Chapter 8
Some Other Procedures Briefly

There are statistical learning procedures not discussed earlier that can be framed as regression analysis and deserve at least conceptual overviews. Perhaps the most widely known is neural networks. Although neural networks was the poster child for early work in machine learning, it is now an important niche player, primarily within some forms of "deep learning." Neural networks will be discussed briefly to give a general sense of its structure, associated concepts, and performance in practice.

A second procedure, squarely in statistical learning traditions, is Bayesian additive regression trees (BART). It has some of the look and feel of random forests, but tuning is done by placing prior distributions on decision tree parameters. Perhaps its primary strength is that uncertainty is explicitly and defensibly addressed within Bayesian perspectives. But so far at least, it too is a niche player. Nevertheless, there is a lot of interest in BART. It is certainly worth a brief overview.

A third approach is reinforcement learning, which some view as the conceptual paradigm for artificial intelligence. We will use genetic algorithms as an illustration. Reinforcement learning shares many features with boosting, but it is less a stand-alone procedure and more a key component in some forms of deep learning or in guidance and control systems used in robotics. Reinforcement learning is briefly discussed because it is considered by many to be a form of machine learning.

8.1 Neural Networks

Neural networks, neural nets for short, was an early attempt within computer science to develop software that approximated the way collections of neurons function. We now know that the neural network algorithms are a vastly oversimplified rendering of

The original version of this chapter was revised: See the "Chapter Note" section at the end of this chapter for details. The erratum to this chapter is available at https://doi.org/10.1007/978-3-319-44048-4_10.

R.A. Berk, *Statistical Learning from a Regression Perspective*,
Springer Texts in Statistics, DOI 10.1007/978-3-319-44048-4_8

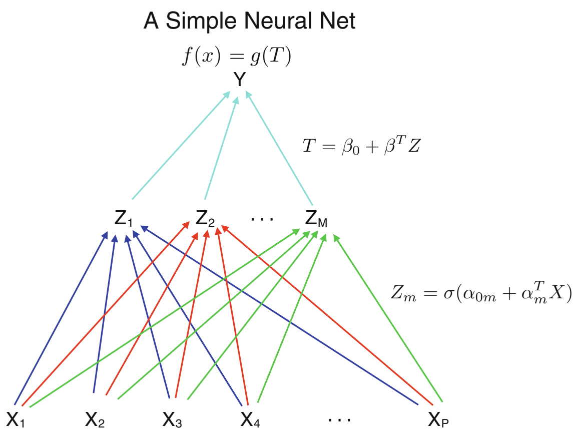

A Simple Neural Net
$f(x) = g(T)$
Y
$T = \beta_0 + \beta^T Z$
Z1
Z2
...
ZM
$Z_m = \sigma(\alpha_{0m} + \alpha_m^T X)$
X1
X2
X3
X4
...
XP

Fig. 8.1 A neural net with one response, one hidden layer, and no feedback

neural activity, but viewed as another machine learning procedure, they can perform well.[1] From a statistical learning perspective, neural nets are a way to combine inputs in a nonlinear manner to arrive at outputs. Put another way, a complicated $f(X)$ is approximated by a composition of many, far more simple functions. By now, this should have a familiar ring.

For notational consistency, we build on Hastie et al. (2009: Sect. 11.3). Figure 8.1 is a schematic of a very simple neural network. The inputs are represented by $x_1, x_2, \ldots, x_p$. These are just the usual set of predictors. There is a single output, Y, although more complicated networks can have several different outputs. Y can be numerical or categorical and is just the usual response variable. There is also a single "hidden layer" $z_1, z_2, \ldots, z_M$ that can be seen as a set of M unobserved, latent variables. All three components (i.e., inputs, output, and latent variables) are linked by associations that would be causal if one were trying to represent the actions of a collection of neurons (with no feedback).

