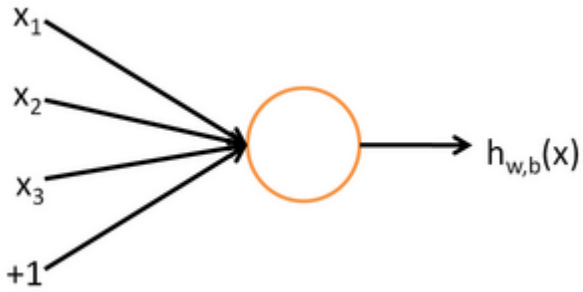
Neural Networks

From Ufldl

Consider a supervised learning problem where we have access to labeled training examples $(x^{(i)},y^{(i)})$. Neural networks give a way of defining a complex, non-linear form of hypotheses $h_{W,b}(x)$, with parameters W,b that we can fit to our data.

To describe neural networks, we will begin by describing the simplest possible neural network, one which comprises a single "neuron." We will use the following diagram to denote a single neuron:



This "neuron" is a computational unit that takes as input x_1,x_2,x_3 (and a +1 intercept term), and outputs $h_{W,b}(x) = f(W^T x) = f(\sum_{i=1}^{3} W_i x_i + b)$, where $f : \Re \mapsto \Re$ is called the **activation function**. In these notes, we will choose $f(\cdot)$ to be the sigmoid function:

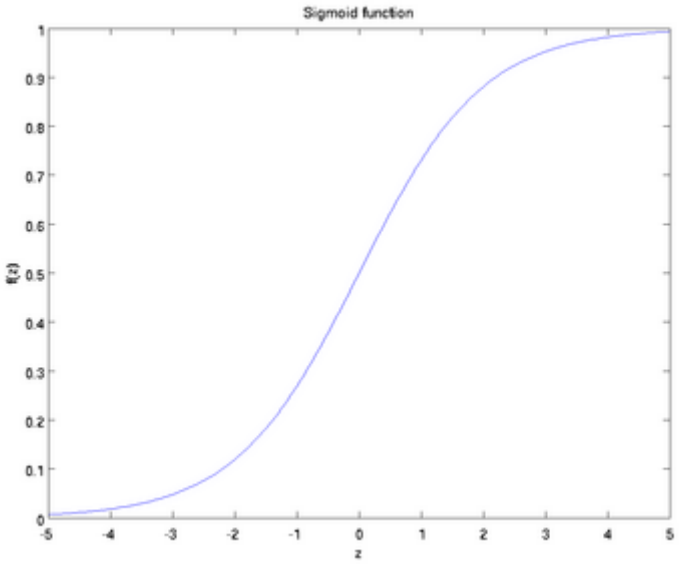
$$f(z) = \frac{1}{1+\exp(-z)}.$$

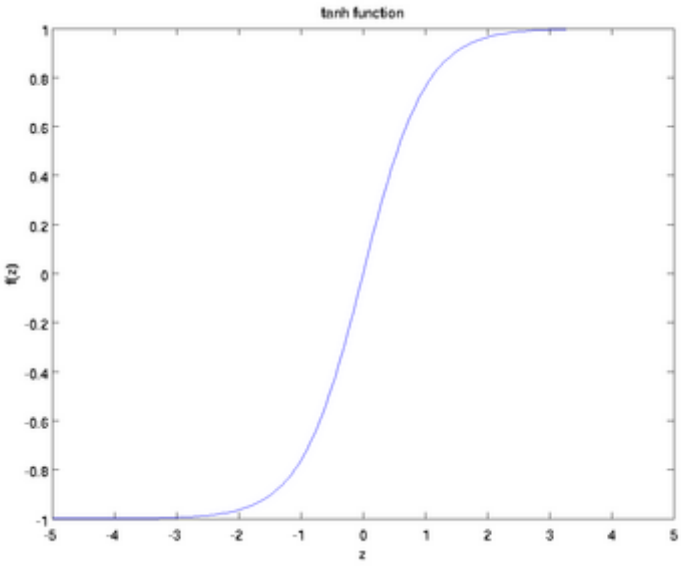
Thus, our single neuron corresponds exactly to the input-output mapping defined by logistic regression.

Although these notes will use the sigmoid function, it is worth noting that another common choice for f is the hyperbolic tangent, or tanh, function:

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}},$$

Here are plots of the sigmoid and tanh functions:





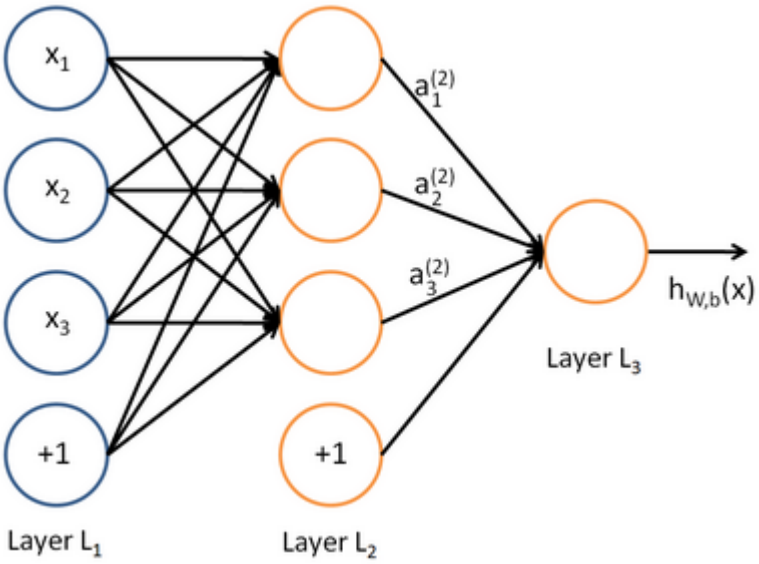
The tanh(z) function is a rescaled version of the sigmoid, and its output range is $[-1,1]$ instead of $[0,1]$.

Note that unlike some other venues (including the OpenClassroom videos, and parts of CS229), we are not using the convention here of $x_0 = 1$. Instead, the intercept term is handled separately by the parameter b.

Finally, one identity that'll be useful later: If $f(z) = 1 / (1 + \exp(-z))$ is the sigmoid function, then its derivative is given by $f'(z) = f(z)(1 - f(z))$. (If f is the tanh function, then its derivative is given by $f'(z) = 1 - (f(z))^2$.) You can derive this yourself using the definition of the sigmoid (or tanh) function.

Neural Network model

A neural network is put together by hooking together many of our simple "neurons," so that the output of a neuron can be the input of another. For example, here is a small neural network:



In this figure, we have used circles to also denote the inputs to the network. The circles labeled "+1" are called **bias units**, and correspond to the intercept term. The leftmost layer of the network is called the **input layer**, and the rightmost layer the **output layer** (which, in this example, has only one node). The middle layer of nodes is called the **hidden layer**, because its values are not observed in the training set. We also say that our example neural network has 3 **input units** (not counting the bias unit), 3 **hidden units**, and 1 **output unit**.

We will let n_l denote the number of layers in our network; thus $n_l = 3$ in our example. We label layer l as L_l, so layer L_1 is the input layer, and layer L_{n_l} the output layer. Our neural network has parameters $(W,b) = (W^{(1)},b^{(1)},W^{(2)},b^{(2)})$, where we write $W_{ij}^{(l)}$ to denote the parameter (or weight) associated with the connection between unit j in layer l, and unit i in layer $l+1$. (Note the order of the indices.) Also, $b_i^{(l)}$ is the bias associated with unit i in layer $l+1$. Thus, in our example, we have $W^{(1)} \in \Re^{3\times 3}$, and $W^{(2)} \in \Re^{1\times 3}$. Note that bias units don't have inputs or connections going into them, since they always output the value +1. We also let s_l denote the number of nodes in layer l (not counting the bias unit).

We will write $a_i^{(l)}$ to denote the **activation** (meaning output value) of unit i in layer l. For $l = 1$, we also use $a_i^{(1)} = x_i$ to denote the i-th input. Given a fixed setting of the parameters W,b, our neural network defines a hypothesis $h_{W,b}(x)$ that outputs a real number. Specifically, the computation that this neural network represents is given by:

$$
\begin{aligned}
a_1^{(2)} &= f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}) \\
a_2^{(2)} &= f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)}) \\
a_3^{(2)} &= f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)}) \\
h_{W,b}(x) = a_1^{(3)} &= f(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)})
\end{aligned}
$$

In the sequel, we also let $z_i^{(l)}$ denote the total weighted sum of inputs to unit i in layer l, including the bias term (e.g., $z_i^{(2)} = \sum_{j=1}^{n} W_{ij}^{(1)}x_j + b_i^{(1)}$), so that $a_i^{(l)} = f(z_i^{(l)})$.

Note that this easily lends itself to a more compact notation. Specifically, if we extend the activation function $f(\cdot)$ to apply to vectors in an element-wise fashion (i.e., $f([z_1,z_2,z_3]) = [f(z_1),f(z_2),f(z_3)]$), then we can write the equations above more compactly as:

$$
\begin{aligned}
z^{(2)} &= W^{(1)}x + b^{(1)} \\
a^{(2)} &= f(z^{(2)}) \\
z^{(3)} &= W^{(2)}a^{(2)} + b^{(2)} \\
h_{W,b}(x) &= a^{(3)} = f(z^{(3)})
\end{aligned}
$$

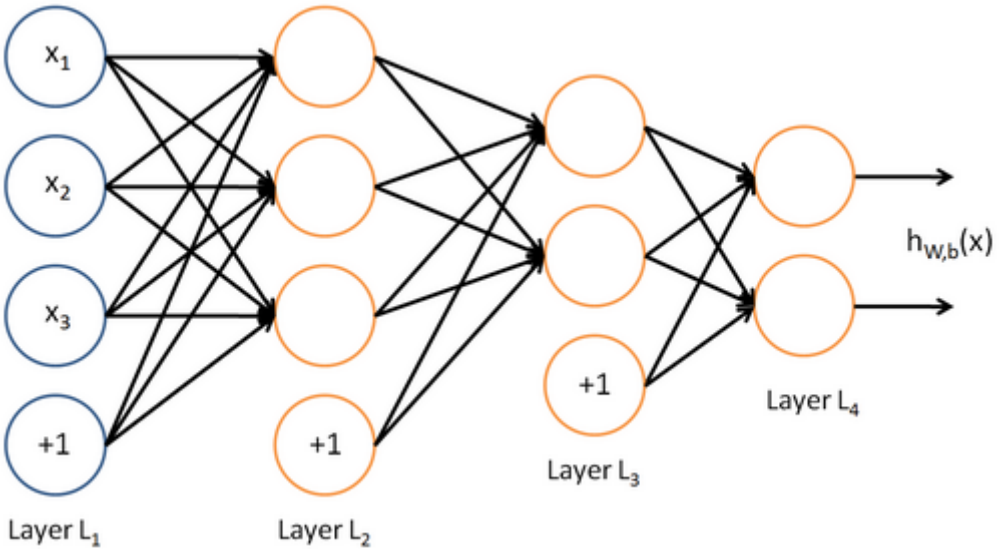
We call this step **forward propagation.** More generally, recalling that we also use $a^{(1)} = x$ to also denote the values from the input layer, then given layer l's activations $a^{(l)}$, we can compute layer $l + 1$'s activations $a^{(l + 1)}$ as:

$$
\begin{aligned}
z^{(l+1)} &= W^{(l)}a^{(l)} + b^{(l)} \\
a^{(l+1)} &= f(z^{(l+1)})
\end{aligned}
$$

By organizing our parameters in matrices and using matrix-vector operations, we can take advantage of fast linear algebra routines to quickly perform calculations in our network.

We have so far focused on one example neural network, but one can also build neural networks with other **architectures** (meaning patterns of connectivity between neurons), including ones with multiple hidden layers. The most common choice is a n_l-layered network where layer 1 is the input layer, layer n_l is the output layer, and each layer l is densely connected to layer $l + 1$. In this setting, to compute the output of the network, we can successively compute all the activations in layer L_2, then layer L_3, and so on, up to layer L_{n_l}, using the equations above that describe the forward propagation step. This is one example of a **feedforward** neural network, since the connectivity graph does not have any directed loops or cycles.

Neural networks can also have multiple output units. For example, here is a network with two hidden layers layers L_2 and L_3 and two output units in layer L_4:



To train this network, we would need training examples $(x^{(i)}, y^{(i)})$ where $y^{(i)} \in \Re^2$. This sort of network is useful if there're multiple outputs that you're interested in predicting. (For example, in a medical diagnosis application, the vector x might give the input features of a patient, and the different outputs y_i's might indicate presence or absence of different diseases.)

Language : 中文

Backpropagation Algorithm

From Ufldl

Suppose we have a fixed training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ of *m* training examples. We can train our neural network using batch gradient descent. In detail, for a single training example (*x*,*y*), we define the cost function with respect to that single example to be:

$$J(W,b;x,y) = \frac{1}{2} \left\| h_{W,b}(x) - y \right\|^2 .$$

This is a (one-half) squared-error cost function. Given a training set of *m* examples, we then define the overall cost function to be:

$$\begin{aligned} J(W,b) &= \left[\frac{1}{m} \sum_{i=1}^{m} J(W,b;x^{(i)},y^{(i)}) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} \left(W_{ji}^{(l)} \right)^2 \\ &= \left[\frac{1}{m} \sum_{i=1}^{m} \left(\frac{1}{2} \left\| h_{W,b}(x^{(i)}) - y^{(i)} \right\|^2 \right) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} \left(W_{ji}^{(l)} \right)^2 \end{aligned}$$

The first term in the definition of *J*(*W*,*b*) is an average sum-of-squares error term. The second term is a regularization term (also called a **weight decay** term) that tends to decrease the magnitude of the weights, and helps prevent overfitting.

[Note: Usually weight decay is not applied to the bias terms $b_i^{(l)}$, as reflected in our definition for *J*(*W*,*b*). Applying weight decay to the bias units usually makes only a small difference to the final network, however. If you've taken CS229 (Machine Learning) at Stanford or watched the course's videos on YouTube, you may also recognize this weight decay as essentially a variant of the Bayesian regularization method you saw there, where we placed a Gaussian prior on the parameters and did MAP (instead of maximum likelihood) estimation.]

The **weight decay parameter** λ controls the relative importance of the two terms. Note also the slightly overloaded notation: *J*(*W*,*b*;*x*,*y*) is the squared error cost with respect to a single example; *J*(*W*,*b*) is the overall cost function, which includes the weight decay term.

This cost function above is often used both for classification and for regression problems. For classification, we let *y* = 0 or 1 represent the two class labels (recall that the sigmoid activation function outputs values in [0,1]; if we were using a tanh activation function, we would instead use -1 and +1 to denote the labels). For regression problems, we first scale our outputs to ensure that they lie in the [0,1] range (or if we were using a tanh activation function, then the [− 1,1] range).

Our goal is to minimize *J*(*W*,*b*) as a function of *W* and *b*. To train our neural network, we will initialize each parameter $W_{ij}^{(l)}$ and each $b_i^{(l)}$ to a small random value near zero (say according to a *Normal*(0,ε^2) distribution for some small ε, say 0.01), and then apply an optimization algorithm such as batch gradient descent. Since *J*(*W*,*b*) is a non-convex function, gradient descent is susceptible to local optima; however, in practice gradient descent usually works fairly well. Finally, note that it is important to initialize the parameters randomly, rather than to all 0's. If all the parameters start off at identical values, then all the hidden layer units will end up learning the same function of the input (more formally, $W_{ij}^{(1)}$ will be the same for all values of *i*, so that $a_1^{(2)} = a_2^{(2)} = a_3^{(2)} = \ldots$ for any input *x*). The random initialization serves the purpose of **symmetry breaking**.

One iteration of gradient descent updates the parameters *W*,*b* as follows:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b)$$

$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W,b)$$

where α is the learning rate. The key step is computing the partial derivatives above. We will now describe the **backpropagation** algorithm, which gives an efficient way to compute these partial derivatives.

We will first describe how backpropagation can be used to compute $\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x,y)$ and $\frac{\partial}{\partial b_i^{(l)}} J(W,b;x,y)$, the partial derivatives of the cost function *J*(*W*,*b*;*x*,*y*) defined with respect to a single example (*x*,*y*). Once we can compute these, we see that the derivative of the overall cost function *J*(*W*,*b*) can be computed as:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b) = \left[\frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x^{(i)},y^{(i)}) \right] + \lambda W_{ij}^{(l)}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W,b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial b_i^{(l)}} J(W,b;x^{(i)},y^{(i)})$$

The two lines above differ slightly because weight decay is applied to *W* but not *b*.

The intuition behind the backpropagation algorithm is as follows. Given a training example (*x*,*y*), we will first run a "forward pass" to compute all the activations throughout the network, including the output value of the hypothesis $h_{W,b}(x)$. Then, for each node *i* in layer *l*, we would like to compute an "error term" $\delta_i^{(l)}$ that measures how much that node was "responsible" for any errors in our output. For an output node, we can directly measure the difference between the network's activation and the true target value, and use that to define $\delta_i^{(n_l)}$ (where layer n_l is the output layer). How about hidden units? For those, we will compute $\delta_i^{(l)}$ based on a weighted average of the error terms of the nodes that uses $a_i^{(l)}$ as an input. In detail, here is the backpropagation algorithm:

1. Perform a feedforward pass, computing the activations for layers L_2, L_3, and so on up to the output layer L_{n_l}.
2. For each output unit *i* in layer n_l (the output layer), set

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2} \left\| y - h_{W,b}(x) \right\|^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$

3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$

   For each node *i* in layer *l*, set

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$$

4. Compute the desired partial derivatives, which are given as:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x,y) = a_j^{(l)} \delta_i^{(l+1)}$$
$$\frac{\partial}{\partial b_i^{(l)}} J(W,b;x,y) = \delta_i^{(l+1)}.$$

Finally, we can also re-write the algorithm using matrix-vectorial notation. We will use "$\bullet$" to denote the element-wise product operator (denoted "`.*`" in Matlab or Octave, and also called the Hadamard product), so that if $a = b \bullet c$, then $a_i = b_i c_i$. Similar to how we extended the definition of $f(\cdot)$ to apply element-wise to vectors, we also do the same for $f'(\cdot)$ (so that $f'([z_1, z_2, z_3]) = [f'(z_1), f'(z_2), f'(z_3)]$).

The algorithm can then be written:

1. Perform a feedforward pass, computing the activations for layers L_2, L_3, up to the output layer L_{n_l}, using the equations defining the forward propagation steps
2. For the output layer (layer n_l), set
$$\delta^{(n_l)} = -(y - a^{(n_l)}) \bullet f'(z^{(n_l)})$$
3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$

   Set
$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)}\right) \bullet f'(z^{(l)})$$
4. Compute the desired partial derivatives:
$$\nabla_{W^{(l)}} J(W,b;x,y) = \delta^{(l+1)} (a^{(l)})^T,$$
$$\nabla_{b^{(l)}} J(W,b;x,y) = \delta^{(l+1)}.$$

Implementation note: In steps 2 and 3 above, we need to compute $f'(z_i^{(l)})$ for each value of i. Assuming $f(z)$ is the sigmoid activation function, we would already have $a_i^{(l)}$ stored away from the forward pass through the network. Thus, using the expression that we worked out earlier for $f'(z)$, we can compute this as $f'(z_i^{(l)}) = a_i^{(l)}(1 - a_i^{(l)})$.

