DXNN Platform: The Shedding of Biological Inefficiencies

Gene I. Sher

Abstract—In this paper I present a novel type of Topology and Weight Evolving Artificial Neural Network (TWEANN) system called Discover & eXplore Neural Network (DXNN). Among the novel features presented in this paper is a simple and database friendly tuple based encoding of NNs, a 2 phase neuroevolutionary approach aimed at removing the need for speciation, a new “Targeted Tuning Phase” aimed at dealing with “the curse of dimensionality”, and a new Random Intensity Mutation (RIM) method that removes the need for cross-over algorithms. I will discuss DXNN’s mutation operators and its built in feature selection method that allows for the evolved system to expand, discover, and explore new sensors and actuators. I will then compare the DXNN platform to other state of the art TWEANNs on a control task, to demonstrate its superior ability to produce highly compact solutions faster than its competitors, and then perform oblation experiments to demonstrate how various features of this new system affect its performance. Experiments will be conducted that demonstrate the platform’s ability to create NN populations with exceptionally high diversity profiles. Finally, the DXNN Platform will be used to evolve 2 dimensional organisms in an open-ended food gathering and predator-prey simulation, which will demonstrate the system’s ability to produce ever more complex Neural Networks, and the system’s applicability to problems in the artificial life and co-evolutionary domains.

Index Terms—Neural Network, TWEANN, Computational Intelligence, Evolution, Artificial Life, Evolutionary Computation, Feature Selection.

1. INTRODUCTION

Neural Networks (NNs) are universal function approximators capable of modeling complex mappings between inputs and outputs. As the problem posed to the NN increases in complexity, the size of the NN must also increase, and as the size of the NN increases, the difficulty of training that NN grows with it. Even before the NNs reach moderate complexity and size, hand setting the weights, topology, and other parameters of the NN becomes impractical. The standard training algorithms like the error back-propagation[11] get stuck too easily in the local maxima of even just moderately difficult problems, like the single pole balancing task for example. An efficient solution for an automated method of setting both, the topology and the parameters in the Neural Network, is accomplished through the application of evolutionary algorithms (EA). By applying EA to evolve weights, general parameters, and topology of the NN, highly complex problems can be solved. The systems capable of topology and weight evolving through EA are referred to as Topology and Weight Evolving Artificial Neural Networks, or TWEANNs. There are many approaches and algorithms when it comes to TWEANNs, but even with these highly advanced systems the “curse of dimensionality” persists[12] and prevents some systems from solving the more complex problems. Thus the search for ever more sophisticated TWEANN systems continues. One such new system is discussed in this paper.

The proposed NN system referred to as: Discover & eXplore Neural Network (DXNN), presents a number of new features and improvements over the currently published TWEANNs. DXNN allows us to further accelerate the production of topological solutions to various problems, while at the same time consistently producing much more compact NNs when compared to even the most advanced TWEANN systems [3][4]. Through a 2 phase approach to neuroevolution and with the utilization of a Targeted Tuning Phase, DXNN produces populations of ever increasing diversity and complexity. Furthermore, DXNN’s simple built-in feature selection system allows us to dynamically establish links to new sensors and actuators as they become available, making DXNN highly fit for artificial life and robotics problems.

This paper shall be organized as follows: Section 2 will introduce the general functionality of TWEANN systems and list the modern state of the art TWEANNs with short descriptions. Section 3 will provide a detailed description of the DXNN Platform and its encoding method. Section 4 will discuss DXNN platform’s learning algorithm. Section 5 will discuss the results of the double pole balancing benchmark for DXNN and other TWEANNs. In section 6 I shall perform feature variation and oblation experiments, demonstrating how the various features affect DXNN’s performance. Section 7 will demonstrate that DXNN possesses an exceptionally high population diversity. Section 8 will explain why cross-over is not utilized by DXNN. Section 9 will demonstrate the results of applying DXNN to an open-ended food gathering and predator-prey simulation. Finally, section 10 will end with a summary and future plans for this platform.

Note: In this document, “DXNN Platform” refers to the entire software package which builds NNs, supervises and monitors the NN population, and applies the mutational operators to the NN. The actual NNs created by the DXNN Platform are referred to as DXNNs.

2. MODERN TWEANN ALGORITHMS:

As the problem to be solved increases in complexity, a static NN becomes too inflexible to tackle it. If the NN is static, its size and topology must be guessed and hand designed...
perturbations, and the topological mutation operators are included in the evolution process. NEAT: uses genetic algorithms to both mutate the weights of the NNs and introduce new topologies, while EPN, GNARL also avoids utilizing cross-over. This is done to mitigate the problem of “the curse of dimensionality” and many state of the art TWEANN approaches can not be used for complex problems or open ended problems like Artificial Life. When using these two methods the topological solution for the problem must be known beforehand. Since topology is not evolved, it is not possible to generate minimal topologies for the problems either, unless such topologies are known beforehand.

CoSyNe: uses a cooperative co-evolution approach, where various permutations of neurons belonging to different groups are tried in combination to determine which of them work best in most combinations. Based on such criteria the final NN is then composed by combining the best and most generally fit neurons.

HyperNEAT: is an extension of NEAT. In HyperNEAT, NEAT is used to drive a substrate of Neurons. The substrate of Neurons is composed of a neural grid. Each neuron on the grid has a coordinate distributed uniformly between -1 and 1 on all axis of the substrate. Instead of using the NN to solve the problem directly, HyperNEAT uses the NN to generate weights for the Neurons on the grid, also referred to as a substrate. The weight is generated for every NN by letting NEAT produce a weight based on the coordinate of a given neuron and the coordinate of the Neuron linking to the given Neuron. When the substrate is a plane, the problem becomes a 4 dimensional one with the \([X_1, Y_1, X_2, Y_2]\) used as the input to the NN and \([W]\) the output. Thus, on a two dimensional substrate a weight \(W\) of a neuron \(A\) with coordinates \([X_b, Y_b]\) is determined by feeding a NN the problem directly, HyperNEAT uses the NN to generate weights for the Neurons on the grid, also referred to as a substrate. The weight is generated for every NN by letting NEAT produce a weight based on the coordinate of a given Neuron and the coordinate of the Neuron linking to the given Neuron. When the substrate is a plane, the problem becomes a 4 dimensional one with the \([X_1, Y_1, X_2, Y_2]\) used as the input to the NN and \([W]\) the output. Thus, on a two dimensional substrate a weight \(W\) of a neuron \(A\) with coordinates \([X_a, Y_a]\) connected to from a neuron \(B\) with coordinates \([X_b, Y_b]\) is determined by feeding a NN the vector \([X_a, Y_a, X_b, Y_b]\). This type of indirect encoding has shown to produce interesting generalization capabilities. In this paper I shall refer to this type of encoding as Substrate Encoding (SE).

EANT2: separates the learning approach into two steps, exploitation through CMA-ES (“Covariance Matrix Adaptation Evolution Strategy”)\([8]\), and then exploration through standard topological mutations. It too does not utilize mating.

Neither CoSyNe nor CMA-ES evolve a topology, instead they optimize a single general topology. Thus these approaches can not be used for complex problems or open ended problems like Artificial Life. When using these two methods the topological solution for the problem must be known beforehand. Since topology is not evolved, it is not possible to generate minimal topologies for the problems either, unless such topologies are known beforehand.

3. DXNN Topology and Elements:

The DXNN platform was created with scalability and artificial life experiments in mind. It was created to be implemented by concurrent languages like MPI and OCaml, and utilize the full power of distributed multi-core and multi-CPU hardware. The platform was made with the ability to change the functionality of activation functions, linking methods, learning methods, and other features by simply updating the configuration parameters of the independent
processing elements: Neurons and the Core. In DXNN each Node is represented as an independent mini server/client with its own address and concurrency to all other nodes. In the following sections I will first discuss the DXNN’s general architecture, and then follow by an elaboration on every element, its role, and its encoding.

