Today's Discussion

To date:

- Neural networks: what are they
- Backpropagation: efficient gradient computation
- Advanced training: conjugate gradient

Today:

- CG postscript: scaled conjugate gradients
- Adaptive architectures
- My favorite neural network learning environment
- Some applications

Conjugate gradient algorithm

1. Choose an initial weight vector $\mathbf{w}_1$ and let $\mathbf{d}_1 = -\mathbf{g}_1$.
2. Perform a line minimization along $\mathbf{d}_j$, such that:

$$E(\mathbf{w}_j + \alpha^* \mathbf{d}_j) \leq E(\mathbf{w}_j + \alpha \mathbf{d}_j),\ \forall \eta .$$

3. Let $\mathbf{w}_{j+1} = \mathbf{w}_j + \alpha^* \mathbf{d}_j$.
4. Evaluate $\mathbf{g}_{j+1}$.
5. Let $\mathbf{d}_{j+1} = -\mathbf{g}_{j+1} + \beta_j \mathbf{d}_j$ where,

$$\beta_j = \frac{\mathbf{g}_{j+1}^T(\mathbf{g}_{j+1} - \mathbf{g}_j)}{\mathbf{g}_j^T \mathbf{g}_j} \quad \textit{(Polak-Ribiere)}$$

6. Let $j = j + 1$ and go to step 2.

Scaled conjugate gradient algorithm

Basic idea: Replace line minimization:

$$E(\mathbf{w}_j + \alpha^* \mathbf{d}_j) \leq E(\mathbf{w}_j + \alpha \mathbf{d}_j),\ \forall \eta .$$

with:

$$\alpha_j = \frac{-\mathbf{d}_j^T \mathbf{g}_j}{\mathbf{d}_j^T \mathbf{H} \mathbf{d}_j}$$

Why #!@$ are we doing this? Didn't we want to avoid computation of $\mathbf{H}$?

Scaled conjugate gradient algorithm

Well, yes but:

- Line minimization can be computationally expensive.
- Don't really have to compute $\mathbf{H}$? Huh?

$$\alpha_j = \frac{-\mathbf{d}_j^T \mathbf{g}_j}{\mathbf{d}_j^T \mathbf{H} \mathbf{d}_j}$$

A closer look at α_j

Don't have to compute $\mathbf{H}$, only $\mathbf{Hd}_j$.

Theorem:

$\mathbf{w}_0$ = current W-dimensional weight vector,

$\mathbf{g}(\mathbf{w}) = \nabla E(\mathbf{w})$ (gradient of E at some vector $\mathbf{w}$), and,

$\mathbf{H}$ = Hessian of E evaluated at $\mathbf{w}_0$,

$\mathbf{d}$ = arbitrary W-dimensional vector.

$$\mathbf{Hd} = \lim_{\varepsilon \to 0} \frac{\mathbf{g}(\mathbf{w}_0 + \varepsilon\mathbf{d}) - \mathbf{g}(\mathbf{w}_0 - \varepsilon\mathbf{d})}{2\varepsilon}$$

Computing $\mathbf{Hd}_j$

$$\mathbf{Hd} = \lim_{\varepsilon \to 0} \frac{\mathbf{g}(\mathbf{w}_0 + \varepsilon\mathbf{d}) - \mathbf{g}(\mathbf{w}_0 - \varepsilon\mathbf{d})}{2\varepsilon}$$

First-order Taylor expansion of $\mathbf{g}(\mathbf{w})$ about $\mathbf{w}_0$:

$$\mathbf{g}(\mathbf{w}) \approx \mathbf{g}(\mathbf{w}_0) + \mathbf{H}(\mathbf{w} - \mathbf{w}_0)$$

$$\frac{\mathbf{g}(\mathbf{w}_0 + \varepsilon\mathbf{d}) - \mathbf{g}(\mathbf{w}_0 - \varepsilon\mathbf{d})}{2\varepsilon} \approx \frac{[\mathbf{g}(\mathbf{w}_0) + \mathbf{H}(\varepsilon\mathbf{d})] - [\mathbf{g}(\mathbf{w}_0) - \mathbf{H}(\varepsilon\mathbf{d})]}{2\varepsilon}$$

Computing $\mathbf{Hd}_j$

$$\frac{\mathbf{g}(\mathbf{w}_0 + \varepsilon\mathbf{d}) - \mathbf{g}(\mathbf{w}_0 - \varepsilon\mathbf{d})}{2\varepsilon} \approx \frac{2\varepsilon\mathbf{Hd}}{2\varepsilon}$$

$$\frac{\mathbf{g}(\mathbf{w}_0 + \varepsilon\mathbf{d}) - \mathbf{g}(\mathbf{w}_0 - \varepsilon\mathbf{d})}{2\varepsilon} \approx \mathbf{Hd}$$

So:

$$\mathbf{Hd} = \lim_{\varepsilon \to 0} \frac{\mathbf{g}(\mathbf{w}_0 + \varepsilon\mathbf{d}) - \mathbf{g}(\mathbf{w}_0 - \varepsilon\mathbf{d})}{2\varepsilon}$$

$\alpha_j = \dfrac{-\mathbf{d}_j^T\mathbf{g}_j}{\mathbf{d}_j^T\mathbf{Hd}_j}$ **now just requires two gradient evaluations...**

New conjugate gradient algorithm

1. Choose an initial weight vector $\mathbf{w}_1$ and let $\mathbf{d}_1 = -\mathbf{g}_1$.
2. Compute α_j:

   $\alpha_j = -\mathbf{d}_j^T\mathbf{g}_j / \mathbf{d}_j^T\mathbf{Hd}_j$, $\forall\eta$.
3. Let $\mathbf{w}_{j+1} = \mathbf{w}_j + \alpha_j\mathbf{d}_j$.
4. Evaluate $\mathbf{g}_{j+1}$.
5. Let $\mathbf{d}_{j+1} = -\mathbf{g}_{j+1} + \beta_j\mathbf{d}_j$ where,

   $\beta_j = \mathbf{g}_{j+1}^T(\mathbf{g}_{j+1} - \mathbf{g}_j) / \mathbf{g}_j^T\mathbf{g}_j$
6. Let $j = j + 1$ and go to step 2.

