Computational statistics

Lecture 3: Neural networks

Thierry Denœux

5 March, 2016

Neural networks

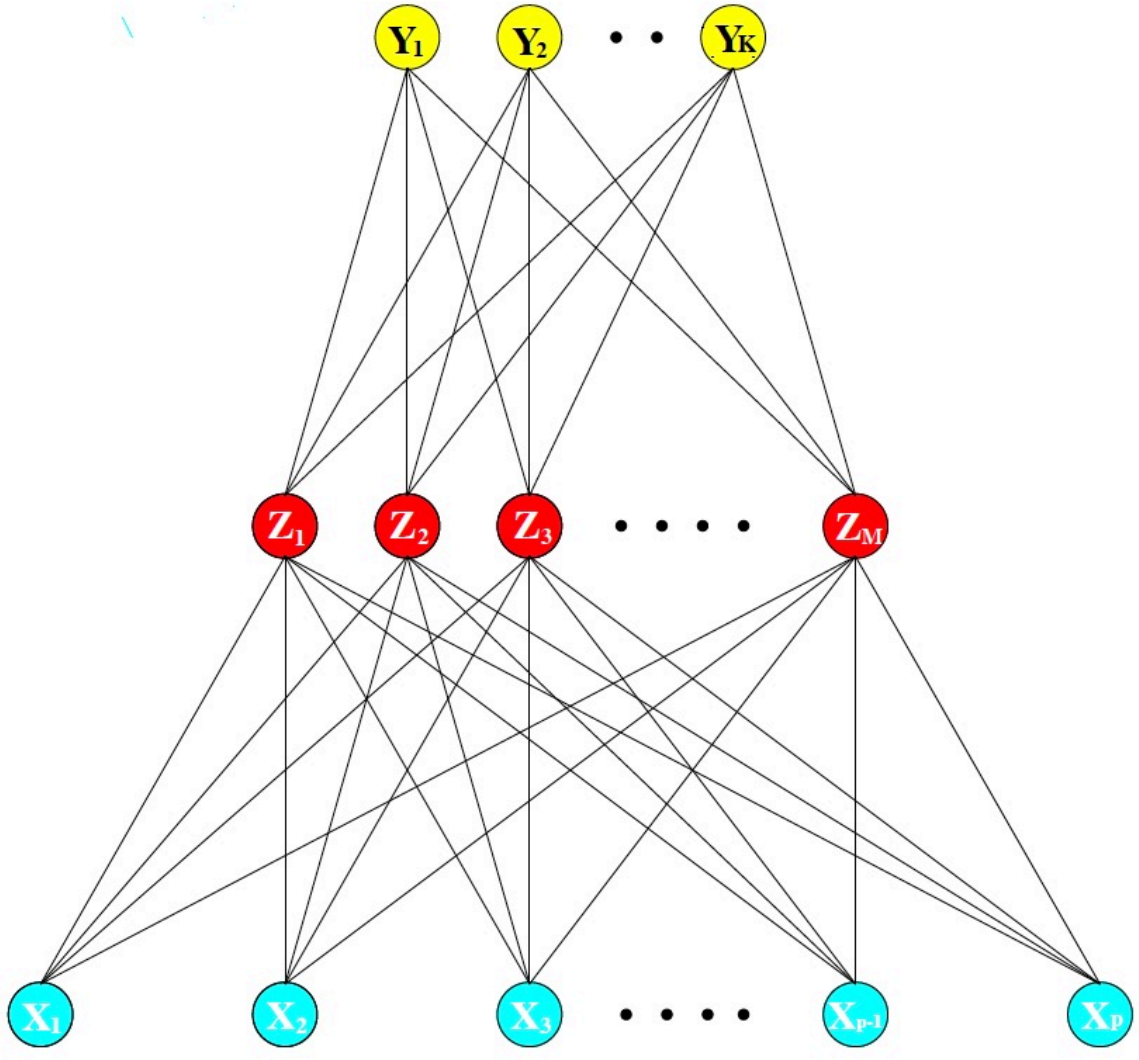
- A class of learning methods that was developed separately in different fields – statistics and artificial intelligence – based on essentially identical models
- The central idea is to extract linear combinations of the inputs as derived features, and then model the target as a nonlinear function of these features
- The result is a powerful learning method, with widespread applications in many fields
- There exist many neural network models. Here we describe the most widely used "vanilla" neural net, sometimes called the single hidden layer back-propagation network, or single layer perceptron

Overview

Multi-layer architecture

- A neural network is a two-stage regression or classification model, typically represented by a network diagram as in the next slide
- It is composed of three layers: input , hidden and output
- This network applies both to regression or classification
  - For regression, typically $K = 1$ and there is only one output unit Y_1 at the top. However, these networks can handle multiple quantitative responses in a seamless fashion, so we will deal with the general case
  - For K-class classification, there are K units at the top, with the kth unit modeling the probability of class k. There are K target values Y_k, $k = 1, \ldots, K$, each being coded as a $0 - 1$ variable for the kth class

Graphical representation



Propagation equations

- Derived features Z_m are created from linear combinations of the inputs, and then the target Y_k is modeled as a function of linear combinations of the Z_m:

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

  where $Z = (Z_1, Z_2, \ldots, Z_M)$, and $T = (T_1, T_2, \ldots, T_K)$

- Terminology
  - α_{mj}: connection weight between input unit j and hidden unit m
  - β_{km}: connection weight between hidden unit m and output unit k
  - α_{0m}, β_{0k}: bias terms
  - σ: hidden unit activation function
  - g_k: output activation functions

Activation function

- The activation function $\sigma(v)$ is usually chosen to be the sigmoid

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

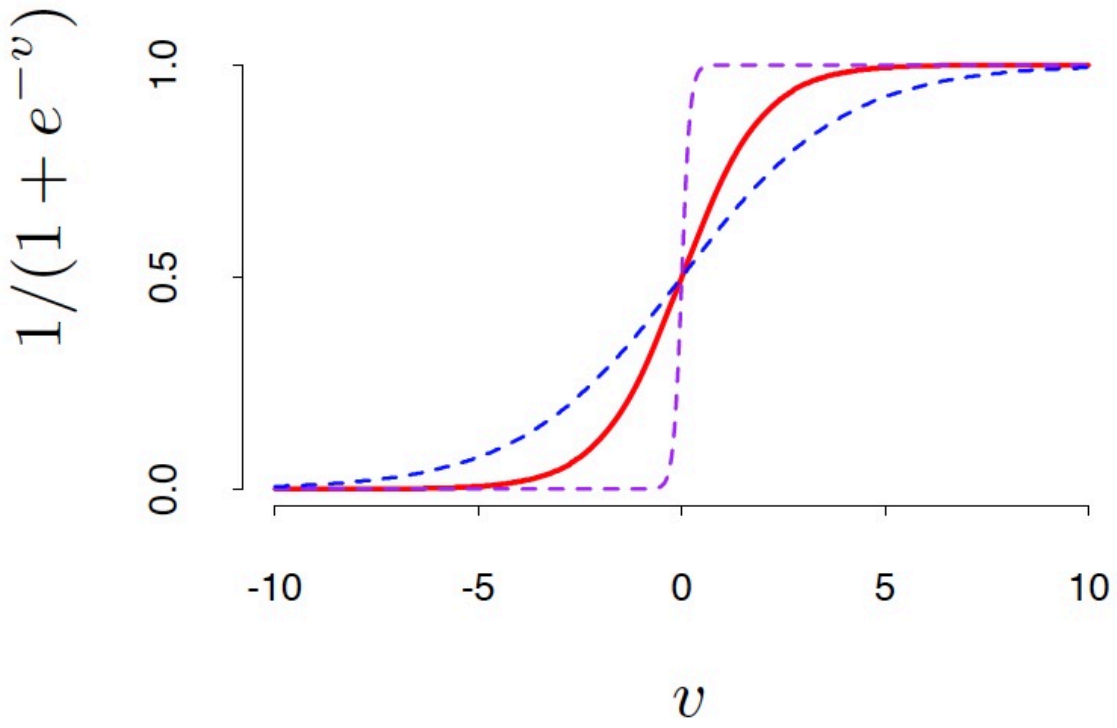
(see next slide)

- Variant: sometimes the output of hidden unit m is defined using a Gaussian radial basis function as

$$Z_m = \exp(-\gamma_m \|X - \alpha_m\|^2)$$

We then have a radial basis function network

Sigmoid activation function



Plot of $\sigma(sv)$ as a function of v for $s = 0.5$ (blue curve), $s = 1$ (red curve) and $s = 10$ (purple curve). The scale parameter s controls the "activation rate"

Output function

- The output function $g_k(T)$ allows a final transformation of the vector of outputs T
- For regression we typically choose the identity function

$$g_k(T) = T_k$$

- Early work in K-class classification also used the identity function, but this was later abandoned in favor of the softmax function

$$g_k(T) = \frac{\exp(T_k)}{\sum_{\ell=1}^{K} \exp(T_\ell)}$$

- This is exactly the transformation used in the multi-class logistic regression model; it produces positive estimates that sum to one

Why "neural" networks?

- The name "neural networks" derives from the fact that they were first developed as models for the human brain
- Each unit represents a neuron, and the connections represent synapses
- In early models, the neurons fired when the total signal passed to that unit exceeded a certain threshold
- In the model above, this corresponds to use of a step function for $\sigma(Z)$ and $g_k(T)$
- Later the neural network was recognized as a useful tool for nonlinear statistical modeling, and for this purpose the step function is not smooth enough for optimization. Hence, the step function was replaced by the smoother sigmoid function

Overview

Learning problem

- The neural network model has unknown parameters, often called weights, and we seek values for them that make the model fit the training data well
- We denote the complete set of weights by θ, which consists of
  - $M(p+1)$ hidden layer weights $\{\alpha_{0m}, \alpha_m; m = 1, 2, \ldots, M\}$
  - $K(M+1)$ output layer weights $\{\beta_{0k}, \beta_k; k = 1, 2, \ldots, K\}$
- To train a neural network, we need a measure of fit, or error function

Error functions

- For regression, we use sum-of-squared errors

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

- For classification we use either squared error or cross-entropy (deviance)

