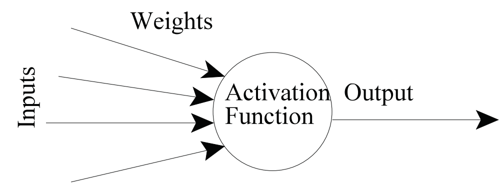
Neural Networks

Bin Li

IIT Lecture Series

Artificial neuron

- Like real neurons, artificial neurons basically consist of:
  - inputs (like synapses), which are multiplied by weights (strength of the respective signals), and then computed by an **activation function**) which determines the activation of the neuron.
  - The **output function** computes the output of the artificial neuron.
- Neural networks combine artificial neurons in order to process information.



Single hidden layer neural networks

- Most widely used neural network.
- Just nonlinear statistical models, closely related to *projection pursuit regression*.
- Inputs: $X_1, X_2, \ldots, X_p$.
- Hidden layer: $Z_1, Z_2, \ldots, Z_M$. Each Z_m is a modelled as a function of linear combination of inputs.
- Outputs: $Y_1, Y_2, \ldots, Y_K$. Each Y_k is a modelled as a function of linear combination of Z_m's.
- For regression: $K = 1$; K-class classification: one for each class.
- Sometimes an additional *bias* unit feeds into every unit of hidden and output layers. Captures the intercepts α_{0m} and β_{0k} in the model.

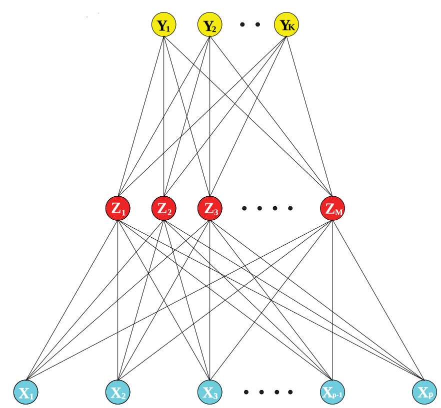


Figure from EOSL 2009

Projection pursuit regression

- Projection pursuit regression (PPR) is an extension of additive model using derived features developed by Friedman and Stuetzle (1981).
- PPR has the form: $f(X) = \sum_{m=1}^{M} g_m(w_m^T X)$
  - The functions g_m are unspecified and estimated along with the directions w_m using some flexible smoothing method.
  - The scalar variable $V_m = w_m^T X$ is the projection of X onto the unit direction vector w_m. We seek w_m so that the model fits well, hence the name "projection pursuit".
- PPR is very general and generates a surprisingly large class of models.
  - For example $[X_1 X_2 = (X_1 + X_2)^2 - (X_1 - X_2)^2]/4$. Higher-order products can be presented similarly.
  - If M is large enough, PPR can approximate any continuous functions in $\mathcal{R}^p$, a *universal approximator*.
- If $M = 1$, known as *single index model* in econometrics. In practice, PPR model typically uses fewer terms (e.g. $M = 5$ or 10).
- PPR can be implemented in R using `ppr` function.

Mathematical model for artificial neural networks

- Derived features Z_m (called *hidden unit*) in the hidden layer is a function of linear combination of inputs.

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

  - Usually the *activation function* $\sigma(v)$ is the *sigmoid* $\sigma(v) = 1/(1 + e^{-v})$.
  - Radial basis function network.
- T_k: a linear combination of Z_m.

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

- The output function:

$$f_k(X) = g_k(\mathbf{T}),\ k = 1, \ldots, K$$

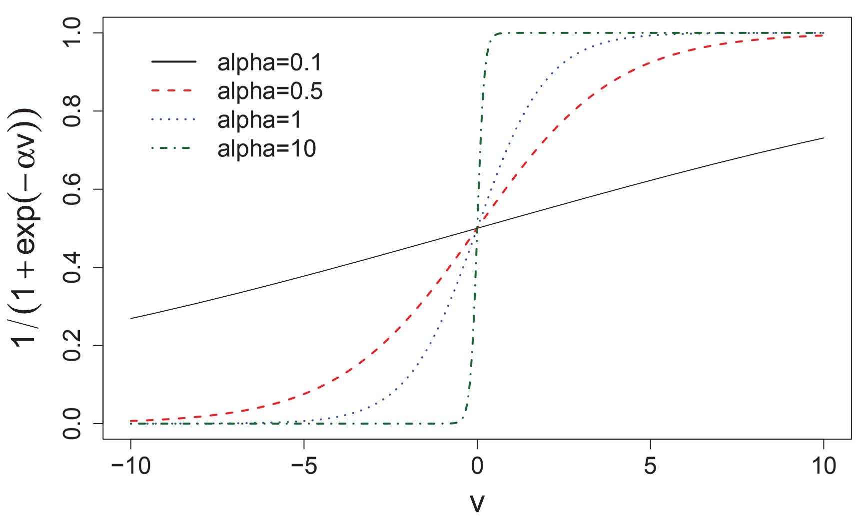
  - For regression, use identity function $g_(T) = T$.
  - For classification, use *softmax* function:

