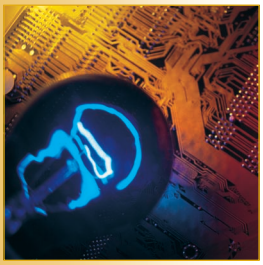

Faster Web Page Allocation with Neural Networks

Exploiting the self-similarity of Web content, the self-organizing neural network improves performance of distributed Web server systems.

Vir V. Phoha
Louisiana Tech University

S. Sitharama Iyengar and Rajgopal Kannan
Louisiana State University

Internet traffic has doubled almost every year since 1997, a growth rate set to continue for some time.[1] To maintain quality of service, some heavily trafficked Web sites use multiple servers, which share information through a shared file system or data space. The Andrews file system (AFS) and distributed file system (DFS), for example, can facilitate this sharing. In other sites, each server might have its own independent file system.

Although scheduling algorithms for traditional distributed systems do not address the special needs of Web server clusters well, a significant evolution in the computational approach to artificial intelligence and cognitive engineering shows promise for Web request scheduling. Not only is this transformation — from discrete symbolic reasoning to massively parallel and connectionist neural modeling — of compelling scientific interest, but also of considerable practical value.

Our novel application of connectionist neural modeling to map Web page requests to Web server caches maximizes hit ratio while load balancing among caches. In particular, we have developed a new learning algorithm for fast Web page allocation on a server using the self-organizing properties of the neural network (NN).

Current Approaches to Request Routing

Figure 1 shows a multiple server system. Requests come from various clients, and the router pools the requests and directs them to different servers.[2] Each server, S_1, S_2, ..., S_N has a cache.

Four basic approaches to routing requests among distributed Web-server nodes exist:[3]

- client based,

 1089-7801/02/$17.00 ©2002 IEEE

- DNS based,
- dispatcher based, and
- server based.

In the client-based approach, requests can be routed to any Web server architecture, even if the nodes are loosely connected or uncoordinated. Embedded code in either the Web client (typically a browser) or a client-side proxy server makes routing decisions. For example, Netscape spreads the load among various servers by selecting a random number i between 1 and the number of servers and directs the requests to the server www$i$.netscape.com. This approach is not easily scalable, however, and few Web sites have dedicated browsers to distribute the server load. A combination of caching and server replication could enhance performance.[4] However, client-side proxy servers require Internet component modification that are beyond the control of many institutions that manage Web server systems.

The DNS can implement a large set of scheduling policies by translating a symbolic name to an IP address. UDP packet size constrains the DNS approach to 32 Web servers per public URL, although it scales easily from LAN to WAN distributed systems.

If a single dispatcher controls all routing decisions, it can achieve finely tuned load balancing, but failure of the centralized controller can disable the whole system. Finally, the server-based approach uses two levels of dispatching. First, a cluster DNS assigns requests to Web servers, and then each server can reassign the request to any other server in the cluster. The server-based approach can attain as good a control as the dispatcher approach, but the redirection mechanisms typically increase user-perceived latency. To our knowledge, only the Internet2 Distributed Storage Infrastructure (I2-DSI) Project proposes a smart DNS that can make routing decisions based on network proximity information such as transmission delays.[5] No approach incorporates any kind of intelligence or learning in routing Web requests, however.

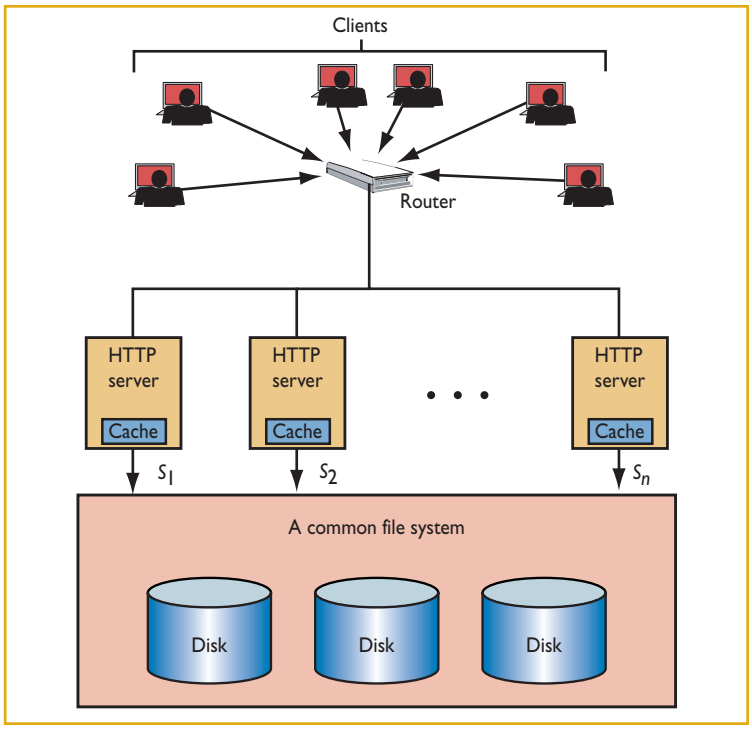

Figure 1. Typical Web page request routing. Requests come from various clients. A router pools the requests and directs them to different servers. Each server has a cache and might share a common file system or have individual storage.

