Chapter 8

NEAT, CPPN & HyperNEAT

- NEAT
- Genetic Algorithms
- CPPN
- HyperNEAT

In this chapter, we discuss three closely related neural network technologies: NEAT, CPPN and HyperNEAT. Kenneth Stanley's EPLEX group at the University of Central Florida conducts extensive research for all three technologies. Information about their current research can be found at the following URL:

http://eplex.cs.ucf.edu/

NeuroEvolution of Augmenting Topologies (NEAT) is an algorithm that evolves neural network structures with genetic algorithms. The compositional pattern-producing network (CPPN) is a type of evolved neural network that can create other structures, such as images or other neural networks. Hypercube-based NEAT, or HyperNEAT, a type of CPPN, also evolves other neural networks. Once HyperNEAT train the networks, they can easily handle much higher resolutions of their dimensions.

Many different frameworks support NEAT and HyperNEAT. For Java and C#, we recommend our own Encog implementation, which can be found at the following URL:

http://www.encog.org

You can find a complete list of NEAT implementations at Kenneth Stanley's website:

http://www.cs.ucf.edu/~kstanley/neat.html

Kenneth Stanley's website also includes a complete list of HyperNEAT implementations:

http://eplex.cs.ucf.edu/hyperNEATpage/

For the remainder of this chapter, we will explore each of these three network types.

8.1 NEAT Networks

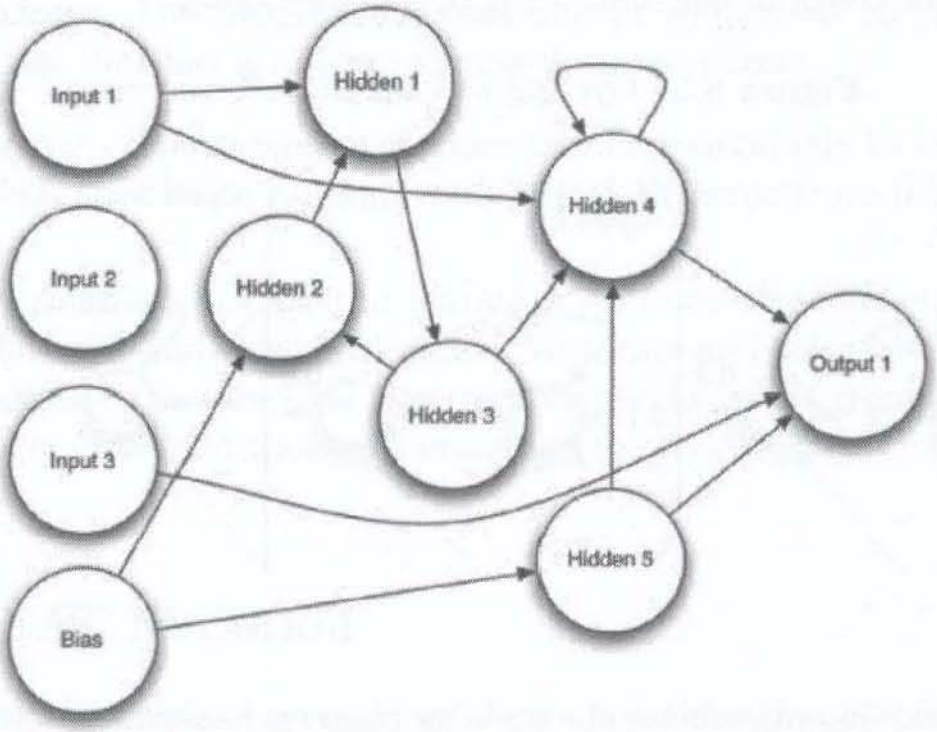
NEAT is a neural network structure developed by Stanley and Miikkulainen (2002). NEAT optimizes both the structure and weights of a neural network with a genetic algorithm (GA). The input and output of a NEAT neural network are identical to a typical feedforward neural network, as seen in previous chapters of this book.

A NEAT network starts out with only bias neurons, input neurons, and output neurons. Generally, none of the neurons have connections at the outset. Of course, a completely unconnected network is useless. NEAT makes no assumptions about whether certain input neurons are actually needed. An unneeded input is said to be statistically independent of the output. NEAT will often discover this independence by never evolving optimal genomes to connect to that statistically independent input neuron.

Another important difference between a NEAT network and an ordinary feedforward neural network is that other than the input and output layers,

NEAT networks do not have clearly defined hidden layers. However, the hidden neurons do not organize themselves into clearly delineated layers. One similarity between NEAT and feedforward networks is that they both use a sigmoid activation function. Figure 8.1 shows an evolved NEAT network:

