Statistical Machine Learning

Lecture 10: Neural Networks

W.Q.Cui Research Group

Department of Statistics and Finance
University of Science and Technology of China

2018 Autumn

Projection Pursuit Regression

Projection Pursuit Regression

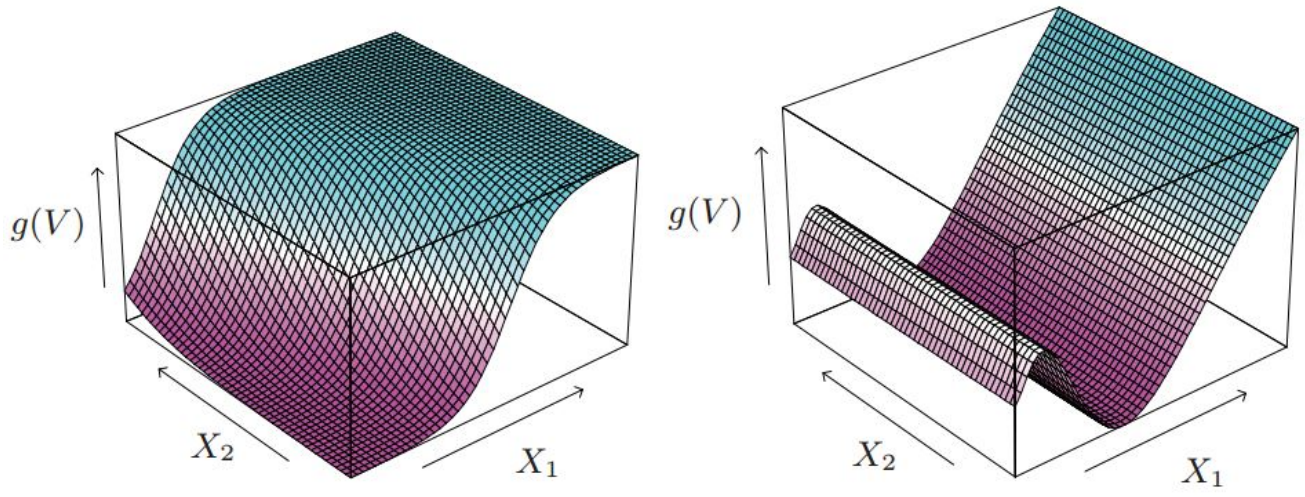
Projection Pursuit Regression model:

$$f(x) = \sum_{i=1}^{M} g_m(\omega_m^t X)$$

where $X \in \mathbb{R}^p$ and have targets $Y \in \mathbb{R}$.

- Additive model in the derived features $V_m = \omega_m^t X$
- $g_m(\omega_m^t X)$ the ridge function in $\mathbb{R}^p$ – only varies in direction of ω_m.
- PPR model can approximate any continuous function in $\mathbb{R}^p$ if M arbitrarily large and appropriate choice of $g_m^{'}$s.
- $\Rightarrow$ PPR model is a universal approximator.

Example Ridge Functions



- **Left graph**

$$g(V) = \frac{1}{1 + exp(-5(V - 0.5))}, \quad V = \frac{X_1 + X_2}{\sqrt{2}}$$

- **Right graph**

$$g(V) = (V + 0.1)sin(\frac{1}{V/3 + 0.1}), \quad V = X_1$$

How to Fit A PPP Model?

- Have training data $\{(x_i, y_i)\}_{i=1}^n$
- Seek to minimize

$$\sum_{i=1}^{n} \left[y_i - \sum_{m=1}^{M} g_m(\omega_m^t x_i) \right]^2$$

over functions g_m and directions ω_m, $m = 1, \ldots, M$.

- **How**??

How to Fit A PPP Model?

- Have training data $\{(x_i, y_i)\}_{i=1}^n$
- Seek to minimize

$$\sum_{i=1}^{n} \left[y_i - \sum_{m=1}^{M} g_m(\omega_m^t x_i) \right]^2$$

  over functions g_m and directions ω_m, $m = 1, \ldots, M$.

- **How**??
- General approach
  - Build model in a forward stage-wise manner. Add a pair (ω_m, g_m) at each stage.
  - At each stage iterate
    - Fix ω_m and update g_m
    - Fix g_m and update ω_m

How to Fit A PPP Model?

- **Fix** ω and **update** g
- Must impose complexity constraints on g_m to avoid overfitting.
- **Fix** g and **update** ω

$$g(\omega^t x_i) \approx g(\omega_{old}^t x_i) + g'(\omega_{old}^t x_i)(\omega - \omega_{old})^T x_i$$

to give

$$\sum_{i=1}^{n}[y_i - g(\omega^t x_i)]^2 \approx \sum_{i=1}^{n} g^{'}(\omega_{old}^t x_i)^2 \left[(\omega_{old}^t x_i + \frac{y_i - g(\omega_{old}^t x_i)}{g'(\omega_{old}^t x_i)}) - \omega^t x_i\right]^2$$

To minimize the rhs:

- Perform least squares regression (no intercept (bias) term).
- Input $\omega^t x_i$ has target $\omega_{old}^t x_i + \frac{y_i - g(\omega_{old}^t x_i)}{g'(\omega_{old}^t x_i)}$
- Weight errors with $g'(\omega_{old}^t x_i)^2$
- This produces the updated coefficient vector ω_{new}.

