

# Chapter 11

## Artificial Neural Networks and Deep Learning for Genomic Prediction of Continuous Outcomes



### 11.1 Hyperparameters to Be Tuned in ANN and DL

Successful applications of ANN or DL are related to the user's ability to choose the right hyperparameters. Hyperparameter selection is done with the goal that a model neither underfits nor overfits the training data sets. However, this task is challenging because the number of hyperparameters required in ANN and DL is large. Below we provide a list of the hyperparameters that need to be tuned for implementing ANN and DL models.

1. Network topology
2. Activation functions
3. Loss function
4. Number of hidden layers
5. Number of neurons in each layer
6. Regularization type
7. Learning rate
8. Number of epochs and number of batches
9. Normalization scheme for input data

Next, we provide some general tips for choosing each hyperparameter.

#### 11.1.1 Network Topology

Choosing the neural network topology is a critical step for the successful implementation of ANN and DL models. A safe approach is to choose a feedforward network with a large amount of neurons since there is a lot of empirical evidence that this neural network performs a good optimization process and finds proper weights while at the same time avoiding overfitting. Even though there is empirical evidence that

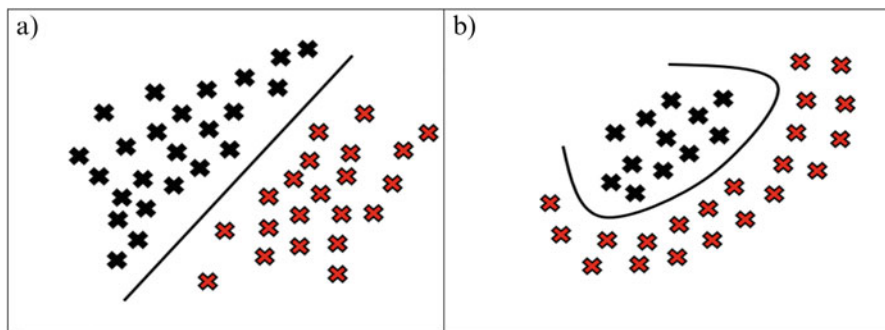
other topologies also work well in some circumstances, the application of DL is still at an early stage (experimental stage) with no definite conclusions. Picking the right network topology is more an art than a science and only practice will make a proper neural network architect (Chollet and Allaire 2017). However, with images data the most powerful DNN are convolutional neural networks, see Chap. 13.

### ***11.1.2 Activation Functions***

The principal rule for choosing activation functions for output layers depends on the type of response variable at hand. For this reason, for binary data, the best option is the sigmoid activation function, for ordinal or multiclass classification problems, the softmax activation function, and for continuous data, the linear activation function is suggested. However, even though there are no clear rules for choosing the activation functions for hidden layers, there is a lot of evidence that using sigmoid activation functions is not a good choice for hidden layers since sigmoid functions very often discard information due to saturation in both forward and backpropagation. RELU activation functions are the most used for hidden neurons, as can be seen in modern deep learning networks today, but RELU activation functions have the problem that some neurons may never activate across the entire training data set. For this reason, the suggestion is to use the leaky RELU activation function in hidden layers which have no zero gradient for all input values (Patterson and Gibson 2017).

### ***11.1.3 Loss Function***

Choosing the right loss function is a key component for the successful implementation of ANN and DL models. Next are given some general rules for selecting the right loss function: (a) when the response variable is continuous, use the sum of squares error loss as the primary option and the sum of absolute percentage error loss as the secondary option; (b) when the response variable is binary, use the hinge loss or logistic (cross-entropy) loss; (c) however, if you are interested in the probability of success of each individual, the logistic loss is the best option; and (d) for categorical responses, the categorical cross-entropy loss is the most used, but when the response variable is a count, a reasonable choice is the Poisson loss or negative binomial loss.



