Lecture 18: Neural Networks

Reading: Chapter 11

GU4241/GR5241 Statistical Machine Learning

Linxi Liu

March 28, 2017

Overview

- A neural network is a supervised learning method. It can be applied to both regression and classification problems.
- The central idea is to extract linear combinations of the inputs as derived features, and then model the target as a nonlinear function of these features.
- The nonlinear transformation contributes to the model flexibility.
- We will focus on the most widely used "vanilla" neural net, also called the single hidden layer feedforward neural networks.

General Description

- Derived features Z_m are obtained by applying the *activation function* σ to linear combinations of the inputs:

α(m) = (αm1…a(mn))^T

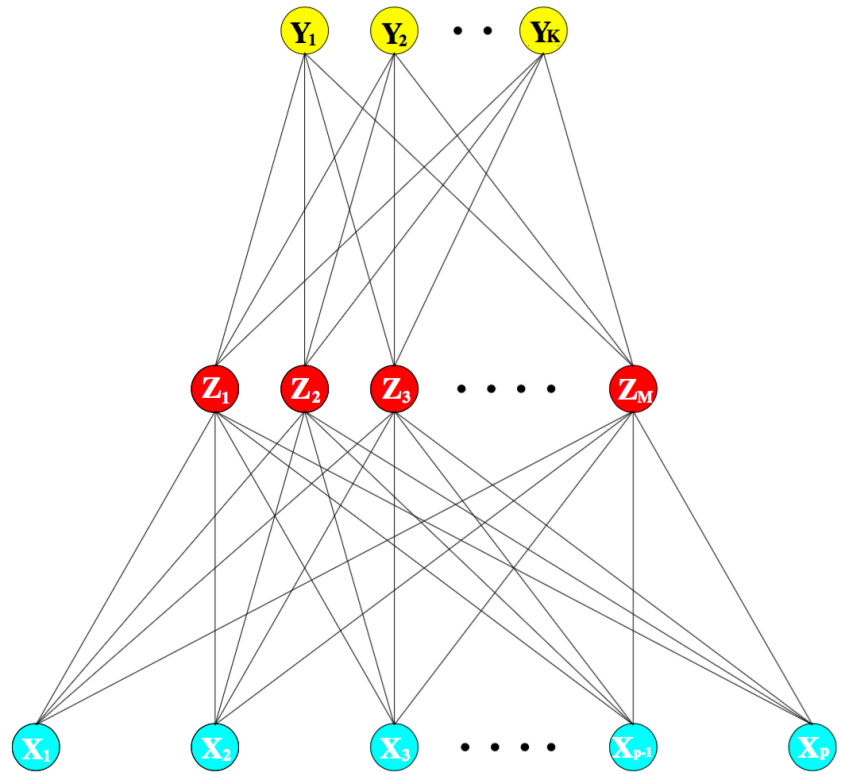
$$Z_m = \sigma(\alpha_{0m} + \alpha_m^T X), m = 1, \ldots, M.$$

- The target Y_k (or T_k in the figure) is modeled as a function of linear combinations of the Z_m:

$$T_k = \beta_{0k} + \beta_k^T Z, \quad k = 1, \ldots, K.$$

- The output function $g_k(T)$ allows a final transformation of the vector of outputs T:

$$f_k(X) = g_k(T), \quad k = 1, \ldots, K.$$



Schematic of a single hidden layer, feed-forward neural network

General Description

- Derived features Z_m are obtained by applying the *activation function* σ to linear combinations of the inputs:

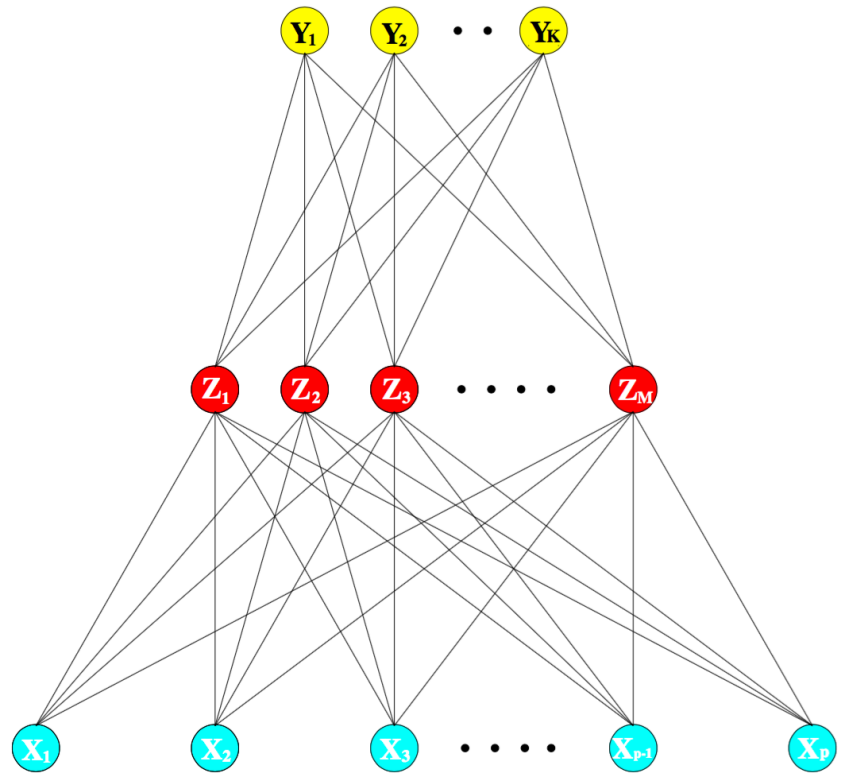
$$Z_m = \sigma(\alpha_{0m} + \alpha_m^T X), m = 1, \ldots, M.$$

- The target Y_k (or T_k in the figure) is modeled as a function of linear combinations of the Z_m:

$$T_k = \beta_{0k} + \beta_k^T Z, \quad k = 1, \ldots, K.$$

- For regression, we typically choose the identity function

$$g_k(T) = T_k.$$



Schematic of a single hidden layer, feed-forward neural network

General Description

- Derived features Z_m are obtained by applying the *activation function* σ to linear combinations of the inputs:

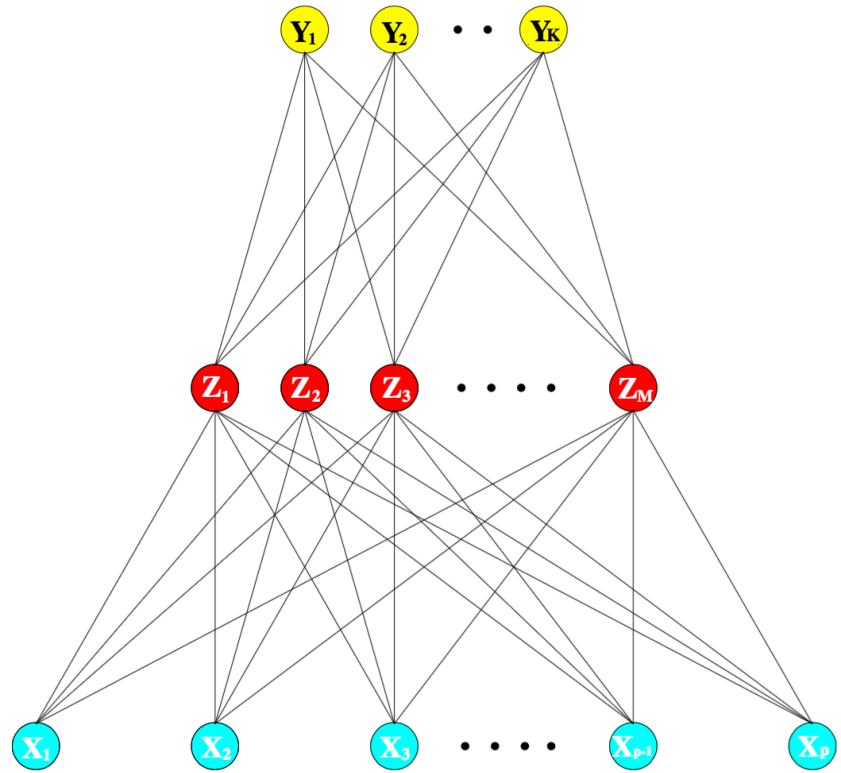
$$Z_m = \sigma(\alpha_{0m} + \alpha_m^T X), m = 1, \ldots, M.$$

- The target Y_k (or T_k in the figure) is modeled as a function of linear combinations of the Z_m:

$$T_k = \beta_{0k} + \beta_k^T Z, \quad k = 1, \ldots, K.$$

- For K-class classification, we use the *softmax* function

$$g_k(T) = \frac{e^{T_k}}{\sum_{l=1}^{K} e^{T_l}}$$



Schematic of a single hidden layer, feed-forward neural network

The activation function

- The activation function σ is usually chosen to be the *sigmoid* $\sigma(v) = 1/(1 + e^{-v})$.
- Notice that is σ is the identity function, then the entire modle collapses to a linear model in the inputs.
- The rate of activation of the sigmoid depends on the norm of α_m.
- We can also choose other σ, like Gaussian radial basis functions.

we use sigmoid function as non-linear transformation function;
it relates to the step function: if the input signod is very small, after the non-linear transformation,the result(output) will be 0, inactive; if the signoid is very strong, then the output will be 1, the particular neural will be active

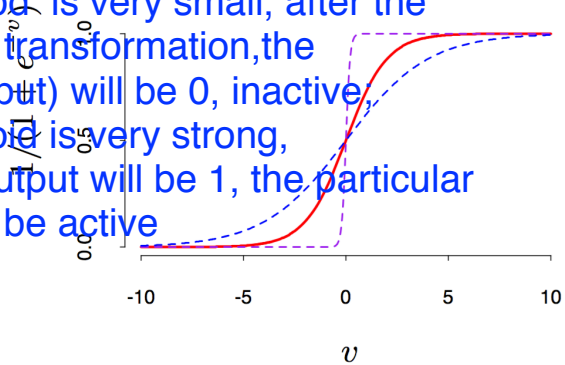


FIGURE 11.3. *Plot of the sigmoid function* $\sigma(v) = 1/(1 + \exp(-v))$ *(red curve), commonly used in the hidden layer of a neural network. Included are* $\sigma(sv)$ *for* $s = \frac{1}{2}$ *(blue curve) and* $s = 10$ *(purple curve). The scale parameter* s *controls the activation rate, and we can see that large* s *amounts to a hard activation at* $v = 0$*. Note that* $\sigma(s(v - v_0))$ *shifts the activation threshold from* 0 *to* v_0*.*

Fitting Neural Networks

Recall our model is:

$$\begin{aligned} Z_m &= \sigma(\alpha_{0m} + \alpha_m^T X), \quad m = 1, \ldots, M. \\ T_k &= \beta_{0k} + \beta_k^T Z, \quad k = 1, \ldots, K. \\ f_k(X) &= g_k(T), \quad k = 1, \ldots, K. \end{aligned}$$

The unknow parameters of the model are often called *weights*. We denote the complete set of weights by θ, which consists of

first layer $\{\alpha_{0m}, \alpha_m;\ m = 1, 2, \ldots, M\}$ $M(p+1)$ weights,

second layer $\{\beta_{0k}, \beta_k;\ k = 1, 2, \ldots, K\}$ $K(M+1)$ weights.

For regression, we use the squared error loss

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

Fitting Neural Networks

Recall our model is:

$$\begin{aligned} Z_m &= \sigma(\alpha_{0m} + \alpha_m^T X), \quad m = 1, \ldots, M. \\ T_k &= \beta_{0k} + \beta_k^T Z, \quad k = 1, \ldots, K. \\ f_k(X) &= g_k(T), \quad k = 1, \ldots, K. \end{aligned}$$

The unknow parameters of the model are often called *weights*. We denote the complete set of weights by θ, which consists of

$$\begin{aligned} \{\alpha_{0m}, \alpha_m;\ m = 1, 2, \ldots, M\} \quad & M(p+1) \text{ weights}, \\ \{\beta_{0k}, \beta_k;\ k = 1, 2, \ldots, K\} \quad & K(M+1) \text{ weights}. \end{aligned}$$

For classification we use either squared error or corss-entropy

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

and the correponding classifier is $G(x) = \text{argmax}_k f_k(x)$.

