Nonlinear regression within the linear regression framework

Geir Storvik

April 5, 2018

1 Neural networks

In chapter 6 of James et al. (2013) we defined new variables through principal components or partial least squares in a linear regression setting. The model in that case can be written as

$$\begin{aligned} z_{im} =& \sum_{j=1}^{p} \alpha_{mj} x_{ij} = \boldsymbol{\alpha}_m^T \mathbf{x}_i && m = 1, ..., M \\ y_i =& \beta_0 + \sum_{m=1}^{M} \beta_m z_{im} + \varepsilon_i = \beta_0 + \boldsymbol{\beta}^T \mathbf{z}_i + \varepsilon_i \end{aligned}$$

Here, $z_{im}, m = 1, , ..., M$ can be considered as a set of *latent variables* that are not directly observed but are defined through a linear combination of the original covariates.

A generalisation of this model, taking non-linear relations into account is to assume

$$z_{im} = h(\boldsymbol{\alpha}_m^T \mathbf{x}_i), \qquad m = 1, ..., M \tag{1}$$

$$T_i = \beta_0 + \boldsymbol{\beta}^T \mathbf{z}_i \tag{2}$$

$$y_i = g(T_i) + \varepsilon_i \tag{3}$$

where both $h(\cdot)$ and $g(\cdot)$ are now possible *non-linear* functions. Figure 1 illustrates this model. Note that we may write

$$y_i = f(\boldsymbol{x}_i) + \varepsilon_i$$

where

$$f(\boldsymbol{x}_i) = g(\beta_0 + \sum_{m=1}^{M} \beta_m h(\sum_{j=1}^{p} \alpha_{mj} x_{ij})).$$

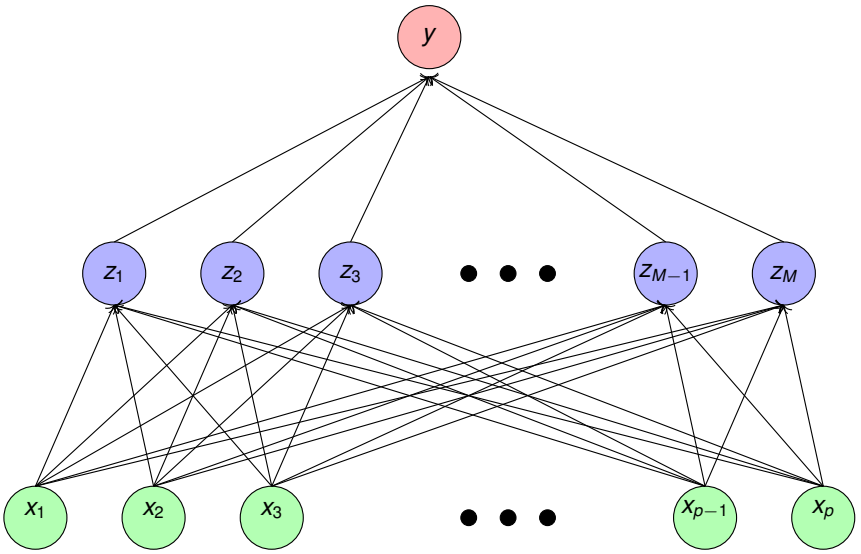

Figure 1: Visualisation of neural network with one hidden layer.

Note that usually, neural networks are not presented through models, but rather through predictions

$$z_{im} = h(\boldsymbol{\alpha}_m^T \mathbf{x}_i), \qquad m = 1, ..., M \tag{4}$$

$$T_i = \beta_0 + \boldsymbol{\beta}^T \mathbf{z}_i \tag{5}$$

$$\hat{y}_i = g(T_i) \tag{6}$$

or

$$\hat{y}_i = f(\boldsymbol{x}_i) \tag{7}$$

where now no distributional assumptions are made on y_i itself. In this case, neural networks are considered more as a blackbox in that the direct relationship between the covariates and the response is not that easy to see.