**Fig. 11.1** Schematic representation of linearly and nonlinearly separable patterns. (a) Linearly separable patterns. (b) Nonlinearly separable patterns

**Table 11.1** Guide for determining the number of hidden layers

Number of hidden layers	Result
None	Only capable of representing linear separable functions or decisions
1	Can approximate well any functions that contain continuous mapping from one finite space to another
2	Represent a decision boundary to arbitrary accuracy with rational activation functions and can approximate any smooth mapping to any degree of accuracy

### 11.1.4 Number of Hidden Layers

Selecting the number of hidden layers is a very important issue in the configuration process of DL models. According to the universal approximation theorem (Cybenko 1989), a network with one hidden layer and a large number of neurons is enough to approximate any arbitrary function to the desired degree of accuracy. However, the reality is that many times the required number of neurons is so large that its implementation is not practical. Also, there is a lot of evidence that more than one hidden layer is required when the data are not separated linearly (Fig. 11.1b). Therefore, if your data are linearly separable (Fig. 11.1a) (which you often unknown by the time you begin coding an ANN or DL model), then you don't need any hidden layers at all. Of course, you don't need an ANN or DL model to resolve your data either, but it will still do the job. One hidden layer is sufficient for most problems.

In general, neural networks with two hidden layers can represent functions with any kind of shape. There are currently no strong theoretical reasons to use neural networks with any more than two hidden layers, but empirical evidence shows that using more than two hidden layers can capture in a better way non-linear patterns and complex interactions (Chollet and Allaire 2017). Further, for many practical problems, there's no reason to use any more than one hidden layer. Problems that require two hidden layers are not very common. Differences between the number of hidden layers are summarized in Table 11.1.

### ***11.1.5 Number of Neurons in Each Layer***

The number of neurons in the network is very important since using too few neurons in the hidden layers will result in underfitting, while using too many neurons in the hidden layers can result in severe overfitting. For the input layer, the number of neurons is predetermined by the number of input features (predictors) of the input data. Also, in the output layer, the number of neurons is determined by the number of response variables (outputs) to be predicted or by the number of classes of the response variable when it is ordinal or categorical. For example, if we want to predict grain yield, a continuous outcome, using information of 3000 markers as input variables, this means that the input layer should have 3000 neurons and the output layer just 1, since we are only interested in one response variable that is continuous. However, if instead of predicting a continuous outcome we are interested in predicting an categorical output with ten categories, in this case the required number of neurons in the output layer should be equal to the number of categories in the categorical response variable. But defining the number of neurons in the hidden layers is challenging, and it is left to the user to decide which number to use prior to training the model. According to the literature review we performed, we found that there is no unique and reliable rule for determining the required number of neurons in the hidden layers, since this number depends on (a) the number of input neurons; (b) the amount of training data; (c) the quality of the training data; and (d) the complexity of the learning task (Lantz 2015).

As a general rule, the more complex the network topology (with more neurons, layers, etc.), the more powerful the network for learning more complex problems. That is, the more neurons and layers are used, the better the representation of the training data, which means that the predicted values should be very close to the true values observed in the training set; however, this runs the risk of overfitting and may generalize poorly to new data. We also need to be aware that increasing the complexity of the network also implies that more computational resources are required for its implementation (Lantz 2015). However, as pointed out before, there is empirical evidence that only one hidden layer with a large number of neurons is enough even for complex applications, since increasing the number of layers increases the computational resources required by the complexity of the neural network. For this reason, in general, increasing the number of neurons instead of the number of layers is suggested, for example, if we have a neural network with 100 neurons in the single hidden layer, computationally it is better to increase to 200 neurons in the same hidden layer than to add a second hidden layer with 100 neurons. Of course, this suggestion is not always valid since there are many applications in the fields of pattern recognition and image classification, among others, where to be able to create adequately complex decision boundaries. It is very important to add more hidden layers to better capture the complexity of the phenomenon we want to predict.

Given the difficulties mentioned above, there are two very general approaches for specifying the required number of neurons in the hidden layers. The first one, called

the *backward* approach, suggests starting with a very large number of neurons and evaluating their performance, and then decreasing the number of neurons until there is no more gain in the reduction of the testing error. The second one, called the *forward* approach, suggests starting with half (50%) of the input neurons and then increasing the number of neurons until no significant gain in the prediction performance of sample data is observed. Since there is no consensus about how to choose the required number of hidden neurons, next we provide some general rules: (a) the number of hidden neurons should be in the range between the size of the input layer and the size of the output layer; (b) the number of hidden neurons should be  $2/3$  of the input layer size, plus the size of the output layer; (c) the number of hidden neurons should be less than twice the input layer size; (d) Hecht-Nielsen (1987) suggested using  $2P + 1$  hidden neurons (where  $P$  is the number of inputs) for a network with one hidden layer to compute an arbitrary function; (e) Lippmann (1987) suggested that the maximum number of neurons required for the single hidden layer should be equal to  $L \times (P + 1)$ , where  $L$  is the number of output neurons; and (f) for a single hidden layer classifier, an upper bound is about  $2P + 1$  hidden neurons, whichever is larger.