Neural Networks and Competitive Learning

Many artificial NN architectures display self-organizing properties: They can extract patterns, features, correlations, or categories from input data and code them in the output. Several researchers have detected self-similarity in Internet traffic flows.[6,7] We can exploit NNs to take advantage of this self-similarity to make smarter routing decisions.

In previous work, we have used NNs' competitive learning to develop an iterative algorithm for image recovery and segmentation.[8,9] We applied Markov random fields to develop an energy function with terms corresponding to smoothing, edge detection, and preservation. An update rule, a gradient descent rule to the energy function, restores and segments images. A. Modares and colleagues use a self-organizing NN to solve multiple traveling salesman and vehicle routing problems.[10] Their algorithm shows significant advances both in the quality of solutions and computational efforts for most of the experimental data. K. Yeung and colleagues present a node placement algorithm in shuffle nets that calculates a communication cost function between a pair of nodes. It develops a gradient descent algorithm to place node pairs one by one.[11]

Special Considerations for Web Server Routing

The Secure Socket Layer (SSL) encrypts client-server communication with a session key. Session keys are expensive to generate, so each SSL request has a lifetime of about 100 seconds and the client and server use the same session key during its lifetime. Routing multiple requests from a client to one server can save the overhead of negotiating a new session key. With our competitive learning algorithm,

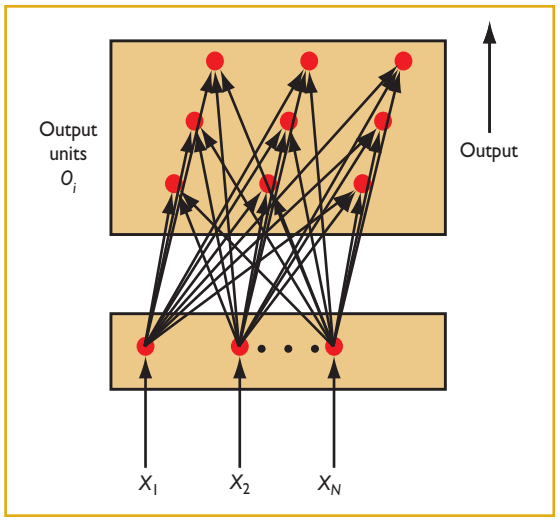

Figure 2. Kohonen's network. Input vectors x_i map to an array of output vectors O_i with the weight functions w_{ij}.

the NN naturally routes requests in this fashion. IBM's Network Dispatcher (ND) routes any two SSL requests received within 100 seconds from the same client to the same server.

In contrast, a simple round-robin scheme will be inefficient because it requires the generation of many session keys. The Web server load information becomes obsolete quickly and is poorly correlated with future load conditions.[12] Because Web dynamics involve high variability of domain and client workloads, exchanging server load condition information is not sufficient for scheduling decisions. Hence, we need a real-time adaptive mechanism that adapts rapidly to a changing environment.

Building Blocks of Our Model

As a foundation for the discussion of our model and results, we introduce the key concepts of the Pareto distribution, competitive learning, and Kohonen's algorithm.

Pareto Distribution

We can model Web traffic with a Pareto distribution. A heavy-tailed distribution, such as the Pareto, is asymptotically hyperbolic — irrespective of the short-term distribution of a random variable x, over the long run, the distribution function of x is hyperbolic. The probability mass function of a Pareto distribution is $p(x) = \gamma k^{\gamma} x^{-\gamma-1}$, $\gamma, k > 0$, $x \geq k$, and its cumulative distribution is $F(x) = P[X \leq x] = 1 - (k/x)^{\gamma}$.

Here k represents the smallest value of the random variable and γ determines the distribution's behavior. We can estimate the parameters from historical data. For example, if $\gamma = 2$, the distribution has infinite variance. If $\gamma = 1$, it has infinite mean.[13]

Competitive Learning

In the simplest competitive learning network, there is a single layer of output units $\mathbf{S} = \{S_1, S_2, ..., S_N\}$. Each output is fully connected to a set of inputs O_i via connection weights w_{ij}. A brief description of a competitive learning algorithm follows.

