In Proceedings of the 2013 Genetic and Evolutionary Computation Conference (GECCO-2013) 2013. ACM Press.

Neuroannealing

Martingale-driven Optimization for Neural Networks

Alan J. Lockett
IDSIA
Galleria 2
6928 Manno-Lugano, Switzerland
alan.lockett@gmail.com

Risto Miikkulainen
University of Texas at Austin
Austin, TX, 78712
risto@cs.utexas.edu

ABSTRACT

Neural networks are effective tools to solve prediction, modeling, and control tasks. However, methods to train neural networks have been less successful on control problems that require the network to model intricately structured regions in state space. This paper presents neuroannealing, a method for training neural network controllers on such problems. Neuroannealing is based on evolutionary annealing, a global optimization method that leverages all available information to search for the global optimum. Because neuroannealing retains all intermediate solutions, it is able to represent the fitness landscape more accurately than traditional generational methods and so finds solutions that require greater network complexity. This hypothesis is tested on two problems with fractured state spaces. Such problems are difficult for other methods such as NEAT because they require relatively deep network topology in order to extract the relevant features of the network inputs. Neuroannealing outperforms NEAT on these problems, supporting the hypothesis. Overall, neuroannealing is a promising approach for training neural networks to solve complex practical problems.

Categories and Subject Descriptors

I.2.m [**Artificial Intelligence**]: Miscellaneous

General Terms

Algorithms

Keywords

evolutionary annealing, martingale optimization, neural networks, evolutionary computation, genetic algorithms, neuroevolution, neuroannealing, applied measure theory

GECCO'13, July 6–10, 2013, Amsterdam, The Netherlands.

1. INTRODUCTION AND MOTIVATION

Neural networks are effective tools to solve prediction, modeling, and control tasks. However, methods to train them have been less successful on control problems that require the network to model intricately structured regions in state space. Such problems have a fractured state space, for which the decision boundaries cannot be well represented by shallow networks due to the mathematical properties of the neurons and their activation. Kohl [3] exhibited several problems with fractured state spaces, such as recognizing concentric spirals and implementing a multiplexer for address-based lookup. Importantly, many practical domains are also fractured and require complex decision boundaries; Kohl examined robotic keepaway soccer as one example.

Kohl studied fractured problems in the context of NEAT [3], an evolutionary method for neural networks developed by Stanley and Miikkulainen [8]. In principle, NEAT is capable of learning arbitrarily deep and complex networks, but in practice it rarely does so. A neural network defines a map between input states and output states. Kohl demonstrated that NEAT's performance tends to degrade with the complexity of the problem, as determined by its total variation, that is, the maximum cumulative change in fitness value along any non-self-intersecting path through the input space.

Kohl showed that the performance of NEAT degrades as the total variation of the problem increases, a property that he termed *fracture* [3]. As Kohl observed, when NEAT does succeed in fractured domains, the successful networks tend to be larger, allowing them to encode higher complexity. Kohl proposed several methods for solving fractured problems, but these solutions required using nonstandard activation functions or iteratively freezing network weights. A more principled approach would be to construct a learning method for neural networks that makes it possible to construct deeper and larger networks systematically.

This paper presents neuroannealing, a method for training neural network controllers on problems that require deep network topologies to represent the relevant structure of the search space. Neuroannealing uses evolutionary computation in order to search the space of neural networks, a process referred to as *neuroevolution* [1, 8, 2]. By using a martingale-driven optimization method, neuroannealing is able to search for and discover larger and deeper networks than has been previously possible using other methods.

Neuroannealing is based on evolutionary annealing [4, 5], a global optimization method that employs a martingale approximation in order to leverage information about the fit-

ness function obtained from successive evaluations. In Euclidean space, its performance is comparable to or better than other general-purpose optimizers on certain problems with complex fitness structure [5]. Like other evolutionary annealing approaches, neuroannealing does not forget any intermediate solutions and so is able to represent fitness landscapes more accurately than traditional methods. Neuroannealing finds solutions that require greater network complexity than is usually obtained with methods such as NEAT and as a consequence is more capable of solving fractured problems.

In this paper, this hypothesis is tested on two of the problems with fractured state spaces proposed by Kohl, the concentric spirals problem and the multiplexer problem. Also, neuroannealing is tested on the classic double pole-balancing problem. Overall, neuroannealing is a promising approach for training neural network controllers to solve complex problems.

In the next section, the evolutionary annealing framework is introduced, after which neuroannealing is presented as a specific instantiation of this method.

2. EVOLUTIONARY ANNEALING

Evolutionary annealing is an evolutionary algorithm with a selection operator that can select any member of any previous population by sampling from a Boltzmann distribution like that used in simulated annealing. It can be used with any mutation operators, and it may or may not use crossover. Evolutionary annealing differs from other evolutionary methods in that its selection mechanism is designed to guarantee full coverage of the search domain. The performance of evolutionary annealing for Euclidean space was explored by Lockett and Miikkulainen [4, 5] for a particular instantiation of the algorithm in Euclidean space with no crossover and Gaussian mutation. In this section, the unique selection mechanism for evolutionary annealing is described in its general form so that the approach can be applied to train neural networks in later sections.

2.1 Evolutionary Annealing Motivation

An evolutionary algorithm learns about the structure of a fitness function by evaluating potential solutions in a sequence of populations. The original genetic algorithms were population-Markov, meaning that each population was discarded after being evaluated and reproduced to form the next population. A population-Markov algorithm explores the search space at random, forgetting where it has been.

