Connectionist Supervised Learning: An Engineering Approach

Thomas G. Dietterich
Department of Computer Science
Oregon State University
Corvallis, OR 97331

Andreas S. Weigend
Department of Computer Science
University of Colorado
Boulder, CO 80302

Outline

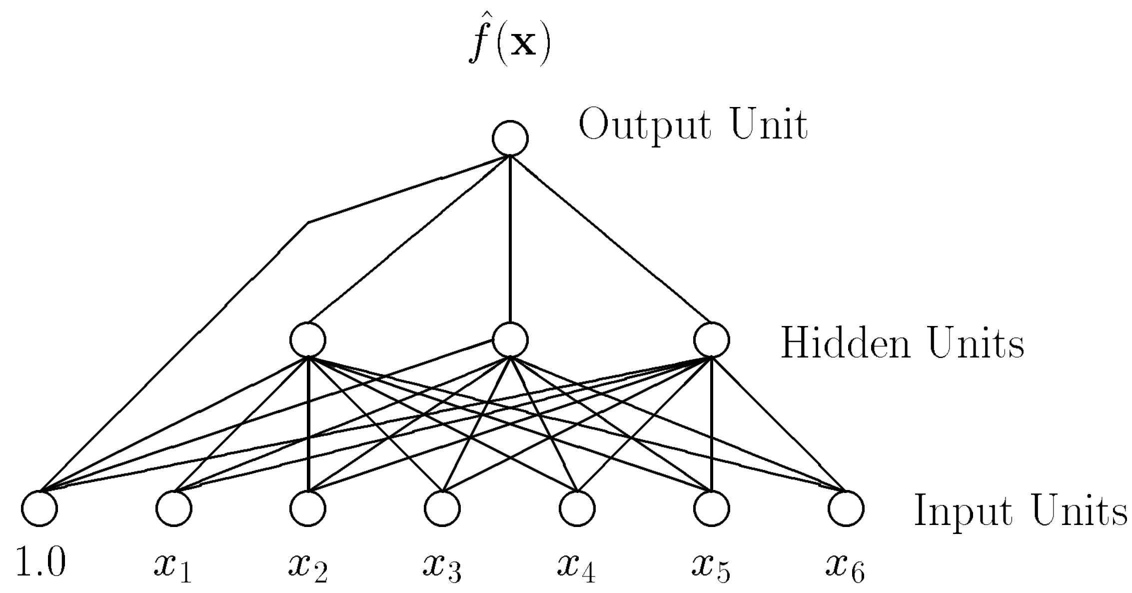
- Representation
- Learning Algorithms
- Examples
- Strengths and Weaknesses
- What we left out

Learning from Examples: Notation

instance	indexed by i.
feature vector	represents instance i as $\mathrm{x}_i \in R^n$.
unknown function	$f : R^n \mapsto R^k$
hypothesis	$\hat{f}$
training example	a pair: $\langle \mathrm{x}_i, f(\mathrm{x}_i) \rangle$.
	or, with additive noise: $\langle \mathrm{x}_i, f(\mathrm{x}_i) + \epsilon \rangle$.
training set	a set S of training examples. $\lvert S \rvert = m$.

Representation

We represent our hypotheses by so-called "three layer networks"



The circles are called "units".

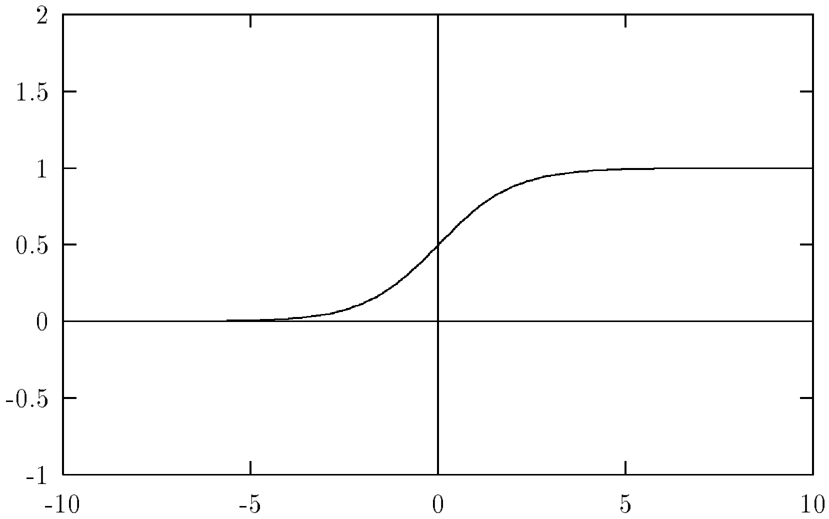
Each edge in the graph corresponds to a parameter value or "weight".

The input units do no computation.

The hidden and output units compute $\sigma(\mathbf{w} \cdot \mathbf{Y})$.

Sigmoid Function

$$\sigma(x) = \frac{1}{1+e^{-x}}.$$



A three-layer network in algebra:

$$\hat{f}(\mathrm{x}, W) = \sigma\left(\mathrm{w}_o \cdot [\sigma(\mathrm{w}_1 \cdot \mathrm{x}), \sigma(\mathrm{w}_2 \cdot \mathrm{x}), \ldots, \sigma(\mathrm{w}_h \cdot \mathrm{x})]\right)$$

The Bias Input

The sigmoid function can approximate a linear function:

$$y = w_0 + w_1 x_1 + \ldots + w_n x_n$$

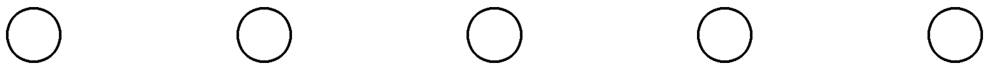
The weight w_0 is the *intercept term.* To make the computation uniform, we can pretend there is an x_0 feature whose value is fixed at 1.0.

Using this convention, w_0 is called the "bias term."

Input Encodings

Continuous features: Scale to zero mean, unit standard deviation.

Nominal features: Each feature value as a separate boolean.



	mild	**moderate**	**normal**	**elevated**	**unmeasured**
mild:	**1**	**0**	**0**	**0**	**0**
moderate:	**0**	**1**	**0**	**0**	**0**
normal:	**0**	**0**	**1**	**0**	**0**
elevated:	**0**	**0**	**0**	**1**	**0**
unmeasured:	**0**	**0**	**0**	**0**	**1**

Goal: Initial value of $\mathrm{w} \cdot \mathrm{x}$ should be in the "linear range" of σ.

Output Encodings: Classification

For classification tasks, where $f(\mathrm{x}) \in \{c_1, \ldots, c_k\}$, use one binary output for each class.

$$\begin{array}{cccccc} & c_1 & c_2 & c_3 & c_4 & c_5 \\ c_2: & 0 & 1 & 0 & 0 & 0 \\ & \circ & \circ & \circ & \circ & \circ \end{array}$$

SoftMax Outputs

If we wish to interpret the outputs as a vector of class probabilities of the form

$$P(f(\mathrm{x}) = c_1|\mathrm{x}), \ldots, P(f(\mathrm{x}) = c_k|\mathrm{x})$$

then it can often be useful to employ the *softmax* transformation:

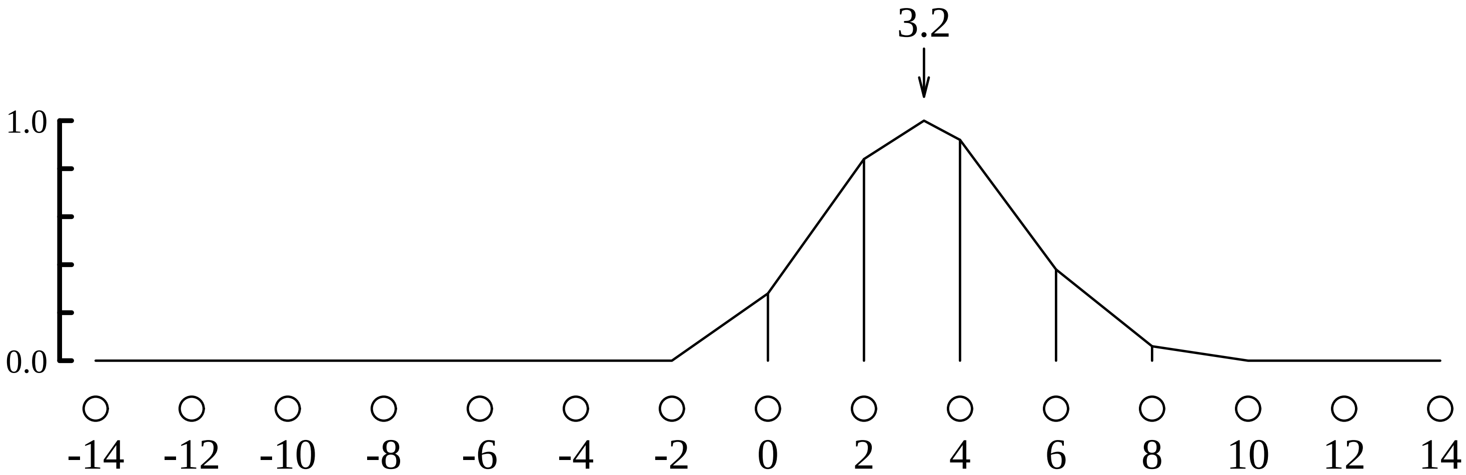
$$Q_j(\mathrm{x}) = \frac{e^{f_j(\mathrm{x})}}{\Sigma_{l=1}^{k} e^{f_l(\mathrm{x})}}$$

This normalizes all of the output activations so that they sum to 1.