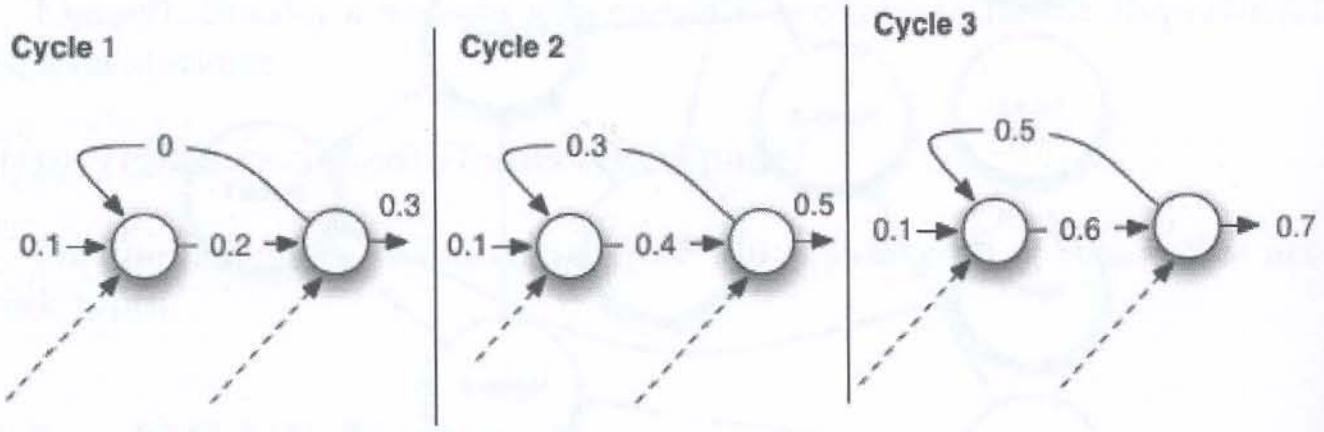
Figure 8.1: NEAT Network



Input 2 in the above image never formed any connections because the evolutionary process determined that input 2 was unnecessary. A recurrent connection also exists between hidden 3 and hidden 2. Hidden 4 has a recurrent connection to itself. Overall, you will note that a NEAT network lacks a clear delineation of layers.

You can calculate a NEAT network in exactly the same way as you do for a regular weighted feedforward network. You can manage the recurrent connections by running the NEAT network multiple times. This works by having the recurrent connection input start at 0 and update them each type you cycle through the NEAT network. Additionally, you must define a hyper-parameter to specify the number of times to calculate the NEAT network. Figure 8.2 shows recurrent link calculation when a NEAT network is instructed to cycle three times to calculate recurrent connections:

Figure 8.2: Cycling to Calculate Recurrences



The above diagram shows the outputs from each neuron, over each connection, for three cycles. The dashed lines indicate the additional connections. For simplicity, the diagram doesn't have the weights. The purpose of Figure 8.2 is to show that the recurrent output stays one cycle behind.

For the first cycle, the recurrent connection provided a 0 to the first neuron because neurons are calculated left to right. The first cycle has no value for the recurrent connection. For the second cycle, the recurrent connection now has the output 0.3, which the first cycle provided. Cycle 3 follows the same pattern, taking the 0.5 output from cycle 2 as the recurrent connection's output. Since there would be other neurons in the calculation, we have contrived these values, which the dashed arrows show at the bottom. However, Figure 8.2 does illustrate that the recurrent connections are cycled through previous cycles.

NEAT networks extensively use genetic algorithms, which we examined in *Artificial Intelligence for Humans, Volume 2: Nature-Inspired Algorithms.* Although you do not need to understand completely genetic algorithms to follow the discussion of them in this chapter, you can refer to Volume 2, as needed.

NEAT uses a typical genetic algorithm that includes:

- **Mutation** - The program chooses one fit individual to create a new individual that has a random change from its parent.
- **Crossover** - The program chooses two fit individuals to create a new individual that has a random sampling of elements from both parents.

All genetic algorithms engage the mutation and crossover genetic operators with a population of individual solutions. Mutation and crossover choose with greater probability the solutions that receive higher scores from an objective function. We explore mutation and crossover for NEAT networks in the next two sections.

8.1.1 NEAT Mutation

NEAT mutation consists of several mutation operations that can be performed on the parent genome. We discuss these operations here:

- **Add a neuron**: By selecting a random link, we can add a neuron. A new neuron and two links replace this random link. The new neuron effectively splits the link. The program selects the weights of each of the two new links to provide nearly the same effective output as the link being replaced.
- **Add a link**: The program chooses a source and destination, or two random neurons. The new link will be between these two neurons. Bias neurons can never be a destination. Output neurons cannot be a source. There will never be more than two links in the same direction between the same two neurons.

- **Remove a link**: Links can be randomly selected for removal. If there are no remaining links interacting with them, you can remove the hidden neurons, which are neurons that are not input, output, or the single bias neuron.

- **Perturb a weight**: You can choose a random link. Then multiply its weight by a number from a normal random distribution with a gamma of 1 or lower. Smaller random numbers will usually cause a quicker convergence. A gamma value of 1 or lower will specify that a single standard deviation will sample a random number of 1 or lower.

You can increase the probability of the mutation so that the weight perturbation occurs more frequently, thereby allowing fit genomes to vary their weights and further adapt through their children. The structural mutations happen with much less frequency. You can adjust the exact frequency of each operation with most NEAT implementations.

8.1.2 NEAT Crossover

NEAT crossover is more complex than many genetic algorithms because the NEAT genome is an encoding of the neurons and connections that comprise an individual genome. Most genetic algorithms assume that the number of genes is consistent across all genomes in the population. In fact, child genomes in NEAT that result from both mutation and crossover may have a different number of genes than their parents. Managing this number discrepancy requires some ingenuity when you implement the NEAT crossover operation.