But in practical terms, the optimal size of the hidden layer is usually between the size of the input and size of the output layers. In some problems, one could probably achieve a decent performance (even without a second optimization step) by setting the hidden layer configuration following just two rules:

1. the number of hidden layers is equal to one and
2. the number of neurons in that layer is the mean of the neurons in the input and output layers.

These rules are only starting points that you may want to consider. Ultimately, the selection of the neurons of your neural network will come down to trial and error following a backward or forward approach. Of course, the idea is to use as few neurons as possible that provide good performance in validation data sets. In general, rules (e) and (f) many times are conservative given that fewer neurons are often sufficient, since in a lot of cases, using only a small number of hidden neurons in the artificial neural network offers a tremendous amount of learning ability. Also, we need to recall that according to the universal approximation theorem, an artificial neural network with at least one hidden layer with a lot of neurons is a good universal function approximator (Lantz 2015).

### 11.1.6 Regularization Type

The main goal of regularization is to avoid overfitting, and the basic idea is to penalize the weights in such a way that the model increases its prediction performance in out-of-sample data. The most common methods are the L2 (Ridge penalization), L1 (Lasso penalization), L2-L1 (Elastic Net penalization), and the dropout method. The L1 regularization forces many weights to be exactly zero, which

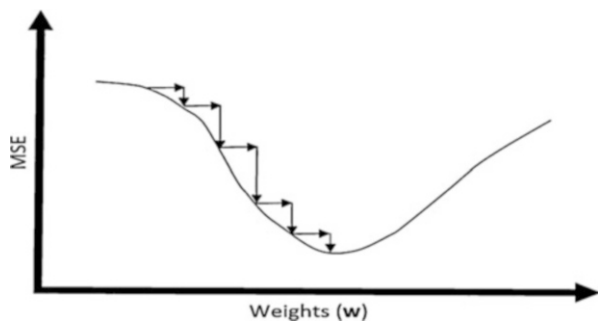
provides a sparse solution, while with L2 regularization, the larger the weight, the stronger the penalization. The L2-L1 is somewhere between the L2 and L1 regularization, while the dropout method fixes a certain percentage of weights to be exactly zero in the input or hidden neurons. However, it is not easy to suggest one of them, since all of them do a reasonable job in particular contexts, but any of them works well for all types of data. But, in general, the L2 and dropout regularizations perform well in many applications.

### 11.1.7 Learning Rate

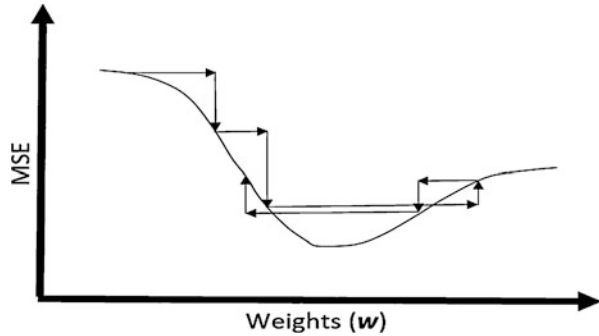
The learning rate modifies weights and thresholds (bias) of the neural network, in order for a given input to the network to produce a favored output. It also determines the speed for convergence of the backpropagation algorithm to an optimal solution. Typically, a value of 0.001 or less is used to initialize the learning rate; however, in general, it is a hard task to choose the appropriate learning rate. Before explaining the problems involved in choosing the learning rate, we will first mention that the backpropagation algorithm can stop because (1) the number of maximum epochs allowed was reached and (2) an optimal solution was found (Baysolow II 2017). Figure 11.2 illustrates how to reach an optimal solution with an optimal learning rate.