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

  and the corresponding classifier is $G(x) = \arg\max_k f_k(x)$

- With the softmax activation function and the cross-entropy error function, the neural network model is exactly a linear logistic regression model in the hidden units, and all the parameters are estimated by maximum likelihood

Back-propagation algorithm

Principle

- The generic approach to minimizing $R(\theta)$ is by gradient descent, called back-propagation in this setting
- Because of the compositional form of the model, the gradient can be easily derived using the chain rule for differentiation
- This can be computed by a forward and backward sweep over the network, keeping track only of quantities local to each unit

Back-propagation algorithm

Gradient calculation

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

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

Back-propagation algorithm

Weight updates

- Given these derivatives, a gradient descent update at the $t+1$st iteration has the form

$$\beta_{km}^{(t+1)} = \beta_{km}^{(t)} - \eta_t \sum_{i=1}^{n} \frac{\partial R_i}{\partial \beta_{km}^{(t)}}$$

$$\alpha_{mj}^{(t+1)} = \alpha_{mj}^{(t)} - \eta_t \sum_{i=1}^{n} \frac{\partial R_i}{\partial \alpha_{mj}^{(t)}}$$

  where η_t is the learning rate, discussed below

- How to compute the gradient efficiently?

Back-propagation algorithm

Back-propagation equations

- We can write

$$\frac{\partial R_i}{\partial \beta_{km}} = \delta_{ki} z_{mi}, \qquad \frac{\partial R_i}{\partial \alpha_{mj}} = s_{mi} x_{ij}$$

- The quantities δ_{ki} and s_{mi} are "errors" from the current model at the output and hidden layer units, respectively. They satisfy

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

- Using this, the weight updates can be implemented with a two-pass algorithm:
  1. In the forward pass, the current weights are fixed and the predicted values $f_k(x_i)$ are computed
  2. In the backward pass, the errors δ_{ki} are computed, and then back-propagated to give the errors s_{mi}

  Both sets of errors are then used to compute the gradients for the weight updates

Advantages of back-propagation

- The advantages of back-propagation are its simple, local nature
- In the back propagation algorithm, each hidden unit passes and receives information only to and from units that share a connection
- Hence it can be implemented efficiently on a parallel architecture computer

Batch vs. online training

- The previous update equations are a kind of batch learning, with the parameter updates being a sum over all of the training cases
- Learning can also be carried out online – processing each observation one at a time, updating the gradient after each training case, and cycling through the training cases many times
- In this case, the update equations become

$$\beta_{km}^{(t+1)} = \beta_{km}^{(t)} - \eta_t \frac{\partial R_t}{\partial \beta_{km}^{(t)}}$$

$$\alpha_{mj}^{(t+1)} = \alpha_{mj}^{(t)} - \eta_t \frac{\partial R_t}{\partial \alpha_{mj}^{(t)}}$$

- A training epoch refers to one sweep through the entire training set
- Online training allows the network to handle very large training sets, and also to update the weights as new observations come in

Learning rate tuning and optimization algorithms

- The learning rate η_t for batch learning was originally taken to be a constant; it can also be optimized by a line search that minimizes the error function at each update
- With online learning η_t should decrease to zero as the iteration $t \to \infty$ to ensure convergence (e.g., $\eta_t = 1/t$)
- Back-propagation can be very slow, and for that reason is usually not the method of choice
  - Second-order techniques such as Newton's method are not attractive, because the second derivative matrix of R (the Hessian) can be very large
  - Better approaches to fitting include conjugate gradients and variable metric methods, which avoid explicit computation of the second derivative matrix while still providing faster convergence

Overview

The "art" of neural network training

- There is quite an art in training neural networks
- The model is generally overparametrized, and the optimization problem is nonconvex and unstable, unless certain guidelines are followed
- Here, we summarize some of the important issues

Starting Values

- If the weights are near zero, then the operative part of the sigmoid is roughly linear, and hence the neural network collapses into an approximately linear model
- 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 increase
- Use of exact zero weights leads to zero derivatives and perfect symmetry, and the algorithm never moves
- Starting instead with large weights often leads to poor solutions

Overfitting
Early stopping

- Often neural networks have too many weights and will overfit the data at the global minimum of R
- In early developments of neural networks, an early stopping rule was used to avoid overfitting: we train the model only for a while, and stop well before we approach the global minimum
- Since the weights start at a highly regularized (linear) solution, this has the effect of shrinking the final model toward a linear model
- A validation dataset is useful for determining when to stop, since we expect the validation error to start increasing

