Unit 8

Deep learning with neural networks

Neural networks

- The idea is to
  - extract linear combinations of the inputs as derived features and
  - model the target as a nonlinear function of these features.
- The “vanilla” neural network is a *single hidden layer back-propagation network*.
- A single hidden layer neural network is a two-stage regression or classification model.
- It is typically represented by a *network diagram*.

Single hidden layer

- Derived features Z_m are created from linear combinations of the inputs.
- The target Y_k is modeled as a function of linear combinations of the Z_m.

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

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

Single hidden layer / 2

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

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

- More recently the preferred choice is to use the ReLU (rectified linear unit) activation function

$$\sigma(v) = (v)_+ = \begin{cases} 0 & \text{if } v < 0 \\ v & \text{otherwise} \end{cases}$$

  A ReLU activation can be computed and stored more efficiently than a sigmoid activation.

- Neural network diagrams are sometimes also drawn with an additional *bias* unit feeding into every unit in the hidden and output layers which captures the intercepts α_{0m} and β_{0k}.

Single hidden layer / 3

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

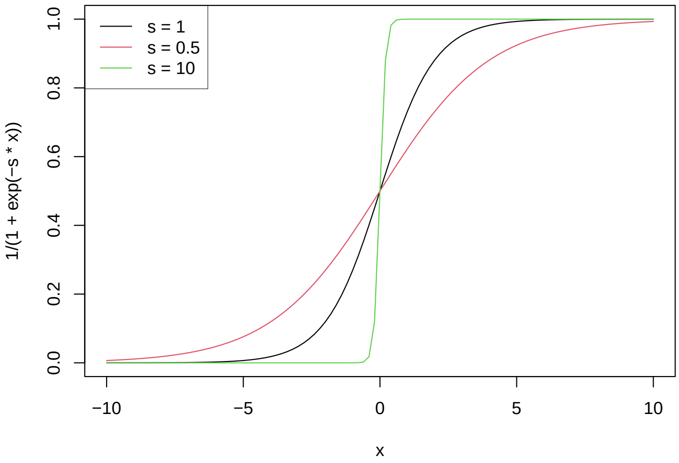
$$g_k(T) = T_k.$$

  - For classification the *softmax* function is typically used

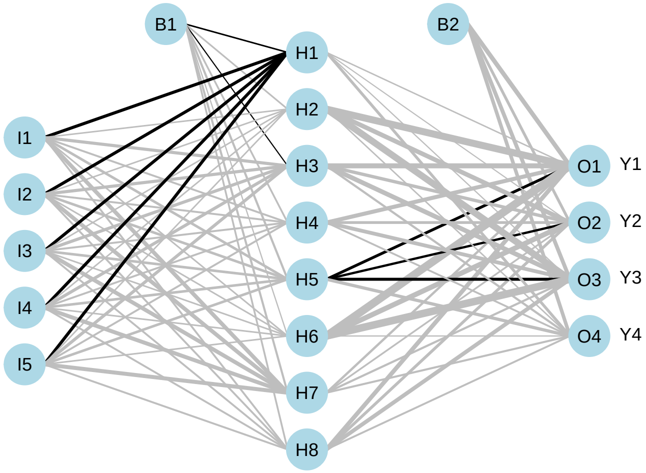
$$g_k(T) = \frac{e^{T_k}}{\sum_{l=1}^{K} e^{T_l}}.$$

    This is the transformation used in the multinomial logit model.

Single hidden layer / 4



Single hidden layer / 5



Single hidden layer / 6

- The units in the middle of the network, computing the derived features Z_m, are called *hidden units*, because the values Z_m are not observed.
- In general there can be more than one hidden layer.
- Z_m can be thought of as a basis expansion of the original inputs X and thus represents a feature extraction step.
- The neural network is then a standard linear or linear multinomial logit model using these transformations as input.
- Compared to other basis-expansion techniques, the parameters of the basis functions are learned from the data.

Single hidden layer / 7

- The name is derived from the fact that they were first developed as models for the human brain.
- Each unit represents a neuron.
- The connections represent the synapses.
- In early models, the neurons fired when the total signal passed to that unit exceeded a certain threshold.
  - This corresponds to the use of a step function instead of the sigmoid.
  - The smoother sigmoid function turned out to be preferable for optimization.