It all starts with the inputs that are combined in a linear fashion for each latent variable. That is, each latent variable is a function of its own linear combination of the predictors. For the mth latent variable, one has

$$Z_m = \sigma(\alpha_{0m} + \alpha_m^T X), \tag{8.1}$$

where αs are coefficients (also called "weights") that vary over the M latent variables, X is the set of p inputs, and σ is commonly a sigmoid "activation function." A key idea behind the S-shape is that a linear combination of inputs will be more likely to trigger an impulse as that linear combination of the inputs increases in value, but

[1]An excellent introductory lecture on neural nets by Patrick Winston of MIT can be found at http://teachingexcellence.mit.edu/inspiring-teachers/patrick-winston-6-034-lecture-12-learning-neural-nets-back-propagation. If one is willing to learn a somewhat different notational scheme, Christopher Bishop's treatment is superb (2006: Chap. 5).

variation in the linear combination towards the middle of its range alters Z_m the most.[2]

In the next step, a linear combination of the latent variable values is constructed as

$$T = \beta_0 + \beta^T Z, \tag{8.2}$$

where now the βs are the coefficients (also called "weights") and Z is the set of latent variables. One has a linear combination of the M latent variables. Finally, the linear combination can be subject to a transformation

$$f(x) = g(T), \tag{8.3}$$

where g is the transformation function. When Y is numerical, the transformation simply may be an identity. When Y is categorical, the transformation may be logistic, much as in logistic regression.

There is no explicit representation of any disturbances, either for Z or Y, which is consistent with early machine learning traditions. The need to fit data with a neural net implies the existence of residuals, but they are not imbued with any formal statistical properties. It is not apparent, therefore, how to get from a level I analysis to a level II analysis. There is also no generative model, let alone a causal model. Despite its name, a neural network is algorithmic (Breiman 2001b).

If one substituted the M versions of Eq. 8.1 into 8.2, each version of Eq. 8.1 would be multiplied by its corresponding value of β. Consequently, the impact of the inputs would be re-weighted as a product of its β_m; a nonlinear, multiplicative transformation has been applied to the inputs. When those results are inserted into Eq. 8.3, there is the option of applying another nonlinear transformation. In short, one has built a set of sequential, nonlinear transformations of the inputs to arrive at the output; a series of simple nonlinear transformations are used to approximate a complicated $f(X)$. In the process, there is a new blackbox algorithm from which the associations between inputs and outputs are no longer apparent. Neural nets succeeds or fails by how well its fitted values for Y correspond to the actual values of Y.

Estimating the values of both sets of weights would be relatively straightforward if Z were observable. One would have something much like a conventional structural equation model in econometrics. But with Z unobservable, estimation is undertaken in a more complicated fashion that capitalizes on the sequential structure of the neural net: from X to Z to Y.

As usual, a loss function associated with the response must be specified. For example, if the response is quantitative, quadratic loss would be a likely choice. Then, because of the sequential nature of Eqs. 8.1–8.3, the inputs are used to construct the values of the latent variables that in turn are combined to arrive at $\hat{Y}$. But one still needs values for both sets of the weights. Consistent with many statistical leaning

[2]The color coding of the arrows in Fig. 8.1 is meant to indicate that each hidden layer has its own set of weights. These weights will typically differ from one another; the set of α_m will typically differ. Their common color is not meant to convey that the weights are the same.

algorithms discussed in earlier chapters, these first need to be initialized. Values randomly chosen close to 0.0 are often a good choice because one starts out with something very close to a linear model; products of the weights do not matter much.

Given the known values of X and an initialized weight for each α, fitted values for each Z_m follow directly. The set of initialized weights for the βs then determine the fitted values for Y. From these, the loss is computed.

One hopes that by revising the weights, it is possible to reduce the loss overall. Recall from the earlier discussion of stochastic gradient boosting, that a gradient is a partial derivative. Consider first the βs. The gradient expression for each β is the partial derivative of the loss with respect to that β. Things are little more complicated for the αs because their impact on the loss is altered by the βs. But by the chain rule in calculus, one can arrive at the gradient expression for each α, which is the partial derivative of the loss with respect to that α (Hastie et al. 2009: Sect. 11.4).