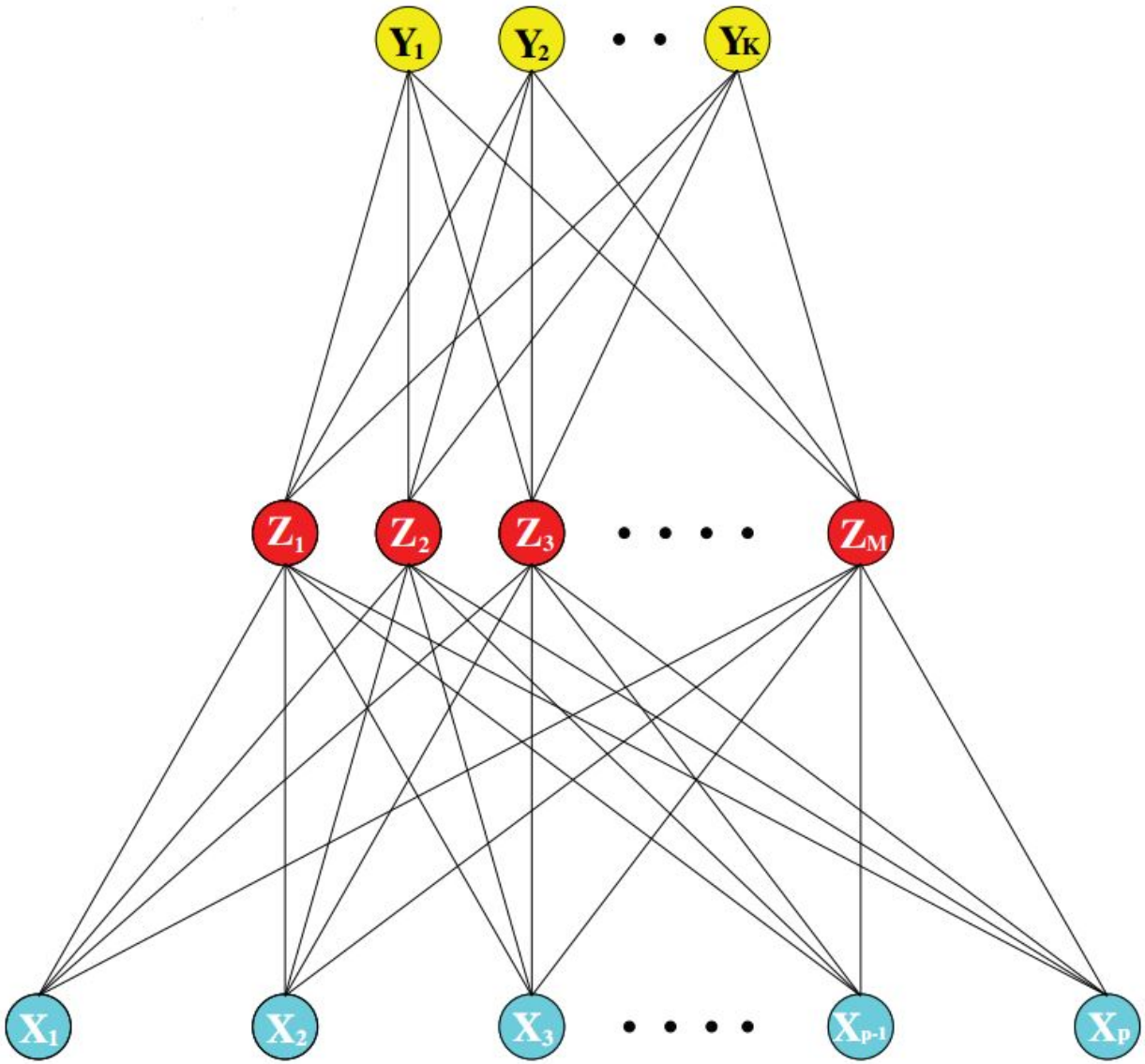
How to Fit A PPP Model?

Iterate these two steps until convergence

- **Fix** ω and **update** g
- **Fix** g and **update** ω

Neural Networks

Single hidden layer, feed-forward network



K-Classification

- **Input**: $X = (X_1, X_2, \ldots, X_p)$ and say it belongs to class k
- **Ideal output**: $Y_1, \ldots, Y_K$ where

$$Y_i = \begin{cases} 0, & \text{if } i \neq k \\ 1, & \text{if } i = k \end{cases}$$

- The 2 layer neural network estimates the outputs by
  - deriving features $Z_1, \ldots, Z_M$ - hidden units - from linear combinations of X.
  - the target Y_k is modeled as a function of linear combinations of $Z_1, \ldots, Z_M$.

K-Classification

Computation of the kth output

$$Y_k = f_k(X) = g_k(T_1, \ldots, T_K)$$

where

$$T_k = \beta_{k0} + \sum_{m=1}^{M} \beta_{km} Z_m$$

$$Z_m = \sigma(\alpha_{m0} + \sum_{\ell=1}^{p} \alpha_{m\ell} X_\ell)$$

the activation function σ can be defined

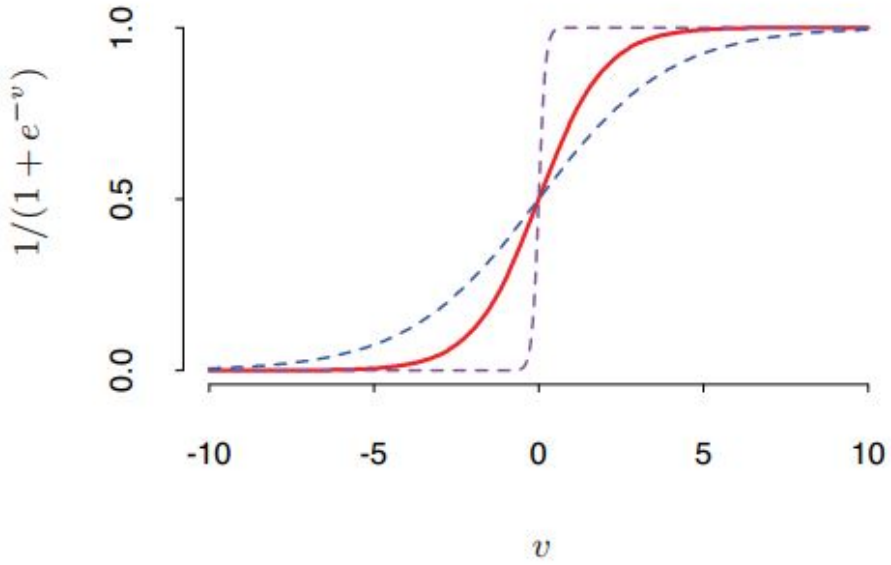
$$\text{sigmoid function:} \quad \sigma(v) = \frac{1}{1 + exp(-v)}$$

and the output function g_k

$$\text{softmax function:} \quad g_k(T_1, \ldots, T_K) = \frac{exp(T_K)}{\sum_{\ell=1}^{K} exp(T_\ell)}$$

The Activation Function

Shown is $\sigma(sv)$ for $s = .5, 1, 10$



- If σ is the identity $\Rightarrow$ each $T_k = \omega_{k0} + \sum_{\ell=1}^{p} \omega_{k\ell} X_\ell$
- Can think of **neural networks** as a non-linear generalization of the linear model.
- Rate of activation of the *sigmoid* depends on the norm of α_m where $Z_m = \sigma(\alpha_{m0} + \sum_{\ell=1}^{p} \alpha_{m\ell} X_\ell)$
- When $\|\sigma_m\|$ is small $\Rightarrow$ unit operates in the linear part of its activation function.