Any problems?

What about $\mathbf{H} < 0$?

$\alpha_j = -\mathbf{d}_j^T\mathbf{g}_j / \mathbf{d}_j^T\mathbf{H}\mathbf{d}_j$ **might take uphill steps...**

Idea:

- Replace $\mathbf{H}$ with $\mathbf{H} + \lambda\mathbf{I}$
- So:

$$\alpha_j = \frac{-\mathbf{d}_j^T\mathbf{g}_j}{\mathbf{d}_j^T\mathbf{H}\mathbf{d}_j + \lambda\|\mathbf{d}_j\|^2}$$

What the #$@! is this?

Examining λ

$$\alpha_j = \frac{-\mathbf{d}_j^T\mathbf{g}_j}{\mathbf{d}_j^T\mathbf{H}\mathbf{d}_j + \lambda\|\mathbf{d}_j\|^2}$$

- **What is the meaning of λ being very large?**
- **What is the meaning of λ being very small (i.e. zero)?**

Model trust regions

Question: When should we "trust"

$$\alpha_j = \frac{-\mathbf{d}_j^T\mathbf{g}_j}{\mathbf{d}_j^T\mathbf{H}\mathbf{d}_j} ?$$

Model trust regions

Question: When should we "trust"

$$\alpha_j = \frac{-\mathbf{d}_j^T\mathbf{g}_j}{\mathbf{d}_j^T\mathbf{H}\mathbf{d}_j} ?$$

1. H is positive definite (denominator > 0)

2. Local quadratic assumption is good

Near a mountain, not a valley

Look at denominator of:

$$\alpha_j = \frac{-\mathbf{d}_j^T \mathbf{g}_j}{\mathbf{d}_j^T \mathbf{H} \mathbf{d}_j + \lambda \|\mathbf{d}_j\|^2}$$

$$\delta = \mathbf{d}_j^T \mathbf{H} \mathbf{d}_j + \lambda \|\mathbf{d}_j\|^2$$

If $\delta < 0$, *increase* λ to make denominator positive.

How to increase λ ?

How about:

$$\lambda' = 2\left(\lambda - \frac{\delta}{\|\mathbf{d}_j\|^2}\right)$$

so that:

$$\delta' = \delta + (\lambda' - \lambda)\|\mathbf{d}_j\|^2$$

$$\delta' = \delta + \left[2\left(\lambda - \frac{\delta}{\|\mathbf{d}_j\|^2}\right) - \lambda\right]\|\mathbf{d}_j\|^2$$

$$\delta' = \delta - 2\delta + \lambda\|\mathbf{d}_j\|^2 = -\delta + \lambda\|\mathbf{d}_j\|^2$$

New effective denominator value

$$\lambda' = 2\left(\lambda - \frac{\delta}{\|\mathbf{d}_j\|^2}\right)$$

$$\delta' = -\delta + \lambda\|\mathbf{d}_j\|^2$$

So:

$$\delta' = -(\mathbf{d}_j^T \mathbf{H} \mathbf{d}_j + \lambda\|\mathbf{d}_j\|^2) + \lambda\|\mathbf{d}_j\|^2$$

$$\delta' = -\mathbf{d}_j^T \mathbf{H} \mathbf{d}_j$$ *(what does this mean?)*

Goin' up? I'll show you...

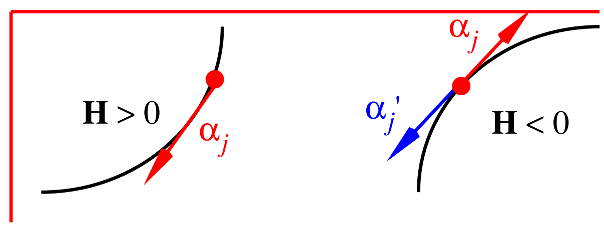
Since the new denominator is:

$$\delta' = -\mathbf{d}_j^T \mathbf{H} \mathbf{d}_j$$

the new value of α_j is:

$$\alpha_j' = \frac{-\mathbf{d}_j^T \mathbf{g}_j}{-\mathbf{d}_j^T \mathbf{H} \mathbf{d}_j} = \frac{\mathbf{d}_j^T \mathbf{g}_j}{\mathbf{d}_j^T \mathbf{H} \mathbf{d}_j}$$

$$\alpha_j' = -\alpha_j$$



Model trust regions

Question: When should we "trust"

$$\alpha_j = \frac{-\mathbf{d}_j^T\mathbf{g}_j}{\mathbf{d}_j^T\mathbf{H}\mathbf{d}_j} \ ?$$

1. H is positive definite (denominator > 0)

2. Local quadratic assumption is good

How to test local quadratic assumption?

Check:

$$\Delta = \frac{E(\mathbf{w}_j) - E(\mathbf{w}_j + \alpha_j\mathbf{d}_j)}{E(\mathbf{w}_j) - E_Q(\mathbf{w}_j + \alpha_j\mathbf{d}_j)}$$

What's E_Q?

$$E_Q(\mathbf{w}) = E(\mathbf{w}_0) + (\mathbf{w} - \mathbf{w}_0)^T\mathbf{b} + \frac{1}{2}(\mathbf{w} - \mathbf{w}_0)^T H(\mathbf{w} - \mathbf{w}_0)$$

So:

$$E_Q(\mathbf{w}_j + \alpha_j\mathbf{d}_j) = E(\mathbf{w}_j) + \alpha_j\mathbf{d}_j^T\mathbf{g}_j + \frac{1}{2}\alpha_j^2\mathbf{d}_j^T\mathbf{H}\mathbf{d}_j$$

What does Δ tell us?

