Lecture 9: Supervised Learning – Neural Networks

Reading Assignment: EoSL Chapter 11.1, 11.3-11.8

Neural Networks in R Tutorial: https://www.datacamp.com/community/tutorials/neural-network-models-r

- Recall dimension reduction methods via derived input methods from Lecture 3: produce *linear* combinations Z_m $[m = 1, \dots, M]$ of original inputs X_j to use in place of X_j in the regression.

$$Z_m = \sum_{j=1}^{p} \phi_{jm} X_j$$

  for some constants $\phi_{1m}, \dots, \phi_{pm},\ m = 1, \dots, M$ where the linear regression model becomes

$$y_i = \beta_0 + \sum_{m=1}^{M} \beta_m \sum_{j=1}^{p} \phi_{jm} x_{ij} + \epsilon_i = \beta_0 + \sum_{j=1}^{p} \beta_j^* x_{ij} + \epsilon_i$$

  thus modeling the target as a linear function of the derived features.

- Also recall nonlinear regression methods from Lecture 4 that relax linearity through linear basis expansion in X:

$$f(X) = \sum_{m=1}^{M} \beta_m h_m(X).$$

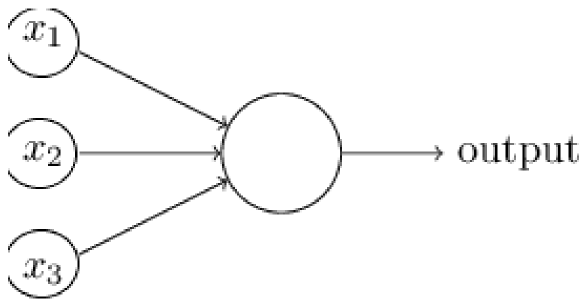
- Neural networks yield a large class of models that expand on these ideas.
- Neural networks extract linear combinations of inputs as derived features, then model the target as a *nonlinear* function of these features.
- They are just nonlinear statistical models – despite their surrounding hype as magical and mysterious!
- A neural network is just a two-stage regression or classification model that is typically represented by a network diagram.
- "Neural network" comes from being first developed as models for the human brain – each unit represents a neuron and the connections are the synapses.
- Lets start with a foundational concept/construct in (artificial) neural networks.

* **Perceptrons**

  – A perceptron is an artificial neuron developed in the field of AI.

  – Perceptrons take binary inputs $X_1, \dots, X_p$ to produce a binary output.

  – *Example*: $p = 3$.



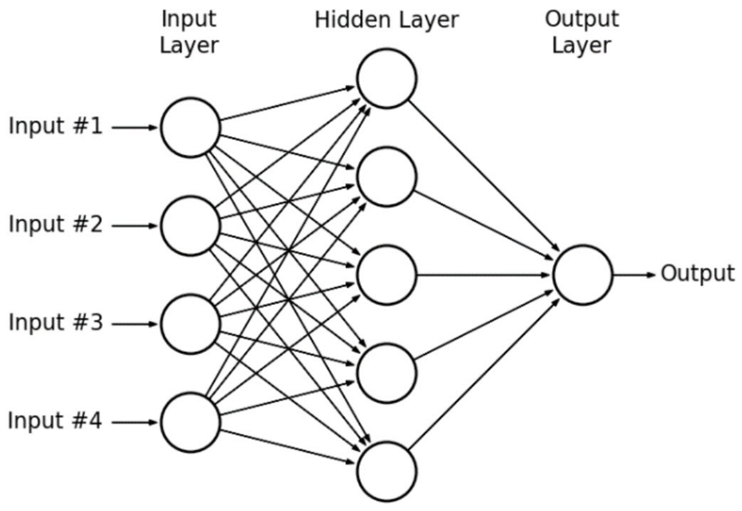
– A simple rule to compute the neurons 0-1 output:

$$\text{output} = \begin{cases} 0 \text{ if } \sum_{j=1}^{p} w_j x_j \leq t \\ 1 \text{ if } \sum_{j=1}^{p} w_j x_j > t, \end{cases}$$

where $w_1, \ldots, w_p$ are weights (reflecting cost/importance of an input on the decision) and t is a real number threshold (the parameter of the neuron).

– The perceptron is a device to make binary decisions by weighting evidence; a model for decision-making.