Continuous Outputs

Direct Method: Use a linear output unit: $\mathbf{w}_o \cdot \langle \text{hidden activations} \rangle$

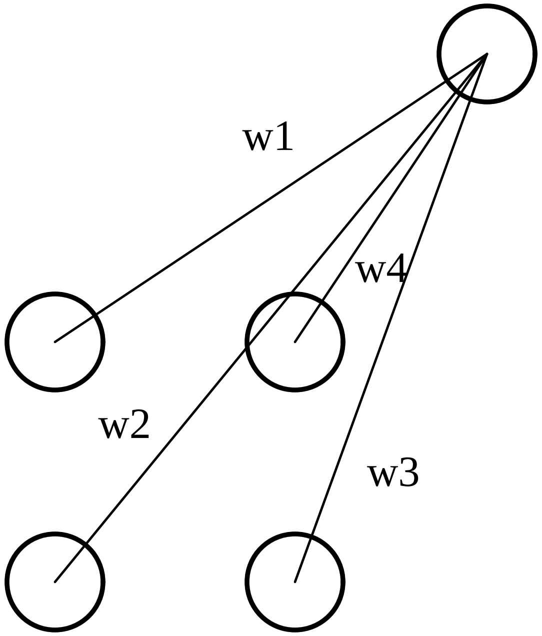
Indirect Method: Overlapping range units:



Weight Sharing

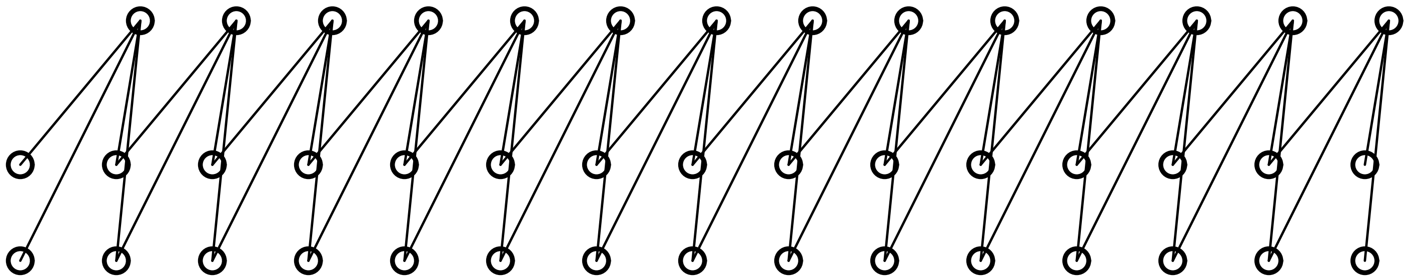
In image processing and speech recognition, we often want a hidden unit to act as a *position-independent feature detector.*

For example, this hidden unit would analyze four input values and detect a particular “feature”:



Weight Sharing (2)

We can replicate this feature detector across an entire array of data. When we update any of the weights on the connections, we are only updating one of the four weight values w_1, w_2, w_3, and w_4.



Outline

- Representation
- **Learning Algorithms**
  - **Objective Functions**
  - **Optimization Algorithms**
  - **Control of Overfitting**
- Examples
- Strengths and Weaknesses
- What we left out

Learning Algorithms

Strategy:

- **Define an error measure** J
- **Compute derivative of** J
- **Employ gradient descent to minimize** J

Error Measures

For a network with a single output unit:

- **Squared Error**

$$J(S,W) = \sum_{\mathrm{x}\in S} [f(\mathrm{x}) - \hat{f}(\mathrm{x},W)]^2$$

- **Cross Entropy**

$$J(S,W) = \sum_{\mathrm{x}\in S} [-f(\mathrm{x})\log \hat{f}(\mathrm{x},W) \; - \; (1-f(\mathrm{x}))\log(1-\hat{f}(\mathrm{x},W))]$$

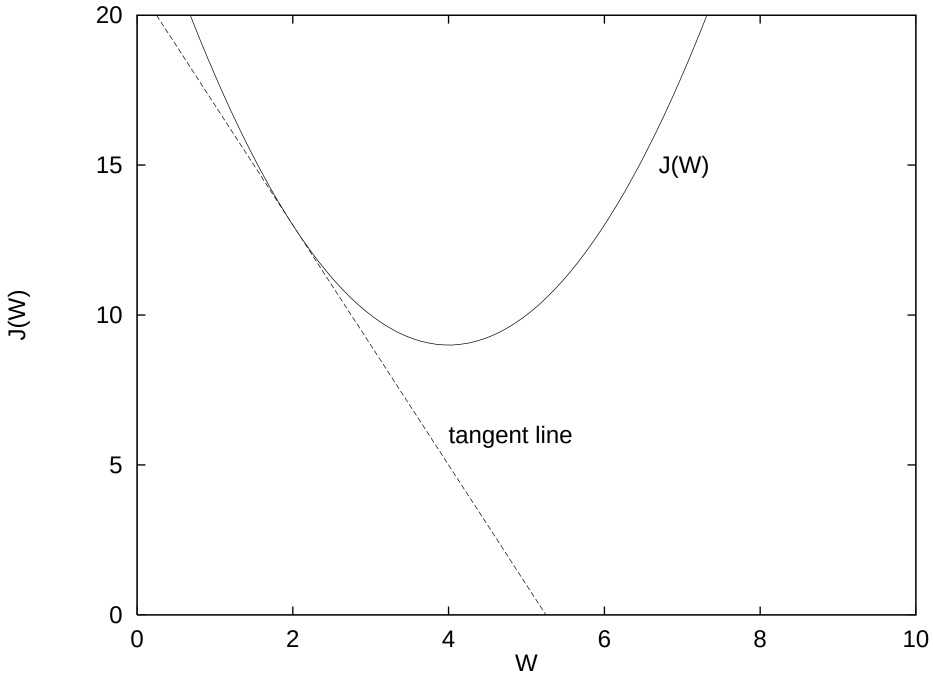
For multiple outputs, sum for each output.

Gradient Search Methods

Approximate J(W) by the first two terms of the Taylor's series:

$$J(S, W) \doteq J(S, W_0) + \frac{dJ(S, W)}{dW}(W - W_0).$$

To decrease J, W should move in the direction opposite the first derivative of $J(S, W)$.



Simple Gradient Descent

The gradient is the vector of first partial derivatives:

$$\nabla_{\mathrm{w}} J(S, W) = \left(\frac{\partial J(S,W)}{\partial w_1}, \ \ldots, \ \frac{\partial J(S,W)}{\partial w_N} \right).$$

Take a step in weight space in direction of negative derivative.

$$W := W - \eta \nabla_W J(S, W)$$

The parameter η is the learning rate.

Computing the gradient requires one complete pass through all of the training data. This is called an *epoch*.

Initializing Gradient Descent

The search must start from some starting point.

Goal: All units should be in their "linear" region.

For each unit, the weights can be initialized to

$$[-1, +1] \times \frac{1}{\sqrt{\text{fan-in}}}.$$

Stochastic Gradient Descent

Update the weights for each training example one-at-a-time, rather than summing over all of S:

$$W := W - \eta \nabla_W J(\mathrm{x}_i, W)$$

- Much faster to compute.
- Converges faster in practice.
- Injects some noise into the search—escapes local minima.

Stochastic Gradient Descent with Momentum

To reduce the "noise", we can smooth the weight updates.

$$\Delta W := -\eta \nabla_W J(\mathrm{x}_i, W) + \mu \Delta W.$$

μ is typically set in the range [0.7,0.95].

Conjugate Gradient Search

1. Compute full gradient.
2. Search along $J(S, W - \alpha\nabla_W J(S, W))$ to find the value of α that minimizes J.
3. Repeat from step 1 until J ceases to improve.

- **Advantage:** Automatically chooses learning rate.
- **Disadvantage:** Slow if S is large.

Second-Order Methods

We can also approximate $J(S, W)$ by a *quadratic* Taylor's series expansion:

$$J(S, W) \doteq J(S, W_0) + \frac{dJ(S, W)}{dW}(W - W_0) + \frac{1}{2}(W - W_0)^T H(W - W_0).$$

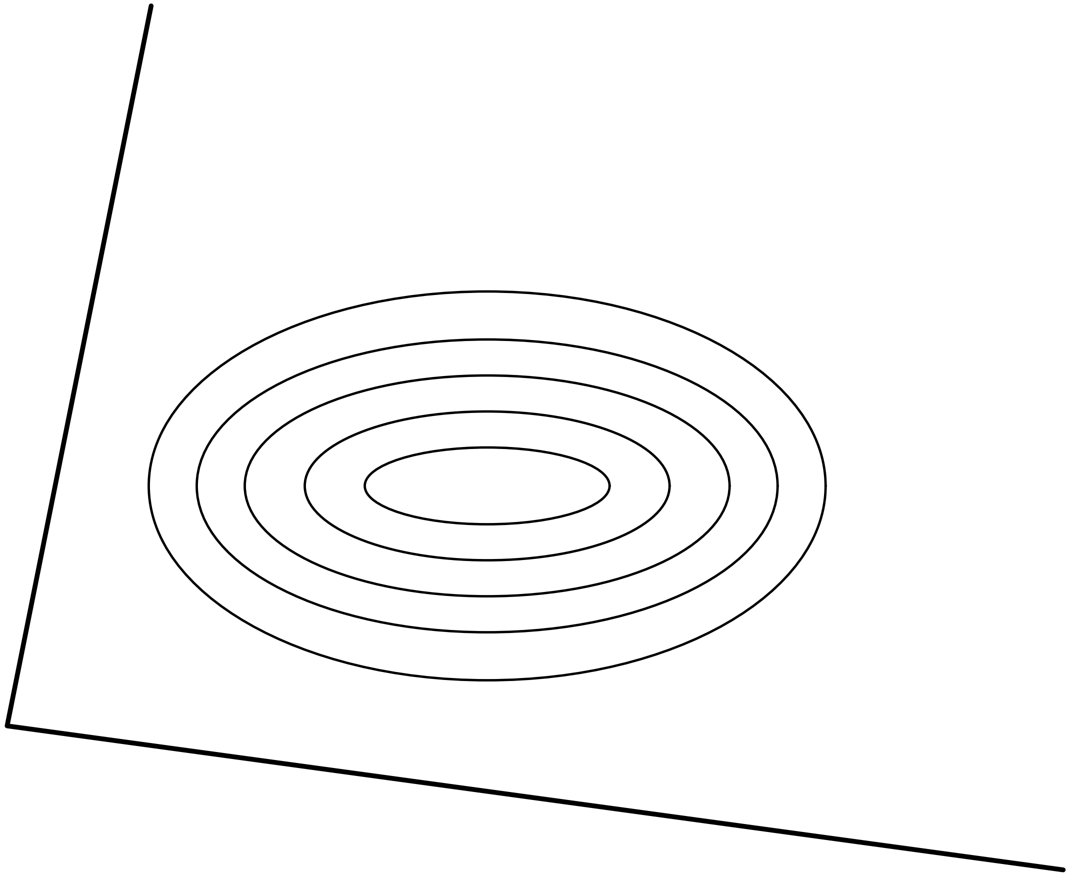
Here H is the *Hessian Matrix*:

$$\begin{bmatrix} \frac{\partial^2 J(S,W)}{\partial w_1 \partial w_1} & \ldots & \frac{\partial^2 J(S,W)}{\partial w_1 \partial w_N} \\ \ldots & & \ldots \\ \frac{\partial^2 J(S,W)}{\partial w_N \partial w_1} & \ldots & \frac{\partial^2 J(S,W)}{\partial w_N \partial w_N} \end{bmatrix}$$

When N is small, we can compute this and solve for the minimum of the quadratic. However, this is rarely practical.

