recognition. Although much quicker than typing the whole thing in, the produced text page still needs to be manually checked and the 20 or so errors, out of the 4,000 or so characters typically found on a A4 sheet, need to be corrected. But worse, on pages with diagrams, or tables, the OCR utility will tend to struggle as typically they are easily confused! So why doesn't the neural net software do better? There are two main reasons :

The first reason is that the existing neural net technology is not up to the whole of the task, so a lot of conventional programming is used to prepare the data for the net, and this inevitably simplifies and therefore loses information that might otherwise be of theoretical use in determining the boundaries of each character on the page.(Note that OCR works much better with single characters with generous surrounding space.)

The second reason is that the neural nets are one hit trained, using artificial methods not particularly designed to simulate human perception. So it is not too surprising that the OCR utility does not reproduce function that can see what a human can readily perceive. (In theory, they also might be able to decipher text undecipherable to humans - but this is not that useful!)

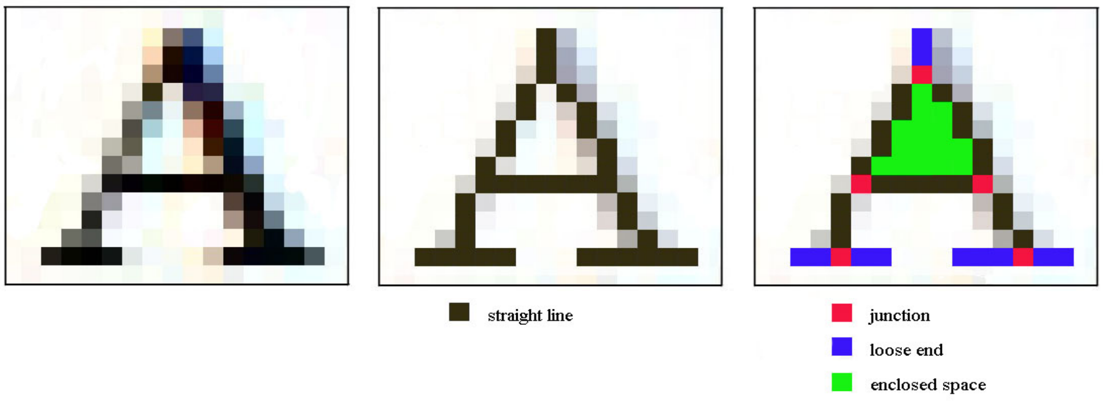
The following diagram shows text which is easy for humans to decipher but difficult for existing OCR utilities :



The big idea for utilising AANNS is to simulate the human way of perceiving text, or at least something a lot closer to it than current methods manage. It is expected that, if achieved, this would produce a performance level of nigh on 100% for straightforward text, quite apart from being able to convincingly tackle a far wider range of difficult text image input. Apart from dealing with the example shown above, such a capability would enable imaged text to be captured from web browsers, which would form an important part of capturing test data for further applications. So what are the main steps needed to form an OCR acolyte system?

The first step is to give the acolyte the ability to perceive very basic graphic features of a printed page (with or without picture/diagram content) in the way cognitive studies have demonstrated humans operate. The first feature of human perception is the overwhelming propensity, at any excuse, to see lines (e.g. canals on Mars!). Give us a few dots and we will automatically join them up!

The three diagrams below shows the first two stages that the AANNS might go through in a much longer multi-stage process to train a final net to reliably read imaged text to the same standard as a human can achieve.



The first diagram shows the pixels of a small part of a scanned image page. When printed or displayed at the correct scale this portion will appear as an apparently pristine capital A. The first task is to train a net to perceive both straight and curved lines despite blurring and/or the minor breaks common in poor quality copies. With our capital A in this font, only straight lines should be discovered. The net is trained to represent lines by the minimum number of contiguous (orthogonal or diagonal) array elements. This is shown in the second diagram (superimposed on the original data).

The third diagram shows the next stage in training to determine further features each of which it has been demonstrated, by cognitive studies, that human perception relies on. Notice that the features are independent of scale, and are already probably sufficient to define a capital A. However, at this stage it is sufficient to detect isolated character-like entities, whilst noting potential character candidates.

The next stage is to train a net to perceive the boundaries defining word-like entities and character clusters. If you look at the very first diagram showing the "pictures." pixels, you might notice that the "t" and the "u" are physically joined so these will need to be demarcated for further processing.

