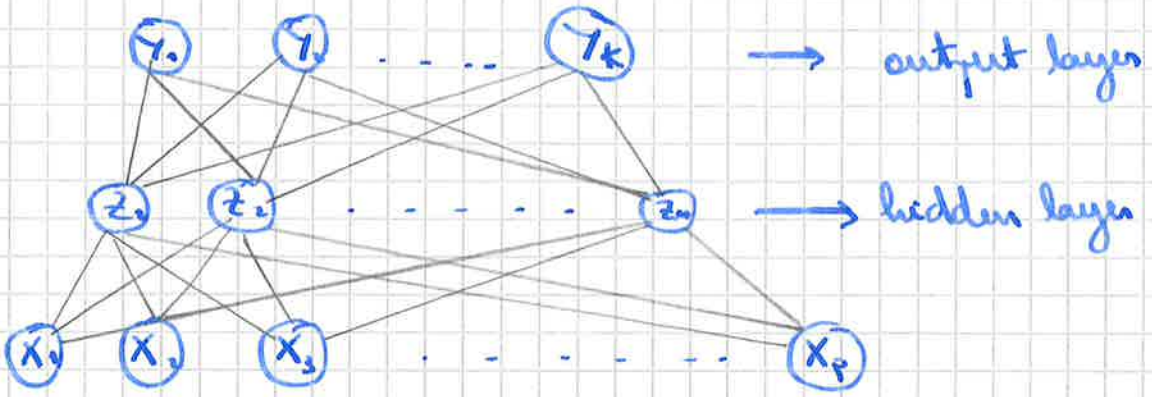
Chapter 6
Neural networks

I) Neural networks

A neural network is a two-stage regression or classification model typically represented by a network diagram. This network applies to both classification and regression.



For regression, typically $K=1$ and there is only one output unit Y_1 at the top. For K-class classification there are K units at the top (k-th unit modeling the probability of class k). There are K target measurements Y_k, $k = 1, \dots, K$.

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

$$(N1)\quad \begin{cases} z_m = \sigma(\alpha_{0m} + \alpha_m^T X) & (m = 1, \dots, n) \\ T_k = \beta_{0k} + \beta_k^T z & (k = 1, \dots, k) \\ f_k(X) = g_k(T) \end{cases}$$

where: $\begin{cases} z = (z_1, z_2, \dots, z_n) \\ T = (T_1, T_2, \dots, T_k) \end{cases}$

Activation function: $\sigma(v)$ is usually chosen to be the sigmoid $\sigma(v) = \frac{1}{1+e^{-v}}$

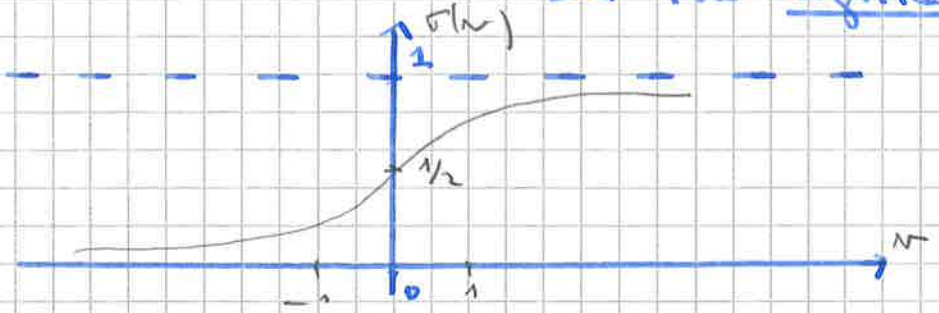


Diagram: You can add a bias unit feeding into every unit in the hidden and output layers. This bias unit captures the intercepts α_{0m} and β_{0k} in the above model.

Output function: $(g_k(T))$ It allows for a final transformation of the vector of outputs. For regression, choose the identity: $g_k(T) = T_k$.

For K-classification, use the softmax function:

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

(you get positive estimates that sum to one).

Hidden units: Units in the middle of the network are called hidden units (because the z_m are not observed). In general, there can be more than one hidden layer.

It is useful to think of the Z_m as a basis expansion of the original inputs X; the neural network can then be seen as a standard linear model. Here, the parameters of the basis on which we expand X (namely, the parameters α) are learned from the data.

