

such as weight decay is also needed, for instance:

$$\Delta w_{ij} = \alpha_{ij} o_i o_j - \beta_{ij} w_{ij}, \quad (4.21)$$

where  $w_{ij}$  is the weight between neurons  $i$  and  $j$  with activations  $o_i$  and  $o_j$ , and  $\alpha_{ij}$  and  $\beta_{ij}$  are learning and decay rate parameters. Unlike gradient descent, Hebbian learning is entirely local to each connection and requires no learning targets at the output. In this sense, it is closer to biological learning than gradient descent, and therefore a proper comparison to adaptation based on recurrency. Note that Hebbian learning also provides an alternative that avoids the second question in this section, i.e. where the targets for development come from—it does not need them. On the other hand, it cannot take advantage of targets either, and therefore it is generally not as powerful as gradient descent.

Nevertheless, Hebbian learning is a compelling approach to developmental indirect encoding on its own. Networks with Hebbian learning can change their behavior based on what they observe during their lifetime. For instance, they can evolve to first perform one task such as turn on a light, and then switch to another such as travel to a target area (D. Floreano and J. Urzelai 2000). While it is biologically plausible, an interesting practical question arises: Can such low-level adaptation be more effectively implemented through recurrent activation?

The above foraging domain with good and bad food items can be used to study this question (Stanley, Bryant, and Miikkulainen 2003). The usual NEAT method for evolving recurrent networks can be compared with a version that takes advantage of Hebbian learning: It evolves the learning rate and decay rate parameters  $\alpha_{ij}$  and  $\beta_{ij}$  for each connection, in addition to the weights and the network topology. Each evolved network is placed into the foraging environment where it can consume food items; if an item is good, it receives a pleasure signal, and if bad, a pain signal. All items in a trial are the same so after it consumes the first item, it needs to either eat all of them or none of them to receive maximum fitness.

While both approaches evolved successful networks, NEAT without adaptation required about half the generations to do so. There were fewer parameters to optimize, and evaluations were more consistent. Indeed the solution networks look very different (Figure 4.6): While the fixed-weight recurrent networks were parsimonious with recurrency focused at the output, the adaptive networks were more complex and holistic, using many more adaptive weights throughout the network. Because many weights adapt, it was not possible to rely on only a few loops, and the behavior became encoded redundantly throughout.

Thus, in such a simple task recurrency was more effective than Hebbian adaptation. It is of course possible that in more complex situations adaptation provides additional power that may be needed. What such tasks might be is an interesting topic for future work.

### 4.3 Indirect Encoding Through Hypernetworks

A common features of the indirect encodings we have encountered in the previous sections is that a specific phenotypic component at a given point in development influences the states of nearby components. In other words, here development progresses through local interactions. This section reviews a particularly popular indirect encoding that, when

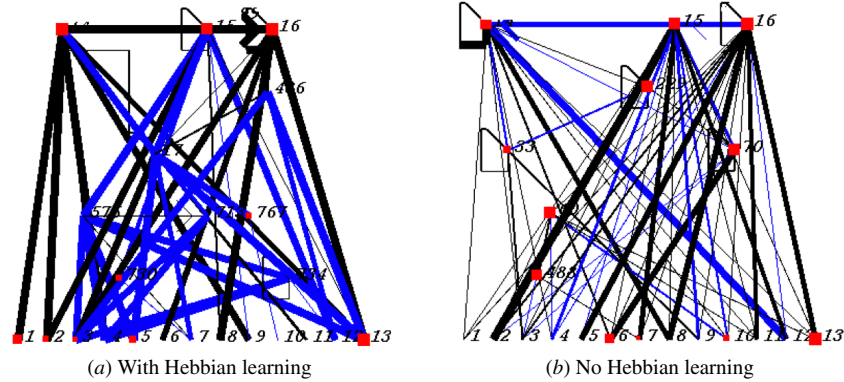


Figure 4.6: **Networks evolved with NEAT with and without Hebbian learning.** Nodes are numbered through historical markings. Black lines represent excitatory and blue lines inhibitory connections; loops indicate recurrent connections; line thickness corresponds to connection weight. (a) With Hebbian adaptation, performance is encoded more holistically, utilizing plastic synapses throughout the network. (b) Without Hebbian adaptation, the network is more parsimonious, with adaptation coded into recurrent connections at the outputs. While both types of solutions are successful, Hebbian adaptation provides a larger search space that is more difficult to navigate. In simple tasks at least it is thus more effective to rely on recurrency to represent adaptation. (Figure from Kenneth O. Stanley 2003).

first introduced, broke with the strong tradition of such local interactions and temporal unfolding.

This approach, now known under the name hypernetwork, is based on the idea of one neural network (the hypernetwork) encoding the parameters of a potentially much larger phenotype in one-shot, i.e. each component in the phenotype is determined independently of any other component. Whereas many indirect encoding approaches illustrate opportunities for utilizing biological principles but do not yet perform as well as the best direct approaches, such hypernetworks already perform better in many standard benchmarks. Initially tested on indirectly encoding images, which we will discuss in the next section, this approach can be extended to many other domains, such as 3D robot morphologies, and even to encode artificial neural networks themselves (Section 4.3.3).

#### 4.3.1 Compositional Pattern Producing Networks

The most common way to implement hypernetworks in neuroevolution is through compositional pattern-producing networks (CPPNs; Kenneth O Stanley 2007). Even though they are fundamentally distinct from developmental systems, CPPNs are inspired by developmental biology: Structures are built within a geometric space analogously to chemical gradients that define the axes of the embryo. For example, when the embryo of *Drosophila melanogaster* (one of developmental biologists' favorite pets and commonly known as the fruit fly) develops, chemical gradients establish axes from front to back, head to tail, and left to right. This way, structures such as the wings can be situated at their correct positions. Inside these structures and substructures, such as the intricate patterning of the wings which are placed within the local coordinate system of the wing itself. In our own bodies,

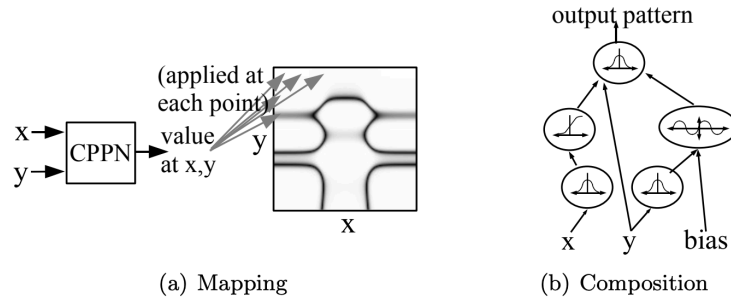


Figure 4.7: **CPPN Encoding.** (a) To create a 2D image, the CPPN is queried with the  $(x, y)$  location of a specific pixel and then outputs its grayscale value. (b) A CPPN is similar to a traditional neural network but instead of only one type of activation functions, they are chosen from a small set of activation functions to create patterns with particular regularities. Composing functions with each other allows the system to produce complex patterns. Figure from Kenneth O Stanley (2007).

such gradients help define the position of e.g. the legs, arms, and hands and within these structures substructures such as the fingers of the hands. It would be expensive to simulate the underlying process of the diffusion of morphogens, which is why CPPNs simplify this process into a network of function compositions represented as a graph. On a high-level, CPPNs are generative neural networks that create structures with regularities in one shot and without going through a period of unfolding/growth.

We will start by looking at how a CPPN can be used as an indirect encoding for image generation but later explore how it can be easily extended to other domains such as generating neural network connectivity patterns, morphologies of 3D soft robots, or agent environments. A CPPN generates an image by taking as input the coordinates of a 2D location  $\mathbf{p}=(x, y)$  and outputting the RGB color or grayscale value of the pixel at that location. By repeating this process for all the pixels of a two-dimensional grid, a two-dimensional image can be created (Figure 4.7a). In contrast to a direct encoding, in which each pixel in the image would be optimized separately, one advantage of the CPPN representation is images can be generated at any resolution, by only changing the resolution of locations we sample and without increasing the number of genotypic parameters of the CPPN itself.

