Fault Tolerance in Feed-Forward Neural Networks

Course Seminar
Neural Networks (CS623)

Faraz Shahbazker
<farazs@cse.iitb.ac.in>

Arindam Bose
<arindam@cse.iitb.ac.in>

Indian Institute of Technology, Bombay
12th November, 2006.

Under: Prof. Pushpak Bhattacharya
<pb@cse.iitb.ac.in>
Development Cycle of ANN

- Design
- Training
- Testing
- Deployment (software)
- Deployment (Hardware)
Motivation

- NN Hardware is faster due to inherent hardware-level parallelism
  - Portable devices for speech recognition / biometrics / image-processing / control systems for safety-critical devices
- On-field deployment requires transparent resilience to component failure
- Not possible to plug-in and modify network hardware on the fly
- For Neural Networks, fault tolerance in s/w design is much cheaper than h/w redundancy
Agenda

• Great Expectations
• Ground Realities
• Defining Fault Tolerance
• Training to improve Fault Tolerance
  – Training with Injected Faults
  – Addition/Deletion Procedure (ADP)
  – Constraint Back-propagation (CBP)
• Conclusion & References
Great Expectations

- ANN are widely considered fault-tolerant
- By virtue of Biological heritage
- Empirical data does **NOT** always support
- **Potentially** fault tolerant
- Redundancy != Fault Tolerance
Ground Realities

- Few proven mathematical model exist...
- Most results based on heuristics and experimental observations
- Training procedures need to be modified
Defining Fault Tolerance

- No single point of failure
- Performance should degrade *gracefully*
- Quantified point of complete breakdown
Types of Failure in ANN

- Neuron Failure
- Link Failure
Types of Failure

- Neuron Failure
  - Stuck at 0
    - Neutral impact on next layer
  - Stuck at 1
    - Maximum activation / inhibition for next layer
Types of Failure

- Link Failure
  - Stuck at MinVal
  - Max Inhibition
  - Stuck at MaxVal
  - Max Activation
  - Stuck at 0
  - Neutral
    (Missing)
Training Algorithms

1. Training with Injected Faults
Training with Injected Faults

- Brute-force approach
- Assume that all components (links / neurons) have \textbf{equal} importance.
- During training, randomly \textit{fail} few neurons/links for few iterations at regular intervals
- Training time increases \textbf{dramatically}
- Network \textbf{learns} to be resilient to failures (hopefully!!)
Training Algorithms

1. Training with Injected Faults
2. Addition Deletion Procedure
Impact of Failure

- Are there any hot-spots in the network?
- Are there any singular point of failure?
- Failure of each neuron/link contributes to an increase in system error (MSE/SSE).
- Failure of which neuron/link, contributes the most?
- Concept of Sensitivity ...
Impact of Failure

- Sensitivity: (of a neuron)
  - The impact of failure of that neuron on overall system error
  - Simulate different types of failure of neuron (stuck at 0 / stuck at 1)
  - Calculate **change in network error** for each failure over entire training set
  - Add these and average over types of failures (in our case 2).
Impact of Failure

- \( W \): vector of \textbf{all weights}
- \( E(W) \): Sum error over all weights
- \( E(W, o(n_j) = \alpha) \): error with neuron \( n_j \) stuck at \( \alpha \)
- \( B \): set of all possible faults \( \alpha = \{0, 1\} \)
- Sensitivity:

\[
\begin{align*}
S^n_\alpha(n_j) &= E(W, o(n_j) = \alpha) - E(W) \\
S(n_j) &= \frac{1}{|B|} \sum_{\alpha \in B} S^n_\alpha(n_j)
\end{align*}
\]
Addition/Deletion Procedure (ADP)

- Smarter approach
- Train network with Back-propagation as usual
- At end of training, different components may have different impact on network
- But we want them to have equal impact
- We will use our knowledge of sensitivity to distribute load equally across all neurons in a layer
Addition/Deletion Procedure (ADP)

- Rank neurons according to sensitivity
- Eliminate dead-spots by substituting the least sensitive neuron with fresh neuron
- This step *increases* network error within limits
- Eliminate hot-spots by configuring new neuron to share load of most sensitive neuron
- Retrain to remove chinks (very few iterations required).
Addition/Deletion Procedure (ADP)

- **Load Sharing algorithm:**
  - Let $n_{\text{max}} = \text{most sensitive neuron}$
    $n_{\text{new}} = \text{freshly added neuron}$
  - input weights to $n_{\text{new}} =$
    input weights to $n_{\text{max}}$
  - output weights from $n_{\text{new}} =$
    $1/2(\text{output weights from } n_{\text{max}})$
  - output weights from $n_{\text{max}} =$
    $1/2(\text{output weights from } n_{\text{max}})$
Training Algorithms

1. Training with Injected Faults
2. Addition Deletion Procedure
3. Constraint Backpropagation
Constraint Backpropagation

- Most sophisticated approach so far
- Fault-tolerance *built-in* into Back-prop
- In each training iteration:
  - minimize Global error (as usual)
  - minimize susceptibility to failure (called Constraint Energy $E_c$)
- Degree of fault-tolerance can be *quantified*
Constraint Backpropagation

- \( m \): degree of fault tolerance
- \( R \): Set of neurons in hidden layer
- \( \alpha \): type of fault \{0, 1\}
- \( \tilde{m}_\alpha \): subset of \( R \) of size \( m \) with output set to \( \alpha \)
- \( o(\tilde{m}_\alpha) \): net output with elements of \( \tilde{m}_\alpha \) set to \( \alpha \)

Minimize:

\[
E = \frac{1}{2} \sum_{i/p} \sum_{o/p} (t - o)^2
\]

For \( \tilde{m}_\alpha \) from \( R \) minimize:

\[
E_C = \frac{1}{2} \sum_{i/p} \sum_{o/p} (t - o(\tilde{m}_\alpha))^2 \quad \ldots \times \binom{|R|}{m}
\]
Constraint Backpropagation

- Trained network **guarantees** fault tolerance of up to \( m \) neurons
- Degree of fault tolerance can be varied via training parameter \((m)\)
- We traverse a set of \( \binom{|R|}{m} \) error surfaces simultaneously
- What is the effect of varying \( m \)?
Constraint Backpropagation

- Smaller $m$
  - $\binom{|R|}{m}$ is small $\Rightarrow$ fewer error surfaces
  - small variations in error surfaces
- Medium $m$
  - $\binom{|R|}{m}$ is large $\Rightarrow$ more error surfaces
  - significant variation in error surfaces
- Large $m$
  - $\binom{|R|}{m}$ is small $\Rightarrow$ fewer error surfaces
  - Large variation across error surfaces
  - CBP may never converge
Conclusion

- Neural networks are **not** inherently fault-tolerant - but potential exists!!
- Training procedures need to be modified
- Sometimes, redundancy helps
- No explicit fault-handling required
- Resilience comes implicitly by clever design and training algorithms
- Guaranteed Fault-tolerance requires extra effort in training stage
References