The above description is only a brief introduction to the way AANNS can solve this complicated problem. After much development, the end result should be a highly saleable OCR utility. Because the method is based on human perception, a little more development should produce a superior handwriting recognition utility. Because there is no reliance on artificial brute force methods, just the final trained neural net, the speed of operation should be excellent. Nor should there be any need for the end user to train the net for their own writing style.

In summary, given a thorough approach to training the net in stages, there is little risk that the aims described above cannot be achieved. It is, however, a lot of work. Marketing is also heavily involved. But the financial rewards would be considerable.

GLOSSARY

AANNS	Acolyte Artificial Neural Net System
acolyte system	The system where multiple linked neural nets are held forming a multi-layer perception capability to provide a final neural net with the necessary data to meet the requirements of a complex application.
active component	A **component** that is a candidate for training adjustment as found by **back-propagation**.
active set	That part of the **load set** that is currently or recently been used in training.
attributes	Items of the AANNS net structure modified in each **training ascent**, i.e. **weights** and **exponents**. a.k.a. components
back-propagation	The process of identifying the training adjustment needed for each **component** by reverse logic using targets progressively projected from the output nodes back along the **links** and **intermediate node**s in the direction towards the **input node**s.
best net	For any specific application, the best performing trained net to date.
components	Items of the AANNS net structure modified in each **training ascent**, i.e. **weights** and **exponents**. a.k.a. attributes.
current active set	That part of the **active set** that is currently being used for training.
derived input nodes	The automatically detected and assigned output nodes from expressed neural nets held by the **acolyte** acting as input to the subject neural net.
exponent component	A node attribute variable which performs an exponent operation on the node input data
expression	Each input record when submitted to a **neural net interpreter** is expressed via the specific neural net logic to form an output record.
identity type	A type of **primary node** input mapped onto an i**ndicator binary node** for indicating the presence of the identity, such as a place name, in an input record.
inactive component	A **component** that is logically determined as being an ineligible candidate for training adjustment as found by **back-propagation**.
indicator binary node	A **secondary input node** that accepts only one or zero as data.
input node	A neural net structure item for receiving input values.
intelligence	The degree of scope for appropriate behaviour of an agent for any given set of knowledge and any given amount of processing used by that agent.
intermediate node	A neural net structure item connected by **link**s **input**, **intermediate** or **output node**s
internal set	The set of training records (all other possible records forming the external set).
link	A neural net structure item for transmitting processed data between nodes.
load set	After allocation of the **reference set**, the set of records available for active training.
neural net interpreter	A type of computer program that takes input data and processes it via any specified neural net structure to produce the desired application output.
neural net trainer	A type of computer program that takes an application's training data and specified **schema** and then produces the whole neural net structure necessary to produce the general solution for the application.
optimal working net	An intermediate trained net that has optimal performance for any given net structure considering the number of links in that structure
output node	A neural net structure item for transmitting output values.
pending set	That part of the **load set** yet to be used in training.
primary node	A class of a node defining some logical purpose, actually implemented via **secondary node**s.

recent active set	Records that have previously been used in training.
reference set	A proportion of the training records (the **internal set**) reserved for the gold standard evaluation of the performance of the **load** set.
representative data	Training data for an application is said to be representative when it theoretically contains enough examples of wide enough scope to embody the underlying principles that form the solutions to novel data.
schema	The specified input and output structure of any particular neural net.
secondary node	A fundamental class of node automatically allocated to implement a declared primary node.
selection set	A set of records selected in the course of determining which links and nodes to prune.
solution attribute topography	The multi-dimensional graph of **component** values plotted against performance.
square grid array	A type of of **primary node** mapped onto **secondary node**s for implementing a square array structure.
subordinate link	A **link** that receives data from a node is subordinate to that node.
superior link	A **link** that transmits data to a node is superior to that node.
threshold	A gate value associated with any one specific **node** to filter its input data.
training ascent	Using the same logical net structure, the series of **component** values, that were progressively modified to form steps on a path of improving performance, the lowest point being a base and the highest a peak.
training expedition	The series of **training ascents** for the same logical net structure.
training step units	The values of the different **component**s expressed in one currency, dynamically modified in order to help the process of determining the candidate training steps in the **solution attribute topography**.
weight	A multiplier value associated with any one specific **link** applied to that link's input data to produce that link's output data.