Let $\mathbf{O} = \{O_1, O_2, ..., O_M\}$ be an input to a network of two layers with an associated set of weights w_{ij}. The standard competitive learning rule[14] is given by $\Delta w_{i^*j} = \eta(O_j - w_{i^*j})$, which moves w_{i^*} toward O. The i^* implies that only the set of weights corresponding to the winning nodes (those with the largest output) is updated. Alternatively, $\Delta w_{ij} = \eta S_i(O_j - w_{ij})$, where

$$S_i = \begin{cases} 1 \text{ for } i \text{ corresponding to} \\ \quad \text{the largest output} \\ 0 \text{ otherwise} \end{cases}$$

This is the adaptive approach taken by Kohonen in his first algorithm (see Kohonen model below).[15] The usual definition of competitive learning requires a winner-take-all strategy. In many cases this requirement is relaxed to update all of the weights in proportion to some criterion. This form of competitive learning is referred to as *leaky learning*. Hertz and colleagues discuss various forms of this adaptive processing for different problems including the traveling salesman problem.[14] It has become a standard practice to refer to all of these as Kohonen-like algorithms.

Kohonen's Algorithm

Kohonen's algorithm adjusts weights from common input nodes to N output nodes arranged in a 2D grid, as Figure 2 shows, to form a vector quantizer.[14,15] After the trainer presents enough input vectors sequentially in time, the weights specify clusters or vector centers. These vector centers sample the input space such that their point density functions approximate the probability density functions of the input vectors. The algorithm also organizes weights such that topologically close nodes are sensitive to physically similar inputs. Output nodes are thus ordered naturally. This algorithm, described below, forms feature maps of the inputs.

Let $x_1, x_2, ..., x_N$ be a set of input vectors, which defines a point in N-dimensional space. The output units O_i are arranged in an array and are fully connected to the inputs via the weights w_{ij}. A com-

Definitions and Background

A *distributed Web server system,* is any architecture of multiple Web server hosts that can spread client requests to the servers. The traffic load on the Web site is the number of HTTP requests it handles. A *session* is the time a single user spends on a given Web site. A session can issue many HTML page requests. Typically, a Web page consists of a collection of objects, and an object request requires accessing a server. Any access to a server for an object is defined as a *hit.* We use *site* and *Web site* interchangeably to mean a collection of Web objects meant to be served. An *object* is an entity such as a Web page or image served by the server. An *object-id* uniquely identifies an object in a Web page. An object-id can be a URL, a mapping of a URL to a unique numerical value, and so on; however, we follow only one method consistently.

N servers—identified as $S_1, S_2, \ldots, S_N$—service Web object requests. The Web-server cluster is scalable and uses one URL to provide a single interface to users. For example, a single domain name can be associated with many IP addresses, and each address can belong to a different Web server. The collection of Web servers is transparent to the users.

Each Web page request is identified as a duplet $<O_i, R_i>$ where $1 \le i \le M$. O_i is the object identifier of the object requested, R_i is the number of requests for the object served so far, and M is the total number of objects present at the site.

Self-similarity is usually associated with fractals, the shapes of which are similar regardless of magnification. For example, a coastline has similar structure at a scale of one mile, 10 miles, or 200 miles. In case of stochastic phenomenon, such as a time series, self-similarity means that the object's correlational structure remains unchanged at different timescales. Both Ethernet and Web traffic are self-similar.[1,2]

References

1. W.E. Leland et al., "On the Self-Similar Nature of Ethernet Traffic (extended version)," *IEEE/ACM Trans. Networking,* vol. 2, no. 1, Jan. 1994, pp. 1-15.
2. M. E. Crovella and A. Bestavros, "Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes," *IEEE/ACM Trans. Networking,* vol. 5, no. 6, Dec. 1997, pp. 835-846.

petitive learning rule is used to choose a winner unit i^*, such that $|w_{i^*} - x \le w_i - x|$ for all i.

Then the Kohonen's update rule is given by

$$\Delta w_i = \eta\, h\,(i, i^*)(x - w_i^{Old}).$$

Here, $h(i, i^*)$ is the neighborhood function such that $h(i, i^*) = 1$ if $i = i^*$, but falls off with distance $|r - r_i^*|$ between units i and i^* in the output array. The winner and nearby units are updated appreciably more than those farther away. A typical choice for $h(i, i^*)$ is

$$e^{-\left(\frac{|r_i - r_i^*|}{2\sigma 2}\right)},$$

where σ is a parameter that is gradually decreased to contract the neighborhood, and η (in the update rule) is decreased to ensure convergence.