Fitting neural networks

- The parameters in neural networks are the *weights*.
- The complete set of weights is denoted by θ consisting of

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

- The measure of fit are:

$$\text{Regression :} \qquad R(\theta) = \sum_{k=1}^{K} \sum_{i=1}^{N} (y_{ik} - f_k(x_i))^2,$$

$$\text{Classification :} \qquad R(\theta) = -\sum_{k=1}^{K} \sum_{i=1}^{N} y_{ik} \log(f_k(x_i)),$$

with classifier $G(x) = \arg\max_k f_k(x)$.

Fitting neural networks / 2

- 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.
- Typically the global minimizer of $R(\theta)$ overfits the data.
  $\Rightarrow$ Some regularization is needed.
- A generic approach to minimizing $R(\theta)$ is gradient descent which is called *back-propagation* in this setting.

Back-propagation for squared error loss

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

with

$$f_k(x_i) = g_k\left(\sum_{m=1}^{M} \beta_{km}\sigma(\alpha_m^T x_i)\right).$$

For simplicity of notation assume that there is no bias, but the intercept may be suitably included in x_i and z_i.

Back-propagation for squared error loss / 2

This gives the derivatives

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

The gradient descent updates are then:

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

where γ_r is the learning rate.

Back-propagation for squared error loss / 3

Write the derivatives as

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

where δ_{ki} and s_{mi} are “errors” from the current model at the output and hidden layer units.
From their definition these errors satisfy

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

known as the *back-propagation equations*.

Back-propagation for squared error loss / 4

The gradient descent updates can be implemented by a *two-pass algorithm* aka *back propagation*:

- *Forward pass:* The current weights are fixed and the predicted values $\hat{f}_k(x_i)$ are computed.
- *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 updates.

Back-propagation for squared error loss / 5

- The advantages of back-propagation are its simple, local nature: Each hidden unit only passes and receives information to and from units that share a connection.
- The updates are a kind of *batch learning*: The parameter updates are obtained based on all training cases.
  Online learning only processes a single training case. A *training epoch* then refers to a pass through the full training set.
- The learning rate γ_r is either a constant for batch learning or is optimized by a line search in each step. For online learning the learning rate γ_r should decrease to zero as $r \to \infty$.
- Back-propagation can be very slow. Other optimization methods are also used, but the use of second derivatives is generally avoided.

Issues in training neural networks

- Starting values
- Overfitting
- Scaling of the inputs
- Number of hidden units and layers
- Multiple minima

Starting values

- If the weights are near zero, the operative part of the sigmoid function is roughly linear.
- Usually starting values for the weights are chosen to be random values near zero.
- Values exactly equal to zero would lead to zero derivatives and thus imply that the algorithm would never move.

Overfitting

Often neural networks have too many weights and the global minimum of $R(\theta)$ would overfit the data.

- *Early stopping rule*: only run the algorithm for a while and stop before the minimum is attained.
  Given that the weights are selected to start at a regularized linear solution, this implies shrinkage towards the linear model.

Overfitting / 2

- *Weight decay*: a more explicit weight of regularization which is similar to Ridge regression.
  - A penalty is added to the error function

$$R(\theta) + \lambda J(\theta) = R(\theta) + \lambda \left[\sum_{km} \beta_{km}^2 + \sum_{ml} \alpha_{ml}^2 \right]$$

    where $\lambda > 0$ is a tuning parameter. Cross-validation could be used to tune λ.
  - The effect is that the terms $2\beta_{km}$ and $2\alpha_{ml}$ are added to the respective gradient expressions.
  - Alternatively Lasso penalization could also be imposed.

Overfitting / 3

- *Stochastic gradient descent*:
  - Determines the gradient on a minibatch sample for the gradient step.
  - Enforces its own form of approximately quadratic regularization.
- *Dropout learning*: randomly remove a fraction ϕ of the units in a layer.