Gradient Descent

Assume we use squared error loss. Let $z_{mi} = \sigma(\alpha_{0m} + \alpha_m^T x_i)$ and let $z_i = (z_{1i}, z_{2i}, \ldots, z_{Mi})$. Then we have

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

where

$$f_k(x_i) = g_k(\beta_{0k} + \beta_k^T z_i) = g_k\left(\beta_{0k} + \sum_{m=1}^{M} \beta_{km} \sigma(\alpha_{0m} + \alpha_m^T x_i)\right).$$

The derivatives are

$$\begin{aligned} \frac{\partial R_i}{\partial \beta_{km}} &= -2(y_{ik} - f_k(x_i)) g_k'(\beta_k^T z_i) z_{mi}, \\ \frac{\partial R_i}{\partial \alpha_{ml}} &= -\sum_{k=1}^{K} 2(y_{ik} - f_k(x_i)) g_k'(\beta_k^T z_i) \beta_{km} \sigma'(\alpha_m^T x_i) x_{il}, \end{aligned}$$

Gradient Descent

Assume we use squared error loss. Let $z_{mi} = \sigma(\alpha_{0m} + \alpha_m^T x_i)$ and let $z_i = (z_{1i}, z_{2i}, \ldots, z_{Mi})$. Then we have

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

where

$$f_k(x_i) = g_k(\beta_{0k} + \beta_k^T z_i) = g_k\left(\beta_{0k} + \sum_{m=1}^{M} \beta_{km} \sigma(\alpha_{0m} + \alpha_m^T x_i)\right).$$

A gradient update at the $(r+1)$st iteration has the form

$$\begin{aligned} \beta_{km}^{(r+1)} &= \beta_{km}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \beta_{km}^{(r)}}, \\ \alpha_{ml}^{(r+1)} &= \alpha_{ml}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \alpha_{ml}^{(r)}}. \end{aligned}$$

Back-propagation

If we write the gradients as

$$\frac{\partial R_i}{\partial \beta_{km}} = -2(y_{ik} - f_k(x_i))g_k'(\beta_k^T z_i)z_{mi},$$

$$\frac{\partial R_i}{\partial \alpha_{ml}} = -\sum_{k=1}^{K} 2(y_{ik} - f_k(x_i))g_k'(\beta_k^T z_i)\beta_{km}\sigma'(\alpha_m^T x_i)x_{il}. \quad .$$

Back-propagation

If we write the gradients as

$$\begin{aligned} \frac{\partial R_i}{\partial \beta_{km}} &= \delta_{ki} z_{mi}, \\ \frac{\partial R_i}{\partial \alpha_{ml}} &= s_{mi} x_{il}. \end{aligned}$$

Back-propagation

If we write the gradients as

$$\frac{\partial R_i}{\partial \beta_{km}} = \delta_{ki} z_{mi},$$
$$\frac{\partial R_i}{\partial \alpha_{ml}} = s_{mi} x_{il}.$$

In some sense, δ_{ki} and s_{mi} are "errors" at the output and hidden layer units. The errors satisfy

$$s_{mi} = \sigma'(\alpha_m^T x_i) \sum_{k=1}^{K} \beta_{km} \delta_{ki}.$$

They are called the *back-propagation equations*. The updates can be implemented with a two-pass algorithm:

- *forward pass*: fix weights, compute the predicted values $\hat{f}_k(x_i)$.
- *backward pass*: errors δ_{ki} are computed, and back-propagated to give the errors s_{mi}. Then use both sets of errors to compute the gradients.

based on the forward prediction, calculate the error; and then use the backward pass, calculate simulation error

Alternative Algorithm

A gradient update at the $(r+1)$st iteration has the form

$$\begin{aligned}
\beta_{km}^{(r+1)} &= \beta_{km}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \beta_{km}^{(r)}}, \\
\alpha_{ml}^{(r+1)} &= \alpha_{ml}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \alpha_{ml}^{(r)}}.
\end{aligned}$$

This algorithm is a kind of *batch learning*.

We compute the gradients as a sum over all the training cases.

We can use an alternative algorithm in which the learning is carried out online.

Starting Values

- If the weights are near zero, then the operative part of the sigmoid is roughly zero.
- Usually starting values for weights are chosen to be random values near zero.
- Hence the model starts out nearly linear, and becomes nonlinear as the weights increases.

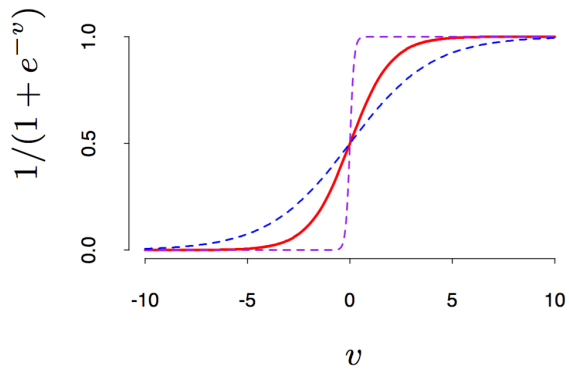


FIGURE 11.3. *Plot of the sigmoid function* $\sigma(v) = 1/(1+\exp(-v))$ *(red curve), commonly used in the hidden layer of a neural network. Included are* $\sigma(sv)$ *for* $s = \frac{1}{2}$ *(blue curve) and* $s = 10$ *(purple curve). The scale parameter* s *controls the activation rate, and we can see that large* s *amounts to a hard activation at* $v = 0$*. Note that* $\sigma(s(v - v_0))$ *shifts the activation threshold from* 0 *to* v_0*.*

Starting instead with large weights often leads to poor solutions.

Use of exact zero weights leads to zero derivatives
and perfect symmetry, and the algorithm never moves

Multiple Minima

One must at least try a number of random starting configurations, and choose the solution giving lowest (penalized) error.

The error function $R(\theta)$ is nonconvex, possessing many local minima.

The solution we obtained from back-propagation is a local minimum.

Usually, we try a number of random starting configuration, and choose the solution giving lowest error, or use the average predictions over the collection of networks as the final prediction.

This is preferable to averaging the weights, since the nonlinearity of the model implies that this averaged solution could be quite poor. Another approach is via bagging, which aver- ages the predictions of networks training from randomly perturbed versions of the training data.