Finally, we are ready to describe the full gradient descent algorithm. In the pseudo-code below, $\Delta W^{(l)}$ is a matrix (of the same dimension as $W^{(l)}$), and $\Delta b^{(l)}$ is a vector (of the same dimension as $b^{(l)}$). Note that in this notation, "$\Delta W^{(l)}$" is a matrix, and in particular it isn't "Δ times $W^{(l)}$." We implement one iteration of batch gradient descent as follows:

1. Set $\Delta W^{(l)} := 0$, $\Delta b^{(l)} := 0$ (matrix/vector of zeros) for all l.
2. For $i = 1$ to m,
   a. Use backpropagation to compute $\nabla_{W^{(l)}} J(W,b;x,y)$ and $\nabla_{b^{(l)}} J(W,b;x,y)$.
   b. Set $\Delta W^{(l)} := \Delta W^{(l)} + \nabla_{W^{(l)}} J(W,b;x,y)$.
   c. Set $\Delta b^{(l)} := \Delta b^{(l)} + \nabla_{b^{(l)}} J(W,b;x,y)$.
3. Update the parameters:

$$W^{(l)} = W^{(l)} - \alpha \left[\left(\frac{1}{m} \Delta W^{(l)} \right) + \lambda W^{(l)} \right]$$
$$b^{(l)} = b^{(l)} - \alpha \left[\frac{1}{m} \Delta b^{(l)} \right]$$

To train our neural network, we can now repeatedly take steps of gradient descent to reduce our cost function $J(W, b)$.

Neural Networks | **Backpropagation Algorithm** | Gradient checking and advanced optimization | Autoencoders and Sparsity | Visualizing a Trained Autoencoder | Sparse Autoencoder Notation Summary | Exercise:Sparse Autoencoder

Language : 中文

Gradient checking and advanced optimization

From Ufldl

Backpropagation is a notoriously difficult algorithm to debug and get right, especially since many subtly buggy implementations of it—for example, one that has an off-by-one error in the indices and that thus only trains some of the layers of weights, or an implementation that omits the bias term—will manage to learn something that can look surprisingly reasonable (while performing less well than a correct implementation). Thus, even with a buggy implementation, it may not at all be apparent that anything is amiss. In this section, we describe a method for numerically checking the derivatives computed by your code to make sure that your implementation is correct. Carrying out the derivative checking procedure described here will significantly increase your confidence in the correctness of your code.

Suppose we want to minimize $J(\theta)$ as a function of θ. For this example, suppose $J : \Re \mapsto \Re$, so that $\theta \in \Re$. In this 1-dimensional case, one iteration of gradient descent is given by

$$\theta := \theta - \alpha \frac{d}{d\theta} J(\theta).$$

Suppose also that we have implemented some function $g(\theta)$ that purportedly computes $\frac{d}{d\theta} J(\theta)$, so that we implement gradient descent using the update $\theta := \theta - \alpha g(\theta)$. How can we check if our implementation of g is correct?

Recall the mathematical definition of the derivative as

$$\frac{d}{d\theta} J(\theta) = \lim_{\epsilon \to 0} \frac{J(\theta + \epsilon) - J(\theta - \epsilon)}{2\epsilon}.$$

Thus, at any specific value of θ, we can numerically approximate the derivative as follows:

$$\frac{J(\theta + \text{EPSILON}) - J(\theta - \text{EPSILON})}{2 \times \text{EPSILON}}$$

In practice, we set EPSILON to a small constant, say around 10^{-4}. (There's a large range of values of EPSILON that should work well, but we don't set EPSILON to be "extremely" small, say 10^{-20}, as that would lead to numerical roundoff errors.)

Thus, given a function $g(\theta)$ that is supposedly computing $\frac{d}{d\theta} J(\theta)$, we can now numerically verify its correctness by checking that

$$g(\theta) \approx \frac{J(\theta + \text{EPSILON}) - J(\theta - \text{EPSILON})}{2 \times \text{EPSILON}}.$$

The degree to which these two values should approximate each other will depend on the details of J. But assuming $\text{EPSILON} = 10^{-4}$, you'll usually find that the left- and right-hand sides of the above will agree to at least 4 significant digits (and often many more).

Now, consider the case where $\theta \in \Re^n$ is a vector rather than a single real number (so that we have n parameters that we want to learn), and $J : \Re^n \mapsto \Re$. In our neural network example we used "$J(W, b)$," but one can imagine "unrolling" the parameters W, b into a long vector θ. We now generalize our derivative checking procedure to the case where θ may be a vector.

Suppose we have a function $g_i(\theta)$ that purportedly computes $\frac{\partial}{\partial \theta_i} J(\theta)$; we'd like to check if g_i is outputting correct derivative values. Let $\theta^{(i+)} = \theta + \text{EPSILON} \times \vec{e}_i$, where

$$\vec{e}_i = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

is the i-th basis vector (a vector of the same dimension as θ, with a "1" in the i-th position and "0"s everywhere else). So, $\theta^{(i+)}$ is the same as θ, except its i-th element has been incremented by EPSILON. Similarly, let $\theta^{(i-)} = \theta - \text{EPSILON} \times \vec{e}_i$ be the corresponding vector with the i-th element decreased by EPSILON. We can now numerically verify $g_i(\theta)$'s correctness by checking, for each i, that:

$$g_i(\theta) \approx \frac{J(\theta^{(i+)}) - J(\theta^{(i-)})}{2 \times \text{EPSILON}}.$$

When implementing backpropagation to train a neural network, in a correct implementation we will have that

$$\nabla_{W^{(l)}} J(W,b) = \left(\frac{1}{m} \Delta W^{(l)} \right) + \lambda W^{(l)}$$
$$\nabla_{b^{(l)}} J(W,b) = \frac{1}{m} \Delta b^{(l)}.$$

This result shows that the final block of psuedo-code in Backpropagation Algorithm is indeed implementing gradient descent. To make sure your implementation of gradient descent is correct, it is usually very helpful to use the method described above to numerically compute the derivatives of $J(W,b)$, and thereby verify that your computations of $\left(\frac{1}{m} \Delta W^{(l)} \right) + \lambda W$ and $\frac{1}{m} \Delta b^{(l)}$ are indeed giving the derivatives you want.

Finally, so far our discussion has centered on using gradient descent to minimize $J(\theta)$. If you have implemented a function that computes $J(\theta)$ and $\nabla_\theta J(\theta)$, it turns out there are more sophisticated algorithms than gradient descent for trying to minimize $J(\theta)$. For example, one can envision an algorithm that uses gradient descent, but automatically tunes the learning rate α so as to try to use a step-size that causes θ to approach a local optimum as quickly as possible. There are other algorithms that are even more sophisticated than this; for example, there are algorithms that try to find an approximation to the Hessian matrix, so that it can take more rapid steps towards a local optimum (similar to Newton's method). A full discussion of these algorithms is beyond the scope of these notes, but one example is the **L-BFGS** algorithm. (Another example is the **conjugate gradient** algorithm.) You will use one of these algorithms in the programming exercise. The main thing you need to provide to these advanced optimization algorithms is that for any θ, you have to be able to compute $J(\theta)$ and $\nabla_\theta J(\theta)$. These optimization algorithms will then do their own internal tuning of the learning rate/step-size α (and compute its own approximation to the Hessian, etc.) to automatically search for a value of θ that minimizes $J(\theta)$. Algorithms such as L-BFGS and conjugate gradient can often be much faster than gradient descent.

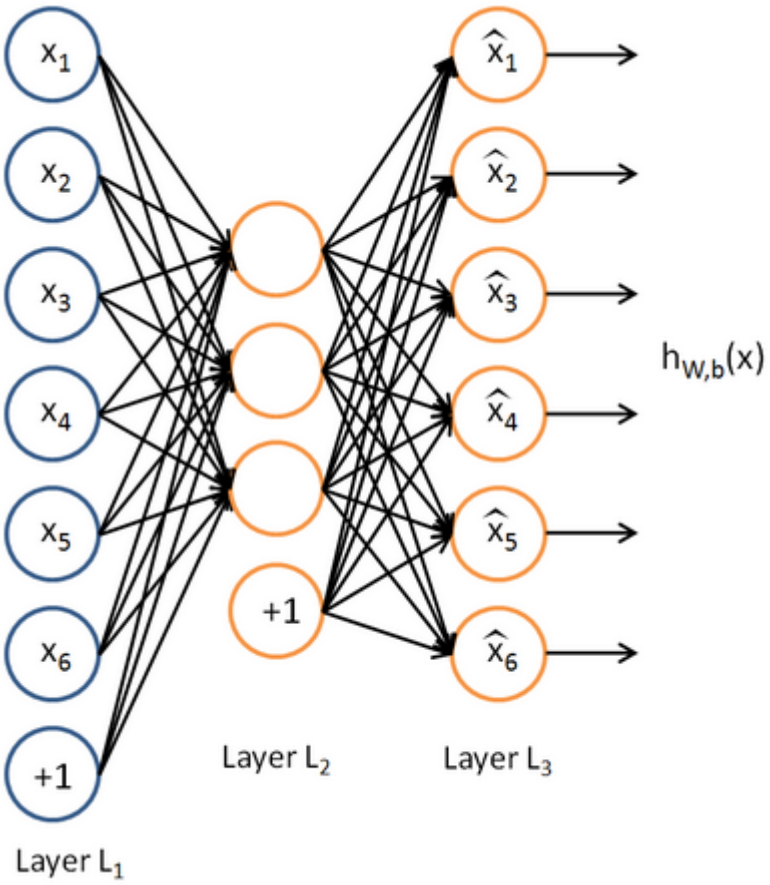
Language : 中文

Autoencoders and Sparsity

From Ufldl

So far, we have described the application of neural networks to supervised learning, in which we have labeled training examples. Now suppose we have only a set of unlabeled training examples $\{x^{(1)}, x^{(2)}, x^{(3)}, \ldots\}$, where $x^{(i)} \in \Re^n$. An **autoencoder** neural network is an unsupervised learning algorithm that applies backpropagation, setting the target values to be equal to the inputs. I.e., it uses $y^{(i)} = x^{(i)}$.

Here is an autoencoder:



The autoencoder tries to learn a function $h_{W,b}(x) \approx x$. In other words, it is trying to learn an approximation to the identity function, so as to output $\hat{x}$ that is similar to x. The identity function seems a particularly trivial function to be trying to learn; but by placing constraints on the network, such as by limiting the number of hidden units, we can discover interesting structure about the data. As a concrete example, suppose the inputs x are the pixel intensity values from a 10×10 image (100 pixels) so $n = 100$, and there are $s_2 = 50$ hidden units in layer L_2. Note that we also have $y \in \Re^{100}$. Since there are only 50 hidden units, the network is forced to learn a *compressed* representation of the input. I.e., given only the vector of hidden unit activations $a^{(2)} \in \Re^{50}$, it must try to **reconstruct** the 100-pixel input x. If the input were completely random---say, each x_i comes from an IID Gaussian independent of the other features---then this compression task would be very difficult. But if there is structure in the data, for example, if some of the input features are correlated, then this algorithm will be able to discover some of those correlations. In fact, this simple autoencoder often ends up learning a low-dimensional representation very similar to PCAs.

Our argument above relied on the number of hidden units s_2 being small. But even when the number of hidden units is large (perhaps even greater than the number of input pixels), we can still discover interesting structure, by imposing other constraints on the network. In particular, if we impose a **sparsity** constraint on the hidden

units, then the autoencoder will still discover interesting structure in the data, even if the number of hidden units is large.

Informally, we will think of a neuron as being "active" (or as "firing") if its output value is close to 1, or as being "inactive" if its output value is close to 0. We would like to constrain the neurons to be inactive most of the time. This discussion assumes a sigmoid activation function. If you are using a tanh activation function, then we think of a neuron as being inactive when it outputs values close to -1.

Recall that $a_j^{(2)}$ denotes the activation of hidden unit j in the autoencoder. However, this notation doesn't make explicit what was the input x that led to that activation. Thus, we will write $a_j^{(2)}(x)$ to denote the activation of this hidden unit when the network is given a specific input x. Further, let

$$\hat{\rho}_j = \frac{1}{m} \sum_{i=1}^{m} \left[a_j^{(2)}(x^{(i)}) \right]$$

be the average activation of hidden unit j (averaged over the training set). We would like to (approximately) enforce the constraint

$$\hat{\rho}_j = \rho,$$

where ρ is a **sparsity parameter**, typically a small value close to zero (say $\rho = 0.05$). In other words, we would like the average activation of each hidden neuron j to be close to 0.05 (say). To satisfy this constraint, the hidden unit's activations must mostly be near 0.

To achieve this, we will add an extra penalty term to our optimization objective that penalizes $\hat{\rho}_j$ deviating significantly from ρ. Many choices of the penalty term will give reasonable results. We will choose the following:

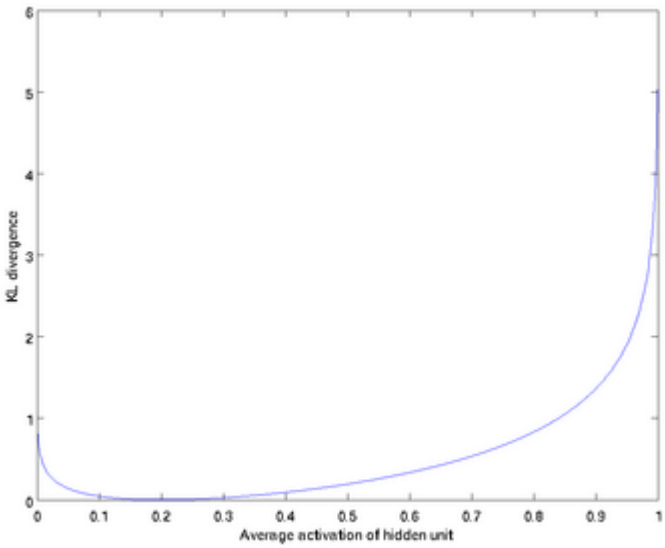
$$\sum_{j=1}^{s_2} \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j}.$$

Here, s_2 is the number of neurons in the hidden layer, and the index j is summing over the hidden units in our network. If you are familiar with the concept of KL divergence, this penalty term is based on it, and can also be written

$$\sum_{j=1}^{s_2} \mathrm{KL}(\rho || \hat{\rho}_j),$$

where $\mathrm{KL}(\rho || \hat{\rho}_j) = \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j}$ is the Kullback-Leibler (KL) divergence between a Bernoulli random variable with mean ρ and a Bernoulli random variable with mean $\hat{\rho}_j$. KL-divergence is a standard function for measuring how different two different distributions are. (If you've not seen KL-divergence before, don't worry about it; everything you need to know about it is contained in these notes.)

This penalty function has the property that $\mathrm{KL}(\rho || \hat{\rho}_j) = 0$ if $\hat{\rho}_j = \rho$, and otherwise it increases monotonically as $\hat{\rho}_j$ diverges from ρ. For example, in the figure below, we have set $\rho = 0.2$, and plotted $\mathrm{KL}(\rho || \hat{\rho}_j)$ for a range of values of $\hat{\rho}_j$:



We see that the KL-divergence reaches its minimum of 0 at $\hat{\rho}_j = \rho$, and blows up (it actually approaches ∞) as $\hat{\rho}_j$ approaches 0 or 1. Thus, minimizing this penalty term has the effect of causing $\hat{\rho}_j$ to be close to ρ.

Our overall cost function is now

$$J_{\text{sparse}}(W,b) = J(W,b) + \beta \sum_{j=1}^{s_2} \text{KL}(\rho || \hat{\rho}_j),$$

where $J(W,b)$ is as defined previously, and β controls the weight of the sparsity penalty term. The term $\hat{\rho}_j$ (implicitly) depends on W,b also, because it is the average activation of hidden unit j, and the activation of a hidden unit depends on the parameters W,b.

To incorporate the KL-divergence term into your derivative calculation, there is a simple-to-implement trick involving only a small change to your code. Specifically, where previously for the second layer ($l = 2$), during backpropagation you would have computed

$$\delta_i^{(2)} = \left(\sum_{j=1}^{s_2} W_{ji}^{(2)} \delta_j^{(3)} \right) f'(z_i^{(2)}),$$

now instead compute

$$\delta_i^{(2)} = \left(\left(\sum_{j=1}^{s_2} W_{ji}^{(2)} \delta_j^{(3)} \right) + \beta \left(-\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i} \right) \right) f'(z_i^{(2)}).$$

One subtlety is that you'll need to know $\hat{\rho}_i$ to compute this term. Thus, you'll need to compute a forward pass on all the training examples first to compute the average activations on the training set, before computing backpropagation on any example. If your training set is small enough to fit comfortably in computer memory (this will be the case for the programming assignment), you can compute forward passes on all your examples and keep the resulting activations in memory and compute the $\hat{\rho}_i$s. Then you can use your precomputed activations to perform backpropagation on all your examples. If your data is too large to fit in memory, you may have to scan through your examples computing a forward pass on each to accumulate (sum up) the activations and compute $\hat{\rho}_i$ (discarding the result of each forward pass after you have taken its activations $a_i^{(2)}$ into account for computing $\hat{\rho}_i$). Then after having computed $\hat{\rho}_i$, you'd have to redo the forward pass for each

example so that you can do backpropagation on that example. In this latter case, you would end up computing a forward pass twice on each example in your training set, making it computationally less efficient.

The full derivation showing that the algorithm above results in gradient descent is beyond the scope of these notes. But if you implement the autoencoder using backpropagation modified this way, you will be performing gradient descent exactly on the objective $J_{\text{sparse}}(W, b)$. Using the derivative checking method, you will be able to verify this for yourself as well.

Neural Networks | Backpropagation Algorithm | Gradient checking and advanced optimization | **Autoencoders and Sparsity** | Visualizing a Trained Autoencoder | Sparse Autoencoder Notation Summary | Exercise:Sparse Autoencoder

Language : 中文

Visualizing a Trained Autoencoder

From Ufldl

Having trained a (sparse) autoencoder, we would now like to visualize the function learned by the algorithm, to try to understand what it has learned. Consider the case of training an autoencoder on 10×10 images, so that $n = 100$. Each hidden unit i computes a function of the input:

$$a_i^{(2)} = f\left(\sum_{j=1}^{100} W_{ij}^{(1)} x_j + b_i^{(1)}\right).$$

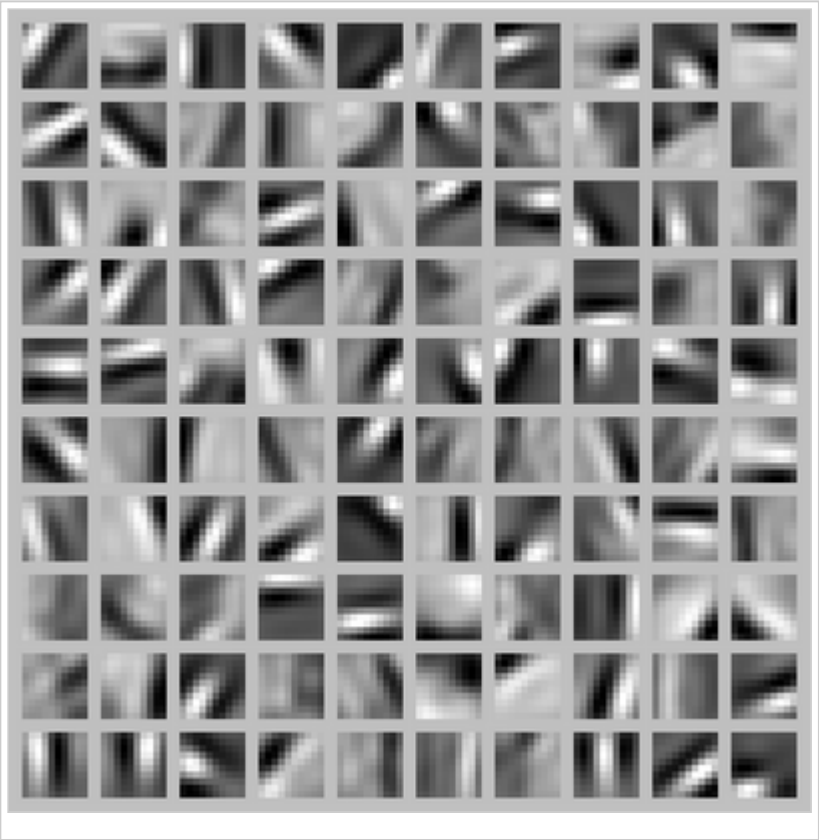
We will visualize the function computed by hidden unit i---which depends on the parameters $W_{ij}^{(1)}$ (ignoring the bias term for now)---using a 2D image. In particular, we think of $a_i^{(2)}$ as some non-linear feature of the input x. We ask: What input image x would cause $a_i^{(2)}$ to be maximally activated? (Less formally, what is the feature that hidden unit i is looking for?) For this question to have a non-trivial answer, we must impose some constraints on x. If we suppose that the input is norm constrained by $||x||^2 = \sum_{i=1}^{100} x_i^2 \leq 1$, then one can show (try doing this yourself) that the input which maximally activates hidden unit i is given by setting pixel x_j (for all 100 pixels, $j = 1, \ldots, 100$) to

$$x_j = \frac{W_{ij}^{(1)}}{\sqrt{\sum_{j=1}^{100} (W_{ij}^{(1)})^2}}.$$

By displaying the image formed by these pixel intensity values, we can begin to understand what feature hidden unit i is looking for.