As discussed earlier in this chapter, one common goal of indirect encodings is to be able to express patterns such as symmetry, repetition, etc. In order to allow CPPNs to more easily express such patterns, nodes in these networks do not all implement the same activation function as in traditional neural networks (including the networks traditionally evolved by NEAT) but are chosen from a small set of activation functions, such as Gaussian, sigmoid, and sine wave functions. For example, a Gaussian function can create something similar to a symmetric chemical gradient, while a sigmoid generates an asymmetric one, and a sine wave can create a repeating pattern. Things get more interesting when functions are composed with each other, which is in some way analogous to the morphogens creating local coordinate systems in real organisms, enabling their incredible levels of complexity. For example, a sine wave composed with the square of a variable  $\sin(x^2)$  produces a pattern that is repeating but with some type of variation. Such patterns are ubiquitous in

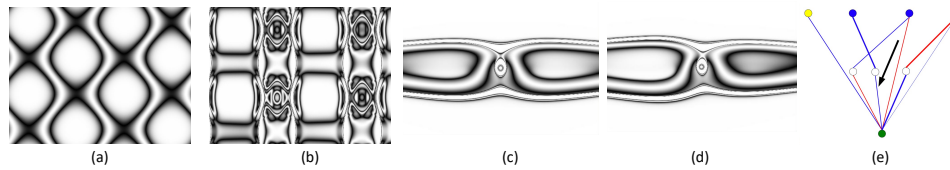


Figure 4.8: **CPPN Examples.** CPPNs can produce patterns with repetition (a) and repetition with variation (b). They can also create symmetric patterns such as the sun glasses shown in (c), which is encoded through the CPPn shown in (e). By changing only a single connection, varying degrees of symmetry can be produced, such as the morphed glasses in (d). Figure from Kenneth O Stanley (2007)

many patterns seen in nature. These networks can produce surprisingly complex structures with few nodes and connections, making them useful in a wide range of applications, as we'll see throughout this book. An example of such a CPPN with different activation functions is shown in Figure 4.7b, which creates the symmetric and repeating pattern shown in Figure 4.7a.

How can we evolve these CPPNs? Traditionally, CPPNs are evolved with NEAT which enables the optimization of both the weights and the network architecture. Additionally, NEAT enables CPPNs to slowly complexify and to produce more and more complex patterns. Augmenting NEAT to evolve CPPNs instead of the typical ANNs is straightforward. Every time a structural mutation is adding a node to the network, the activation function of that node is randomly chosen from a pre-defined set of activation functions, often with equal probability. However, it is certainly possible to also use a method like ES (Chapter 3) to optimize the weights of a fixed-topology network which includes randomly assigned activation functions for each node. We will leave this as an exercise for the reader.

[TODO: Add figure of CPPN images from original CPPN paper?; figure that shows elaboration on a concept; and a 3D example; not the same ones we show for Picbreeder later]

One way to explore the representational power of an encoding is through interactive evolutionary computation (IEC) (Takagi 2001). Instead of evolving towards a certain target, in interactive evolution, the user guides the evolutionary search by selecting parents from a set of candidate solutions (often by visually taking a look at them and deciding what they like most). The benefit of IEC is that it can reveal an encoding's ability to being able to encode a diversity of artefacts, while being able to establish and exploit regularities. We'll further discuss how this idea of interactive evolution allows human designers to drive evolutionary discovery, how it enables multiple humans to collaboratively evolve artefacts, and how it can even lay the foundation for new types of machine learning-based games in Chapter 8.

Exploring the space of CPPN-encoded images through IEC demonstrates that the representation is able to capture many of the desirable regularities we identified earlier in this chapter (Figure 4.8). For example, it is able to create patterns that show repetition (Figure 4.8a) but also repetition with variation (Figure 4.8b). Figure 4.8c illustrates a set of "sunglasses" that exhibit bilateral symmetry, meaning they are mirror images on either side. This symmetry serves as an example of how genetic elements can be effectively reused. In



this case, the CPPN-based function that defines one lens (the left one) are identically used for the other lens (the right one). Intriguingly, modifying just one connection gene, as shown in Figure 4.8e, can alter the symmetry of the lenses, resulting in a slight asymmetry while still preserving the overall pattern's coherence, as seen in Figure 4.8d. Even though the “genetic code” is the same for both sides, one lens displays a variant of the pattern seen in the other. This ability to evolve and refine specific features without disrupting the fundamental pattern is significant and possible because changes in the coordinate frame within a CPPN do not ruin the overall pattern being created. Therefore, even if the symmetry of the underlying coordinates is disrupted by a single gene alteration, the intricate pattern created within these coordinates remains intact and unaltered.

Additionally, one of the fundamental properties of natural evolution is that it is able to elaborate on discovered designs in subsequent generations. For example, the fundamental bilateral body plan, discovered early on during the Cambrian explosion, has undergone extensive development over hundreds of millions of years, yet its core structure has been consistently preserved. In a similar vein, the question arises: Can a CPPN effectively replicate a bilateral body plan and, over generations, both preserve and refine this bilateral symmetry. IEC experiments demonstrate that after discovering a spaceship-like design with bilateral symmetry (Figure 4.9a), that design can then be elaborated upon, with the underlying regularities becoming more complex in subsequent generations. Importantly, the basic parts that form the spaceship are conserved during this elaboration, such as its nose, tail, and wings. In the subsequent sections, we will see that this ability to elaborate on previous discoveries is an important property of CPPNs.

CPPNs are also not restricted to 2D and can easily be extended to generate 3D-forms instead of 2D-images by adding a third  $z$ -input and can even encode locomoting 3D soft robots, as we will see in the next section.

#### 4.3.2 Case Study: Evolving Virtual Creatures with CPPN-NEAT

A good test domain for different indirect encodings are evolved virtual creatures, which refer to digital entities that interact within a computational environment. These creatures are typically part of a simulation in which various forms of artificial life compete, survive, reproduce, and evolve over time based on certain predefined criteria or environmental pressures. In this section we will have a look at how the morphologies of such creatures can be defined through a CPPN. We will encounter virtual creatures again throughout the book, such as in the context of collective intelligence (Section 7.3.2) or co-evolving of morphology and control (Section 9.2).

Unlike the static CPPN-encoded images we have encountered in the previous section, virtual creatures often have to interact with their environment, requiring a form of embodied cognition. This dynamism challenges the encoding schemes to not only create viable forms but also to encode behaviors that are effective in a given environment. Virtual creatures, with their varied morphologies and behaviors, present a complex and diverse space to explore. This complexity makes them ideal for testing the capabilities of indirect encodings to generate a wide range of solutions, where there is a coherent link between form and function.

The particular virtual creatures we are looking at next are three-dimensional soft robots (Cheney et al. 2014). Each robot is made out of an arrangement of voxels, where each voxel

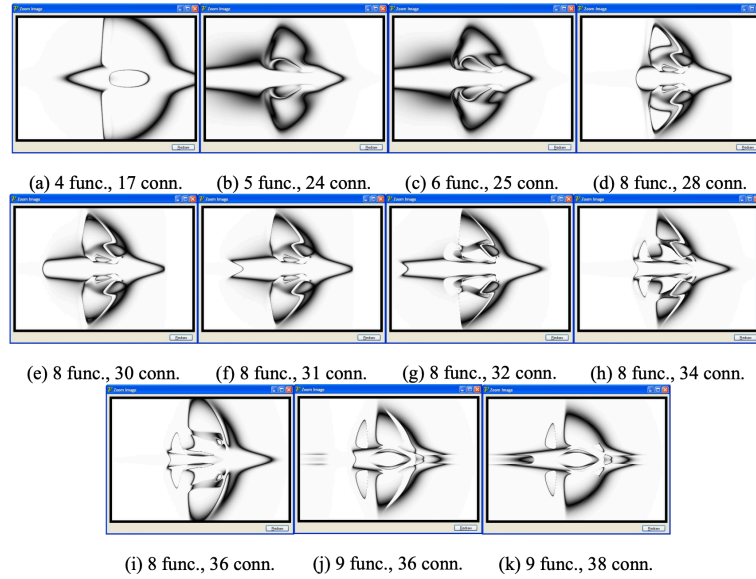


Figure 4.9: **CPPN pattern elaboration over generations.** The figure shows a chronological sequence of CPPN-encoded designs, discovered and elaborated upon during interactive evolution. Together with the designs, the number of hidden node functions and connections are also shown. Figure from Kenneth O Stanley (2007).