Local quadratic test

$$\Delta = \frac{E(\mathbf{w}_j) - E(\mathbf{w}_j + \alpha_j\mathbf{d}_j)}{E(\mathbf{w}_j) - E_Q(\mathbf{w}_j + \alpha_j\mathbf{d}_j)}$$

Adjustment of trust region:

- If $\Delta > 0.75$ then decrease λ (e.g. $\lambda = \lambda/2$)
- If $\Delta < 0.25$ then increase λ (e.g. $\lambda = 4\lambda$)
- Otherwise, leave λ unchanged

Scaled conjugate gradient algorithm (α_j, λ)

1. Compute $\delta = \mathbf{d}_j^T\mathbf{H}\mathbf{d}_j + \lambda\|\mathbf{d}_j\|^2$.

2. If $\delta < 0$, **set** $\lambda = 2(\lambda - \delta/\|\mathbf{d}_j\|^2)$.

3. Compute $\alpha_j = -\mathbf{d}_j^T\mathbf{g}_j/(\mathbf{d}_j^T\mathbf{H}\mathbf{d}_j + \lambda\|\mathbf{d}_j\|^2)$.

4. Compute Δ:

5. $\Delta = \dfrac{E(\mathbf{w}_j) - E(\mathbf{w}_j + \alpha_j\mathbf{d}_j)}{E(\mathbf{w}_j) - E_Q(\mathbf{w}_j + \alpha_j\mathbf{d}_j)}$

6. If $\Delta > 0.75$, **set** $\lambda = \lambda/2$, **else if** $\Delta < 0.25$, **set** $\lambda = 4\lambda$.

Scaled conjugate gradient algorithm

1. Choose an initial weight vector $\mathbf{w}_1$ and let $\mathbf{d}_1 = -\mathbf{g}_1$.
2. Compute α_j, λ:

$$\alpha_j = \frac{-\mathbf{d}_j^T \mathbf{g}_j}{\mathbf{d}_j^T \mathbf{H} \mathbf{d}_j + \lambda \|\mathbf{d}_j\|^2}, \ \forall \eta .$$

3. Let $\mathbf{w}_{j+1} = \mathbf{w}_j + \alpha_j \mathbf{d}_j$.
4. Evaluate $\mathbf{g}_{j+1}$.
5. Let $\mathbf{d}_{j+1} = -\mathbf{g}_{j+1} + \beta_j \mathbf{d}_j$ where,

$$\beta_j = \mathbf{g}_{j+1}^T (\mathbf{g}_{j+1} - \mathbf{g}_j) / \mathbf{g}_j^T \mathbf{g}_j$$

6. Let $j = j + 1$ and go to step 2.

Today's Discussion

To date:

- Neural networks: what are they
- Backpropagation: efficient gradient computation
- Advanced training: conjugate gradient

Today:

- CG postscript: scaled conjugate gradients
- Adaptive architectures
- My favorite neural network learning environment
- Some applications

Adaptive architectures

Standard learning:

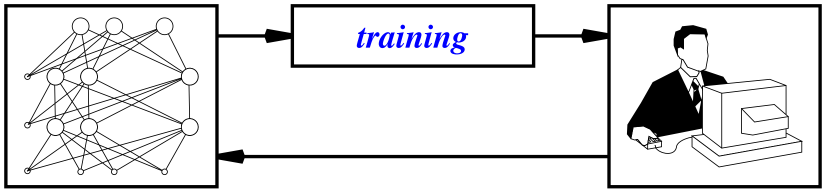
- Select neural network architecture
- Train neural network
- If failure, go back to first step

Better approach:

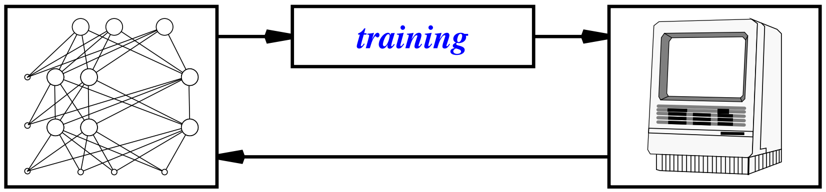
- Adapt neural network architecture as function of training

Adaptive architectures

Standard learning:



Adaptive approach:



Adaptive architectures

Problem: How do we do this?

Two main approaches:

- Pruning (destructive algorithms)
- Growing (constructive algorithms)

Pruning algorithms

Basic idea:

- Start with really “big” network
- Eliminate “unimportant” weights/nodes
- Retrain neural network

Advantages?

Disadvantages?

Problems?

Pruning algorithms

Basic idea:

- Start with really “big” network
- Eliminate “unimportant” weights/nodes
- Retrain neural network

Advantages? ***(smaller final architectures)***

Disadvantages? ***(training cost of large network, retraining)***

Problems? ***(what is “unimportant?”)***

Weight elimination schemes

Idea: eliminate weights based on “saliency.”

Definition: *saliency* S_i = relative importance of weight ω_i

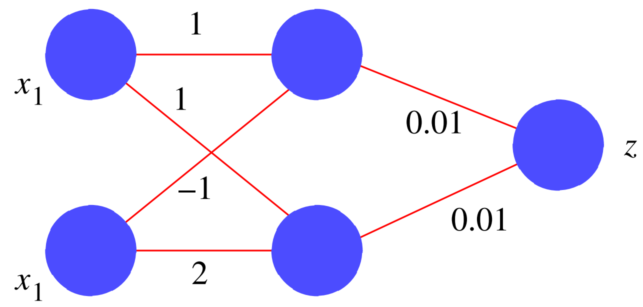
Any suggestions?