If we have an autoencoder with 100 hidden units (say), then we our visualization will have 100 such images---one per hidden unit. By examining these 100 images, we can try to understand what the ensemble of hidden units is learning.

When we do this for a sparse autoencoder (trained with 100 hidden units on 10x10 pixel inputs[1] we get the following result:



Each square in the figure above shows the (norm bounded) input image x that maximally actives one of 100 hidden units. We see that the different hidden units have learned to detect edges at different positions and orientations in the image.

These features are, not surprisingly, useful for such tasks as object recognition and other vision tasks. When applied to other input domains (such as audio), this algorithm also learns useful representations/features for those domains too.

---

[1] *The learned features were obtained by training on **whitened** natural images. Whitening is a preprocessing step which removes redundancy in the input, by causing adjacent pixels to become less correlated.*

Neural Networks | Backpropagation Algorithm | Gradient checking and advanced optimization | Autoencoders and Sparsity | **Visualizing a Trained Autoencoder** | Sparse Autoencoder Notation Summary | Exercise:Sparse Autoencoder

Language : 中文

Sparse Autoencoder Notation Summary

From Ufldl

Here is a summary of the symbols used in our derivation of the sparse autoencoder:

Symbol	Meaning
x	Input features for a training example, $x \in \Re^n$.
y	Output/target values. Here, y can be vector valued. In the case of an autoencoder, $y = x$.
$(x^{(i)}, y^{(i)})$	The i-th training example
$h_{W,b}(x)$	Output of our hypothesis on input x, using parameters W, b. This should be a vector of the same dimension as the target value y.
$W_{ij}^{(l)}$	The parameter associated with the connection between unit j in layer l, and unit i in layer $l+1$.
$b_i^{(l)}$	The bias term associated with unit i in layer $l+1$. Can also be thought of as the parameter associated with the connection between the bias unit in layer l and unit i in layer $l+1$.
θ	Our parameter vector. It is useful to think of this as the result of taking the parameters W, b and ``unrolling *them into a long column vector.*
$a_i^{(l)}$	Activation (output) of unit i in layer l of the network. In addition, since layer L_1 is the input layer, we also have $a_i^{(1)} = x_i$.
$f(\cdot)$	The activation function. Throughout these notes, we used $f(z) = \tanh(z)$.
$z_i^{(l)}$	Total weighted sum of inputs to unit i in layer l. Thus, $a_i^{(l)} = f(z_i^{(l)})$.
α	Learning rate parameter
s_l	Number of units in layer l (not counting the bias unit).
n_l	Number layers in the network. Layer L_1 is usually the input layer, and layer L_{n_l} the output layer.
λ	Weight decay parameter.
$\hat{x}$	For an autoencoder, its output; i.e., its reconstruction of the input x. Same meaning as $h_{W,b}(x)$.
ρ	Sparsity parameter, which specifies our desired level of sparsity
$\hat{\rho}_i$	The average activation of hidden unit i (in the sparse autoencoder).
β	Weight of the sparsity penalty term (in the sparse autoencoder objective).

Neural Networks | Backpropagation Algorithm | Gradient checking and advanced optimization | Autoencoders and Sparsity | Visualizing a Trained Autoencoder | **Sparse Autoencoder Notation Summary** | Exercise:Sparse Autoencoder

Language : 中文

Exercise:Sparse Autoencoder

From Ufldl

Contents

Download Related Reading

- sparseae_reading.pdf (http://nlp.stanford.edu/~socherr/sparseAutoencoder_2011new.pdf)
- sparseae_exercise.pdf (http://www.stanford.edu/class/cs294a/cs294a_2011-assignment.pdf)

Sparse autoencoder implementation

In this problem set, you will implement the sparse autoencoder algorithm, and show how it discovers that edges are a good representation for natural images. (Images provided by Bruno Olshausen.) The sparse autoencoder algorithm is described in the lecture notes found on the course website.

In the file sparseae_exercise.zip (http://ufldl.stanford.edu/wiki/resources/sparseae_exercise.zip) , we have provided some starter code in Matlab. You should write your code at the places indicated in the files ("`YOUR CODE HERE`"). You have to complete the following files: `sampleIMAGES.m, sparseAutoencoderCost.m, computeNumericalGradient.m`. The starter code in `train.m` shows how these functions are used.

Specifically, in this exercise you will implement a sparse autoencoder, trained with 8×8 image patches using the L-BFGS optimization algorithm.

A note on the software: The provided .zip file includes a subdirectory `minFunc` with 3rd party software implementing L-BFGS, that is licensed under a Creative Commons, Attribute, Non-Commercial license. If you need to use this software for commercial purposes, you can download and use a different function (fminlbfgs) that can serve the same purpose, but runs ~3x slower for this exercise (and thus is less recommended). You can read more about this in the Fminlbfgs_Details page.

Step 1: Generate training set

The first step is to generate a training set. To get a single training example *x*, randomly pick one of the 10 images, then randomly sample an 8×8 image patch from the selected image, and convert the image patch (either in row-major order or column-major order; it doesn't matter) into a 64-dimensional vector to get a training example $x \in \Re^{64}$.

Complete the code in `sampleIMAGES.m`. Your code should sample 10000 image patches and concatenate them into a 64×10000 matrix.

To make sure your implementation is working, run the code in "Step 1" of `train.m`. This should result in a plot of a random sample of 200 patches from the dataset.

Implementational tip: When we run our implemented `sampleImages()`, it takes under 5 seconds. If your implementation takes over 30 seconds, it may be because you are accidentally making a copy of an entire 512×512 image each time you're picking a random image. By copying a 512×512 image 10000 times, this can make your implementation much less efficient. While this doesn't slow down your code significantly for this exercise (because we have only 10000 examples), when we scale to much larger problems later this quarter with 10^6 or more examples, this will significantly slow down your code. Please implement `sampleIMAGES` so that you aren't making a copy of an entire 512×512 image each time you need to cut out an 8x8 image patch.

Step 2: Sparse autoencoder objective

Implement code to compute the sparse autoencoder cost function $J_{\text{sparse}}(W,b)$ (Section 3 of the lecture notes) and the corresponding derivatives of J_{sparse} with respect to the different parameters. Use the sigmoid function for the activation function, $f(z) = \frac{1}{1+e^{-z}}$. In particular, complete the code in `sparseAutoencoderCost.m`.

The sparse autoencoder is parameterized by matrices $W^{(1)} \in \Re^{s_1 \times s_2}$, $W^{(2)} \in \Re^{s_2 \times s_3}$ vectors $b^{(1)} \in \Re^{s_2}$, $b^{(2)} \in \Re^{s_3}$. However, for subsequent notational convenience, we will "unroll" all of these parameters into a very long parameter vector θ with $s_1 s_2 + s_2 s_3 + s_2 + s_3$ elements. The code for converting between the $(W^{(1)},W^{(2)},b^{(1)},b^{(2)})$ and the θ parameterization is already provided in the starter code.

Implementational tip: The objective $J_{\text{sparse}}(W,b)$ contains 3 terms, corresponding to the squared error term, the weight decay term, and the sparsity penalty. You're welcome to implement this however you want, but for ease of debugging, you might implement the cost function and derivative computation (backpropagation) only for the squared error term first (this corresponds to setting $\lambda = \beta = 0$), and implement the gradient checking method in the next section to first verify that this code is correct. Then only after you have verified that the objective and derivative calculations corresponding to the squared error term are working, add in code to compute the weight decay and sparsity penalty terms and their corresponding derivatives.

Step 3: Gradient checking

Following Section 2.3 of the lecture notes, implement code for gradient checking. Specifically, complete the code in `computeNumericalGradient.m`. Please use `EPSILON` = 10^{-4} as described in the lecture notes.

We've also provided code in `checkNumericalGradient.m` for you to test your code. This code defines a simple quadratic function $h : \Re^2 \mapsto \Re$ given by $h(x) = x_1^2 + 3x_1x_2$, and evaluates it at the point $x = (4,10)^T$. It allows you to verify that your numerically evaluated gradient is very close to the true (analytically computed) gradient.

After using `checkNumericalGradient.m` to make sure your implementation is correct, next use `computeNumericalGradient.m` to make sure that your `sparseAutoencoderCost.m` is computing derivatives correctly. For details, see Steps 3 in `train.m`. We strongly encourage you not to proceed to the next step until you've verified that your derivative computations are correct.

Implementational tip: If you are debugging your code, performing gradient checking on smaller models and smaller training sets (e.g., using only 10 training examples and 1-2 hidden units) may speed things up.

Step 4: Train the sparse autoencoder

Now that you have code that computes J_{sparse} and its derivatives, we're ready to minimize J_{sparse} with respect to its parameters, and thereby train our sparse autoencoder.

We will use the L-BFGS algorithm. This is provided to you in a function called `minFunc` (code provided by Mark Schmidt) included in the starter code. (For the purpose of this assignment, you only need to call minFunc with the default parameters. You do not need to know how L-BFGS works.) We have already provided code in `train.m` (Step 4) to call `minFunc`. The `minFunc` code assumes that the parameters to be optimized are a long parameter vector; so we will use the "θ" parameterization rather than the "$(W^{(1)},W^{(2)},b^{(1)},b^{(2)})$" parameterization when passing our parameters to it.

Train a sparse autoencoder with 64 input units, 25 hidden units, and 64 output units. In our starter code, we have provided a function for initializing the parameters. We initialize the biases $b_i^{(l)}$ to zero, and the weights $W_{ij}^{(l)}$ to random numbers drawn uniformly from the interval $\left[-\sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}+1}}, \sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}+1}}\right]$, where n_{in} is the fan-in (the number of inputs feeding into a node) and n_{out} is the fan-in (the number of units that a node feeds into).

The values we provided for the various parameters (λ,β,ρ, etc.) should work, but feel free to play with different settings of the parameters as well.

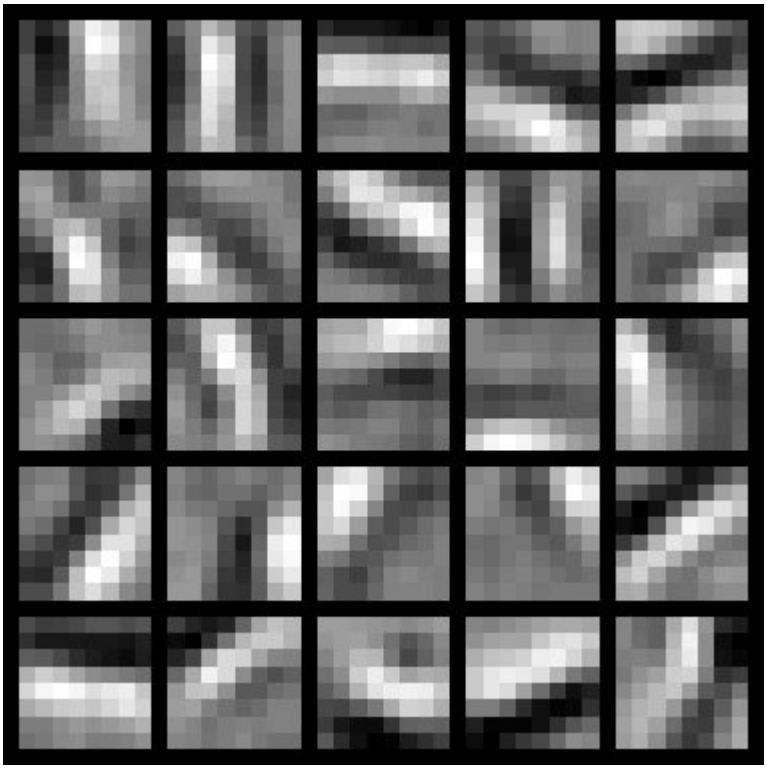
Implementational tip: Once you have your backpropagation implementation correctly computing the derivatives (as verified using gradient checking in Step 3), when you are now using it with L-BFGS to optimize $J_{\text{sparse}}(W,b)$, make sure you're not doing gradient-checking on every step. Backpropagation can be used to compute the derivatives of $J_{\text{sparse}}(W,b)$ fairly efficiently, and if you were additionally computing the gradient numerically on every step, this would slow down your program significantly.

Step 5: Visualization

After training the autoencoder, use `display_network.m` to visualize the learned weights. (See `train.m`, Step 5.) Run "`print -djpeg weights.jpg`" to save the visualization to a file "`weights.jpg`" (which you will submit together with your code).

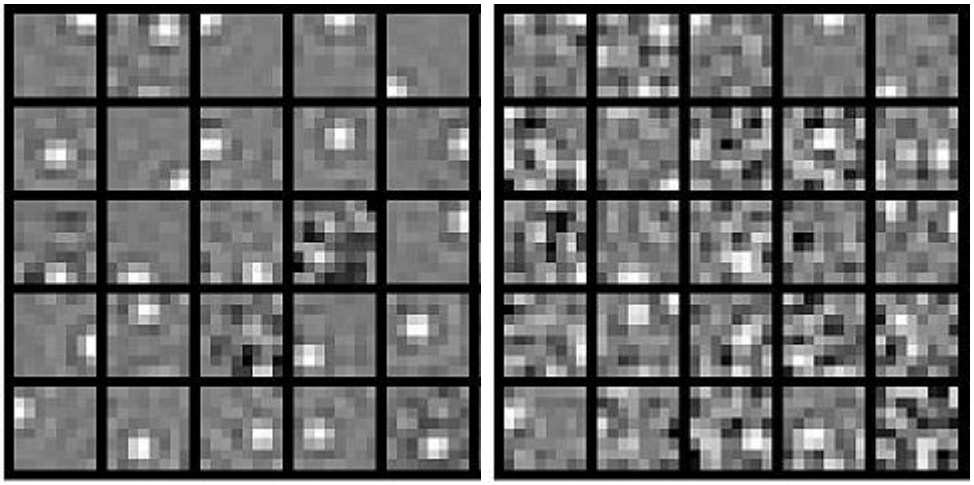
Results

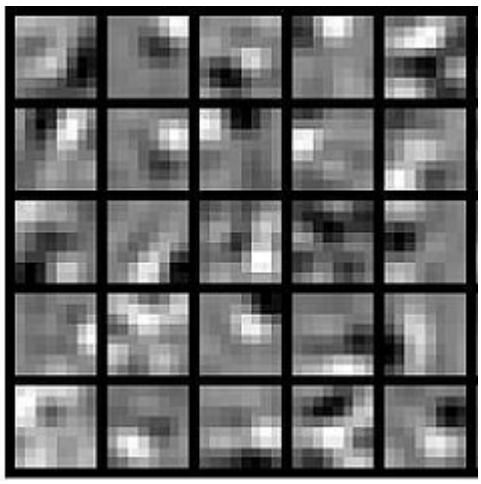
To successfully complete this assignment, you should demonstrate your sparse autoencoder algorithm learning a set of edge detectors. For example, this was the visualization we obtained:

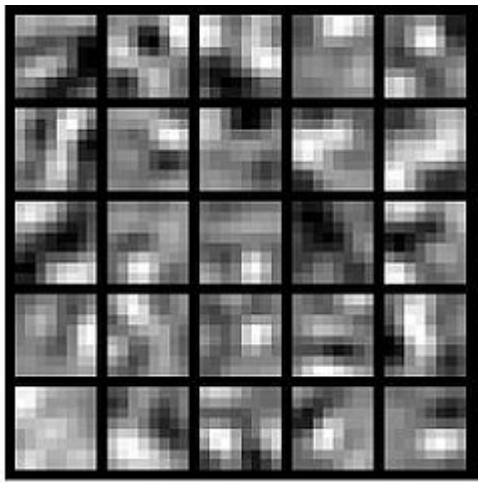


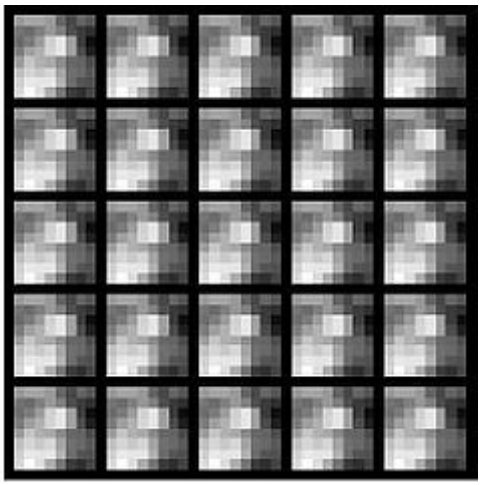
Our implementation took around 5 minutes to run on a fast computer. In case you end up needing to try out multiple implementations or different parameter values, be sure to budget enough time for debugging and to run the experiments you'll need.

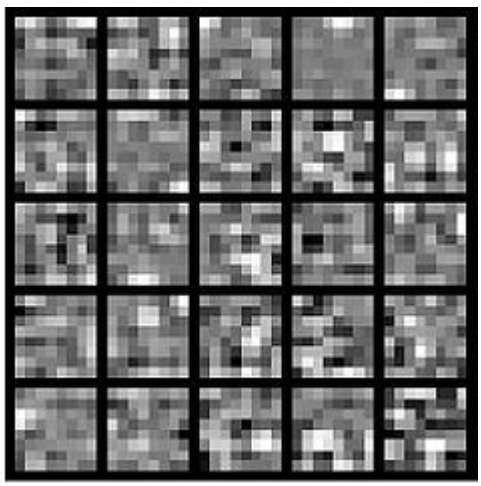
Also, by way of comparison, here are some visualizations from implementations that we do not consider successful (either a buggy implementation, or where the parameters were poorly tuned):











Neural Networks | Backpropagation Algorithm | Gradient checking and advanced optimization | Autoencoders and Sparsity | Visualizing a Trained Autoencoder | Sparse Autoencoder Notation Summary | **Exercise:Sparse Autoencoder**

Vectorization

From Ufldl

When working with learning algorithms, having a faster piece of code often means that you'll make progress faster on your project. For example, if your learning algorithm takes 20 minutes to run to completion, that means you can "try" up to 3 new ideas per hour. But if your code takes 20 hours to run, that means you can "try" only one idea a day, since that's how long you have to wait to get feedback from your program. In this latter case, if you can speed up your code so that it takes only 10 hours to run, that can literally double your personal productivity!

Vectorization refers to a powerful way to speed up your algorithms. Numerical computing and parallel computing researchers have put decades of work into making certain numerical operations (such as matrix-matrix multiplication, matrix-matrix addition, matrix-vector multiplication) fast. The idea of vectorization is that we would like to express our learning algorithms in terms of these highly optimized operations.