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

Neural networks as a statistical model

- The values of the hidden units Z_m's can be viewed as a basis expansion of the original inputs X. Note the parameters of the basis functions are learned from the data!
- If the activation function $\sigma(v)$ is the identity function, then
  - for regression: it is a traditional linear regression model.
  - for K-classification: it is a linear multilogit model.
- Neural network is a nonlinear generalization of the linear model for both regression and classification.
  - Viewing the NN model as PPR model: $g_m(w_m^T X) = \beta_m \sigma(\alpha_{0m} + \alpha_m^T X)$.
  - PPR uses a flexible g_m, while NN uses a far simpler sigmoid function with 3 parameters.
  - PPR uses few terms ($M = 5$ or 10) while NN uses 20 or 100.
- By using nonlinear activation function σ, it greatly enlarges the class of linear models. The rate of activation of the sigmoid depends on $||\alpha_m||$.
  - if $||\alpha_m||$ is small $\to \sigma$ is approximately linear.
  - if $||\alpha_m||$ is large $\to \sigma$ is approximately a step function.

Sigmoid function



Fitting neural networks

- A single hidden layer NN has unknown parameters, called *weights*. Denote θ as complete set of weights which consists of

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

- For regression, use SSE as measure of fit: $R(\theta) = \sum_k \sum_i (y_{ik} - f_k(x_i))^2$.
- For classification, use cross-entropy: $R(\theta) = -\sum_i \sum_k y_{ik} \log f_k(x_i)$.
- The generic approach to minimizing $R(\theta)$ is by gradient descent, known as *back-propagation* (also called *delta rule*). It iteratively updates the estimate of weights by a two-pass procedure:
  - *forward pass*: fixed current weights and predict $\hat{f}_k(x_i)$.
  - *backward pass*: compute the gradients and update the weights.

  Back-propagation is simple and local nature. Training can be carried online and on a parallel architecture computer.
- Back-propagation can be very slow. Other choices are conjugate gradients and variable metric methods with faster convergence.

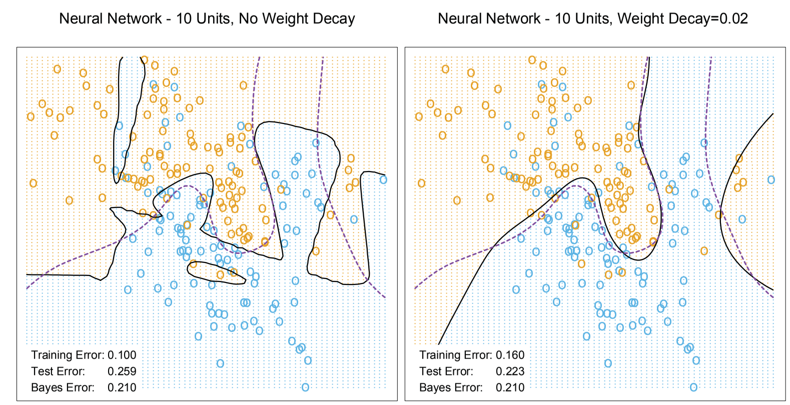
Issues in training neural networks

- Model is overparametrized. Optimization problem is nonconvex and unstable.
- Initial values of weights: random values near zero (i.e. start with a roughly linear model).
- Regularization to avoid overfitting.
  - *early stopping*: Use a validation set to determine when to stop.
  - *weight decay*: add a penalty to loss: $R(\theta) + \lambda(\sum \beta_{km}^2 + \sum \alpha_{ml}^2)$.
  - *weight elimination*: use another penalty:

$$R(\theta) + \lambda \left(\sum \frac{\beta_{km}^2}{1+\beta_{km}^2} + \sum \frac{\alpha_{ml}^2}{1+\alpha_{ml}^2} \right)$$

  . This has the effect of shrinking smaller weights more than weight decay does.
- Scaling of the inputs has a large effect on quality of the solution. Standardize all inputs to mean zero and variance one.
- Number of hidden units and layers: generally better to have too many than too few. Then train them with regularization.
- Multiple local minima: try several random starting configurations and use the best one or **bagging**.