Hessian Properties

The Hessian captures the curvature of the error surface.



Hessian Properties (2)

The Hessian can be diagonalized into the form:

$$H = R\Lambda R^T$$

Where Λ is a vector of *eigenvalues.*
and R is a matrix that rotates the error surface to be axis-aligned.
Each eigenvalue tells how quickly ∇J changes along the rotated axes.

Ideal Learning Rates

The learning rate η should be set to

$$\eta_{opt} = \frac{1}{\lambda_{max}}.$$

This ensures that we don't overshoot the quadratic minimum.

λ_{max} can be estimated by $\|\mathrm{v}\|$, where

$$\mathrm{v} := (1-\gamma)\mathrm{v} + \gamma \left[\frac{\nabla J(W + \alpha \mathcal{N}(\mathrm{v})) - \nabla J(W)}{\alpha} \right]$$

This formula is iterated for 50-100 training examples.
$\mathcal{N}(\mathrm{v})$ normalizes v to be a unit vector.

γ and α are small constants (e.g., 0.01).

Outline

- Representation
- **Learning Algorithms**
  - **Objective Functions**
  - **Optimization Algorithms**
  - **Control of Overfitting**
- Examples
- Strengths and Weaknesses
- What we left out

Degrees of Freedom

- **Representation**
- **Architecture**
- **Cost Function**
- **Minimization Algorithm**

Control of Overfitting (in trees)

In decision-tree algorithms, it is always possible to obtain a tree that classifies the training data correctly (unless the same x vector appears with different values of $f(\mathrm{x})$ in S).

However, such a tree may *overfit* the data and give poor predictions on new data points.

Two approaches to preventing overfitting:

- **Early Stopping.**
  Halt the growth of the tree before complete fit to the data.
- **Pruning.**
  Grow the tree to fit the data.
  Prune branches to improve generalization.

Control of Overfitting (in nets)

- **Representation**
  - **Smoothing**
  - **Generate pseudo-data that incorporate your beliefs**
- **Architecture**
  - **Hard-Code Constraints, Invariances**

    Hard weight-sharing/linked-weights.
  - **Pruning**

    Train an overly-large network. Delete weights.
  - **Network Growing**

    Add units incrementally during training.
- **Cost Function (Regularization)**

  Add a "complexity" penalty to the objective function J.
- **Early Stopping**

  Halt gradient search before a minimum is reached.

Pruning Methods

Recall the Taylor's series expansion of J:

$$J(S,W) \doteq J(S,W_0) + \frac{dJ(S,W)}{dW}(W - W_0) + \frac{1}{2}(W - W_0)^T H (W - W_0).$$

At a local minimum, the second term (gradient) is 0.
Rewrite as

$$J(S,W) - J(S,W_0) \doteq \frac{1}{2}(W - W_0)^T H (W - W_0).$$

The effect of setting any weight w_l to 0 could be determined if we knew the Hessian.

Pruning methods approximate the Hessian in various ways to choose weights to eliminate.

Weight Pruning (2)

- **Optimal Brain Damage:**
  Approximate Hessian by diagonal matrix.
  Iteratively prune network, then retrain.
- **Optimal Brain Surgeon:**
  Approximate Hessian via recursive covariance computation.
  Can optimally modify weights without retraining.
- **Small Eigenvectors:**
  Compute small eigenvectors of Hessian.
  Prune along those directions.

Optimal brain surgeon appears to be the best of these methods, but it is also the most expensive.

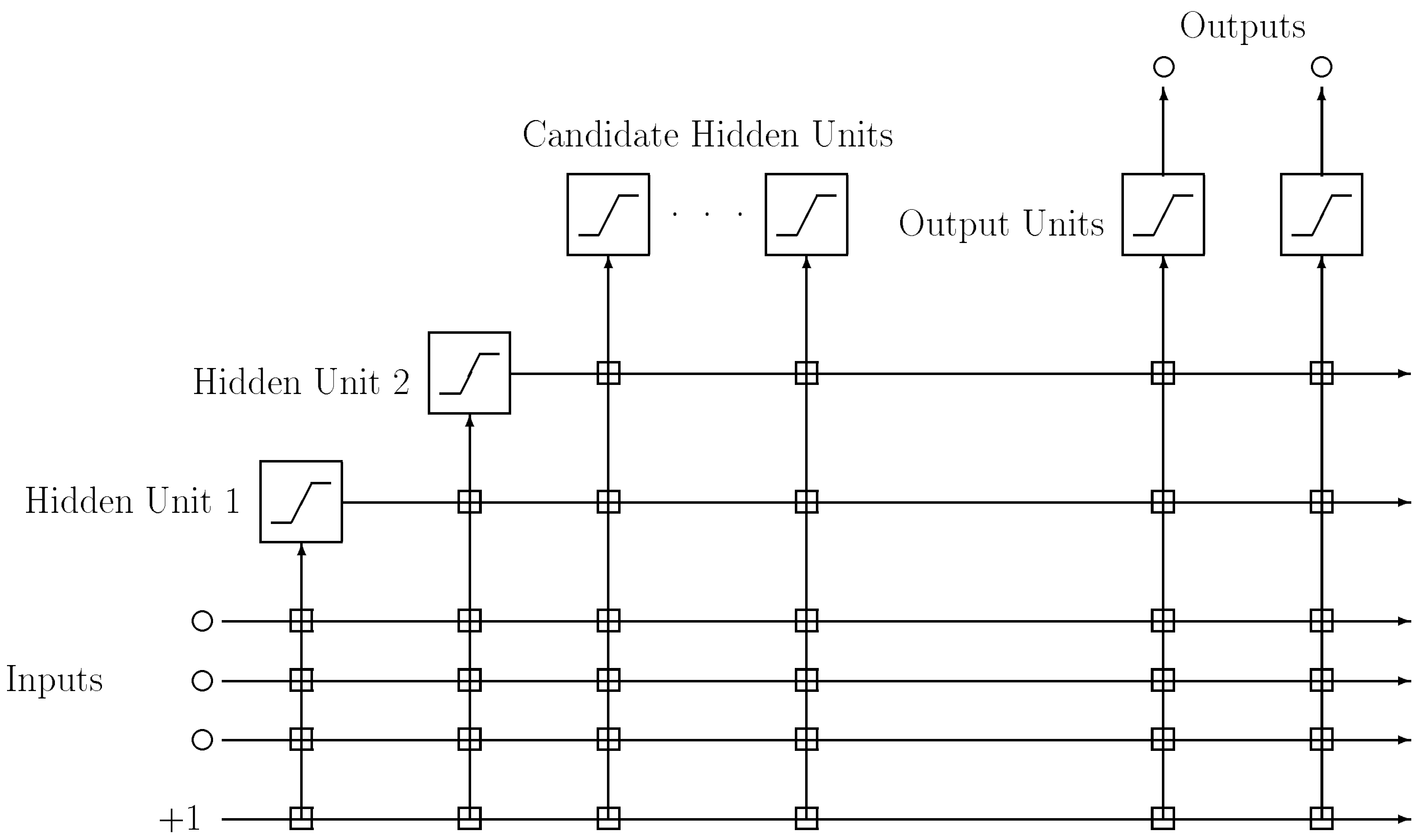
Example: Pruned a 18,000-weight NETtalk network down to 1,560 weights.

Growing Networks

Several algorithms have been developed that employ greedy heuristics to grow networks. The best known of these are

- **Cascade-Correlation**
- **Ramps**

Cascade Correlation (Fahlman)



Training Cascade Correlation

Phase 1: Train hidden-to-output weights.
If error is small enough, quit.

Phase 2: Train a pool of 8 candidate hidden units.
The candidates attempt to correlate their activities with the *errors* from Phase 1.
Select best hidden unit and add to the network.

Cascade Correlation

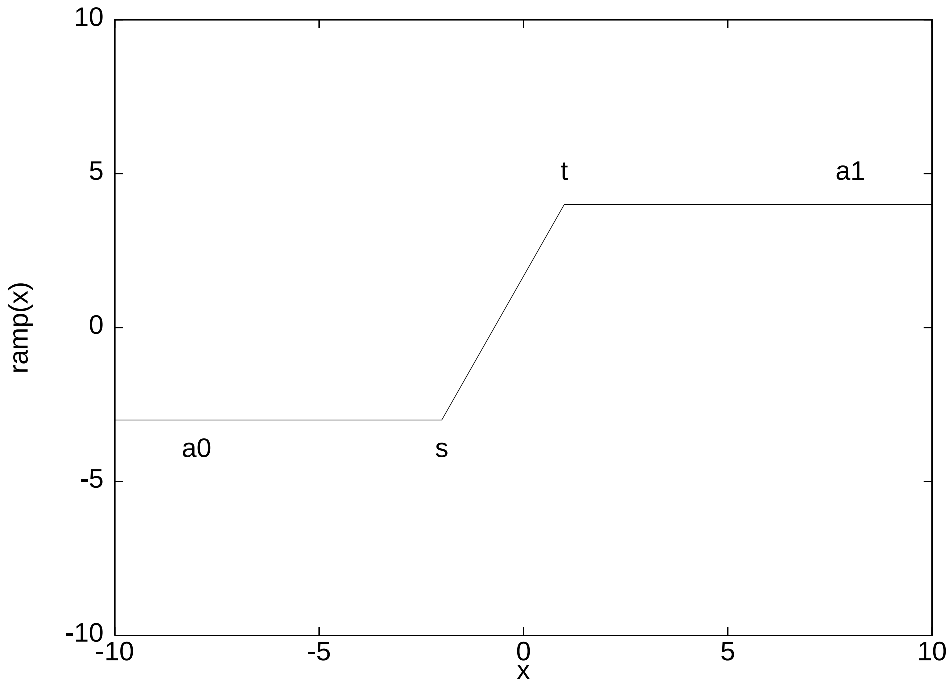
- **Advantages:** Finds small networks.
- **Disadvantages:**

  If the best network is large, it takes a long time to discover this.