Saliency

First guess: $S_i = \|\omega_i\|$

Will this work?

Why won't this measure of saliency work



A better idea for saliency

Try to find relationship:

$$\delta E = \delta \mathbf{w}$$

How can we do this?

- Brute force:

$$\delta E_i = \|E(\mathbf{w}) - E(\mathbf{w} + \delta \mathbf{w}_i)\|$$

$$\delta \mathbf{w}_i = [0, \ldots, 0, -\omega_i, 0, \ldots, 0]$$ *(problems?)*

More on saliency

Use ol' reliable: 2nd order Taylor approximation

$$E(\mathbf{w}) = E(\mathbf{w}_0) + (\mathbf{w} - \mathbf{w}_0)^T \nabla E(\mathbf{w}_0) + \frac{1}{2}(\mathbf{w} - \mathbf{w}_0)^T H (\mathbf{w} - \mathbf{w}_0)$$

Now:

$$\delta \mathbf{w} = \mathbf{w}_1 - \mathbf{w}_0$$ *(what are $\mathbf{w}_1$ and $\mathbf{w}_0$?)*

$$\delta E = E(\mathbf{w}_1) - E(\mathbf{w}_0) = \delta \mathbf{w}^T \nabla E(\mathbf{w}_0) + \frac{1}{2} \delta \mathbf{w}^T \mathbf{H} \delta \mathbf{w}$$

Can we simply this?

Optimal Brain Damage

$$\delta E = \frac{1}{2}\delta\mathbf{w}^T\mathbf{H}\delta\mathbf{w}$$

1. Idea: assume Hessian is diagonal

$$\delta E = \frac{1}{2}\sum_i H_{ii}\delta\omega_i^2$$

2. Resulting saliency:

$$S_i = \frac{H_{ii}\omega_i^2}{2}$$

3. Eliminate weights with smallest saliency

4. Retrain remaining weights

Optimal Brain Surgery

- Smarter idea: don't assume Hessian is diagonal
- Eliminate need for retraining

Now, assume you want to remove weight ω_i:

We want to minimize,

$$\delta E = \frac{1}{2}\delta\mathbf{w}^T\mathbf{H}\delta\mathbf{w}$$

subject to constraint

$\delta\omega_i = -\omega_i$ *(why?)*

Optimal Brain Surgery

Use Lagrange multipliers:

- We can minimize $f(\mathbf{x})$ subject to constraint $g(\mathbf{x}) = 0$ by minimizing $L = f(\mathbf{x}) + \lambda g(\mathbf{x})$
- λ = Lagrange multiplier

For our case:

$$f(\mathbf{x}) = \delta E = \frac{1}{2}\delta\mathbf{w}^T\mathbf{H}\delta\mathbf{w}$$

$$g(\mathbf{x}) = \delta\omega_i + \omega_i$$

Optimal Brain Surgery

Minimize:

$$L = \frac{1}{2}\delta\mathbf{w}^T\mathbf{H}\delta\mathbf{w} + \lambda(\delta\omega_i + \omega_i)$$

...

Solution:

$$\delta\mathbf{w} = -\frac{\omega_i}{[\mathbf{H}^{-1}]_{ii}}\mathbf{H}^{-1}\mathbf{u}_i$$

$$\delta E_i = \frac{1}{2}\frac{\omega_i^2}{[\mathbf{H}^{-1}]_{ii}}$$ *(what's the problem?)*

Optimal Brain Surgery

1. Evaluate the inverse Hessian $\mathbf{H}^{-1}$.
2. Evaluate:

$$\delta E_i = \frac{1}{2}\frac{\omega_i^2}{[\mathbf{H}^{-1}]_{ii}}$$

3. Eliminate weight ω_i, $\delta E_i < \delta E_j$, $i \neq j$.
4. Update all weights (no retraining)

$$\delta \mathbf{w} = -\frac{\omega_i}{[\mathbf{H}^{-1}]_{ii}}\mathbf{H}^{-1}\mathbf{u}_i$$

Node elimination scheme

Idea: Node pruning — need saliency of node, not weight

Define:

$$z_j = \gamma\left(\alpha_j \sum_i \omega_{ij} z_i\right)$$ *(output of unit j with addition of α_j)*

Then:

$$s_j = E(\alpha_j = 1) - E(\alpha_j = 0)$$

$$s_j \approx \partial E / \partial \alpha_j \big|_{j=1}$$

Pruning algorithms: key issues

- Large network to small network
- Need definition of saliency
- May need retraining step

Big problem: *lots of wasted training effort*

Growing algorithms

Basic idea:

- Start with really small network
- Add hidden units as required

Advantages?

Disadvantages?

Problems?

Growing algorithms

Basic idea:

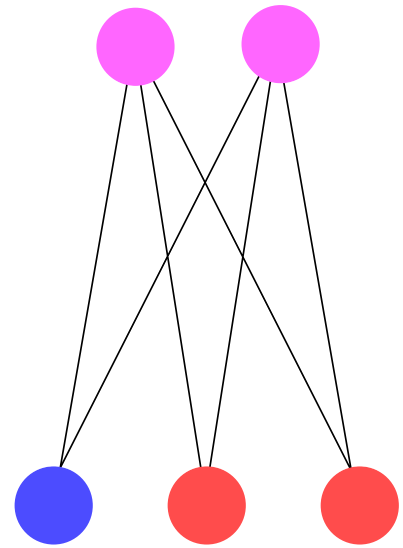
- Start with really small network
- Add hidden units as required