Overfitting
Weight decay

- A more explicit method for regularization is weight decay, which is analogous to ridge regression used for linear models
- We add a penalty to the error function $R(\theta) + \lambda J(\theta)$, with

$$J(\theta) = \sum_{k,m} \beta_{km}^2 + \sum_{m,j} \alpha_{mj}^2$$

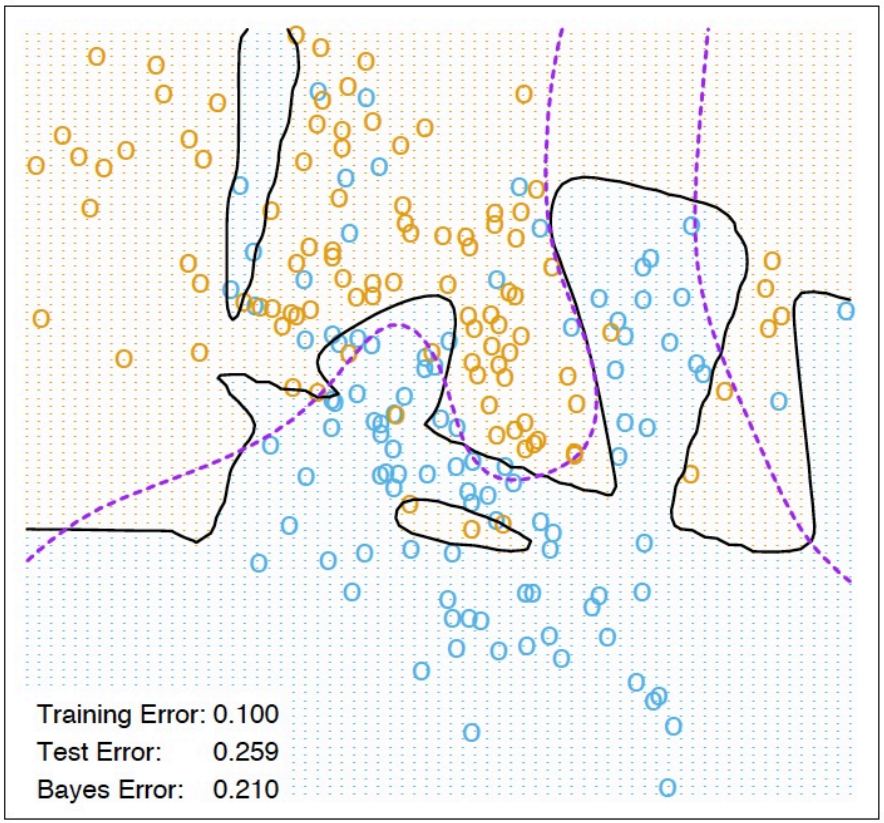
- The hyper-parameter λ is usually determined by cross-validation
- Other forms for the penalty have been proposed, for example,

$$J(\theta) = \sum_{k,m} \frac{\beta_{km}^2}{1+\beta_{km}^2} + \sum_{m,j} \frac{\alpha_{mj}^2}{1+\alpha_{mj}^2}$$

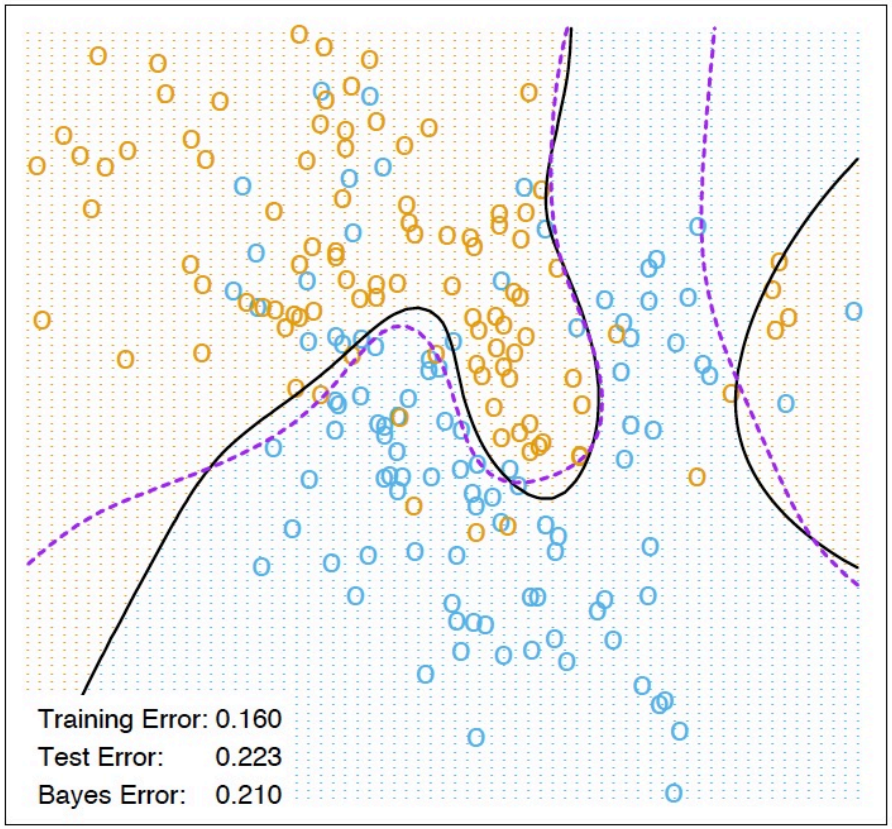
known as the weight elimination penalty. This has the effect of shrinking smaller weights more than weight decay does