Regularization

avoid over-fitting

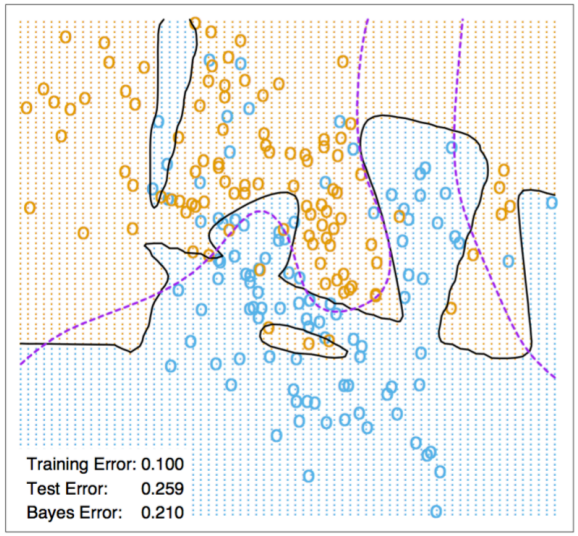
theta is the weight

- Often neural networks have too many weights and will overfit the data at the global minimum of R.
- A regularization method is *weight decay*. We add a penalty to the error function $R(\theta) + \lambda J(\theta)$, where

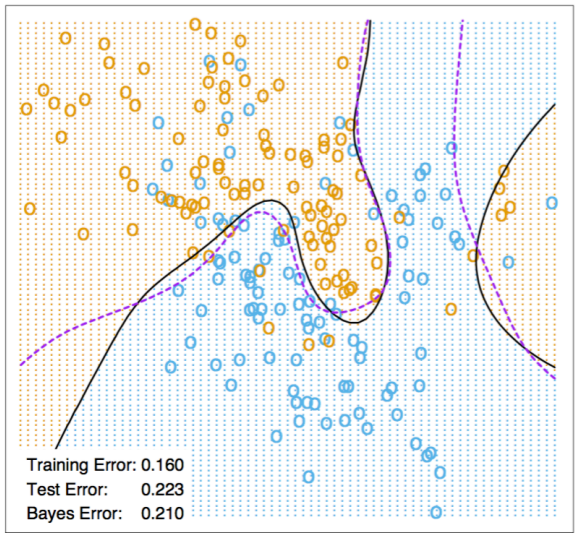
$$J(\theta) = \sum_{k,m} \beta_{km}^2 + \sum_{m,l} \alpha_{ml}^2.$$

- $\lambda \geq 0$ is a tuning parameter, can be chosen by cross-validation.

Neural Network - 10 Units, No Weight Decay



Neural Network - 10 Units, Weight Decay=0.02



Example: Simulated Data

We generate data from two additive error models $Y = f(X) + \epsilon$:

$$\text{Sum of sigmoids: } Y = \sigma(a_1^T X) + \sigma(a_2^T X)\epsilon_1;$$

$$\text{Radial: } Y \overset{\text{Text}}{=} \prod_{m=1}^{10} \phi(X_m) + \epsilon_2.$$

Here $X^T = (X_1, X_2, \ldots, X_p)$, each X_j being a standard Gaussian variate, with $p = 2$ in the first model, and $p = 10$ in the second.

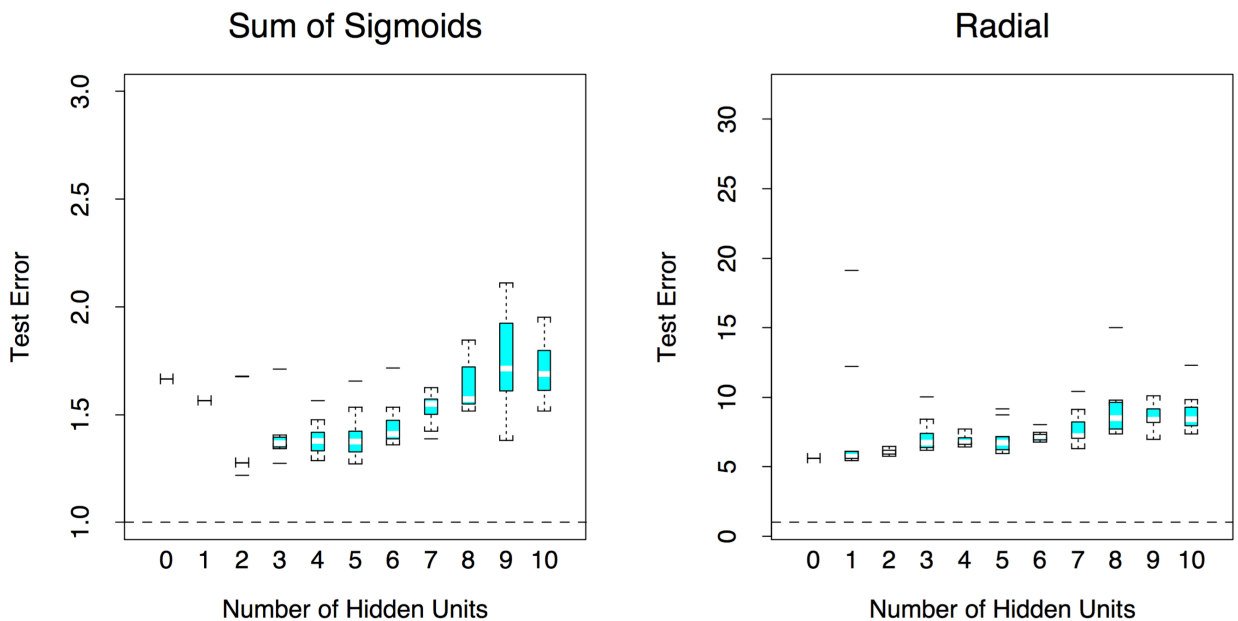
Example: Simulated Data

We generate data from two additive error models $Y = f(X) + \epsilon$:

$$\text{Sum of sigmoids: } Y = \sigma(a_1^T X) + \sigma(a_2^T X)\epsilon_1;$$

$$\text{Radial: } Y = \prod_{m=1}^{10} \phi(X_m) + \epsilon_2.$$

Here $X^T = (X_1, X_2, \ldots, X_p)$, each X_j being a standard Gaussian variate, with $p = 2$ in the first model, and $p = 10$ in the second.