Different choices of the $h(\cdot)$ and $g(\cdot)$ functions are possible. Typical choices are

$$h(x) = \frac{1}{1 + \exp(-x)} \qquad \text{the sigmoid function;} \tag{8}$$

$$g(z) = z \qquad \text{the identity function.} \tag{9}$$

The resulting model is called a *neural network* model with one hidden layer. This note discuss such models. For more a detailed description, see Friedman et al. (2001, chapter 11)

2 Estimation of parameters

There are a range of parameters to be estimated:

$$\boldsymbol{\theta} = \{\alpha_{mj}, m = 1, ..., M, j = 1, .., p, \beta_m, m = 0, ..., M\}$$

As usual for regression, we will use the sum-of squares errors

$$R(\boldsymbol{\theta}) = \sum_{i=1}^{n} (y_i - f(\mathbf{x}_i))^2.$$

Due to the large number of parameters, overfitting will be a problem for such models. In order to avoid such overfitting, a penalty term, similar to the Ridge regression approach, is typically applied by $Q(\boldsymbol{\theta}) = R(\boldsymbol{\theta}) + \lambda J(\boldsymbol{\theta})$ where

$$J(\boldsymbol{\theta}) = \sum_{m=1}^{M} \sum_{j=1}^{p} \alpha_{mj}^2 + \sum_{m=1}^{M} \beta_m^2.$$

The tuning parameter λ is usually termed the *weight decay* parameter within the neural network literature.

Assuming the $h(\cdot)$ and $g(\cdot)$ are smooth, we can find the optimal value of $Q(\boldsymbol{\theta})$ through derivation.

$$\frac{\partial Q(\boldsymbol{\theta})}{\partial \beta_{m^*}} = -2 \sum_{i=1}^{n} (y_i - f(\mathbf{x}_i)) \frac{\partial f(\mathbf{x}_i)}{\partial \beta_{m^*}} + 2\lambda \beta_{m^*}$$

where, using the chain rule,

$$\begin{aligned} \frac{\partial f(\mathbf{x}_i)}{\partial \beta_{m^*}} =& g'(\beta_0 + \sum_{m=1}^{M} \beta_m h(\sum_{j=1}^{p} \alpha_{mj} x_{ij})) h(\sum_{j=1}^{p} \alpha_{m^* j} x_{ij}) \\ =& g'(\beta_0 + \boldsymbol{\beta}^T \mathbf{z}_i) z_{im} \end{aligned}$$

and

$$\frac{\partial Q(\boldsymbol{\theta})}{\partial \alpha_{m^* j^*}} = -2 \sum_{i=1}^{n} (y_i - f(\mathbf{x}_i)) \frac{\partial f(\mathbf{x}_i)}{\partial \alpha_{m^* j^*}} + 2\lambda \alpha_{m^* j^*}$$

where, again using the chain rule,

$$\begin{aligned} \frac{\partial f(\mathbf{x}_i)}{\partial \alpha_{m^* j^*}} =& g'(\beta_0 + \sum_{m=1}^{M} \beta_m h(\sum_{j=1}^{p} \alpha_{mj} x_{ij})) \beta_{m^*} h'(\sum_{j=1}^{p} \alpha_{m^* j} x_{ij}) x_{ij^*} \\ =& g'(\beta_0 + \boldsymbol{\beta}^T \mathbf{z}_i) h'(\boldsymbol{\alpha}_{m^*}^T \mathbf{x}_i) x_{ij^*} \end{aligned}$$

Computation of these quantities for a given set of parameters can be performed through first performing a *forward* pass of the model (following the arrows in Figures 1) giving the z_{im}'s