NEAT keeps a database of all the changes made to a genome through mutation. These changes are called innovations, and they exist in order to implement mutations. Each time an innovation is added, it is given an ID. These IDs will also be used to order the innovations. We will see that it is important to select the innovation with the lower ID when choosing between two innovations.

It is important to realize that the relationship between innovations and mutations is not one to one. It can take several innovations to achieve one mutation. The only two types of innovation are creating a neuron and a link

between two neurons. One mutation might result from multiple innovations. Additionally, a mutation might not have any innovations. Only mutations that add to the structure of the network will generate innovations. The following list summarizes the innovations that the previously mentioned mutation types could potentially create.

- **Add a neuron**: One new neuron innovation and two new link innovations
- **Add a link**: One new link innovation
- **Remove a link**: No innovations
- **Perturb a weight**: No innovations

You also need to note that NEAT will not recreate innovation records if you have already attempted this type of innovation. Furthermore, innovations do not contain any weight information; innovations only contain structural information.

Crossover for two genomes occurs by considering the innovations, and this trait allows NEAT to ensure that all prerequisite innovations are also present. A naïve crossover, such as those that many genetic algorithms use, would potentially combine links with nonexistent neurons. Listing 8.1 shows the entire NEAT crossover function in pseudocode:

Listing 8.1: NEAT Crossover

```
def neat_crossover(rnd,mom,dad):
# Choose best genome (by objective function), if tie, choose
    random.
  best = favor_parent(rnd, mom, dad)
  not_best = dad if (best <> mom) else mom
  selected_links = []
  selected_neurons = []
# current gene index from mom and dad
  cur_mom = 0
  cur_dad = 0
  selected_gene = None
# add in the input and bias, they should always be here
always_count = mom.input_count + mom.output_count + 1
```

```
for i from 0 to always_count-1:
  selected_neurons.add(i, best, not_best)
# Loop over all genes in both mother and father
  while (cur_mom < mom.num_genes) or (cur_dad < dad.num_genes):
# The mom and dad gene object
    mom_gene = None
    mom_innovation = -1
    dad_gene = None
    dad_innovation = -1
# grab the actual objects from mom and dad for the specified
# indexes
# if there are none, then None
    if cur_mom < mom.num_genes:
      mom_gene = mom.links[cur_mom];
      mom_innovation = mom_gene.innovation_id
    if cur_dad < dad.num_genes:
      dad_gene = dad.links[cur_dad]
      dad_innovation_id = dad_gene.innovation_id
# now select a gene fror mom or dad. This gene is for the baby
# Dad gene only, mom has run out
    if mom_gene == None and dad_gene <> None:
      cur_dad = cur_dad + 1
      selected_gene = dad_gene
# Mom gene only, dad has run out
    else if dadGene == null and momGene <> null:
      cur_mom = cur_mom + 1
      selected_gene = mom_gene
# Mom has lower innovation number
    else if mom_innovation_id < dad_innovation_id:
      cur_mom = cur_mom + 1
      if best == mom:
        selected_gene = mom_gene
# Dad has lower innovation number
    else if dad_innovation_id < mom_innovation_id:
      cur_dad = cur_dad + 1
      if best == dad:
        selected_gene = dad_gene
# Mom and dad have the same innovation number
# Flip a coin.
    else if dad_innovation_id == mom_innovation_id:
      cur_dad = cur_dad + 1
      cur_mom = cur_mom + 1
      if rnd.next_double() >0.5:
```

```
        selected_gene = dad_gene
      else:
        selected_gene = mom_gene
# If a gene was chosen for the child then process it.
# If not, the loop continues.
    if selected_gene <> None:
# Do not add the same innovation twice in a row.
      if selected_links.count == 0:
        selected_links.add(selected_gene)
      else:
        if selected_links[selected_links.count-1]
            .innovation_id <> selected_gene.innovation_id {
          selected_links.add(selected_gene)
# Check if we already have the nodes referred to in
# SelectedGene.
# If not, they need to be added.
      selected_neurons.add(
        selected_gene.from_neuron_id, best, not_best)
      selected_neurons.add(
        selected_gene.to_neuron_id, best, not_best)
# Done looping over parent's genes
  baby = new NEATGenome(selected_links, selected_neurons)
  return baby
```

The above implementation of crossover is based on the NEAT crossover operator implemented in Encog. We provide the above comments in order to explain the critical sections of code. The primary evolution occurs on the links contained in the mother and father. Any neurons needed to support these links are brought along when the child genome is created. The code contains a main loop that loops over both parents, thereby selecting the most suitable link gene from each parent. The link genes from both parents are essentially stitched together so they can find the most suitable gene. Because the parents might be different lengths, one will likely exhaust its genes before this process is complete.

Each time through the loop, a gene is chosen from either the mother or father according to the following criteria:

- If mom or dad has run out, choose the other. Move past the chosen gene.
- If mom has a lower innovation ID number, choose mom if she has the best score. In either case, move past mom's gene.
- If dad has a lower innovation ID number, choose dad if he has the best score. In either case, move past dad's gene.
- If mom and dad have the same innovation ID, pick one randomly, and move past their gene.