Boxplots of test error

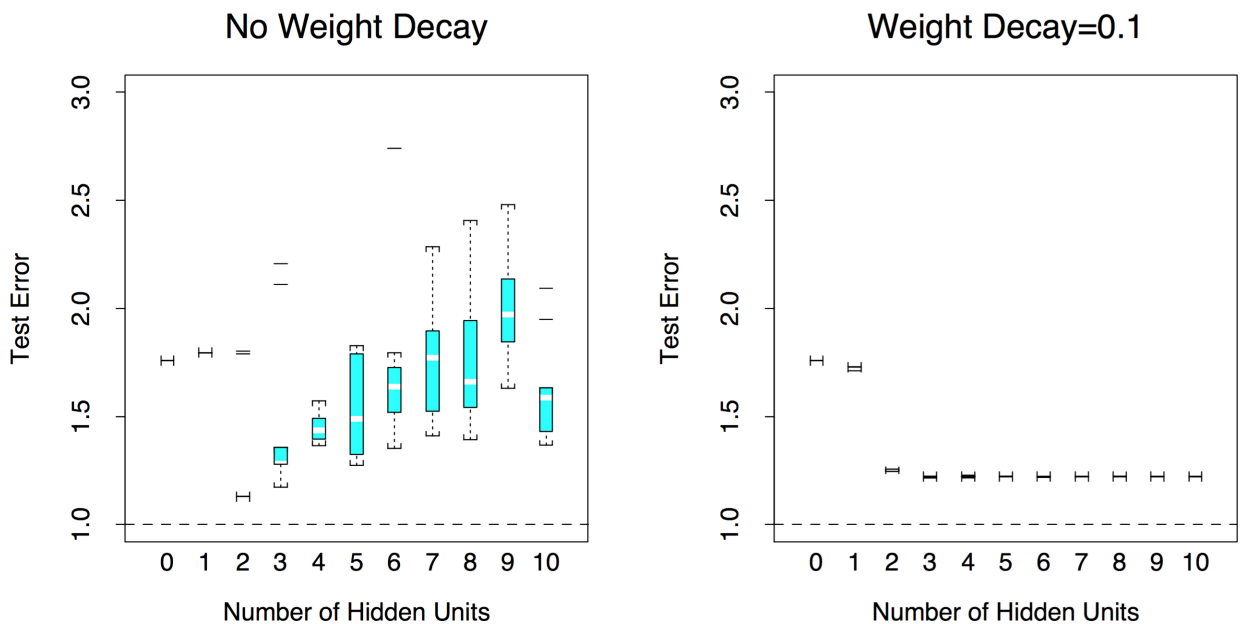
Example: Simulated Data

We generate data from two additive error models $Y = f(X) + \epsilon$:

$$\text{Sum of sigmoids: } Y = \sigma(a_1^T X) + \sigma(a_2^T X)\epsilon_1;$$

$$\text{Radial: } Y = \prod_{m=1}^{10} \phi(X_m) + \epsilon_2.$$

Here $X^T = (X_1, X_2, \ldots, X_p)$, each X_j being a standard Gaussian variate, with $p = 2$ in the first model, and $p = 10$ in the second.



Regularization

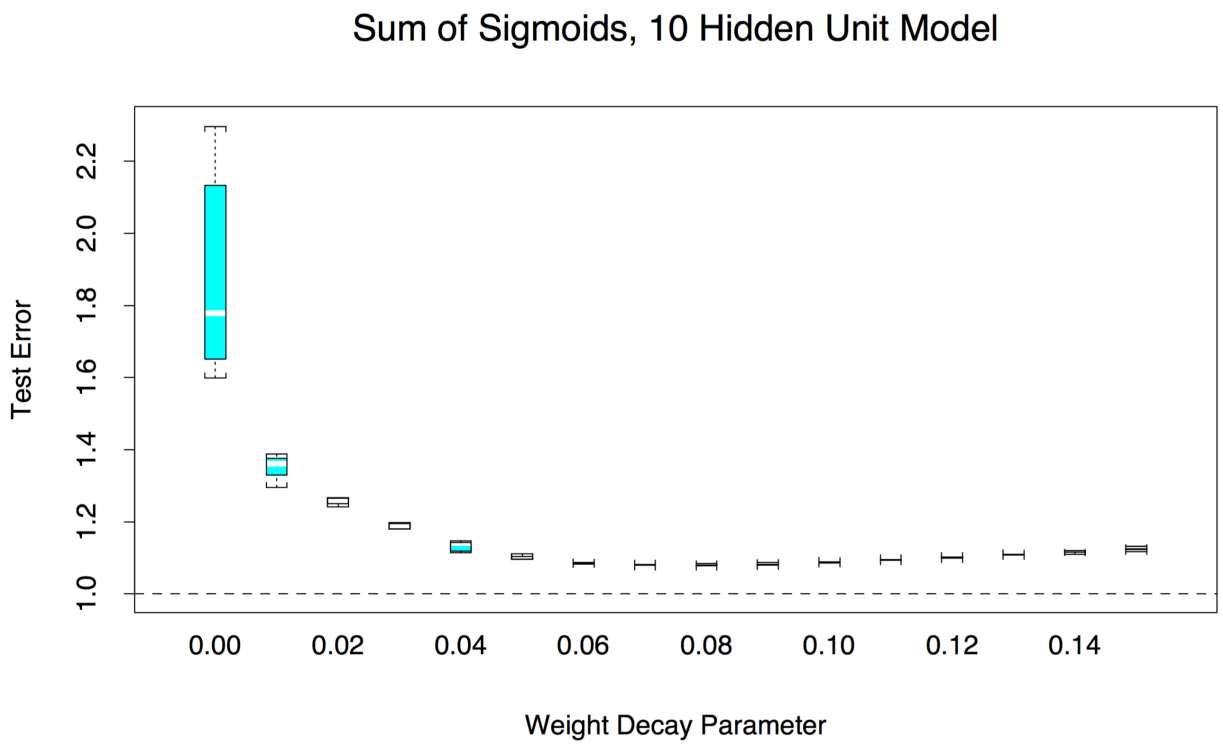
Example: Simulated Data

We generate data from two additive error models $Y = f(X) + \epsilon$:

$$\text{Sum of sigmoids: } Y = \sigma(a_1^T X) + \sigma(a_2^T X)\epsilon_1;$$

$$\text{Radial: } Y = \prod_{m=1}^{10} \phi(X_m) + \epsilon_2.$$

Here $X^T = (X_1, X_2, \ldots, X_p)$, each X_j being a standard Gaussian variate, with $p = 2$ in the first model, and $p = 10$ in the second.