  If training set S is large, each phase is very lengthy.

  Many parameters to set.

Ramps (Breiman & Friedman)



Piecewise-linear approximation to sigmoid function.

Very fast algorithm for fitting a ramp to data.

$$\hat{f} = \sum_j w_j \, \mathrm{ramp}_j(x, \Theta_j)$$

One-dimensional Ramp Algorithm

1. Make initial guesses for $a0$, $a1$, s, and t.
2. Compute new values:

   $a0 =$ mean of all values $< s$.

   $a1 =$ mean of all values $> t$.

   compute linear regression model for all (x, y) values $s \leq x \leq t$

   $s =$ value where linear model intersects $a0$

   $t =$ value where linear model intersects $a1$
3. Repeat until convergence.

Converges exponentially to local optimum.

Can be extended to multiple dimensions directly.

Greedy Fitting

Greedy Ramp Fitting

1. Fit ramp to data.
2. Compute new data set as residual errors on data.
3. Repeat until errors are small enough.

Greedy Sigmoid Fitting

1. Fit ramp to data.
2. Convert ramp into sigmoid.
3. Compute new data set as residual errors on data.
4. Repeat until errors are small enough.

Backfitting algorithms can improve quality of fit.

Assessment

Algorithm is still under construction.

Ramps may be "Cascade Correlation Done Right"

Ramps may be "too aggressive" in fitting to data.

More gradual fitting by gradient descent may give better control of over-fitting.

Regularization

Define

$$J_{reg}(S, W) = J(S, W) + C(W)$$

where C is the "cost" of the weight vector W.

- **MDL interpretation**

$$TotalDescLength = DescLength(S|W) + DescLength(W)$$

- **Bayesian interpretation**

$$\log P(W|S) \propto \log P(S|W) + \log P(W).$$

Regularizers

Weight Decay: $\lambda \sum_i w_i^2$

Weight-Elimination: $\lambda \sum_i \frac{w_i^2/w_0^2}{1 + w_i^2/w_0^2}$

w_0 large encourages many small weights.

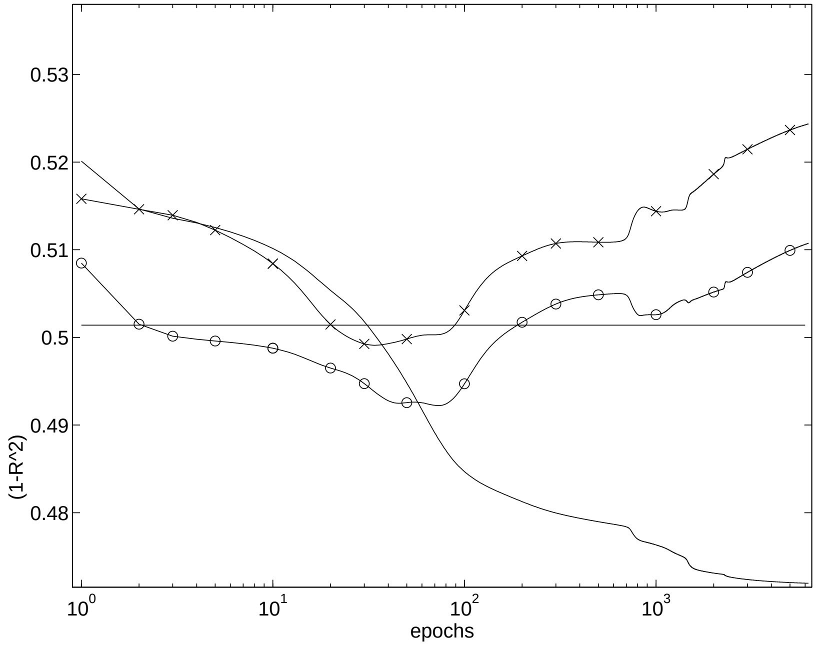
w_0 small encourages a few large weights.

Acts as a counter of large weights.

Strategy: Begin search with no regularizer ($\lambda = 0$).

Gradually increase λ.

Overfitting: When Does It Happen?



Test set performance degrades beyond a certain point.

(Data set: Dow Jones volume, 6000 days)

Early Stopping Heuristic

Cross-validation techniques can be applied to determine when to stop.

1. Divide the training set S into two sets V and R.

2. Apply gradient training with examples from R.
   After each epoch, evaluate performance on V.
   Determine the network that gives peak generalization performance on V.

Best point on validation set used to predict best point on test set.

We can understand this by visualizing the training process.

Estimating Number of Effective Hidden Units

Analyze learning *as function of training time* (not number of examples).

- Compute variances of activations of each hidden unit.
  (They show variation of its response for different patterns).

→ All hidden units get "busy" at the same time in training.

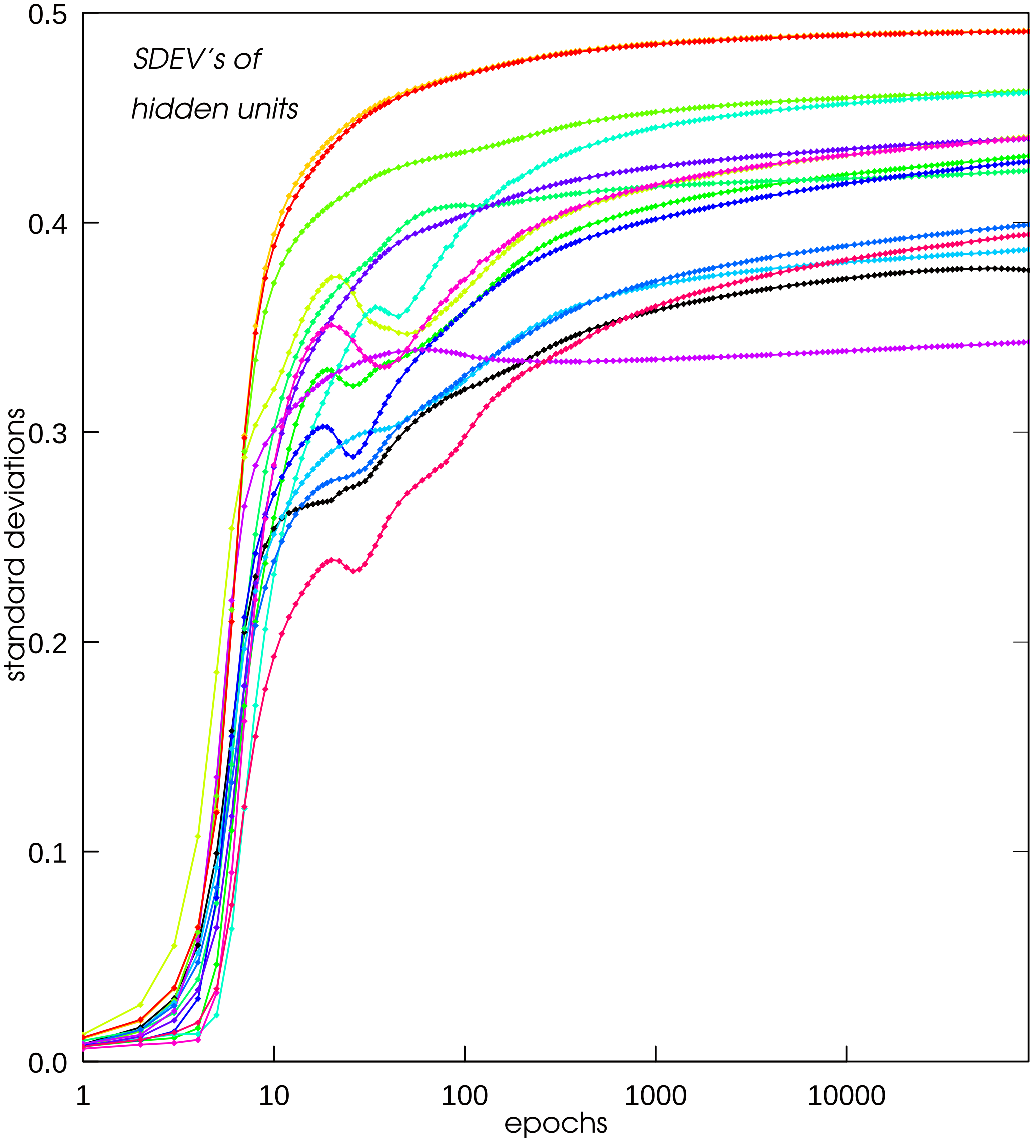
- Compute covariance matrix of hidden unit activations.
- Compute eigenvalue spectrum of the covariance matrix.