You can consider that the mother and father's genes are both on a long tape. A marker for each tape holds the current position. According to the rules above, the marker will move past a parent's gene. At some point, each parent's marker moves to the end of the tape, and that parent runs out of genes.

8.1.3 NEAT Speciation

Crossover is a tricky for computers to properly perform. In the animal and plant kingdoms, crossover occurs only between members of the same species. What exactly do we mean by species? In biology, scientists define species as members of a population that can produce viable offspring. Therefore, a crossover between a horse and humming bird genome would be catastrophically unsuccessful. Yet a naive genetic algorithm would certainly try something just as disastrous with artificial computer genomes!

The NEAT speciation algorithm has several variants. In fact, one of the most advanced variants can group the population into a predefined number of clusters with a type of k-means clustering. You can subsequently determine the relative fitness of each species. The program gives each species a percentage of the next generation's population count. The members of each species then compete in virtual tournaments to determine which members of the species will be involved in crossover and mutation for the next generation.

A tournament is an effective way to select parents from a species. The program performs a certain number of trials. Typically we use five trials. For each trial, the program selects two random genomes from the species. The fitter of each genome advances to the next trial. This process is very efficient for threading, and it is also biologically plausible. The advantage to this selection method is that the winner doesn't have to beat the best genome in the species. It has to beat the best genome in the trials. You must run a tournament for each parent needed. Mutation requires one parent, and crossover needs two parents.

In addition to the trials, several other factors determine the species members chosen for mutation and crossover. The algorithm will always carry one or more elite genomes to the next species. The number of elite genomes is configurable. The program gives younger genomes a bonus so they have a chance to try new innovations. Interspecies crossover will occur with a very low probability.

All of these factors together make NEAT a very effective neural network type. NEAT removes the need to define how the hidden layers of a neural network are structured. The absence of a strict structure of hidden layers allows NEAT neural networks to evolve the connections that are actually needed.

8.2 CPPN Networks

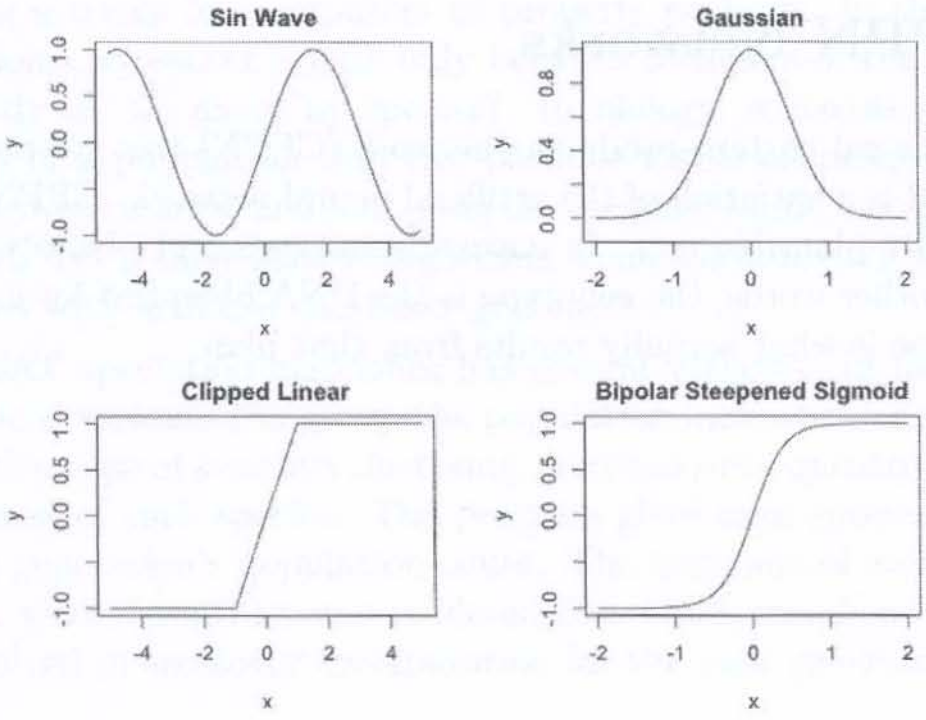
The compositional pattern-producing network (CPPN) was invented by Stanley (2007) and is a variation of the artificial neural network. CPPN recognizes one biologically plausible fact. In nature, genotypes and phenotypes are not identical. In other words, the genotype is the DNA blueprint for an organism. The phenotype is what actually results from that plan.

In nature, the genome is the instructions for producing a phenotype that is much more complex than the genotype. In the original NEAT, as seen in the last section, the genome describes link for link and neuron for neuron how to produce the phenotype. However, CPPN is different because it creates a population of special NEAT genomes. These genomes are special in two ways. First, CPPN doesn't have the limitations of regular NEAT, which always uses a sigmoid activation function. CPPN can use any of the following activation functions:

- Clipped linear
- Bipolar steepened sigmoid
- Gaussian
- Sine
- Others you might define

You can see these activation functions in Figure 8.3:

Figure 8.3: CPPN Activation Functions



The second difference is that the NEAT networks produced by these genomes are not the final product. They are not the phenotype. However, these NEAT genomes do know how to create the final product.