Remark 1: If we take $\sigma = Id$ then T is a linear function of X (no need of a hidden layer). Hence a neural network is a nonlinear generalization of the linear model (both for regression and classification). If $\|\alpha_m\|$ is very small then the σ in Z_m will operate in its linear part (and we are back to a linear model).

Remark 2: Initially the network was used to model part of the brain and σ was a step function. Such a step function is not suitable for optimization purposes and so it was replaced by a sigmoid function.

II) Fitting neural networks

Unknown parameters are called weights. The complete set of weights is denoted by θ and consists of:

$\{\alpha_{0m}, \alpha_m : m = 1, 2, \dots, M\}$ (M is dimension p) $\quad M(p+1)$ weights

$\{\beta_{0k}, \beta_k : k = 1, 2, \dots, K\}$ (β_k has dimension M) $\quad K(M+1)$ weights

For regression, we use the sum-of-squared errors as our measure of fit:

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

↑ minimize this

↳ cares for the training data (training set of size N)

For classification, we use either the squared error or the cross-entropy:

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

↑ minimize this

<u>Remark</u>: Why does this make sense?

Take $H(y_i, \cdot) : \eta \in \mathbb{R}^K \longmapsto -\sum_{k=1}^{K} y_{ik} \log \eta_{ik}$ ↳ propagate the connection

constraint: $\sum_{k=1}^{K} y_{ik} = 1$; $\sum_{k=1}^{K} \eta_{ik} = 1$

We want to minimize H under the constraint.

$$\nabla H(y_i, \eta) = \begin{pmatrix} -y_1/\eta_1 \\ \vdots \\ -y_K/\eta_K \end{pmatrix} \qquad \nabla\left(\sum_{i=1}^{K} \eta_i\right) = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$$

Critical point (where $\nabla H(y, \eta)$ colinear to $(1, 1, \dots, 1)^T$):

$$-\frac{y_i}{\eta_i} = \lambda \quad (\forall i)$$

Under the constraint, this becomes: $\eta_i = y_i$ $(\forall i)$

As $H(y_i, \cdot)$ is convex, this is a minimum.

The global minimizer of $R(\theta)$ is likely to be an overfit solution. So some regularization is needed; this is achieved through a penalty term or, indirectly, by early stopping.

The way to minimize $R(\theta)$ is by gradient-descent, called <u>back-propagation</u> in this setting.

Let us have a look at the computation in detail for the squared error loss.

We have: $\begin{cases} z_{mi} = \sigma(\alpha_{0m} + \alpha_m^T x_i) \\ z_i = (z_{1i}, z_{2i}, \dots, z_{Mi})^T \end{cases}$

Then we have: $R(\theta) = \sum_{i=1}^{N} R_i$

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

with derivatives: (δ_{ki})

$$(6.1) \begin{cases} \frac{\partial R_i}{\partial \beta_{km}} = -2 (y_{ik} - f_k(x_i)) g_k'(\beta_k^T z_i) z_{mi} \\ \frac{\partial R_i}{\partial \alpha_{ml}} = -\sum_{k=1}^{K} 2 (y_{ik} - f_k(x_i)) g_k'(\beta_k^T z_i) \beta_{km} \sigma'(\alpha_m^T x_i) x_{il} \end{cases}$$

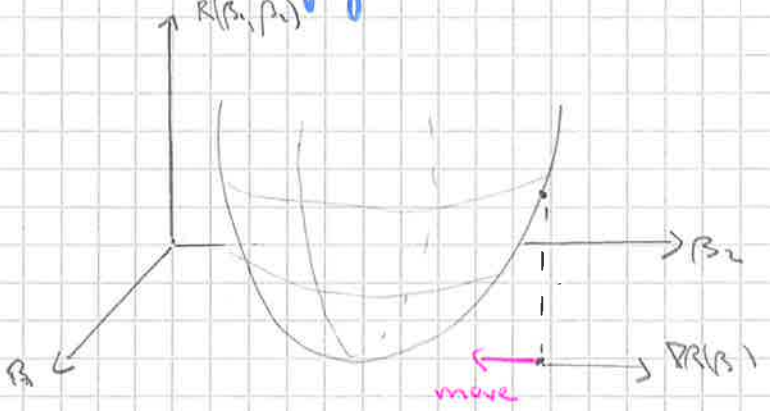
(s_{mi})

The gradient descent update at the $(r+1)$st iteration has the form:

$$(6.2) \begin{cases} \beta_{km}^{(r+1)} = \beta_{km}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \beta_{km}^{(r)}} \\ \alpha_{ml}^{(r+1)} = \alpha_{ml}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \alpha_{ml}^{(r)}} \end{cases}$$

where γ_r is the learning rate (to be chosen later).