Advantages? ***(reduced training cost, optimized networks)***

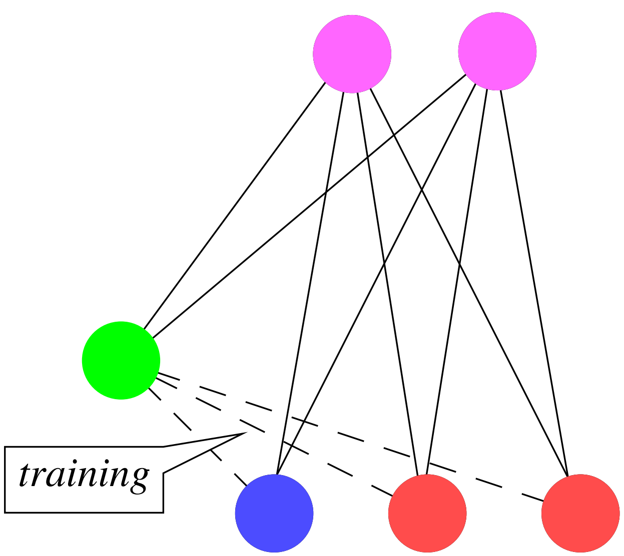
Disadvantages? ***(?)***

Problems? ***(arrangement of added weights/nodes)***

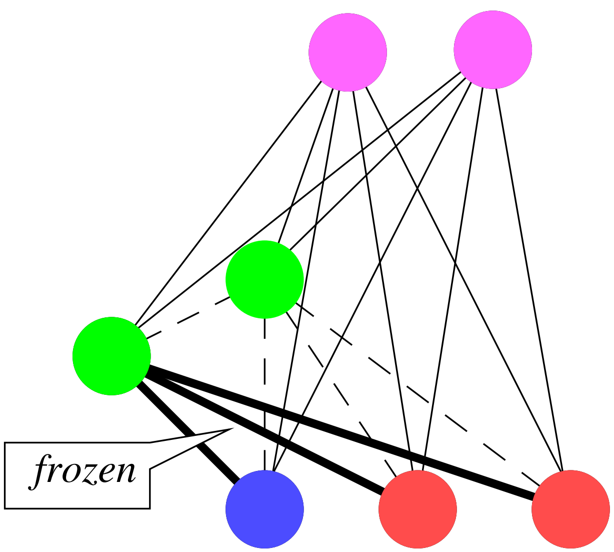
Cascade growing: initial network



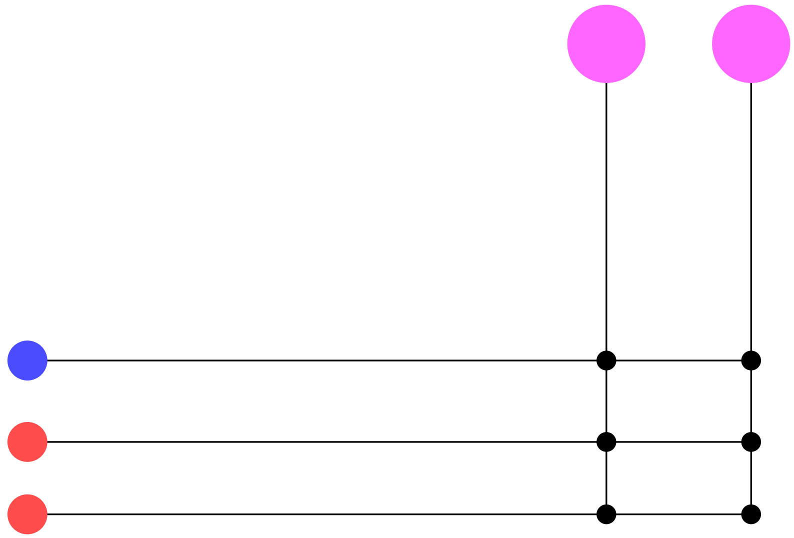
Cascade growing: first hidden unit



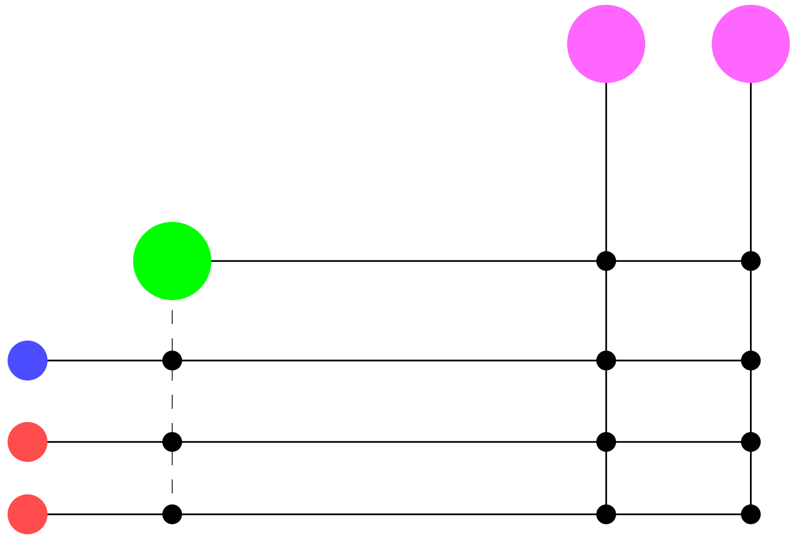
Cascade growing: second hidden unit



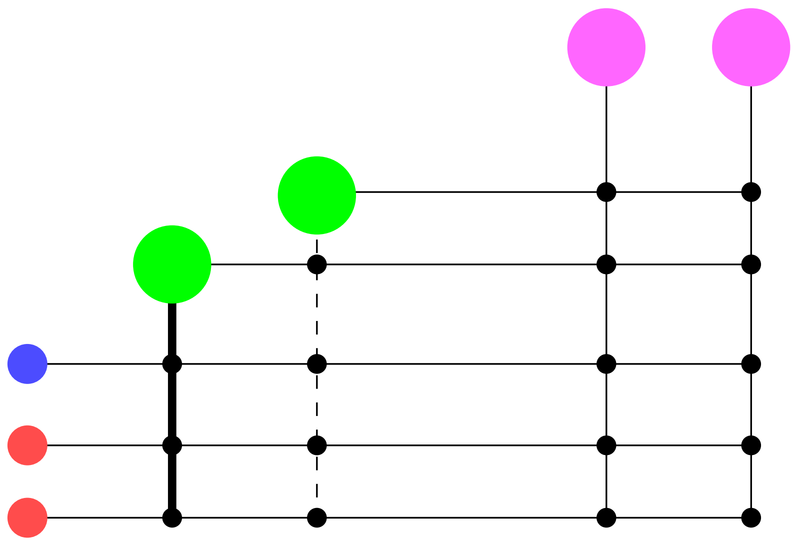
Cascade growing: alternative visualization