The final phenotype is a regular NEAT network with a sigmoid activation function. We can use the above four activation functions only for the genomes. The ultimate phenotype always has a sigmoid activation function.

8.2.1 CPPN Phenotype

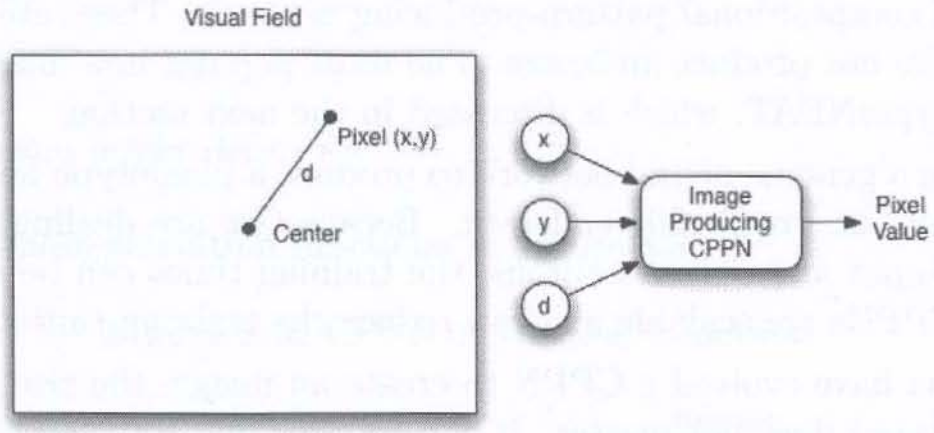
CPPNs are typically used in conjunction with images, as the CPPN phenotype is usually an image. Though images are the usual product of a CPPN, the only real requirement is that the CPPN compose something, thereby earning its name of compositional pattern-producing network. There are cases where a CPPN does not produce an image. The most popular non-image producing CPPN is HyperNEAT, which is discussed in the next section.

Creating a genome neural network to produce a phenotype neural network is a complex but worthwhile endeavor. Because we are dealing with a large number of input and output neurons, the training times can be considerable. However, CPPNs are scalable and can reduce the training times.

Once you have evolved a CPPN to create an image, the size of the image (the phenotype) does not matter. It can be 320x200, 640x480 or some other resolution altogether. The image phenotype, generated by the CPPN will grow to the size needed. As we will see in the next section, CPPNs give HyperNEAT the same sort of scalability.

We will now look at how a CPPN, which is itself a NEAT network, produces an image, or the final phenotype. The NEAT CPPN should have three input values: the coordinate on the horizontal axis (x), the coordinate on the vertical axis (y), and the distance of the current coordinate from the center (d). Inputting d provides a bias towards symmetry. In biological genomes, symmetry is important. The output from the CPPN corresponds to the pixel color at the x-coordinate and y-coordinate. The CPPN specification only determines how to process a grayscale image with a single output that indicates intensity. For a full-color image, you could use output neurons for red, green, and blue. Figure 8.4 shows a CPPN for images:

Figure 8.4: CPPN for Images



You can query the above CPPN for every x-coordinate and y-coordinate needed. Listing 8.2 shows the pseudocode that you can use to generate the phenotype:

Listing 8.2: Generate CPPN Image

```
def render_cppn(net,bitmap):
  for y from 1 to bitmap.height:
    for x from 1 to bitmap.width:
# Normalize x and y to -1,1
      norm_x = (2*(x/bitmap.width))-1
      norm_y = (2*(y/bitmap.height))-1
# Distance from center
      d = sqrt( (norm_x/2)^2
    + (norm_y /2)^2 )
```

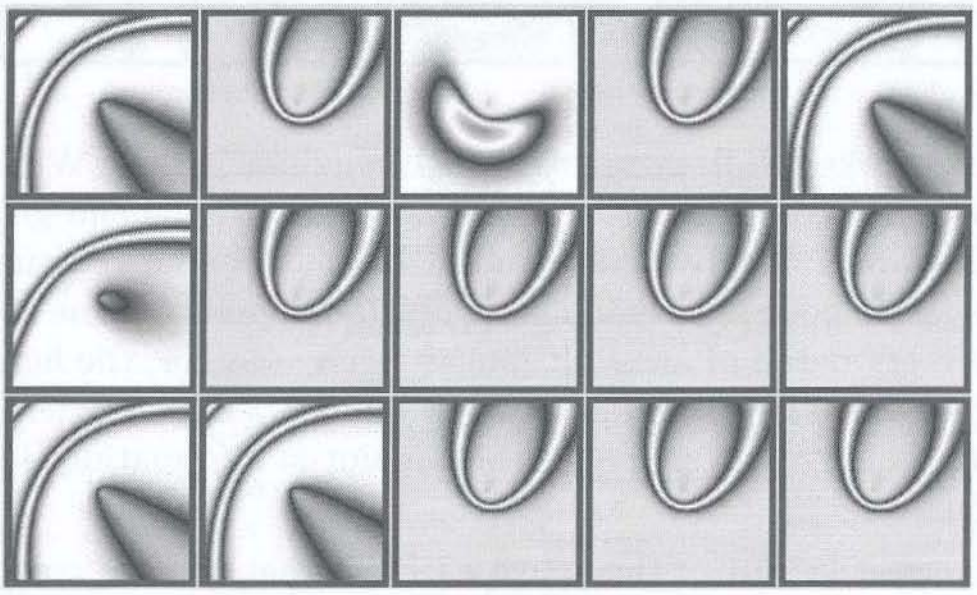
```
# Call CPPN
    input = [x,y,d]
  color = net.compute(input)
# Output pixel
  bitmap.plot(x-1,y-1, color)
```