The expressions for the gradients are evaluated using the fitting disparities employed as arguments in the loss function. One proceeds by working backwards from the disparities to arrive the gradients' numerical values for the βs and then, the αs. This process is called backpropagation. With these values in hand, one can apply gradient descent to update the weights. At that point, the fitting and backpropagations begin again and repeat until the loss cannot be further reduced.

The backpropagation approach comes with several complications. First, start values can really matter because the loss functions are not convex. One can get stuck in a local minimum. A common approach is to repeat the estimation several times with different sets of start values and then choose the result with the smallest value of the loss. Also, there is some reason to think that for very high dimensional data, the local minimums will not be all that different from the global minimum. Second, with so many weights, overfitting can be a serious problem. Some form of regularization can help. Penalizing the fit in the spirit of ridge regression is one option. Test data can also be very important. Third, the different units in which the inputs are measured can make a big difference. Standardizing the inputs is usually helpful. Fourth, in Fig. 8.1, each input is connected to each latent variable, and each latent variable is connected to the response. The network is saturated. No inputs are directly linked to the response, although that can be an option. In short, there can be a large number of different network structures, which implies that inductively some weights can be set to 0.0. A form of model selection has been introduced. Finally, the number of latent variables and hidden layers are tuning parameters typically arrived at through some combination of subject-matter knowledge and performance in cross-validation. In effect, they are tuning parameters. Sometimes as many as 100 latent variables will be required.

There have been some recent enhancements in neural networks that are beyond the scope of this discussion, but two examples are worth brief mention. First, with so many parameters, a form of regularization can be introduced by imposing a prior distributions on each. The result is "Bayesian neural nets" that for estimation relies on extensive preprocessing of the data and very sophisticated Markov Chain Monte Carlo (MCMC) methods (Neal and Zhang 2006). In practice, Bayesian neural nets seems to perform very well and can be competitive with the other statistical learning procedures discussed earlier.

Second, one of the problems with traditional neural networks is that by today's standards, its fitting procedure is insufficiently adaptive and flexible. Deep learning can be seen as an effort to overcome these liabilities. It is common in deep learning to employ a very large number of latent variables and latent variable ("hidden") layers that are used to fit highly complex and highly nonlinear relationships (Deng and Yu 2014). This has led to considerable success in pattern recognition with very high dimensional image or speech datasets. One example is identifying small tumors on x-rays of lung tissue. In addition, one can introduce feedback loops and ensembles of neural networks, both of which can make deep learning deeper (Schmidthuber 2014).

It is also possible to deploy a neural network in a manner that reduces computational burdens apparently with no appreciable decline in accuracy. For example, one can fit the network by moving through hidden layers in a stagewise fashion. That is, the weights for the latent variables within one hidden layer are determined over many iterations before moving on to the weights for latent variables within the next hidden layer. Another strategy is to partition the x-values into subsets that can be learned separately before being combined. For example, different parts of an image can be learned separately. An important variant of this idea is to capitalize on spatial autocorrelation in an image and treat contiguous pixels that are much alike as a single observational unit by applying a transformation called "convolution". For example, 36 pixels can be treated as a single spatial neighborhood. The set of such spatial neighborhoods can comprise an image of reduced dimensionality that is further processed. In the end, one has a "convolution neural network" (Bishop 2006: Sect. 5.5.6). Much more is going on than can be considered here, and there are excellent treatments on the web (e.g., https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/).

Deep learning seems to dominate when extremely precise fits are required, and very small improvements can make a large practical difference. For example, a reduction in a loss function of 1 % can be a big deal. There are, however, at least three significant obstacles to widespread use. First, the computational burdens are enormous and growing. Despite innovations in hardware and software, the appetite for more computing power will not soon (or ever) be satisfied. Second, time consuming and complicated tuning is required. There is virtually no formal guidance, and the available craft lore is too rarely definitive. It can take weeks to tune a single application. Finally, there are now many flavors of deep learning with the number expanding rapidly. New claims of superior performance are routinely made even when the improvements seem to be data dependent. It will take a while for the field of deep learning to become better consolidated.