3.1 The General Architecture:

DXNN is composed of 2 structural levels [Fig1]. At the lowest level are Neurons that form the NN. At the second level is the Core that monitors and supervises this NN, and interfaces it with sensors and actuators. In DXNN a Neuron can utilize any type of activation function, such as a sigmoid, gaussian, sine... What input the NN gets and what its output is used for is determined by the supervising Core. The Core is the element which controls and deals with pre/post processing of data, I/O of the sensors and actuators, and the interface with the OS. Core polls the sensors for data which come in vectors and passes those vectors to the appropriate input layer Neurons in its network. The Core then gathers the processed signals from the Neurons in the output layers, post-processes these output signals to produce output vectors, packages each vector in the form appropriate for the actuator it is destined for, summons the actuator programs, and finally passes each actuator its own vector.

To explain how the DXNN Platform functions, I will first cover the functionality of the Core, Neuron elements, and the flow of information within the system, and then follow up with the discussion on DXNN platform learning algorithm.

3.2 Core:

Core is the supervisor of the entire NN, it is a program that is the interface between the actual NN and the OS/Environment/Sensors/Actuators. When live it is represented as a mini server/client with an Id/Address, a SensorList, an ActuatorList, a ParameterList that can further augment the DXNN's functionality, and a list of Neuron_Ids that the Core supervises. When stored in a database, it is represented as a tuple: {Id, SensorList, ActuatorList, ParameterList, SupervisedNeuronIds, Generation, History}. SensorList is itself a list of tuples where each tuple is composed of a tag representing the name of a SensorProgram that the Core needs to run to get an input vector associated with a particular sense and an associated list of Neuron Ids/Addresses which should receive the signal from that SensorProgram. SensorList can be represented as follows: {{[Neuron_Id1...], InfraredSensor_Id}...{[Neuron_Id1...], ChemicalSensor_Id}}. ActuatorList is a list of tuples where each tuple is composed of a tag representing a name of an ActuatorProgram and an associated list of Neuron Ids/Addresses whose outputs are gathered, packaged and then forwarded to the ActuatorProgram. ActuatorList can be represented as follows: {{{[Neuron_Id1...], LegServos_Id}...{[Neuron_Id1...], CameraTiltPanServos_Id}}. When the Neurons send a signal to the Core, the Core gathers and sorts the signal for each Actuator, then calls the associated ActuatorProgram and passes it this vector. The ActuatorProgram parses the vector and executes its function, whether it be moving a virtual agent, writing a value into a database, moving an actual robot by driving the servos, or even modifying some part of the NN's own topology.

The “Generation” variable is an integer that increments every time the DXNN goes through a topological mutation phase. "History" is a list composed of all the mutations applied to the DXNN, listed in the order they were applied. The History list is composed of the following tuples: {MutationOperator, ElementAppliedTo, Info}. Where the MutationOperator is a tag/name of the mutation operator, ElementAppliedTo is an Id, and Info is extra information, if any, and depends on the type of MutationOperator.

When connected, Core, Neurons, Sensors, and Actuators might function as in the following example: The Core might begin by going through the SensorList, calling the programs and passing the resulting vectors to the appropriate Neurons. A SensorProgram can be one that polls a camera for data and then encodes an image as a vector of length n: [Val1,Val2,Val3...Valn] where Val is a scaled floating point. This vector is then passed to the Neurons for processing. At some later point, the Neurons in the output layer pass to the Core their output signals. Based on the Id/Address of the output layer Neurons, the Core chooses the appropriate ActuatorProgram, and then passes that actuator the accumulated Vector. The ActuatorProgram can for example control the servos to move a camera [Fig2] by sending it some signal which it derives by processing the vector. An example would be a vector: [Val1,Val2], which can represent the pan and tilt signals respectively. Afterwards, the Core begins to go through the SensorList again.
3.4 Neuron:

Neuron is a Level 1 element. A Neuron accepts vectors as inputs and outputs a resulting vector of length 1 [Fig3]. When live, the Neuron is a mini server/client program with an Id/Address, InputList, OutputList, ActivationFunction, LearningMethod, a WeightList, and a ParameterList which might further augment the Neuron's functionality. When stored in a database it is represented as a tuple: \( \{ \text{Id}, \text{InputList}, \text{OutputList}, \text{ActivationFunction}, \text{LearningMethod}, \text{WeightList}, \text{ParameterList}, \text{Generation} \} \). Neurons can have any type of Activation Function, from sigmoid to the mexican hat function. During the initial DXNN creation process and during the topological mutation phase, a random Activation Function (AF) is chosen from a list of available AF programs represented by a list of tags. In such a list each tag is a program name that can operate on a value passed to it. Thus, as soon as a new AF program is created, the name of that program can be added to the existing list as a tag, and later during a topological mutation phase be acquired by some neuron. Neurons also have a Learning Method (LM) which determines how to change the neuron’s weights over time. A LM is a program which accepts 3 parameters, a current weight list, an input vector, and an ActivationFunction. The output of the LM is an updated weight list and an output vector. The LM can be "hebbian" and output an appropriate output vector from the input vector and a modified weight list by applying the hebbian learning algorithm and using the ActivationFunction on the dot product. Like the AF list, the LM list can also be easily expanded, letting future neurons stumble upon new LMs through mutation or acquire them when initially created. Finally, all Neurons are initially created without a bias and can acquire that bias input through mutation.

Both, Neurons and the Core, have a Generation variable. The Core’s Generation is incremented every time it participates in the topological mutation phase, while the Generation variable of a Neuron is reset to that of the Core’s whenever it undergoes a mutation during the topological mutation phase. During the population initialization all Elements start with Generation equaling to 0. The purpose for which the Generation and History list is used will be explained in later sections.

Putting the Core and the Neurons together into one system [Fig1], produces the following flow of information: The Core’s list of sensors produce data vectors and distributes them to the appropriate Neurons in the input layer of the NN. The NN processes these vectors and produces an output that is passed to the Core. The Core receives this output from the output layer Neurons, packages it into vectors, and passes those vectors to their appropriate actuator programs. The actuators parse the vectors that are passed to them, and then act upon the environment. Afterwards, the Core polls the sensor programs for new data vectors once again and the cycle of information flow repeats.

3.5 Representing DXNN inside a Database:

The following encoding is used to represent DXNN within a database.

Population: \( \{ \text{Population_Id, DXNN_Id_List} \} \)

DXNN: \( \{ \text{DXNN_Id, Core_Id, ElementList} \} \)

ElementList: \( \{ \text{ElementTuple1...ElementTupleN} \} \)

Core Element: \( \{ \text{Id, SensorList, ActuatorList, ParameterList, SupervisedNeuronIds, Generation, History} \} \)

Neuron Element: \( \{ \text{Id, InputList, OutputList, ActivationFunction, LearningMethod, WeightList, ParameterList, Generation} \} \)

Most of the elements within these tuples are lists themselves and are represented in a similar fashion. Since these are all
nothing but lists of tuples, they are very easy to store in a relational database and traverse through. For example, to get at any Neuron one only needs a program that asks the population for a DXNN_Id, the proper DXNN then provides the Core_Id, and the Core_Id leads to the requested Neuron_Id. In this fashion, any mutation can be applied and any resulting topological perturbation due to the mutation can be followed and applied through these Id links.