Using principles derived from competitive learning, and building on earlier work on the self-organizing properties of NNs,[14] we propose and evaluate an NN approach to routing Web page requests in a multiple-server environment. The scheduling algorithms for traditional distributed systems are not applicable to Web server clusters because the loads from different client domains are nonuniform, the real Web workload is highly variable, and Web requests have a high degree of self-similarity (see the sidebar, "Definitions and Background").

Conceptual Framework

Our model ensures high Web server performance using NNs. Caching data at the client site can improve Web site performance by reducing server overloads. Cache capacity limits this approach, however, which is also very inefficient for serving dynamic Web objects. The data path may include several layers of software and file systems, intermediate processing steps, object fetches from cache, and, finally, operating system layers of the client machine. To handle increasing Web traffic, we propose a technique to assign Web objects to the server cache at the router level. This router can be one of the servers or a dedicated machine acting as a server.

Figure 3 (next page) presents a conceptual framework of the proposed model. The object id and request count of the most frequently accessed objects are in the router's memory. The actual objects reside in the server's cache.

The foundation of our model is the competitive learning properties of the NN. Each server competes to serve the object request. The server closest to the request wins, and if the object is already in the server's cache, we strengthen the relationship between the server and the object request. We use a competitive basis algorithm to make routing decisions.

This approach is also related to Kohonen's self-organizing feature map technique,[15] which facilitates mapping from higher to lower dimensional spaces so that the relationships among the inputs are

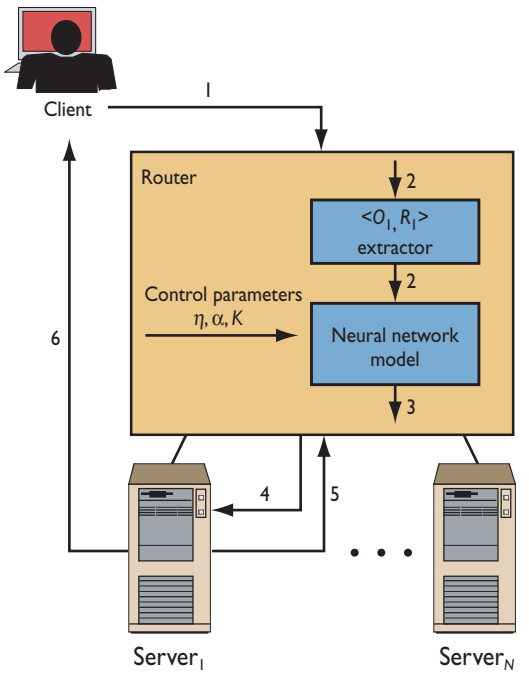

Figure 3. A conceptual framework of our model. A client requests an object (1). The router extracts the request count for the object and applies it as an input to the neural network (2). The NN determines which server to forward the request to (3). The router forwards the request to the server (4). The server presents feedback to the router (5) and sends an object response to the client (6).

mapped onto highly reliable outputs. In Kohonen's algorithm, the generated output neurons are very close to the winning neurons based on the topology structure. In our model, connection strength between the object requests and the server is analogous to weight connections between the two layers of Kohonen's model. The input layer consists of the object request and the output layer consists of the servers. In addition to mapping the input space of Web object requests to the weight space, we use this property to assign the object requests to the servers.

Description of the New Model

In our neural placement model, we define two layers in the network, the input layer W and the output layer S. W contains M nodes, where each object is assigned a node. The object request count, R_i, is applied as the input to the node corresponding to the object in the W layer. Layer S has N nodes, one for each server, and each node, j, is assigned to a server S_j. The weight w_{ij} connect snode i in the input layer to node j in the output layer. This weight forms a connection strength from an object $<O_i, R_i>$ to a server S_j. Figure 4 illustrates this architecture.

We approximate the number of input nodes (M) from the number of objects, such as images, HTML pages, and sound clips, in a given Web site. Adding or deleting input nodes (new objects) does not affect the model's existing structure. We add or delete links from the affected input node alone, without changing the remaining links or weights. In our simulations, the number (M) of objects ranged from 150 to 1,050.

Kohonen's algorithm maps the inputs' topological structure onto the weights. In our model, we map the page popularity structure of the Web object requests to the weight structure between the input and output layers and use competition to transfer the request to the winning server cache. At the same time, we add a load-balancing factor in the decision process to equitably allocate object requests among server caches.