Scaling of the inputs

- The effective scaling of the weights in the bottom layer depends on the scaling of the inputs.
- It can have a large effect on the quality of the final solution.
- Standardize inputs to mean zero and standard deviation one or to be in the interval $[0, 1]$.
- Draw the weights randomly from a uniform distribution on $[-0.7, +0.7]$.

Number of hidden units and layers

- Generally it is better to have too many hidden units and apply suitable shrinkage.
- In general the number is selected depending on the number of inputs and sample size of the training data.
- It seems better to use cross-validation to tune λ instead of tuning the number of hidden units.
- Choice of number of layers is guided by background knowledge and experimentation.

Multiple minima

- The error function $R(\theta)$ is non-convex with potentially many local minima.
- The result depends on the initialization and either the best solution or an averaged solution could be then used as final model.

Special networks

- *Convolutional neural networks*
  - Developed for classifying images.
  - Combine convolution layers made up of convolution filters with pooling layers.
- *Recurrent neural networks*
  - Developed for sequential input data.
  - The hidden layer combines the input with the value of the activation vector from the previous element.

Example: Spam

- The implementation in **nnet** is used which only provides single hidden layer neural networks.
- The weight decay parameter is selected using a training / validation split.

```
> data("spam", package = "ElemStatLearn")
> set.seed(1234)
> spam <- spam[sample(nrow(spam)),]
> index.test <- seq_len(1536)
> spam.test <- spam[index.test, ]
> spam <- spam[-index.test, ]
```

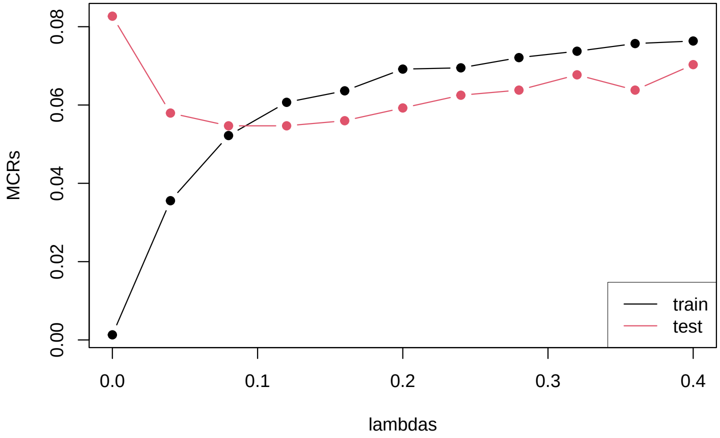
Example: Spam / 2

```
> Min <- apply(spam[-ncol(spam)], 2, min)
> Max <- apply(spam[-ncol(spam)], 2, max)
> spam[-ncol(spam)] <-
+   sweep(sweep(spam[-ncol(spam)], 2, Min, "-"),
+     2, Max - Min, "/")
> spam.test[-ncol(spam.test)] <-
+   sweep(sweep(spam.test[-ncol(spam.test)], 2, Min, "-"),
+     2, Max - Min, "/")
> size <- 10
> lambdas <- seq(0, 0.4, by = 0.04)
```

Example: Spam / 3

```
> library("nnet")
> MCRs <- lapply(lambdas, function(lambda) {
+   fit <- nnet::nnet(factor(spam) ~ ., data = spam,
+     size = size, decay = lambda, trace = 1,
+     skip = TRUE, maxit = 1000)
+   c(train = mean(predict(fit, spam, type = "class") !=
+       spam$spam),
+     test = mean(predict(fit, spam.test,
+       type = "class") != spam.test$spam))
+ })
> MCRs <- do.call("rbind", MCRs)
```

Example: Spam / 4



Example: Spam / 5

```
> lambda <- lambdas[which.min(MCRs[, "test"])]
> model <- nnet::nnet(factor(spam) ~ ., data = spam,
+   size = size, trace = 0, decay = lambda, skip = TRUE,
+   maxit = 1000)
> mean(predict(model, spam.test, type = "class") !=
+   spam.test$spam)
[1] 0.05533854
```

Software for R

- Package **keras** interfaces the high-level neural networks API **Keras**.
- Package **torch** uses the **LibTorch** library to define and train neural networks.

For example code see the R files for Chapter 10 at `https://www.statlearning.com/resources-second-edition`.