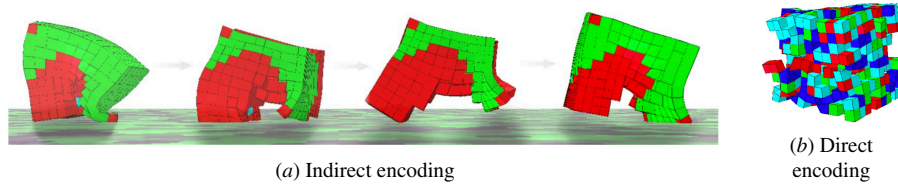


Figure 4.10: 3D soft robot generated with the indirect CPPN encoding (a) and a direct encoding (b) in which each voxel is optimized independently. In contrast to the direct encoding, the CPPN-based encoding is able to produce 3D structures with symmetries and repeating motifs. **Figure from (Cheney et al. 2014)**

can be one of four materials, displayed as different colors (Figure 4.10). Voxels colored green undergo periodic volumetric actuations at 20% intervals. Voxels colored light blue are passive and soft, with no inherent actuation; they deform only in response to the actions of nearby voxels. Red voxels behave like green ones but with counter-phase actuations. The dark blue voxels are also passive, but they are more rigid and resistant to deformation than their light blue counterparts. These soft robots do not have sensors and the patterns of material types thus fully determines the robot's actuation pattern. This means that the optimisation task equals finding a pattern of materials that makes the robot move as fast as possible.

The robot-generating CPPNs take as input the  $x$ ,  $y$ , and  $z$  coordinates and distance from center ( $d$ ) of each voxel. One of the network's outputs indicates the presence of material,

while the other four outputs, each representing the specific material mentioned above, output the maximum value indicating the type of material present at that voxel. Separating the phenotypic component's presence and its parameters into distinct CPPN outputs has been demonstrated to enhance performance. If there are several disconnected patches, only the central patch is considered in creating the robot morphology.

Optimizing these CPPN representations with NEAT shows that they are indeed not restricted to generating static structures but can produce fully functional three-dimensional soft robots. An example of such an evolved robot locomoting is shown in Figure 4.10a. This robot morphology, together with other morphologies discovered during evolution, display interesting regularities often including symmetry and repetition. This is in stark contrast to robots that use a direct encoding, in which the parameters of each voxel are encoded individually. These robots often fail to perform well without any clear regularities in their morphologies (Figure 4.10b). A direct encoding makes it more challenging to find structures that display the globally coordinated behaviors necessary for efficient locomotion strategies.

CPPNs can generate structures with regularities by giving the network access to the locations of each element of the structure to be generated. In biological systems, this information is not directly available so it is an interesting question if we are also able to generate complex patterns artificially solely based on the local communication of the structure's components. We'll return to this question in Section 7.3 on neuroevolutionary approaches for collective intelligence, where we will also again encounter three-dimensional soft robots.

#### 4.3.3 Hypercube-based NEAT (HyperNEAT)

We started this chapter with a discussion of the intricate structure of the human brain and its complex regularities. For example, in the brain, we often find neural modules with repeating connectivity patterns and left/right symmetry. Given a CPPN's ability to express complex 2D and 3D patterns, it makes sense to also consider if they could be used to generate such complex neural *connectivity patterns* as well. With this goal in mind, the question becomes what should such a CPPN look like and what should its input be.

To answer this question, we again consider convolutional connectivity patterns. In a convolutional neural network, one of the fundamental building blocks in deep neural networks, the same feature detector is employed at multiple locations in a network. If we want our algorithm to discover such heuristics by itself, we need a method that can learn that there should be correlations between the weights of nearby neurons. Essentially, this involves generating weight patterns based on the geometry of the input and output domains. For instance, if the input and output domains are both two-dimensional, the weight of a connection between two neurons can be expressed as a function  $f$  of the positions  $(x1, y1)$  and  $(x2, y2)$  of the source and target neurons respectively.

This is the fundamental insight behind the method called hypercube-based NEAT (hypercube-based NEAT) (Stanley, D'Ambrosio, and Gauci 2009), which can be viewed as one of the most foundational and impactful applications of CPPNs. In essence, in HyperNEAT every neuron is given a role (e.g. input, hidden, output) and a location in space (traditionally by a user but this process can also be automated, as we will see in the next section). The collection of roles and positions in HyperNEAT is often referred to as the *substrate*, to distinguish it from the CPPN itself. The connectivity patterns between the neurons

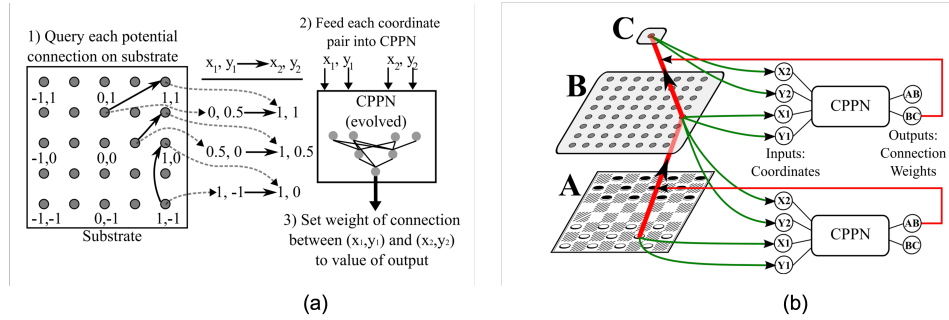


Figure 4.11: HyperNEAT Substrates. Two different types of HyperNEAT substrates are shown, which are the arrangement of nodes and their roles. In (a), nodes are arranged on a 2D plane. The CPPN is queried with all pairs of nodes to determine how they are connected to each other. A more complex substrate for evaluated checker board game position is shown in (b). The input layer reflects the geometry of the board. The output layer C has one node that determines the quality of a board state. The CPPN has two outputs AB and BC. To query a connection from layer A to B, output AB is used while for layer B to output layer C, it uses output BC. Figure from (Gauci, Stanley, et al. 2008)

are determined by CPPNs evolved through NEAT, which take as input the location of two nodes. Querying the CPPN with every possible connection between two points, with the output of the CPPN representing the weight of the connection, produces an artificial neural network. This process is visualized in Figure 4.11a. To not only produce fully connected networks, connections might only be expressed if the CPPN output is higher than a certain threshold. In other HyperNEAT variants, a second output determines if a connection should be expressed (Verbancsics and Stanley 2011). This approach can be helpful because it decouples the pattern of weights from the pattern of expressed connections

Given neurons positions in space, allows us to create a variety of regular connectivity patterns. For example, in a typical convolutional network, a filter is applied across the geometry of the input space. HyperNEAT can invent the concept of convolution by itself, because it can be expressed as a function based on the distance of the source to the target neuron:  $x_1 - x_2$  and  $y_1 - y_2$ . The intriguing aspect of HyperNEAT lies in its ability to go beyond conventional convolution as the sole significant pattern of connectivity. Through HyperNEAT, evolved neural networks have the potential to uncover and leverage various patterns of regularity, inaccessible to traditional learning algorithms for neural networks.

For example, consider the task of creating a neural network that evaluates board positions in the game of checkers, that is, a specific board configuration is given to a neural network as input and it has to determine how good this position is. This game is intuitively geometric, with the movement rules for each piece being the same for every location on the board. The HyperNEAT approach should be able to take advantage of the CPPN's ability to calculate the connection weights based on the positional differences between two nodes enables it to uniformly apply a repeating concept throughout the entire board. In a sense, HyperNEAT is able to see the geometry of the task. We thus expect that an indirect representation that can learn to repeat strategies across the board should have an advantage when compared to

a direct encoding like NEAT, which has to learn this pattern for each square on the board separately. In the adaptation of HyperNEAT for the game of checkers, the input layer is designed as a two-dimensional structure, mirroring the checkerboard's layout, as illustrated in Figure 4.11b (Gauci and Stanley 2010). This substrate has one input *A* and one hidden layer *B* and a single output node *C*, which role it is to output is an evaluation of the board position. Note that the CPPN here has two outputs AB and BC. Therefore, the *x* and *y* coordinates of each node are adequate to pinpoint the specific connection being queried, with the two separate outputs differentiating the connections between A&B and B&C from each other.