Example

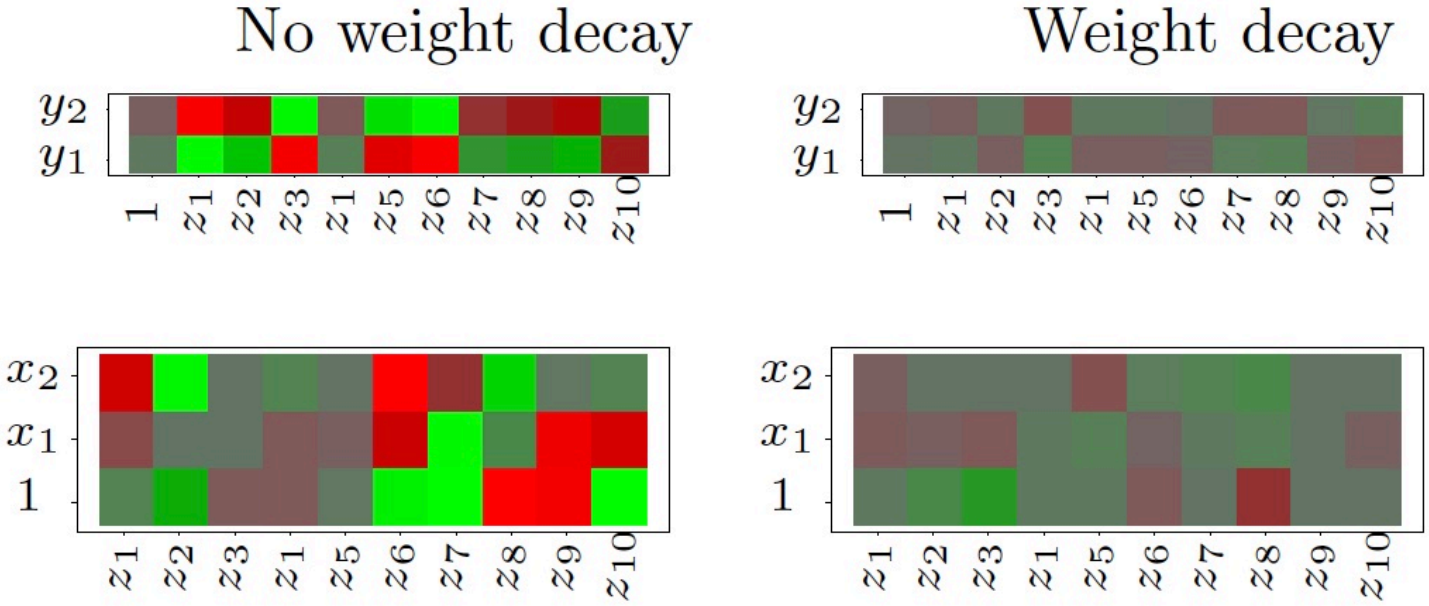
Neural Network - 10 Units, No Weight Decay



Neural Network - 10 Units, Weight Decay=0.02



Heat maps of estimated weights



Scaling of the Inputs

- Since the scaling of the inputs determines the effective scaling of the weights in the bottom layer, it can have a large effect on the quality of the final solution
- It is best to standardize all inputs to have mean zero and standard deviation one
- This ensures all inputs are treated equally in the regularization process, and allows one to choose a meaningful range for the random starting weights
- With standardized inputs, it is typical to take random uniform weights over the range $[-0.7, +0.7]$