For example, if $x \in \Re^{n+1}$ and $\theta \in \Re^{n+1}$ are vectors and you need to compute $z = \theta^T x$, you can implement (in Matlab):

```
z = 0;
for i=1:(n+1),
  z = z + theta(i) * x(i);
end;
```

or you can more simply implement

```
z = theta' * x;
```

The second piece of code is not only simpler, but it will also run *much* faster.

More generally, a good rule-of-thumb for coding Matlab/Octave is:

Whenever possible, avoid using explicit for-loops in your code.

In particular, the first code example used an explicit `for` loop. By implementing the same functionality without the `for` loop, we sped it up significantly. A large part of vectorizing our Matlab/Octave code will focus on getting rid of `for` loops, since this lets Matlab/Octave extract more parallelism from your code, while also incurring less computational overhead from the interpreter.

In terms of a strategy for writing your code, initially you may find that vectorized code is harder to write, read, and/or debug, and that there may be a tradeoff in ease of programming/debugging vs. running time. Thus, for your first few programs, you might choose to first implement your algorithm without too many vectorization tricks, and verify that it is working correctly (perhaps by running on a small problem). Then only after it is working, you can vectorize your code one piece at a time, pausing after each piece to verify that your code is still computing the same result as before. At the end, you'll then hopefully have a correct, debugged, and vectorized/efficient piece of code.

After you become familiar with the most common vectorization methods and tricks, you'll find that it usually isn't much effort to vectorize your code. Doing so will make your code run much faster and, in some cases, simplify it too.

Language : 中文

Logistic Regression Vectorization Example

From Ufldl

Consider training a logistic regression model using batch gradient ascent. Suppose our hypothesis is

$$h_\theta(x) = \frac{1}{1+\exp(-\theta^T x)},$$

where (following the notational convention from the OpenClassroom videos and from CS229) we let $x_0 = 1$, so that $x \in \Re^{n+1}$ and $\theta \in \Re^{n+1}$, and θ_0 is our intercept term. We have a training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ of m examples, and the batch gradient ascent update rule is $\theta := \theta + \alpha \nabla_\theta \ell(\theta)$, where $\ell(\theta)$ is the log likelihood and $\nabla_\theta \ell(\theta)$ is its derivative.

[Note: Most of the notation below follows that defined in the OpenClassroom videos or in the class CS229: Machine Learning. For details, see either the OpenClassroom videos (http://openclassroom.stanford.edu/MainFolder/CoursePage.php?course=MachineLearning) or Lecture Notes #1 of http://cs229.stanford.edu/ .]

We thus need to compute the gradient:

$$\nabla_\theta \ell(\theta) = \sum_{i=1}^{m} \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)}.$$

Suppose that the Matlab/Octave variable `x` is a matrix containing the training inputs, so that `x(:,i)` is the i-th training example $x^{(i)}$, and `x(j,i)` is $x_j^{(i)}$. Further, suppose the Matlab/Octave variable `y` is a *row* vector of the labels in the training set, so that the variable `y(i)` is $y^{(i)} \in \{0, 1\}$. (Here we differ from the OpenClassroom/CS229 notation. Specifically, in the matrix-valued `x` we stack the training inputs in columns rather than in rows; and `y` $\in \Re^{1\times m}$ is a row vector rather than a column vector.)

Here's truly horrible, extremely slow, implementation of the gradient computation:

```
% Implementation 1
grad = zeros(n+1,1);
for i=1:m,
  h = sigmoid(theta'*x(:,i));
  temp = y(i) - h;
  for j=1:n+1,
    grad(j) = grad(j) + temp * x(j,i);
  end;
end;
```

The two nested for-loops makes this very slow. Here's a more typical implementation, that partially vectorizes the algorithm and gets better performance:

```
% Implementation 2
grad = zeros(n+1,1);
for i=1:m,
  grad = grad + (y(i) - sigmoid(theta'*x(:,i)))* x(:,i);
end;
```

However, it turns out to be possible to even further vectorize this. If we can get rid of the for-loop, we can significantly speed up the implementation. In particular, suppose `b` is a column vector, and `A` is a matrix. Consider the following ways of computing `A * b`:

```
% Slow implementation of matrix-vector multiply
grad = zeros(n+1,1);
for i=1:m,
  grad = grad + b(i) * A(:,i);  % more commonly written A(:,i)*b(i)
end;

% Fast implementation of matrix-vector multiply
grad = A*b;
```

We recognize that Implementation 2 of our gradient descent calculation above is using the slow version with a for-loop, with `b(i)` playing the role of `(y(i) - sigmoid(theta'*x(:,i)))`, and `A` playing the role of `x`. We can derive a fast implementation as follows:

```
% Implementation 3
grad = x * (y- sigmoid(theta'*x))';
```

Here, we assume that the Matlab/Octave `sigmoid(z)` takes as input a vector `z`, applies the sigmoid function component-wise to the input, and returns the result. The output of `sigmoid(z)` is therefore itself also a vector, of the same dimension as the input `z`

When the training set is large, this final implementation takes the greatest advantage of Matlab/Octave's highly optimized numerical linear algebra libraries to carry out the matrix-vector operations, and so this is far more efficient than the earlier implementations.

Coming up with vectorized implementations isn't always easy, and sometimes requires careful thought. But as you gain familiarity with vectorized operations, you'll find that there are design patterns (i.e., a small number of ways of vectorizing) that apply to many different pieces of code.

Vectorization | **Logistic Regression Vectorization Example** | Neural Network Vectorization | Exercise:Vectorization

Neural Network Vectorization

From Ufldl

In this section, we derive a vectorized version of our neural network. In our earlier description of Neural Networks, we had already given a partially vectorized implementation, that is quite efficient if we are working with only a single example at a time. We now describe how to implement the algorithm so that it simultaneously processes multiple training examples. Specifically, we will do this for the forward propagation and backpropagation steps, as well as for learning a sparse set of features.

Forward propagation

Consider a 3 layer neural network (with one input, one hidden, and one output layer), and suppose `x` is a column vector containing a single training example $x^{(i)} \in \Re^n$. Then the forward propagation step is given by:

$$
\begin{aligned}
z^{(2)} &= W^{(1)}x + b^{(1)} \\
a^{(2)} &= f(z^{(2)}) \\
z^{(3)} &= W^{(2)}a^{(2)} + b^{(2)} \\
h_{W,b}(x) &= a^{(3)} = f(z^{(3)})
\end{aligned}
$$

This is a fairly efficient implementation for a single example. If we have *m* examples, then we would wrap a `for` loop around this.

Concretely, following the Logistic Regression Vectorization Example, let the Matlab/Octave variable `x` be a matrix containing the training inputs, so that `x(:,i)` is the i-th training example. We can then implement forward propagation as:

```
% Unvectorized implementation
for i=1:m,
  z2 = W1 * x(:,i) + b1;
  a2 = f(z2);
  z3 = W2 * a2 + b2;
  h(:,i) = f(z3);
end;
```

Can we get rid of the `for` loop? For many algorithms, we will represent intermediate stages of computation via vectors. For example, `z2`, `a2`, and `z3` here are all column vectors that're used to compute the activations of the hidden and output layers. In order to take better advantage of parallelism and efficient matrix operations, we would like to *have our algorithm operate simultaneously on many training examples*. Let us temporarily ignore `b1` and `b2` (say, set them to zero for now). We can then implement the following:

```
% Vectorized implementation (ignoring b1, b2)
z2 = W1 * x;
a2 = f(z2);
z3 = W2 * a2;
h = f(z3)
```

In this implementation, `z2`, `a2`, and `z3` are all matrices, with one column per training example. A common design pattern in vectorizing across training examples is that whereas previously we had a column vector (such as `z2`) per training example, we can often instead try to compute a matrix so that all of these column vectors are stacked together to form a matrix. Concretely, in this example, `a2` becomes a s_2 by *m* matrix (where s_2 is the

number of units in layer 2 of the network, and m is the number of training examples). And, the i-th column of `a2` contains the activations of the hidden units (layer 2 of the network) when the i-th training example `x(:,i)` is input to the network.

In the implementation above, we have assumed that the activation function `f(z)` takes as input a matrix `z`, and applies the activation function component-wise to the input. Note that your implementation of `f(z)` should use Matlab/Octave's matrix operations as much as possible, and avoid `for` loops as well. We illustrate this below, assuming that `f(z)` is the sigmoid activation function:

```
% Inefficient, unvectorized implementation of the activation function
function output = unvectorized_f(z)
output = zeros(size(z))
for i=1:size(z,1),
  for j=1:size(z,2),
    output(i,j) = 1/(1+exp(-z(i,j)));
  end;
end;
end

% Efficient, vectorized implementation of the activation function
function output = vectorized_f(z)
output = 1./(1+exp(-z));     % "./" is Matlab/Octave's element-wise division operator.
end
```

Finally, our vectorized implementation of forward propagation above had ignored `b1` and `b2`. To incorporate those back in, we will use Matlab/Octave's built-in `repmat` function. We have:

```
% Vectorized implementation of forward propagation
z2 = W1 * x + repmat(b1,1,m);
a2 = f(z2);
z3 = W2 * a2 + repmat(b2,1,m);
h = f(z3)
```

The result of `repmat(b1,1,m)` is a matrix formed by taking the column vector `b1` and stacking m copies of them in columns as follows

$$\begin{bmatrix} | & | & & | \\ \text{b1} & \text{b1} & \cdots & \text{b1} \\ | & | & & | \end{bmatrix}.$$

This forms a s_2 by m matrix. Thus, the result of adding this to `W1 * x` is that each column of the matrix gets `b1` added to it, as desired. See Matlab/Octave's documentation (type "`help repmat`") for more information. As a Matlab/Octave built-in function, `repmat` is very efficient as well, and runs much faster than if you were to implement the same thing yourself using a `for` loop.

Backpropagation

We now describe the main ideas behind vectorizing backpropagation. Before reading this section, we strongly encourage you to carefully step through all the forward propagation code examples above to make sure you fully understand them. In this text, we'll only sketch the details of how to vectorize backpropagation, and leave you to derive the details in the Vectorization exercise.

We are in a supervised learning setting, so that we have a training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ of m training examples. (For the autoencoder, we simply set $y^{(i)} = x^{(i)}$, but our derivation here will consider this more general setting.)

Suppose we have s_3 dimensional outputs, so that our target labels are $y^{(i)} \in \Re^{s3}$. In our Matlab/Octave datastructure, we will stack these in columns to form a Matlab/Octave variable `y`, so that the *i*-th column `y(:,i)` is $y^{(i)}$.

We now want to compute the gradient terms $\nabla_{W^{(l)}} J(W,b)$ and $\nabla_{b^{(l)}} J(W,b)$. Consider the first of these terms. Following our earlier description of the Backpropagation Algorithm, we had that for a single training example (*x*,*y*), we can compute the derivatives as

$$\begin{aligned}
\delta^{(3)} &= -(y - a^{(3)}) \bullet f'(z^{(3)}), \\
\delta^{(2)} &= ((W^{(2)})^T \delta^{(3)}) \bullet f'(z^{(2)}), \\
\nabla_{W^{(2)}} J(W,b;x,y) &= \delta^{(3)} (a^{(2)})^T, \\
\nabla_{W^{(1)}} J(W,b;x,y) &= \delta^{(2)} (a^{(1)})^T.
\end{aligned}$$

Here, $\bullet$ denotes element-wise product. For simplicity, our description here will ignore the derivatives with respect to $b^{(l)}$, though your implementation of backpropagation will have to compute those derivatives too.

Suppose we have already implemented the vectorized forward propagation method, so that the matrix-valued `z2`, `a2`, `z3` and `h` are computed as described above. We can then implement an *unvectorized* version of backpropagation as follows:

```
gradW1 = zeros(size(W1));
gradW2 = zeros(size(W2));
for i=1:m,
  delta3 = -(y(:,i) - h(:,i)) .* fprime(z3(:,i));
  delta2 = W2'*delta3(:,i) .* fprime(z2(:,i));

  gradW2 = gradW2 + delta3*a2(:,i)';
  gradW1 = gradW1 + delta2*a1(:,i)';
end;
```

This implementation has a `for` loop. We would like to come up with an implementation that simultaneously performs backpropagation on all the examples, and eliminates this `for` loop.

To do so, we will replace the vectors `delta3` and `delta2` with matrices, where one column of each matrix corresponds to each training example. We will also implement a function `fprime(z)` that takes as input a matrix `z`, and applies $f'(\cdot)$ element-wise. Each of the four lines of Matlab in the `for` loop above can then be vectorized and replaced with a single line of Matlab code (without a surrounding `for` loop).

In the Vectorization exercise, we ask you to derive the vectorized version of this algorithm by yourself. If you are able to do it from this description, we strongly encourage you to do so. Here also are some Backpropagation vectorization hints; however, we encourage you to try to carry out the vectorization yourself without looking at the hints.

Sparse autoencoder

The sparse autoencoder neural network has an additional sparsity penalty that constrains neurons' average firing rate to be close to some target activation ρ. When performing backpropagation on a single training example, we had taken into the account the sparsity penalty by computing the following:

$$\delta_i^{(2)} = \left(\left(\sum_{j=1}^{s_2} W_{ji}^{(2)} \delta_j^{(3)} \right) + \beta \left(-\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i} \right) \right) f'(z_i^{(2)}).$$

In the *unvectorized* case, this was computed as:

```
% Sparsity Penalty Delta
sparsity_delta = - rho ./ rho_hat + (1 - rho) ./ (1 - rho_hat);
for i=1:m,
  ...
  delta2 = (W2'*delta3(:,i) + beta*sparsity_delta).* fprime(z2(:,i));
  ...
end;
```

The code above still had a `for` loop over the training set, and `delta2` was a column vector.

In contrast, recall that in the vectorized case, `delta2` is now a matrix with *m* columns corresponding to the *m* training examples. Now, notice that the `sparsity_delta` term is the same regardless of what training example we are processing. This suggests that vectorizing the computation above can be done by simply adding the same value to each column when constructing the `delta2` matrix. Thus, to vectorize the above computation, we can simply add `sparsity_delta` (e.g., using `repmat`) to each column of `delta2`.

Vectorization | Logistic Regression Vectorization Example | **Neural Network Vectorization** | Exercise:Vectorization

Language : 中文

Exercise:Vectorization

From Ufldl

Contents

Vectorization

In the previous problem set, we implemented a sparse autoencoder for patches taken from natural images. In this problem set, you will vectorize your code to make it run much faster, and further adapt your sparse autoencoder to work on images of handwritten digits. Your network for learning from handwritten digits will be much larger than the one you'd trained on the natural images, and so using the original implementation would have been painfully slow. But with a vectorized implementation of the autoencoder, you will be able to get this to run in a reasonable amount of computation time.

Support Code/Data

The following additional files are required for this exercise:

- MNIST Dataset (Training Images) (http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz)
- MNIST Dataset (Training Labels) (http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz)
- Support functions for loading MNIST in Matlab

Step 1: Vectorize your Sparse Autoencoder Implementation

Using the ideas from Vectorization and Neural Network Vectorization, vectorize your implementation of `sparseAutoencoderCost.m`. In our implementation, we were able to remove all for-loops with the use of matrix operations and `repmat`. (If you want to play with more advanced vectorization ideas, also type `help bsxfun`. The `bsxfun` function provides an alternative to `repmat` for some of the vectorization steps, but is not necessary for this exercise). A vectorized version of our sparse autoencoder code ran in under one minute on a fast computer (for learning 25 features from 10000 8x8 image patches).

(Note that you do not need to vectorize the code in the other files.)

Step 2: Learn features for handwritten digits

Now that you have vectorized the code, it is easy to learn larger sets of features on medium sized images. In this part of the exercise, you will use your sparse autoencoder to learn features for handwritten digits from the MNIST dataset.

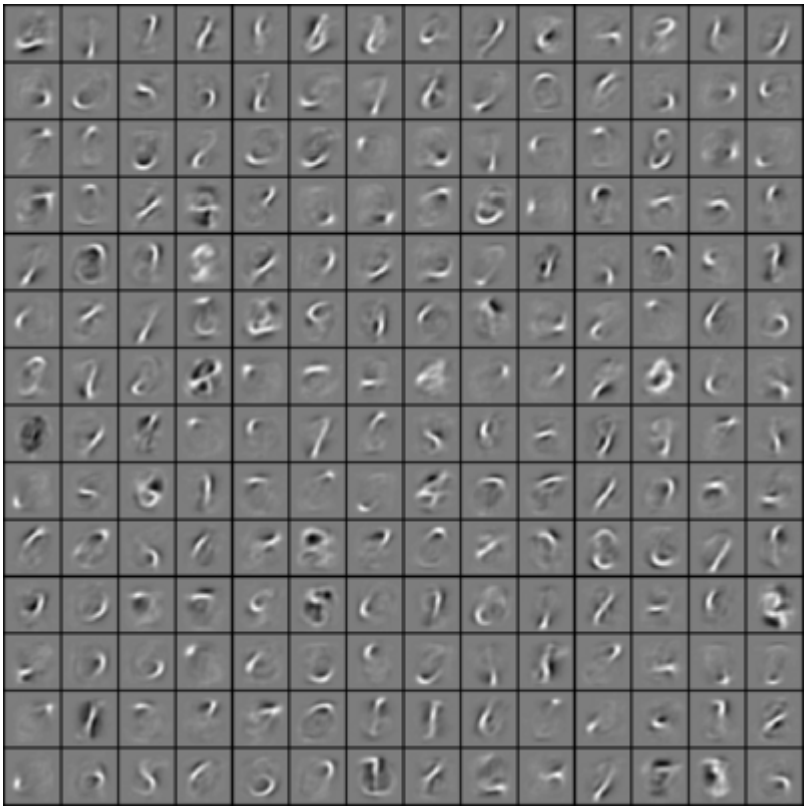
The MNIST data is available at [1] (http://yann.lecun.com/exdb/mnist/) . Download the file `train-images-idx3-ubyte.gz` and decompress it. After obtaining the source images, you should use helper functions that we provide to load the data into Matlab as matrices. While the helper functions that we provide will load both the input examples x and the class labels y, for this assignment, you will only need the input examples x since the

sparse autoencoder is an *unsupervised* learning algorithm. (In a later assignment, we will use the labels *y* as well.)

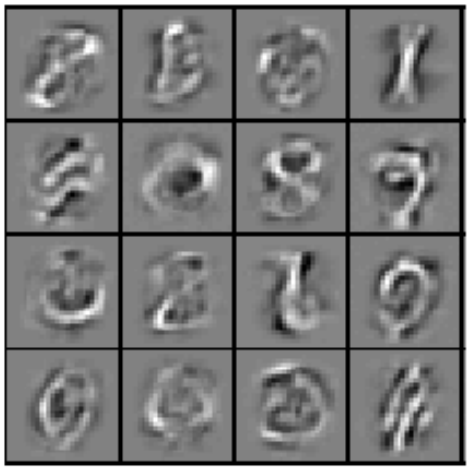
The following set of parameters worked well for us to learn good features on the MNIST dataset:

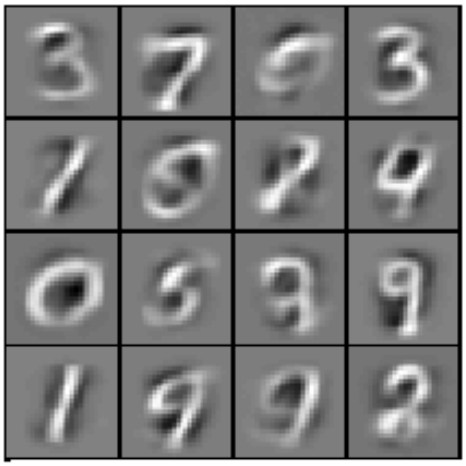
```
visibleSize = 28*28
hiddenSize = 196
sparsityParam = 0.1
lambda = 3e-3
beta = 3
patches = first 10000 images from the MNIST dataset
```

After 400 iterations of updates using minFunc, your autoencoder should have learned features that resemble pen strokes. In other words, this has learned to represent handwritten characters in terms of what pen strokes appear in an image. Our implementation takes around 15-20 minutes on a fast machine. Visualized, the features should look like the following image:



If your parameters are improperly tuned, or if your implementation of the autoencoder is buggy, you may get one of the following images instead:





If your image looks like one of the above images, check your code and parameters again. Learning these features are a prelude to the later exercises, where we shall see how they will be useful for classification.