HyperNEAT is able to find a high-performing board evaluator significantly faster than NEAT, which is in part due to HyperNEAT's ability to search through a smaller genotypic space. Additionally, when comparing the most general solutions found by both approaches to randomized opponents, HyperNEAT shows a significantly higher win rate and also loses significantly fewer games than NEAT solutions. These improved generalization abilities are due to HyperNEAT ability to discover the necessary regularities in the geometry of the game. This observation is supported by examinations of the connectivity patterns of the most general HyperNEAT solutions, which are often smoother and more continuous than less general solutions.

Beyond board games, we hypothesized at the beginning of this chapter that indirect encodings should also be useful for tasks such as controlling a quadruped robot (Figure 4.12a), taking advantage of the task's symmetry and regularities. For HyperNEAT, the positions of sensor and motor neurons within a quadruped body can be exploited to efficiently develop consistent gait patterns that rely on connectivity patterns unrelated to convolution (Clune, Stanley, et al. 2011). Each leg can be viewed as a repeated module, with different gaits having different regularities themselves. For example, in a typical horse trot gait, the diagonal pairs of legs move forward at the same time while in other gaits such as the pace gait, the two legs on the same side move forward at the same time. The HyperNEAT substrate, which features three 2D sheets for the inputs, hidden layer, and output layer is shown in Figure 4.12b. Input on the substrate are arranged to reflect the geometry of the task, with each row receiving information about the state of a single leg (e.g. the current angle of the three joints of the leg, a sensor that indicates if the leg is touching the ground). The output substrate also reflects the morphology of the robot, with the three elements in each row outputting the desired new joint angle.

It is interesting to look at the performance of indirect vs. direct encodings across the continuum of regularity. For example, in the quadruped domain, the regularity of the problem can be decreased by introducing faulty joints, in which noise is added to the requested joint angle and the actual motor command that is sent. As expected, HyperNEAT's performance increases with increased task regularity, outperforming all other approaches (NEAT and FT-NEAT, which is a variant of NEAT that has a fixed number of hidden nodes, which is the same as the number used in the HyperNEAT substrate) with no or 1 faulty joint. When problem regularity is sufficiently low (8 and 12 faulty joint treatment), FT-NEAT and NEAT start to outperform HyperNEAT. The important lesson here is that the type of method to be used, highly depends on the target domain and how much regularities there are to exploit.

Interestingly, going beyond pure quantitative results, the gaits produced by HyperNEAT are also often more regular and coordinated than the ones from NEAT. HyperNEAT often

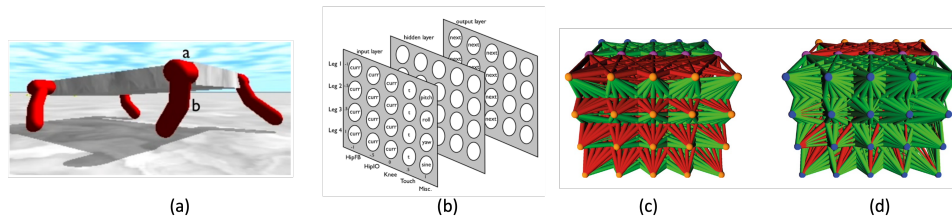


Figure 4.12: A neural Network controller for a quadruped robot produced by HyperNEAT. The HyperNEAT substrate has three layers: input, hidden, and output. The input and output nodes are arranged in a way to take the task geometry into account. A front view of the network is shown to the left, and a view from the back on the right. Input nodes are shown in yellow, and output nodes in blue. Line thickness represents the magnitude of the weight. HyperNEAT is able to create connectivity patterns with varying degrees of regularity, based off the location of the nodes in space. Figure from (Clune, Stanley, et al. 2011).

produces two types of gaits. In one of them, all legs move forward in unison at the same time, which suggests that HyperNEAT repeated the same connectivity pattern for each leg. The other gait resembles more of a horse gallop gait, in which three legs move together and one of the legs moving in opposite phase. This gait indicates that HyperNEAT can also produces regularities with variation (i.e. one leg moves different to the other three legs). These regularities are also reflected in the HyperNEAT-produced weight patterns. Figure 4.12c and d shows the view of the same network from the front and from the back, respectively. Observe the intricate and consistent geometric patterns of weight distribution, such as the inhibitory connections from input nodes directed towards the upper hidden nodes and excitatory connections aimed at the lower hidden nodes. Additionally, there is a notable regularity with variations, exemplified by the spread of inhibitory connections into the output nodes, which changes along both the  $x$  and  $y$  axes.

In summary, an indirect encoding such HyperNEAT can offer great benefits, allowing relatively compact CPPNs with only a handful of connections to encode functional neural networks with millions of weights. In fact, even before DeepMind demonstrates that it is possible to learn to play Atari games from pixels, which has been a significant milestones in their early successes and shaping the landscape of deep RL, HyperNEAT was the first method used to train neural networks to play Atari games from pixels alone (Hausknecht et al. 2014). However, HyperNEAT is also not panacea for every task; it does perform best in domains where regularities can be exploited but works less well in domains with many irregularities. There have been attempts at combining the best properties of both direct and indirect encodings. One such method is Hybridized Indirect and Direct encoding (HybriD), which discovers the regularities of the domain with an indirect encoding but then accounts for the irregularities through a fine-tuning phase that optimizes these weight parameters directly (Clune, Beckmann, et al. 2011). Another more biologically plausible solutions is a combination of an indirect encoding together with lifetime learning. While indirect encodings are useful to create regular neural structures, these provide a good starting point for local learning rules such as the Hebbian rules we first encountered in Section 4.2.3. And

indeed, neuroevolutionary experiments show that neural connectivity motifs that are indirectly encoded and thus more regular do demonstrate the best learning abilities in a simple operant conditioning task (Tonelli and Mouret 2013), when compared to directly encoding those starting weights.

This strong relationship between indirect representations and synaptic plasticity underscores a crucial interplay between development and adaptability in biological systems. Synaptic plasticity interacts closely with the structured neural connectivity formed during development. This interplay allows for both the initial formation of efficient networks and their subsequent adaptation to new information and experiences. In biological systems, such connectivity patterns are not only shaped by genetic encoding but are also dynamically refined through experience-dependent plasticity. Understanding this connection could significantly impact the types of representations that will define the next generation of indirect encodings. However, despite its potential implications for developing more adaptable neural networks, this interplay between indirect encoding and synaptic plasticity has yet to receive substantial attention from the broader neuroevolutionary research community.

#### 4.3.4 Evolvable Substrate HyperNEAT

While HyperNEAT showed that NE can benefit from neurons that exist at locations in space, one drawback of its original formulation is that the location and number of hidden nodes have to be defined by the user. While it is often clear how the location of the inputs relate to the output units and thus where they should be placed within the substrate (e.g. the rangefinders of a robot should relate to the network's outputs that control its movement), how to decide on the position of the hidden nodes is less straightforward. A less obvious effect is that requiring a hidden node  $n$  to be at position  $(a, b)$ , as specified in the original HyperNEAT, inadvertently demands that any weight pattern created by the CPPN must intersect exactly at position  $(a, b)$  with the appropriate weights. This means the CPPN in HyperNEAT has to align the correct weights precisely across all coordinates  $(a, b, x_2, y_2)$  and  $(x_1, y_1, a, b)$ . However, this raises the question: why enforce such a random constraint on weight locations? The CPPN might more easily represent the desired pattern slightly off the specified location, but this would not work with the constraints set by the user.

These limitations are addressed by an extension of HyperNEAT, called Evolvable Substrate HyperNEAT (ES-HyperNEAT) (Risi and Stanley 2012b). The basic idea behind ES-HyperNEAT is that the weight pattern generated by the CPPN should give some indication of where the hidden nodes should be placed and how many there should be. That is, areas in the 4D hypercube that contain a lot of information, should result in more points being chosen from these areas. Remember, each point in that 4-dimensional weight space is a connection in two dimensions.