Overview

Simulated data

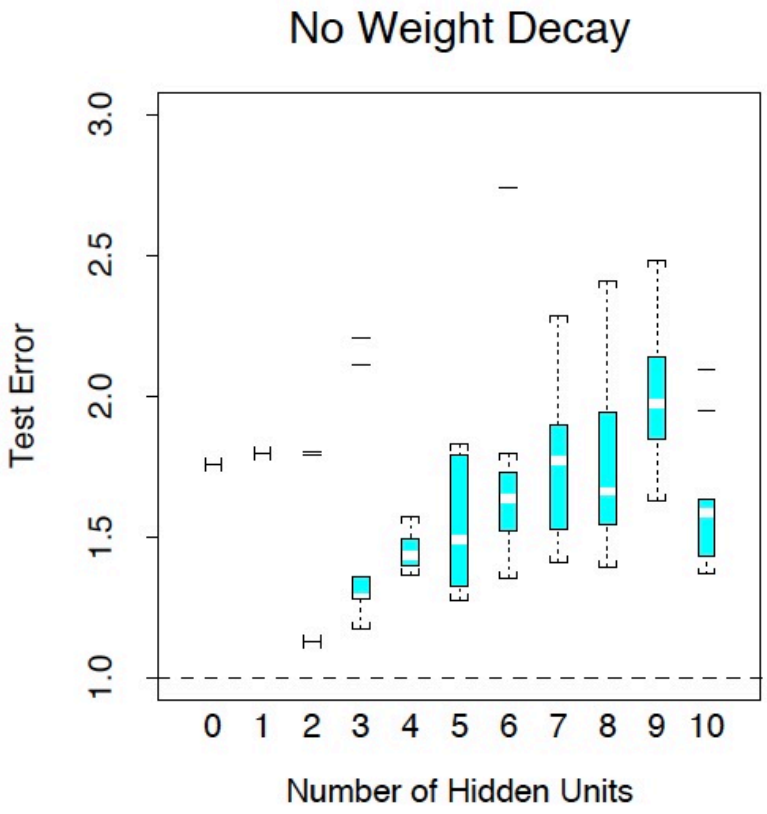
- Model $Y = f(X) + \varepsilon$ with

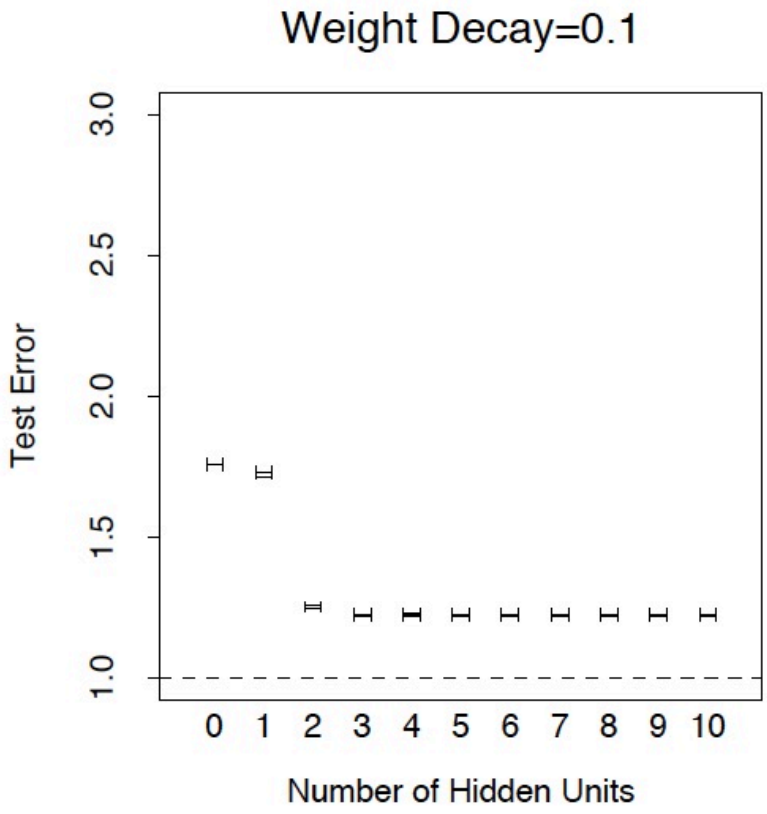
$$f(X) = \sigma(a_1^T X) + \sigma(a_2^T X),$$

$X = (X_1, X_2)$, $a_1 = (3, 3)$, $a_2 = (3, -3)$, $\mathrm{Var}(f(X))/\mathrm{Var}(\varepsilon) = 4$

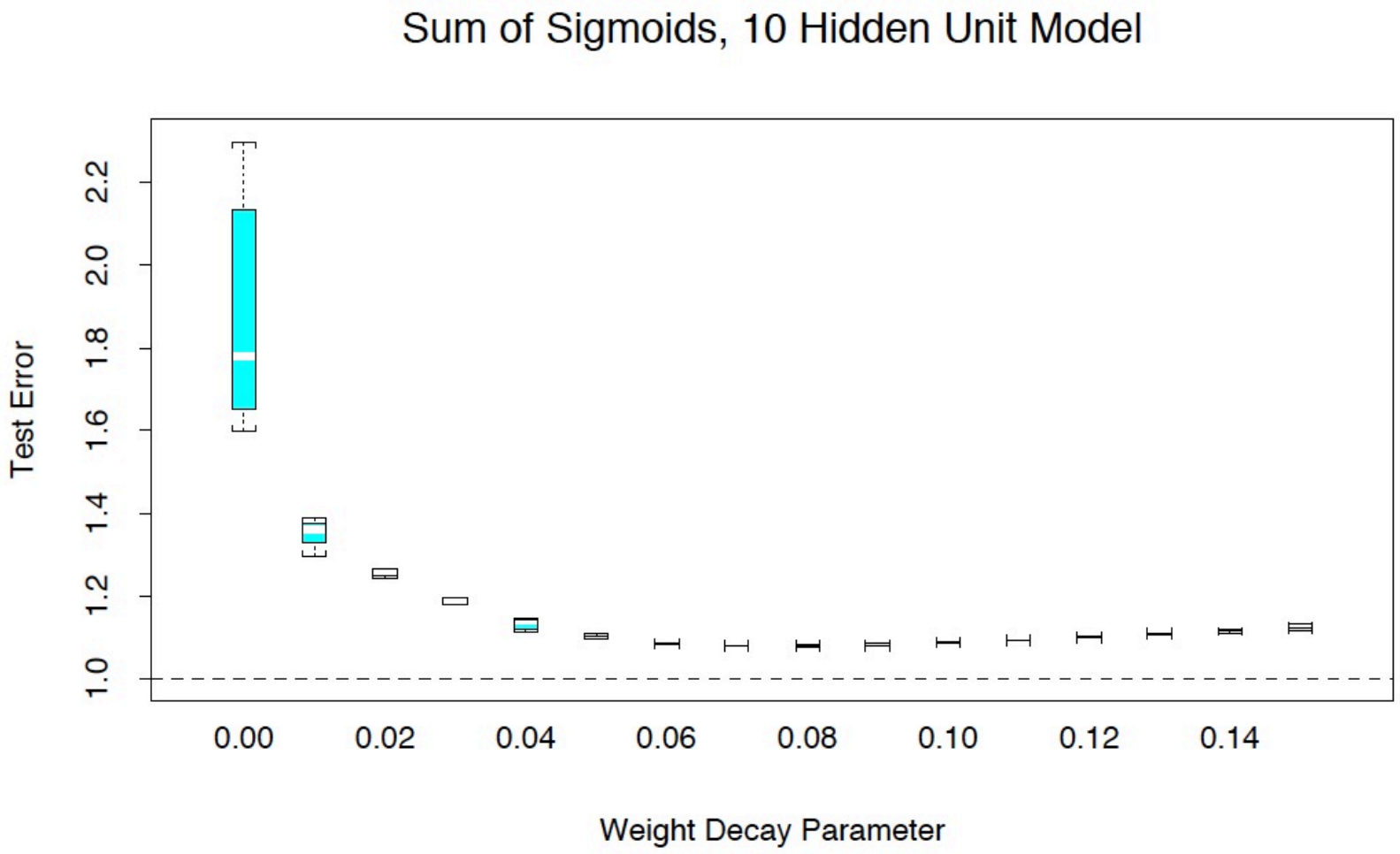
- Training sample of size 100, a test sample of size 10,000
- Neural networks with weight decay and various numbers of hidden units
- Average test error for each of 10 random starting weights