Evolving Neural Networks through Augmenting Topologies

Kenneth O. Stanley kstanley@cs.utexas.edu
Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, USA

Risto Miikkulainen risto@cs.utexas.edu
Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, USA

Abstract

An important question in neuroevolution is how to gain an advantage from evolving neural network topologies along with weights. We present a method, NeuroEvolution of Augmenting Topologies (NEAT), which outperforms the best fixed-topology method on a challenging benchmark reinforcement learning task. We claim that the increased efficiency is due to (1) employing a principled method of crossover of different topologies, (2) protecting structural innovation using speciation, and (3) incrementally growing from minimal structure. We test this claim through a series of ablation studies that demonstrate that each component is necessary to the system as a whole and to each other. What results is significantly faster learning. NEAT is also an important contribution to GAs because it shows how it is possible for evolution to both optimize *and complexify* solutions simultaneously, offering the possibility of evolving increasingly complex solutions over generations, and strengthening the analogy with biological evolution.

Keywords

Genetic algorithms, neural networks, neuroevolution, network topologies, speciation, competing conventions.

1 Introduction

Neuroevolution (NE), the artificial evolution of neural networks using genetic algorithms, has shown great promise in complex reinforcement learning tasks (Gomez and Miikkulainen, 1999; Gruau et al., 1996; Moriarty and Miikkulainen, 1997; Potter et al., 1995; Whitley et al., 1993). Neuroevolution searches through the space of behaviors for a network that performs well at a given task. This approach to solving complex control problems represents an alternative to statistical techniques that attempt to estimate the utility of particular actions in particular states of the world (Kaelbling et al., 1996). NE is a promising approach to solving reinforcement learning problems for several reasons. Past studies have shown NE to be faster and more efficient than reinforcement learning methods such as Adaptive Heuristic Critic and Q-Learning on single pole balancing and robot arm control (Moriarty and Miikkulainen, 1996; Moriarty, 1997). Because NE searches for a behavior instead of a value function, it is effective in problems with continuous and high-dimensional state spaces. In addition, memory is easily represented through recurrent connections in neural networks, making NE a natural choice for learning non-Markovian tasks (Gomez and Miikkulainen, 1999, 2002).

In traditional NE approaches, a topology is chosen for the evolving networks before the experiment begins. Usually, the network topology is a single hidden layer of neurons, with each hidden neuron connected to every network input and every network output. Evolution searches the space of connection weights of this fully-connected topology by allowing high-performing networks to reproduce. The weight space is explored through the crossover of network weight vectors and through the mutation of single networks' weights. Thus, the goal of fixed-topology NE is to optimize the connection weights that determine the functionality of a network.

However, connection weights are not the only aspect of neural networks that contribute to their behavior. The topology, or *structure*, of neural networks also affects their functionality. Modifying the network structure has been shown effective as part of supervised training (Chen et al., 1993). There has also been a great deal of interest in evolving network topologies as well as weights over the last decade (Angeline et al., 1993; Branke, 1995; Gruau et al., 1996; Yao, 1999). The basic question, however, remains: Can evolving topologies along with weights provide an advantage over evolving weights on a fixed-topology? A fully connected network can in principle approximate any continuous function (Cybenko, 1989). So why waste valuable effort permuting over different topologies?

The answers provided thus far are inconclusive. Some have argued that network complexity can affect the speed and accuracy of learning (Zhang and Mühlenbein, 1993). Although this assertion is true for the backpropagation algorithm, it is not clear whether it applies when weights are being optimized by evolution and *not* backpropagation.

A persuasive argument for the evolution of both topology and weights was put forward by Gruau et al. (1996), who claimed that evolving structure saves the time wasted by humans trying to decide on the topology of networks for a particular NE problem. Although almost all fixed-topology NE systems use a fully connected hidden layer, deciding how many hidden nodes are needed is a trial-and-error process. Gruau et al. supported their argument by evolving the topology and weights of an artificial neural network that solved the hardest pole-balancing benchmark problem to date. However, later results suggested that structure was not necessary to solve the difficult problem. A fixed-topology method called *Enforced Subpopulations* (ESP) (Gomez and Miikkulainen, 1999) was able to solve the same problem 5 times faster simply by restarting with a random number of hidden neurons whenever it became stuck .

This article aims to demonstrate the opposite conclusion: if done right, evolving structure along with connection weights can significantly enhance the performance of NE. We present a novel NE method called *NeuroEvolution of Augmenting Topologies* (NEAT) that is designed to take advantage of structure as a way of minimizing the dimensionality of the search space of connection weights. If structure is evolved such that topologies are minimized and grown incrementally, significant gains in learning speed result. Improved efficiency results from topologies being minimized *throughout* evolution, rather than only at the very end.

Evolving structure incrementally presents several technical challenges: (1) Is there a genetic representation that allows disparate topologies to cross over in a meaningful way? (2) How can topological innovation that needs a few generations to be optimized be protected so that it does not disappear from the population prematurely? (3) How can topologies be minimized *throughout evolution* without the need for a specially contrived fitness function that measures complexity?

The NEAT method consists of solutions to each of these problems as will be described below. The method is validated on pole balancing tasks, where NEAT performs 25 times faster than Cellular Encoding and 5 times faster than ESP. The results show that structure is a powerful resource in NE when appropriately utilized. NEAT is unique because structures become increasingly more complex as they become more optimal, strengthening the analogy between GAs and natural evolution.

2 Background

Many systems have been developed over the last decade that evolve both neural network topologies and weights (Angeline et al., 1993; Braun and Weisbrod, 1993; Dasgupta and McGregor, 1992; Fullmer and Miikkulainen, 1992; Gruau et al., 1996; Krishnan and Ciesielski, 1994; Lee and Kim, 1996; Mandischer, 1993; Maniezzo, 1994; Opitz and Shavlik, 1997; Pujol and Poli, 1998; Yao and Liu, 1996; Zhang and Mühlenbein, 1993). These methods encompass a range of ideas about how *Topology and Weight Evolving Artificial Neural Networks* (TWEANNs) should be implemented. In this section, we address some of the ideas and assumptions about the design of TWEANNs, and offer solutions to some unsolved problems. Our goal is to find how a neuroevolution method can use the evolution of topology to increase its efficiency.

2.1 TWEANN Encoding

The question of how to encode networks using an efficient genetic representation must be addressed by all TWEANNs. We will discuss several prototypical representational schemes.

TWEANNs can be divided between those that use a direct encoding, and those that use an indirect one. Direct encoding schemes, employed by most TWEANNs, specify in the genome every connection and node that will appear in the phenotype (Angeline et al., 1993; Braun and Weisbrod, 1993; Dasgupta and McGregor, 1992; Fullmer and Miikkulainen, 1992; Krishnan and Ciesielski, 1994; Lee and Kim, 1996; Maniezzo, 1994; Opitz and Shavlik, 1997; Pujol and Poli, 1998; Yao and Liu, 1996; Zhang and Mühlenbein, 1993). In contrast, indirect encodings usually only specify rules for constructing a phenotype (Gruau, 1993; Mandischer, 1993). These rules can be layer specifications or growth rules through cell division. Indirect encoding allows a more compact representation than direct encoding, because every connection and node are not specified in the genome, although they can be derived from it.

2.1.1 Binary Encoding

Direct encodings usually require simpler implementations than indirect encodings. The simplest implementation is based on the traditional bit string representation used by GAs. For example, Dasgupta and McGregor (1992) use such an encoding in their method, called *Structured Genetic Algorithm* (sGA), where a bit string represents the connection matrix of a network. sGA is notable for its simplicity, allowing it to operate almost like a standard GA. However, there are several limitations as well. First, the size of the connectivity matrix is the square of the number of nodes. Thus, the representation blows up for a large number of nodes. Second, because the size of the bit string must be the same for all organisms, the maximum number of nodes (and hence connections as well) must be chosen by a human running the system, and if the maximum is not sufficient, the experiment must be repeated. Third, using a linear string of bits to represent a graph structure makes it difficult to ensure that crossover will yield useful combinations.

2.1.2 Graph Encoding

Because bit strings are not the most natural representation for networks, most TWEANNs use encodings that represent graph structures more explicitly. Pujol and Poli (1997) use a dual representation scheme to allow different kinds of crossover in their *Parallel Distributed Genetic Programming* (PDGP) system. The first representation is a graph structure. The second is a linear genome of node definitions specifying incoming and outgoing connections. The idea is that different representations are appropriate for different kinds of operators. Subgraph-swapping crossovers and topological mutations use the grid, while point crossovers and connection parameter mutations use the linear representation.

As in sGA, PDGP has a finite limit on the number of nodes in the network, corresponding to the number of nodes in the two-dimensional grid that represents the graph version of the genome. PDGP uses graph encoding so that subgraphs can be swapped in crossover. Subgraph swapping is representative of a prevailing philosophy in TWEANNs that subgraphs are functional units and therefore swapping them makes sense because it preserves the structure of functional components. However, we cannot be sure whether the particular subgraphs being combined in PDGP are the right ones to create a functional offspring.

2.1.3 Nonmating

Because crossover of networks with different topologies can frequently lead to a loss of functionality, some researchers have given up on crossover altogether in what is called Evolutionary Programming (Yao and Liu, 1996). Angeline et al. (1993) implemented a system called *GeNeralized Acquisition of Recurrent Links* (GNARL), commenting that "the prospect of evolving connectionist networks with crossover appears limited in general." Although GNARL uses a graph encoding, it is fundamentally different from PDGP in that it sidesteps the issue of crossover entirely. GNARL demonstrates that a TWEANN does not need crossover to work, leaving the problem of demonstrating the advantages of crossover to other methods.

2.1.4 Indirect Encoding

Gruau's (1993) *Cellular Encoding* (CE) method is an example of a system that utilizes indirect encoding of network structures. In CE, genomes are programs written in a specialized graph transformation language. The transformations are motivated by nature in that they specify *cell divisions*. Different kinds of connectivities can result from a division, so there are several kinds of cell divisions possible. A major advantage of CE is that its genetic representations are compact. Genes in CE can be reused multiple times during the development of a network, each time requesting a cell division at a different location. CE shows that cell divisions can encode the development of networks from a single cell, much as organisms in nature begin as a single cell that differentiates as it splits into more cells.

Although CE demonstrates that it is possible to evolve developmental systems, we chose direct encoding for NEAT because, as Braun and Weisbrod (1993) argue, indirect encoding requires "more detailed knowledge of genetic and neural mechanisms." In other words, because indirect encodings do not map directly to their phenotypes, they can bias the search in unpredictable ways. To make good use of indirect encodings, we need to first understand them well enough to make sure that they do not focus the search on some suboptimal class of topologies. Further, experimental results suggest that CE is not necessarily more efficient than direct encoding methods (Section 4.3.3).

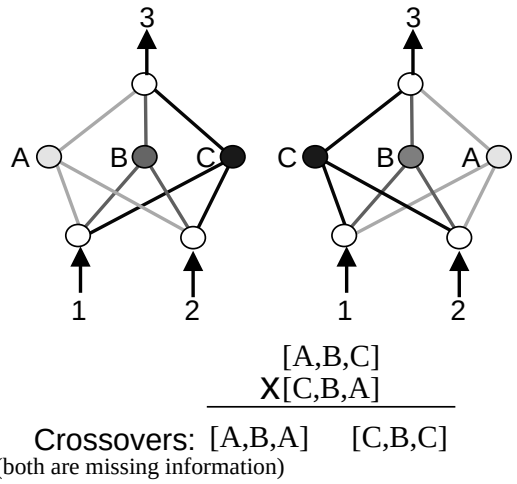


Figure 1: The competing conventions problem. The two networks compute the same exact function even though their hidden units appear in a different order and are represented by different chromosomes, making them incompatible for crossover. The figure shows that the two single-point recombinations are both missing one of the 3 main components of each solution. The depicted networks are only 2 of the 6 possible permutations of hidden unit orderings.

We now turn to several specific problems with TWEANNs and address each in turn.

2.2 Competing Conventions

One of the main problems for NE is the *Competing Conventions Problem* (Montana and Davis, 1989; Schaffer et al., 1992), also known as the *Permutations Problem* (Radcliffe, 1993). Competing conventions means having more than one way to express a solution to a weight optimization problem with a neural network. When genomes representing the same solution do not have the same encoding, crossover is likely to produce damaged offspring.

Figure 1 depicts the problem for a simple 3-hidden-unit network. The three hidden neurons A, B, and C, can represent the same general solution in $3! = 6$ different permutations. When one of these permutations crosses over with another, critical information is likely to be lost. For example, crossing $[A, B, C]$ and $[C, B, A]$ can result in $[C, B, C]$, a representation that has lost one third of the information that both of the parents had. In general, for n hidden units, there are $n!$ functionally equivalent solutions. The problem can be further complicated with *differing* conventions, i.e., $[A, B, C]$ and $[D, B, E]$, which share functional interdependence on B.

An even more difficult form of competing conventions is present in TWEANNs, because TWEANN networks can represent similar solutions using entirely different topologies, or even genomes of different sizes. Because TWEANNs do not satisfy strict constraints on the kinds of topologies they produce, proposed solutions to the competing conventions problem for fixed or constrained topology networks such as nonredundant genetic encoding (Thierens, 1996) do not apply. Radcliffe (1993) goes as far as calling an integrated scheme combining connectivity and weights the "Holy Grail in

this area." Although some TWEANNs such as PDGP have attempted to address the problem by assuming that subnetworks represent functional units that can be recombined, different topologies may not be based on the same subnetworks at all, in which case no meaningful combination of substructures exists.

The main intuition behind NEAT originates from the fundamental problem with representing different structures: their representations will not necessarily match up. Sometimes, the genomes can have different sizes. Other times, genes in the exact same position on different chromosomes may be expressing completely different traits. In addition, genes expressing the same trait may appear at different positions on different chromosomes. How can these complications be resolved?

Nature faces a similar problem with gene alignment in sexual reproduction. Genomes in nature are not of fixed-length either. Somewhere along the evolution from single cells to more complex organisms, new genes were added to the genomes in a process called *gene amplification* (Darnell and Doolittle, 1986; Watson et al., 1987). If new genes could just randomly insert themselves in positions on the genome without any indication of which gene is which, life probably would not have succeeded, because the competing conventions problem would decimate a huge chunk of offspring. There needed to be some way to keep crossover orderly, so that the *right* genes could be crossed with the right genes.

Nature's solution utilizes *homology*: two genes are homologous if they are alleles of the same trait. For example, in *E. coli*, in a process called synapsis, a special protein called RecA goes through and lines up homologous genes between two genomes before crossover occurs (Radding, 1982; Sigal and Alberts, 1972). Actual homology between neural networks cannot be easily ascertained by direct structural analysis (hence, the competing conventions problem). The main insight in NEAT is that the *historical origin* of two genes is direct evidence of homology if the genes share the same origin. Thus, NEAT performs *artificial synapsis* based on historical markings, allowing it to add new structure without losing track of which gene is which over the course of a simulation.

2.3 Protecting Innovation with Speciation

In TWEANNs, innovation takes place by adding new structure to networks through mutation. Frequently, adding new structure initially causes the fitness of a network to decrease. For example, adding a new node introduces a nonlinearity where there was none before; adding a new connection can reduce fitness before its weight has a chance to optimize. It is unlikely that a new node or connection just happens to express a useful function as soon as it is introduced. Some generations are required to optimize the new structure and make use of it. Unfortunately, because of the initial loss of fitness caused by the new structure, the innovation is unlikely to survive in the population long enough to be optimized. Thus, it is necessary to somehow protect networks with structural innovations so they have a chance to make use of their new structure.

The GNARL system addresses the problem of protecting innovation by adding nonfunctional structure. A node is added to a genome without any connections, in the hopes that in the future some useful connections will develop. However, nonfunctional structures may never end up connecting to the functional network, adding extraneous parameters to the search.

In nature, different structures tend to be in different species that compete in different niches. Thus, innovation is implicitly protected within a niche. Similarly, if networks with innovative structures could be isolated into their own species, they would

have a chance to optimize their structures before having to compete with the population at large.

Speciation, also known as *niching*, has been studied in GAs, but is not usually applied to neuroevolution. Speciation is most commonly applied to multimodal function optimization (Mahfoud, 1995), where a function has multiple optima, and a GA with several species is used to find those optima. Speciation has also been applied in the cooperative coevolution of modular systems of multiple solutions (Darwen and Yao, 1996; Potter and De Jong, 1995).

Speciation requires a compatibility function to tell whether two genomes should be in the same species or not. It is difficult to formulate such a compatibility function between networks of different topologies, which may be the reason why speciation has not been brought into TWEANNs. The competing conventions problem makes measuring compatibility particularly problematic because networks that compute the same function can appear very different.

However, because NEAT has a solution to the competing conventions problem, using historical information about genes, the population in NEAT can easily be speciated. We use *explicit fitness sharing*, which forces individuals with similar genomes to share their fitness payoff (Goldberg and Richardson, 1987). The original implicit version of fitness sharing introduced by Holland (1975) grouped individuals by performance similarity rather than genetic similarity. The explicit version is appropriate for TWEANNs because it allows grouping networks based on topology and weight configurations. The result of sharing fitness is that the number of networks that can exist in the population on a single fitness peak is limited by the size of the peak. Therefore, the population divides into a number of species, each on a different peak, without the threat of any one species taking over. Explicit fitness sharing is well-suited for NEAT, because similarity can easily be measured based on the historical information in the genes. Thus, innovations in NEAT are protected in their own species.

2.4 Initial Populations and Topological Innovation

In many TWEANN systems, the initial population is a collection of random topologies. Such a population ensures topological diversity from the start. However, random initial populations turn out to produce many problems for TWEANNs. For example, under many of the direct encoding schemes, there is a chance that a network will have no path from each of its inputs to its outputs. Such infeasible networks take time to weed out of the population.

However, there is a more subtle and more serious problem with starting randomly. As our experiments will confirm (Section 5.4), it is desirable to evolve minimal solutions; that way, the number of parameters that have to be searched is reduced. Starting out with random topologies does not lead to finding minimal solutions, since the population starts out with many unnecessary nodes and connections already present. None of these nodes or connections have had to withstand a single evaluation, meaning there is no justification for their configuration. Any minimization of networks would have to be spent getting rid of apparatus that should not have been there in the first place, and nothing in the process of recombining different topologies pushes towards such minimization. Since there is no fitness cost in creating larger networks, they will dominate as long as they have high fitness.