Early on, this feature was recognized as a deficiency for the static fitness functions, i.e., fitness function that do not change as evaluations progress. Many modern evolutionary methods use some form of summary or average to retain information about previously evaluated solutions; such techniques include elitism and $(\mu+\lambda)$-selection. These algorithms have proven that remembering information derived from fitness evaluations can help locate the optima.

Evolutionary annealing takes this concept to its full extent. Rather than generating a summary, it stores the results of all evaluations in order to build a model of the fitness function that it can use to search the space more efficiently. Because computer memory is cheap, it is practical to do so for a wide range of optimization problems. The experiments in this paper run entirely in memory and do not exceed one gigabyte of RAM. The challenge for this approach is to find a way to use this information effectively. The next subsection discusses how evolutionary annealing addresses this challenge.

2.2 Annealed Selection

Evolutionary annealing organizes its search by partitioning the search domain into a set of disjoint regions, one for each previously evaluated solution. The partition is represented by a binary tree, which can be traversed in logarithmic time in the average case. Higher nodes in the tree represent the union of the regions that they span, and the root represents the entire search domain.

The search domain is assumed to be a Hausdorff topological space (X, τ), where X is the set of acceptable solutions, and τ is a *topology*, that is, a list of the open sets for the domain that are used to determine which points in the search domain are close and which sequences converge. Additionally, the space is assumed to be a measure space with a measure λ that is Borel-measurable for the given topology and finite on the search domain. The measure λ assigns a volume to each region and may be thought of as determining the size of a partition region. As a simple example, any closed subset of Euclidean space with the Euclidean distance and the standard notion of volume (i.e., $\lambda(B) = \int_B dx$) meets these requirements, as does any finite space with the counting measure ($\lambda(B) = |B|$).

As each population is evaluated, the members of the population are inserted into the partition tree using Algorithm 1, described in Section 3.3. In general, an evolutionary annealing algorithm requires a method for separating an existing partition region into two smaller regions, one for each of two evaluated solutions that fall within the existing region. This paper only describes the mechanism for neural networks; see [5] for details in other spaces.

Evolutionary annealing uses an annealed selection mechanism that selects a previous member of the population by walking a partition tree from the root to a leaf, performing a Bernoulli trial at each node in order to decide which child will be selected. The sample probability for these Bernoulli trials is chosen to achieve a particular probability distribution over the leaf nodes, that is, over all previously evaluation solutions, as described next. The partition-backed annealed selection mechanism is the distinguishing feature of evolutionary annealing; the motivation for this approach as well as its relationship to simulated annealing and genetic algorithms is discussed in detail in [5]. In the terminology of stochastic processes, the sequence of partitions forms a filtration, and at fixed temperature, evolutionary annealing is an approximate martingale. Thus evolutionary annealing is described as a *martingale-driven optimization method*.

Lockett and Miikkulainen [5] discussed two different methods for choosing the probability distribution over the previously evaluated solutions, one based on proportional selection, and one based on tournament selection. Neuroannealing uses the tournament selection variant because it is less sensitive to problem scaling and performs better experimentally. Like simulated annealing, these methods use a cooling schedule in order to achieve asymptotic convergence. A cooling schedule is a sequence of temperatures $(T_n)_{n\in\mathbb{N}}$ such that $T_n \to 0$ asymptotically. The schedule $T_n^{-1} = \eta \log n$ is used for neuroannealing, where η is a learning rate that can be increased or decreased to speed up or slow down search. The value $\eta = .1$ is used for neuroannealing.

Tournament annealing views the selection problem as a sequence of contests between members of the population, decided probabilistically according to their fitness rank. The best member of all previous populations is chosen with probability q. If it is not selected, then the next best member is selected with probability q, and so on. Let $A_n \subseteq X$ be the set of all individuals evaluated after n populations. The probability of selecting a point $x \in A_n$ is given by

$$p(x) = \xi_n^{-1} q^{1/T_n} \left(1 - q^{1/T_n}\right)^{rank(x)} \lambda(E_n^x), \qquad (1)$$

where ξ_n is a normalizing factor, f is the fitness function, and E_n^x is the partition region containing x after the n^{th} population is evaluated and inserted into the partition tree. The function $rank(x)$ is the index of x in A_n when A_n is sorted by fitness, so that the best solution so far has $rank(x) = 0$. The parameter value $q = 0.025$ is generally fixed, since the actual value can be varied by changing the cooling schedule. The region weight $\lambda(E_n^x)$ is used to promote exploration of the search space.

It is possible to derive a tree sampling algorithm for tournament annealing, but a second tree is needed to sort the solutions by fitness. This tree is termed the *score tree*. Each leaf of the tree is associated with exactly one partition region and exactly one previously evaluated solution. The tree is generated by inserting each partition region into the tree one at a time. The region is inserted in such a way that the leaves are ordered from left to right based on the fitness of the previously evaluated solution within the partition region. This tree can be balanced for efficient traversal, and the sampling algorithm can rely on the left child of each non-leaf node as containing higher ranks.