Cascade growing: alternative visualization

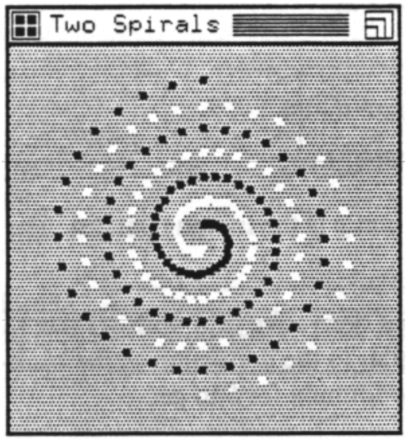


Cascade growing: alternative visualization

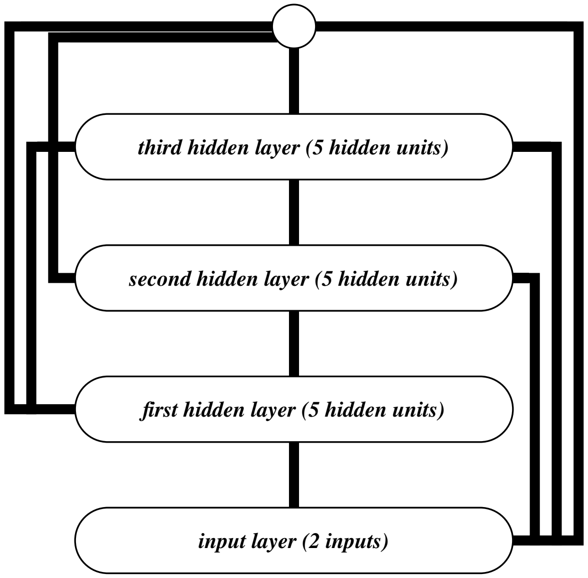


Cascade neural networks

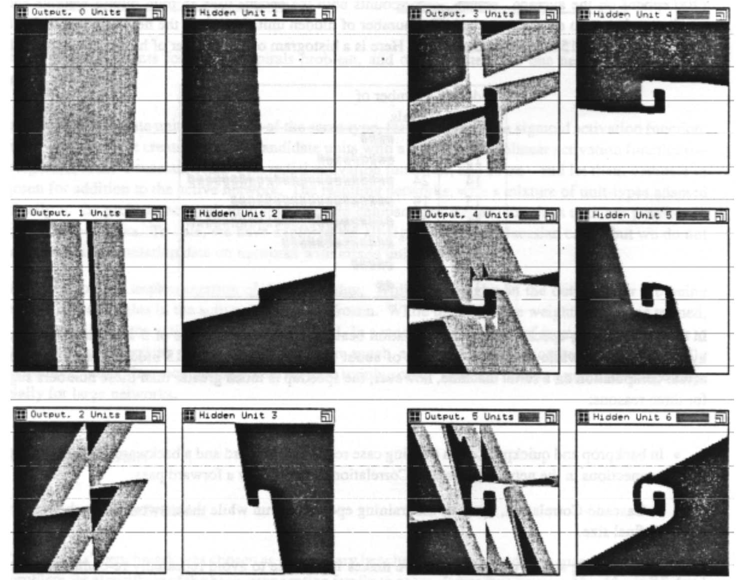
Do you ever need deeply nested structure?



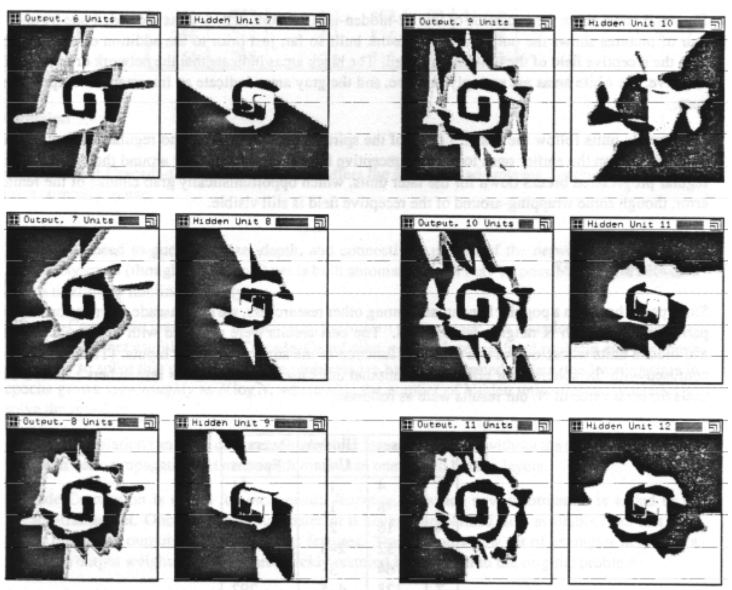
Two-spiral problem: best fixed architecture



Two-spiral problem: cascade architecture



Two-spiral problem: cascade architecture



Today's Discussion

To date:

- Neural networks: what are they
- Backpropagation: efficient gradient computation
- Advanced training: conjugate gradient

Today:

- CG postscript: scaled conjugate gradients
- Adaptive architectures
- My favorite neural network learning environment
- Some applications

NN environment that rocks...