– It is natural to envision a network of perceptrons.



– The column of perceptrons – called the first layer – is making simple decisions by weighting the input layer.

– *Note*: nodes send signals in only one direction thus called a feed forward network. Recurrent neural networks allow loops in the network.

– A simpler way to write the simple rule

$$\text{output} = \begin{cases} 0 \text{ if } w_j^T x_j + b \leq 0 \\ 1 \text{ if } w_j^T x_j + b > 0, \end{cases}$$

where b is the perceptron's <u>bias</u> that measures how easy it is to get the perceptron to output a 1.

– Use learning algorithms to tune the weights and the bias of a network of artificial neurons.

- Lets use this to formulate the plain vanilla neural net as a learning algorithm.

* **Single Hidden Layer (Single Layer Perceptron) Network**

– A single hidden layer neural network diagram for regression (typically Y_1 only) or classification $(Y_1, \ldots, Y_K)$.

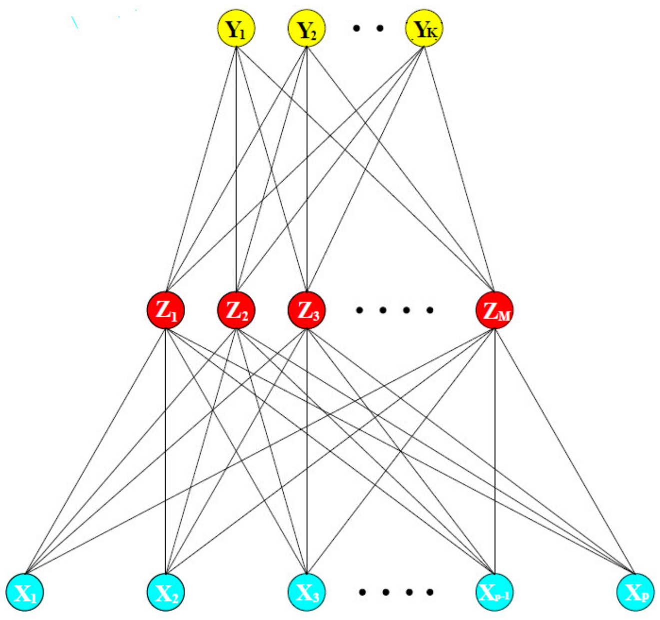


FIGURE 11.2. *Schematic of a single hidden layer, feed-forward neural network.*

– All neural networks have an input and output layer.

– Edges "link" the nodes – a "fully connected" network ensures that all nodes in the lower layer contribute to each layer above.

– Derived features, a.k.a. perceptrons or <u>hidden units</u>, Z_m define the <u>hidden layer</u>. They are created from linear combinations of the inputs

$$Z_m = \sigma\Big(\alpha_{0m} + \alpha_m^T X\Big) \tag{1}$$

for $m = 1, \dots, M$ [from earlier notation $\alpha_j = w_j$ and $\alpha_0 = b$].

– In original formulations σ is a step function – i.e. neurons fire when the signal exceeded a threshold – but it is more useful for statistical modeling (and optimization) to use a smooth threshold function.

– That is, for the perceptron a small change in input values can cause a large change in output; a better solution would be to output a continuum of values.

– The <u>activation function</u> $\sigma(\nu)$ is usually chosen to be the sigmoid (a.k.a. logit, softmax) function

$$\sigma(\nu) = \frac{1}{1 + e^{-\nu}}, \tag{2}$$

a smoothed version of a step function.

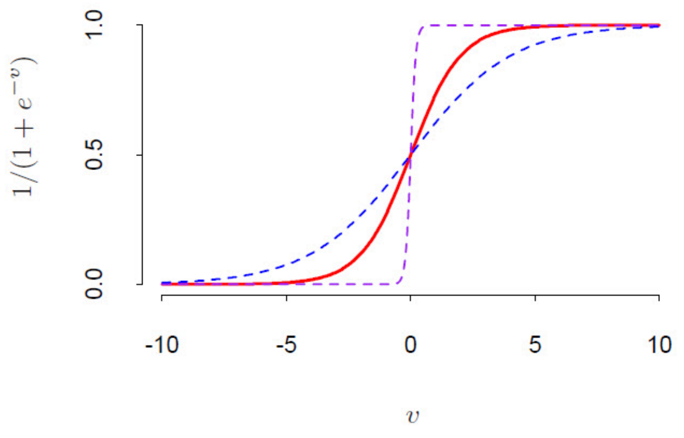


FIGURE 11.3. *Plot of the sigmoid function* $\sigma(v) = 1/(1+\exp(-v))$ *(red curve),*

– The sigmoid neuron model allows inputs to take any value between 0 and 1 and yields an output that is not just 0 or 1.