Mathematical Formulation Using Competitive Learning

We formulate the problem of assigning Web objects to the servers as a mapping for the placement of $<O_i, R_i> \in$ W onto a server space S as

$$\varnothing_k : \mathrm{W} \rightarrow \mathrm{S}, \text{ such that } O_i \in \mathrm{W} \rightarrow S_j \in \mathrm{S}; \quad (i = 1, M \text{ and } j = 1, N)$$

We include an object classification condition such that the NN model distributes the objects O_i equally among the servers S_j, and at the same time it maximizes the server cache hits. We have two objectives:

- Increase the number of hits, in the sense that a server gets the same object request it previously served. As our simulation results show, this improves Web site performance by allowing fast response and loading of dynamic Web objects.
- Distribute the object requests equitably among servers.

Our algorithm achieves both goals by learning both the previous requests and the object request distribution.

The server k for a given object is selected as follows. An object's request count is applied as input to its corresponding node. The NN model chooses a server to forward the request to based on the lowest value of the request count mod to the weight connecting to that server. Thus,

$$|R_i - w_{ik}| = \min |R_i - w_{ij}| \text{ where } j = 1, N \qquad (1)$$

To avoid dealing with very large numbers, it is better to use $R_i/\Sigma R_i$ instead of R_i. Learning is achieved by updating the connection strength between object and winning server using the update rule

$$\Delta w_{ik} = \eta \Lambda(R_i, w_{ik}, K)(R_i - W_{ik}) + \alpha K(\sum W_{i\tau} - N W_{ik}) \quad (2)$$

Here, the parameters η, α, and K determine the strength of controlling the maximum hit or balancing the load among the servers. $\Lambda(R_i, w_{ik}, K)$ is given by

$$\Lambda(R_i, w_{ik}, K) = \frac{e^{\frac{-g*g}{(2*K*K)}}}{\Psi(d,K)}$$

where $g = (R_i - W_{ik})$, and

$$\Psi(d,K) = \sum_j e^{\frac{-d*d}{(2*K*K)}}$$

and

$$d = (R_i - W_{ij}), j = 1, N.$$

Now, by integrating Equation 2 and performing some algebraic manipulations, we get the energy function[8]

$$E = \eta K \ln \sum_{m,k} e^{\frac{-d*d}{2K*K}} + \alpha \sum_{m,k} \left(w_{i,j-1} + w_{i,j+1} - 4w_{i,j} + w_{i-1,j} + w_{i,j}\right)^2. \quad (3)$$

Because the update rule in Equation 2 is of the form $\partial E/\partial w_{ik}$, it is a gradient descent rule for the energy function in Equation 3.

Heuristics on Parameter Selection

The neighborhood function $\Lambda(R_i, w_{ik}, K)$ is 1 for $i = k$ and falls off with the distance $|R_i - w_{ij}|$. Ideally, we should select the neighborhood such that the servers with the highest mean hit count for a given object, or servers that serve related objects are close together.

The first term on the right-hand side of Equation 3 pushes the request count R_k toward w_{ik} so our scheme directs requests for objects in server S_i's cache to it. This increases the likelihood that the router will route subsequent requests for an object to the same server S_i,, thereby increasing the hit ratio. The second term on the right ensures that no one server will be overloaded — that is, the object requests are distributed evenly among the servers. By properly balancing the parameters η, α, and K, we can direct the flow of traffic in an optimal fashion. η, α, and K are related as follows:

$$\eta \propto \frac{1}{\alpha K}$$

A higher value of η (hence lower αK) stresses object hits, whereas a higher value of αK emphasizes balancing among the servers. We have achieved close to 100 percent hits using small values of α and still had good load balancing among servers.

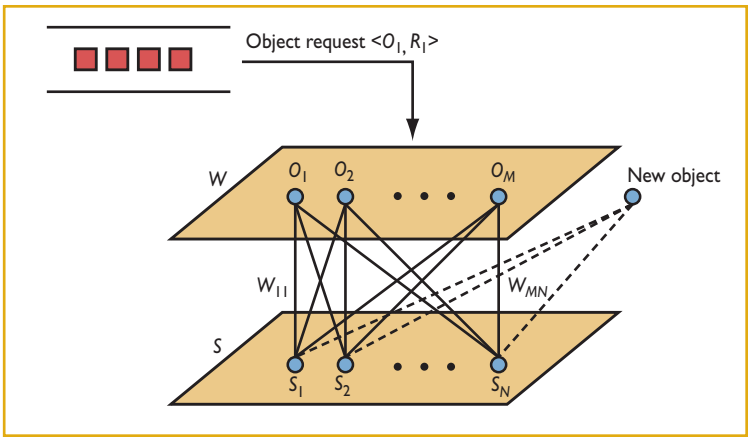

Figure 4. Neural architecture to direct Web page requests. Weights w_{ij} connect input layer nodes to output layer nodes.