4. THE DXNN ALGORITHM:

The DXNN Platform’s learning algorithm is divided into multiple phases. The Initialization Phase which is executed only once to create the seed minimalistic population of DXNNs. The Tuning Phase in which the DXNNs interact with the environment or some problem, and undergo parametric mutation. The Selection Phase during which some DXNNs are put into the fit (valid) group and others into unfit (invalid) group, letting only the valid DXNNs create offspring and themselves survive into the next generation. Finally followed by the Topological Mutation Phase during which mutational operators are applied to the DXNNs, affecting topology of the Neural Network, and the various parameters of the system in general. When all the phases complete, the DXNNs and their offspring (mutated versions of the valid DXNNs) are released back into the environment if the experiment is Artificial Life (AL), or applied again to the problem during the non AL experiments. All the steps can be conducted in real time without taking the DXNNs offline, the selection phase would run after a predetermined amount of time, destroying/taking offline the DXNNs with low fitness and creating mutated copies of the fit DXNNs and then releasing them into the environment, as will be demonstrated in the food gathering and the predator-prey simulation.

4.1 INITIALIZATION PHASE:

During the initialization phase, every element created has its Generation set to 0. Initially a seed population of size X is created. Each DXNN in the population starts with a minimal network, where the minimal starting topology depends on the total number of Sensors and Actuators the researcher decides to start the system with. If the DXNN is set to start with only 1 Sensor and 1 Actuator with vector length of 1, then the DXNN starts with a single Core containing a single Neuron. For example, if the output is a vector of length 1 like in the Double Pole Balancing (DPB) control problem, the Core contains a single Neuron [Fig4]. If on the other hand the DXNN is initiated with N number of Sensors and and K number of actuators, the Core will contain 2 layers of fully interconnected Neurons. The first layer which contains S Neurons, and the second A1+...Ak Neurons. Where S is the total number of Sensors, and A is the size of the vector that is destined for each Actuator. It is customary for the DXNNs to be initialized with a single Sensor and a single Actuator, letting the DXNNs discover any other auxiliary Sensors and Actuators through topological evolution.

All this information is kept in the Core, the Neuron neither knows what type nor originally from which sensor the signal is coming. Each neuron only keeps track of the list of nodes it is connected from and the vector lengths coming from those nodes. Thus, to the Neuron all 3 of the previous link-types look exactly the same in its InputList, represented by a simple tuple {From_Id, Vector_Length}. The Vector_Length variable might of course be different for each of those connections.

The different link-types add to the flexibility of the system and allow the Neurons to evolve a connection where they can concentrate on processing a single value or an entire vector coming from a Sensor, depending on the problem’s need. I think this improves the general diversity of the population, allows for greater compactness to be evolved, and also improves the NN’s ability to move through the fitness landscape. Since it is never known ahead of time what sensory values are needed and how they need to be processed to produce a proper output, different types of links should be allowed.

For example, a Core is routing to the Neurons a vector of length 100 from one of its Sensors. Assume that a solution
requires that a Neuron needs to concentrate on the 53rd value in the vector and pass it through a cosine activation function. To do this, the Neuron would need to evolve weights equaling to 0 for all other 99 values in the vector. This is a difficult task since zeroing each weight will take multiple attempts, and during random weight perturbations zeroing one weight might un-zero another. On the other hand evolving a single link-type to that Sensor has a 1/100 chance of being connected to the 53rd value, a much better chance. On the other hand, assume now that a solution requires for a neuron to have a connection to all of the 100 values in the vector. That is almost impossible to achieve, and would require at least 100 topological mutations if only a single link-type is used, but has a 1/3 chance of occurrence if we have block, all, and single type links at our disposal.

In a population, the Cores themselves can also be of different types: Type = “neural” which is covered in this paper’s benchmark, and a Type = “substrate” which is not. The “neural” type Core is a Core that supervises a standard recursive Neural Network. The substrate type Cores use their supervised NNs to drive a neural substrate, an encoding popularized by HyperNEAT. In such Cores the sensory vector is routed to the substrate and the output vector that comes from the substrate is parsed and routed to the actuators. The supervised NN is polled to produce the weights for the neurons in the substrate. The type of substrates can further differ in density, and dimensionality. Examples of different substrates are shown in [Fig5].

4.2 Tuning Phase:

The first step that must be taken is to construct/summon the Core and its Neurons for every DXNN from the list of tuples representing these nodes within the database. The database is scanned for the \{DXNN_Id, Core_Id, ElementList\} tuples, each tuple has its own Id to identify each separate DXNN. The ElementList containing the Core and Neuron tuples is then analyzed and depending on whether the correlated tuple represents a Core or a Neuron, a proper independent mini server/client is summoned for each, with the parameters and links specified by the data in that tuple. The Core then composes a list of New Generation Neurons (NGN). To create an NGN list the following steps are taken:

1. Neuron Ids are sorted based on the neuron’s generation, most recent(highest) to least recent(lowest).
2. Ids belonging to the 2 most recent generations are extracted, and designated CurGenIds.
3. Square root of the total number of remaining Ids is calculated, and then this number of Ids is extracted from the remaining Id list, starting from the most recent side. We designate this Id list: RecentGenIds.
4. \( \text{NGN} = \text{concatenate}(\text{CurGenIds}, \text{RecentGenIds}) \)

After NGN is composed, a variable MaxMistakes is created and set to \( \text{BaseMaxMistakes} + \sqrt{\text{TotWeights from NGNs}} \) rounded to the nearest integer. The BaseMaxMistakes variable is set by the researcher. Finally, a variable by the name AttemptCounter is created and set to 1.

The reason for the creation of the NGN list is due to the weight perturbations being applied only to the subset of these new or recently modified Neurons, a method I refer to as "Targeted Tuning". The reason to only apply perturbations to the NGNs is because evolution in the natural world works primarily through complexification and elaboration, there is no time to re-perturb all the neurons in the network after some minor topological or other type of addition to the system. As NNs grow in size it becomes harder and harder to set all the weights and parameters of all the Neurons at the same time to such values that produces a fit individual. A system composed of thousands of neurons might have millions of parameters in it. The odds of finding proper values for them all by perturbing random weights in random Neurons throughout the entire system after some minor topological mutation, all at the same time, is slim to none. The problem only becomes more intractable as the number of Neurons continues grow. By concentrating on tuning only the newly created or newly topologically/structurally augmented Neurons and make them work with an already existing DXNN, we make the problem much more tractable. Indeed in many respects it is how complexification and elaboration works in the biological NNs. In our organic brains the relatively recent evolutionary addition of the Neocortex was not done through some refurbishing of an older NN structure, but through a completely new addition of neural tissue covering and working with the more primordial and older parts. The Neocortex works concurrently with the older regions, contributing and when possible overwriting the signals coming from our more ancient neural structures.

During the Tuning Phase each DXNN tries to solve the problem. Afterwards, the DXNNs receive fitness scores based on their performance and a fitness function for that problem. After being scored the DXNN backs up its parameters. Every neuron in the NGN list has a probability of \( 1/\sqrt{\text{Tot}\_\text{NGNs}} \) of being chosen for weight perturbation. The Core sends these randomly chosen Neurons a request to perturb some of their weights. Each chosen Neuron when receiving such a request then perturbs its own weights. The total number of weights to be perturbed is chosen randomly by every Neuron itself. The number of weights chosen for perturbation by each neuron is a random value between 1 and square root of total number of weights in that Neuron. The perturbation value is chosen with uniform distribution to be
between \((\text{WeightLimit}/2)\) and \((\text{WeightLimit}/2)\), where the WeightLimit is set to \(\pi\). By randomly selecting the total number of Neurons, the total number of weights to perturb, and using such a wide range for the perturbation intensity, we can achieve a very wide range of parametric perturbation. Sometimes the DXNN might have only a single weight in a single Neuron perturbed slightly, while at other times it might have multiple Neurons with multiple weights perturbed to a great degree. This allows the DXNN platform to make small intensity perturbations to fine tune the parameters, but also sometimes very large intensity (number of Neurons and weights) perturbations to allow DXNN to jump over or out of local maximas that with small perturbations applied to a small number of Neurons and weights would pose an impossibility. This high mutation variability method is referred to in the DXNN platform as a Random Intensity Mutation (RIM). The range of mutation intensities grows as the square root of the total number of NGNs, as it logically should since the greater the number of new or recently augmented Neurons in the DXNN, the greater the number of perturbations that needs to be applied to make a significant affect on the information processing capabilities of the system. At the same time the number of Neurons/Weights affected during perturbation is limited only to the newly/recently topologically augmented elements.