– For particularly positive or negative ν, results will be nearly the same as the perceptron (0 or 1); but intermediate values will be spread out.

– The target Y_k is modeled as a function of linear combinations of Z_m

$$f_k(X) = g_k\Big(\beta_{0k} + \beta_k^T Z\Big) \tag{3}$$

for $k = 1, \ldots, K$ where $Z = (Z_1, \ldots, Z_m)$.

– The output function $g_k(T_k)$ allows a final transformation of the outputs $T_k = \beta_{0m} + \beta_k^T Z$. Typically chosen to be

▷ identity function for regression: $g_k(T_k) = T_k$,

▷ softmax/multilogit function for classification: $g_k(T_k) = \dfrac{e^{T_k}}{\sum_{\ell=1}^K e^{T_\ell}}$.

– In summary, components of neural networks:

▷ *input layer*: p features $X_1, \dots, X_p$,

▷ *hidden layer* (1): $Z_m = \sigma\Big(\alpha_{0m} + \alpha_m^T X\Big)$ for $m = 1, \dots, M$,

▷ *output layer* (3): $f_k(X) = g_k\Big(\beta_{0k} + \beta_k^T Z\Big) = g_k\Big(T_k\Big)$,

▷ *activation function* (2): $\sigma(\alpha_{0m} + \alpha_m^T X) = \frac{1}{1+e^{-(\alpha_{0m}+\alpha_m^T X)}}$, and

▷ *link function* (3): g_k, e.g. logit or identity.

– Considering Z_m as a pre-defined basis expansion of original inputs and $\sigma(\nu)$ assumed to be the identity function $\Rightarrow$ standard nonlinear models from Lecture 4, i.e. nonlinear generalization of the linear model.

– But now introducing the nonlinear transformation σ greatly expands the class of models.

– Also in neural networks, *the basis functions are learned from the data* jointly with the target model; the Z_m are not directly observed – they are hidden!

* Fitting Neural Networks (Single Layer)

– Need to estimate the unknown parameters, a.k.a. weights, denoted by θ which consists of

$$\{\alpha_{0m}, \alpha_m; m = 1, \ldots, M\} \to M(p+1) \text{ weights},$$
$$\{\beta_{0k}, \beta_k; k = 1, \ldots, K\} \to K(M+1) \text{ weights}.$$

where, recall, M is the number of linear combinations of X, and K is the dimension of the target.

– As always, we are looking to minimize error (loss)

$$R(\theta) = \sum_{k=1}^{K} \sum_{i=1}^{N} L\big(y_{ik}, f_k(x_i)\big).$$

– *Example*: classification using cross-entropy/deviance loss

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

with the corresponding classifier $G(x) = \max_k f_k(x)$. *Note*: softmax activation function and deviance error $\Rightarrow$ logistic regression model in Z so can use ML estimation.

– A generic approach to minimizing $R(\theta)$ is by <u>gradient descent</u>, a.k.a. <u>back-propagation</u> in this setting.

– This approach is reasonable because the form of the gradient can easily be derived via chain rule – the model is defined in a compositional form.

– *Example*: regression using squared error loss for multivariate target $Y =$

$(Y_1, \ldots, Y_K)$,

$$\begin{aligned}
R(\theta) &= \sum_{i=1}^{N} R_i = \sum_{i=1}^{N} \sum_{k=1}^{K} \Big(y_{ik} - f_k(x_i)\Big)^2, \\
&= \sum_{i=1}^{N} \sum_{k=1}^{K} \Big(y_{ik} - g_k\big(\beta_{0k} + \beta_k^T z_i\big)\Big)^2, \\
&= \sum_{i=1}^{N} \sum_{k=1}^{K} \left(y_{ik} - g_k\left(\beta_{0k} + \sum_{m=1}^{M} \beta_{km} z_{im}\right)\right)^2, \\
&= \sum_{i=1}^{N} \sum_{k=1}^{K} \left(y_{ik} - g_k\left(\beta_{0k} + \sum_{m=1}^{M} \beta_{km} \sigma\left(\alpha_{0m} + \alpha_m^T x_i\right)\right)\right)^2,
\end{aligned}$$