Two problems with traditional neural networks:

- *Fixed* architecture
  - Difficult to guess "appropriate" architecture
  - Functional complexity requirements can vary widely
- *Slow* learning algorithms (e.g. backprop, quickprop)

My neural network approach:

- *Flexible* architecture
  - Cascade neural networks
  - Variable activation functions
- *Fast* learning algorithm (e.g. NDEKF)

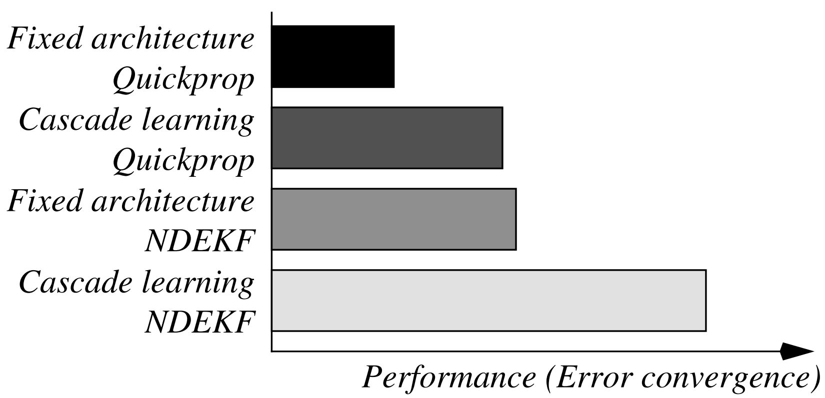
Cascade neural networks with node-decoupled extended Kalman filtering (NDEKF)

Types of problems investigated:

- Continuous function approximation
- Dynamic system modeling

Cascade learning and NDEKF combine to result in better error convergence.

A sneak peak at results

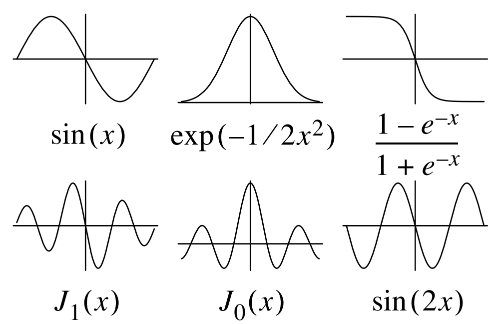


Additional flexibility: variable activations

Cascade neural networks already offer great flexibility...

However, why restrict candidate activation functions?

- Sigmoidal activation functions may not offer best results.
- Sinusoidals and/or others may be more appropriate:



Additional flexibility: variable activations

For continuous mapping problems:

- Variable networks converge to better minima.
- Sinusoidal networks — about same as variable networks.

Better learning: extended Kalman filtering

View neural network training problem as *system identification problem*.

- Let weights of neural network represent *state* of nonlinear dynamic system.
- Let neural network be that nonlinear system.

Extended Kalman filter training:

- Advantage: Explicitly accounts for pairwise interdependence of weights with conditional error covariance matrix.
- Disadvantage: $O(W^2)$ computational complexity, where W is number of weights in network.

Decoupled extended Kalman filtering

Key insight:

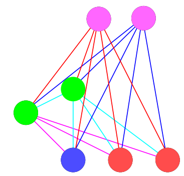
- Some weights are more interdependent than others.
- Group weights into groups.
- Ignore interdependence between groups of weights (block diagonalize conditional error covariance matrix).

Even better idea: Group weights by node!

Node-decoupled extended Kalman filtering

Key insight: Decouple (group) weights by node: *Natural formulation for cascade learning*

- One weight group for current hidden unit
- One additional weight group for each output unit



- Matrix operations reduce to vector operations.
- Computational complexity reduces to $O\left(\sum_i W_i^2\right)$.

Computational complexity

NDEKF requires inversion of an $m \times m$ matrix, (m = number of outputs)

Cascade learning with NDEKF typically requires less than 10 epochs/hidden unit.

- Several orders of magnitude less than backprop or quickprop approaches.
- Computational complexity similar to fixed-architecture networks trained with NDEKF.

Computational complexity

Ratio of computational cost between a cascade/NDEKF epoch and an equivalent fixed-architecture/backprop epoch (for few outputs):

- Example: for 400 inputs and 20 hidden units ratio is less than 100.
- Example: for 20 or less inputs, ratio is less than 10.

Experimental studies

Four learning approaches:

Symbol	*Explanation*
Fq	*fixed-architecture training with quickprop*
Cq	*cascade-network training with quickprop*
Fk	*fixed-architecture training with NDEKF*
Ck	*cascade-network training with NDEKF*

Experimental studies

Key questions:

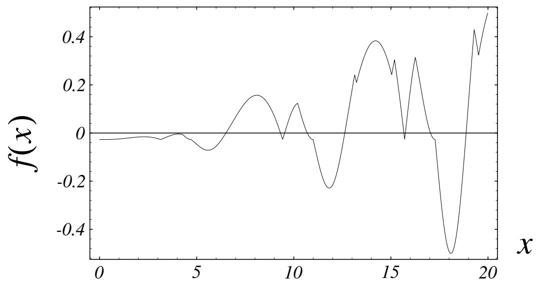
- Do we improve learning using NDEKF by going from fixed-architecture networks to cascade-type learning?
- Do we improve cascade learning by switching from quickprop (simple training) to NDEKF?
- Are any of more advanced methods (*Cq*, *Fk*, *Ck*) an improvement over baseline *Fq* (fixed-architecture/ quickprop) training method?