Formally, deep learning seems to be solely a level I enterprise. The data are treated as a very large finite population. There is no scaffolding on which to build inferences beyond the data. But in practice, inferences are often drawn to realizations that have not yet materialized. One might argue that with so many observations, there is no sampling error to worry about. This view only addresses the variance in fitted values.

Should the new realizations come from a different joint probability distribution, there can be substantial bias. One can get very reliable estimates of values that are systematically wrong. In response, some might claim that because very large neural networks can fit complicated functions, bias must be very small. Yet, this overlooks that complex neural networks can only fit the data they have. If the new realizations come from a different joint probability distribution, it is possible that fitting the data on hand better can make generalization error worse. In short, deep learning has the same inferential problems that all machine learning procedures share.

Deep learning procedures in R currently are a bit behind the curve, but they are catching up rapidly. For example, the library *h20()* (Candel et al. 2016) is a platform for a wide variety of machine learning procedures including random forests, gradient boosting, and "deep" neural nets as well as several unsupervised machine learning procedures such as principal components. There are claims that *h20()* can handle billions of observations in memory. An essential requirement for such impressive overall performance is that the data analyst has access to multiple cores, usually in a computer cluster. Other promising implementations of deep learning in R include the packages *mxnet*, *darch*, *deeplearning*, and *deepnet*.

8.2 Bayesian Additive Regression Trees (BART)

Bayesian additive regression trees (Chipman et al. 2010) is a procedure that capitalizes on an ensemble of classification or regression trees in the spirit of random forests and stochastic gradient boosting. Random forests generates an ensemble of random trees by treating the tree parameters as fixed while sampling the training data and predictors. Stochastic gradient boosting also capitalizes on sampling. Both operate with frequentist statistical traditions. Parameters such as the terminal node proportions are treated as fixed, and the data are treated as a collection of random realizations from a joint probability distribution of random variables. Bayesian additive regression trees turn this upside-down. Just as in Bayesian traditions more generally, the data are treated as fixed, and parameters characterizing the ensemble of trees are treated as random.

Each tree in the BART ensemble is realized at random in a manner that is determined by three kinds of hyperparameters:

1. Two hyperparameters for determining the probability that a node will to be split;
2. A hyperparameter determining the probability that any given predictor will be selected for the split; and
3. A hyperparameter determining the probability that a particular split value for the selected predictor will be used.

From the first two hyperparameters, the probability of a split is determined by

$$p(\text{split}) = \alpha(1 + d)^{-\beta}, \tag{8.4}$$

where d is the tree "depth", defined as the number of stages beyond the root node; 0 for the root node (i.e., $d = 0$), 1 for the first split (i.e., $d = 1$), 2 for two splits beyond the root node (i.e., $d = 2$), and so on. The values of α and β affect how large a given tree will be. For a given value of β, smaller values for α make a split less likely. For a given value of α, smaller values of β make the penalty for tree depth more binding so that a split for a given value of d is less likely. The probability of a split also declines as the value of d increases. Possible values of α range from 0 to 1 with .95 a common choice. Possible values for β are nonnegative, but small values such 2 often work well and can be treated as sensible defaults (Kapelner and Bleich 2014).

If there is to be a split, a predictor is chosen at random by the value of the second hyperparameter. For example, if there are p predictors, each predictor can have a probability of $1/p$ of being selected. In a similar manner for the third hyperparameter, the split value for the selected predictor can be chosen with equal probability. It is possible to alter either random selection strategy if the situation warrants it. For example, if on subject-matter grounds some predictors are thought to be more important than others, they may be chosen with a higher probability than the rest. The relevant hyperparameter value would be altered accordingly.