For example, take a hypercube whose weights are all uniform, meaning that  $CPPN(x_1, y_1, x_2, y_2) = k$  for all different input combinations; it would not make much sense to express many connections if there is not much information in the underlying weight pattern. On the other hand, if the variance of the weight pattern is high in some regions, it might indicate that there is more information available and thus more connections should be expressed. In ES-HyperNEAT, if a connection is chosen to be expressed, the nodes that it connects must therefore also be expressed. Which nodes to include thus becomes

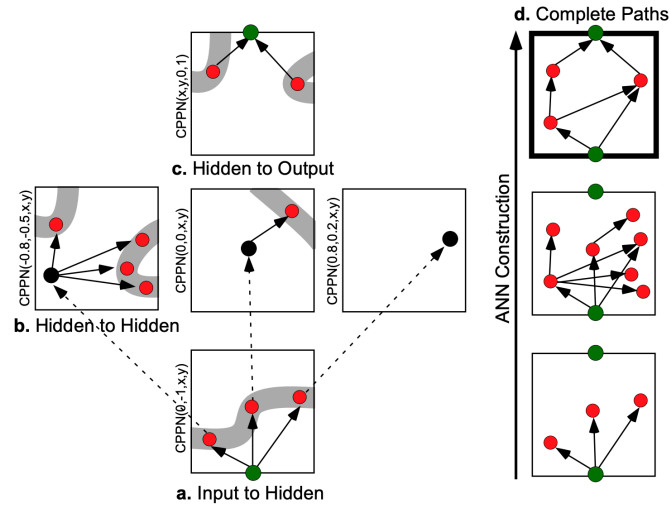


Figure 4.13: Evolvable-Substrate HyperNEAT. (a) Starting from the input nodes, ES-HyperNEAT analyses sequences of 2D slices through the hypercube weight pattern to discover areas of high variance. This information is then used to determine which connections, and thereby nodes should be expressed. The approach continues from the discovered hidden nodes (b) until some maximum depth has been reached. (c) Similarly we start from the output nodes, so determine to which hidden nodes they should be connected. (d) Once the approach has run a maximum number of iterations or when no new nodes are discovered, the resulting ANN is pruned, removing any nodes that do not connect both to the inputs and outputs of the network. Figure from (Risi and Stanley 2012b).

implicit in the question which connections to include from the infinite set of potential connections encoded by the CPPN. By making the number and location of nodes depending on the CPPN-generated pattern, we give the system a “language”, i.e. a way to increase or decrease the number of connections (and thus nodes) and change their location by varying the underlying pattern.

For this approach to work, it is useful to have a datastructure that can represent space at variable levels of granularity. One such datastructure is the quadtree, which has found successful applications in various fields, including pattern recognition and image encoding, and partitions a two-dimensional space by recursively subdividing it into four quadrants or regions. This process creates a subtree representation, where each decomposed region becomes a descendant with the original region as the parent. The recursive splitting continues until the desired resolution is achieved or until further subdivision becomes unnecessary, indicating that additional resolution would not reveal new information.

ES-HyperNEAT works as follows: For each input neuron at position  $(p1, p2)$  apply the quadtree to analyse regions for their variance of the 2-dimensional sub-slice through the hypercube spanned by  $\text{CPPN}(p1, p2, x2, y2)$  (Figure 4.13). In areas of high variance, as detected by the quadtree algorithm, connections and their corresponding nodes are created. The process is then repeated from those discovered hidden nodes until some maximum depth is reached, after which only the neurons are kept that have a path to an input and



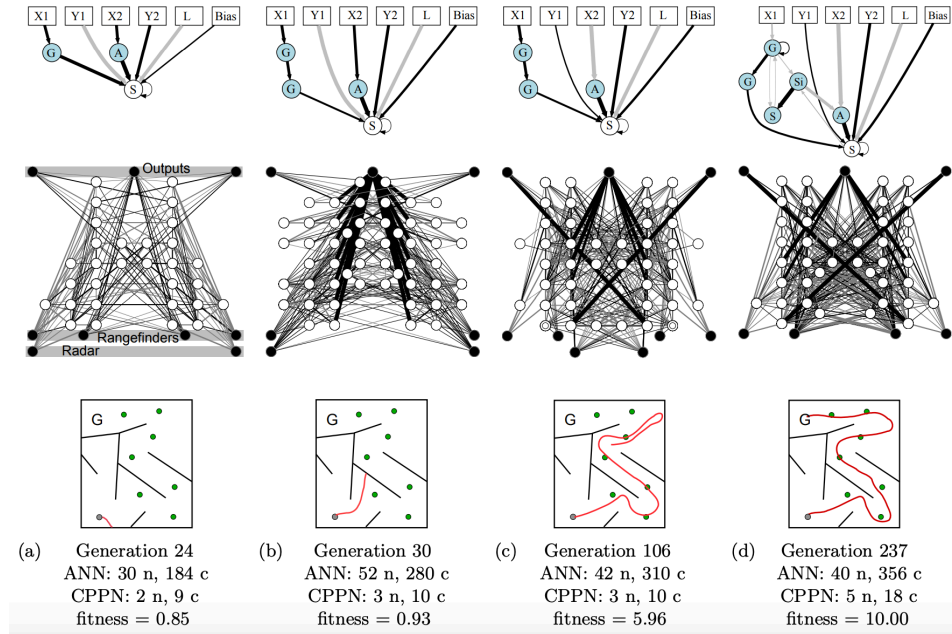


Figure 4.14: **ES-HyperNEAT Example Lineage** Shown are four milestones in one of the maze solution lineages. The CPPN is shown at the top with the decoded neural network in the middle (CPPN activation functions are G=Gaussian, A=absolute value, S=sigmoid, Si=sine). In addition to the location of nodes, the CPPN also receives the length  $L$  of a connection as an additional input. The resulting maze navigation behaviour is shown at the bottom, together with the number of connections and nodes in the neural network and in the CPPN. One can observe a gradual increase in the complexity of the CPPN which increases the information in the underlying hypercube pattern and thus results in an increase in the number of ANN weights and neurons.

output neuron. After this process is repeated for each input (and output) node, the ANN is constructed and can be applied to the task at hand.

A good domain to evaluate this approach should test its ability to build and elaborate on previously discovered stepping stones. While it is easy to see how a method such as NEAT would be able to accomplish this task, it is less obvious how an indirect encoding would fare. For example, the original HyperNEAT has the tendency to often produce fully connected, which makes it harder to elaborate on intermediate milestones since all connections are already used for the current partial solutions. On the other hand, ES-HyperNEAT should be able to do so because it can increase the number of nodes and connections in the substrate.

One such task is the hard maze domain, originally introduced to study more explorative search methods such as Novelty Search (Section 5.3). Here, the agent has rangefinder sensors to detect walls and a pie-slice radar sensors that fire when the goal is within the agent's corresponding pie-slice sensor (Figure 4.14). To encourage the agent to discover the intermediate stepping stones, the original task is modified to specifically reward the agent for traversing the green way points (which are not visible to the agent).

As hypothesized, the original HyperNEAT indeed struggles with this task, and only finds solutions in 45% of 20 independent evolutionary runs. ES-HyperNEAT on the other hand, is able to find a solution in 95% of all runs. As shown in Figure 4.14, analysis of an example lineage shows that ES-HyperNEAT is able to elaborate on previously discovered stepping stones. This figure shows four milestone ANNs (middle row), together with the underlying CPPN (top) and the resulting agent trajectory (bottom). Interestingly, all the ANNs display common geometrical features which are kept during evolution, such as the symmetric network topology. While larger changes happen earlier in evolution, the networks from generations 106 and 237 show a clear holistic resemblance to each other, with strong connections to the three output neurons. These results also demonstrate that ES-HyperNEAT is able to encode a larger network with a compact CPPN. In fact, the solution ANN with 40 hidden nodes and 256 connections is encoded by a CPPN with only 5 nodes and 18 connections.