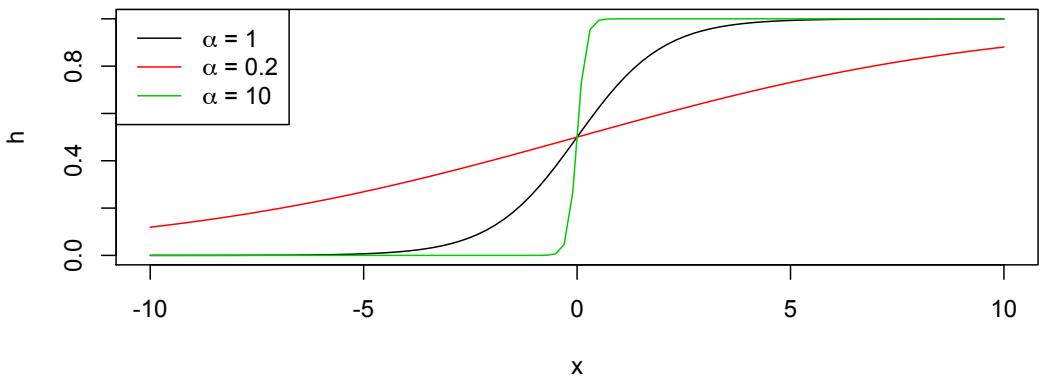

Figure 2: The sigmoid function $1/(1+\exp(-\alpha * x))$ for different values of α.

and $f(\mathbf{x}_i)$. The derivatives are then computed by a *backward* pass through the model. These derivatives are then used to update the parameters through a gradient decent approach given by

$$\beta_m^{r+1} = \beta_m^r - \gamma_r \frac{\partial Q(\boldsymbol{\theta})}{\beta_m}|_{\boldsymbol{\theta}=\boldsymbol{\theta}^r} \tag{10}$$

$$\alpha_{mj}^{r+1} = \alpha_{mj}^r - \gamma_r \frac{\partial Q(\boldsymbol{\theta})}{\alpha_{mj}}|_{\boldsymbol{\theta}=\boldsymbol{\theta}^r} \tag{11}$$

which results in the so-called *back-propagation* algorithm. The tuning parameter γ_r is called the *learning rate*. This learning rate is usually defined as a constant, but can also be a sequence decreasing (slowly) to zero.

The back-propagation algorithm has the advantage of being *local* in that computation of hidden variables are only performed through variables that are connected (arrows in Figure 1). Due to the additive nature of the gradient with respect to the observations, computation can also be performed in batches, enabling parallel computation in cases of huge data sets.

The algorithm will typically be slow in convergence, but including second order derivatives and applying Newton's method can be applied for improving convergence speed.

2.1 The sigmoid function

A popular choice for $h(\cdot)$ is the sigmoid function (8). In Figure 2 the function $1/(1+\exp(-\alpha*x))$ for different values of α are displayed. Note in particular that for small α values the function is almost linear, showing that hidden layers with low weights will correspond to linear terms in the model. For large values of α the sigmoid function works more like a threshold function (resempling regression trees).

3 Neural network for classification

Although most simple to explain for regression, neural networks are most popular for classification tasks. In that case *binary dummy* variables are introduced:

$$y_{ik} = \begin{cases} 1 & \text{if } y_i = k; \\ 0 & \text{otherwise.} \end{cases}$$

Note that $y_i = \text{argmax}_k \, y_{ik}$.

A neural network is then design to predict the responses y_{ik} through

$$z_{im} = h(\boldsymbol{\alpha}_m^T \mathbf{x}_i), \qquad m = 1, ..., M \tag{12}$$

$$T_{ik} = \beta_{0k} + \boldsymbol{\beta}_k^T \mathbf{z}_i, \qquad k = 1, ..., K \tag{13}$$

$$\hat{y}_{ik} = g_k(\mathbf{T}_i), \qquad \mathbf{T}_i = (T_{i1}, ..., T_{iK}) \tag{14}$$

Figure 3 illustrates the model in this case. For $h(\cdot)$ again the sigmoid function is typically used. For the $g_k(\cdot)$ functions, a popular choice has been the *softmax* function

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

which guarantees that the response will be in the interval $[0, 1]$ and sum to one. A classification can then be performed by the rule

$$\hat{y}_i = \underset{k}{\text{argmax}} \, g_k(\mathbf{T}_i).$$

3.1 Zip code data

Le Cun et al. (1990) presented an application of zip code recognition where normalized handwritten digits were automatically scanned from envelopes by the U.S. Postal Service. The original scanned digits are binary and of different sizes and orientations; the images here have been deslanted and size normalized, resulting in 16 x 16 grayscale images. Some examples of these data are shown in Figure 4.