The hyperparameters define very large Bayesian forests composed of potential trees that can be realized at random. Each tree is grown as usual with indicator variables for linear basis expansions of the predictors. Over trees, one has a very rich menu of possible expansions, a subset of which is realized for any given data analysis. In other words, the role of the hyperparameters is to produce a rich dictionary of linear basis expansions.

Figure 8.2 is meant to provide a sense of how this works in practice. Imagine a ball rolling down an inclined plane. In Fig. 8.2, the ball is shown in red at the top. The ball hits the first nail, and its path is displaced to the left or to the right in a manner that cannot be predicted in advance. The first nail is analogous to the first

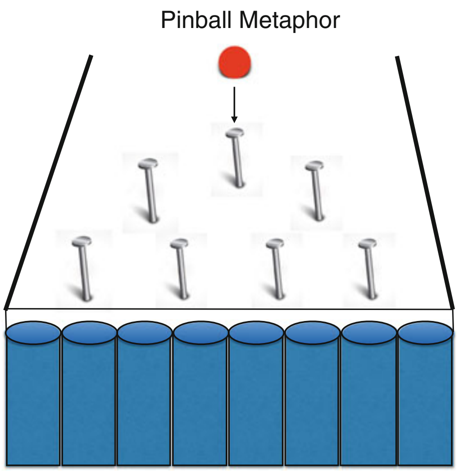


Fig. 8.2 Cartoon illustration of a pinball process (The ball is at the top, the nails are the splits, and the cups at the bottom are the terminal nodes.)

random split. Each subsequent nail is a subsequent potential split that can shift the path of the ball to the left or to the right in the same unpredictable fashion. After the third set of nails, the ball drops into the closest canister. The canisters at the bottom represent terminal nodes.

Imagine now that 25 balls are sequentially rolled down the plane. The 25 balls represent the cases constituting a dataset. Some of the balls are red and some are blue. The balls will follow a variety of paths to the canisters, and the proportions of red and blue balls in each canister will likely vary. The exercise can be repeated over and over with the same 25 balls. With each replication, the proportion of red balls and blue balls in each canister will likely change.

In Fig. 8.2, there are two sets of nails after the first nail at the top. To be more consistent with BART, the number of nail rows can vary with each replication. In addition, the rows can vary in the number nails and where they are placed. For example, the two left most nails in the bottom row of Fig. 8.2 might not be included. The result is a large number of inclined planes, with varying numbers of nail rows, nail placements and proportions of red and blue balls in each canister. One has a fixed collection of red and blue balls that does not change from replication to replication, but what happens to it does.[3]

Unlike random forests, the realized trees are not designed to be independent, and the trees are used in a linear model (Chipman et al. 2010):

$$Y = \sum_{j=1}^{m} g(x; T_j, M_j) + \varepsilon, \tag{8.5}$$

and as usual,

$$\varepsilon \sim \mathrm{N}(0, \sigma^2). \tag{8.6}$$

Y is numerical, and there are m trees. Each tree is defined by the predictors x, T_j, which represents the splits made at each interior node, and M_j, which represents the set of means over terminal nodes. The trees are combined in a manner something like conventional backfitting such that each tree's set of conditional means is related to the response by a function that has been adjusted for the sets of conditional means of all other trees.

But, we are not done. First, we need a prior for the distribution of the means of Y over terminal nodes conditional on a given tree. That distribution is taken to be normal. Also, Y is rescaled in part to make the prior's parameters easier to specify and in part because the rescaling shrinks the conditional means toward 0.0. The impact of individual trees is damped down, which slows the learning process. Details are provided by Chipman and colleagues (2010: 271). Second, we need a prior distribution for σ^2 in Eq. 8.6. An inverse χ^2 distribution is imposed that has two hyperparameters. Here too, details are provided by Chipman and colleagues (2010: 272).

[3]Wu, Tjelmeland and West (2007) define a "pinball prior" for tree generation. The pinball prior and Fig. 8.2 have broadly similar intent, but the details are vastly different.