The above code simply loops over every pixel and queries the CPPN for the color at that location. The x-coordinate and y-coordinate are normalized to being between -1 and +1. You can see this process in action at the Picbreeder website at following URL:

http://picbreeder.org/

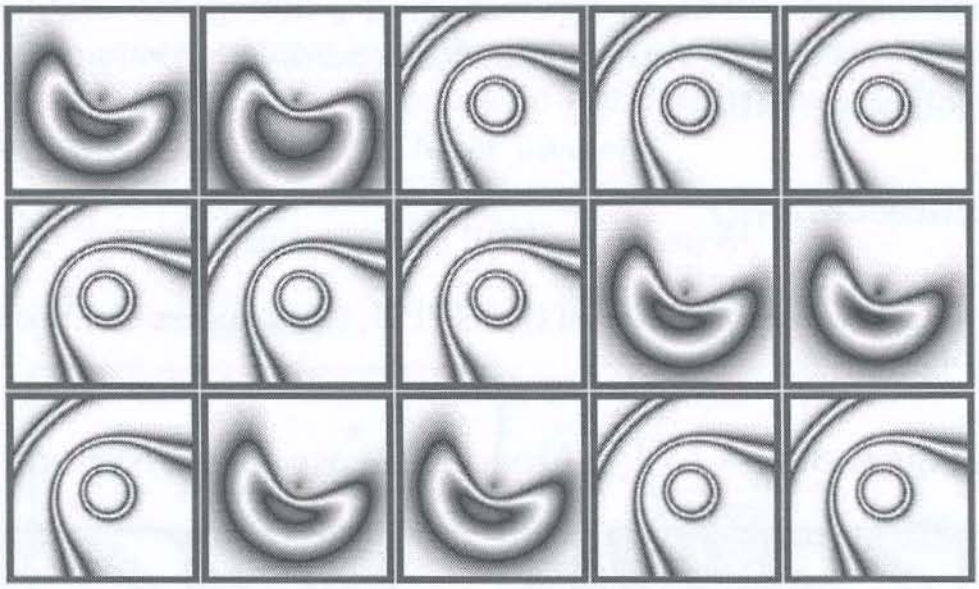
Depending on the complexity of the CPPN, this process can produce images similar to Figure 8.5:

Figure 8.5: A CPPN-Produced Image (picbreeder.org)



Picbreeder allows you to select one or more parents to contribute to the next generation. We selected the image that resembles a mouth, as well as the image to the right. Figure 8.6 shows the subsequent generation that Picbreeder produced.

Figure 8.6: A CPPN-Produced Image (picbreeder.org)



CPPN networks handle symmetry just like human bodies. With two hands, two kidneys, two feet, and other body part pairs, the human genome seems to have a hierarchy of repeated features. Instructions for creating an eye or various tissues do not exist. Fundamentally, the human genome does not have to describe every detail of an adult human being. Rather, the human genome only has to describe how to build an adult human being by generalizing many of the steps. This greatly simplifies the amount of information that is needed in a genome.

Another great feature of the image CPPN is that you can create the above images at any resolution and without retraining. Because the x-coordinate and y-coordinate are normalized to between -1 and +1, you can use any resolution.

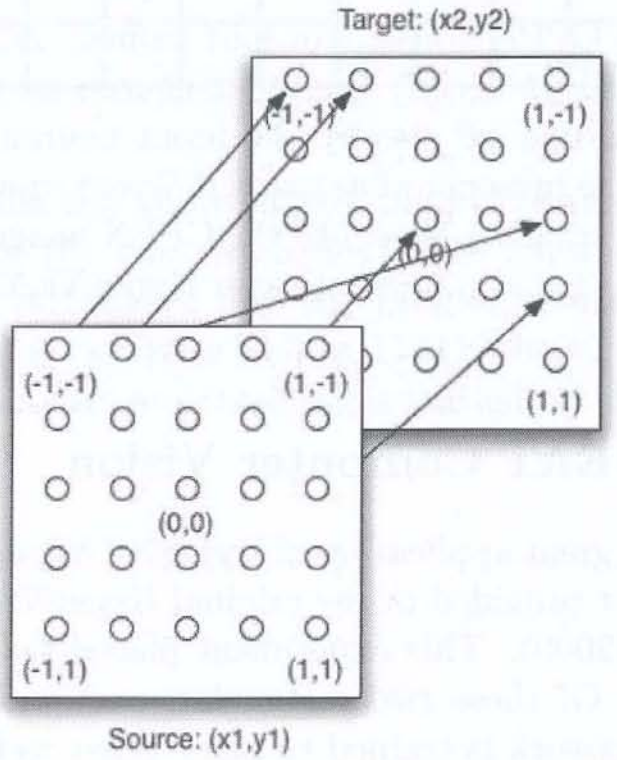
8.3 HyperNEAT Networks

HyperNEAT networks, invented by Stanley, D'Ambrosio, & Gauci (2009), are based upon the CPPN; however, instead of producing an image, a HyperNEAT network creates another neural network. Just like the CPPN in the last section, HyperNEAT can easily create much higher resolution neural networks without retraining.