One way to force minimal topologies is to incorporate network size into the fitness function, and some TWEANNs actually do this (Zhang and Mühlenbein, 1993). In such methods, larger networks have their fitnesses penalized. Although altering the

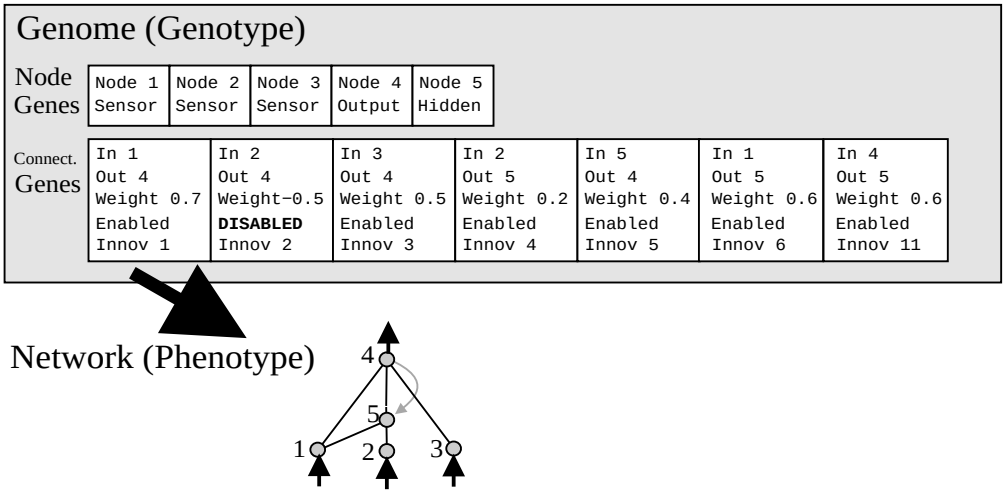


Figure 2: A genotype to phenotype mapping example. A genotype is depicted that produces the shown phenotype. There are 3 input nodes, one hidden, and one output node, and seven connection definitions, one of which is recurrent. The second gene is disabled, so the connection that it specifies (between nodes 2 and 4) is not expressed in the phenotype.

fitness function in this way can encourage smaller networks, it is difficult to know how large the penalty should be for any particular network size, particularly because different problems may have significantly different topological requirements. Altering the fitness function is ad hoc and may cause evolution to perform differently than the designer of the original unmodified fitness function intended.

An alternative solution is for the neuroevolution method itself to tend towards minimality. If the population begins with no hidden nodes and grows structure only as it benefits the solution, there is no need for ad hoc fitness modification to minimize networks. Therefore, starting out with a minimal population and growing structure from there is a design principle in NEAT.

By starting out minimally, NEAT ensures that the system searches for the solution in the lowest-dimensional weight space possible over the course of *all* generations. Thus, the goal is not to minimize only the final product, but all *intermediate* networks along the way as well. This idea is they key to gaining an advantage from the evolution of topology: it allows us to minimize the search space, resulting in dramatic performance gains. One reason current TWEANNS do not start out minimally is that without topological diversity present in the initial population, topological innovations would not survive. The problem of protecting innovation is not addressed by these methods, so networks with major structural additions are likely not to reproduce. Thus, speciating the population enables starting minimally in NEAT.

3 NeuroEvolution of Augmenting Topologies (NEAT)

The NEAT method, as described in detail in this section, consists of putting together the ideas developed in the previous section into one system. We begin by explaining

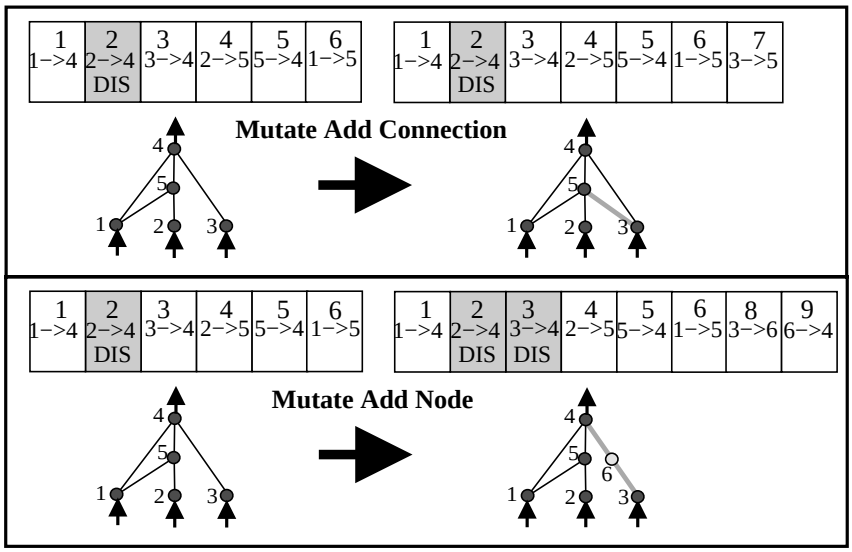


Figure 3: The two types of structural mutation in NEAT. Both types, adding a connection and adding a node, are illustrated with the connection genes of a network shown above their phenotypes. The top number in each genome is the *innovation number* of that gene. The innovation numbers are historical markers that identify the original historical ancestor of each gene. New genes are assigned new increasingly higher numbers. In adding a connection, a single new connection gene is added to the end of the genome and given the next available innovation number. In adding a new node, the connection gene being split is disabled, and two new connection genes are added to the end the genome. The new node is between the two new connections. A new node gene (not depicted) representing this new node is added to the genome as well.

the genetic encoding used in NEAT and continue by describing the components that specifically address each of the three problems of TWEANNs.

3.1 Genetic Encoding

NEAT's genetic encoding scheme is designed to allow corresponding genes to be easily lined up when two genomes cross over during mating. Genomes are linear representations of network connectivity (Figure 2). Each genome includes a list of *connection genes*, each of which refers to two *node genes* being connected. Node genes provide a list of inputs, hidden nodes, and outputs that can be connected. Each connection gene specifies the in-node, the out-node, the weight of the connection, whether or not the connection gene is expressed (an enable bit), and an *innovation number*, which allows finding corresponding genes (as will be explained below).

Mutation in NEAT can change both connection weights and network structures. Connection weights mutate as in any NE system, with each connection either perturbed or not at each generation. Structural mutations occur in two ways (Figure 3). Each mutation expands the size of the genome by adding gene(s). In the *add connection* mutation, a single new connection gene with a random weight is added connecting two previously unconnected nodes. In the *add node* mutation, an existing connection is

split and the new node placed where the old connection used to be. The old connection is disabled and two new connections are added to the genome. The new connection leading into the new node receives a weight of 1, and the new connection leading out receives the same weight as the old connection. This method of adding nodes was chosen in order to minimize the initial effect of the mutation. The new nonlinearity in the connection changes the function slightly, but new nodes can be immediately integrated into the network, as opposed to adding extraneous structure that would have to be evolved into the network later. This way, because of speciation, the network will have time to optimize and make use of its new structure.

Through mutation, the genomes in NEAT will gradually get larger. Genomes of varying sizes will result, sometimes with different connections at the same positions. The most complex form of the competing conventions problem, with numerous differing topologies and weight combinations, is an inevitable result of allowing genomes to grow unbounded. How can NE cross over differently sized genomes in a sensible way? The next section explains how NEAT addresses this problem.

3.2 Tracking Genes through Historical Markings

There is unexploited information in evolution that tells us exactly which genes match up with which genes between *any* individuals in a topologically diverse population. That information is the historical origin of each gene. Two genes with the same historical origin must represent the same structure (although possibly with different weights), since they are both derived from the same ancestral gene of some point in the past. Thus, all a system needs to do to know which genes line up with which is to keep track of the historical origin of every gene in the system.

Tracking the historical origins requires very little computation. Whenever a new gene appears (through structural mutation), a *global innovation number* is incremented and assigned to that gene. The innovation numbers thus represent a chronology of the appearance of every gene in the system. As an example, let us say the two mutations in Figure 3 occurred one after another in the system. The new connection gene created in the first mutation is assigned the number 7, and the two new connection genes added during the new node mutation are assigned the numbers 8 and 9. In the future, whenever these genomes mate, the offspring will inherit the same innovation numbers on each gene; innovation numbers are never changed. Thus, the historical origin of every gene in the system is known throughout evolution.

A possible problem is that the same structural innovation will receive different innovation numbers in the same generation if it occurs by chance more than once. However, by keeping a list of the innovations that occurred in the current generation, it is possible to ensure that when the same structure arises more than once through independent mutations in the same generation, each identical mutation is assigned the same innovation number. Thus, there is no resultant explosion of innovation numbers.

The historical markings give NEAT a powerful new capability. The system now knows exactly which genes match up with which (Figure 4). When crossing over, the genes in both genomes with the same innovation numbers are lined up. These genes are called *matching* genes. Genes that do not match are either *disjoint* or *excess*, depending on whether they occur within or outside the range of the other parent's innovation numbers. They represent structure that is not present in the other genome. In composing the offspring, genes are randomly chosen from either parent at matching genes, whereas all excess or disjoint genes are always included from the more fit parent. This way, historical markings allow NEAT to perform crossover using linear genomes with-

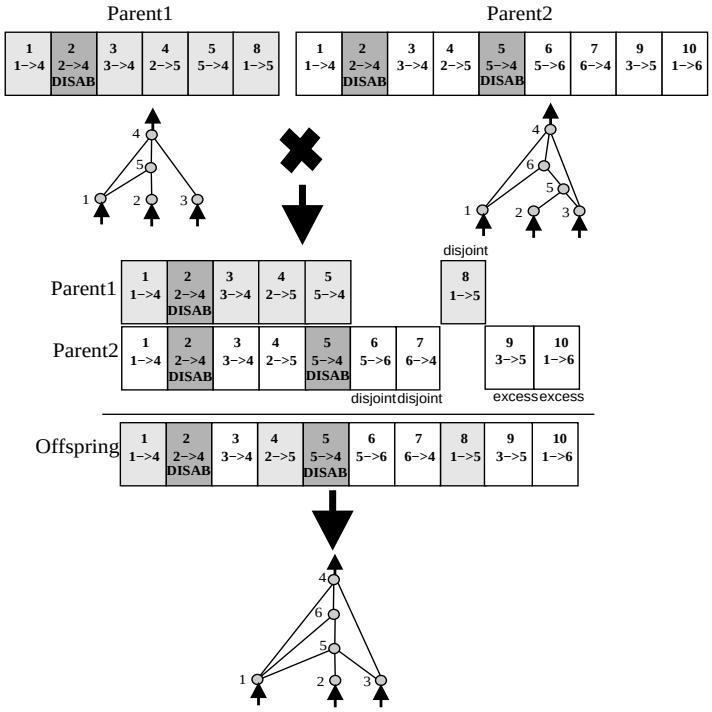


Figure 4: Matching up genomes for different network topologies using innovation numbers. Although Parent 1 and Parent 2 look different, their innovation numbers (shown at the top of each gene) tell us which genes match up with which. Even without any topological analysis, a new structure that combines the overlapping parts of the two parents as well as their different parts can be created. Matching genes are inherited randomly, whereas disjoint genes (those that do not match in the middle) and excess genes (those that do not match in the end) are inherited from the more fit parent. In this case, equal fitnesses are assumed, so the disjoint and excess genes are also inherited randomly. The disabled genes may become enabled again in future generations: there's a preset chance that an inherited gene is disabled if it is disabled in either parent.

out the need for expensive topological analysis.

By adding new genes to the population and sensibly mating genomes representing different structures, the system can form a population of diverse topologies. However, it turns out that such a population on its own cannot maintain topological innovations. Because smaller structures optimize faster than larger structures, and adding nodes and connections usually initially decreases the fitness of the network, recently augmented structures have little hope of surviving more than one generation even though the innovations they represent might be crucial towards solving the task in the long run. The solution is to protect innovation by speciating the population, as explained in the next section.

3.3 Protecting Innovation through Speciation

Speciating the population allows organisms to compete primarily within their own niches instead of with the population at large. This way, topological innovations are

protected in a new niche where they have time to optimize their structure through competition within the niche. The idea is to divide the population into species such that similar topologies are in the same species. This task appears to be a topology matching problem. However, it again turns out that historical markings offer an efficient solution.

The number of excess and disjoint genes between a pair of genomes is a natural measure of their compatibility distance. The more disjoint two genomes are, the less evolutionary history they share, and thus the less compatible they are. Therefore, we can measure the compatibility distance δ of different structures in NEAT as a simple linear combination of the number of excess E and disjoint D genes, as well as the average weight differences of matching genes $\overline{W}$, including disabled genes:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W}. \quad (1)$$

The coefficients c_1, c_2, and c_3 allow us to adjust the importance of the three factors, and the factor N, the number of genes in the larger genome, normalizes for genome size (N can be set to 1 if both genomes are small, i.e., consist of fewer than 20 genes).

The distance measure δ allows us to speciate using a compatibility threshold δ_t. An ordered list of species is maintained. In each generation, genomes are sequentially placed into species. Each existing species is represented by a random genome inside the species from the *previous generation*. A given genome g in the current generation is placed in the first species in which g is compatible with the representative genome of that species. This way, species do not overlap.[1] If g is not compatible with any existing species, a new species is created with g as its representative.

As the reproduction mechanism for NEAT, we use *explicit fitness sharing* (Goldberg and Richardson, 1987), where organisms in the same species must share the fitness of their niche. Thus, a species cannot afford to become too big even if many of its organisms perform well. Therefore, any one species is unlikely to take over the entire population, which is crucial for speciated evolution to work. The adjusted fitness f'_i for organism i is calculated according to its distance δ from every other organism j in the population:

$$f'_i = \frac{f_i}{\sum_{j=1}^{n} \mathrm{sh}(\delta(i,j))}. \quad (2)$$

The sharing function sh is set to 0 when distance $\delta(i,j)$ is above the threshold δ_t; otherwise, $\mathrm{sh}(\delta(i,j))$ is set to 1 (Spears, 1995). Thus, $\sum_{j=1}^{n} \mathrm{sh}(\delta(i,j))$ reduces to the number of organisms in the same species as organism i. This reduction is natural since species are already clustered by compatibility using the threshold δ_t. Every species is assigned a potentially different number of offspring in proportion to the sum of adjusted fitnesses f'_i of its member organisms. Species then reproduce by first eliminating the lowest performing members from the population. The entire population is then replaced by the offspring of the remaining organisms in each species.[2]

The net desired effect of speciating the population is to protect topological innovation. The final goal of the system, then, is to perform the search for a solution as efficiently as possible. This goal is achieved through minimizing the dimensionality of the search space.

[1] It is also possible to determine the compatibility of a genome g with a species s by using the average compatibility of g with every genome in a species s, but in practice, only comparing to the first genome in s is sufficient and takes constant time.

[2] In rare cases when the fitness of the entire population does not improve for more than 20 generations, only the top two species are allowed to reproduce, refocusing the search into the most promising spaces.

3.4 Minimizing Dimensionality through Incremental Growth from Minimal Structure

As discussed in Section 2.4, TWEANNs typically start with an initial population of random topologies in order to introduce diversity from the outset. In contrast, NEAT biases the search towards minimal-dimensional spaces by starting out with a *uniform* population of networks with zero hidden nodes (i.e., all inputs connect directly to outputs). New structure is introduced incrementally as structural mutations occur, and only those structures survive that are found to be useful through fitness evaluations. In other words, the structural elaborations that occur in NEAT are always justified. Since the population starts minimally, the dimensionality of the search space is minimized, and NEAT is always searching through fewer dimensions than other TWEANNs and fixed-topology NE systems. Minimizing dimensionality gives NEAT a performance advantage compared to other approaches, as will be discussed next.

4 Performance Evaluations

We evaluate the system's performance in order to answer two questions: (1) Can NEAT evolve the necessary structures? (2) Can NEAT find solutions more efficiently than other neuroevolution systems? The first question establishes that topology building indeed happens in NEAT in a reliable way, meaning that NEAT will grow new structure to cope with problems that require it. For this reason, NEAT is applied to the problem of building an XOR network. Although this task is simple, it requires growing hidden units, and therefore serves as a simple test for the method.

The second question is answered in the course of successively more difficult pole balancing tasks, where the objective is to balance two poles attached to a cart by moving the cart in appropriate directions to keep the pole from falling. Pole balancing is a good benchmark task because there are many different systems available for comparison. The most difficult problem of balancing two poles without velocity information, a non-Markovian task, provides very strong evidence that evolving augmenting topologies is not only interesting for its capacity to find structures, but is also efficient in difficult control tasks.

4.1 Parameter Settings

The same experimental settings are used in all experiments; they were not tuned specifically for any particular problem. The one exception is the hardest pole balancing problem (Double pole, no velocities, or DPNV) where a larger population size was used to match those of other systems in this task. Because some of NEAT's system parameters are sensitive to population size, we altered them accordingly.

All experiments except DPNV, which had a population of 1,000, used a population of 150 NEAT networks. The coefficients for measuring compatibility were $c_1 = 1.0$, $c_2 = 1.0$, and $c_3 = 0.4$. With DPNV, c_3 was increased to 3.0 in order to allow for finer distinctions between species based on weight differences (the larger population has room for more species). In all experiments, $\delta_t = 3.0$, except in DPNV where it was 4.0, to make room for the larger weight significance coefficient c_3. If the maximum fitness of a species did not improve in 15 generations, the networks in the stagnant species were not allowed to reproduce. The champion of each species with more than five networks was copied into the next generation unchanged. There was an 80% chance of a genome having its connection weights mutated, in which case each weight had a 90% chance of being uniformly perturbed and a 10% chance of being assigned a new random value. (The system is tolerant to frequent mutations because of the protection speciation pro-

vides.) There was a 75% chance that an inherited gene was disabled if it was disabled in either parent. In each generation, 25% of offspring resulted from mutation without crossover. The interspecies mating rate was 0.001. In smaller populations, the probability of adding a new node was 0.03 and the probability of a new link mutation was 0.05. In the larger population, the probability of adding a new link was 0.3, because a larger population can tolerate a larger number of prospective species and greater topological diversity. We used a modified sigmoidal transfer function, $\varphi(x) = \frac{1}{1+e^{-4.9x}}$, at all nodes. The steepened sigmoid allows more fine tuning at extreme activations. It is optimized to be close to linear during its steepest ascent between activations -0.5 and 0.5. These parameter values were found experimentally: links need to be added significantly more often than nodes, and an average weight difference of 3.0 is about as significant as one disjoint or excess gene. Performance is robust to moderate variations in these values.

4.2 Verification: Evolving XORs

Because XOR is not linearly separable, a neural network requires hidden units to solve it. The two inputs must be combined at some hidden unit, as opposed to only at the output node, because there is no function over a linear combination of the inputs that can separate the inputs into the proper classes. These structural requirements make XOR suitable for testing NEAT's ability to evolve structure. For example, NEAT's method for adding new nodes might be too destructive to allow new nodes to get into the population. Or, it could find a local champion with a wrong kind of connectivity that dominates the population so much that the systems fails to evolve the proper connectivity. Third, maybe the changing structure renders past connection weight values obsolete. If so, the algorithm would have trouble enlarging topologies that are already largely specialized. This experiment is meant to show that NEAT is not impeded by such potential obstacles, but can grow structure efficiently and consistently when needed.

To compute fitness, the distance of the output from the correct answer was summed for all four input patterns. The result of this error was subtracted from 4 so that higher fitness would reflect better network structure. The resulting number was squared to give proportionally more fitness the closer a network was to a solution.

The initial generation consisted of networks with no hidden units (Figure 5(a)). The networks had 2 inputs, 1 bias unit, and 1 output. The bias unit is an input that is always set to 1.0. There were three connection genes in each genome in the initial population. Two genes connected the inputs to the output, and one connected the bias to the output. Each connection gene received a random connection weight.

On 100 runs, the first experiment shows that the NEAT system finds a structure for XOR in an average of 32 generations (4,755 networks evaluated, std 2,553). On average a solution network had 2.35 hidden nodes and 7.48 nondisabled connection genes. The number of nodes and connections is close to optimal considering that the smallest possible network has a single hidden unit (Figure 5(b)). NEAT is very consistent in finding a solution. It did not fail once in 100 simulations. The worst performance took 13,459 evaluations, or about 90 generations (compared to 32 generations on average). The standard deviation for number of nodes used in a solution was 1.11, meaning NEAT very consistently used 1 or 2 hidden nodes to build an XOR network. In conclusion, NEAT solves the XOR problem without trouble and in doing so keeps the topology small.