→ Features (i.e., principal components) extracted *sequentially.*

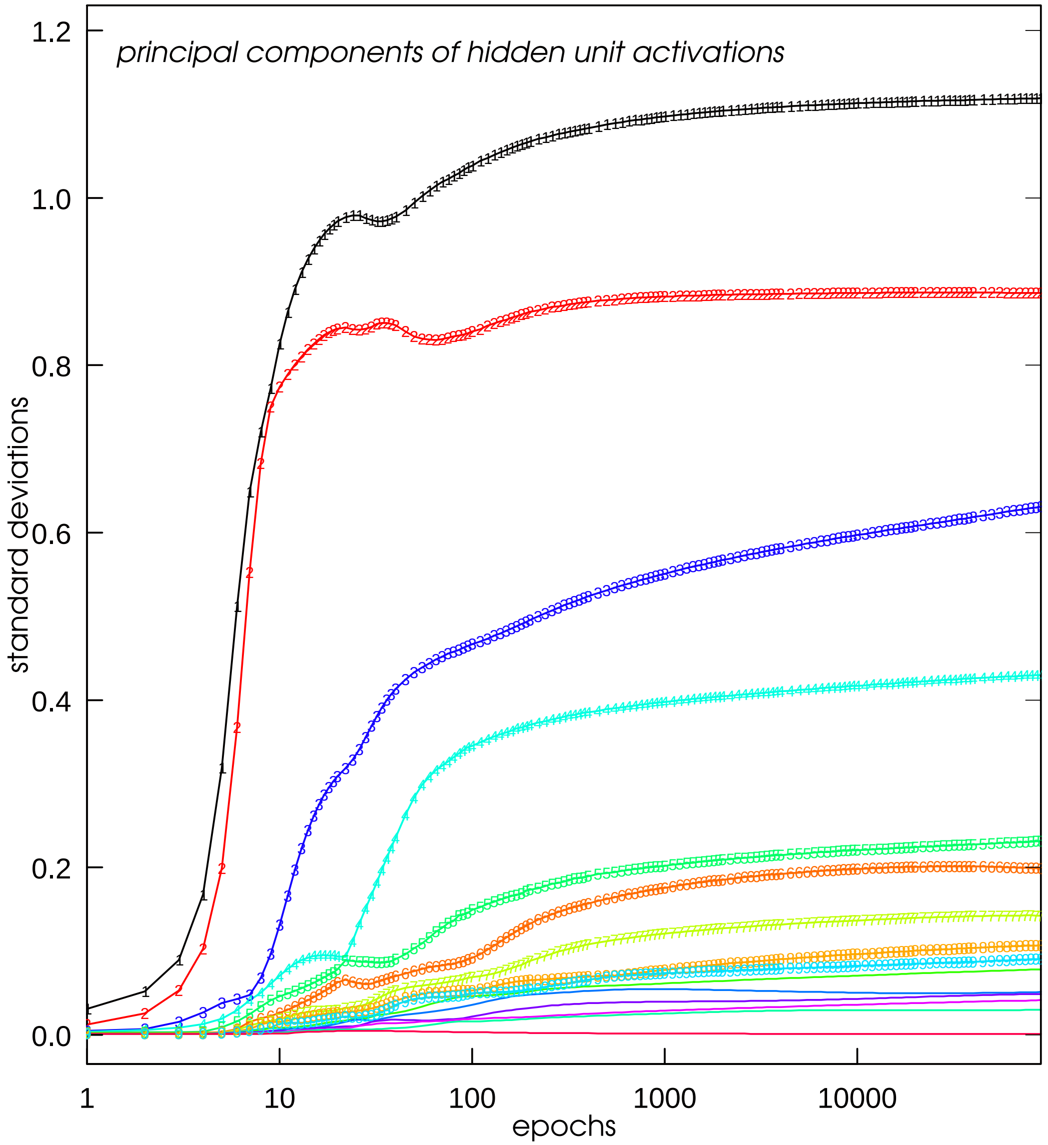
The number of nonzero eigenvalues can serve as a measure of the *effective number of hidden units.*

Example: Phoneme recognition: 15 hidden units, 6 normalized exponentials as outputs.

Hidden Unit Variances

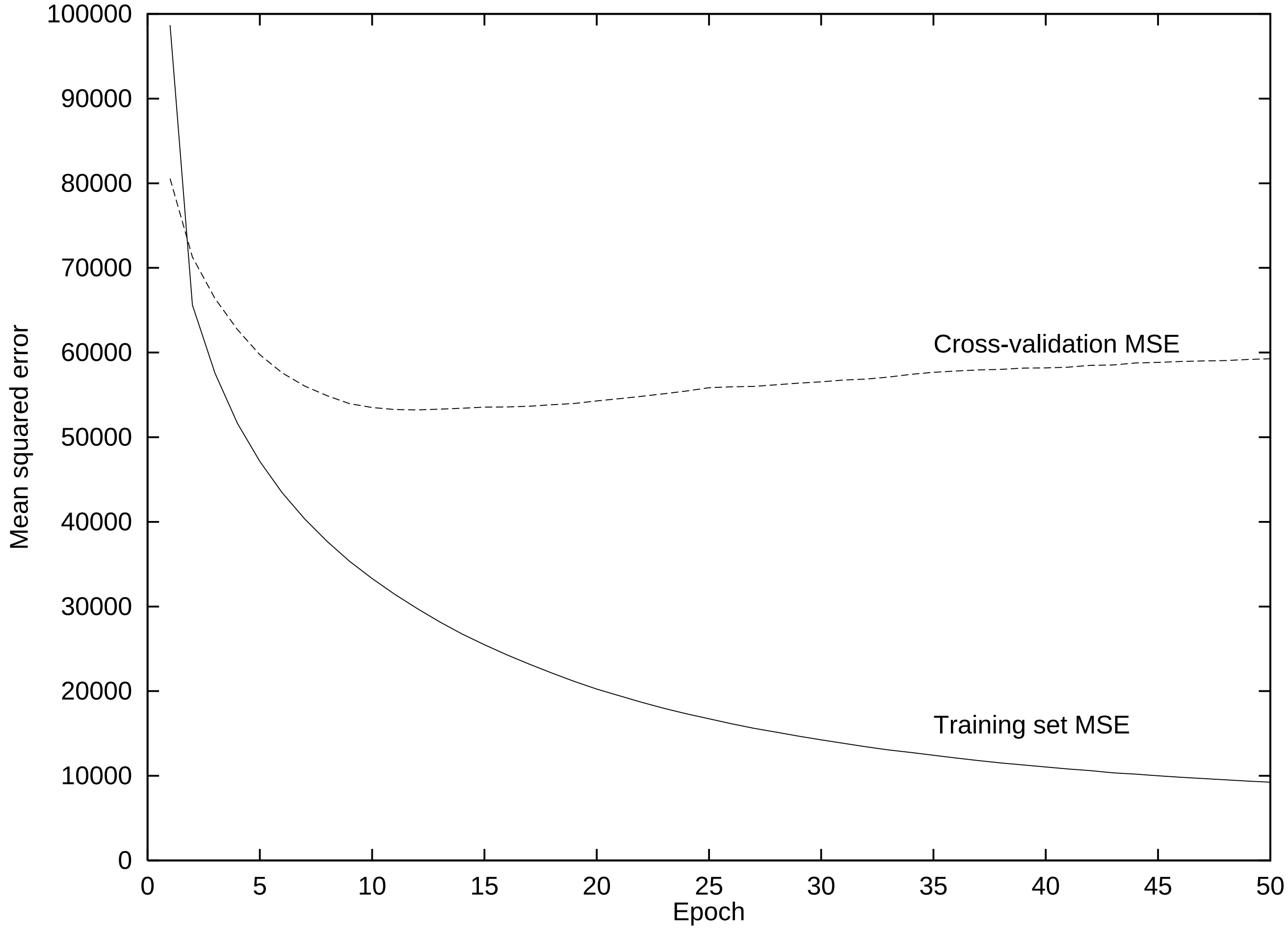


Eigenvalues of Covariance Matrix of Hidden Unit Responses (Principal Components).



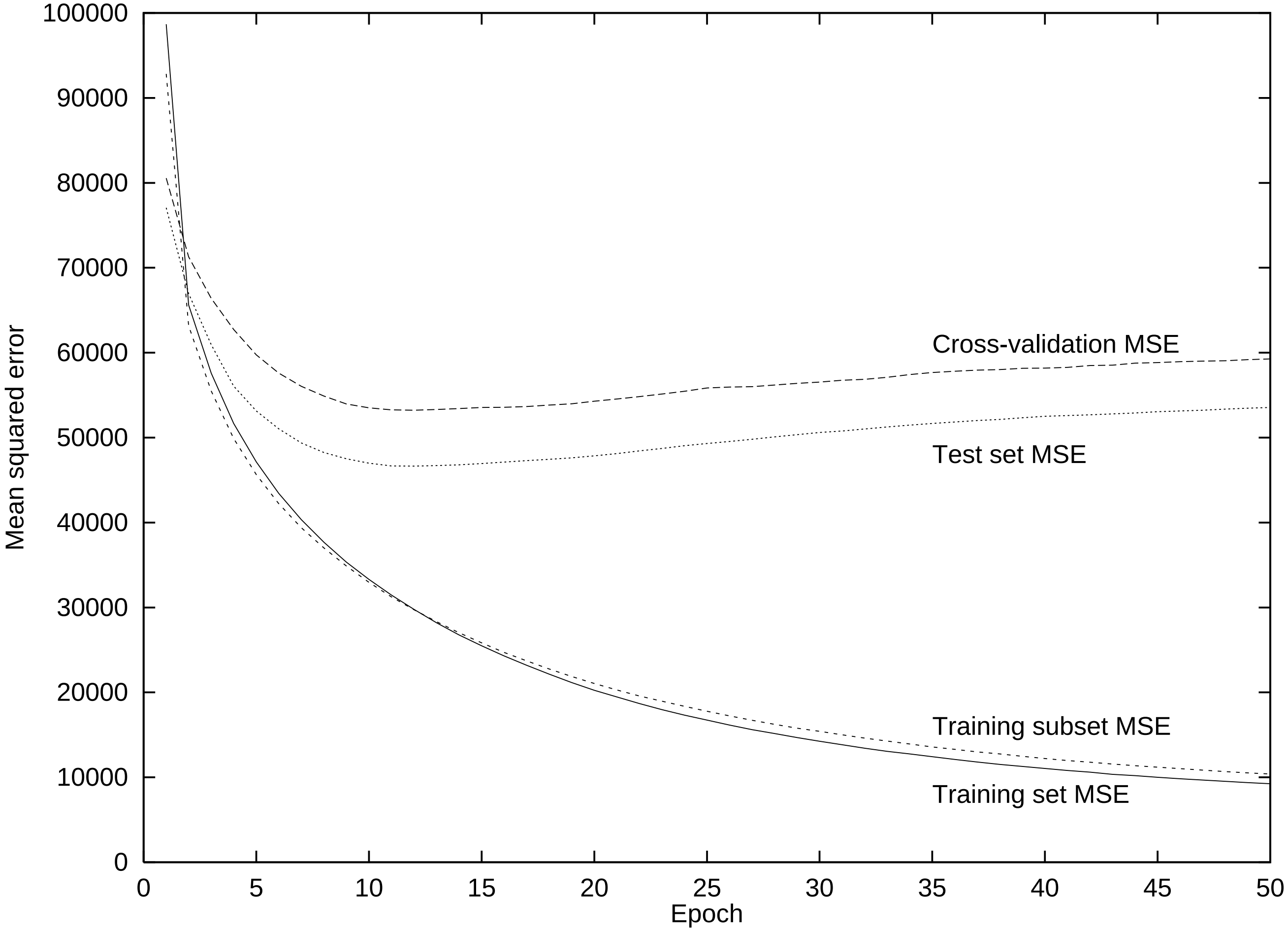
Example: NETtalk Cross-validation Training