Five learning problems

Problem (A): ***smooth, continuous FA***

$$f_1(x, y, z) = z\sin(\pi y) + x$$
$$f_2(x, y, z) = z^2 + \cos(\pi xy) - y^2$$

Problem (B): ***nonsmooth, continuous FA***

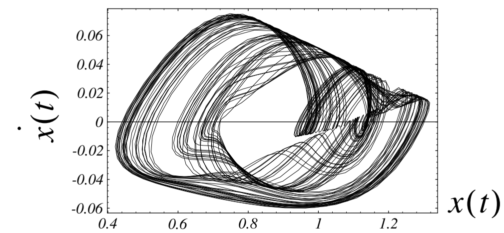


Five learning problems

Problem (C): ***deterministic dynamic system***

$$u(k+1) = f[u(k), u(k-1), u(k-2), x(k), x(k-1)]$$
$$f[x_1, x_2, x_3, x_4, x_5] = \frac{x_1 x_2 x_3 x_5 (x_3 - 1) + x_4}{1 + x_3^2 + x_2^2}$$

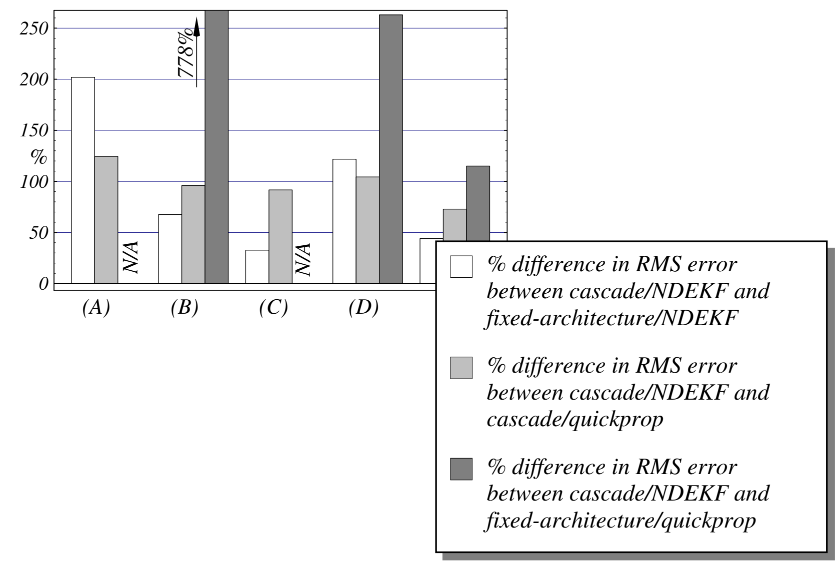
Problems (D) & (E): ***chaotic Mackey-Glass dynamic system (t+6)*** **and** ***(t+84)***



Learning results (avg. RMS error)

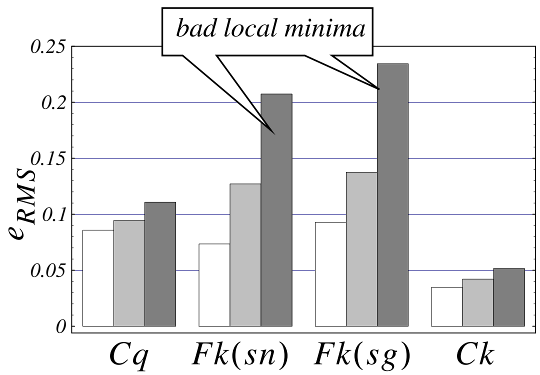
	Ck	*Fk*	*Cq*	*Fq*
(A)	*42.1 (4.2)*	*127.1 (37.3)*	*94.5 (6.2)*	*N/A*
(B)	*7.4 (2.0)*	*12.4 (3.2)*	*14.5 (4.0)*	*65.0 (18.2)*
(C)	*15.6 (1.5)*	*20.7 (4.8)*	*29.9 (2.0)*	*N/A*
(D)	*4.6 (0.6)*	*10.2 (4.0)*	*9.4 (2.7)*	*16.7 (2.2)*
(E)	*42.0 (5.9)*	*60.5 (3.1)*	*72.6 (16.3)*	*90.3 (8.3)*

Learning results



Why is *Ck* better than *Fk*?

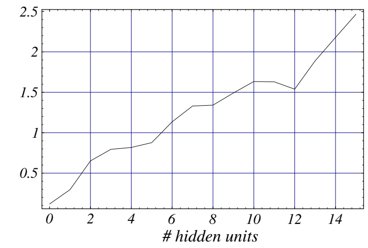
"NDEKF at times requires a small amount of redundancy in network in terms of total number of nodes in order to avoid poor local minima..." — [Puskorius & Feldkamp, 1991]



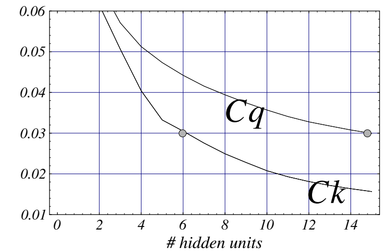
Problem (A)

Why is *Ck* better than *Cq*?

As hidden units are added in cascade learning, NDEKF is better equipped to handle increasingly correlated weights to new hidden units.



Problem (C)



Cascade/NDEKF advantages/disadvantages

- Cascade learning and NDEKF complement each other well.
- Cascade learning minimizes the potentially detrimental effect of node-decoupling.
- Cascade learning minimizes the problem of poor local minima in NDEKF.
- NDEKF better handles the increased correlation of weights as the number of hidden units increases in cascade learning.
- NDEKF requires no learning parameter tuning.
- Cascade/NDEKF converges efficiently to better local minima than either cascade or NDEKF by themselves.
- ***Disadvantage:*** computationally efficient with few outputs.

Today's Discussion

To date:

- Neural networks: what are they
- Backpropagation: efficient gradient computation
- Advanced training: conjugate gradient

Today:

- CG postscript: scaled conjugate gradients
- Adaptive architectures
- My favorite neural network learning environment
- Some applications