Next, we explain the consequences of choosing a learning rate that is too small or too large. When the learning rate is very small, it is more likely to stop the backpropagation algorithm in a non-optimal condition since the first stopping condition is satisfied. Although apparently this can be fixed by increasing the number of epochs, this solution is not optimal since increasing the number of epochs considerably increases the required computation resources. On the other hand, when the learning rate is very large, it is possible not to reach the optimal solution due to the fact that the loss function in each epoch may overcorrect and give updated values for the coefficient that are far too small or far too large (Figs. 11.3 and 11.4). For this reason, the suggestion is to try several learning rate values and see how the algorithm performs across each epoch (Baysolow II 2017).

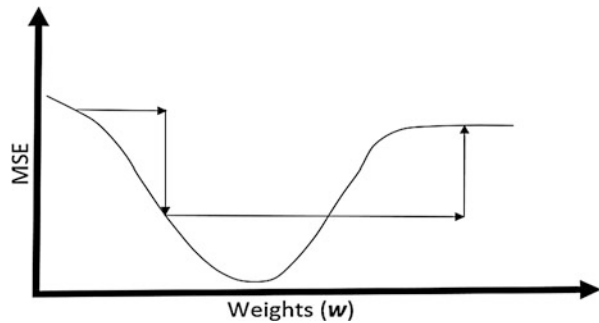
**Fig. 11.2** Optimal learning rate for efficient error minimization (Samarasinghe 2007)



**Fig. 11.3** The effect of a high learning rate on the learning process (Samarasinghe 2007)



**Fig. 11.4** The effect on the learning process of a learning rate that is too high. (Samarasinghe 2007)



### 11.1.8 Number of Epochs and Number of Batches

Before defining an epoch, we will define a sample (individual) as a single row of data that contains inputs that are fed into the algorithm and an output (prediction) that is compared to the true value to calculate an error. The number of epochs is a hyperparameter that defines the number of times that the learning algorithm will work across the entire training data set. One epoch means that each sample in the training data set has had an opportunity to update the internal model parameters. An epoch is comprised of one or more batches. For example, assume you have a data set with 600 samples (rows of data) and you choose a batch size of 15 and 500 epochs. This means that the data set will be divided into 40 batches, each with 15 samples. The model weights will be updated after each batch of 15 samples. This also means that one epoch will involve 40 batches or 40 updates of 15 samples to be completed. With 500 epochs, the model will be exposed to or will pass through the whole data set 500 times. That is, a total of  $40 \times 500 = 20,000$  batches during the entire training process. Therefore, with this example, it is clear that the number of batches in each epoch is equal to the number of samples in the data sets divided by the batch size. To better understand these two hyperparameters, you can think of a for-loop over the number of epochs where each loop proceeds over batches of the training data set. Within this for-loop is another nested for-loop that iterates over each batch of samples, where one batch has the specified “batch size” number of samples.

The number of epochs is traditionally large, often hundreds or thousands, allowing the learning algorithm to run until the error from the model has been sufficiently minimized. You may see examples of the number of epochs in the literature and in tutorials set to 10, 100, 500, 1000, and larger. This means that the number of epochs can be set to an integer value between one and a very large value. You can run the algorithm for as long as you like and even stop using other criteria besides a fixed number of epochs, such as a change (or lack of change) in the model error over time. The more complex the features and relationships in your data, the more epochs you'll require for your model to learn, adjust the weights, and minimize the loss function. With regard to the batch size, that is, the number of samples processed before the model is updated, this hyperparameter should be more than or equal to one and less than or equal to the number of samples in the training data set. Popular batch sizes include 32, 64, 128 samples and so on.

It is common to create line plots that show epochs along the  $x$ -axis as time, and the error or skill of the model on the  $y$ -axis. These plots were mentioned in Chap. 10 and were called learning curves under the *early stopping rule*. These plots can help to diagnose whether the model has over learned, under learned, or is suitably fit to the training data set. Since there are no magic rules on how to configure these parameters, you must try different values and see what works best for your problem.