#### 4.3.5 General Hypernetworks and Dynamic Indirect Encodings

More generally, HyperNEAT and its variations are particular examples of a family of algorithms now called hypernetworks (Ha, Dai, and Le 2016). Hypernetworks generalize HyperNEAT to any approach in which one network (termed the hypernetwork) generates the weights of another target neural network. The hypernetwork is typically a smaller network designed to learn a mapping from a low-dimensional input space to the high-dimensional weight space of the target network. The target network is the actual network that performs the main task, such as classification, regression, or controlling an agent. Pioneering work on hypernetworks goes back to the early 90s, where Schmidhuber (1992) introduced the idea of Fast Weight Programmers, where a "slow" neural network trained through gradient descent learned the "fast" weights of another network.

Mathematically, given an input  $x$  to the target network, a hypernetwork  $H$  takes an auxiliary input  $z$  and outputs the weights  $\theta_{TN}$  for the target network. This relationship is expressed as  $\theta_{TN} = H(z)$ . The target network then uses these weights to perform its task, represented as  $y = T(x; \theta_{TN})$ , where  $x$  is the input to the target network,  $z$  is the auxiliary input to the hypernetwork,  $\theta_{TN}$  are the weights generated by the hypernetwork, and  $y$  is the output of the target network.

In the previous section on HyperNEAT, we have seen a special case of such a hypernetwork, that was geometrically-aware, i.e. the auxiliary input  $z$  gave nodes locations in space, and which was trained through NEAT. Other approaches, such as Compressed Network Search (Koutnik, Gomez, and Schmidhuber 2010) do not employ CPPN-NEAT but instead use discrete cosine transformations (DCS) to compress the weights of a larger weight matrix into a smaller number of DCS coefficients, resembling the popular JPEG compression. It is also possible to combine evolving the neural architecture with gradient-based weight training, which was demonstrated in an approach called Differentiable Pattern Producing Networks (DPPNs) (Fernando et al. 2016).

More recently, Ha, Dai, and Le 2016 showed that the phenotype can be made directly dependent on the network's input, by training a hypernetwork end-to-end through a gradient-descent-based training approach. This work strikes a balance between the Compressed Network Search approach, where a DCS prior limits the type of weight matrixes that can be produced, and the more flexible HyperNEAT approach which requires evolving

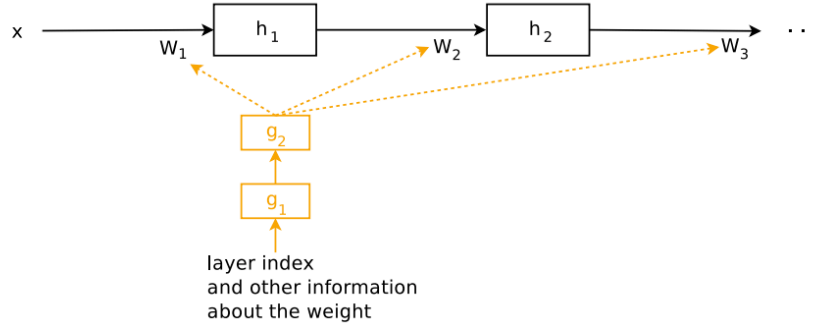
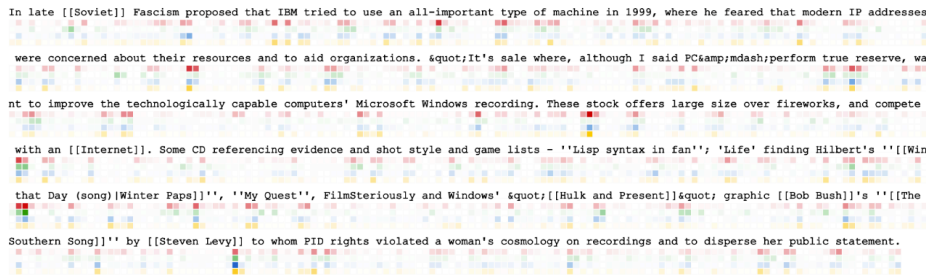


Figure 4.15: **Static Hypernetwork.** In this example, the hypernetwork (shown in orange) generates the weights of each layer of the main network (shown in black) by conditioning the network on layer embeddings. Figure from Ha, Dai, and Le 2016.

both the architecture and weights (which adds significant complexity for many practical problems). In Ha et al.'s approach, a hypernetwork generates the weights of a feedforward network one layer at a time by conditioning the hypernetwork on the specific layer embedding (Figure 4.15). These layer embeddings can either be fixed or they can also be learned, allowing the system itself to learn approximate weight sharing within and across layers. This approach was able to produce the weights for deep convolutional network for CIFAR-10 classification, with only a small decrease in classification accuracy but a drastic reduction in the number of trainable model parameters. Interestingly, when applying the hypernetwork approach to create the weights for a target network that is fully-connected, it is able to learn convolutional-like filters when the location of the target weight and the  $x, y$  location of each input pixel is provided.

Importantly, hypernetworks offer the intriguing ability to serve as a *dynamic indirect encoding*, in which the produced weight pattern is allowed to change over time and made dependent on the inputs for the task at hand. For example, a hypernetwork can be trained to produce the weights of an RNN target network for handwriting sequence generation, which can change over time and is dependent on the agent's internal state and the inputs (the previous output of the RNN) (Figure 4.16). In other words, a hypernetwork takes a low-dimensional representation of the input character and the hidden state of the RNN as inputs and outputs the weights for the next prediction step. This approach allows the RNN to dynamically adapt its parameters based on the current context and is a good demonstration of how concepts from neuroevolution are being effectively combined with those from the traditional machine learning field.

In summary, hypernetwork-like approaches can significantly reduce the number of trainable parameters while still perform well across different domains. This type of dynamic indirect encoding is also closely related to the idea of neural self-attention, which we will have a look at in the next section and which has been the basis for many recent deep learning revolutions such as the transformer architecture. Here larger input-dependent weight matrices are created through the outer product of two smaller matrices called keys and values. As we will see, this type of indirect encoding allows encoding Matrix  $\mathbf{A}$ , of size  $O(n^2)$  using only  $O(d)$  number of genotype parameters.



In late [[Soviet]] Fascism proposed that IBM tried to use an all-important type of machine in 1999, where he feared that modern IP addresses were concerned about their resources and to aid organizations. &quot;It's sale where, although I said PC&mdash;perform true reserve, wa nt to improve the technologically capable computers' Microsoft Windows recording. These stock offers large size over fireworks, and compete with an [[Internet]]. Some CD referencing evidence and shot style and game lists - ''Lisp syntax in fan''; 'Life' finding Hilbert's ''[[Win that Day (song)|Winter Paps]]'', 'My Quest'', FilmSteriously and Windows' &quot;[[Hulk and Present]]&quot; graphic [[Bob Bush]]'s ''[[The Southern Song]]'' by [[Steven Levyl]] to whom PID rights violated a woman's cosmology on recordings and to disperse her public statement.

Figure 4.16: **Application of dynamic hypernetworks for handwriting sequence generation.** In the dynamic indirect encoding approach, the hypernetwork takes as input the internal state of the neural network and its previous action to dynamically generate the weights of the RNN target network (shown as four different colours) Figure from Ha, Dai, and Le 2016.

#### 4.4 Self-Attention As Dynamic Indirect Encoding

In the preceding section, we explored the concept of hypernetworks, illustrating their role as indirect encoding methods where one neural network, the hypernetwork, generates the weights for another network, termed the target network. Typically, hypernetworks generate these weights without directly considering the specific input  $x$  to the target network. Transitioning from this, we introduce the concept of self-attention mechanisms, which embody a sophisticated method of dynamically generating contextual relationships within data. Unlike hypernetworks, self-attention mechanisms inherently account for the input  $x$  during the processing phase, tailoring the computational focus in a data-driven manner. This capability not only allows self-attention to act as a form of indirect encoding but also enhances it to be a dynamic encoding process. The dynamic nature arises from its ability to adjust the internal model representations in response to the particularities of the input data at any given moment, thereby offering a more flexible and context-aware approach to encoding information.