The algorithms used for estimation involve a complicated combination of Gibbs sampling and Markov Chain Monte Carlo methods that can vary somewhat depending on the software (Kapelner and Bleich 2014). In R, there is *bartMachine* written by Adam Kapelner and Justin Bleich, and *BayesTree* written by Hugh Chipman and Robert McCulloch. Currently, *bartMachine* is the faster in part because it is parallelized, and it also has a richer set of options and outputs. For example, there is a very clever way to handle missing data.

A discussion of the estimation machinery is beyond the scope of this short overview and requires considerable background in Bayesian estimation. Fortunately, both procedures can be run effectively without that background and are actually quite easy to use. In the end, one obtains the posterior distributions for the conditional means from which one can construct fitted values. These represent the primary output of interest. One also gets "credibility intervals" for the fitted values.

BART is readily applied when the response is binary. Formally,

$$p(Y = 1|x) = \Phi[G(x)], \tag{8.7}$$

where

$$G(x) \equiv \sum_{j=1}^{m} g(x; T_j, M_j), \tag{8.8}$$

and $\Phi[G(x)]$ is the probit link function used in probit regression. When used as a classifier, the classes are assigned in much the same way they are for probit regression. A threshold is applied to the fitted values (Chipman et al. 2010: 278). Otherwise, very little changes compared to BART applications with a numerical response.

It is difficult to compare the performance of BART to the performance of random forests, stochastic gradient boosting, and support vector machines. The first three procedures have frequentist roots and make generalization error and expected prediction error the performance gold standard. Because within a Bayesian perspective the data are fixed, it is not clear what sense out-of-sample performance makes. If the data are fixed, where do test data come from? If the test data are seen as random realizations of some data generation process, uncertainty in the data should be taken into account, and one is back to a frequentist point of view. Nevertheless, if one slides over these and other conceptual difficulties, BART seems to perform about as well as random forests, stochastic gradient boosting, and support vector machines. Broadly conceived, this makes sense. BART is "just" another way to construct a rich menu of linear basis expansions to be combined in a linear fashion.

BART's main advantage is that statistical inference is an inherent feature of the output; a level II analysis falls out automatically. But one has to believe the model. Are there omitted variables, for example? Does one really want to commit to a linear combination of trees with an additive error term? One also has to make peace with the priors. Most important, one has to be comfortable with Bayesian inference. But, even for skeptics, Bayesian inference might be considered a reasonable option given all of the problems discussed earlier when frequentist inference is applied to statistical learning procedures. It is also possible to use BART legitimately as a level I tool. The hyperparameters and priors distributions can be seen as tuning parameters and given no Bayesian interpretations. BART becomes solely a data fitting exercise.

BART has some limitations in practice. At the moment, only binary categorical response variables can be used. There is also no way to build in asymmetric costs of classification or fitting errors. And if one wants to explore the consequences of changing the hyperparameters (e.g., α and β), it not clear that conventional resampling procedures such as cross-validation make sense with fixed data.

In summary, BART is a legitimate competitor to the random forests, stochastic gradient boosting, and support vector machines. If one wants to do a level II analysis within a Bayesian perspective, BART is the only choice. Otherwise, it is difficult to see why one would pick BART over the alternatives. On the other hand, BART is rich in interesting ideas and for the more academically inclined, can be great fun.

8.3 Reinforcement Learning and Genetic Algorithms

Recall how boosting algorithms work. Observations that are fit less well in one pass through the data are given more weight in the next pass through the data. The algorithm learns with each pass how to better target problematic observations and is rewarded with an improved set of fitted values. One can say that the reweighting strategy is reinforced with each iteration, and the algorithm alters how it proceeds in a manner that responds to its performance. A fitting task can be more challenging when the training data are sampled because the data to be fit change with each iteration. But the same reinforcement process applies. In the end, the sets of fitted values are linearly combined.