### ***11.1.9 Normalization Scheme for Input Data***

It is generally good practice to scale input variables to have the same scale. When input variables have different scales, the scale of the weights of the network will, in turn, vary accordingly. This introduces a problem when using weight regularization because the absolute or squared values of the weights must be added for use in the penalty. Also, without normalization, the input variable with greater numeric range has larger impact than the one having less numeric range, and this could, in turn, impact prediction accuracy. This problem can be addressed by the following methods applied to each input variable:

**Standardizing.** Subtract the mean and divide by the standard deviation. Many times it is good practice to only divide each input by the standard deviation without centering.

**Min-max normalization.** Subtract from each observation the min value of the input and divide this by the range (max value-min value) of the input. The disadvantage of the min-max normalization technique is that it tends to bring the data toward the mean. Details of this method and other methods of scaling were provided in Chap. 2 of this book.



## 11.2 Popular DL Frameworks

Given that the adoption of DL has proceeded at an alarming pace, the maturity of the ecosystem has also shown phenomenal improvement. Thanks to many large tech organizations and open-source initiatives, we now have a plethora of options to choose from. Due to the evolution of the software industry, today it is far easier to develop high-end software than it was a few years back, thanks to the available tools that have automated complex problems in a way that's simple to use.

Given the maturity of software tools available today, we can afford to automate several complexities that happen in the background. These tools are nothing but building blocks for software systems. You technically don't need to start from scratch; you can instead rely on available powerful tools that have matured significantly to take care of several software-building services. Given the level of abstraction a framework provides, we can classify it as a:

- (a) Low-level DL framework and
- (b) High-level DL framework

A few of the popular low-level frameworks for DL are Theano, Torch, PyTorch, MxNet, TensorFlow, etc. The previously mentioned frameworks can be defined as the first level of abstraction for DL models. You would still need to write fairly long codes and scripts to get your DL model ready, although much less than using just Python or C++. The advantage of using the first-level abstraction is the flexibility it provides in designing a model. However, to simplify the process of DL models, we have frameworks that work on the second level of abstraction called High-Level DL Frameworks. Rather than using the previously mentioned frameworks directly, we can use a new framework on top of an existing framework and thereby simplifying DL model development even further. The most popular high-level DL framework that provides a second-level abstraction to DL model development is Keras. Other frameworks like Gluon, Lasagne, and so on are also available, but Keras has been the most widely adopted one.

Keras is a high-level neural network API written in Python, and it can help you in developing a fully functional DL model with few lines of code. Since it is written in Python, it has a larger community of users and supporters and is extremely easy to get started on. The simplicity of Keras is that it helps users to quickly develop DL models and provides a ton of flexibility while still being a high-level API. This really makes Keras a special framework to work with. By far the most widely adopted usage of Keras is with TensorFlow as a back end (i.e., Keras as a high-level DL API and TensorFlow as its low-level API back end). In a nutshell, the code you write in Keras gets converted to TensorFlow, which then runs on a computing instance. You can read more about Keras and its recent developments here: <https://keras.io/>.

For this reason, in this book, we use Keras for implementing ANN and DL models (Allaire and Chollet 2019). The examples that follow use this framework.