##### 4.4.1 Background on Self-Attention

The attention mechanism (Vaswani et al. 2017), a groundbreaking innovation in the field of neural networks, particularly in natural language processing, has revolutionized how models handle and interpret sequential data like text and time series. At its core, attention allows a model to focus on different parts of the input sequence when producing each part of the output sequence, mimicking the human cognitive process of focusing more on certain aspects while perceiving or processing information. The introduction of attention mechanisms, especially in architectures like BERT (Devlin et al. 2018) and GPT (Brown et al. 2020; OpenAI 2023) models, has led to substantial improvements in various complex tasks in language understanding and generation.

While modern attention mechanisms can adopt various configurations, including positional encoding and scaling, its fundamental concept can be described by the following

equations:

$$A = \text{softmax}\left(\frac{1}{\sqrt{d}}(X_q W_q)(X_k W_k)^T\right)$$

$$Y = A \times (X_v W_v)$$

where  $W_q, W_k, W_v \in \mathbb{R}^{d_{in} \times d}$  are the matrices that map the input matrix  $X \in \mathbb{R}^{n \times d_{in}}$  to components called *Query*, *Key* and *Value* (i.e.,  $\text{Query} = X_q W_q$ ,  $\text{Key} = X_k W_k$ ,  $\text{Value} = X_v W_v$ ). Since the average value of the dot product grows with the vector's dimension, each entry in the Query and the Key matrices can be disproportionally too large if  $d$  is large. To counter this, the factor  $\frac{1}{\sqrt{d}}$  is used to normalize the inputs. The attention matrix  $A \in \mathbb{R}^{n \times n}$  is obtained by applying a non-linear activation function, typically a softmax operation, to each row of the matrix. This mechanism is referred to as *self-attention* when  $X_q = X_k$ ; otherwise it is known as *cross-attention*.

#### 4.4.2 Self-Attention as a Form of Indirect Encoding

As we described previously, indirect encoding methods represent the weights of a neural network, the *phenotype*, with a smaller set of *genotype* parameters. How a genotype encodes a larger solution space is defined by the indirect encoding algorithm. As we have seen, HyperNEAT encodes the weights of a large network via a coordinated-based CPPN-NEAT, while Compressed Network Search (Koutnik et al. 2013) uses discrete cosine transform (DCT) to compress the weights of a large weight matrix into a small number of DCT coefficients, similar to JPEG compression. Due to compression, the space of possible weights an indirect encoding scheme can produce is only a small subspace of all possible combination of weights. The constraint on the solution space resulting from indirect encoding enforces an inductive bias into the phenotype. While this bias determines the types of tasks that the network is naturally suited at doing, it also restricts the network to a subset of all possible tasks that an unconstrained phenotype can (in theory) perform.

Similarly, self-attention enforces a structure on the attention weight matrix  $A$  that makes it also input-dependent. If we remove the Query and the Key transformation matrices, the outer product  $X_q X_k^T$  defines an association matrix where the elements are large when two distinct input terms are in agreement. This type of structure forced in  $A$  has been shown to be suited for associative tasks where the downstream agent has to learn the relationship between unrelated items. For example they are used in the Hebbian learning (Hebb 2005) rule, inspired by neurons that fire together wire together, which is shown to be useful for associative learning (Ba et al. 2016; Miconi, Stanley, and Clune 2018).

Because the outer product  $X_q X_k^T$  has no free parameters, the corresponding matrix  $A$  will not be suitable for arbitrary tasks beyond association. The role of the small Query and Key transformation matrices (i.e.,  $W_q$  and  $W_k$ ) allow  $A$  to be modified for the task at hand.  $W_q$  and  $W_k$  can therefore be viewed as the *genotype* of this indirect encoding method.  $W_q, W_k \in \mathbb{R}^{d_{in} \times d}$  are the matrices that contain the free parameters and  $d_{in}$  is a constant depending on the inputs. The number of free parameters in self-attention is therefore in the order of  $O(d)$ , While the number of parameters in  $A$  is in the order of  $O(n^2)$ . This form of indirect encoding allows us to represent the phenotype with a much smaller set of trainable

genotype parameters. Additionally, this type of indirect encoding dynamically adapts to various inputs.

Building on the concepts discussed in the previous section, we formulated the output of a hypernetwork  $H$  as  $\theta_{TN} = H(z)$  where  $\theta_{TN}$  are the parameters for a target network (TN) and  $z$  is an auxiliary input (e.g., layer index). In a similar vein, self-attention can be conceptualized as  $\theta_{TN} = SA(x)$  where  $x$  is the target network’s input. This adaptation allows for a more flexible and responsive model configuration, tailored to specific input characteristics and demands.

Furthermore, the aforementioned dynamic adaptation mechanism in self-attention, which allows real-time modulation of connection strengths based on input, also echoes the concept of “Fast Weights” (Schmidhuber 1992) where the idea of rapidly adaptable weights that could temporarily store information over short sequences was introduced. Similarly, self-attention leverages dynamic encoding to adjust the attention matrix  $A$ , effectively using  $W_q$  and  $W_k$  to reshape the network’s responses based on the input characteristics. This adaptability is critical for tasks where the relevance of specific input features varies markedly across contexts, akin to how fast weights facilitate short-term synaptic plasticity for rapid learning adaptation.

#### 4.4.3 Self-attention Based Agents

AttentionAgent (Tang, Nguyen, and Ha 2020), inspired by concepts related to inattentive blindness – when the brain is involved in effort-demanding tasks, it assigns most of its attention capacity only to task relevant elements and is temporarily blind to other signals, is one of such adaptation that utilizes an attention-based agent for video game play, offering enhanced interpretability in pixel-space reasoning, as illustrated in Figure 4.17.

This approach is grounded in self-attention (specifically,  $X_k = X_q$ ), with cropped game screen image patches serving as inputs. Key modifications to the attention mechanism in AttentionAgent include: (1) condensing the attention matrix into an importance vector, and (2) omitting the *Value* component in favor of extracting the top- $k$  ( $k = 10$  in the paper) most significant patch features as the output  $Y$ . This extraction is achieved through sorting and pruning, detailed in Figure 4.18 and the paragraphs below.

Concretely speaking, given an input game screen, AttentionAgent segments the input image into small square patches in a fashion similar to how a 2D convolution layer works. It then flattens these patches and treats the output of shape  $N \times CM^2$  as the input  $X \in \mathbb{R}^{n \times d_{in}}$  (see Figure 4.18 left). Here  $N$  is the number of patches,  $C$  is the number of channels in the image and  $M$  is the length/width of each patch, therefore  $n = N$  and  $d_{in} = CM^2$ .

Upon receiving this transformed data, the self-attention module follows the equations we mentioned above to get the attention matrix  $A$  of shape  $(N, N)$ . After softmax, each row in  $A$  sums to one, so the attention matrix can be viewed as the results from a voting mechanism between the patches. If each patch can distribute fractions of a total of 1 vote to other patches (including itself), row  $i$  thus shows how patch  $i$  has voted and column  $j$  gives the votes that patch  $j$  acquired from others. In this interpretation, entry  $(i, j)$  in  $A$  is regarded as how important patch  $j$  is from patch  $i$ ’s perspective. Taking sums along the columns of  $A$  results in a vector that summarizes the total votes acquired by each patch, and this vector is called the patch importance vector (see Figure 4.18 middle). Unlike the self-attention we

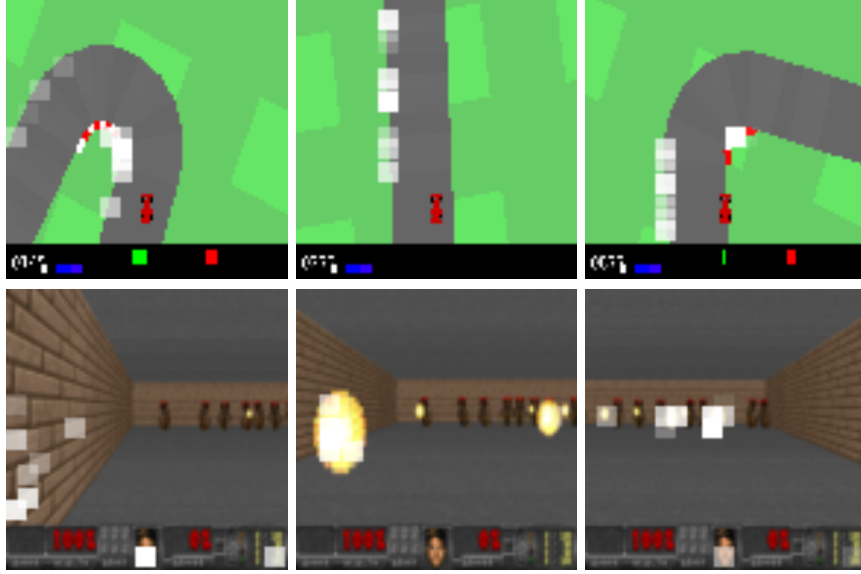


Figure 4.17: Demonstrating indirect encoding in AttentionAgent for enhanced interpretability. White patches on the game screens signify the agent’s focus areas, with their opacity indicating the relative importance of each patch. Figure from (Tang, Nguyen, and Ha 2020).