In order to sample the score tree for selection, the node sampling probabilities are needed. Define

$$\tilde{q}\,(h, T) = \frac{1}{\left(1 - q^{1/T}\right)^{(2^h)}} \qquad (2)$$

to be a temperature-adjusted selection probability, where h is the height of the current node and T is the temperature. Suppose that at node ν, the sampler must choose between left child μ and right child κ, so that μ contains points with better fitness. Then given temperature T, the probability of selecting μ should be

$$\mathbb{P}_T\,(\mu \mid \nu) = \frac{\tilde{q}\,(h, T)\,\lambda\,(\mu)}{\tilde{q}\,(h, T)\,\lambda\,(\mu) + (1 - \tilde{q}\,(h, T))\,\lambda\,(\kappa)}, \qquad (3)$$

where $\lambda\,(\mu)$ and $\lambda\,(\kappa)$ are the cumulative weights of the partition regions of the points in the span of μ and κ, respectively. It is shown in [5] that choosing μ or κ according to Equation 3 will result in a probability over the points at the leaf nodes equal to Equation 1. Neuroannealing uses tournament annealing exclusively.

Because the selection mechanisms for evolutionary annealing are complex, a reference implementation is provided for reproducibility at http://pypi.python.org/pypi/PyEC. It includes the implementation of neuroannealing.

In order to apply evolutionary annealing, one must make the algorithm concrete for a particular search domain. The next section describes how evolutionary annealing can be used to search the space of recurrent neural networks (RNNs).

3. NEUROANNEALING

In order to apply evolutionary annealing to the space of neural networks, three components must be defined: (1) a base measure over neural networks, (2) an algorithm for partitioning sets of neural networks, and (3) a sequence of mutation distributions likely to improve the fitness value of a network. This section proposes a particular approach to defining these components that is collectively termed *neuroannealing*. First, the concept of a layer of nodes is introduced as a building block for RNNs, and then each of the three components are described in turn.

3.1 Layered RNNs

Neuroannealing searches the space of RNNs for the optimal networks to solve a control problem. In order to generate different network topologies, neuroannealing stochastically adds and removes new links and nodes to existing networks. In addition, neuroannealing organizes nodes into layers and provides mutation operators to add and remove entire layers of neurons. A layer is a group of nodes such that within a layer, all nodes are of the same type, either inputs, outputs, or hidden nodes. In a layered RNN, links interconnect neural layers, so that two nodes are connected if and only if their respective layers are connected. Links between two layers are associated with a weight matrix containing the connection strengths between each layer's nodes.

The concept of layers is standard when training neural networks using supervised techniques. Layers merely add a conceptual separation that is useful for computational efficiency, since it reduces the number of weights that must be stored and multiplied. In neuroannealing, layers also play a role in allowing the structure of the network to expand in useful ways. Neuroannealing probabilistically inserts layers that are designed to store the prior state of another layer, providing a natural way for RNNs to develop an otherwise improbable memory.

To represent a NEAT RNN as a layered network, each node can be assigned to its own layer. The effect on neuroannealing's optimization ability can be tested by enforcing this property on all proposed networks.

3.2 Base Measure for RNNs

The measure over RNNs used by neuroannealing is built up from simpler measures. The space of layered RNNs can be partitioned according to the following four features: (1) the number of layers ℓ, (2) the number of nodes in each layer s, (3) the connectivity pattern among the links c, and (4) the weight values w. A layered RNN representation can be identified exactly by the tuple (ℓ, s, c, w). The base measure is constructed by addressing each of these items in reverse. Since the value of the base measure appears in the selection probability for the next population of networks (Equation 1), networks that are preferred by the base measure will be explored more thoroughly. In general, one wishes to emphasize smaller, less complex networks without penalizing extra structure too severely.

The first three criteria above comprise the network topology. If ℓ, s, and c are all fixed, then an RNN may be described completely by listing its weights and biases. There are a fixed number of weights and biases, and so an RNN with a given topology may be treated as a vector in $\mathbb{R}^C$ where $C = C(c)$ is the number of weights and biases. Neuroannealing utilizes a Gaussian measure to allow unbounded weights with a preference for small weights. For a given ℓ, s, and c, the measure over RNNs matching this profile is

$$\lambda_{\ell,s,c}(A) = \int_A \exp\left(-\frac{x^2}{2\gamma^2}\right) dx \tag{4}$$

where A is a measurable set in $\mathbb{R}^C$. The factor γ is termed the *space scale*; it reflects the average absolute correlation between connected nodes. The default space scale is $\gamma = 1$.

Next, networks with the same number of layers and layer sizes but different connectivity are handled. The connectivity pattern c can be represented as a binary string of size $L = N^2$ where N is the total number of nodes in the network, $N = \sum_i s_i$. L is the number of possible links. Let $n(c) = \sum_i c_i$ be the number of actual links in c. Given ℓ and s, there are exactly 2^L distinct connectivity patterns. Let P be the set of such patterns. A set A of RNN representations with different connectivity patterns may be partitioned into a finite family of sets $\{A_c\}_{c\in P}$, separating out RNNs by connectivity. A measure over such sets is given by

$$\lambda_{\ell,s}(A) = \sum_{c\in P} \frac{1}{n(c)} \begin{pmatrix} L \\ n(c) \end{pmatrix} \lambda_{\ell,s,c}(A_c). \tag{5}$$

Here the factor $1/n(c)$ is applied to prefer networks with lower connectivity, and hence fewer parameters. The combinatorial factor is added to emphasize networks that have about half of the possible number of links. The combined effect of the two parameters prefers smaller networks that possess a reasonable number of links.