Results without and with weight decay





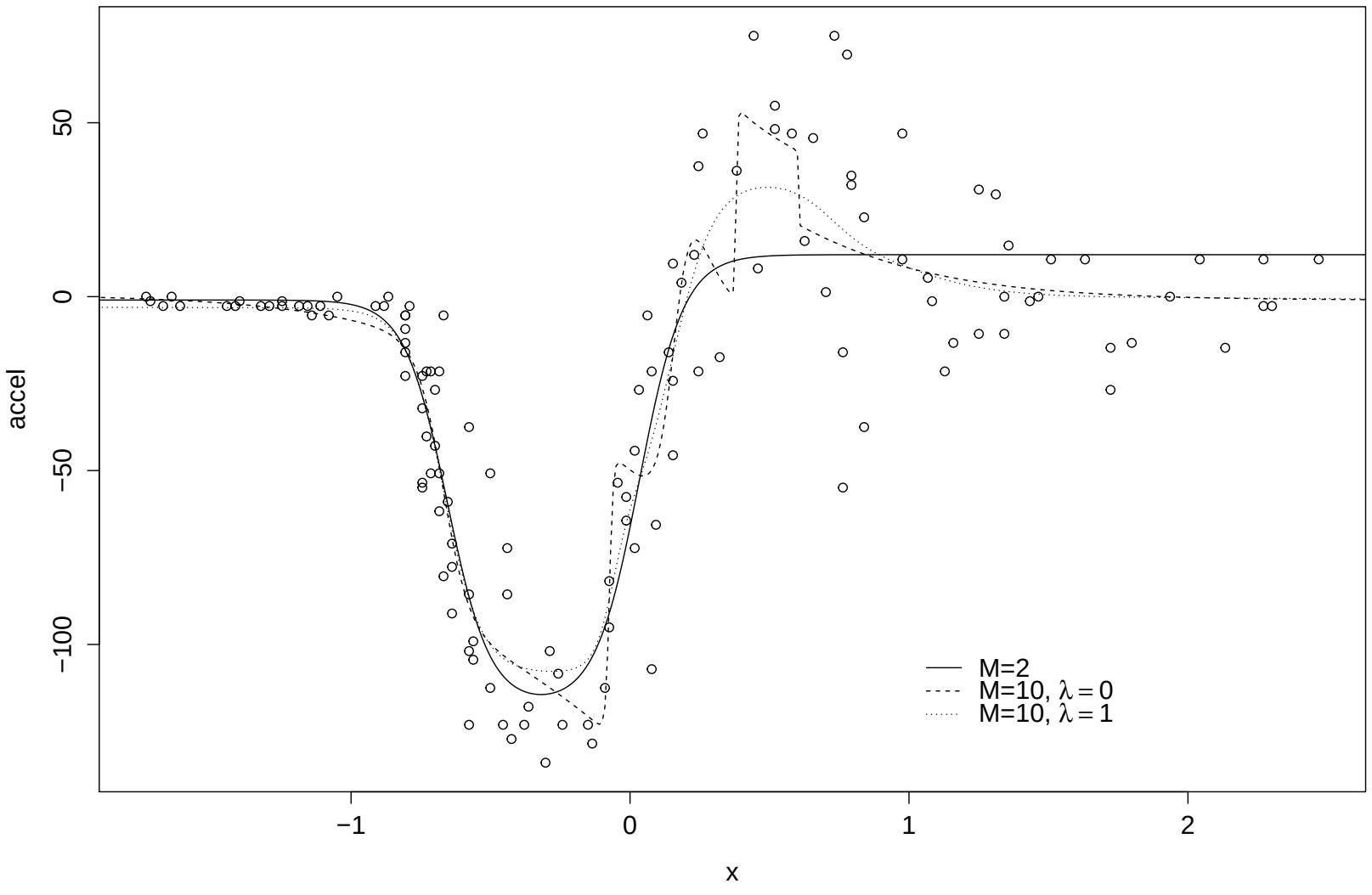
Influence of the weight decay hyper-parameter