Remember the idea of gradient descent:



Re-write Equation ((6.1)) as:

$$((6.2')) \quad \begin{cases} \dfrac{\partial R_i}{\partial \beta_{km}} = \delta_{ki} z_{mi}, & z_{mi} \\ \dfrac{\partial R_i}{\partial \alpha_{ml}} = s_{mi} x_{il}, & x_{il} \end{cases}$$

The quantities δ_{ki} and s_{mi} are called errors from the current model at the output and hidden layers units, respectively. From their definitions, we have:

$$((6.3)) \quad s_{mi} = \sigma'(\alpha_m^T x_i) \sum_{k=1}^{K} \beta_{km} \delta_{ki} \qquad (1 \le i \le N, 1 \le m \le M)$$

known as the back-propagation equations.

So, the updates in ((6.2)) can be implemented in a two-pass algorithm:

* Forward pass: the current weights are fixed and the predicted values $\hat{f}_k(x_i)$ are computed using Equation (11.1) (p. 48).
* Backward pass: the errors δ_{ki} are computed and then back-propagated via (6.3) to give the errors s_{mi}.

Both sets of errors are then used to compute the gradients for the updates in ((6.2)) (via ((6.2'))).

Complexity of the scheme: Each unit passes and receives information only to and from units that share a connection. In the forward pass, we go from bottom to top to compute the $\hat{f}_k(x_i)$'s using Equation (11.1). In the backward pass, we go from top to bottom to compute the δ_{ki}'s and then the s_{mi}.

So this algorithm can be implemented efficiently on a parallel architecture computer.

Remark / notation: → This is called batch learning

Here we used all the training data to update the parameters in (6.2). We could use part of the training data (mini-batch gradient descent: 1 < batch size < size of training set) or only one point in the training data (stochastic gradient descent: batch size = 1). We would have to cycle through all the training data. One sweep through the entire training set is called a training epoch.

Learning rate: the learning rate γ_n for batch learning is usually taken to be a constant. It can be optimized by a line search that minimizes the error function at each update (line search: see Chapter 3). When we update the parameters (in (6.4)) one at a time, we say we are doing online learning and in this case, γ_n should decrease to 0 as $n \to +\infty$. This learning is a form of stochastic approximation (the book cites the following paper: Robbins & Munro, 1951) and it is known that the following conditions ensure convergence:

$$\begin{cases} \gamma_n \to 0 \\ \sum_n \gamma_n = +\infty \\ \sum_n \gamma_n^2 < \infty \end{cases}$$

(take, for example, $\gamma_n = 1/n$)

III) Some issues in training neural networks

1) Starting values

Weights are chosen to be random values near zero.

Exact zero weights ⇒ the algorithm never moves.

Large weights ⇒ poor solutions

2) Overfitting

A method for regularization is weight decay. We add a penalty to the error function: $R(\theta) + \lambda J(\theta)$ where

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

($\lambda \geq 0$ is the tuning parameter). Typically, cross-validation is used to estimate λ.

Another form of penalty:

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

(weight elimination penalty)

3) Scaling of the inputs

It is best to standardize all inputs to have mean zero and deviation one. It is typical (with such standardized inputs) to take to weights uniform in $[-0.7, 0.7]$

4) Number of hidden units and layers

« Choice of the numbers of hidden unit and layers is guided by background knowledge and experimentation. »

5) Multiple minima

The error function $R(\theta)$ is non-convex, possessing many local minima. As a result, the final solution obtained is quite dependent on the choice of starting weights. One must at least try a number of random starting configurations, and choose the solution giving the lowest (regularized) error.

* One can use the average predictions over the collection of networks as the final prediction.
* Averaging the weights is a bad idea since the non-linearity of the model implies that this averaged solution can be quite poor.

Exercises: python book p. 104 - 119

$ pip install graphviz

correction: mlp = MLPClassifier(solver='lbfgs', ...)
cf p. 109

p. 117: You want to use GPU's because they are made for parallelization.

lbfgs is not the gradient descent, it uses the Hessian (Is it Newton's method from chapter 3?)

→ scikit-learn.org does not specify