If only the number of layers is fixed, the number of sizes s is a vector of positive integers greater than one with dimension ℓ. Networks with smaller layer sizes are preferable, but layers of size one should not be emphasized too strongly, or else neuroannealing will not consider larger layer sizes. This balance was accomplished by weighting each size profile inversely to the total number of nodes in the network. There are countably many possible layer sizes, and these can be enumerated. Let S be the set of size profiles, and define

$$\lambda_\ell(A) = \sum_{s\in S} \frac{1}{\sum_i s_i} \lambda_{\ell,s}(A_s), \tag{6}$$

where A_s, like A_c in the last paragraph, decomposes A according to size profiles. It is notable that λ_ℓ is not finite, unlike $\lambda_{\ell,s}$ and $\lambda_{\ell,s,w}$. First, there are many size profiles with equivalent sums, and second $\sum 1/k = \infty$ even if there were not. The theory of evolutionary annealing only applies to finite measures. A finite measure over size profiles can be obtained by capping the total size of the network with some large value. In practice, the experiments in this paper never produced a network larger than 256 nodes, and so this value was used as a maximum network size.

The base measure over RNNs is achieved by handling arbitrary numbers of layers. This number is an integer greater than one. As with sizes, a set of RNNs may be decomposed according to the number of layers, so that for a given set of RNNs A, the set A_ℓ is the subset of A with ℓ layers. Then a measure over arbitrary layered RNNs is given by

$$\lambda(A) = \sum_{\ell=2}^{\infty} \frac{1}{\ell} \lambda_\ell(A_\ell). \tag{7}$$

Once again, this measure is not finite, but a finite measure can be obtained by bounding the size of the network at some large value. In the experiments that follow, the number of layers was bounded above by 256; more than 20 layers were rarely observed.

The next section describes how this measure can be used to partition the space.

Algorithm 1 Algorithm to Generate a Partition Of RNNs

$\{x_m\}_{m=1}^M \subseteq X$, the observed networks as (ℓ, s, c, w) tuples

$\mathcal{T} \leftarrow \{X\}$, the partition tree
$k(i) \leftarrow \emptyset$ for all $i = 1, \ldots, M$, node assignment function
$\mu(\{X\}) \leftarrow (0, 0, 0, 0)$, the node marking function
$idx(\{X\}) = 4$, the node separation index function
for $m \leftarrow 1$ to M **do**
  $N \leftarrow$ highest node in $\mathcal{T}$ s.t. $x_m \in N$ and $\exists i \leq idx(N)\, s.t.\, \mu(N)_i \neq x_{m,i}$
  if $\exists j \neq m$ s.t. $k(j) = N$ **then**
    $N_0, N_1 \leftarrow separate\,(x_j, x_m, N)$
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{N_0, N_1\}$
    $k(j) \leftarrow N_0$, $k(m) \leftarrow N_1$
    $\mu(N_0) \leftarrow \mu(N)$, $\mu(N_1) \leftarrow x_m$
    $idx(N_0) \leftarrow idx(N)$, $idx(N_1) \leftarrow 4$
    $idx(N) \leftarrow$ the minimum i s.t. $x_{m,i} \neq \mu(N_0)_i$
  else
    $k(m) \leftarrow N$
    $\mu(N) = x_m$
    $idx(N) \leftarrow 4$
  end if
end for

3.3 Partitioning Networks

Evolutionary annealing works by partitioning the search space at increasingly fine resolution one point at a time. There are many ways in which such partitioning could be done. Neuroannealing uses the four levels in described in the last section in order to partition sets of neural networks. For this purpose, the partition tree is conceptually stratified into four sections, one for each of the four levels used to define the base measure in Section 3.2.

The stratification can be best understood by starting with the node-separation algorithm used to divide an existing partition region between two networks. Given two networks x_1 and x_2 and a set A, neuroannealing must create disjoint sets A_1 and A_2 such that $x_1 \in A_1$ and $x_2 \in A_2$. The networks can be decomposed so that $x_i = (\ell_i, s_i, c_i, w_i)$ for $i = 1, 2$. If $\ell_1 \neq \ell_2$, then compute the midpoint $\tilde{\ell} = \lceil \frac{\ell_1+\ell_2}{2} \rceil$, and let A_1 be the set of networks in A with less than $\tilde{\ell}$ layers, and let $A_2 = A \setminus A_1$. If $\ell_1 = \ell_2$ but $s_1 \neq s_2$, then a vector separation method like that in [4, 5] can be applied to the size vectors s_1 and s_2. The same approach can also be applied if $c_1 \neq c_2$, and finally if $w_1 \neq w_2$. This approach to separation assumes a hierarchy of separation levels, so that ℓ is separated first, then s, then c, and finally w.

Provided that any traversal through the partition tree from the root respects the ordering of this hierarchy, the tree will correspond to a valid partition. If the ordering is violated, for example, by separating on w at a higher node in the tree, by ℓ at a lower level, and then by w at the the leaf, then the regions contained in distinct branches of the tree may overlap, with deleterious results. Thus a traversal through the tree must be stratified. Any separation on ℓ must occur first, then separation on s, and so on.