After all the weight perturbations have been applied within the DXNN, it attempts to solve the problem again. If the new fitness achieved by the DXNN is greater than the previous fitness, then the new weights overwrite the old backed up weights, the AttemptCounter is reset to 1, and a new set of weight perturbations is applied to the DXNN. Alternatively, if the new fitness is not greater than the previous fitness, then the old weights are restored, the AttemptCounter is incremented, and another set of weight perturbations is applied to the individual.

When the DXNN's AttemptCounter reaches the value of MaxAttempts, implying that a MaxAttempts number of RIMs have been applied and not a single one produced an increase in fitness, the DXNN with its final best fitness and the correlated weights is backed up to the database through the conversion back to a list of tuples followed by a shut down of the DXNN itself. Utilizing the AttemptCounter and MaxAttempts strategy allows us, to some degree at least, test each topology and thus let each DXNN after the tuning phase to represent roughly the best fitness that its topology can achieve. In this way there is no need to forcefully and artificially speciate and protect the various topologies since each DXNN represents roughly the highest potential that its topology can reach in a reasonable amount of time after the tuning phase completes. This allows us to judge each DXNN purely on its fitness. If one increases the BaseMaxAttempts value, on the average each DXNN will have more testing done on it with regard to weight perturbations, thus testing the particular topology more thoroughly before giving it the final fitness score. The MaxAttempts variable grows in proportion to the square root of the total sum of NGN weights that should be tuned, since the greater the number of new weights that need to be tuned, the more attempts it would take to properly test the various permutations. To maintain the "reasonable amount of time" clause, the MaxAttempts variable is hard limited to 100.

4.3 Selection Phase

There are many TWEENNs that implement speciation during selection. Speciation is used to promote diversity and protect unfit individuals who in current generation do not possess enough fitness to get a chance of producing offspring or mutating and achieving better results in the future. The developer of NEAT states that new ideas need time to develop and speciation protects such innovations. Though I agree with the sentiment of giving ideas time to develop, I must point to [7] in which it was shown that such artificial and forced speciation and protection of unfit organisms can easily lead to neural bloating. DXNN platform does not implement forced speciation, instead it properly tests its individuals during the Tuning Phase and utilizes natural selection that also takes into account the complexity of each DXNN during the Selection Phase. In my system, as in the natural world, smaller organisms require less energy and material to reproduce than their larger counterparts. As an example, for the same amount of material and energy that is required for a human to produce and raise an offspring, millions of ants can produce and raise offspring. When calculating who survives and how many offspring to allocate to each survivor, the DXNN platform takes complexity into account instead of blindly and artificially defending the unfit and insufficiently tested Neural Networks. Speciation and niching should be done not forcefully by the researcher, but by the artificial organisms themselves within the artificial environments they inhabit, if such environments allow for such a feat. When the organisms find their niches, they will automatically acquire higher fitness and secure their survival that way.

Due to the Tuning Phase, by the time Selection Phase starts, each individual presents its topology in roughly the best light it can reach within reasonable time. This is due to the consistent application of RIM to each DXNN, and that only after a substantial number of continues failures to improve is the individual considered to be somewhere at the limits of its potential. Thus each Individual can be judged purely by its fitness rather than have a need for artificial protection. When individuals are artificially protected within the population, more and more Neurons are added to the NNs unnecessarily producing the dreaded topological bloat. Topological bloating dramatically and catastrophically hinders any further improvements due to a greater number of Neurons that need to have their parameters set concurrently to just the right values to get the whole system functional. An example of such topological bloating was demonstrated in the robot arm control experiment using NEAT and EANT2 [7]. In that experiment, NEAT continued to fail due to significant neural bloating, whereas EANT2 was successful. Once the NN bloats past a certain size, it simply can not find a solution due to the high number of Neurons that need to have their parameters set concurrently to a proper value. At the same time, most TWEENN algorithms allow for only a small number of perturbations to be applied at any one instance. Once a NN passes some topological bloating point, it simply can not generate enough of concurrent perturbations to fix the parameters of all the new neurons it acquired. In
DXNN, through the use of Targeted Tuning and RIMs applied during the Tuning and Topological Mutation phases, we can successfully avoid bloating. Indeed, as will be demonstrated during the experiments in later sections, the DXNN platform consistently produces highly compact NN solutions.

4.4 The “Competition” Selection Algorithm:

When all NNs have been given their fitness rating, we must use some method to choose those NNs that will be used for offspring creation. DXNN platform uses a selection algorithm I call “Competition”, which tries to take into account not just the fitness of each NN, but also the NN’s complexity. This selection algorithm is composed of the following steps:

1. Calculate the average energy cost of the Neuron using the following steps:

   \[
   \text{TotEnergy} = \text{DXNN1\_Fitness} + \text{DXNN2\_Fitness}(2)...
   \]

   \[
   \text{TotNeurons} = \text{DXNN1\_TotNeurons} + \text{DXNN2\_TotNeurons}...
   \]

   \[
   \text{AverageEnergyCost} = \frac{\text{TotEnergy}}{\text{TotNeurons}}
   \]

2. Sort the DXNNs in the population based on fitness. If 2 or more DXNNs have the same fitness, they are then sorted further based on size, more compact solutions are considered of higher fitness than less compact solutions.

3. Remove the bottom 50% of the population.

4. Calculate the number of allotted offspring for each DXNN:

   \[
   \text{AllotedNeurons} = \frac{\text{Fitness}}{\text{AverageEnergyCost}},
   \]

   \[
   \text{AllotedOffsprings}(i) = \text{round} (\frac{\text{AllotedNeurons}(i)}{\text{DXNN}(i)\_\text{TotNeurons}})
   \]

5. Calculate total number of offspring currently being produced for the next generation:

   \[
   \text{TotalNewOffsprings} = \text{AllotedOffsprings}(1) + ... + \text{AllotedOffsprings}(n).
   \]

6. Calculate PopulationNormalizer, to keep the population within a certain limit:

   \[
   \text{PopulationNormalizer} = \frac{\text{TotalNewOffsprings}}{\text{PopulationLimit}}
   \]

7. Calculate the normalized number of offspring allotted to each DXNN:

   \[
   \text{NormalizedAllotedOffsprings}(i) = \text{round}(\frac{\text{AllotedOffsprings}(i)}{\text{PopulationNormalizer}(i)}).
   \]

8. If \( \text{NormalizedAllotedOffsprings} (\text{NAO}) = 1 \) then the DXNN is allowed to survive to the next generation without offspring, if \( \text{NAO} > 1 \), then the DXNN is allowed to produce \( (\text{NAO} - 1) \) number of mutated copies of itself, if \( \text{NAO} = 0 \) the DXNN is removed from the population and deleted.

9. The Topological Mutation Phase is initiated, and the mutator program then passes through the database creating the appropriate \( \text{NAO} \) number of mutated clones of the surviving individuals.

From this algorithm it can be noted that it becomes very difficult for bloated NNs to survive when smaller systems produce better or similar results. Yet when a large NN produces significantly better results justifying its complexity, it can begin to compete and even push out the smaller NNs. This selection algorithm takes into account that a NN composed of 2 Neurons is doubling the size of a 1 Neuron NN, and thus should bring with it sizable fitness gains if it wants to produce just as many offspring. On the other hand, a NN of size 101 is only slightly larger than a NN of size 100, and thus should pay only slightly more per offspring.