yielding derivatives (indicating how quickly loss changes with the weights)

$$\begin{aligned}
\frac{\partial R_i}{\partial \beta_{km}} &= \frac{\partial R_i(\theta)}{\partial g_k} \frac{\partial g_k}{\partial T_k} \frac{\partial T_k}{\partial \beta_{km}} \\
&= \Big[-2\,\big(y_{ik} - f_k(x_i)\big)\; g_k'(\beta_{0k} + \beta_k^T z_i)\Big] z_{im} \\
&= \delta_{ki}\; z_{im}, \\
\frac{\partial R_i}{\partial \alpha_{mj}} &= \sum_{k=1}^{K} \frac{\partial R_i(\theta)}{\partial g_k} \frac{\partial g_k}{\partial T_k} \frac{\partial T_k}{\partial \sigma} \frac{\partial \sigma}{\partial \alpha_{mj}} \\
&= \left[-2 \sum_{k=1}^{K} \big(y_{ik} - f_k(x_i)\big)\; g_k'(\beta_{0k} + \beta_k^T z_i)\; \beta_{km}\; \sigma'(\alpha_{0m} + \alpha_m^T x_i)\right] x_{ij} \\
&= s_{mi}\; x_{i\ell},
\end{aligned}$$

so that the gradient descent update at the $(r+1)$st iteration is

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

$$\alpha_{mj}^{(r+1)} = \alpha_{mj}^{(r)} \; - \; \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \alpha_{mj}^{(r)}}$$

with <u>learning rate</u> γ_r.

The quantities δ_{ki} and s_{mi} are the "errors" from the current model at the output and hidden layers, respectively and satisfy

$$\begin{aligned} s_{mi} \;&=\; -2 \;\sum_{k=1}^{K} \big(y_{ik} - f_k(x_i)\big)\; g_k'(\beta_{0k} + \beta_k^T z_i)\; \beta_{km}\; \sigma'(\alpha_{0m} + \alpha_m^T x_i) \\ &=\; \sum_{k=1}^{K} \Big[-2\; \big(y_{ik} - f_k(x_i)\big)\; g_k'(\beta_{0k} + \beta_k^T z_i)\Big] \beta_{km}\; \sigma'(\alpha_{0m} + \alpha_m^T x_i) \\ &=\; \sigma'(\alpha_{0m} + \alpha_m^T x_i) \sum_{k=1}^{K} \delta_{ki}\; \beta_{km} \end{aligned}$$

which are called the <u>back-propagation equations</u>.

The updates using gradient descent can be implemented in a *two-pass algorithm* called <u>back-propagation</u>:

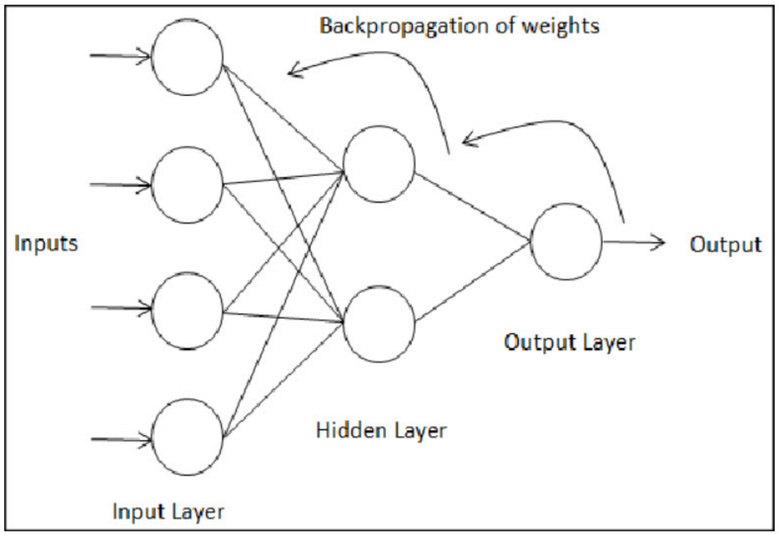
▷ <u>forward pass</u>: fix weights θ at current value $\hat{\theta}^{(r)}$ and compute $\hat{f}_k(x_i)$,