Algorithm 1 implements this stratification. The network partitioning algorithm for neuroannealing locates the first separating boundary for the new network. If this node is a leaf, then the algorithm separates the points using the vector separation algorithm in [4], which basically divides

the region along the midpoint in the axis of largest difference between the two networks for the current stratum. But if this boundary occurs at an internal node of the partition tree, then a new internal node must be created, and the point being inserted must be separated from every node under the span of the boundary node. In order to make this approach possible, each node in the partition tree must be marked with the representation (ℓ, s, c, w) that was used to create the node and the index of the tuple that was most recently used to separate the node. Note that the portion of this representation that creates the boundary is shared among all points under the space of the boundary node. For example, if the boundary occurs at s, so that $s' \neq s$ where s' is the size profile of the network being inserted, then it holds that every node underneath the boundary shares the size profile s. By separating s' from s using the vector separation algorithm, the inserted network is partitioned away from every node under the internal boundary node.

The basic partitioning algorithm in Section 2 described a partition tree that represents the entire area of the search space. In contrast, the hierarchical partitioning method only represents the area of the network topologies discovered at each point during execution. When neuroannealing is initialized, the area of the first topology inserted into the tree is used to compute the area of the whole tree for sampling purposes. Thus if the first point is $x_1 = (\ell_1, s_1, c_1, w_1)$, the partition tree is assigned the initial area $\lambda_{\ell_1,s_1,c_1}(X_{\ell_1,s_1,c_1})$. Whenever a point with a distinct topology is encountered, say, $x_2 = (\ell_2, s_2, c_2, w_2)$, the new node for this topology is assigned the area $\lambda_{\ell_2,s_2,c_2}(X_{\ell_2,s_2,c_2})$. Thus the total area of the partition tree is increased whenever a new topology is inserted. This increase is ignored for the purpose of sampling, as though the area of the new topology had previously been uniformly distributed among the existing leaf nodes. Since sampling from the tree is normalized, this effect is invisible.

The approach of adding new area as topologies are discovered avoids an otherwise troublesome problem of reallocating area from existing nodes in the trees. As a result, when a new topology appears, it immediately acquires substantial area, forcing some exploration of the new topology due to the factor $\lambda(E_n^a)$ in Equation 1. This effect parallels the use of speciation in NEAT, but is a natural mathematical property of the hierarchical partitioning method.

3.4 Network Mutations

Once neuroannealing has selected a network to mutate out of the partition tree, a sequence of NEAT-like mutations is applied to modify the network, possibly changing its topology. Eight types of mutation are employed, in the following order: (1) uniform crossover, (2) addition of a hidden layer, (3) removal of a hidden layer, (4) addition of a node to a hidden layer, (5) removal of a node from a hidden layer, (6) addition of a link between any two unconnected layers, (7) removal of an existing link, and (8) mutation of the weights with an area-sensitive Gaussian.

Neuroannealing selects a second point and applies crossover with probability 0.5. Crossover combines two networks to form a third network that shares properties of the two parents; this operation is useful for neural networks because networks are naturally modular. When crossover is used in neuroannealing, a second network is selected independently of the first using annealed tournament selection. The structure of the networks is aligned according to the indices of their layers, then the weights from any shared links are recombined using either uniform crossover (randomly selection from either parent) with probability 0.6 or intermediate crossover (averaging) with probability 0.4. The combined network retains the topology of the first parent, but integrates weights and biases from the second parent where they share structure.

After crossover, further mutations are performed with some probability in the order presented above. Only one such mutation is allowed. Once a layer, node, or link has been added or removed, no further structural changes are permitted. The probability for adding or removing a layer and for adding or removing a node is 0.01. The probability of adding or removing a link is 0.025.

When adding layers, neuroannealing uses a *chained layer* of hidden nodes that copies an existing layer of the network and adds two links. The first link runs from the copied layer to the chain layer with the identity matrix as the link weight matrix. The second link connects to a random layer in the network other than the chain layer, including possibly the copied layer. If the copied layer was already connected to the target layer, then the weights are also copied from the existing to the new link. Otherwise, the new weights are sampled from a Gaussian with variance $\hat{\sigma}^2$, defaulting to $\hat{\sigma} = 0.1$. A chain layer preserves the prior state of the copied layer into the next step. Successive chain layers can quickly add a short-term memory to the RNN that would otherwise be difficult to attain randomly.

When adding or removing links, any layer is chosen as the source, and any non-input layer as the target. If the link already exists, no new link is added. Or, if the link does not exist, no links are removed. This feature is intended to prefer networks with a medium number of links, and to prevent the removal of links in a sparse network.

If no structural mutations are performed, each existing weight of the network is randomly modified with probability 0.5 using a Gaussian that reflects the structure of the current partition of the space. The partition tree is traversed to obtain the current upper and lower boundaries on the weights of the potentially recombined network, u and ℓ, respectively. These boundaries are used to determine distinct variances for each weight or bias. Because the weight space is unbounded, these vectors may be infinite on either side. Therefore, u and ℓ are modified by using the cumulative distribution of the Gaussian,

$$\Phi_\gamma(z) = \frac{1}{\sqrt{2\pi}\gamma} \int_{-\infty}^{z} \exp\left(-\frac{x^2}{2\gamma^2}\right) dz, \tag{8}$$

reflecting the warping of the weight space that is also applied by the base measure of Section 3.2. The standard deviation for mutating each weight or bias is then given by

$$\sigma_{n,i} = \frac{\Phi_\gamma(u_i) - \Phi_\gamma(\ell_i)}{2 \log n}, \tag{9}$$

where n is the number of the generation and i is the index of the component within the weight and bias vector as used for partitioning in Section 3.3. Each weight or bias is mutated independently. Scaling the variance in this way preserves well-explored parameters, for which the distance between the upper and lower boundaries is small, while forcing exploration of parameters that have not been partitioned much. The extra logarithmic factor is used to compel faster convergence in higher dimensional spaces.