Neural Network is A Universal Approximator

A Neural Network with one hidden units, can approximate arbitrarily well any functional continuous mapping from one finite dimensional space to another, provided number of hidden units is sufficiently large.

Fitting Neural Networks

Error measure

- This 2-layer neural network has unknown parameters θ,

$$\{\alpha_{m0}, \alpha_{m1}, \ldots, \alpha_{mp}; m = 1, \ldots, M\} M(p+1) \text{weights}$$
$$\{\beta_{k0}, \beta_{k1}, \ldots, \beta_{kM}; k = 1, \ldots, K\} K(M+1) \text{weights}$$

- Aim: Estimate parameters θ, from labeled training data:

$$\{x_i, g_i\}_{i=1}^{n} \quad \text{with each} x_i \in \mathbb{R}^p, g_i \in \{1, \ldots, K\}$$

- Do this by minimizing a measure-of-fit such as

$$R(\theta) = \sum_{i=1}^{n} \sum_{k=1}^{K} (y_{ik} - f_k(x_i))^2 \quad \leftarrow \text{sum-of-squared error}$$

or

$$R(\theta) = -\sum_{i=1}^{n} \sum_{k=1}^{K} y_{ik} log f_k(x_i) \quad \leftarrow \text{cross-entropy error}$$

Minimizing $R(\theta)$

- Typically don't want
$$\hat{\theta} = \arg\min_{\theta} R(\theta)$$
$\Rightarrow$ an overfit solution.
- Some form of regularization is required. - will come back to this.
- Generic approach to minimizing $R(\theta)$ is by gradient descent a.k.a. back-propagation.
- This amounts to implementation of the chain rule for differentiation.

Back-propagation for squared-error loss

- Let $z_i = (z_{1i}, \ldots, z_{Mi})$ and

$$z_{im} = \sigma(\alpha_{m0} + \alpha_m^t x_i) \quad \text{where} \alpha_m = (\alpha_{m1}, \ldots, \alpha_{mp})$$

- Have

$$R(\theta) = \sum_{i=1}^{n} R_i = \sum_{i=1}^{n} \sum_{k=1}^{K} (y_{ik} - f_k(x_i))^2$$

with derivatives

$$\frac{\partial R_i(\theta)}{\partial \beta_{km}} = -2(y_{ik} - f_k(x_i)) g_k'(\beta_{10} + \beta_1^t z_i, \ldots, \beta_{K0} + \beta_K^t z_i) z_{im} = \delta_{ki} z_{im}$$

$$\frac{\partial R_i(\theta)}{\partial \alpha_{m\ell}} = \sum_{k=1}^{K} \delta_{ki} \beta_{km} \sigma'(\alpha_{m0} + \alpha_m^t x_i) x_{i\ell}$$

$$= x_{i\ell} \sigma'(\alpha_{m0} + \alpha_m^t x_i) \sum_{k=1}^{K} \delta_{ki} \beta_{km} = x_{i\ell} s_{mi}$$

Back-propagation for squared-error loss

- Given these derivatives update at the $(r+1)$st iteration

$$\beta_{km}^{(r+1)} = \beta_{km}^{(r)} - \gamma_r \sum_{i=1}^{n} \frac{\partial R_i(\theta)}{\partial \beta_{km}}\Big|_{\beta_{km}=\beta_{km}^{(r)}}$$

$$\alpha_{km}^{(r+1)} = \alpha_{km}^{(r)} - \gamma_r \sum_{i=1}^{n} \frac{\partial R_i(\theta)}{\partial \alpha_{m\ell}}\Big|_{\alpha_{m\ell}=\alpha_{m\ell}^{(r)}}$$

  where γ_r is the learning rate.

- The quantities δ_{ki} and s_{mi} are "errors" from the current model at the output and hidden layer units respective

$$\frac{\partial R_i(\theta)}{\partial \beta_{km}} = \delta_{ki} z_{im}, \qquad \frac{\partial R_i(\theta)}{\partial \alpha_{m\ell}} = x_{i\ell} s_{mi}$$

- Remember the errors satisfy

$$s_{mi} = \sigma'(\alpha_{m0} + \alpha_m^t x_i) \sum_{k=1}^{K} \delta_{ki} \beta_{km}$$

Back-propagation update equations

The updates can be implemented in a two-pass algorithm:

- **Forward pass**: current weights are fixed and compute $\hat{f}_k(x_i)$
- **Backward pass**: Compute errors δ_{ki} and then back-propagated with

$$s_{mi} = \sigma'(\alpha_{m0} + \alpha_m^t x_i) \sum_{k=1}^{K} \delta_{ki} \beta_{km}$$

  to give the errors s_{mi}.

- Use both sets of errors to compute the gradients for the updates.

Details of back propagation

- Can do updates with batch learning.
  Parameters updated by summing over all training examples.

- Can do updates with online learning.
  Parameters updated after each training example.
  $\Rightarrow$ can train network with very large trained datasets.

- Training epoch $\equiv$ one sweep through the entire training set.

- **Learning rate**:γ_r
  - Batch learning — usually taken to be constant and can be optimized by a line search.
  - Online learning — $\gamma_r \to 0$ as $r \to \inf$
- **Note**: Back-prop is very slow.

Some Issues in Training Neural Networks

Training Neural Networks

Training a neural networks is non-trivial!

Why?

1. Model is overparametrized
2. Optimization problem is nonconvex and unstable

Starting Values

- If weights are near zeros
  $\Rightarrow \sigma(\cdot)$ is roughly linear
  $\Rightarrow$ neural network collapses into an approx linear model.

- Usually start with random values close to zero.
  $\Rightarrow$ model starts out linear and becomes non-linear as weights increase.

- Use of exact zero weights gives zero derivatives, perfect symmetry and the algorithm never moves.