There are many packages within **R** that contain functions for neural network. We will start by using the `nnet` package. First we read the data into **R** by

```
zip.train = read.table("zip.train",col.names=c("cl",paste("x",1:256,sep="")))
zip.test = read.table("zip.test", col.names=c("cl",paste("x",1:256,sep="")))
zip = rbind(zip.train,zip.test)
zip[,-1] = scale(zip[,-1])
```

We first try out a neural network with 10 hidden variables, using the `nnet` routine:

```
ntrain = nrow(zip.train)
zip.nnet = nnet(cl~.,data=zip[1:ntrain,],size=10,decay=0.1,
  MaxNWts=3000,maxit=300)
```

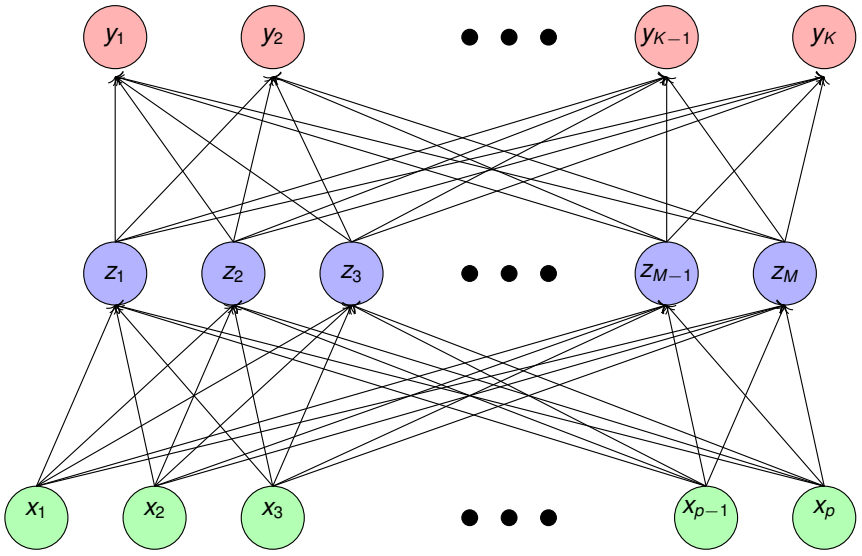

Figure 3: Illustration of the neural network model for classification with K classes.

Here the variable `decay` specifies the penalty term λ, which usually is called the *weight decay* in the neural network literature. The call to `nnet` is quite time-consuming, which should not be a surprise given the number of weights that needs to be calculated in this case, $256 * 10$ from the input variables to the hidden variables and further $10 * 10$ weights from the hidden layers to the output variables. This gives in total 2660 weight parameters to be estimated! The default call to `nnet` do not allow so many weights, but the option `MaxNWts=3000` makes this possible. Prediction on the test set can be performed by the following commands:

```
ntest = nrow(zip.test)
pred = predict(zip.nnet,zip[ntrain+1:ntest,],type="class")
table(zip.test$cl,pred)
mean(zip.test$cl!=pred)
```

This produces the confusion matrix below with an error rate of 10.76%.

	0	1	2	3	4	5	6	7	8	9
0	343	0	8	0	2	0	5	0	1	0
1	0	249	3	2	1	0	3	3	1	2
2	4	2	168	4	5	3	4	3	5	0
3	5	0	6	139	1	10	0	0	4	1
4	2	2	7	1	164	2	5	3	4	10
5	4	1	1	10	1	136	2	1	2	2
6	0	0	4	0	2	4	159	0	1	0
7	1	1	1	0	7	0	0	133	0	4
8	4	5	2	4	2	5	0	5	134	5
9	0	0	1	0	1	2	0	3	4	166

The table below shows how the error rate varies with the number of latent variables. We see that the error rate decreases with the number of latent variables. This is the case despite that the