4.5 Topological Mutation Phase:

An offspring of a DXNN is produced by first creating a clone of the parent DXNN, giving it a new unique Id, and then applying Mutation Operators to it. The Mutation Operators (MOs) that operate on the individual’s topology are randomly chosen with uniform distribution from the following list:

1. “Add Neuron” to the NN and link it randomly to and from randomly chosen Neurons within the Core, or one of the Sensors/Actuators.

2. “Add Link” (can be recurrent) to or from a Neuron.

3. “Splice Neuron” such that that two random Neurons which are connected to each other are disconnected and reconnected through a newly created Neuron.


6. “Add Bias” connection (all neurons are initially created without bias).

7. If the ratio of Neurons to Sensors exceeds some value \( S \), then the following mutation operator becomes available:

   "Add Sensor Tag", where a new Sensor Tag, if available, is added to the available SensorList. In this manner new connections can be made to the newly added sensors, thus expanding the sensory system of the NN.

8. If the ratio of Neurons to Actuators exceeds some value \( A \), then the following operator becomes available:

   "Add Actuator Tag", where a new Actuator Tag, if available, is added to the available ActuatorList. In this manner new connections can be made to the newly added actuators, thus expanding the types of tools or morphological properties that are available for control by the NN.

The "Add Sensor Tag" and "Add Actuator Tag" can both allow for new links (through the Core) from/to the Sensor and Actuator programs not previously used by the DXNN to
become available to the DXNN. In this manner the DXNN can expand its senses and control over new actuators and body parts. This feature becomes especially important when the DXNN platform is applied to the Artificial Life and Robotics experiments where new tools, sensors, and actuators might become available over time. The different sensors can also simply represent various features of a problem, and in this manner the DXNN platform naturally incorporates feature selection capabilities.

The total number of Mutation Operators (MOs) applied to each offspring of the DXNN is a value randomly chosen between 1 and square root of the total number of Neurons in the parent DXNN. In this way, once again a type of random intensity mutation (RIM) approach is utilized. Some mutant clones will only slightly differ from their DXNN parent, while others might have a very large number of Mutation Operators applied to them and thus differ drastically. This gives the offspring a chance to jump out of large local maximas that would otherwise prove impassible if a constant number of mutational operators were to have been applied every time independent of the parent NN's complexity/size. As the complexity and size of each DXNN increases, each new topological mutation plays a smaller and smaller part in changing its behavior, thus a larger and larger number of mutations needs to be applied to produce significant differences to the processing capabilities of that individual. When the size of the NN is a single neuron, adding another one has a large impact on the processing capabilities of that NN. When the original size is a million neurons, adding the same single neuron to the network might not produce the same amount of change in computational capabilities of that system. Increasing the number of MOs applied when the size and complexity of the parent DXNN increases allows us to make the mutation intensity significant enough to allow the mutant offspring to continue producing innovations in its behavior when compared to the parent. At the same time, due to RIM, some offspring will only acquire a few MOs and differ topologically only slightly and thus have a chance to tune and explore the local topological areas on the topological fitness landscape, while others will explore far and wide.

Because the sensors and actuators are represented by simple lists of existing sensor and actuator programs, the DXNN platform allows for the individuals within the population to expand their affecting and sensing capabilities. Such abilities integrated naturally into the NN lets individuals gather new abilities and control over functions as they evolve. For example, originally a population of very simple individuals with only distance sensors is created. At some point when enough fitness is achieved based on some criteria, and the DXNN itself is composed of enough Neurons, the "Add Sensor Tag" and "Add Actuator Tag" mutational operators become available. When either of these mutational operators is randomly applied to one of the offspring of the DXNN, that offspring then has a chance of randomly linking from or to this new Sensor or Actuator. In this manner the offspring can acquire sonar or other types of sensors present in the sensor list, or acquire control of a new body part and further expand its own morphology. These types of expansions and experiments can be undertaken in the artificial life/robotics simulation environments like the Player/Stage/Gazebo Project [9]. Player/Stage/Gazebo in particular has a list of existing sensor and actuator types, making such experiments accessible at a very low cost.

Once all the offspring are generated, they and their parents once more enter the tuning phase to continue the cycle.

5. Simple Experiments:

Three simple experiments will be discussed in this section. The first experiment will test whether DXNN can evolve the topology needed to solve the XOR problem when started with a single Neuron without a bias. The second and third experiment will be that of the double pole balancing with and without velocities as specified in [4]. The results produced by DXNN platform will then be compared with other state of the art TWEANNs. In each of the following experiments DXNN platform performs 100 runs with a population size limited to 10. Though DXNN benefits from using large populations, it can manage with very small populations due to the tuning phase. The parameter BaseMaxMistakes was set to 10 in the Double Pole Balancing (DPB) with velocity information experiment, and 20 in the DPB without velocity information experiments. To make the system comparable to other TWEANNs, only the hyperbolic tangent activation function was used, with the Learning Method parameter restricted to: “none”.

5.1 XOR Simulation:

The minimal requirement for a TWEANN is the ability to solve the XOR benchmark starting with a single Neuron. To learn to mimic XOR it is necessary for the NN to evolve at least a single hidden Neuron, thus demonstrating its ability to evolve the necessary topology.

The DXNN platform started with single Neuron topologies without bias. During 100 simulations the platform was able to find the solution 100% of the time, with NN solutions containing 2-3 Neurons. After having demonstrated that it could evolve rudimentary topologies, the DXNN Platform was applied to the double pole balancing problems.

5.2 Double Pole Balancing Experimental Setup

The simulation is created using a realistic physical model incorporating friction and fourth order Runge-Kutta integration. A step size of 0.01s was used, with DXNN producing Force values at 0.02s time steps.

The state variables for the problem were as follows:

1. Cart Position
2. Cart Velocity
3. Pole1 Position
4. Pole1 Velocity
5. Pole2 Position
6. Pole2 Velocity

At every time step DXNN receives scaled state variables from the simulation and outputs vector [N], where N is force. N is further scaled to be within the -10 and 10 range.

To pass the test, DXNN must balance 2 poles of different sizes (1m and 0.1m) for 100k time-steps (30 minutes of
simulated time). The two poles have initial positions of 4 degrees for the long pole and 0 degrees for the short pole. Both poles must be kept within 36 degrees of the vertical. Furthermore, the cart starts at the center of a 4.8 meter track at X = 0, and must remain within -2.4 and 2.4 meters of that center throughout the experiment. Finally, the Force produced by the DXNN is set to be no less than \((1/256)*10N\) as in [4]. The general setup of the experiment is graphically demonstrated in [Fig6].

A single experiment will run until either a maximum number of evaluations is reached, which I shall set to be 50000, or until the problem is solved. The average number of evaluations during the 100 experiments will be compared to the average number of evaluations taken by other TWEANN systems. An evaluation is counted every time DXNN is given a fitness, in other words, each perturbation of weights in the Tunning Phase and subsequent application to the domain counts as an evaluation.

5.3. Double Pole Balancing with Velocities:
Both, the Pole Balancing Simulation and the fitness function used are made to the specifications of [4]. The data for [Table1] is taken from [Table 3] in [4], where the average number of evaluations was calculated from 50 runs.

Table1

<table>
<thead>
<tr>
<th>Method</th>
<th>Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>RWG</td>
<td>474329</td>
</tr>
<tr>
<td>EP</td>
<td>307200</td>
</tr>
<tr>
<td>CNE</td>
<td>22100</td>
</tr>
<tr>
<td>SANE</td>
<td>12600</td>
</tr>
<tr>
<td>Q-MLP</td>
<td>10582</td>
</tr>
<tr>
<td>NEAT</td>
<td>3600</td>
</tr>
<tr>
<td>ESP</td>
<td>3800</td>
</tr>
</tbody>
</table>

All differences are statistically significant \((p < 0.001)\).

As can be noted by the results, DXNN platform outperforms all other systems, even those that did not have to evolve a topology. The DXNN sizes ranged from 1-2 Neurons, highly compact.