Training on R, testing on V



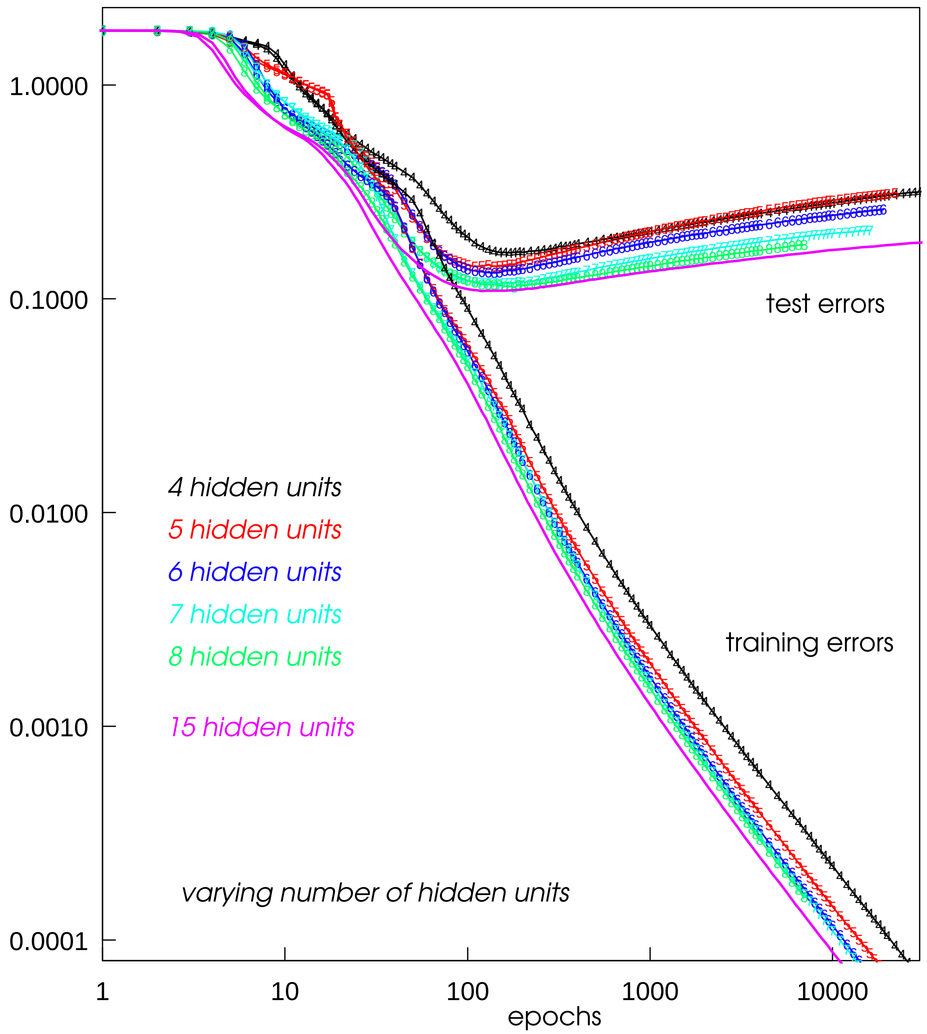
NETtalk Full Training

Training on S, testing on independent data

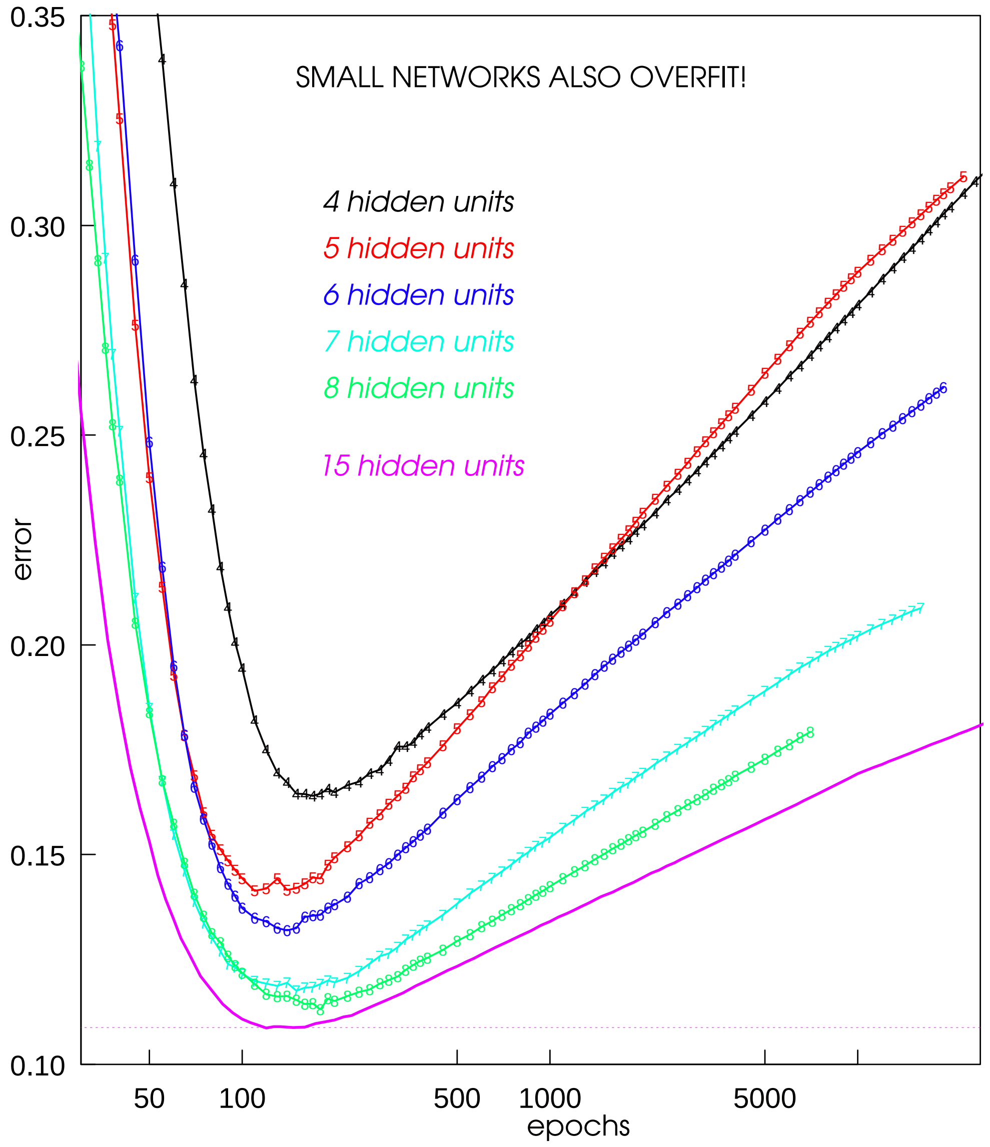


Early Stopping and Large Networks

Performance of early stopping improves with increasing network size.