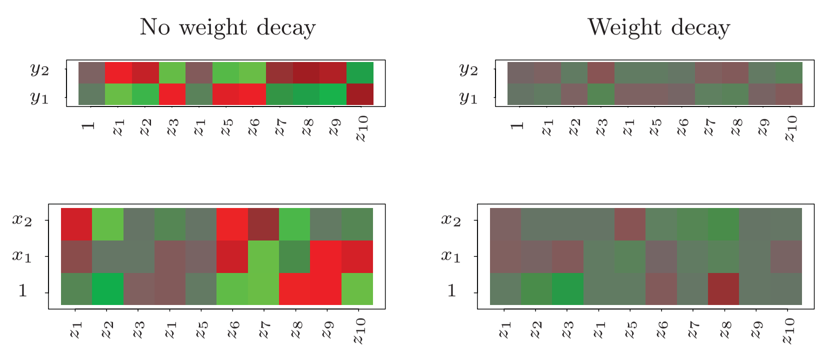
Regularization effects on prediction



The left panel uses no weight decay, and overfits the training data. The right panel uses weight decay, and achieves close to the Bayes error rate (broken purple boundary). Both use the softmax activation function and cross-entropy error.

Figure from EOSL 2009

Regularization effects on weights



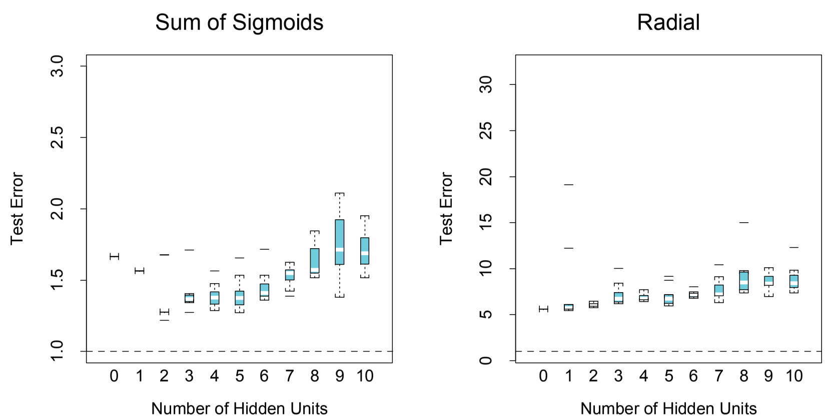
Heat maps of the estimated weights from the training of neural networks. The display ranges from bright green (negative) to bright red (positive). We see that weight decay has dampened the weights in both layers: the resulting weights are spread fairly evenly over the ten hidden units.

Figure from EOSL 2009

Simulation examples

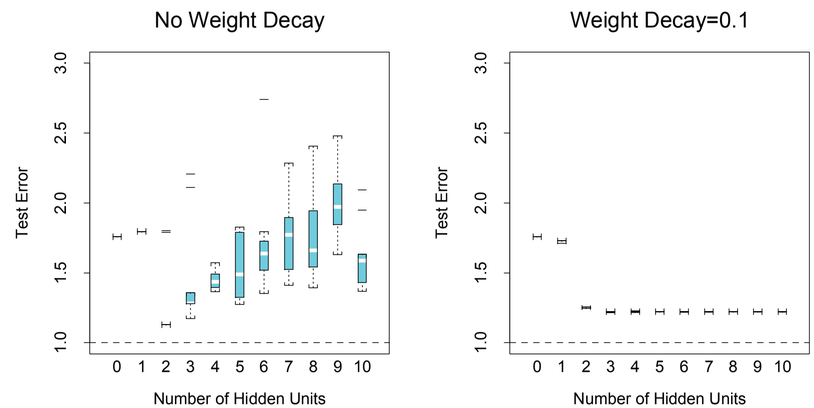
- Two additive error models:
  - Sum of sigmoids: $Y = \sigma(a_1^T X) + \sigma(a_2^T X) + \epsilon_1$
  - Radial: $Y = \prod_{m=1}^{10} \phi(X_m) + \epsilon_2$
- $p = 2$ for "Sum of sigmoids" model with $a_1 = (3,3)$ and $a_2 = (3,-3)$. $p = 10$ for "radial" model where $\phi(t)$ is the standard normal's pdf.
- ϵ_1 and ϵ_2 are mean zero Normal error with signal-to-noise ratio is 4 (i.e. $\mathrm{Var}(f(X))/\mathrm{Var}(\epsilon) = 4$.
- Train set: 100 obs. Test set: 10,000 obs.
- Fit neural networks with and without "weight decay" and various number of hidden units based on 10 random starting weights.
- For figure on next slide, fixed weight decay parameter $\lambda = 0.0005$, a mild amount of regularization.
- For the figure on slide 17, fixed weight decay parameter $\lambda = 0.1$, a much stronger regularization.