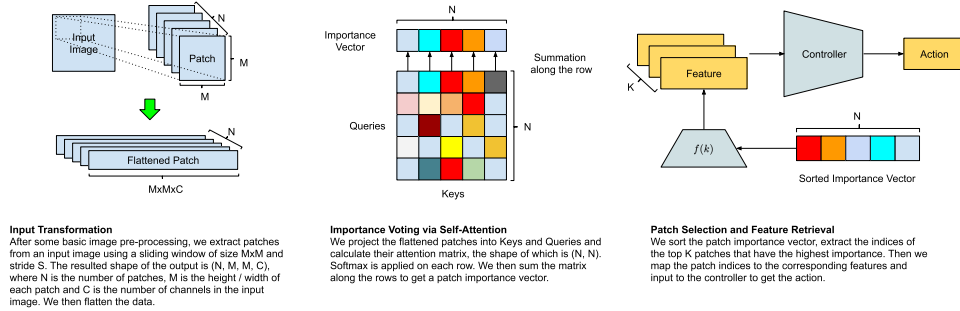


Figure 4.18: Method overview of AttentionAgent. Key modifications to the attention mechanism include (1) condensing the attention matrix into an important vector, and (2) omitting the *value* component in favor of extracting the top- $k$  most significant patch features as the output  $Y$ . Figure from (Tang, Nguyen, and Ha 2020).

introduced earlier, AttentionAgent relies solely on the patch importance vector and does not utilize the Value component of self-attention.

Finally, based on the patch importance vector, AttentionAgent picks the  $K$  patches with the highest importance and throws away the rest. It passes the indices of these  $K$  patches into a feature retrieval function, which returns the features extracted from the corresponding patches. These features are then fed into a neural network based controller to output the appropriate actions the agent should take (see Figure 4.18 right). By discarding patches of low importance, AttentionAgent becomes temporarily blind to other signals, this effectively creates a bottleneck that forces it to focus on patches only if they are critical to the task. Once

learned, it is possible to visualize the  $K$  patches and have agent’s reasoning interpreted in the pixel space. Given the non-differentiable nature of the sorting and the pruning operations, AttentionAgent is optimized using CMA-ES.

The major building block of AttentionAgent is the self-attention mechanism. Although slightly modified in that context (i.e., the Value component is not utilized), as we have established previously, the indirect-encoding nature of the mechanism remains the same. More explicitly, the patch importance vector is based on the attention matrix  $A$ , which is the phenotype that is controlled by the two parameter matrices  $W_q, W_k$ , the genotype.

The advantages of employing indirect encoding in this context are clear: First, for an input image of size  $n$  (which can be substantial, e.g.,  $100\text{px} \times 100\text{px}$ , translating to tens of thousands of pixels), the attention matrix spans a space of size  $O(n^2)$ . Conversely,  $W_q, W_k$  transition image patches from  $d_{\text{in}} = 3$  (representing RGB colors) to a lower feature dimension  $d \ll n$ , resulting in a more manageable size of  $O(d)$ . Despite this significant reduction in representation space, the inductive bias inherent in the model’s design enables the genotype to effectively map to a set of phenotypes that are pertinent to the task at hand. As a result, AttentionAgent can solve complex problems with only a few thousand parameters, unlike other methods which may require hundreds of thousands or even millions of parameters. Second, the dynamic adaptive capability of self-attention allows AttentionAgent to flexibly adjust its decision-making based on the received inputs, resulting in more robust decisions that are not susceptible to external extractions such as changed background colors or hovering text on the screen, see Figure 4.19 for examples.

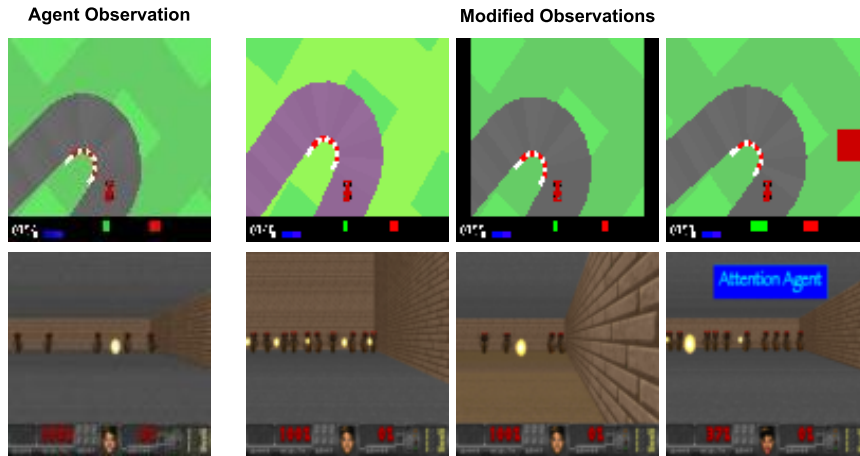


Figure 4.19: CarRacing and DoomTakeCover. *Left*: Original tasks. *Right*: Modified CarRacing environments: Color Perturbation, Vertical Frames, Background Blob. Modified DoomTakeCover environments: Higher Walls, Different Floor Texture, Hovering Text. Figure from (Tang, Nguyen, and Ha 2020).



#### 4.5 Chapter Review Questions

1. **Direct vs. Indirect Encoding:** What is the primary difference between direct and indirect encodings in neuroevolution? Why is indirect encoding particularly advantageous for tasks requiring large and complex neural networks?
2. **Biological Analogy:** How does the process of morphogenesis in biology inspire the concept of indirect encodings in neuroevolution? Provide an example of a biological principle that aligns with the goals of indirect encoding.
3. **Regularity in Neural Networks:** Why is the concept of regularity, such as symmetry and repetition with variation, important in indirect encodings? How does this principle enhance the efficiency and functionality of evolved solutions?
4. **Applications of Indirect Encodings:** How can indirect encodings be applied to a task such as evolving a quadrupedal robot controller? Discuss how they can utilize patterns and symmetries without manual intervention.
5. **Challenges of Direct Encoding:** Why is NEAT limited to smaller networks, and how do indirect encodings address this limitation? Provide an example illustrating how indirect encodings can simplify the representation of a complex neural network.
6. **Hypernetworks Overview:** What distinguishes hypernetworks from traditional local interaction-based indirect encodings? How does the "one-shot" generation of phenotypes make hypernetworks different from development-based approaches?
7. **CPPNs in Neuroevolution:** How do Compositional Pattern-Producing Networks (CPPNs) leverage geometric space and function composition to generate complex patterns? Provide an example of a regularity that CPPNs can encode effectively.
8. **HyperNEAT Substrate:** Explain how HyperNEAT utilizes neuron positions in a geometric space to generate connectivity patterns. Why is this approach particularly advantageous for tasks involving spatial regularities like controlling a quadrupedal robot?
9. **Strengths and Limitations:** In what types of tasks do HyperNEAT and CPPNs perform better compared to direct encodings like NEAT? Conversely, what are the limitations of these indirect encodings when applied to irregular or noisy domains?
10. **Evolvable Substrate HyperNEAT (ES-HyperNEAT):** How does ES-HyperNEAT improve upon the original HyperNEAT by evolving the substrate? Discuss how this extension enables the discovery of new nodes and connections in the network.