8.3.1 HyperNEAT Substrate

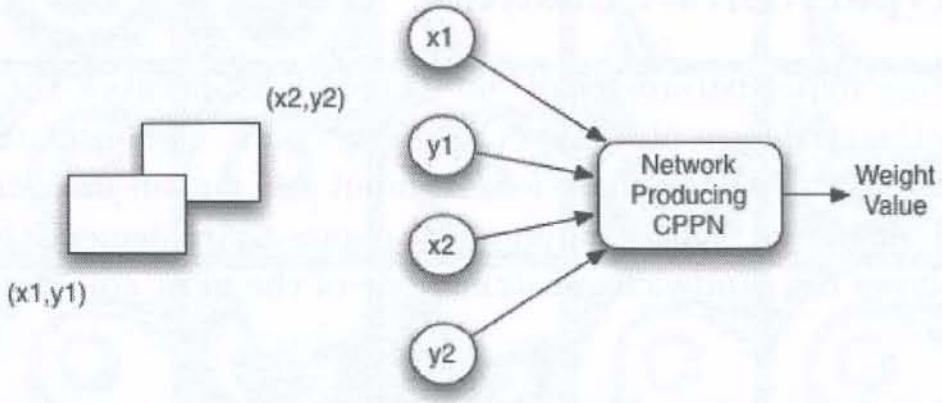
One interesting hyper-parameter of the HyperNEAT network is the substrate that defines the structure of a HyperNEAT network. A substrate defines the x-coordinate and the y-coordinate for the input and output neurons. Standard HyperNEAT networks usually employ two planes to implement the substrate. Figure 8.7 shows the sandwich substrate, one of the most common substrates:

Figure 8.7: HyperNEAT Sandwich Substrate



Together with the above substrate, a HyperNEAT CPPN is capable of creating the phenotype neural network. The source plane contains the input neurons, and the target plane contains the output neurons. The x-coordinate and the y-coordinate for each are in the -1 to +1 range. There can potentially be a weight between each of the source neurons and every target neuron. Figure 8.8 shows how to query the CPPN to determine these weights:

Figure 8.8: CPPN for HyperNEAT



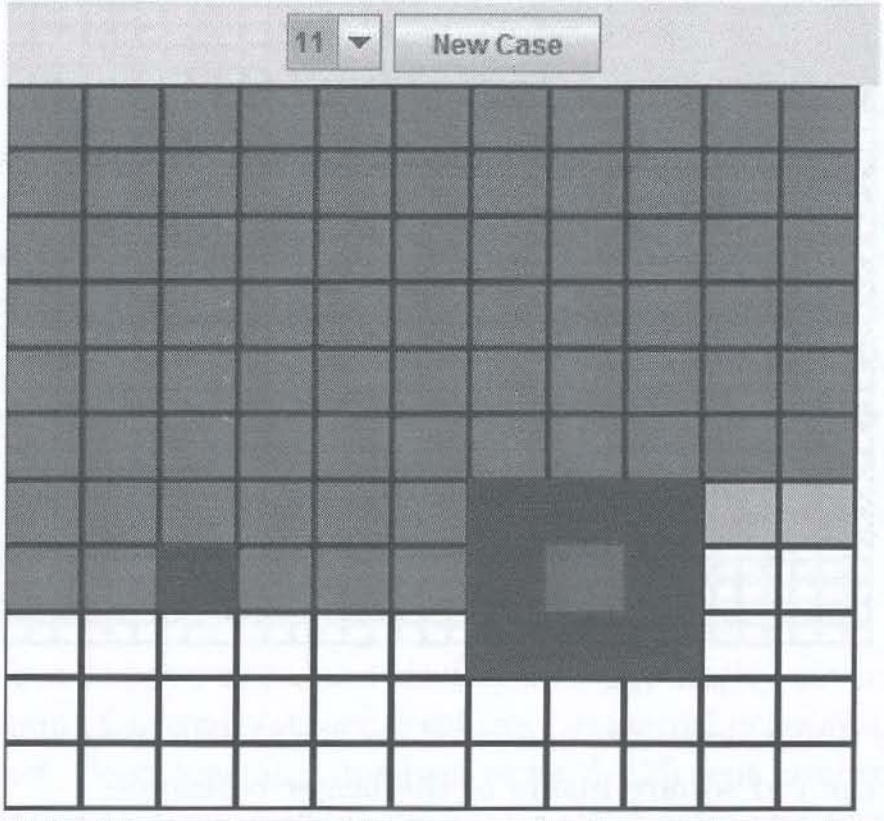
The input to the CPPN consists of four values: *x1*, *y1*, *x2*, and *y2*. The first two values *x1* and *y1* specify the input neuron on the source plane. The second two values *x2* and *y2* specify the input neuron on the target plane. HyperNEAT allows the presence of as many different input and output neurons as desired, without retraining. Just like the CPPN image could map more and more pixels between -1 and +1, so too can HyperNEAT pack in more input and output neurons.