The Algorithm and Its Distinctive Features

Figure 5 (next page) provides an outline of our algorithm.

Our approach has several distinctive features. First, it maps the self-similarity of Web traffic to the self-organizing property of the neural model. A Web object request must be allocated to a server. A learning rule optimizes the allocator, whereas the network adapts to unpredictable changes using the competitive learning framework. The learning process produces stable responses. Second, the learning rule's associated energy function captures global knowledge about server load. The learning rule is a gradient descent function to this energy function; we make local decisions about load balance based on global knowledge.

Furthermore, the algorithm creates a balance between conflicting requirements to equitably distribute Web requests among servers and simultaneously maximize hit ratio. Finally, though the

```
Initialize M to the number of objects and N to the number of servers;
Initialize with random values the weights {w_ij} between the page requests and the servers and select
parameters η, α, and K.
While (server available)                    //Service page requests until server is running.
  {    //begin while//
       Calculate |R_i - w_ik| and using Equation 1 select server S_k;
       if (<O_i, R_i> is in the cache of  S_k )
  {
              Update the weight using Equation 2
              //This is a hit so we want the algorithm to strengthen this connection//
  }
  else
  {
       Find the correct server S_j using |R_j - w_ij|
       Update the connection weight for this server using Equation 2
       // In this case this is a miss-hit so we want the algorithm to learn the
       //correct response by strengthening the appropriate weight //
  }
} //end while //
```

Figure 5. Algorithm for routing Web page requests in a multiple-server environment. The algorithm distributes Web requests equitably among servers and simultaneously maximizes hit ratio.

Table 1. Simulation characteristics.

Sample size	Number of Web objects ranged from 150 to 1,050. We collected statistics at 50-object intervals.
Number of servers	Statistics were collected for 4, 8, 16, and 32 servers
Web page distribution	Uniform and nonuniform (Pareto)
Algorithms	NN, round-robin and round-robin 2, and adaptive time-to-live

algorithm currently uses a winner-take-all strategy, we can incorporate leaky learning by defining a neighborhood of servers whose weights we wish to update. Proportional weights can help us account for server interconnection and dependencies between objects on closely connected servers.

Experimental Results

Our model incorporates the characteristics of Web traffic and its inherent self-similarity with a heavy-tailed distribution of the type $P[X > x] \sim x^{-\gamma}$ as $x \to \infty$ for $0 < \gamma < 2$. Our results correspond to a Pareto distribution, with probability density function $p(x) = \gamma k^{\gamma} x^{-\gamma-1}$, where $\gamma = 0.9$ and $k = 0.1$.

We conducted the simulations in two separate environments.

- At the Louisiana State University Networks Lab, we simulated Web traffic using a Pareto distribution on a single PC.
- At the Computer Science laboratory at Louisiana Tech University, the simulation environment consisted of a network of five IBM PCs. We used one PC as a dedicated Web server written in Java specifically for this experiment; another as a proxy server; and the three remaining PCs as clients, with Microsoft Internet Explorer (IE) and Netscape browser settings pointed to the proxy server for all requests. Each client PC simulated 300 clients by creating separate threads (using Java) for each client. We implemented the neural model on the PC running the proxy server.

We compared the performance of our algorithm with round-robin (RR), round-robin 2 (RR2), and a special case of the adaptive time-to-live (TTL) algorithm. RR2 partitions a Web cluster into two classes — normal domains and hot domains — based on domain load information. RR2 applies a round-robin scheme to each domain separately.[12] To reduce the skew on Web objects in our adaptive TTL implementation, we assign a lower TTL value when a request originates from a hot domain, and a higher TTL value when it originates from a normal domain.

Table 1 summarizes the characteristics of our simulations.

The comparison charts in Figures 6 and 7 relate only to the RR scheme and our NN-based algorithm. The hit ratios for adaptive TTL varied widely with Web object size and distribution, but never

ranged higher than 0.68.

As Figure 6 shows, the NN competitive learning algorithm performs much better than both RR schemes when input pages follow a Pareto distribution. As the number of input objects increases, our algorithm achieves a hit ratio close to 0.98, whereas the RR schemes never achieve a hit ratio of more than 0.4.

For the NN algorithm, lower hit ratios (0.86) with fewer objects is attributed to some learning on the part of the algorithm, but as the algorithm learns, the hit ratio asymptotically stabilizes to 0.98 for a larger number of objects.