3.5 Neuroannealing Instantiation

With the previous subsections in mind, the complete neuroannealing algorithm can be stated. Neuroannealing is evolutionary annealing in the space of layered RNNs with annealed tournament selection using the base measure from Section 3.2 and the hierarchical partitioning algorithm of Section 3.3. Selected networks are mutated using the chain of mutations described in Section 3.4. The hidden and output layers of the RNNs use hyperbolic tangent activations.

All networks in the initial population have the same topology, which consists of a single input layer and a single output layer, with the input layer fully connected to the output layer. Within this topology, the initial weights and biases are chosen uniformly at random inside $[-\hat{\sigma}, \hat{\sigma}]$ where $\hat{\sigma}$ is the variance to be used when adding layers, nodes, and links. At initialization, the weights are intended to be small so that the activation can quickly change with new mutations, promoting fast exploration of the space.

Neuroannealing has four parameters that must be configured: (1) the population size K, (2) the learning rate η, (3) the space scale γ, and (4) the standard deviation, $\hat{\sigma}$. Based on preliminary experiments, a reasonable set of defaults is $K = 50$, $\eta = 0.1$, $\gamma = 1.0$, and $\hat{\sigma} = 0.1$. The defaults work well for all of the experiments below except for non-Markov double pole-balancing, where the values $K = 50$, $\eta = 0.025$, $\gamma = 2.5$ and $\hat{\sigma} = 0.25$ were used instead.

4. NEUROANNEALING EXPERIMENTS

Experiments were performed in three domains. The first domain is double pole-balancing, a standard benchmark task that is not fractured, and on which NEAT thus performs well. The next two domains, multiplexers and concentric spirals, have fractured state spaces on which NEAT does not perform well, and for which we hypothesize that neuroannealing will outperform NEAT.

4.1 Experimental Setup

For the experiments in this section, except as noted otherwise, both neuroannealing and NEAT were run for $1,000$ generations with a population size of 50, totaling $50,000$ evaluations. Each experiment was run 200 times for each method. The parameters for NEAT were set according to the defaults distributed with the publicly available C++ package, except for non-Markov double-pole balancing, where they were set to match [7]. Each task is now described in turn along with its experimental results.

4.2 Double Pole-Balancing

The double pole-balancing task is a control problem in which two poles are attached to a moving cart with hinges placed on a fixed length track. A successful controller must remain on the track and keep both poles within the tolerance for $100,000$ steps, or about half an hour of real time. Further details of the simulation can be found in the literature [7, 2].

The neural network is tasked with controlling the direction and magnitude of a force applied to the cart. Six state variables are available: the position and velocity of the cart, the angle and angular velocity of the first pole, and the angle and angular velocity of the second pole. In the Markov version of the task, all six variables are provided to the network, and the network output is scaled to $[-10, 10]$ and applied as the force; this problem can be solved without any hidden nodes. A non-Markov version of the task provides only the position and angles to the network, requiring the network to infer the velocities over time. This second task is more difficult, but can be solved with as few as two hidden nodes [7].

Table 1: Published results for selected methods on both versions of the Double Pole-Balancing task, as given by Gomez et al [2]. Reported quantity is the average number of evaluations before success, with failed trials excluded. Results for neuroannealing are new (as indicated by the asterisks), as well as the results for NEAT (determined experimentally using the parameters published by Stanley [7]).

Method	Markov	non-Markov
SANE	12,600	262,700
Q-MLP	10,582	–
Neuroannealing	*7,767	*7,499
ESP	3,800	7,374
NEAT	*1,819	*4,676
CMA-ES	895	3,521
CoSyNE	954	1,249

The fitness value of a network for double pole-balancing with or without velocities is the number of steps for which the cart remains on the track with the poles upright. The non-Markov task is more challenging and has so far only been solved through neuroevolution. The number of network evaluations required to solve the problem is available for several methods and is compared with the results for neuroannealing in Table 1.

Neuroannealing solved the double pole-balancing task in 84.5% of all trials for the Markov version, requiring on average $7,767 \pm 4,871$ time steps to reach the solution on average. On the non-Markov version, neuroannealing solved the task in 96% of trials in $7,499 \pm 3,157$ time steps on average. The averages here are taken over successful trials only. In comparison, NEAT solved both tasks on every trial, requiring $1,819 \pm 2,276$ steps in the Markov case and $4,676 \pm 2,107$ steps in the non-Markov case. The solutions to the double pole-balancing problem found by neuroannealing used few hidden nodes and did not show any evidence of bloat. It is interesting and important to note that neuroannealing performs better on the more complex version of the problem, justifying the claim that neuroannealing is better suited than NEAT to problems that require more complex networks.

Both neuroannealing and NEAT can solve the double pole-balancing task effectively. Neuroannealing pays a performance penalty for its more exhaustive search in this domain. However, in more fractured domains such a search pays off, as is seen in the non-Markov version of the problem.