Expanded View of Test Only

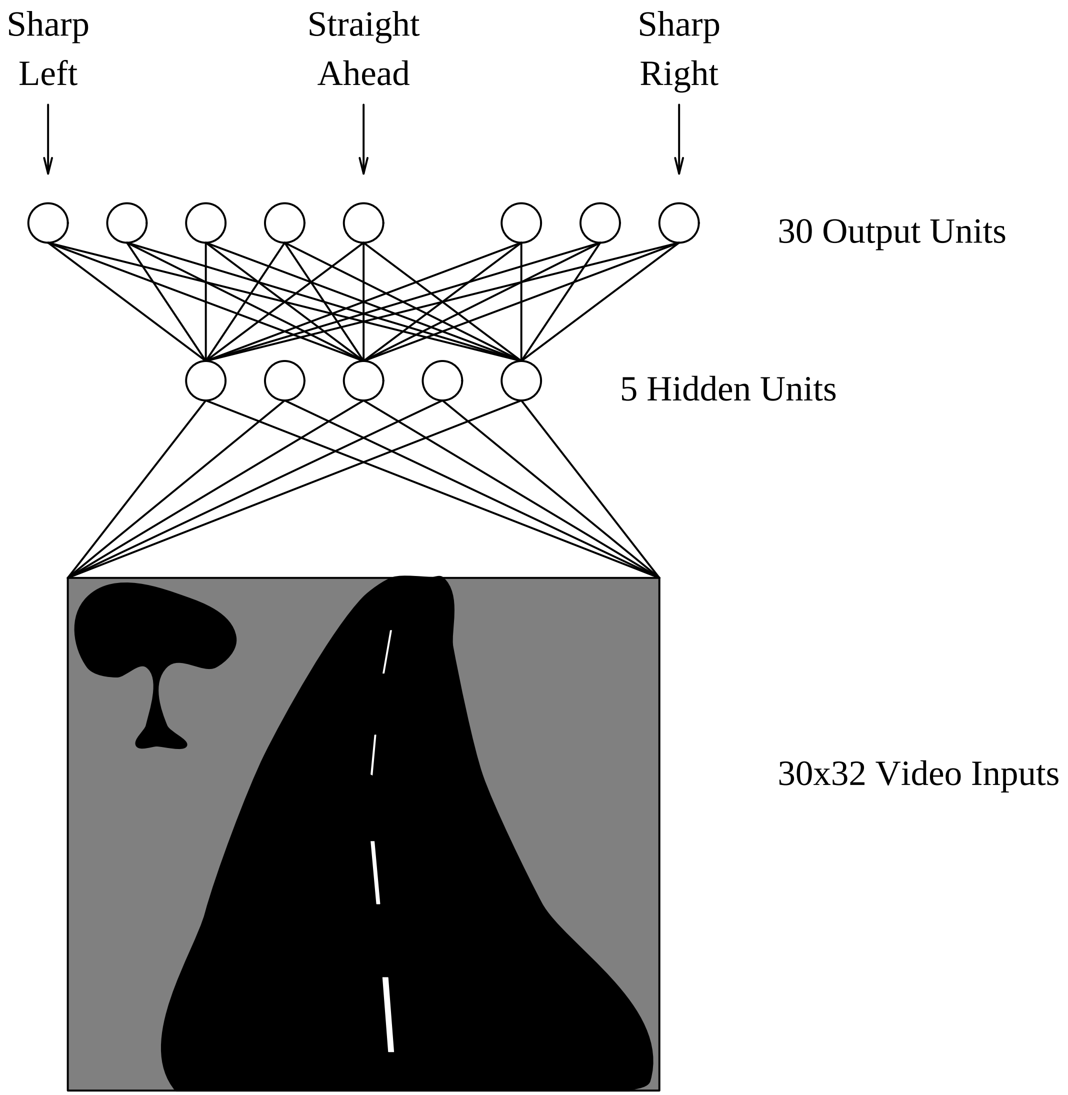


Outline

- Representation
- Learning Algorithms
- **Examples**
  - **ALVINN**
  - **AT&T Optical Character Recognition System**
  - **Sunspots: Soft Weight Sharing**
- Strengths and Weaknesses
- What we left out

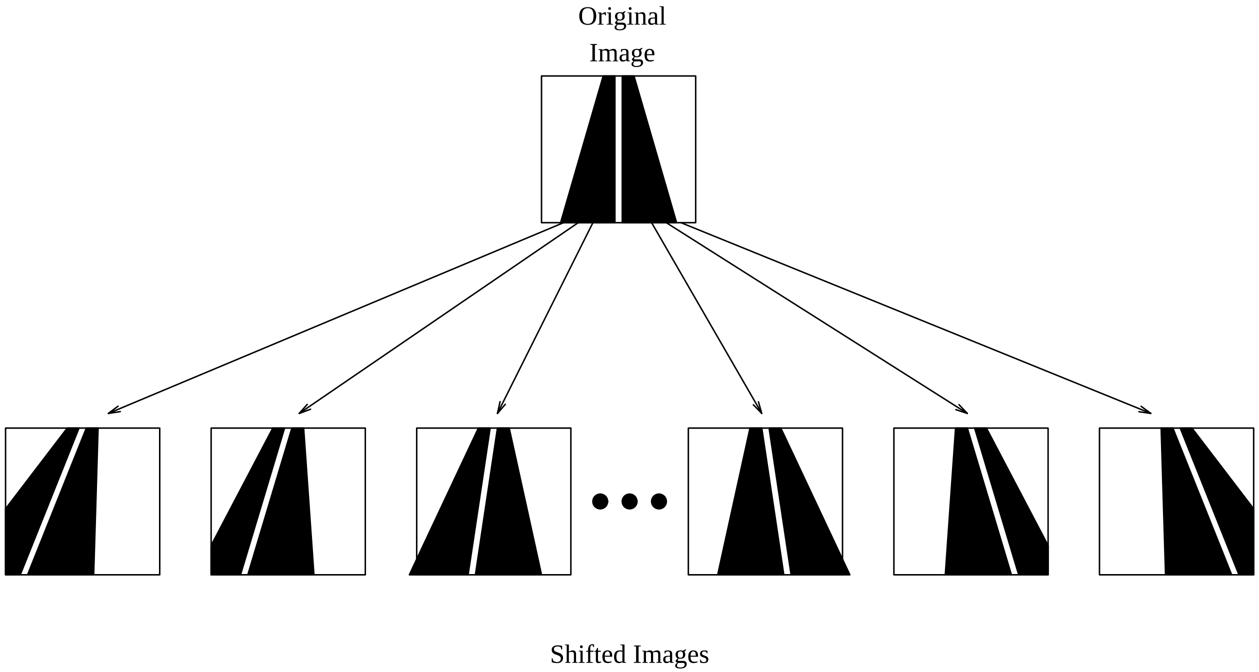
ALVINN (Pomerleau)

Task: Learn to steer the CMU NavLab van



ALVINN (2)

- **Outputs encoded as overlapping range units.**
- **Supervised training from human drivers.**
  Does not provide "off road" experiences.
- **Training from perturbed images.**
  Apply camera model to shift image and driving signal.



AT&T OCR System

Sunspots with Soft Weight Sharing (Nowlan & Hinton)

Bayesian approach to deriving a regularizer.
Assume the weights come from a mixture of Gaussians:

$$w_j \sim \sum_l \pi_l N(\mu_l, \sigma_l).$$

From this, we can compute the probability of w_j given the various parameters π_l, μ_l, and σ_l:

$$P(w_j) = \sum_l \pi_l N(\mu_l - w_j, \sigma_l).$$

Weight-sharing Algorithm

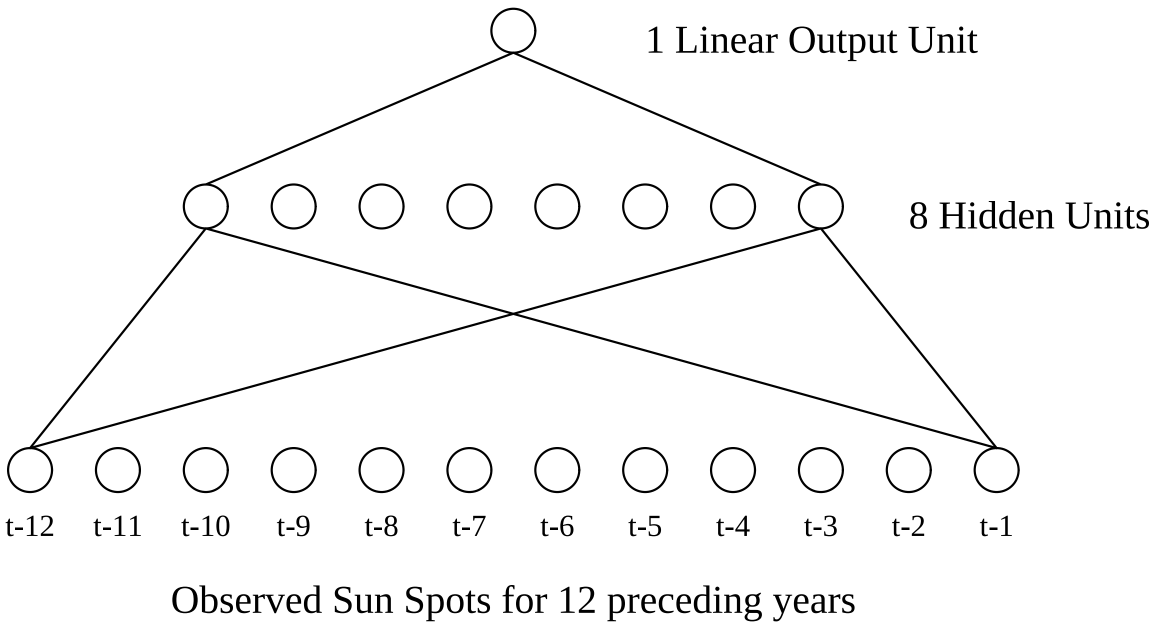
Compute the gradient of J_{reg} and then apply conjugate gradient search.

This is not always stable.
Alternative EM-style algorithm:

1. Train network with no regularizer for awhile.
2. Fit a gaussian mixture model to the weights
   This determines π_l, μ_l, and σ_l.
3. Add regularizer:
   $$-\lambda \sum_l \log P(w_j)$$
4. Train network for awhile.
5. Repeat from step 2 until regularized error is minimum.

Sunspots Time Series

Goal: Predict the number of sunspots observed each year.



Train on data from 1700–1920.

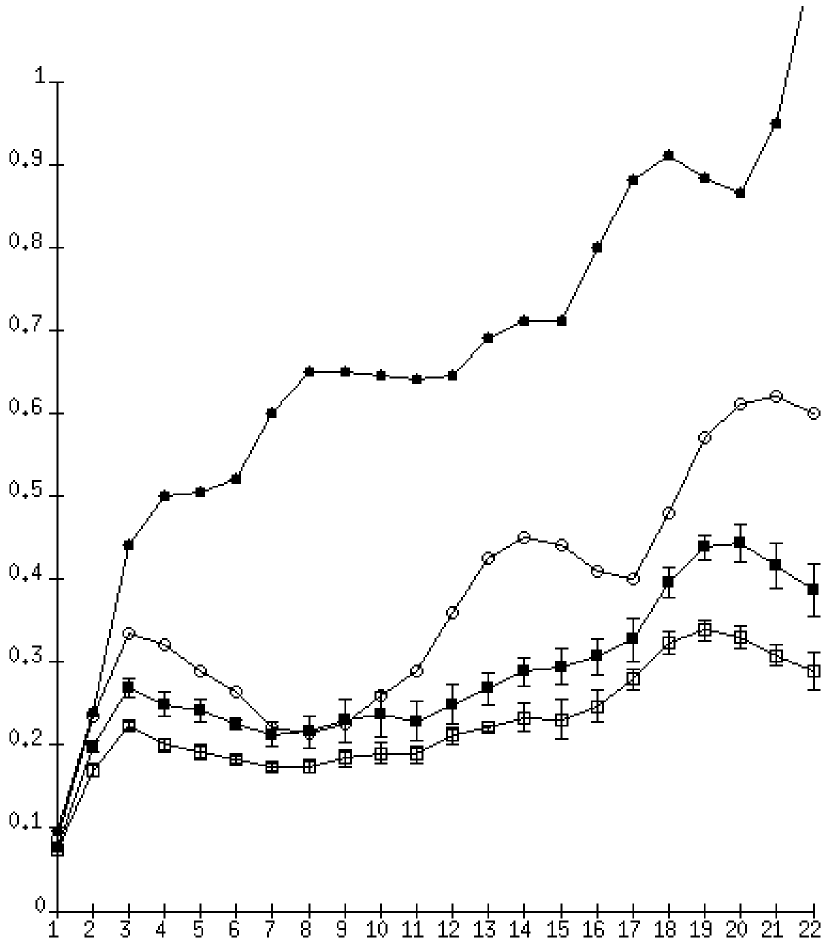
Test on data from 1921–1955.