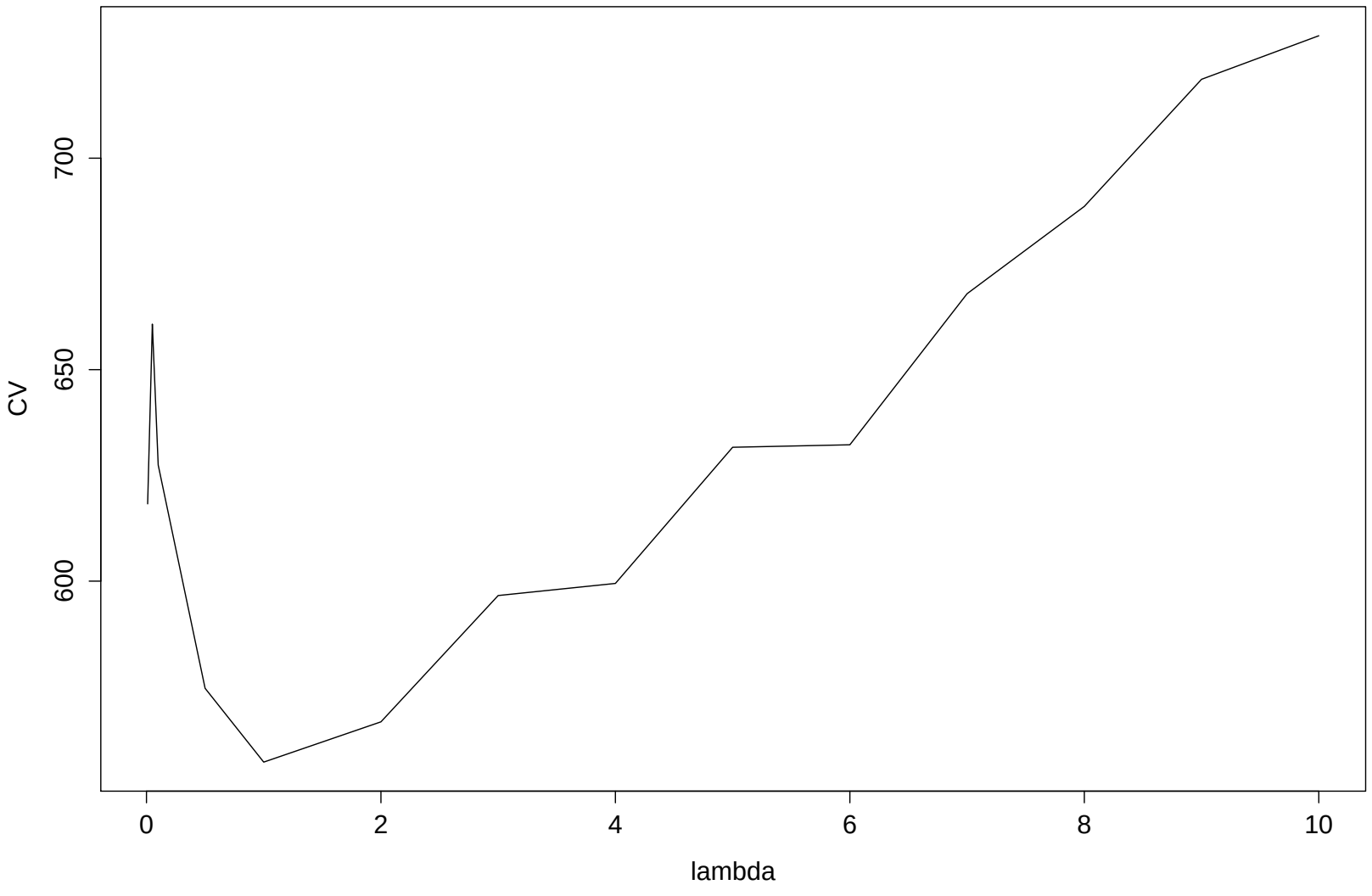
Example with the nnet package

```
library('MASS')
mcycle.data<-data.frame(mcycle,x=scale(mcycle$times))
test.data<-data.frame(x=seq(-2,3,0.01))

library('nnet')

nn1<- nnet(accel ~ x, data=mcycle.data, size=2, linout = TRUE, decay=0)
pred1<- predict(nn1,newdata=test.data)

nn2<- nnet(accel ~ x, data=mcycle.data, size=10, linout = TRUE, decay=0)
pred2<- predict(nn2,newdata=test.data)

nn3<- nnet(accel ~ x, data=mcycle.data, size=10, linout = TRUE, decay=1)
pred3<- predict(nn3,newdata=test.data)
```

Results



Selection of λ by 10-fold cross-validation



Conclusions

- Powerful and very general approach for regression and classification
- Training a neural network implies solving a non-convex optimization problem with a large number of parameter: computer-intensive approach
- Neural networks are especially effective in settings where prediction without interpretation is the goal
- They are less effective for problems where the goal is to describe the physical process that generated the data and the roles of individual inputs. The difficulty of interpreting these models has limited their use in fields like medicine, where interpretation of the model is very important