The XOR problem has been used to demonstrate performance of several prior TWEANN algorithms. Unfortunately, quantitative performance comparisons are dif-

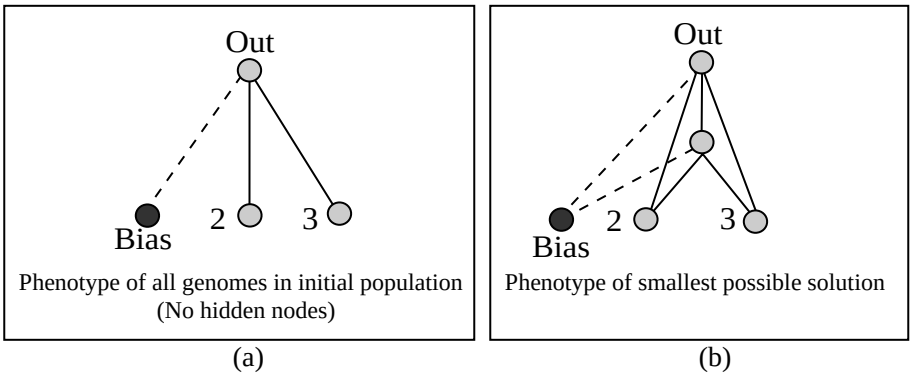


Figure 5: Initial phenotype and optimal XOR. Figure (a) shows the phenotype given to the entire initial population. Notice that there are no hidden nodes. In NEAT, a bias is a node that can connect to any node other than inputs. Figure (b) shows an optimal solution with only 1 hidden node. (A network without hidden nodes cannot compute XOR.) The bias connections are not always needed depending on the solution; All other connections are necessary. The optimal (1 hidden node) solution was found in 22 of 100 runs. The average solution had 2.35 hidden nodes with a standard deviation of 1.11 nodes.

ficult in this domain because the methodologies vary widely across experiments (Dasgupta and McGregor, 1992; Pujol and Poli, 1998; Yao and Shi, 1995; Zhang and Mühlenbein, 1993). Also, XOR is simple and artificial and does not challenge modern methods in the way real-world problems do. Let us next turn to benchmark problems where more substantial performance comparisons are possible.

4.3 Pole Balancing as a Benchmark Task

There are many control learning tasks where the techniques employed in NEAT can make a difference. Many of these potential applications, like robot navigation or game playing, present problems without known solutions. We use the pole balancing domain for comparison because it is a known benchmark in the literature, which makes it possible to demonstrate the effectiveness of NEAT compared to others. It is also a good surrogate for real problems, in part because pole balancing in fact *is* a real task, and also because the difficulty can be adjusted. Earlier comparisons were done with a single pole (Moriarty and Miikkulainen, 1996), but this version of the task has become too easy for modern methods. Balancing two poles simultaneously is on the other hand challenging enough for all current methods.

Therefore, we demonstrate the advantage of evolving structure through double pole balancing experiments. Two poles are connected to a moving cart by a hinge and the neural network must apply force to the cart to keep the poles balanced for as long as possible without going beyond the boundaries of the track. The system state is defined by the cart position x and velocity $\dot{x}$, the first pole's position θ_1 and angular velocity $\dot{\theta}_1$, and the second pole's position θ_2 and angular velocity $\dot{\theta}_2$. Control is possible because the poles have different lengths and therefore respond differently to control inputs.

Standard reinforcement learning methods have also been applied to this task (Anderson, 1989; Pendrith, 1994). However, we limit the comparisons in this paper to

NE methods for two reasons: (1) The focus is on developing and demonstrating better performance on evolving neural networks and (2) NE methods in this comparison have outperformed reinforcement learning methods in prior comparisons on the pole balancing task (Moriarty and Miikkulainen, 1996). Thus, the question here is whether evolving structure can lead to greater NE performance.

4.3.1 Pole Balancing Comparisons

We set up the pole balancing experiments as described by Wieland (1991) and Gomez and Miikkulainen (1999). The Runge-Kutta fourth-order method was used to implement the dynamics of the system, with a step size of 0.01s. All state variables were scaled to $[-1.0, 1.0]$ before being fed to the network. Networks output a force every 0.02 seconds between $[-10, 10]N$. The poles were 0.1m and 1.0m long. The initial position of the long pole was 1° and the short pole was upright; the track was 4.8 meters long.

Two versions of the double pole balancing task are used: one with velocity inputs included and another without velocity information. The first task is Markovian and allows comparing to many different systems. Taking away velocity information makes the task more difficult because the network must estimate an internal state in lieu of velocity, which requires recurrent connections.

On the double pole balancing with velocity (DPV) problem, NEAT is compared to published results from four other NE systems. The first two represent standard population-based approaches. Saravanan and Fogel (1995) used Evolutionary Programming, which relies entirely on mutation of connection weights, while Wieland (1991) used both mating and mutation. The second two systems, SANE (Moriarty and Miikkulainen, 1996) and ESP (Gomez and Miikkulainen, 1999), evolved populations of neurons and a population of network blueprints that specifies how to build networks from the neurons that are assembled into fixed-topology networks for evaluation. The topologies are fixed because the individual neurons are always placed into predesignated slots in the neural networks they compose. SANE maintains a single population of neurons. ESP improves over SANE by maintaining a separate population for each hidden neuron position in the complete network. To our knowledge, the results of ESP are the best achieved so far in this task.

On the double pole balancing without velocity problem (DPNV), NEAT is compared to the only two systems that have been demonstrated able to solve the task: CE and ESP. The success of CE was first attributed to its ability to evolve structures. However, ESP, a fixed-topology NE system, was able to complete the task five times faster simply by restarting with a random number of hidden nodes whenever it got stuck. Our experiments will attempt to show that evolution of structure can lead to better performance if done right.

4.3.2 Double Pole Balancing with Velocities

The criteria for success on this task was keeping both poles balanced for 100,000 time steps (30 minutes of simulated time). A pole was considered balanced between -36 and 36 degrees from vertical. Fitness on this task was measured as the number of time steps that both poles remained balanced.

Table 1 shows that NEAT takes the fewest evaluations to complete this task, although the difference between NEAT and ESP is not statistically significant. The fixed-topology NE systems evolved networks with 10 hidden nodes, while NEAT's solutions always used between 0 and 4 hidden nodes. Thus, it is clear that NEAT's minimization

Table 1: Double pole balancing with velocity information (DPV). Evolutionary programming results were obtained by Saravanan and Fogel (1995). Conventional neuroevolution data was reported by Wieland (1991). SANE and ESP results were reported by Gomez and Miikkulainen (1999). In addition, Gruau et al. (1996) reported 34,000 evaluations in this task; however, their results are not directly comparable because they used a different fitness function (Equations 3 and 4). NEAT results are averaged over 120 experiments. All other results are averages over 50 runs. The standard deviation for the NEAT evaluations is 2,704 evaluations. Although standard deviations for other methods were not reported, if we assume similar variances, all differences are statistically significant ($p < 0.001$), except that between NEAT and ESP.

Method	Evaluations	Generations	No. Nets
Ev. Programming	**307,200**	150	2048
Conventional NE	**80,000**	800	100
SANE	**12,600**	63	200
ESP	**3,800**	19	200
NEAT	**3,600**	24	150

of dimensionality is working on this problem. The result is important because it shows that NEAT performs as well as ESP while finding more minimal solutions.

4.3.3 Double Pole Balancing Without Velocities

Gruau et al. (1996) introduced a special fitness function for this problem to prevent the system from solving the task simply by moving the cart back and forth quickly to keep the poles wiggling in the air. (Such a solution would not require computing the missing velocities.) Because both CE and ESP were evaluated using this special fitness function, NEAT uses it on this task as well. The fitness penalizes oscillations. It is the sum of two fitness component functions, f_1 and f_2, such that $F = 0.1f_1 + 0.9f_2$. The two functions are defined over 1000 time steps:

$$f_1 = t/1000, \tag{3}$$

$$f_2 = \begin{cases} 0 & \text{if } t < 100, \\ \frac{0.75}{\sum_{i=t-100}^{t}(|x^i|+|\dot{x}^i|+|\theta_1^i|+|\dot{\theta}_1^i\ |)} & \text{otherwise.} \end{cases} \tag{4}$$

where t is the number of time steps the poles remain balanced during the 1,000 total time steps. The denominator in (4) represents the sum of offsets from center rest of the cart and the long pole. It is computed by summing the absolute value of the state variables representing the cart and long pole positions and velocities. Thus, by minimizing these offsets (damping oscillations), the system can maximize fitness. Because of this fitness function, swinging the poles is penalized, forcing the system to internally compute the hidden state variables.

Under Gruau et al.'s criteria for a solution, the champion of each generation is tested on generalization to make sure it is robust. This test takes a lot more time than the fitness test, which is why it is applied only to the champion. In addition to balancing both poles for 100,000 time steps, the winning controller must balance both poles from 625 different initial states, each for 1,000 times steps. The number of successes is called the *generalization performance of the solution*. In order to count as a solution, a network needs to generalize to at least 200 of the 625 initial states. Each start state is chosen by

Table 2: Double pole balancing without velocity information (DPNV). CE is Cellular Encoding of Gruau et al. (1996). ESP is Enforced Subpopulations of Gomez and Miikkulainen (1999). All results are averages over 20 simulations. The standard deviation for NEAT is 21,790 evaluations. Assuming similar variances for CE and ESP, all differences in number of evaluations are significant ($p < 0.001$). The generalization results are out of 625 cases in each simulation, and are not significantly different.

Method	Evaluations	Generalization	No. Nets
CE	**840,000**	300	16,384
ESP	**169,466**	289	1,000
NEAT	**33,184**	286	1,000

giving each state variable (i.e., x, $\dot{x}$, θ_1, and $\dot{\theta}_1$) each of the values 0.05, 0.25, 0.5, 0.75, 0.95 scaled to the range of the input variable ($5^4 = 625$). At each generation, NEAT performs the generalization test on the champion of the highest-performing species that improved since the last generation.

Table 2 shows that NEAT is the fastest system on this challenging task. NEAT takes 25 times fewer evaluations than Gruau's original benchmark, showing that the way in which structure is evolved has significant impact on performance. NEAT is also 5 times faster than ESP, showing that structure evolution can indeed perform better than evolution of fixed topologies. There was no significant difference in the ability of any of the 3 methods to generalize.

Why is NEAT so much faster than ESP on the more difficult task when there was not much difference in the easier task? The reason is that in the task without velocities, ESP needed to restart an average of 4.06 times per solution while NEAT never needed to restart. If restarts are factored out, the systems perform at similar rates. The best characterization of the difference is that NEAT is more reliable at avoiding deception. NEAT evolves many different structures simultaneously in different species, each representing a space of different dimensionality. Thus, NEAT is always trying many different ways to solve the problem at once, so it is less likely to get stuck.

The experimental results demonstrate both that NEAT can evolve structure when necessary, and that NEAT gains a significant performance advantage from doing so. We now turn to understanding how the system works, and whether it indeed solves the three problems with evolving a population of diverse topologies raised in the introduction.

5 Analysis of NEAT

We have argued that NEAT's performance is due to historical markings, speciation, and incremental growth from minimal structure. In order to verify the contribution of each component, we performed a series of ablations. In addition, we introduce a new species visualization technique in order to better understand the dynamics of the system.

Ablations are meant to establish that each component of NEAT is necessary for its performance. For example, it is possible that growth from minimal structure is not really important; maybe the rest of the system, speciation and historical markings, is sufficient for NEAT's optimal performance. This hypothesis will be checked by ablating both growth and starting from minimal structure from the system. On the other

Table 3: NEAT ablations summary. The table compares the average number of evaluations for a solution in the double pole balancing with velocities task. Each ablation leads to a weaker algorithm, showing that each component is necessary.

Method	Evaluations	Failure Rate
No-Growth NEAT (Fixed-Topologies)	**30,239**	80%
Nonspeciated NEAT	**25,600**	25%
Initial Random NEAT	**23,033**	5%
Nonmating NEAT	**5,557**	0
Full NEAT	**3,600**	0

hand, perhaps the situation is the opposite, and speciation buys nothing: protecting innovation might not be as important as we have argued. This hypothesis will be checked by ablating speciation from the system. Finally, we claimed that NEAT is able to make use of crossover even though genomes in NEAT have different sizes. This point is more controversial than it might seem. For example, Angeline et al. (1993) claimed that crossover in TWEANNs does more harm than good. We will check this hypothesis by ablating crossover from the system.

The reason why we do not ablate historical markings directly is that without historical markings the system would be a conventional NE system. Historical markings are the basis of every function in NEAT: Speciation uses a compatibility operator that is based on historical markings, and crossover would not be possible without them. All other system components can be ablated systematically.

5.1 Ablations Setup

Ablations can have a significant detrimental effect on performance, potentially to the point where the system cannot solve the task at all. Therefore, we used double pole balancing with velocities as the task for ablation studies. The task is complex enough to be interesting, yet still not too hard, so that ablated systems work as well. Thus, it is possible to compare the ablated versions of the system to the unablated system.

All settings were the same as in the double pole balancing with velocities experiment. Results are averages over 20 runs, except nonmating and full NEAT, which are averages over 120 runs (nonmating NEAT was fast enough to allow many runs).

5.2 Ablations Results

Table 3 shows the results of all the ablations, in terms of average evaluations required to find a solution. Averages in this table exclude trials that failed to find a solution in $1,000$ generations. A failure rate denotes how often such failures occurred for each ablation. The main result is that the system performs significantly worse ($p \leq 0.001$) for every ablation. We will explain how each ablation was performed and then interpret the results.

5.3 No-Growth Ablation

Since populations in NEAT start out with no hidden nodes, simply removing growth from the system would disable NEAT by barring hidden nodes from all networks. NE systems where structures are fixed start with a fully connected hidden layer of neurons (Wieland, 1991). Therefore, to make the experiment fair, the no-growth ablation was also allowed to start with a fully connected hidden layer. Every genome specified 10

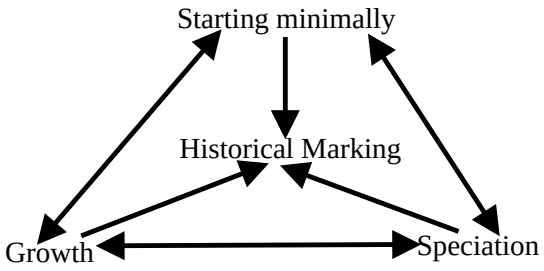


Figure 6: Dependencies among NEAT components. Strong interdependencies can be identified among the different components of NEAT.

hidden units like the fixed topology methods in this task (Saravanan and Fogel, 1995; Wieland, 1991). Without growth, NEAT was still able to speciate, but only based on weight differences. Given 1,000 generations to find a solution, the ablated system could only find a solution 20% of the time! When it did find a solution, it took 8.5 times more evaluations than full NEAT. Clearly, speciation and historical markings alone do not account for full NEAT's performance.

5.4 Initial Random Ablation

TWEANNs other than NEAT typically start with a random population (Angeline et al., 1993; Gruau et al., 1996; Yao, 1999). The structures in these systems can still grow (in most cases up to some bound). It is still possible that although growth is necessary, starting minimally is not.

We examined this question by starting evolution with random topologies as in other TWEANNs. Each network in the initial population received between 1 and 10 hidden neurons with random connectivity (as implemented by Pujol and Poli (1997)). The result is that random-starting NEAT was 7 times slower than full NEAT on average. The random-starting system also failed to find a solution within 1000 generations 5% of the time. The result suggests that starting randomly forces NE to search higher-dimensional spaces than necessary, thereby wasting time. If topologies are to grow, they should start out as small as possible.

5.5 Nonspeciated Ablation

We have argued that speciation is important because it protects innovation and allows search to proceed in many different spaces simultaneously. To test this claim, speciation should be ablated from the system. However, if this is done and nothing else is changed in the system, no structural innovations can survive, causing all networks to be stuck in minimal form.

To make the speciation ablation more meaningful, the nonspeciated NEAT must be started with an initial random population. This way, a variety of structures exist in the population from the beginning, and speciation is not necessary to attain structural diversity. The resulting nonspeciated NEAT was able to find solutions, although it failed in 25% of the attempts. When it found a solution, it was 7 times slower on average than full NEAT.

The reason for the dramatic slowdown is that without speciation, the population quickly converges on whatever topology happens to initially perform best. Thus, a lot of diversity is drained immediately (within 10 generations). On average, this initially

best-performing topology has about 5 hidden nodes. Thus, the population tends to converge to a relatively high-dimensional search space, even though the smaller networks in the initial population would have optimized faster. The smaller networks just do not get a chance because being small offers no immediate advantage in the initially random weight space. Of course, once in a while small networks are found that perform well, allowing a solution to be found more quickly. Whether or not such networks are found early on accounts for the large standard deviation of 41,704 evaluations.

The result shows that growth without speciation is not sufficient to account for NEAT's performance. For growth to succeed, it requires speciation, because speciation gives different structures a chance to optimize in their own niches.

5.6 Nonmating NEAT

For the last ablation, we removed mating from NEAT. This ablation tests the claim that crossover is a useful technique in TWEANNs. If NEAT's method of crossover works, then NEAT should perform significantly better with mating and mutation than with mutation alone.

A total of 120 simulations were run with crossover disabled but all other settings the same as before. It took on average 5,557 evaluations to find a solution without mating, compared to 3,600 with mating enabled. The difference is statistically significant ($p = 0.001$). Thus, it is clear that mating *does* contribute when it is done right. However, the nonmating version of NEAT is still significantly faster than the other ablations.

5.7 Ablation Conclusions

An important conclusion is that all of the parts of NEAT work together (Figure 6). None of the system can work without historical markings because all of NEAT's functions utilize historical markings. If growth from minimal structure is removed, speciation can no longer help NEAT find spaces with minimal dimensionality. If speciation is removed, growth from minimal structures cannot proceed because structural innovations do not survive. When the system starts with a population of random topologies without speciation, the system quickly converges onto a nonminimal topology that just happens to be one of the best networks in the initial population. Thus, each component is necessary to make NEAT work.

5.8 Visualizing Speciation

The ablation studies demonstrate that speciation is a necessary part of the overall system. To understand how innovation takes place in NEAT, it is important to understand the dynamics of speciation. How many species form over the course of a run? How often do new species arise? How often do species die? How large do the species get? We answer these questions by depicting speciation visually over time.

Figure 7 depicts a typical run of the double pole balancing with velocities task. In this run, the task took 29 generations to complete, which is slightly above average. In the visualization, successive generations are shown from top to bottom. Species are depicted horizontally for each generation, with the width of each species proportional to its size during the corresponding generation. Species are divided from each other by white lines, and new species always arrive on the right hand side. A species becomes bright when the fitness of its most fit member is one standard deviation above the mean fitness of the run, indicating that the species is a highly promising one. A species becomes very bright when it is two standard deviations above the mean, suggesting that the species is very close to a solution. Thus, it is possible to follow any species

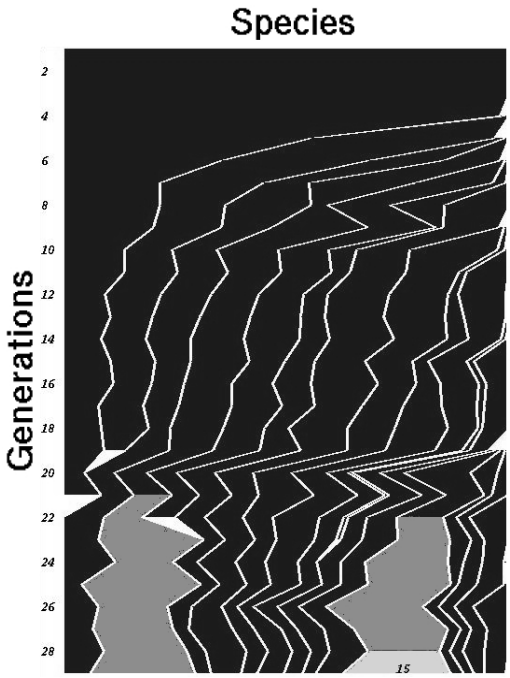


Figure 7: Visualizing speciation during a run of the double pole balancing with velocity information task. Two species begin to close in on a solution soon after the 20th generation. Around the same time, some of the oldest species become extinct.

from its inception to the end of the run.