Simulation example: number of hidden units effect



- Boxplots of test error relative to the Bayes error (broken horizontal line).
- Zero hidden units is linear least squares regression.
- NN perfectly suits sigmoid example with 2-unit model perform the best. With more hidden units, overfits. Some does worse than linear model.
- Radial example is most difficult to NN (spherically symmetric with no preferred directions). NN performs increasingly worse than the mean (a constant fit achieves a relative error of 5).

Simulation example: weight decay effect



- With no weight decay, overfitting becomes even more severe for larger numbers of hidden units.
- The weight decay value $\lambda = 0.1$ produces good results for all numbers of hidden units. No overfitting as the number of units increases.

Simulation example: weight decay effect (cont.)

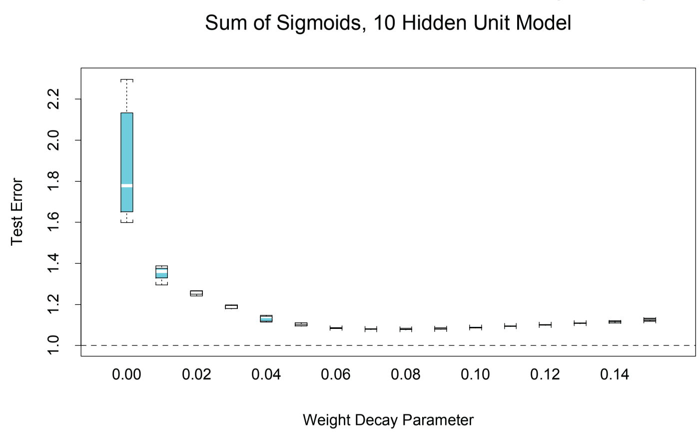


FIGURE 11.8. *Boxplots of test error, for simulated data example. True function is a sum of two sigmoids. The test error is displayed for ten different starting weights, for a single hidden layer neural network with ten hidden units and weight decay parameter value as indicated.*

Figure from EOSL 2009

Neural network packages in R

Neural networks have many variants and usage (as described in previous slide). Here I only listed some packages for *feed-forward* (from input to output without direct circles, different from *recurrent* neural networks) multi-layer perceptrons.

- `nnet` is the `R` standard neural network package. It implements a multi-layer perceptron with one hidden layer. For weight adjustment, it does not use back-propagation, but a general quasi-Newton optimization procedure, the BFGS algorithm.
- `neuralnet` implements standard back-propagation. It allows multiple hidden layers and units. It provides functions to visualize the fitted networks.
- `AMORE` implements the "TAO-robust backpropagation learning algorithm", which is a back-propagation learning algorithm designed to be robust against outliers in the data.
- `monmlp` implements a multi-layer perceptron with partial monotonicity constraints. The algorithm allows for the definition of monotonic relations between inputs and outputs, which are then respected during training.
- `RSNNS` is the *most* comprehensive neural network package in `R`. It includes several types of NNs with different tasks (supervised and unsupervised). On the other hand, there are many options for the main functions. You have to know what they are and what to use!

Main function and its options in `neuralnet` package

```
neuralnet(formula, data, hidden = 1, threshold = 0.01,
          stepmax = 1e+05, rep = 1, startweights = NULL,
          learningrate.limit = NULL,
          learningrate.factor = list(minus = 0.5, plus = 1.2),
          learningrate=NULL, lifesign = "none",
          lifesign.step = 1000, algorithm = "rprop+",
          err.fct = "sse", act.fct = "logistic",
          linear.output = TRUE, exclude = NULL,
          constant.weights = NULL, likelihood = FALSE)
```

- `formula`: a symbolic description of the model to be fitted.
- `hidden`: a vector of integers specifying the number of hidden neurons (vertices) in each layer.
- `threshold`: a numeric value specifying the threshold for the partial derivatives of the error function as stopping criteria.
- `stepmax`: the maximum steps for the training of the neural network. Reaching this maximum leads to a stop of the neural network's training process.
- `rep`: the number of repetitions for the neural network's training.
- `startweights`: a vector containing starting values for the weights. The weights will not be randomly initialized.

Main function and its options in `neuralnet` package (cont.)