8.3.2 HyperNEAT Computer Vision

Computer vision is a great application of HyperNEAT, as demonstrated by the rectangles experiment provided in the original HyperNEAT paper by Stanley, Kenneth O., et al. (2009). This experiment placed two rectangles in a computer's vision field. Of these two rectangles, one is always larger than the other. The neural network is trained to place a red rectangle near the center

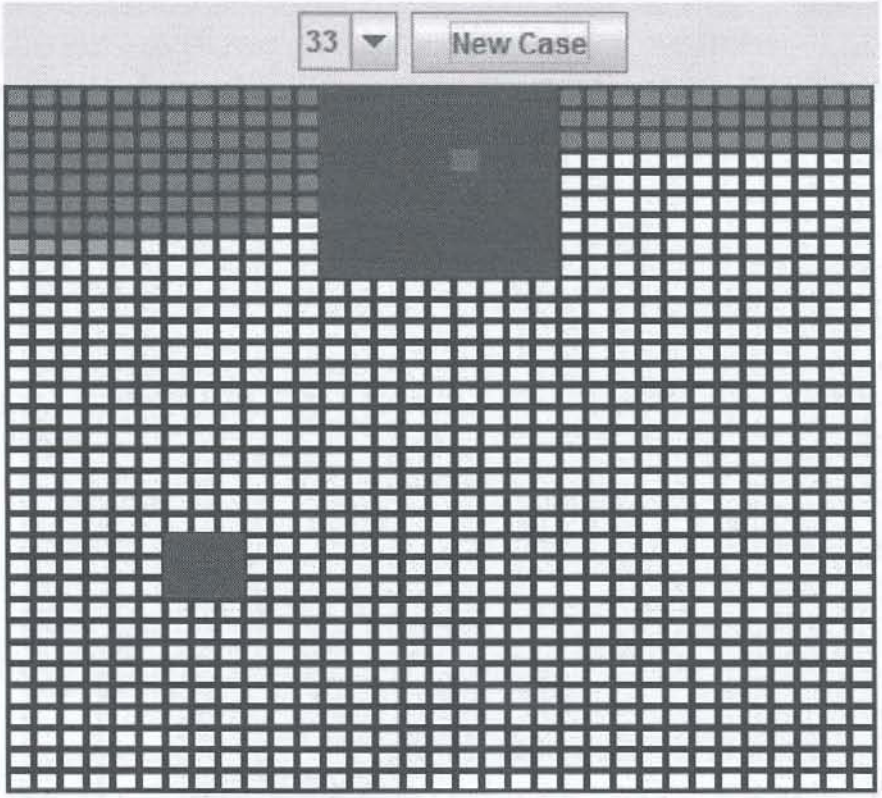
of the larger rectangle. Figure 8.9 shows this experiment running under the Encog framework:

Figure 8.9: Boxes Experiment (11 resolution)



As you can see from the above image, the red rectangle is placed directly inside of the larger of the two rectangles. The "New Case" button can be pressed to move the rectangles, and the program correctly finds the larger rectangle. While this works quite well at 11x11, the size can be increased to 33x33. With the larger size, no retraining is needed, as shown in Figure 8.10:

Figure 8.10: Boxes Experiment (33 resolution)



When the dimensions are increased to 33x33, the neural network is still able to place the red square inside of the larger rectangle.

The above example uses a sandwich substrate with the input and output plane both equal to the size of the visual field, in this case 33x33. The input plane provides the visual field. The neuron in the output plane with the highest output is the program's guess at the center of the larger rectangle. The fact that the position of the large rectangle does not confuse the network shows that HyperNEAT possesses some of the same features as the convolutional neural networks that we will see in Chapter 10, "Convolutional Networks."

8.4 Chapter Summary

This chapter introduced NEAT, CPPN, and HyperNEAT. Kenneth Stanley's EPLEX group at the University of Central Florida extensively researches all three technologies. NeuroEvolution of Augmenting Topologies (NEAT) is an

algorithm that uses genetic algorithms to automatically evolve neural network structures. Often the decision of the structure of a neural network can be one of the most complex aspects of neural network design. NEAT neural networks can evolve their own structure and even decide what input features are important.

The compositional pattern-producing network (CPPN) is a type of neural network that is evolved to create other structures, such as images or other neural networks. Image generation is a common task for CPPNs. The Picbreeder website allows new images to be bred based on previous images generated at this site. CPPNs can generate more than just images. The HyperNEAT algorithm is an application of CPPNs for producing neural networks.

Hypercube-based NEAT, or HyperNEAT, is a type of CPPN that evolves other neural networks that can easily handle much higher resolutions of their dimensions as soon as they are trained. HyperNEAT allows a CPPN to be evolved that can create neural networks. Being able to generate the neural network allows you to introduce symmetry, and it gives you the ability to change the resolution of the problem without retraining.

Neural networks have risen and declined in popularity several times since their introduction. Currently, there is interest in neural networks that use deep learning. In fact, deep learning involves several different concepts. The next chapter introduces deep neural networks, and we expand this topic throughout the remainder of this book.