▷ <u>backward pass</u>: fix Z_m, T_k and $f_k(x_i)$ (output) and update θ using a gradient descent step. Specifically, compute errors δ_{ki} to obtain s_{mi} [i.e. back-propagate], then use errors to compute the gradients to get $\hat{\theta}^{(r+1)}$,

i.e. the back propagation algorithm is a numerical way to compute the gradient to use in the gradient descent numerical optimization algorithm.

– Computational components for deviance loss have same form as for squared error.

– *Intuition*: assess the output error at each step and update the weights of the neural network to reduce error.



– *Advantages*:

▷ hidden unit passes/receives information only to/from units with a connection – can parallelize, and

  - ▷ can be done as batch learning by summing over all training cases; or online learning processing each observation at a time (cycling through the entire training dataset a specified number of times called epochs), updating gradient after each training case to allow very large training sets.

- Training neural networks is an art – they are generally overparameterized, are a nonconvex optimization problem and can be very unstable.

- There are some very important practical considerations that can make your life easier.

* **Practical Issues in Training Neural Networks**

(1) *Starting Values*

- If the weights are near zero, then the operable part of the sigmoid is roughly linear $\Rightarrow$ network collapses to an approximately linear model.
- Good practice to start model out nearly linear and let the network become nonlinear as the weights increase.
- Starting at exactly zero leads to zero derivatives – algorithm never moves.
- Starting with large weights often leads to poor solutions.

(2) *Overfitting*

- Typically the global minimizer of $R(\theta)$ will overfit [lots of weights!], so need to add regularization.

- Early developments used an early stopping rule to stop well before approaching the global minimum. Weights start at a highly regularized (linear) solution so this effective shrinks toward a linear model.

- Can explicitly regularize using <u>weight decay</u> which is analogous to ridge regression for linear models to minimize

$$R(\theta) + \lambda \left(\sum_{k=1}^{K} \sum_{m=1}^{M} \beta_{km}^2 + \sum_{m=1}^{M} \sum_{j=1}^{p} \alpha_{mj}^2 \right)$$

where $\lambda \geq 0$ is a tuning parameter.

- $\nearrow \lambda \Rightarrow$ shrink weights to zero.

- Typically use CV to select λ.

- Other forms of penalty have been proposed.

(3) *Scaling of Inputs*

- Scaling of inputs determines the effective scaling of weights in the input layer and can have a large effect on quality of the final solution.

- Always standardize inputs to have mean zero and standard deviation one.

- Ensures all inputs are treated equally in the regularization process.

- Also can then choose meaningful range for random starting weights: typical to take random uniform weights over the range $[-0.7, +0.7]$.

(4) *Number of Hidden Units*

– Generally better to have too many hidden units than too few (not enough flexibility).

– With too many, the extraneous can be shrunk to zero with regularization.

– Best to choose reasonable large M and train them with regularization.

– Typical choice is in range of 5 to 100, with the number increasing with p and N.

– Can use CV, but not necessary if you use CV to estimate the regularization parameter λ.

(4) *Multiple Minima*

– The error function $R(\theta)$ is nonconvex with many local minima.

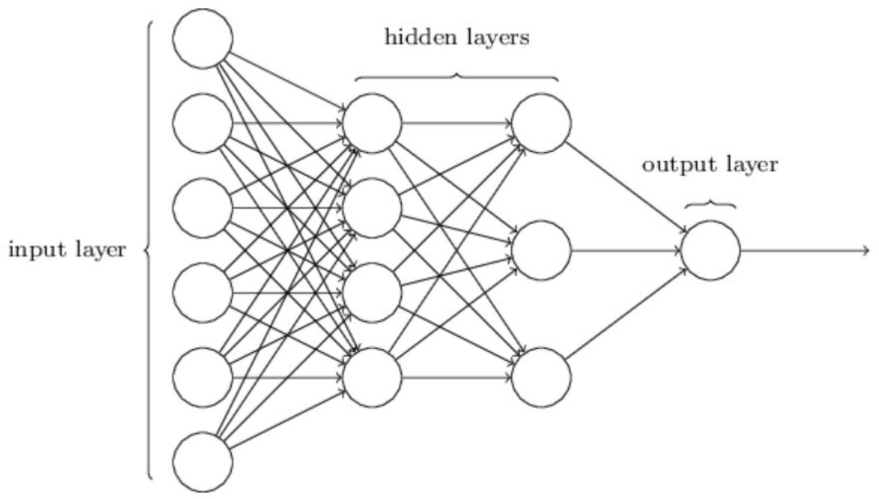
– Final solution is quite dependent on starting weights.