Figure 7 shows that the initial minimal topology species was the only species in the population until the 5th generation. Recall that species are computed by the system according to a compatibility distance metric, indicating that before generation 7, all organisms were sufficiently compatible to be grouped into a single species. The visualization shows how the initial species shrinks dramatically in order to make room for the new species.

A few species can be observed becoming extinct during this run. When a species becomes extinct, we see a white triangle between the generation it expired and the next generation. Thus, in this run, the initial species finally became extinct at the 21st generation after shrinking for a long time. It was unable to compete with newer, more innovative species. The second species to appear in the population met a similar fate in the 19th generation.

In the 21st generation, a structural mutation in the second-oldest surviving species connected the long pole angle sensor to a hidden node that had previously only been connected to the cart position sensor. This gave networks in the species the new capability to combine these observations, leading to a significant boost in fitness (and brightening of the species in Figure 7). The innovative species subsequently expanded but did not take over the population. Nearly simultaneously, in the 22nd generation, a younger species also made its own useful connection, this time between the short pole velocity sensor and long pole angle sensor, leading to its own subsequent expansion. In the 28th generation, this same species made a pivotal connection between the cart

position and its already established method for comparing short pole velocity to long pole angle. This innovation was enough to solve the problem within one generation of additional weight mutations. In the final generation, the winning species was 11 generations old and included 38 neural networks out of the population of 150.

Most of the species that did not come close to a solution survived the run even though they fell significantly behind around the 21st generation. This observation is important, because it visually demonstrates that innovation is indeed being protected. The winning species does not take over the entire population.

Ablation studies confirm the interdependence of all of NEAT's components, and the speciation visualization offers a means of visualizing the dynamics of the system. We now turn to a discussion of the advantages and shortcomings of the method and its future potential.

6 Discussion and Future Work

NEAT presents several advances in the evolution of neural networks. Through historical markings, NEAT offers a solution to the problem of competing conventions in a population of diverse topologies. NEAT also demonstrates that a meaningful metric for comparing and clustering similar networks easily derives from the availability of historical information in the population, such that costly topological analysis is not necessary to speciate or mate networks. Our ablation studies confirm the hypothesis that starting with a population of minimal topologies is advantageous. Finally, performance comparisons suggest that the evolution of structure can be used to gain efficiency over the evolution of fixed topologies.

A parallel can be drawn between structure evolution in NEAT and incremental evolution (Gomez and Miikkulainen, 1997; Wieland, 1991). Incremental evolution is a method used to train a system to solve harder tasks than it normally could by training it on incrementally more challenging tasks. NE is likely to get stuck on a local optimum when attempting to solve the harder task directly. However, after solving the easier version of the task first, the population is likely to be in a part of fitness space closer to the solution to the harder task, allowing it to avoid local optima. Adding structure to a solution is analogous to taking a solution to an easy task as the starting point for evolving the solution to a harder task. The network structure before the addition is optimized in a lower-dimensional space. When structure is added, the network increments into a more complex space where it is already close to the solution. The difference between the incrementality of adding structure and general incremental evolution is that adding structure is *automatic* in NEAT, whereas a sequence of progressively harder tasks requires human design.

A key insight behind NEAT is that it is *not* the ultimate structure of the solution that really matters, but rather the structure of all the intermediate solutions along the way to finding the solution. The connectivity of every intermediate solution represents a parameter space that evolution must optimize, and the more connections there are, the more parameters need to be optimized. Therefore, if the amount of structure can be minimized throughout evolution, so can the dimensionality of the spaces being explored, leading to significant performance gains.

Figure 8 illustrates the advantage of evolving minimal structures with a picture of an elegant solution to the DPNV task. All evolved solutions displayed the same pattern of weights on direct connections from the input nodes to the output nodes. The connections from nodes one and two were always inhibitory, while those from nodes three and four were always excitatory. NEAT also evolved a recurrent loop on the single

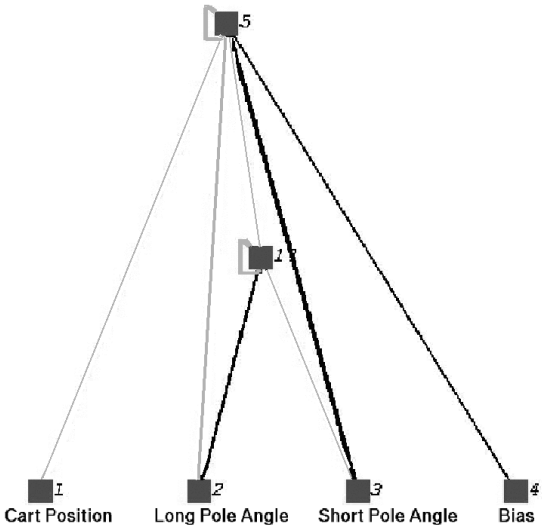


Figure 8: A NEAT solution to the DPNV problem. This clever solution works by taking the derivative of the difference in pole angles. Using the recurrent connection to itself, the single hidden node determines whether the poles are falling away or towards each other. This solution allows controlling the system without computing the velocities of each pole separately. Without evolving structure, it would be difficult to discover such subtle and compact solutions. Starting minimally makes discovering such compact solutions more likely.

output node of all solution networks. In 90% of solutions, this recurrent connection was excitatory and served to maintain continuity in the actions. Upon this general theme, solutions included varying degrees of structural elaborations utilizing from zero to two hidden nodes. For example, the solution shown in Figure 8 uses one hidden node to determine whether the poles are moving towards or away from one another. The basic underlying structure is the same across all these solutions and suggests that once a proper minimal foundation is discovered, a variety of paths open up to a solution.

In order to minimize structure throughout evolution, NEAT incrementally elaborates structure in a stochastic manner from a minimal starting point. Because of speciation, useful elaborations survive even if they are initially detrimental. Thus, NEAT strengthens the analogy between GAs and natural evolution by not only performing the optimizing function of evolution, but also a *complexifying* function, allowing solutions to become incrementally more complex at the same time as they become more optimal.

It is this complexifying function that makes NEAT unique among GAs. Although GAs have been proposed where bit string chromosomes can increase in length indefinitely (Harvey, 1993), NEAT goes beyond a gradual uniform growth in genome size. While Harvey's method increases the size of a chromosome incrementally across the entire population, NEAT simultaneously searches over *different* landscapes, all complexifying in different ways.

Such speciated search of incrementally increasing complexity offers a possibly powerful new approach to the problem of competitive coevolution. In competitive coevolution, increasingly sophisticated strategies are evolved by allowing networks in a

population to compete against each other. The hope is that an "arms race" will force the opponents to continually evolve strategies better than the strategies of other networks in the population. This method is useful because it can produce high-level strategies and tactics without the need for an expert player to teach the system. Ideally, strategies should become more sophisticated as evolution progresses. However, evolution tends to find the simplest solutions that can win, meaning that strategies oscillate between different idiosyncratic yet uninteresting variations (Darwen, 1996; Rosin and Belew, 1997).

We hypothesize that once competitive coevolution converges onto a dominant strategy, it cannot be improved upon, because it takes the entire set of connection weight values to represent the strategy. Altering the weights means altering the strategy, rather than building upon and complexifying the strategy. Thus, if a new strategy is to take hold, it must win by being *different* than the previous dominant strategy, rather than by being *more sophisticated*. In contrast, NEAT can evolve increasingly more sophisticated strategies continually, because as soon as the population converges on a new dominant strategy, new connections and new nodes can be added to the current strategy. New structure means new *expressive space* for *elaborating* on the existing strategy, rather than replacing it. Thus, this approach allows *continual coevolution*, i.e., nonconvergent innovation on dominant strategies. In addition, because different strategies in different species are protected, there will be multiple dominant and continually more complex strategies.

In addition to continual coevolution, the evolution of structure should allow the integration of separate expert neural networks. For example, suppose one neural network can "kick" a ball towards the goal from any position on a field, and another network can dribble the ball around the field without losing control. Neither of these two networks alone can play soccer. However, if we could somehow combine their expertise into one, perhaps we could get a soccer player out. Combining these controllers is not a simple matter of processing both their outputs. Just because a robot can dribble and shoot does not mean it knows *where* to dribble or *when* to shoot. Both shooting and dribbling affect each other as well. Where you dribble affects how easy your shot is, and shooting forces a robot to stop dribbling. In order to optimally combine the two skills, the hidden nodes of the two networks must share information so that the new combined expert can make intelligent decisions and combine the two skills effectively. We hypothesize that NEAT has the capability of searching for the right interconnections between two distinct networks to create an integrated supernetwork that takes advantage of the expertise of both its component networks.

Finally, we would like to establish a characterization of what NEAT is best suited for. The experimental results show the difference between ESP and NEAT is significantly higher on the hardest pole balancing task. This result implies that evolving diverse topologies is particularly suited for problems where other methods are likely to get stuck. Such problems may be deceptive, meaning local optima have large basins of attraction compared to global optima, and the global optima are significantly different from the local optima (Goldberg, 1989). Because NEAT can always add more structure, it is not necessarily trapped even if the current weights of a networks represent a local optimum in fitness space. By adding additional structure, NEAT adds new dimensions to weight space, thereby opening up potential new avenues for escape. We plan to test NEAT on problems with varying fitness landscapes to get a better idea of the kinds of problems the method tackles best.

7 Conclusion

The main conclusion is that NEAT is a powerful method for artificially evolving neural networks. NEAT demonstrates that evolving topology along with weights can be made a major advantage. Experimental comparisons verify that such evolution is several times more efficient than the neuroevolution methods so far. Ablation studies show that historical markings, protection of innovation through speciation, and incremental growth from minimal structure all work together to produce a system that is capable of evolving solutions of minimal complexity. NEAT strengthens the analogy between GAs and natural evolution by both optimizing *and complexifying* solutions simultaneously. We believe that the capacity to complexify solutions over the course of evolution offers the possibility of continual competitive coevolution and evolution of combinations of experts in the future.

Acknowledgments

This research was supported in part by the National Science Foundation under grant IIS-0083776 and by the Texas Higher Education Coordinating Board under grant ARP-003658-476-2001. Thanks to Faustino Gomez for providing pole balancing code and Michael Shao for pointers to the biology literature.

References

Anderson, C. W. (1989). Learning to control an inverted pendulum using neural networks. *IEEE Control Systems Magazine*, 9:31–37.

Angeline, P. J., Saunders, G. M., and Pollack, J. B. (1993). An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5:54–65.

Branke, J. (1995). Evolutionary algorithms for neural network design and training. In Alander, J. T., editor, *Proceedings First Nordic Workshop on Genetic Algorithms and their Applications*, pages 145–163, University of Vaasa Press, Vaasa, Finland.

Braun, H. and Weisbrod, J. (1993). Evolving feedforward neural networks. In Albrecht, R. F., Reeves, C. R., and Steele, N. C., editors, *Proceedings of ANNGA93, International Conference on Artificial Neural Networks and Genetic Algorithms*, pages 25–32, Springer-Verlag, Innsbruck.

Chen, D. et al. (1993). Constructive learning of recurrent neural networks. In Petsche, T., Judd, S., and Hanson, S., editors, *Computational Learning Theory and Natural Learning Systems III*. MIT Press, Cambridge, Massachusetts.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314.

Darnell, J. E. and Doolittle, W. F. (1986). Speculations on the early course of evolution. *Proceedings of the National Academy of Sciences, USA*, 83:1271–1275.

Darwen, P. J. (1996). *Co-Evolutionary Learning by Automatic Modularisation with Speciation*. Ph.D. thesis, School of Computer Science, University College, University of New South Wales, Sydney, Australia.

Darwen, P. and Yao, X. (1996). Automatic modularization by speciation. In *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation (ICEC '96)*, pages 88–93, Nagoya, IEEE Press, Piscataway, New Jersey.

Dasgupta, D. and McGregor, D. (1992). Designing application-specific neural networks using the structured genetic algorithm. In Whitley, D. and Schaffer, J. D., editors, *Proceedings of the International Conference on Combinations of Genetic Algorithms and Neural Networks*, pages 87–96, IEEE Press, Piscataway, New Jersey.

Fullmer, B. and Miikkulainen, R. (1992). Using marker-based genetic encoding of neural networks to evolve finite-state behaviour. In Varela, F. J. and Bourgine, P., editors, *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*, pages 255–262, MIT Press, Cambridge, Massachusetts.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Reading, Massachusetts.

Goldberg, D. E. and Richardson, J. (1987). Genetic algorithms with sharing for multimodal function optimization. In Grefenstette, J. J., editor, *Proceedings of the Second International Conference on Genetic Algorithms*, pages 148–154, Morgan Kaufmann, San Francisco, California.

Gomez, F. and Miikkulainen, R. (1997). Incremental evolution of complex general behavior. *Adaptive Behavior*, 5:317–342.

Gomez, F. and Miikkulainen, R. (1999). Solving non-Markovian control tasks with neuroevolution. In Dean, T., editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 1356–1361, Morgan Kaufmann, San Francisco, California.

Gomez, F. and Miikkulainen, R. (2002). Learning robust nonlinear control with neuroevolution. Technical Report AI02-292, Department of Computer Sciences, The University of Texas at Austin, Austin, Texas.

Gruau, F. (1993). Genetic synthesis of modular neural networks. In Forrest, S., editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 318–325, Morgan Kaufmann, San Francisco, California.

Gruau, F., Whitley, D., and Pyeatt, L. (1996). A comparison between cellular encoding and direct encoding for genetic neural networks. In Koza, J. R. et al., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 81–89, MIT Press, Cambridge, Massachusetts.

Harvey, I. (1993). *The Artificial Evolution of Adaptive Behavior* Ph.D. thesis, School of Cognitive and Computing Sciences, University of Sussex, Sussex, UK.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. University of Michigan Press, Ann Arbor, Michigan.

Kaelbling, L. P., Littman, M., and Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence*, 4:237–285.

Krishnan, R. and Ciesielski, V. B. (1994). Delta-gann: A new approach to training neural networks using genetic algorithms. In Tsoi, A. C. and Downs, T., editors, *Proceedings of the Australian Conference on Neural Networks*, pages 194–197, University of Queensland, Brisbane, Australia.

Lee, C.-H. and Kim, J.-H. (1996). Evolutionary ordered neural network with a linked-list encoding scheme. In *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation*, pages 665–669, IEEE Press, Piscataway, New Jersey.

Mahfoud, S. W. (1995). *Niching Methods for Genetic Algorithms* Ph.D. thesis, Department of General Engineering, IlliGAL Report 95001, University of Illinois at Urbana-Champaign, Urbana, Illinois.

Mandischer, M. (1993). Representation and evolution of neural networks. In Albrecht, R. F., Reeves, C. R., and Steele, U. C., editors, *Artificial Neural Nets and Genetic Algorithms*, pages 643–649, Springer Verlag, New York, New York.

Maniezzo, V. (1994). Genetic evolution of the topology and weight distribution of neural networks. *IEEE Transactions on Neural Networks*, 5(1):39–53.

Montana, D. J. and Davis, L. (1989). Training feedforward neural networks using genetic algorithms. In Sridharan, S., editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 762–767, Morgan Kaufmann, San Francisco, California.

Moriarty, D. E. (1997). *Symbiotic Evolution of Neural Networks in Sequential Decision Tasks.* Ph.D. thesis, Department of Computer Sciences, The University of Texas at Austin. Technical Report UT-AI97-257.

Moriarty, D. E. and Miikkulainen, R. (1996). Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, 22:11–32.

Moriarty, D. E. and Miikkulainen, R. (1997). Forming neural networks through efficient and adaptive co-evolution *Evolutionary Computation*, 5(4):373–399.

Opitz, D. W. and Shavlik, J. W. (1997). Connectionist theory refinement: Genetically searching the space of network topologies. *Journal of Artificial Intelligence Research*, 6:177–209.

Pendrith, M. (1994). On reinforcement learning of control actions in noisy and non-Markovian domains. Technical Report UNSW-CSE-TR-9410, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia.

Potter, M. A. and De Jong, K. A. (1995). Evolving neural networks with collaborative species. In Oren, T. I. and Birta, L. G., editors, *Proceedings of the 1995 Summer Computer Simulation Conference*, pages 340–345, Society for Computer Simulation, San Diego, California..

Potter, M. A., De Jong, K. A., and Grefenstette, J. J. (1995). A coevolutionary approach to learning sequential decision rules. In Eshelman, L. J., editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 366–372, Morgan Kaufmann, San Francisco, California.

Pujol, J. C. F. and Poli, R. (1998). Evolving the topology and the weights of neural networks using a dual representation. *Special Issue on Evolutionary Learning of the Applied Intelligence Journal*, 8(1):73–84.

Radcliffe, N. J. (1993). Genetic set recombination and its application to neural network topology optimisation. *Neural Computing and Applications*, 1(1):67–90.

Radding, C. M. (1982). Homologous pairing and strand exchange in genetic recombination. *Annual Review of Genetics*, 16:405–437.

Rosin, C. D. and Belew, R. K. (1997). New methods for competitive evolution. *Evolutionary Computation*, 5(1):1–29.

Saravanan, N. and Fogel, D. B. (1995). Evolving neural control systems. *IEEE Expert*, 10(3):23–27.

Schaffer, J. D., Whitley, D., and Eshelman, L. J. (1992). Combinations of genetic algorithms and neural networks: A survey of the state of the art. In Whitley, D. and Schaffer, J., editors, *Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN-92)*, pages 1–37, IEEE Press, Piscataway, New Jersey.

Sigal, N. and Alberts, B. (1972). Genetic recombination: The nature of a crossed strand-exchange between two homologous DNA molecules. *Journal of Molecular Biology*, 71(3):789–793.

Spears, W. (1995). Speciation using tag bits. In *Handbook of Evolutionary Computation*. IOP Publishing Ltd. and Oxford University Press, Oxford, UK.

Thierens, D. (1996). Non-redundant genetic coding of neural networks. In *Proceedings of the IEEE International Conference on Evolutionary Computation*, pages 571–575, IEEE Press, Piscataway, New Jersey.

Watson, J. D. et al. (1987). *Molecular Biology of the Gene Fourth Edition*. The Benjamin Cummings Publishing Company, Inc., Menlo Park, California.

Whitley, D. et al. (1993). Genetic reinforcement learning for neurocontrol problems. *Machine Learning*, 13:259–284.

Wieland, A. (1991). Evolving neural network controllers for unstable systems. In *Proceedings of the International Joint Conference on Neural Networks*, pages 667–673, IEEE Press, Piscataway, New Jersey.

Yao, X. (1999). Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447.

Yao, X. and Liu, Y. (1996). Towards designing artificial neural networks by evolution. *Applied Mathematics and Computation*, 91(1):83–90.

Yao, X. and Shi, Y. (1995). A preliminary study on designing artificial neural networks using coevolution. In *Proceedings of the IEEE Singapore International Conference on Intelligent Control and Instrumentation*, pages 149–154, IEEE Singapore Section.

Zhang, B.-T. and Mühlenbein, H. (1993). Evolving optimal neural networks using genetic algorithms with Occam's razor. *Complex Systems*, 7:199–220.