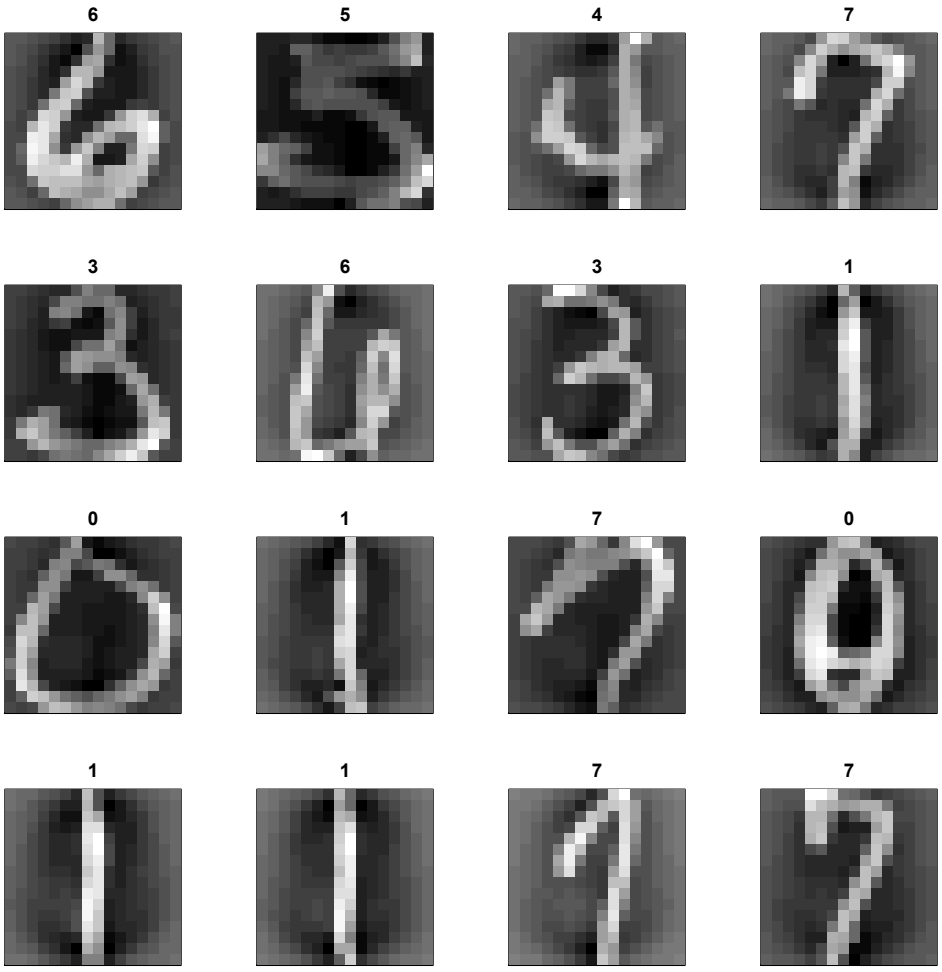

Figure 4: 16 examples of handwritten digits.

number of parameters are increasing significantly. At the same time, the computational time (not shown) is increasing considerable.

Antall latente variable	2	5	10	15	20	30
Antall parametre	532	1330	2660	3990	5320	7980
Feilrate	0.42	0.17	0.11	0.09	0.07	0.065

4 Deep learning

So far, we have looked at one layer of hidden variables. There is however nothing that prevents us from applying several layers of variables, as shown in Figure 5. When working with more than one layer, the name *deep learning* is usually used instead of neural networks.

The point with multiple layers is to increase the flexibility in the model. Each layer uses the