4.3 Multiplexers

A multiplexer is a circuit that selects one of several input lines using a binary address. Multiplexers are used to implement computer memory circuits and are easily implemented in hardware. The function of a multiplexer is difficult for a network to learn because it requires the use of a large percentage of the binary input space. A single perceptron can only distinguish a fraction of the binary numbers, and thus multiple neurons must be used in concert to solve the

Table 2: Results of neural network experiments on the multiplexer problem. Columns show the probability of success (σ_ϵ^N), the average time to success ($\hat{\psi}_\epsilon^N$), and the average final percent misclassified (%). A trial is considered successful if the algorithm incorrectly maps at most 1%, 20%, 25%, and 30% of the possible inputs respectively on Mux12, Mux24, Mux35, and Mux36. Neuroannealing outperforms NEAT on the multiplexer problems in all categories; these results are statistically significant ($p < 0.01$).

Neuroannealing

Task	σ_ϵ^N	$\hat{\psi}_\epsilon^N$	%
Mux12	0.130	15,376 ± 15,930	10.8 ± 4.2
Mux24	0.047	19,833 ± 10,351	24.7 ± 3.7
Mux35	0.028	20,566 ± 15,509	28.5 ± 1.3
Mux36	0.036	17,675 ± 12,449	30.5 ± 1.3

NEAT

Task	σ_ϵ^N	$\hat{\psi}_\epsilon^N$	%
Mux12	0.000	∞	16.6 ± 2.7
Mux25	0.000	∞	27.9 ± 0.1
Mux35	0.000	∞	32.2 ± 0.1
Mux36	0.000	∞	34.8 ± 0.3

multiplexer problem. As a result, methods like NEAT have difficulty discovering the required complexity [3].

The experiments below test the ability of neuroannealing to learn multiplexers with four different inputs. Mux12 has one address line and four binary inputs. Mux24 uses two address lines and four binary inputs. Mux35 has three address lines and five binary inputs, while Mux36 has three address lines and six inputs. The versions with three address lines use less than the possible eight data inputs in order to simplify the task for neural networks. The task in each case is to learn a network that reads the binary address lines and outputs the binary input at the specified address line. The data inputs are numbered in the standard binary order.

The fitness function sums the error at each feasible address and data input. The network outputs are scaled to $[0, 1]$ for this purpose. The results in Table 2 show that neuroannealing performs better than NEAT on the multiplexer problems. On 13% of all runs, neuroannealing completely solves Mux12, whereas NEAT was unable to find a solution after 200 runs. The best solution discovered by neuroannealing for Mux24 was also completely correct, although the average solution achieved a fitness of 0.75 against an average of 0.72 for NEAT. On the versions of the problem with three address lines, Mux35 and Mux36, neuroannealing similarly performed well, with an average fitness of 0.72 and 0.70, compared to 0.68 and 0.65 for NEAT. The best fitness in 200 trials for neuroannealing on Mux35 was 0.97, and on Mux36 it was 0.92. The best networks on this task were indeed large. Typical solutions for neuroannealing used 4-6 layers with about 20 nodes. Thus neuroannealing solves the multiplexer problems better than NEAT because it discovers complex networks with high objective values that NEAT cannot reach. The next task, learning concentric spirals, reinforces this point.

4.4 Concentric Spirals

In the Concentric Spirals problem, originally introduced by Minsky and Papert, the state space is divided into two interlocking spirals, one "black" and the other "white", and the task is to label each point in the state space accordingly [6, 3]. The spate space is divided between the two spirals, and the resulting state space is shown in Figure 1(a). An evenly spaced 100×100 grid was overlaid on the state space over the region $[-6.5, 6.5]^2$, and the resulting $10,000$ points were used to test the networks.

The neural network for this problem has two inputs and one output. The Cartesian coordinates of the state space are passed to the network as input, and the single output should read 1.0 for black, and 0.0 for white. For this experiment, the objective function summed the errors at each output for every point on the 100×100 grid, scaled between 0.0 and 1.0. Thus the sigmoidal outputs of NEAT were used directly, and the hyperbolic tangent outputs of neuroannealing were shifted and scaled as required. It is possible to score a fitness of 0.67 on this problem by learning a correctly angled hyperplane on the state space. To achieve higher scores, the network must learn the spiral structure. Concentric spirals tests the ability of a network to distinguish nearby points in the state space that should be classified differently. In Kohl's terms, the state space is fractured. Such a task requires networks with many nodes to represent the space.

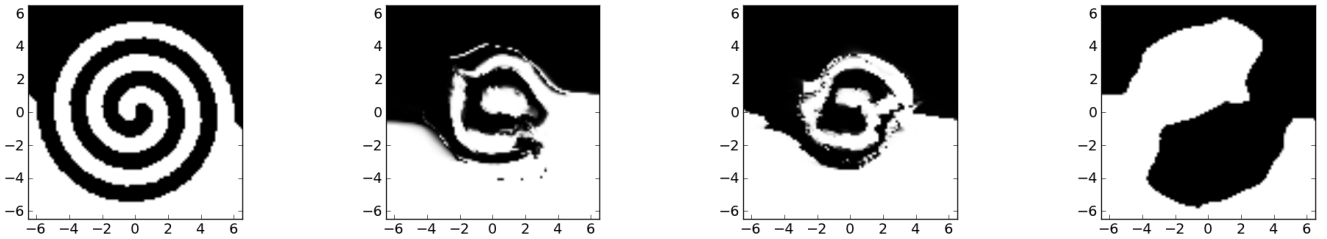
As expected, NEAT performed poorly, rarely exceeding the basic hyperplane solution with fitness 0.67. By contrast, neuroannealing outperformed the hyperplane approximation on about half of the runs, correctly classifying 69% of the points on the average. Complete results are in Table 3.