5.4 Double Pole Balancing without Velocities:
The DPB without velocities is a significantly more complex problem, requiring a recurrent NN to be evolved. Both, the Pole Balancing Simulation and the fitness function used are made to the specifications of [4]. The data for [Table2] is taken from [Table 4] in [4] where the average number of evaluations was calculated from 50 runs.

Table2

<table>
<thead>
<tr>
<th>Method</th>
<th>Without-Damping</th>
<th>With-Damping</th>
</tr>
</thead>
<tbody>
<tr>
<td>RWG</td>
<td>415209</td>
<td>1232296</td>
</tr>
<tr>
<td>SANE</td>
<td>262700</td>
<td>451612</td>
</tr>
<tr>
<td>CNE</td>
<td>76906</td>
<td>87623</td>
</tr>
<tr>
<td>ESP</td>
<td>7374</td>
<td>26342</td>
</tr>
<tr>
<td>NEAT</td>
<td>6929</td>
<td>6929</td>
</tr>
<tr>
<td>CMA-ES</td>
<td>3521</td>
<td>6061</td>
</tr>
<tr>
<td>CoSyNE</td>
<td>1249</td>
<td>3416</td>
</tr>
<tr>
<td>DXNN</td>
<td>2359</td>
<td>2313</td>
</tr>
</tbody>
</table>

All differences are statistically significant \((p < 0.001)\).

Using the undamped fitness function the DXNN platform produced highly competitive solutions with DXNN sizes ranging between 2 - 3 Neurons. DXNN Platform lost in its evaluation count only to CoSyNE, which did not have to evolve a topology. When damped fitness function was implemented, the DXNN sizes stayed between 2 - 3 neurons. The results show that DXNN platform in some cases outperforms even the topologically static methods, while at the same time consistently producing minimal topologies and outperforming all other TWEANNs.

Based on these results the DXNN Platform is shown to produce results much faster than other topology evolving algorithms. The topological compactness can be further increased by simply increasing the BaseMaxMistakes parameter. By setting this parameter to 100, DXNN Platform produces NN solutions composed of 2 neurons almost exclusively.

6. Parameter Variation and Feature Oblation
In this section I will be recreating the double pole balancing with damping experiments, only this time I will vary the numerous parameters of the DXNN Platform to demonstrate
how the average number of evaluations and solution compactness varies with the configuration parameters. For all the following experiments, initial/seed population was set to 10.

### 6.1 The effects of BaseMaxMistakes

The compactness of the solution can be improved by increasing the BaseMaxMistakes variable. As this variable increases, the DXNN Platform more thoroughly evaluates the true fitness of each topology.

As can be noted from [Table3], though it takes more evaluations to produce a solution as one increase this variable, the average number of neurons in the solution decreases. On the other hand, if BaseMaxMistakes and the Population Size variables are both set to very low numbers, below 10, the system begins to get stuck at times. The system is considered to have failed if it did not solve the problem after 50000 evaluations or 100 Topological Mutation Generations.

<table>
<thead>
<tr>
<th>BaseMaxMistakes</th>
<th>Population Size</th>
<th>Average # of Evaluations</th>
<th>Average # of Neurons</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>1596</td>
<td>8.5</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>1545</td>
<td>4.7</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>2084</td>
<td>3.59</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>2313</td>
<td>2.74</td>
</tr>
<tr>
<td>30</td>
<td>10</td>
<td>2803</td>
<td>2.73</td>
</tr>
<tr>
<td>50</td>
<td>10</td>
<td>2951</td>
<td>2.62</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>3919</td>
<td>2.44</td>
</tr>
</tbody>
</table>

### 6.2 The effects of Population Size

As can be noted from [Table4], DXNN solutions become more compact as one increases the Population Size. Once again we can note that the system begins to fail when the Population Size limit is decreased to very small numbers.

<table>
<thead>
<tr>
<th>Population Size</th>
<th>BaseMaxMistakes</th>
<th>Average # of Evaluations</th>
<th>Average # of Neurons</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>10</td>
<td>1968</td>
<td>3.74</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>2084</td>
<td>3.59</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>2214</td>
<td>3.1</td>
</tr>
<tr>
<td>30</td>
<td>10</td>
<td>2274</td>
<td>2.79</td>
</tr>
<tr>
<td>50</td>
<td>10</td>
<td>2953</td>
<td>2.61</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>4576</td>
<td>2.38</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>4215</td>
<td>2.76</td>
</tr>
<tr>
<td>10</td>
<td>50</td>
<td>2951</td>
<td>2.62</td>
</tr>
</tbody>
</table>

### 6.3 The effects of RIM

One of the primary claims in this paper is that Random Intensity Mutations, or RIM, is an important addition to a TWEANN system and a superior substitute to cross-over algorithms. RIM acts as a very efficient approach to the exploration and exploitation of the topological and parametric space. In [Table5] I demonstrate what happens when I slowly decrease RIM intensity. For these experiments, the population limit was set to 10.

<table>
<thead>
<tr>
<th>Weight RIM</th>
<th>BaseMaxMistakes</th>
<th>Average # of Evaluations</th>
<th>Average # of Neurons</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Pi to Pi</td>
<td>50</td>
<td>2951</td>
<td>2.62</td>
</tr>
<tr>
<td>-Pi/2 to Pi/2</td>
<td>50</td>
<td>3880</td>
<td>2.66</td>
</tr>
<tr>
<td>-1 to 1</td>
<td>50</td>
<td>4106</td>
<td>2.68</td>
</tr>
<tr>
<td>-0.5 to 0.5</td>
<td>50</td>
<td>5135</td>
<td>2.92</td>
</tr>
<tr>
<td>-0.3 to 0.3</td>
<td>50</td>
<td>14594</td>
<td>5.94</td>
</tr>
<tr>
<td>-0.2 to 0.2</td>
<td>50</td>
<td>25038</td>
<td>11.76</td>
</tr>
<tr>
<td>-0.1 to 0.1</td>
<td>50</td>
<td>Failed</td>
<td></td>
</tr>
</tbody>
</table>

It can be observed that as the RIM intensity is decreased, the system begins to take longer and longer to solve the double pole balancing problem, and eventually begins to fail completely. At the same time, the average number of evaluations is getting closer to what is seen from the standard evolutionary algorithm systems, 25000+ evaluations. With RIM, even when BaseMaxMistakes is taken to be 50, the DXNN Platform is able to produce solutions with an average number of evaluations in the 3000 area, and NN size of 2-3.

Cross-over is the vestigial, and inefficient method that somehow was picked up by the GA community from the biological field. But we don't need it, technology allows us to create algorithms and methods that are much more robust and dynamic than any bio-based cross-over approach. Parameter and topologically wide RIM utilization, is one of such approaches.

### 7. DXNN Population Diversity Profile

In this section I demonstrate that DXNN platform is able to produce populations of excellent diversity. This property is due to the two phase approach. After every tuning phase, the remaining 50% of the population produce topological mutants, which by their very definition have to differ topologically from their parents. DXNN is able to do this
This work has been submitted to the IEEE for possible publication.

efficiently because the tuning phase allows it to thoroughly test every topology in its parameter space. The result is a population that thoroughly explores topological space due to the Topological RIM, while at the same time has its NNs explore and be fine tuned in the parameter space due to the Parametric RIM in the tuning phase.

In the following experiments a NN is considered topologically different from the others if it has either a different number of input connections, a different number of output connections, a different number of neurons in total, or a different set of activation functions. The organisms can of course differ even further, since even if two NNs have the same activation function set, the same number of neurons, input and output links, they can still differ in how the neurons are actually interlinked. But nevertheless, this simple count provides a good method of calculating the minimum amount of diversity in a population of small NNs.