- Starting with large weights often leads to poor solutions.

Combating Overfitting

Neural networks will overfit at the global minimum of R.
Therefore different approaches to *regularization* have been adopted:

- **Early stopping**
  1. Only train the model for a while.s
  2. Stop before converging to a minimum of $R(\theta)$
  3. As initial weights are close to 0
     $\Rightarrow$ initially have a highly regularized linear solution
     $\Rightarrow$ early stopping shrinks the model towards a linear model.
  4. Can use a validation dataset to determine when to stop.

Combating Overfitting

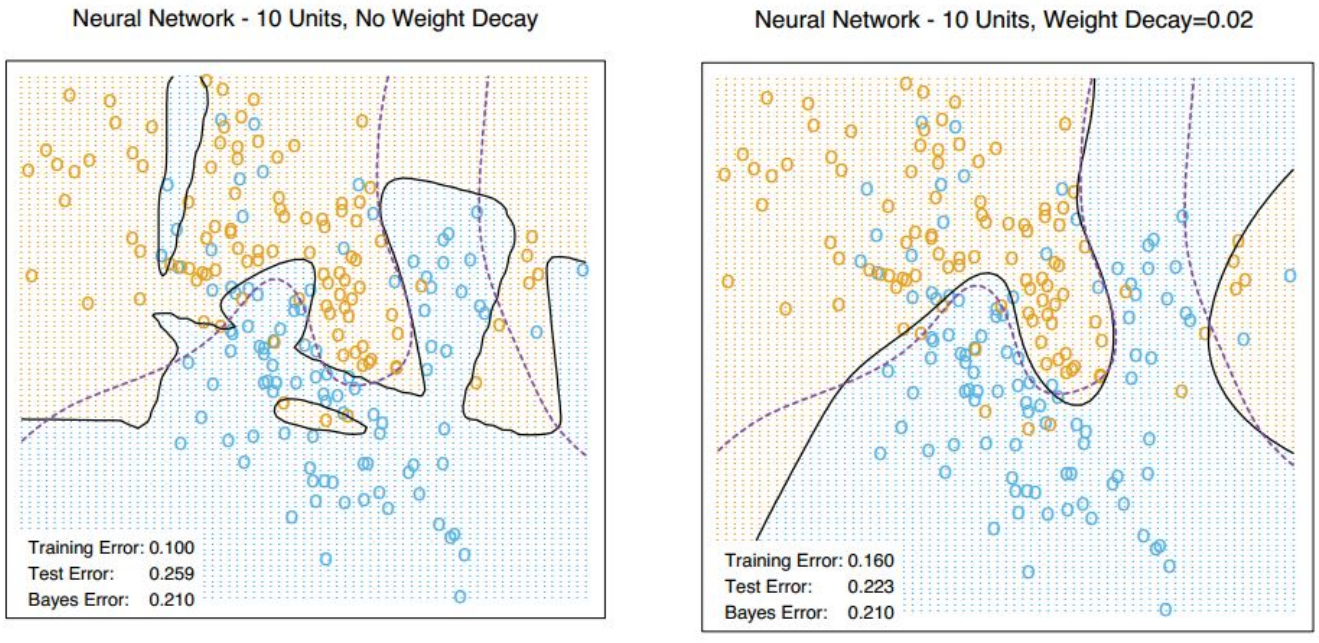
Neural networks will overfit at the global minimum of R.
Therefore different approaches to *regularization* have been adopted:

- **Early stopping**
  1. Only train the model for a while.s
  2. Stop before converging to a minimum of $R(\theta)$
  3. As initial weights are close to 0
     $\Rightarrow$ initially have a highly regularized linear solution
     $\Rightarrow$ early stopping shrinks the model towards a linear model.
  4. Can use a validation dataset to determine when to stop.
- **Weight decay**
  1. Add a penalty to the error function $R(\theta) + \lambda J(\theta)$, where

$$J(\theta) = \sum_{km} \beta_{km}^2 + \sum_{m\ell} \alpha_{m\ell}^2$$

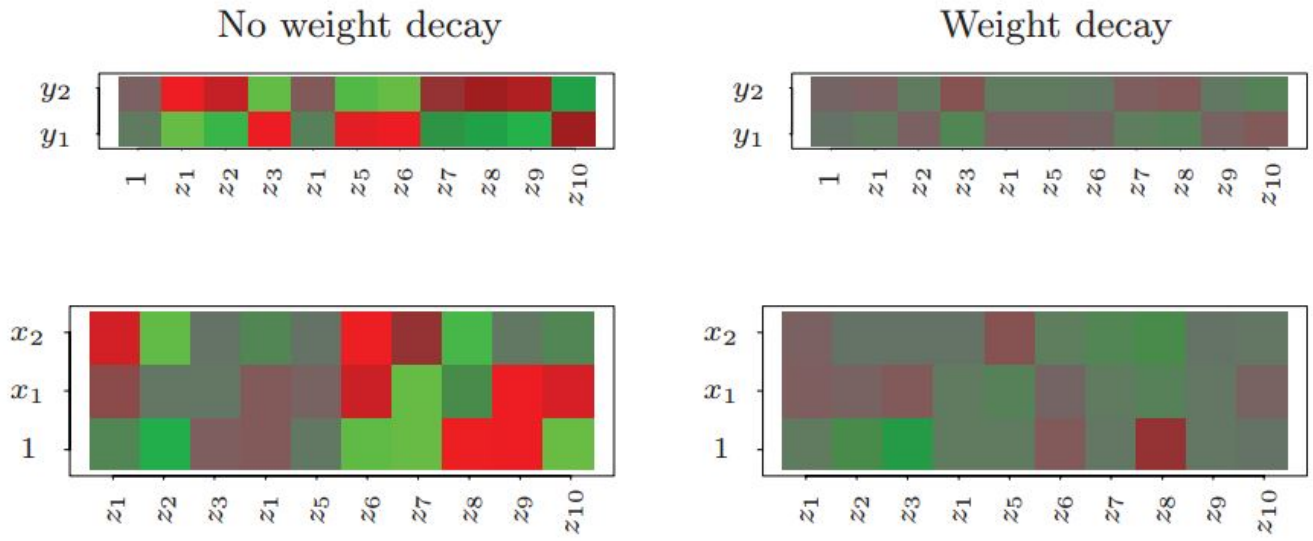
     and $\lambda \leq 0$ is a tuning parameter.
  2. Larger values of λ tend to shrink weights towards zero.