Reinforcement learning can be seen as variants on, and extensions of, the foundations of boosting. Sutton and Barto (2016) provide an excellent introduction. For reinforcement learning, there are rules by which algorithms operate and an environment in which the algorithm is applied. The algorithm receives feedback on its performance in that environment and alters its actions in response. The environment may or may not change as well. Then, the process is repeated. The entire process is repeated many times until some satisfactory outcome is achieved. And like boosting, it never looks back; there are no do-overs.

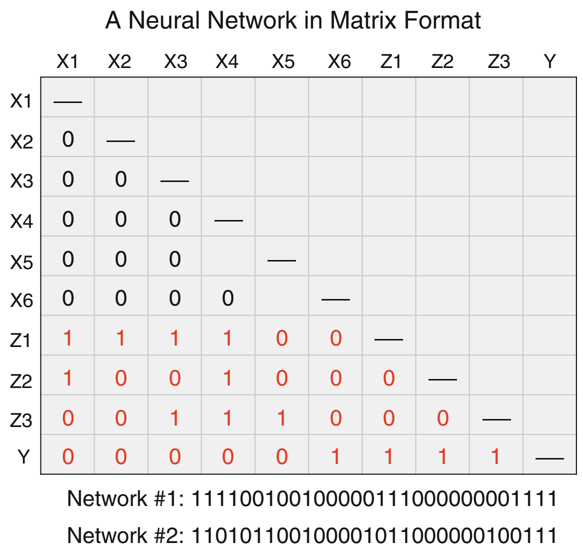
Reinforcement learning is sometimes included within the tent of machine learning. Of late, it has gotten lots of attention as a component of deep learning. To provide a grounded sense of reinforcement learning, we briefly consider genetic algorithms (Mitchell 1998: 95–96).

8.3.1 Genetic Algorithms

Sometimes computer code written to simulate some natural phenomenon turns out to be a potential data analysis tool. Neural networks seems to have morphed in this fashion. Genetic algorithms appear to be experiencing a similar transformation. Whereas neural networks were initially grounded in how neurons interact, genetic algorithms share the common framework of natural selection. When used to study evolution, they are not tasked with data analysis, either as a data summary tool or as a data generation model. Applied to optimization problems, they can proceed in the same spirit as gradient descent or Newton-Raphson when as a practical matter, the loss function cannot be optimized. This came up earlier in a different context when for CART, a greedy algorithm was employed because evaluating all possible trees was untenable.

Here's the basic idea. There is an optimization problem and an initial population of candidate solutions, each evaluated for their fitness. Based on that fitness, an initial culling of the population follows. The survivors "reproduce," but in the process, the algorithm introduces "genetic" mutations, crossovers and sometimes other alterations to some of the progeny. Crossovers, for example, produce offspring whose makeup is a combination of the features of two members of the population. Fitness is again evaluated, and the population is culled a second time. The process continues until population fitness attains a target level of fitness (Affenseller 2009: Sect. 1.2). Thus, genetic algorithms are not "algorithmic" in Breiman's sense. They do not link inputs to outputs but can be a key components of how the linking gets done.

To illustrate how this can work, we revisit neural nets and the problem of which links should be included between inputs, latent variables and outputs. In a somewhat artificial fashion, we lift out one part of the fitting challenge: what should be the structure of the neural network? The discussion that follows draws heavily from Mitchell's introductory application of genetic algorithms to neural network architecture (1998: 70–71).



A Neural Network in Matrix Format

	X1	X2	X3	X4	X5	X6	Z1	Z2	Z3	Y
X1	—									
X2	0	—								
X3	0	0	—							
X4	0	0	0	—						
X5	0	0	0		—					
X6	0	0	0	0		—				
Z1	1	1	1	1	0	0	—			
Z2	1	0	0	1	0	0	0	—		
Z3	0	0	1	1	1	0	0	0	—	
Y	0	0	0	0	0	1	1	1	1	—

Network #1: 111100100100000111000000001111

Network #2: 110101100100001011000000100111

Fig. 8.3 A matrix representation of simple neural network (A 1 denotes a link, and a 0 denotes no link. The red entries can be varied between 1 and 0. The black entries are fixed. Two solution candidates are shown below the matrix.)