To get the diversity value, I first calculate the noted features for each organism in a population, and then group the organisms based on those features. The total number of different groups is the diversity value. This diversity value is calculated after the entire population finishes with its tuning phase. In the following graphs, the Generation value is plotted on the X axis while the topological diversity value is on the Y axis. In [Fig7] the population had access only to the Sigmoid Activation Function, while in [Fig8] Sigmoid, Sin, and Gaussian AFs were available.

Every point is produced by averaging the diversity values from 50 experiments. The rather extended topological generation range is due to the fact that every set of 50 experiments had 1 or 2 experiments that took more than an average number of topological generations to complete.

In [Fig7] where only the sigmoid activation function is used, we can clearly see that minimum diversity reaches 25-90% of the population within the first 5 generations. In [Fig8] where the available activation function list is composed of sigmoid, sin, and gaussian, the population diversity is even higher. Both of these graphs demonstrate that DXNN has a very healthy population diversity profile, and that even for such small networks a very wide range of topologies is present and explored at all times.

This diversity profile signifies that as DXNN Platform is allowed to operate with larger and larger populations, it will explore far and wide within the topological space. Even when the population has almost found the solution to the problem, we do not see any type of significant drops in the diversity profile. This implies that the population is not being taken over by any one particular topology. This trait is present because 1. Due to the “Competition” selection algorithm, an organism needs orders of magnitude higher fitness to push out all other organisms from the population. And 2. Even if an organism does somehow manage to push out all other organism from the population, the next generation will still be composed of a highly diverse population since the offspring can only be topological mutants.

8. Why DXNN Platform does not use Crossover.

I think that when one uses too tiny and too few mutations, there are limits to the local fitness maximas that a system can jump over or out of, thus frequently becoming stuck in those parts of the fitness landscape. Cross-over, though having a poor chance of producing superior offspring, does every once in a while kick the system into another region of the fitness landscape without completely ruining the already evolved solution. In this manner crossover can save a NN from getting stuck. The same type of jolt only in a more controlled manner can be easily achieved through large scale mutations, and it is for this reason why Random Intensity Mutation (RIM) is employed by the DXNN Platform. RIM allows for both small and large topological and parametric changes to take place, letting the DXNNs explore and exploit different areas of the fitness landscape, avoid getting stuck in local maximas, and completely remove the need for complex mating algorithms.

With regards to crossover, I think that perhaps over time the Evolutionary Computation community became too caught up in trying to emulate nature, and forgot to ask about the
principles and benefits behind what they were trying to emulate.

In “The Selfish Gene” [10], Richard Dawkins writes a few paragraphs on the role that sexual reproduction and mating plays and why it has evolved in nature. He writes that “Efficiency from the whole individual’s point of view is then seen to be irrelevant. Sexuality versus non-sexuality will be regarded as an attribute under single-gene control, just like the blue eyes versus brown eyes. A gene for sexuality manipulates all other genes for its own selfish ends. So does a gene for crossing-over. There are even genes called mutators that manipulate the rates of copying-errors in other genes. By definition, a copying error is to the disadvantage of the gene which is mis-copied. But if it is to the advantage of the selfish mutator gene that induces it, the mutator can spread through the gene pool. Similarly, if crossing-over benefits a gene for crossing-over, that is a sufficient explanation for the existence of the crossing-over. And if sexual, as opposed to non-sexual, reproduction benefits a gene for sexual reproduction, that is a sufficient explanation for the existence of sexual reproduction. Whether or not it benefits all the rest of the individual’s genes is comparatively irrelevant...” In short, he writes that it is not that mating has evolved due to it producing superior organisms, but instead due to it being a self perpetuating genetic fluke. Thus perhaps it is time to stop trying to copy this particular aspect of nature for the mere sake of being faithful to it. Perhaps a more controlled method that produces superior results is in order, with all the benefits of population diversification but with none of the complexities of mating algorithms. I think, and as I’ve hopefully demonstrated in sections 5, and 7, RIM is one of such approaches. It explores, exploits, and in combination with the separation of parametric and topological mutation phases, produces incredible diversity and rapid positive evolution.

9. Open-ended artificial life experiments

To test DXNN on a more complex project, I will apply it to a food gathering and a predator-prey problem. In these problems DXNN will be used to evolve the brains of artificial robots inhabiting a two dimensional world. The first experiment will demonstrate that DXNN Platform can evolve robots which can successfully use a 2 wheel differential drive to move around the 2d environment and gather food. The second experiment will scatter poison among the food and gather food. The second experiment will scatter poison among the food and see if DXNN platform can evolve NNs that can gather food and avoid poison. Finally, the third experiment will evolve 2 separate populations inhabiting the same 2d environment, a predator population and a prey population, thus demonstrating that DXNN can be applied to the problems in the complex field of co-evolution.

9.1 Environmental setup

In these experiments I shall create a 2 dimensional environment without walls. Two robot types will be created, predator and prey, as shown in [Fig9]. These two robot types will be contained in different populations, since their goals and representations are mutually exclusive. Both robot types will start with 1000 energy points. Every action will cost energy, when a robot moves 1 meter, it costs 1 energy point. When a robot turns 90 degrees, it costs 1 energy point. When the robot moves at the fraction of the speed or turns a fraction of 90 degrees per step, the energy used for that action is linearly scaled to the same fraction. Another 0.1 energy points is subtracted from the robot during the use of any actuator with any level of intensity, ensuring that if the robot sits still, it will still eventually lose all of its energy, and be removed from the environment. Finally, the robot will be automatically killed after it uses its actuator 20000 times. This will prevent robots from living indefinitely, and should produce an evolutionary push towards a strategy that maximizes the total number of prey caught, or plants eaten per the allocated number of maximum steps/actions, 20000. BaseMaxMistakes is set to 25. The population size is set to 10 for Simple Food gathering, 10 for Dangerous Food Gathering, and 10 for each population in the Predator vs. Prey experiment.

The predator, represented by a red circle with an arrow, can not eat plants and can only gain energy by eating prey. When a predator comes in contact with a prey, the prey dies and the predator’s energy is incremented by the total amount of energy that the prey had at that point in time. The prey, represented by a blue circle, can gain energy only by eating plants. Plants are represented by small green circles, and poison is represented by small black circles. After being eaten, both regrow at a random coordinate calculated by taking a random number from a uniform distribution between 0 to 800 for X, and 0 to 500 for Y. A plant gives the Prey 500 energy points, and a poison gives -2000 energy points. The organism dies when its energy reaches zero.

The fitness of a robot is calculated using the following equation:
StayAlivePoints = TotStepsAlive/1000 for the first 1000 steps, and 0 for the rest.
Prey: Fitness = StayAlivePoints + PlantsEaten
Predator: Fitness = StayAlivePoints + PreyEaten

When a Prey or Predator dies, it is removed from the environment, its fitness score is calculated, and then a parametrically mutated version is dropped back into the environment in a random location. When the tuning phase ends for a DXNN, a new mutant offspring is created from a random dead DXNN. The dead DXNN is selected using an Augmented Competition (AC) selection algorithm. The AC selection algorithm keeps a list of size PopulationSize of dead DXNNs. When a DXNN dies, first it is entered into the list of these dead DXNNs, then the lowest scoring DXNN is removed from the list permanently, and then the Competition selection algorithm is applied. In this augmented version of the competition selection algorithm, the AlottedOffspring variables are converted into normalized probabilities used to select a parent to produce a new mutated offspring. Finally, there is a 10% chance that instead of creating an offspring, the parent itself will enter the environment. Using this “re-entry” system, if the environment or the manner in which the fitness is alloted changes, the old strategies and their high fitness scores can be re-evaluated in the changed environment to see if they still deserve to be in the dead pool.
The robot sensors are basic “visual cone” scanners. The visual area is separated into N angular sectors from the robot’s perspective, and the signal for that sector is a number that equals 1/Distance, where Distance is the distance to the closest Element to the robot in that angle of view. If there is nothing in that cone of view, the value is -1. Thus for example, a Plant_Sensor positioned at the origin and looking straight in the positive Y direction, with the closest plant located in the first quadrant and 1 meter away with emptiness in all other quadrants, will produce the following vector output to the robot: [1,-1,-1,-1]. If the robot were to have been facing in the positive X direction, the sensory vector would be: [-1,-1,-1,1], since the sectoring of the visual field is counter clockwise from the perspective of the organism.