One-Year Prediction Performance

Method	Test NMSE
Threshold autoregressive	0.097
RBF network	0.092
Weight-elimination	0.086
Soft-share (3 components)	0.077±0.0029
Soft-share (8 components)	0.072±0.0022

$$\text{NMSE} = \frac{\Sigma_k(\text{target}_k - \text{prediction}_k)^2}{\Sigma_k(\text{target}_k - \text{mean}_k)^2}$$

Iterated Performance



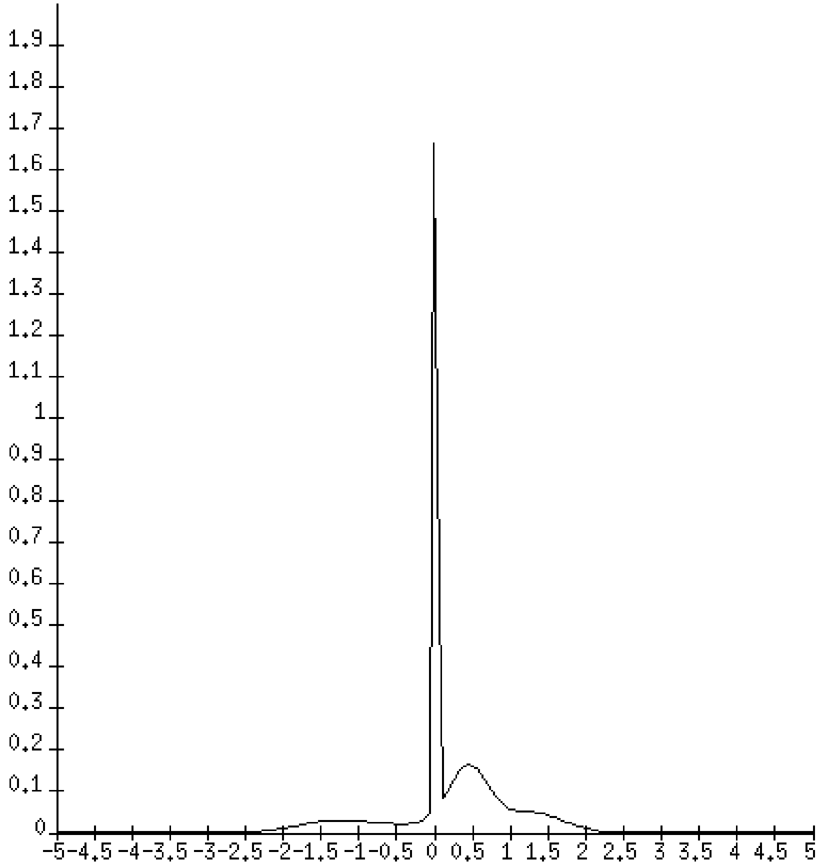
● Threshold auto-regressive (Tong & Lim)

○ Weight elimination network (Weigend, Rumelhart, Huberman)

■ Soft weight-sharing (3 components)

□ Soft weight-sharing (8 components)

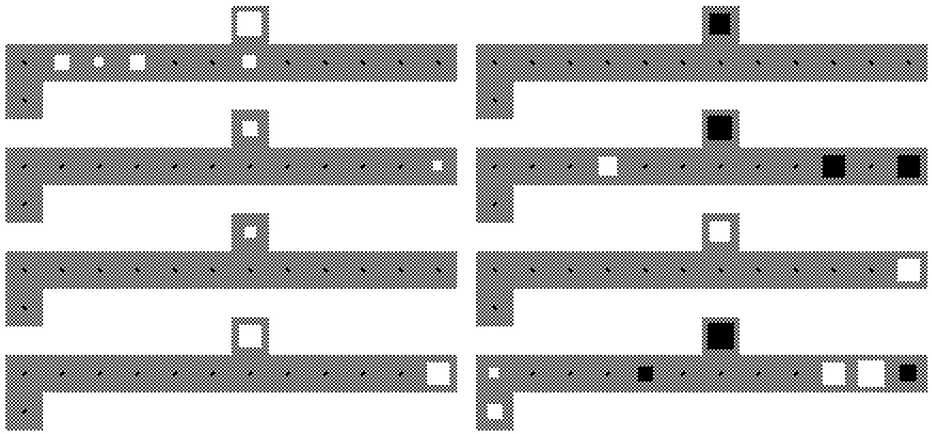
Weight distribution



Major component at 0.

Other components blend into a skewed distribution peaking at 0.5

Weights are Easy to Interpret



Depends on previous year.

Simple moving average.

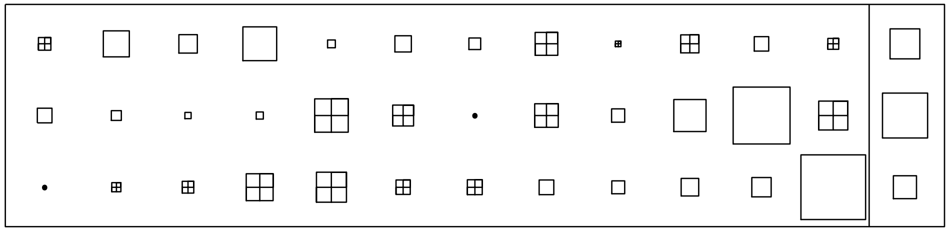
11 year cycle.

A peak has just passed.

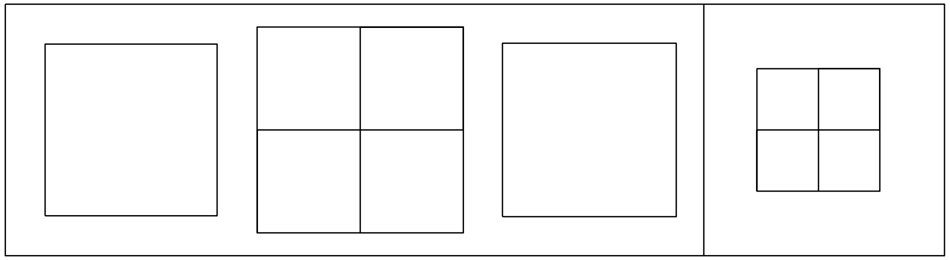
Don't overpredict if peak was 9 years ago and recent activity is low.

Comparison to Weight Elimination

Inputs to hidden units (and biases)



Hidden units to output unit.



Outline

- Representation
- Learning Algorithms
- Examples
- Strengths and Weaknesses
  - **Hidden Units**
  - **Configurable Architectures**
  - **Expressiveness**
  - **Understandability**
  - **Cumulative Learning**
- What we left out

Hidden Units

- **Supervised feature learning:** Hidden units learn useful linear combinations of input features.
  The inputs are projected into hidden-unit space.
  From here, it may be easier to learn the target function $f(\mathrm{x})$.
  No symbolic learning algorithm can discover these kinds of features in such a supervised way.
  This overcomes many problems of learning from highly correlated input features, which traditional statistical methods cannot handle in a supervised fashion.

- **Learning multiple functions:** Hidden unit features are discovered and shared among these functions. Learning to predict multiple, correlated functions is an area that traditional statistics has only recently begun to consider.

- **Learning convolution operators:** Shared weights permit supervised learning of image operators.

Configurable Architectures

The network architecture can be configured to encode domain knowledge.

Image operators encode translation invariance.
Different kinds of inputs can be isolated or combined.

Separately-learned networks can be “glued” together and then jointly optimized.

Expressiveness

Individual units can represent conjunctions, disjunctions, and n-of-m functions.

Units are particularly adept at evidence-gathering applications where many features are all weakly relevant.

Comparison to statistics:

Classical statistics systematically increases the *order* of interactions.

Neural networks control the *number* of interactions irrespective of the order.

Weaknesses

- **Not inspectable**. It is difficult to interpret the knowledge learned in a network (particularly if early stopping is used with a large net).
- **Expensive**. Training can be very expensive. Special-purpose hardware can provide 1000-fold speedups. Such speedups are needed.
- **Non-cumulative learning**. Gradient descent must continue to visit previously-learned examples to prevent "forgetting". It is hard to take an existing network and train it with additional data without losing previously-learned knowledge.
- **Missing data**. Unlike decision-tree methods, missing feature values are difficult and expensive to handle.

Outline

- Representation
- Learning Algorithms
- Examples
- Strengths and Weaknesses
- **What we left out**

What We Left Out

- **Radial-basis function (RBF) networks**

$$\hat{f}(\mathrm{x}) = \sum_l w_l B(\|\mathrm{x} - \mu_l)\|$$

  "Soft nearest-neighbor" methods.

- **Resource Allocating Network (RAN)**
  An algorithm for growing RBF-like networks.

- **Learning Vector Quantizer (LVQ)**
  Learn a set of exemplars in space (by moving them around)
  Classify new examples with nearest-neighbor rule.

- **Self-Organizing Maps**

  Map training examples onto a low-dimensional manifold by moving grid points on the manifold.

- **Unsupervised Learning**

  Competitive learning. IMAX.

- **Recurrent Networks**

  Networks that contain continuous state information and feedback loops.

- **Other Statistical Approaches**

  Adaptive mixtures of experts (Jacobs & Jordan)

  Project-pursuit regression (Friedman & Stuetzle)

Acknowledgements

Thank-you to Steve Nowlan for providing soft weight-sharing figures.

Thank-you to Dean Pomerleau and Todd Leen for providing video tapes of their work.