For uniform distribution of input objects, the NN algorithm performs much like it does for nonuniform distribution and much better than the RR schemes (see Figure 7).

Table 2 shows that RR never achieves a hit ratio higher than 0.32, whereas NN achieves hit ratios close to 0.98. In the worst case (minimum hit ratio), NN does not fall below 0.85, whereas RR schemes go as low as a 0.02 hit ratio.

The NN algorithm's performance improves considerably with increased traffic, whereas the performance of RR remains the same or worsens slightly. This result holds true irrespective of the number of servers. This results from pushing an object toward the same server based on the learning component in Equation 2. For a nonuniform distribution (Pareto distribution), our algorithm performs considerably better for lower and higher traffic rates, irrespective of the number of servers. Similar results hold for a uniform distribution.

For the nonuniform Pareto distribution, which closely models real Web traffic, our algorithm's improved performance for large numbers of Web objects is a very attractive result. Our simulation results show an order-of-magnitude improvement over some existing techniques.

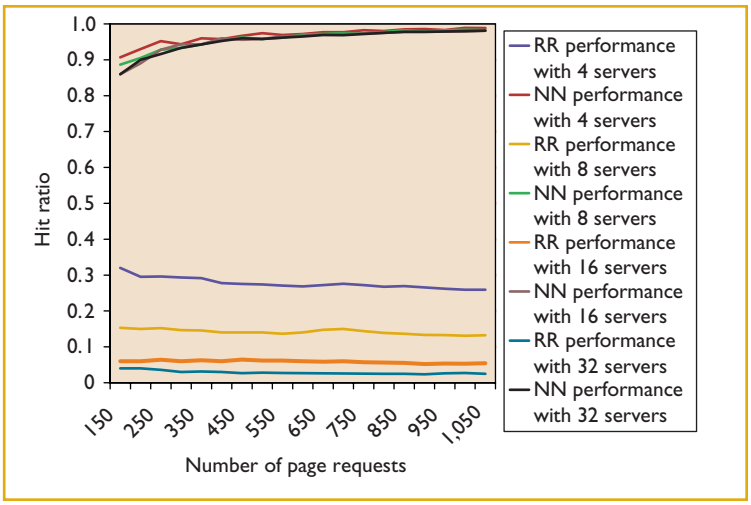

Figure 6. Performance for a nonuniform input data distribution. The page placement algorithm using competitive learning (NN), achieves high hit ratios for four server configurations and outperforms the round-robin (RR) scheme.

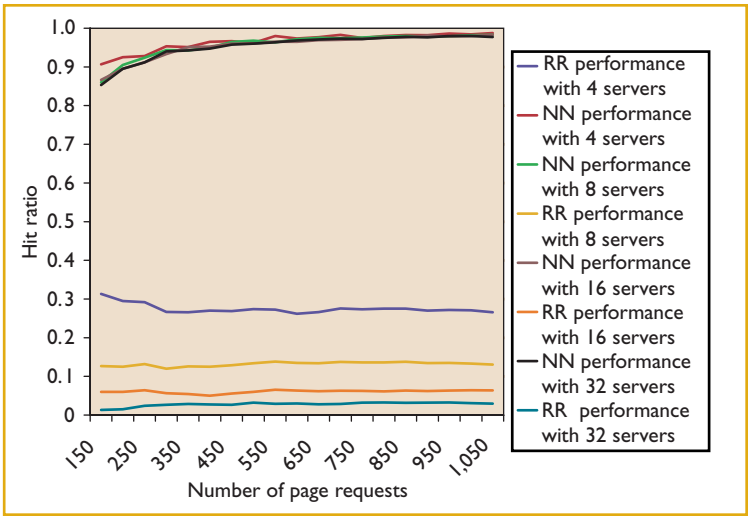

Figure 7. Performance for a uniform input data distribution. The NN scheme outperforms the RR scheme for all configurations, similar to the nonuniform input distribution case.

Table 2. Maximum and minimum hit ratios for RR and NN with input size and number of servers.

Distribution	Round Robin	Neural Network
Uniform, max	(0.31,150,4)	(0.98,1050,4)
Uniform, min	(0.03,1050,32)	(0.85,150,32)
Nonuniform, max	(0.32,150,4)	(0.98,1050,4)
Nonuniform, min	(0.02,1050,32)	(0.86,150,32)

Conclusion

We are currently developing methodologies to include Web page metadata in our algorithm to further improve our approach. The metadata is part of the feature vector used to train the NN, and for new requests it is helpful in directing the requests to the appropriate Web servers.