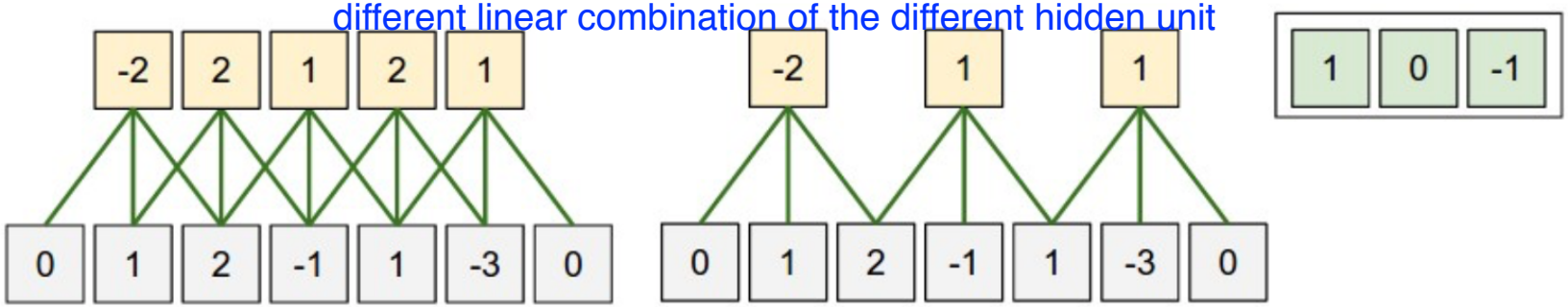


Convolutional Neural Networks: Sharing the Weights

it been used as image analysis

- Convolutional Neural Networks (CNN) have been widely used in image analysis.
- They are similar to the neural networks we discussed before. The difference is that they force the derived features for different hidden units to be computed by the *same* linear functional, or in other words, the hidden units *share* the weights.

for different hidden unit, covolutional nueral networks share the same α, the weight of the linear combination of the different hidden unit;
however, for the normal nueral work, α is different for different linear combination of the different hidden unit

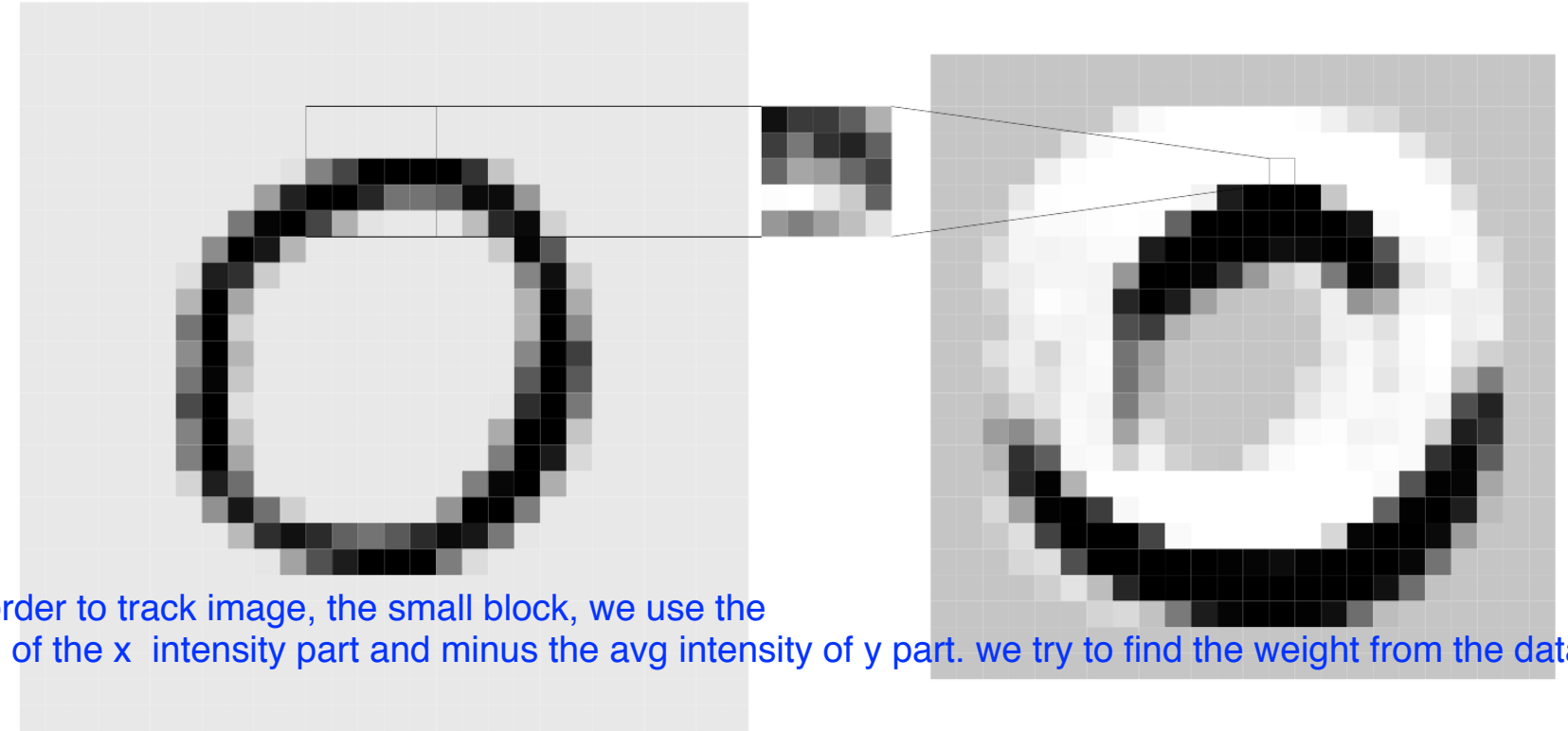


The weights are (1, 0, -1) (shown on top right), and the bias is zero. These weights are shared across all yellow neurons.