– Best to try many random starting value configurations and choose solution with lowest error.

– Or could use averaging of predictions over the collections of networks or bagging (averages of bootstrapped training data).

- More complicated neural networks can be created with more hidden layers, i.e. multilayer perceptron (MLP).

- Additional layers make simple decisions by weighting the results from the previous layer; they make decisions at a more complex and abstract level than previous layer.



For single layer only [not estimated with gradient descent] with regularization:

— `nnet()` function in `nnet` package in R —

Allows multiple layers (vector for `hidden` option) and a variety of algorithms but not regularization:

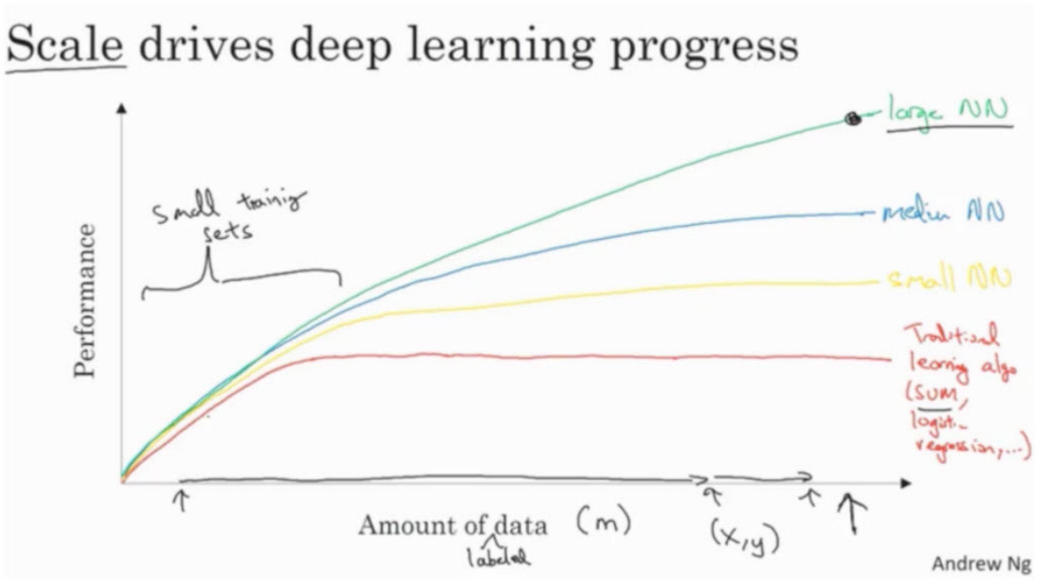
— `neuralnet()` function in `neuralnet` package in R —

Allows multiple layers (vector for `hidden.layers` option) with regularization options and batch options:

— `neuralnetwork()` function in `ANN2` package in R —

- What makes neural networks so different and great?

- Scale! As larger neural networks are constructed and trained with more and more data, their performance continues to increase, whereas for other ML methods performance plateaus.



Shows how scale drives performance in DNN's

- Computing power continues to improve – able to fit larger and larger neural networks.

- This is the crux of "deep learning" – large neural networks, e.g. MLP – where the graph is deep with many layers.

- But the require more computation power and more data.

- Also require more development time – not "off-the-shelf" (i.e. fast, easy to train). A variety of hyperparameters can be tuned:

- ▷ *neural network structure*: λ, number of layers, M, $\sigma(.)$, $g_k(.)$;
- ▷ *optimization/training algorithm*: loss function, γ_r, starting values, batch size, number of epochs.