Effect of Weight Decay



- Both use softmax g_k and cross-entropy error.
- Bayes optimal decision boundray is the purple curve

Weights learnt



- Both use softmax g_k and cross-entropy error.
- The display ranges from bright green (negative) to bright red (positive).

Combating Overfitting

- **Scaling the Inputs**
  - Scale of inputs determines scale of bottom layer weights.
  - At beginning best to standardize all inputs to have mean 0 and standard deviation 1
  - Ensures all inputs are treated equally in the regularization process.

- **Number of Hidden Units and Layers**
  - Generally better to have too many than too few hidden units.
  - Fewer hidden units $\Rightarrow$ less flexibility in the model
  - Proper regularization should shrink unnecessary hidden unit weights to zero.
  - Multiple hidden layers allows construction of hierarchical features at different resolutions.

Combating Overfitting

Multiple Minima

- $R(\theta)$ non-convex $\Rightarrow$ final solution depends on initial weights.
- Option 1:
  Learn different networks for different random initial weights.
  Choose the network with lowest penalized error.
- Option 2:
  Learn different networks for different random initial weights.
  For a test example average the prediction of each network.
- Option 3: (bagging)
  Learn different networks from random subsets of the training data.
  For a test example average the prediction of each network.

Example: Simulated Data

Example 1: Underlying model

- Generated data from this additive model

$$Y = \sigma(a_1^t X) + \sigma(a_2^t X) + \epsilon$$

where

$$\begin{aligned} X &= (X_1, X_2)^t \quad \text{with } X_i \sim \mathcal{N}(0,1) \text{ for } i = 1,2 \\ a_1^t &= (3,3) \\ a_2^t &= (3,-3) \\ \epsilon &\sim \mathcal{N}(0,\sigma^2) \end{aligned}$$

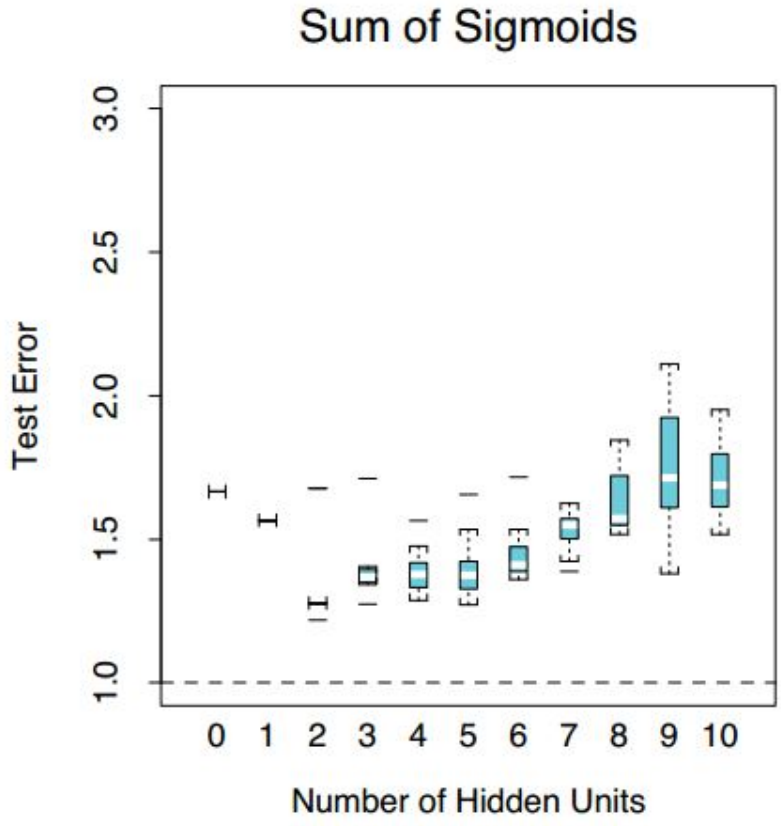
and σ^2 is chosen so the s-n-r is 4 that is