use the same weights<1,0,-1> to apply dot product with the inout data, to calculate the different hiddent unit

Convolutional Neural Networks: Sharing the Weights

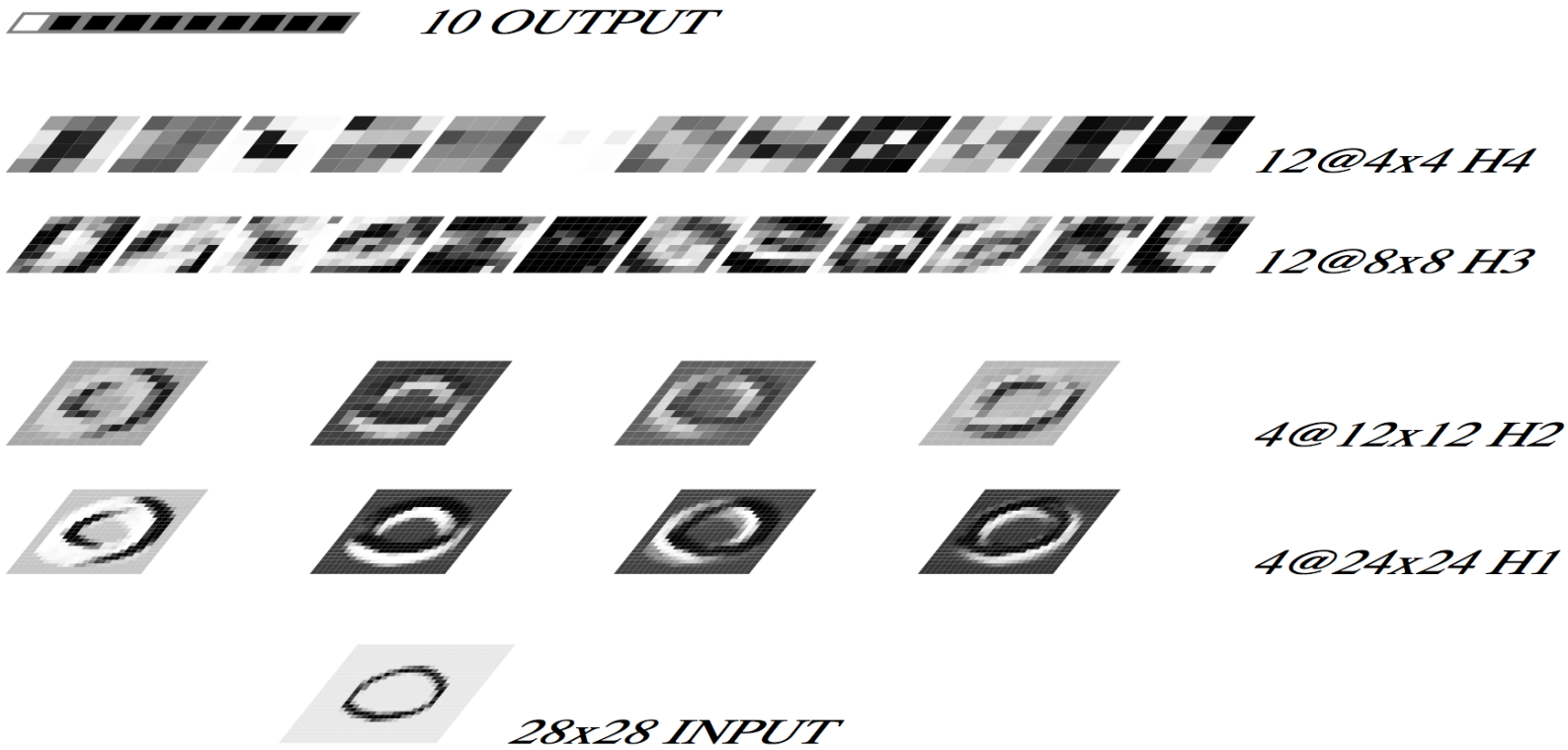
use small block of the weights, and apply these block
with the 5*5 pictals, then calculate the combination



in order to track image, the small block, we use the
avg of the x intensity part and minus the avg intensity of y part. we try to find the weight from the datath

Input image (left), weight vector, and the resulting feature map (right). White represents corresponds to intensity -1.