Figure 1 shows the learned classifications from several runs of neuroannealing. Over time, neuroannealing eventually discovers solutions that correspond to a spiral shape on the state space. Such solutions generally correspond to larger networks. Networks in the figure generally consisted of $4-7$ layers: The largest network, with 77 nodes, had a chained layer of size 37 that allowed correct classification of 30 extra points more than the network without the chained layer. As the networks become larger, they are better able to model the concentric spirals, but the learning progress slows down because larger networks have higher dimension. In general, the evidence supports the claim that neuroannealing is more capable of discovering complex solutions in this fractured domain in part because annealed selection follows suboptimal intermediate steps to arrive at more complex optima.

5. DISCUSSION AND FUTURE WORK

The experiments show that neuroannealing is an effective method for training neural networks in three domains: double pole-balancing, multiplexers, concentric spirals. Neuroannealing works well on these problems because it searches more thoroughly through complex networks and is not constrained by population size. Annealed selection enables neuroannealing to attempt more ways of increasing network complexity without forgetting previous solutions. Thus neuroannealing can step through regions of suboptimal fitness in order to find successful complex networks. When simple solutions exist, neuroannealing finds them. When complexity is required, however, neuroannealing considers progressively more complex solutions.

In double pole-balancing, neuroannealing does not find so-



(a) Goal State (b) $f = .7389$, 37 nodes (c) $f = .7494$, 30 nodes (d) $f = .7511$, 40 nodes

Figure 1: State space classification for the concentric spirals problem as learned by neuroannealing. Panel (a) shows the target classification on a 100×100 grid. Panels (b)-(d) show the best classifiers learned by neuroannealing in three different trials. Because the state space is fractured, more accurate solutions require larger networks. Neuroannealing is able to discover these solutions, whereas NEAT does not.

Table 3: Results of neural network experiments on the concentric spirals problem. The table shows the success probability (σ_ϵ^N), the average time to success ($\hat{\psi}_\epsilon^N$), and percent of points incorrectly classified (%). Success for concentric spirals is defined as correctly classifying 70% of the points in the domain. A hyperplane is sufficient to categorize 67% of the points, and this is the solution found by NEAT in almost every case. Neuroannealing is able to do better, outperforming NEAT substantially in all categories; this result is statistically significant ($p < 0.01$).

Concentric Spirals

Method	σ_ϵ^N	$\hat{\psi}_\epsilon^N$	%
Neuroannealing	0.261	21,687 ± 7,834	31.0 ± 2.1
NEAT	0.000	∞	33.1 ± 0.0

lutions as quickly as NEAT, ESP, or CoSyNE, but it does solve the problem. This success is achieved despite the fact that neuroannealing is designed to focus on thorough optimization rather than speed. Neuroannealing is a robust optimizer even in domains where NEAT performs well.

On the multiplexer problems and on concentric spirals, neuroannealing performs substantially better than NEAT, supporting the claim that neuroannealing is more capable of learning the fractured state space encoded in these problems. The size of the networks learned by neuroannealing can exceed those of NEAT by a full order of magnitude, as demonstrated by networks with up to 77 nodes in Figure 1. As noted by Kohl [3], these problems require complexity in order to be solved, and neuroannealing is able to deliver.

One qualification to these results is that the mutators and structure of neuroannealing have many features that are not shared in common with NEAT; the gains for neuroannealing may be due to the layered structure or the chain layer mutation, and not due to annealed selection. While this possibility cannot be ruled out at present, it is nonetheless clear that neuroannealing as a whole performs quite well.

Future research into neuroannealing could focus on determining the effect of the various mutation operators and tuning their parameters. In addition, the good use of chain layers suggest that there may be other large-scale agglomerative combination methods for constructing large neural networks from known modular components.

6. CONCLUSION

Neuroannealing is a method designed to work on fractured problems where more traditional neuroevolution methods fail. It is based on a martingale optimization, which makes it possible to construct more complex networks than before. Neuroannealing was shown to be an effective optimizer in diverse domains, including pole-balancing, multiplexers, and concentric spirals. In fractured domains, neuroannealing solidly outperforms NEAT due to its ability to discover larger networks with higher objective values. These results demonstrate that neuroannealing is an effective method for optimizing neural networks in challenging domains.

7. REFERENCES

[1] F. Gomez and R. Miikkulainen. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5, 1997.

[2] F. Gomez, J. Schmidhuber, and R. Miikkulainen. Accelerated neural evolution through cooperatively coevolved synapses. *Journal of Machine Learning Research (JMLR)*, 9, 2008.

[3] N. Kohl. *Learning in Fractured Problems for Constructive Neural Network Algorithms*. PhD thesis, University of Texas at Austin, 2009.

[4] A. Lockett and R. Miikkulainen. Real-space evolutionary annealing. In *Proceedings of the 2011 Genetic and Evolutionary Computation Conference (GECCO-2011)*, 2011.

[5] A. Lockett and R. Miikkulainen. Evolutionary annealing: Global optimization in arbitrary measure spaces. *Journal of Global Optimization*, 2013.

[6] M. A. Potter and K. A. D. Jong. Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation*, 8(1), 2000.

[7] K. O. Stanley. *Efficient Evolution of Neural Networks through Complexification*. PhD thesis, University of Texas at Austin, 2004.

[8] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2), 2002.