$$Var\{f(X)\} = 4\sigma^2$$

- $n_{train} = 100$ and $n_{test} = 10000$

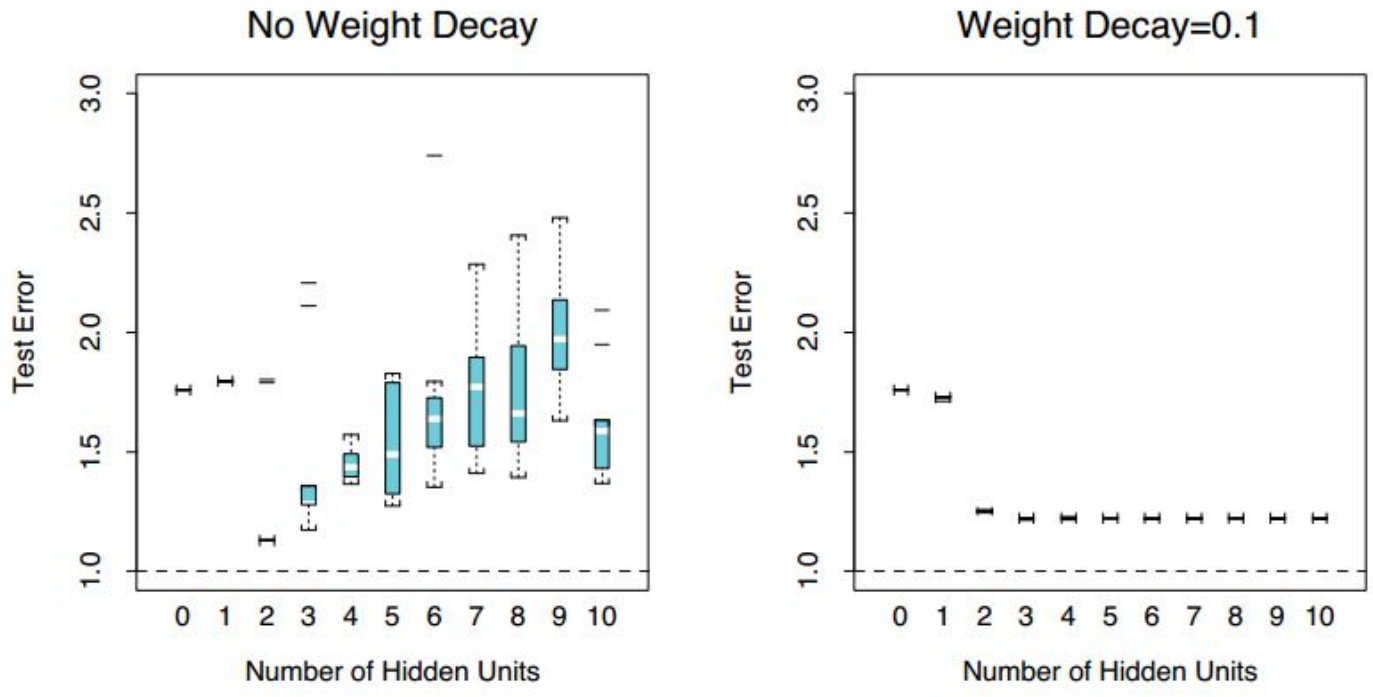
Example 1: Neural network fit

- Fit neural network with weight decay and various number of hidden units.
- Recorded the average test error for 10 random starting weights.
- Zero hidden unit model refers to linear least squares regression.

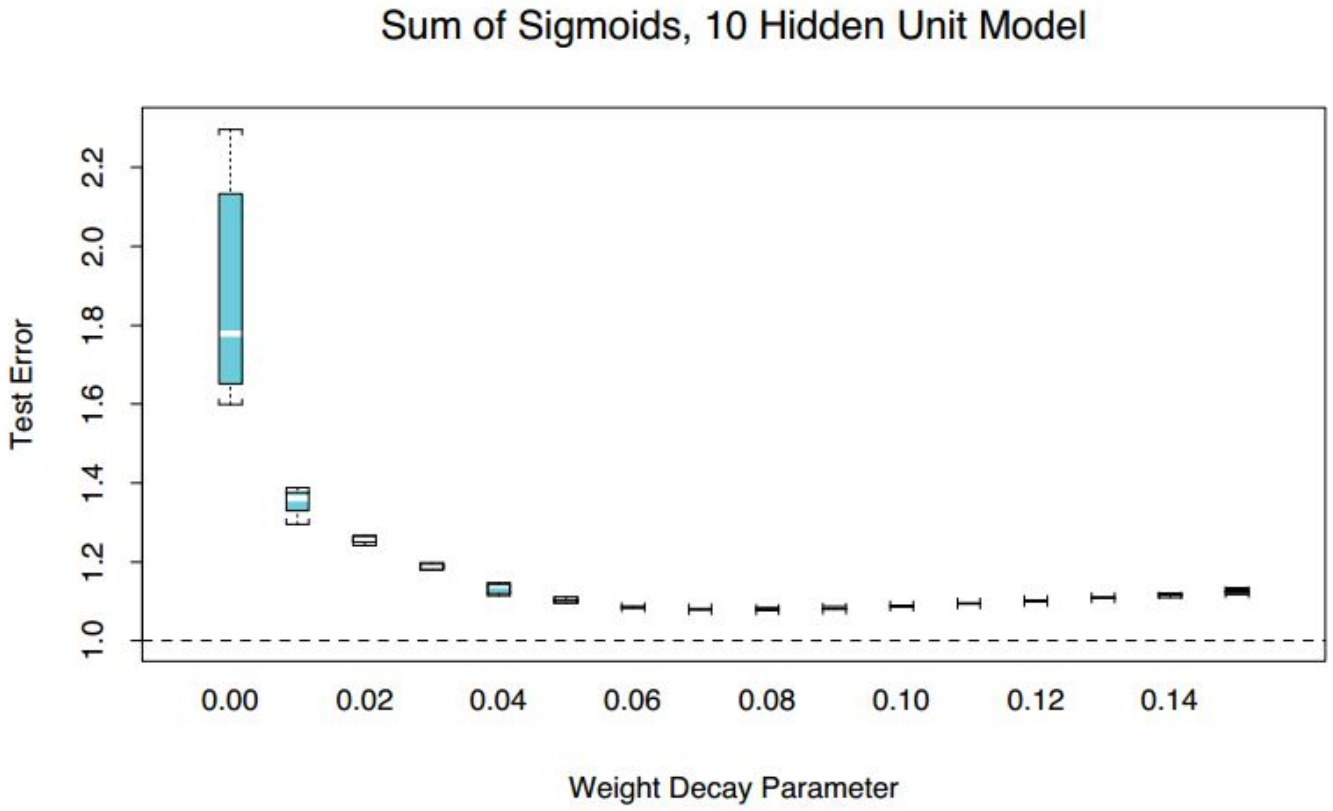


Test error quoted relative to the Bayes error, $\lambda = 0.0005$

Example 1: Effect of weight decay on test error



Example 1: Fixed number of hidden units, vary λ



Example 2: Underlying model

- Generated data from this additive model

$$Y = \prod_{j=1}^{10} \phi(X_j) + \epsilon$$

where

$$X = (X_1, \ldots, X_10)^t \quad \text{with } X_i \sim \mathcal{N}(0,1) \text{ for } i = 1, \ldots, 10$$
$$\phi(v) = exp(-v^2/2)/\sqrt{2\pi}$$
$$\epsilon \sim \mathcal{N}(0, \sigma^2)$$