- `learningrate`: a numeric value specifying the learning rate used by traditional backpropagation. Used only for traditional backpropagation.
- `algorithm`: a string containing the algorithm type to calculate the neural network. The following types are possible: `backprop`, `rprop+`, `rprop-`, `sag`, or `slr`. `backprop` refers to back-propagation, `rprop+` and `rprop-` refer to the resilient back-propagation with and without weight backtracking, while `sag` and `slr` induce the usage of the modified globally convergent algorithm (grprop).
- `err.fct`: a differentiable function that is used for the calculation of the error. Alternatively, the strings 'sse' and 'ce' which stand for the sum of squared errors and the cross-entropy can be used.
- `act.fct`: a differentiable function that is used for smoothing the result of the cross product of the covariate or neurons and the weights. Additionally the strings, 'logistic' and 'tanh' are possible for the logistic function and tangent hyperbolicus.
- `linear.output`: logical. If `act.fct` should not be applied to the output neurons, then set linear output to TRUE, otherwise to FALSE.

Infertility example

- This data set contains data of a case-control study that investigated infertility after spontaneous and induced abortion (Trichopoulos et al., 1976).
- The data set consists of 8 variables on 248 observations, 83 women, who were infertile (cases), and 165 women, who were not infertile (controls).
- Response: case is a binary variable with 1=case and 0=control.
- Seven input variables: `education` (3 levels); age; `parity` (count); `induced` and `spontaneous` (number of prior induced and spontaneous abortions); `stratum` and `pooled.stratum` (matched set number).

Infertility example (cont.)

```
> library(datasets)
> str(infert)
'data.frame':   248 obs. of  8 variables:
 $ education     : Factor w/ 3 levels "0-5yrs","6-11yrs",..
 $ age           : num  26 42 39 34 35 36 23 32 21 28 ...
 $ parity        : num  6 1 6 4 3 4 1 2 1 2 ...
 $ induced       : num  1 1 2 2 1 2 0 0 0 0 ...
 $ case          : num  1 1 1 1 1 1 1 1 1 1 ...
 $ spontaneous   : num  2 0 0 0 1 1 0 0 1 0 ...
 $ stratum       : int  1 2 3 4 5 6 7 8 9 10 ...
 $ pooled.stratum: num  3 1 4 2 32 36 6 22 5 19 ...
> table(infert$case)
  0   1
165  83
> head(infert)
  education age parity induced case spontaneous stratum pooled.stratum
1    0-5yrs  26      6       1    1           2       1              3
2    0-5yrs  42      1       1    1           0       2              1
3    0-5yrs  39      6       2    1           0       3              4
4    0-5yrs  34      4       2    1           0       4              2
5   6-11yrs  35      3       1    1           1       5             32
6   6-11yrs  36      4       2    1           1       6             36
```

Infertility example (cont.)

```
> set.seed(1)
> indx <- sample(1:248,size=248,replace=F)
> dat1 <- infert[indx[1:200],]   #train set
> dat2 <- infert[indx[201:248],] #test set
>
> set.seed(2)
> nn <- neuralnet(case~age+parity+induced+spontaneous,
+       data=dat1, hidden=4, err.fct="ce",linear.output=FALSE)
> nn
Call: neuralnet(formula=case~age+parity+induced+spontaneous,
      data=dat1, hidden=4, err.fct="ce", linear.output=FALSE)
1 repetition was calculated.
        Error Reached Threshold Steps
1 95.54252438    0.009925027434 17321
> names(nn)
 [1] "call"               "response"           "covariate"
 [4] "model.list"         "err.fct"            "act.fct"
 [7] "linear.output"      "data"               "net.result"
[10] "weights"            "startweights"       "generalized.weights"
[13] "result.matrix"
```


Infertility example (cont.)

```
> out <- cbind(nn$covariate,nn$net.result[[1]])
> dimnames(out) <- list(NULL, c("age", "parity","induced",
+                                "spontaneous","nn-output"))
> head(out)
     age parity induced spontaneous    nn-output
[1,]  32      1       0           1 0.6545318157
[2,]  21      1       0           1 0.1661967105
[3,]  28      2       2           0 0.4609267548
[4,]  28      2       1           0 0.1226684520
[5,]  38      3       0           2 0.9204410272
[6,]  28      1       0           1 0.6540600521
> plot(nn)
> pred1 <- compute(nn,subset(dat2,select=c("age","parity",
+          "induced","spontaneous")))$net.result
> table(round(pred1),dat2$case)
     0  1
  0 28  6
  1  4 10
```


Neural network plot