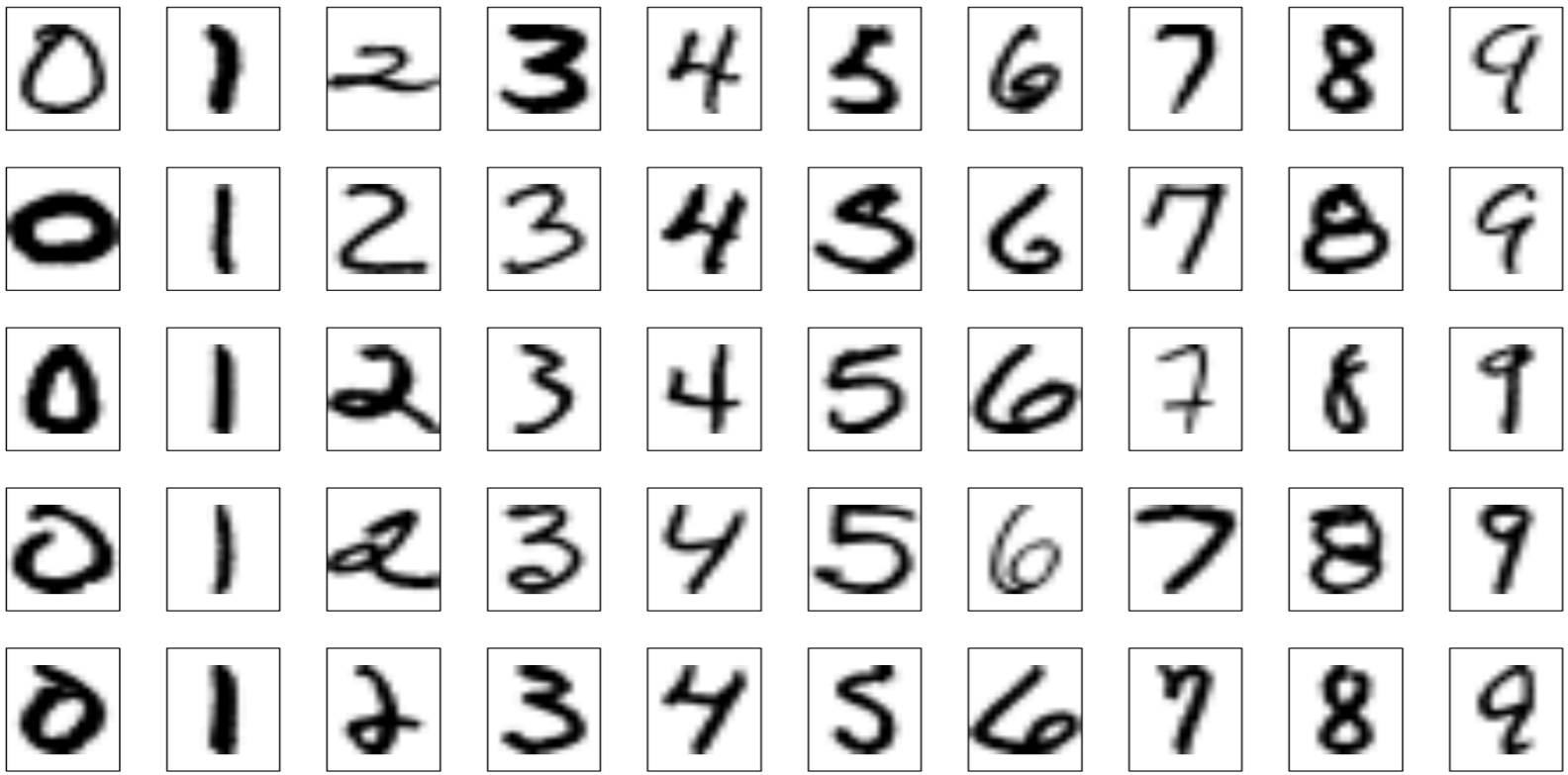
Convolutional Neural Networks



Network Architecture with 5 layers of fully adaptive connections (Le Cun, 1989).

the first block is samll, 3*3, and extract the local features
then pooling(from LDA) and convolutional again
then each unit connect with 4 *4 figures
we repeat with higher features of the local features,
it becomes as global features

Example: ZIP Code Data



Example: ZIP Code Data

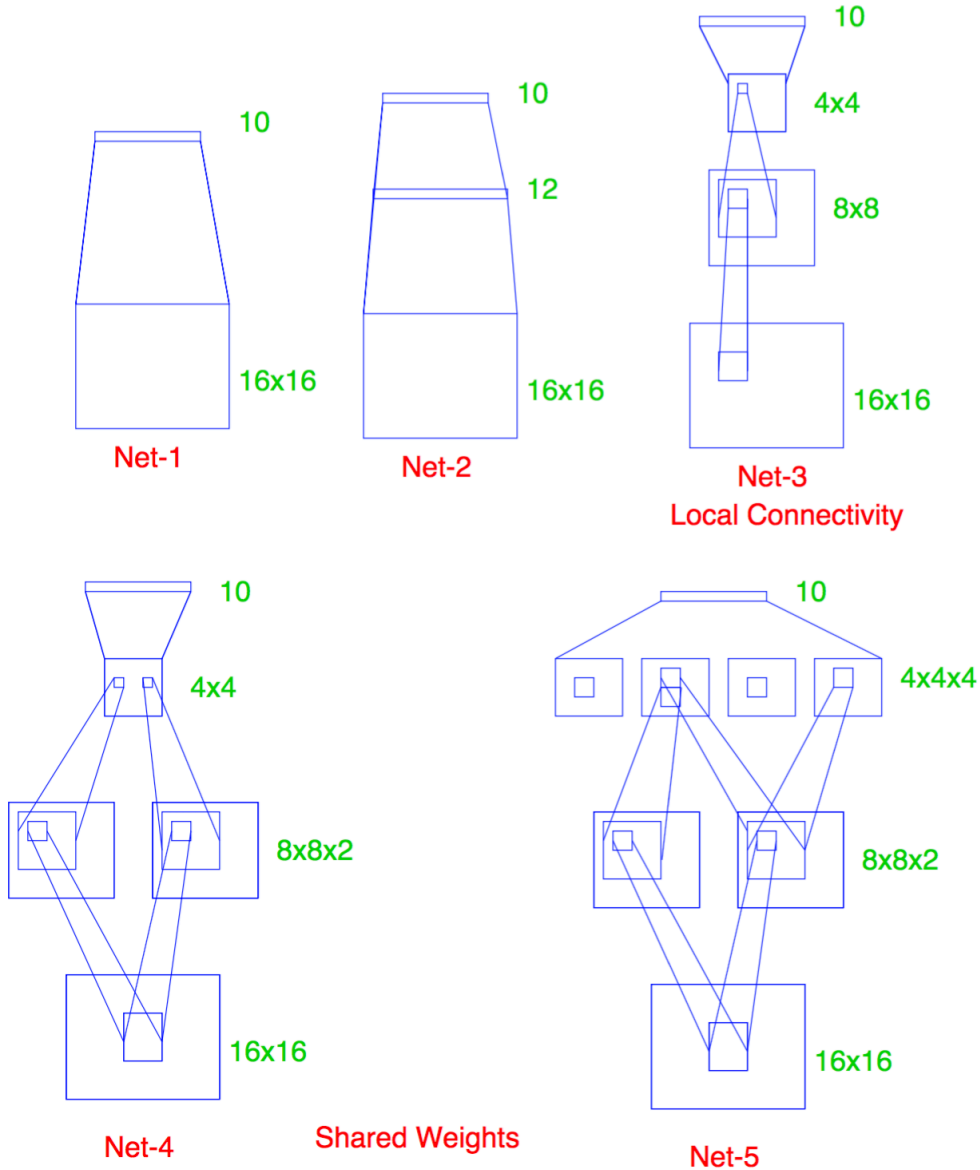


FIGURE 11.10. *Architecture of the five networks used in the ZIP code example.*

Example: ZIP Code Data