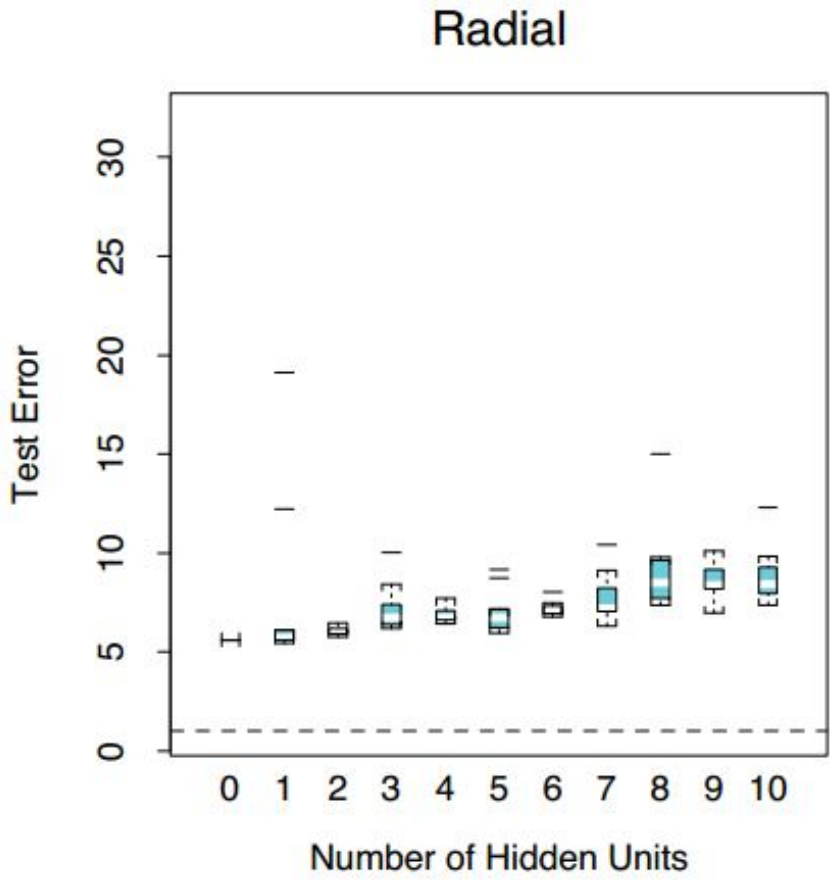
and σ^2 is chosen so the s-n-r is 4 that is

$$Var\{f(X)\} = 4\sigma^2$$

- $n_{train} = 100$ and $n_{test} = 10000$

Example 2: Neural network does not produce a good fit

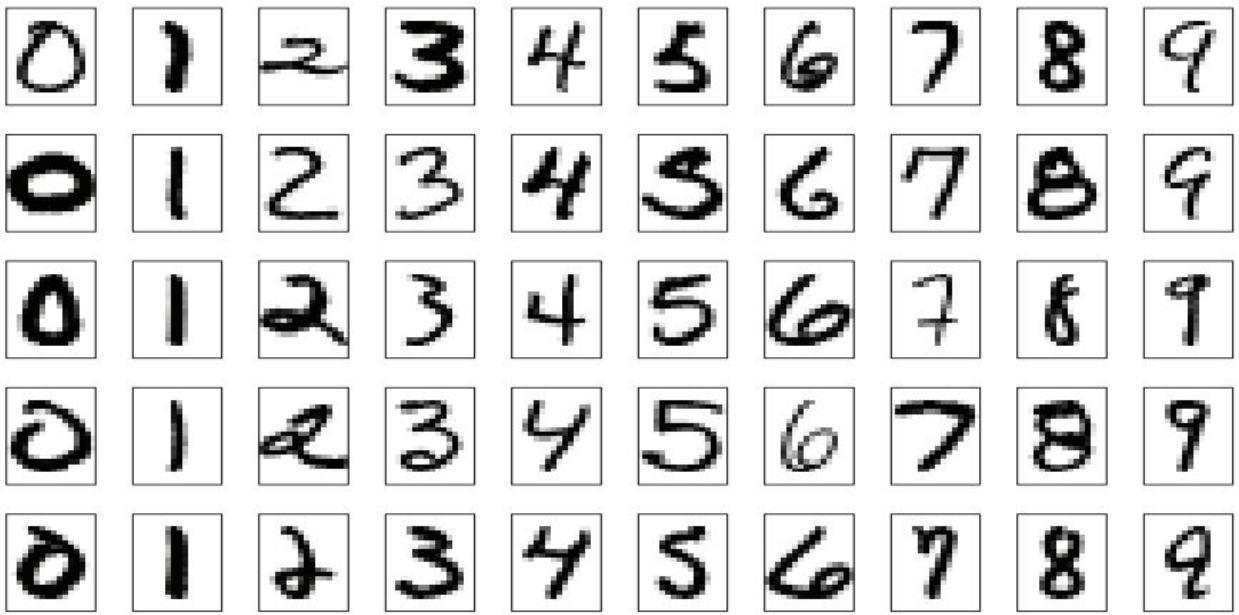
- Fit neural network with weight decay and various number of hidden units.
- Recorded the average test error for 10 random starting weights.
- Zero hidden unit model refers to linear least squares regression.



Test error quoted relative to the Bayes error, $\lambda = 0.0005$

Example: ZIP Code Data

The Data

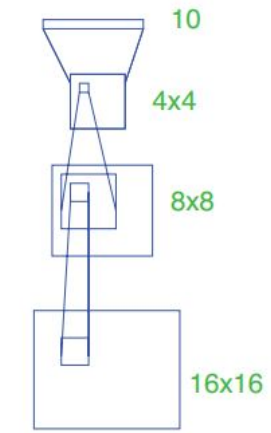


Each image is a 16×16 8-bit grayscale representation of a handwritten digit.
For the experiments in the book: $n_{train} = 320$ and $n_{test} = 160$.

Five networks fit to the data

- **Net-1**
  No hidden layer, equivalent to multinomial logistic regression.
- **Net-2**
  One hidden layer, 12 hidden units fully connected.
- **Net-3**
  Two hidden layers locally connected.

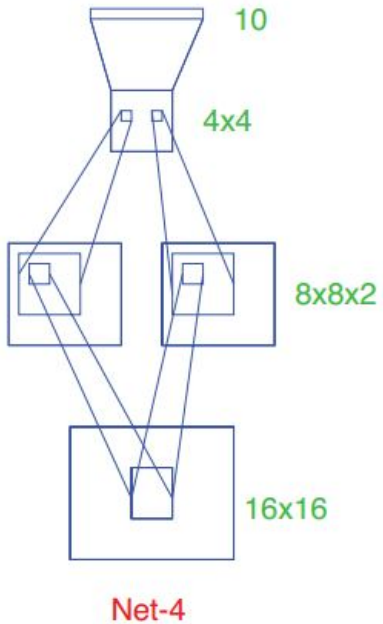
  - 1st hidden layer (8×8 array), each unit takes inputs from a 3×3 patch of the input layer after subsampling by 2.
  - 2nd hidden layer, inputs are from a 5×5 patch of the input layer after subsampling by 2.
  - Local connectivity makes each unit responsible for extracting local features from the layer below.