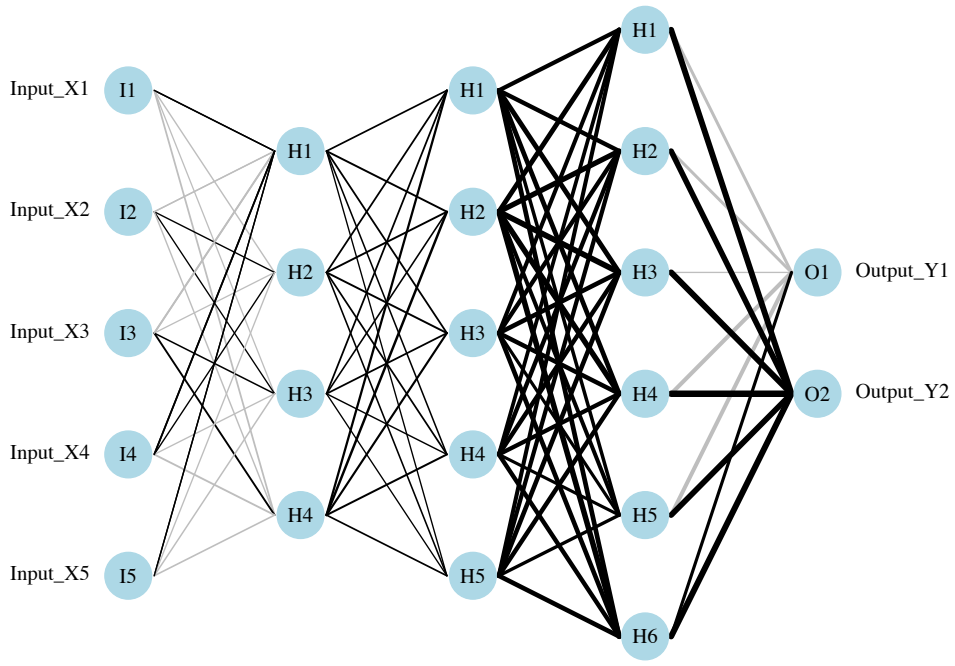

Figure 5: Illustration of a neural net with 5 input variables and 2 output variables (classification with two classes). The net contains 3 hidden layers and 4, 5 and 6 variables within each layer, respectively.

output from the previous layer as input. The number of parameters (weights) will typically be high in deep learning networks. Taking the example in Figure 5 there will be

$$5 * 4 + 4 * 5 + 5 * 6 + 6 * 2 = 82$$

weight parameters to be defined. Note however that if the same number of hidden variables were to be applied in a single layer model, the number of parameters would be

$$4 * 15 + 15 * 2 = 90$$

weights. As with other methods with great flexibility, the problem of overfitting must be taken into account. This can again be tackled by including a penalty term in the object function to be optimised. Estimation of weights can be performed similar to how neural networks with one hidden layer is performed, using the chain rule to obtain derivatives. The expressions will however be more complicated and will be omitted here.

Deep learning has been successfully applied in many problems where a large number of training data are available, e.g. in computer vision, automatic speech recognition and bioinformatics. Some applications can be seen at e.g. `http://machinelearningmastery.com/inspirational-applications-deep-learning/`

4.1 Overfitting

As for neural networks with one hidden layer, deep learning networks can severely overfit. As earlier, a penalty term can be introduced to avoid such overfitting. However, in many implementations of deep learning algorithms, a penalty is introduced more indirectly through *early stopping* of the optimization algorithm. Since the optimization is usually initialized by small weight values, this results in that the weights will not grow too large, corresponding to the penalty introduced earlier. In this case, the number of iterations will serve as a tuning parameter instead of the weight decay parameter λ.

4.2 Deep learning applied to the Zip data

As for neural networks with one hidden layer, there are many packages available for the multilayer neural network (or deep learning) approach. We will consider the `mlp` routine within the `RSNNS` package. The following code produces a network with three hidden layers and 10 variables within each layer. Note that the response needs to be decoded to a specific type in this case, using the built in function `decodeClassLabels`. The `learnFuncParams` option specifies the learning rate γ_r in this case. Note that there is no specification of the penalty term λ in this case.

```
library(RSNNS)
zipTargets <- decodeClassLabels(zip[,1])
zipTargets.train = zipTargets[1:N.tr,]
zipTargets.test = zipTargets[N.tr+1:N.te,]
zip.dnet = mlp(as.matrix(zip.train[,-1]),zipTargets.train, size = c(10,10,10),
               learnFuncParams=c(0.3),maxit=500)
lab = 0:9
pred.dnet = lab[apply(predict(zip.dnet,zip.test[,-1]),1,which.is.max)]
table(zip.test$cl,pred.dnet)
mean(zip.test$cl!=pred.dnet)
```

This produces the following confusion matrix while the error rate is 0.095 in this case.