References

1. K.G. Coffman and A.G. Odlyzko, *Growth of the Internet*, tech. report, AT&T Labs–Research, www.dtc.umn.edu/~odlyzko/doc/oft.internet.growth.pdf.
2. A. Iyengar and J. Challenger, "Improving Web Server Performance by Caching Dynamic Data," *Proc. Usenix Symp. Internet Technologies and Systems*, Usenix Assoc., Berkeley, Calif., 1997, pp. 49-60.
3. V. Cardellini, M. Colajanni, and P. Yu, "Dynamic Load Bal-

ancing on Web-Server Systems," *IEEE Internet Computing,* vol. 3, no. 3, May/June 1999, pp. 28-39.
4. M. Baentsch, L. Baum, and G. Molter, "Enhancing the Web's Infrastructure: From Caching to Replication," *IEEE Internet Computing,* vol. 1, no. 2, Mar/Apr. 1997, pp. 18-27.
5. M. Beck and T. Moore, "The Internet2 Distributed Storage Infrastructure Project: An Architecture for Internet Content Channels," *Proc. 3rd Workshop WWW Caching,* Manchester, UK, 1998, pp. 2141-2148.
6. M. Grossglauser and J.-C. Bolot, "On the Relevance of Long-Range Dependence in Network Traffic," *IEEE/ACM Trans. Networking,* vol. 7 no. 5, Oct. 1999, pp. 629-640.
7. W.E. Leland, M.S. Taqqu, and D. V. Wilson, "On the Self-Similar Nature of Ethernet Traffic (Extended Version)," *IEEE/ACM Trans. Networking,* vol. 2, no. 1, Feb. 1994, pp. 1-15.
8. V. Phoha, *Image Recovery and Segmentation Using Competitive Learning in a Computational Network,* doctoral dissertation, Texas Tech Univ., Lubbock, 1992.
9. V. Phoha and W.J.B. Oldham, "Image Restoration and Segmentation Using Competitive Learning in a Layered Network," *IEEE Trans. Neural Networks,* vol. 7, no. 4, July 1996, pp. 843-856.
10. A. Modares, S. Somhom, and T. Enkawa, "A Self-Organizing Neural Network for Multiple Traveling Salesman and Vehicle Routing Problems," *Int'l Trans. Operational Research*, vol. 6, 1999, pp. 591-606.
11. K. Yeung and T. Yum, "Node Placement Optimization in Shuffle Nets," *IEEE/ACM Trans. Networking,* vol. 6, no. 3, June 1998, pp. 319-324.
12. V. Cardellini, M. Colajanni, and P. Yu, "DNS Dispatching Algorithms with State Estimators for Scalable Web-Server Clusters," *World Wide Web J.,* vol. 2, no. 2, July 1999, pp. 101-113.
13. V. Paxson and S. Floyd, "Wide-Area Traffic: The Failure of Poisson Modeling," *IEEE/ACM Trans. Networking,* vol. 3, no. 3, June 1995, pp. 226-244.
14. J. Hertz et al., *Introduction to the Theory of Neural Computation,* Lecture Notes on Computer Science, vol. 1, Addison-Wesley, Boston, 1991.
15. T. Kohonen, *Self-Organization and Associative Memory,* 3rd ed., Springer-Verlag, Berlin, 1989.

Acknowledgments

We thank Ashok Chigide and Wen Tian for their help in carrying out the simulations, and Sunil Babu for help in drawing the figures.

Vir V. Phoha is an associate professor of computer science at Louisiana Tech University in Ruston. He received an MS and a PhD in computer science from Texas Tech University. His research interests include Web caching, Web mining, network and Internet security, intelligent networks, and nonlinear systems. He is a senior member of the IEEE and a member of the ACM.

S. Sitharama Iyengar is a distinguished research master award-winning professor of the Computer Science Department at Louisiana State University in Baton Rouge. He received an MS from the Indian Institute of Science and a PhD from Mississippi State University. He is presently a chaired professor and chair of the department at Louisiana State University. His research interests include high-performance parallel and distributed algorithms and data structures for image processing and pattern recognition, and distributed data-mining algorithms. He is a fellow of the IEEE, the ACM, and the American Association of Advancement of Science.

Rajgopal Kannan is an assistant professor of computer science at Louisiana State University in Baton Rouge. He obtained a BTech degree in computer science and engineering from the Indian Institute of Technology–Bombay and a PhD in mathematics and computer science from the University of Denver. His research interests include ATM, multicast switching and protocols, optical switching networks, distributed sensor networks, and game theoretic models for network design and control.

Readers can contact the authors at phoha@latech.edu, {iyengar,rkannan}@bit.csc.lsu.edu.