The 2d representations of Prey, Predator, Plant, and Poison are shown in [Fig 9].

Due to the computational cost of these experiments, each simulation is ran only 10 times in total, for 100000 evaluations each. Finally, for all 3 of the following experiments, Add Sensor Tag and Add Actuator Tag mutation operators will become available to a DXNN with a probability of 25% if it has a ratio of 10+ Neurons/Sensors, and 10+ Neurons/Actuators respectively.

Finally, there is a slight difference in the location where Plant/Poison/Prey and Predators are spawned and re-spawned. As noted above, the Plants and Poison are spawned in the X=(0-800) and Y=(0-500) rectangle. The Prey are also spawned in the same area. On the other hand the Predators, in the Predator vs. Prey experiment, are spawned and re-spawned in X(800-1400) and Y=(0-500) rectangle. This is done to ensure that there are no instances where Predators and Prey spawn in the same location, resulting in the Prey being killed without any type of evaluation.

9.2 Simple Food Gathering

In this simulation there are only plants and prey. The Prey inhabiting the 2d environment are controlled by DXNNs. Prey are started with a simple Plant_Sensor that has a resolution of 4. The graph in [Fig 10] shows the average population fitness vs. evaluations. The graph in [Fig 11] shows NN size vs evaluations. The average population fitness and NN size values were calculated every 500 evaluations.

From the graphs we can see that the prey quickly learn to navigate and move towards the food sources. The prey begin to demonstrate adaptation after only 5000 evaluations. After 25000 evaluations the prey learned to eat a plant and continue moving in the same direction, then switch to a plant that is the smallest angle away from their current direction. By the end of 50000 evaluations, the prey learned to move from one plant to the next based on the shortest distance, though the full population continued to contain both types of strategies to the end of the experiment [14]. The average NN sizes ranged from 6-20 neurons.

9.3 Dangerous Food Gathering

In this simulation the 2d environment is filled with prey controlled by the DXNNs, plants as in the previous experiment, but also poisonous plants that are scattered around in the same area as the plants. In this experiment the Prey are started with a PlantAndPoison_Sensor that has a resolution of 4, and a total vector output of 8. This sensor is like the Plant_Sensor, but it also senses poisonous plants, represented by the black circles in the environment [15].
The graph in [Fig12] shows the average population fitness vs. evaluations. The graph in [Fig13] shows NN size vs evaluations. The average population fitness and NN size values were calculated every 500 evaluations.

As the fitness plot demonstrates, the prey eventually learn to pick up the plants even with the presence of poison. On average, after the first 15000 evaluations the Prey evolved to properly move around to gather food and avoid poison. After 50000 evaluations, in most experiments the Prey reached a stable fitness score. The strategy that evolved on most of the occasions is to gather as many plants as possible as quickly as possible, even if it means consuming poisonous plants every once in a while. The average NN size was 9-23 neurons. On 3 of the 10 experiments the evolved strategy followed a more conservative approach, where avoiding poison took precedence over gathering plants. In those experiments the average fitness scores were 2X to 3X lower.

In general, the fact that navigation through the poison infested environment is at all possible with such low resolution (R=4) sensors is already remarkable, for it means that the Prey learned to move around and discern what is poison and what is plant when the plant and poison are in very close proximity.

9.4 Predators vs. Prey

In this simulation there are plants, prey and predators. The prey and predators are controlled by the DXNNs. The Prey and Predators both start with a PlantPredatorPrey_Sensor, working analogously to the PlantAndPoison_Sensor. Both of the species have access to the following list of sensors that they can link to through topological mutations: Prey_Sensor, Predator_Sensor, Plant_Sensor, and Internal_Energy_Sensor.

It is not useful creating a graph of fitness or complexity vs. the number of evaluations, since the behavior was different in most of the experiments, making the average meaningless. It seems that this highly dynamic environment with free interaction between multiple organisms, and the evolution of one organism based on the behavior of the other, produces different results in almost every experiment.

Both prey and predators never became larger than 30 Neurons. Among the strategies evolved, one of the more prevalent end game strategies is shown in [16]. In this strategy the predators chose to attack the prey from the side, or wait around and then when the prey are near by, simply attack from the front instead of chasing them around. In the context of the fitness function and the environmental settings used, this strategy makes a lot of sense. The Prey have to keep moving around and eating plants, otherwise they die from either losing all energy, or simply spending too much time running away from Predators, and thus not eating enough plants and getting the fitness score which would allow them to create offspring. On the other hand, if the Predator spends its time simply chasing Prey, since both have the same top speed, the Predator would simply lose all its energy. Thus, the best strategy is to ambush, since the prey has to move from one plant to another, catching the prey head-on or from the side is the best move. For the Prey, not bothering to avoid Predators is the best strategy for this fitness function and evolutionary setup, since any prey that avoids predators will end up running away, with both the predator and the prey having their fitness score suffer due the wasted time which is a limited commodity. Due to chance alone, there will be those prey that are not caught/chased by the predators, and thus be able to just move around and eat the plants at full speed, such prey will have a higher score than the prey that also tries to avoid the predators. Thus making this resulting end game[16], a logical strategy. Changing the fitness function to make survivability, or some other aspect a priority, would certainly produce a different result.

Another interesting observation was that during the 10 experiments, when the Prey were the first to learn how to gather plants, the Predators took a lot longer to catch up and learn how to hunt the Prey. When the Predators learned how to catch the Prey first, the Prey took a very long time to learn how to gather food. This makes sense, because when Prey learn how to gather food they become much harder targets due to their constant movement, and so evolving a hunting strategy in such an already dynamic environment becomes more difficult. On the other hand, when the Predators learn how to catch prey first, the Prey do not have enough time to learn to gather plants, since they get eaten before their plant eating strategy is properly evaluated. The evolutionary path
of both organisms was very sensitive to early conditions, and the competing population’s progress.

9.4 DISCUSSION

DXNN produced the strategies for all 3 problems very rapidly. The strategies were efficient, given the fitness function that concentrated on maximum plants eaten or prey caught, the time limit, and the dynamical nature of the 3rd problem. The minute long movies of the various stages of all 3 of these experiments are available in [17]. Which demonstrate various strategies that evolved, and the general behavior of the population during the various phases of evolution.

10. SUMMARY AND CONCLUSION

In this paper I presented the DXNN Platform, a novel Topology and Weight Evolving Artificial Neural Network platform that separates the parametric and topological mutation phases. DXNN uses a database friendly, tuple based, human readable, and non analog encoded NN representation. DXNN demonstrated its superior performance in double pole balancing with velocity information, and double pole balancing without velocity information benchmarks. This paper then further demonstrated that this system has an exceptional population diversity profile, and that RIM and its other novel features do indeed play significant factors in its ability to produce results with high efficiency. Finally, DXNN was tested on a food foraging and predator-prey simulation, where the generated NNs controlled the brains of virtual robots inhabiting a 2 dimensional environment. The artificial life experiment results demonstrated that the DXNN platform can evolve the strategies needed to survive in such environments, and that it can also successfully be implemented in co-evolutionary problems. I think that all combined, the results in this paper demonstrate that DXNN and its approach to neuroevolution, is a strong contender for state of the art.

REFERENCES


[14] Simple food gathering simulation: http://www.youtube.com/watch?v=r7q9p00nWk4


[16] Predator Vs. Prey simulation: http://www.youtube.com/watch?v=YybeD-8cdNY

[17] Various phases and examples in the 2d robot simulations: http://www.youtube.com/user/DXNNsystem