Size:	Number of latent z's	Number of weight parameters	Test error rate
(12)	12	240	0.160
(6,6)	12	156	0.209
(4,4,4)	12	112	0.254
(3,3,3,3)	12	87	0.449
(11,6)	17	236	0.160
(6,10,6)	22	240	0.160
(5,8,8,5)	26	244	0.218

Table 1: Test error rates for different sizes of deep learning networks.

	0	1	2	3	4	5	6	7	8	9
0	348	0	3	1	2	1	1	0	2	1
1	0	252	1	2	1	2	4	1	0	1
2	3	0	175	2	6	2	2	1	7	0
3	5	0	8	135	0	12	0	0	5	1
4	1	2	5	2	176	0	2	1	2	9
5	4	0	1	7	1	140	3	1	2	1
6	4	0	4	0	2	3	156	0	1	0
7	1	1	1	1	3	1	0	132	1	6
8	3	2	3	2	3	7	5	2	134	5
9	0	0	0	0	2	1	0	2	3	169

A natural question is how many layers and how many latent variables one should use. There is no clear strategy for selecting these, and typically one do a lot of experiments before ending up in a final model. Table 1 shows results from different number of layers and different numbers of latent variables within each layer *when using only the first 10 principal components as input variables*. Note that, at least for this example, the performance seems to go down with the number of weights while being quite stable with different types of sizes when the total number of weights are similar.

It was also mentioned earlier that the number of iterations can be used to control the overfitting. In Figure 6, the error rate is given for different values on the maximum number of iterations. We see that the error rate seems to stabilize in the region 100-1000 indicating that the method is not so sensitive to this tuning parameter.

References

J. Friedman, T. Hastie, and R. Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics Springer, Berlin, 2001.

G. James, D. Witten, T. Hastie, and R. Tibshirani. *An introduction to statistical learning*, volume 6. Springer, 2013.

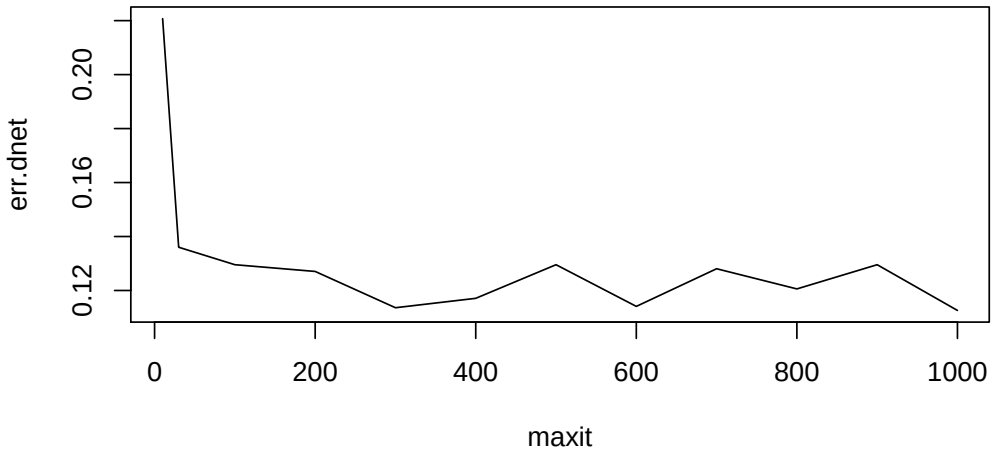

Figure 6: Error rate of deep learning method with size=$(6, 10, 6)$ and with maximum number of iterations varying from 10 to 1000.

Y. Le Cun, O. Matan, B. Boser, J. S. Denker, D. Henderson, R. Howard, W. Hubbard, L. Jacket, and H. Baird. Handwritten zip code recognition with multilayer networks. In *Pattern Recognition, 1990. Proceedings., 10th International Conference on*, volume 2, pages 35–40. IEEE, 1990.