Five networks fit to the data

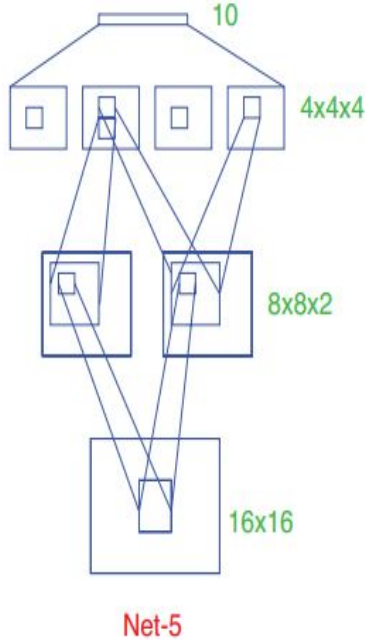
- **Net-4** (convolutional neural network)
  Two hidden layers, locally connected with weight sharing.

  - 1st hidden layer has two 8×8 arrays. Each unit takes inputs from a 3×3 patch of the input layer after subsampling by 2. The units in the feature map share the same set of nine weights (but have their own bias parameter).
  - 2nd hidden layer, inputs are from a $5 \times 5 \times 2$ volume of the two input layers after subsampling by 2. It has no weight sharing.
  - Local connectivity makes each unit responsible for extracting local features from the layer below.



Five networks fit to the data

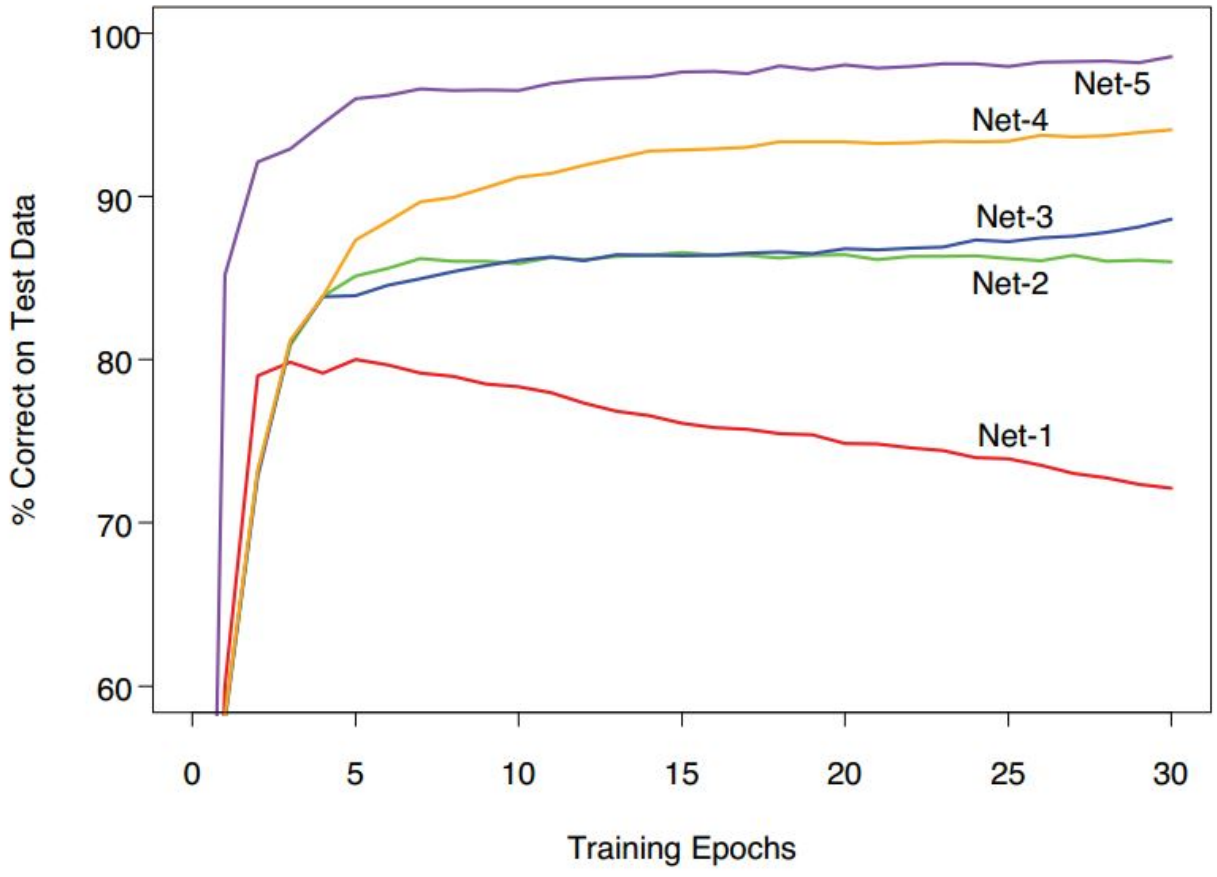
- **Net-5** (convolutional neural network)
  Two hidden layers, locally connected with weight sharing.
  - 1st hidden layer has two 8×8 arrays. Each unit takes inputs from a 3×3 patch of the input layer after subsampling by 2. The units in the feature map share the same set of nine weights (but have their own bias parameter).
  - 2nd hidden layer has four 4×4 feature maps. Inputs are from a $5 \times 5 \times 2$ volume of the two input layers after subsampling by 2. The units in the feature map share the same set of 50 weights (but have their own bias parameter).
  - Local connectivity makes each unit responsible for extracting local features from the layer below.



Number of parameters

Network Architecture	Links	Weights	%Correct
Net-1: Single layer network	2570	2570	80.0%
Net-2: Two layer network	3214	3214	87.0%
Net-3: Locally connected	1226	1226	88.5%
Net-4: Constrained network 1	2266	1132	94.0%
Net-5: Constrained network 2	5194	1060	98.4%

Results



The networks all have sigmoidal output units, and were all fit with the sum-of-squares error function.