Figure 8.3 is a matrix representation of a simple neural network with 6 inputs ($X1 \ldots X6$), 3 latent variables ($Z1 \ldots Z3$), and 1 numerical response (Y). There is no feedback. The 1s denote links, and the 0s denote the absence of links. Consistent with usual practice, there are no links between inputs. But there are potential links between inputs, latent variables, and the response that need to be specified. For example, $X5$ is connected to $Z3$, but not to the other latent variables. $X6$ is connected to none of the latent variables but is connected directly to Y.

Network #1 is a vector of binary indicators consistent with the figure. The indicators are entered by row as one would read words in an English sentence. Network #2 is another vector of binary indicators that represents another structure for the 6 inputs, 3 latent variables, and 1 response. Both representations can be seen as candidate solutions for which fitness is measured by a loss function with Y and $\hat{Y}$ as arguments. The algorithm would begin with a substantial number of such candidate solutions. A neural network would be fit with each. The task is to find the neural net specification that minimizes the loss, and that loss will depend in part on the architecture of the network.

One might think that there is a brute-force solution. Just try all possible network specifications and find the one with the best fit. However, for all but relatively simple networks, the task may not be computationally feasible. Each possible network specification would require its own set of fitted values. We faced a similar problem earlier when we considered all possible classification trees for a given dataset. Much as a greedy algorithm can provide a method to arrive at a good tree, a genetic algorithm can provide a method to arrive at a good network specification. "Good" should not be read as "best."

One might proceed as follows (Mitchell 1998: 10–11).

1. Generate a random population of N network specifications, two of which are illustrated by network #1 and network #2 in Fig. 8.3. There would be some constraints such as not allowing links between inputs and requiring at least one path from an input to the response. The population of specifications would not be exhaustive.
2. Suppose Y is numeric. Compute the mean squared error for each candidate specification as a measure of fitness. This means fitting a neural net for each.
3. Sample with replacement a pair of candidate specifications with the probability proportional to fitness.
4. With a specified probability (a tuning parameter) cross the pair of candidate specifications at a randomly chosen point. That point would be selected with equal probability. For example, suppose for network #1 and #2, the equivalent of a coin flip comes up heads. A crossover is required. There are 28 possible break points, and suppose the 5th possible break point (going left to right) is selected at random. The values of 1, 1, 0, 0, 1 from the network solution #1 would be swapped with values 1, 1, 0, 1, 0 from the network #2 solution.
5. Mutate locations in each pair with some small probability (another tuning parameter). This means on occasion changing a 0 to a 1 and a 1 to a 0 where such changes are allowed (e.g., not between inputs).
6. Compute the mean squared error for each of the two progeny.

7. Repeat steps 3–6 until *N* offspring have been produced and replace the old population with the new population.
8. Repeat steps 2–7 and repeat until there have been a sufficient number of generations (e.g., 100).
9. Select the network structure from the population of network structures that is most fit (i.e., has the smallest mean squared error).[4]

One can now see why genetic algorithms are said to "learn", and why they can be a form of reinforcement learning. The algorithms discover what works. With steps that have parallels to natural selection, solutions that are more fit survive. In contrast, to conventional optimization methods like gradient descent, there is no overall loss function being minimized. Still, a solution is likely to be good in part because of built-in random components that help prevent a genetic algorithm from getting mired in less desirable, local results. For these and other more technical reasons, genetic algorithms can be folded into discussions of machine learning. They can also be a key feature of recent developments in deep learning when conventional numerical methods may be overmatched, and local solutions can be a serious pitfall.

[4]Readers interested in running genetic algorithms in R should consider using the library *GA* written by Luca Scrucca (Scrucca 2013).