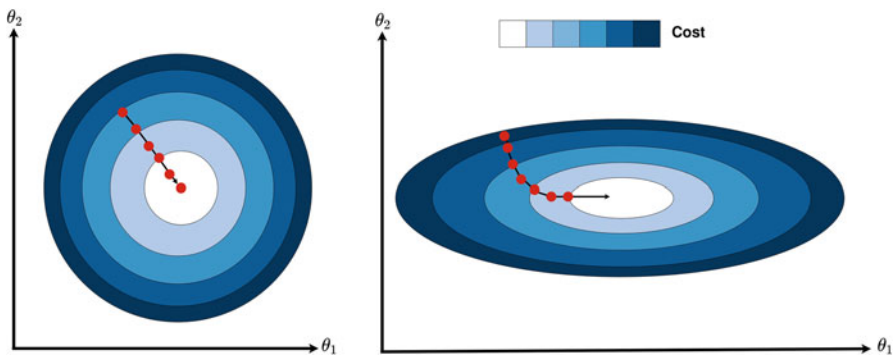
### 11.3 Optimizers

The optimizer is a key component of the model training process. Up to this point, we are only familiar with the backpropagation algorithm that provides feedback to the model to reach a good trained process. The loss function of ANN is the main element to understand how well or poorly the current set of weights has performed on the training sample. The next step for the model is to reduce the loss. How does it know what steps or updates it should perform on the weights to reduce the loss? As we observed in the previous chapter, the optimizer function is a mathematical algorithm that uses calculus (derivatives and partial derivatives) to estimate how much change the network will need in the loss function to be able to reach the minimum. The change in the loss function, which would be a decrease or increase, helps in finding the direction of the change required in the weight of the connection. The math equations required are specific for each optimizer and are beyond the scope of this book (Géron 2019), but the general issues are those provided in Chap. 10 for the back propagation algorithm.

*Batch Gradient Descent (BGD)*. The implementation of BGD requires training the model with the full data sets at every step. As a result, it is considerably slow on very large training sets. The BGD will eventually converge to the optimal solution for convex functions with a fixed learning rate, but requires a considerable amount of time (Géron 2019). Under BGD, the cost function has the shape of a bowl when the inputs have the same scale, but an elongated bowl with different scales of the features is also possible. Figure 11.5 shows gradient descent on a training set where features 1 and 2 have the same scale (on the left), and on a training set where feature 1 has much smaller values than feature 2 (on the right).

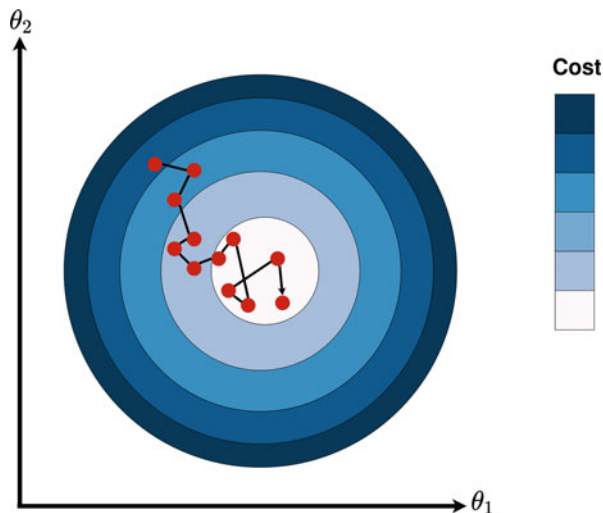
Other optimizers based on gradient descent available in Keras are listed next.

*Stochastic Gradient Descent (SGD)*. SGD, at the extreme, simply chooses a random instance in the training set at each step and calculates the gradients only in that single instance, that is, SGD uses `batch_size = 1` in the model training function.



**Fig. 11.5** Gradient descent with and without input (feature) scaling (Géron 2019). On the left are shown the two features on the same scale that reach the target faster, while on the right the two features have different scales and take longer to reach the target

**Fig. 11.6** Stochastic gradient descent (Géron 2019)



Obviously, this makes the algorithm much faster, since in each iteration, it has less data to manipulate (Fig. 11.6). To reduce high fluctuations in SGD optimizations, a better approach would reduce the number of affected iterations in a mini-batch. Mini-batch gradient descent is a modification of the gradient descent method that splits the training data set into small batches that are used to calculate model prediction error and update model coefficients. This approach has been more successful and results in a more fluid training process. The batch size is usually set to powers of 2 (i.e., 32, 64, 128, etc.) (Géron 2019).

*Adaptive Moment Estimation (Adam).* This is by far the most popular and widely used optimizer in DL. In most cases, you can blindly choose the Adam optimizer and forget about the optimization alternatives. This optimization technique computes an adaptive learning rate for each parameter. It defines momentum and variance of the gradient of the loss and leverages a combined effect to update the weight parameters. The variance and momentum together help smooth the learning curve and effectively improve the learning process.

*Other important optimizers.* There are many other popular optimizers for DL models. We will not discuss all of them in this book. We will list only a few of the most popular optimization alternatives used and available within Keras (Adadelta, RMSProp, Adagrad, Adamax, Nadam). There is no perfect optimization method since each of them has its own pros and cons. Each optimizer requires tunable parameters like learning rate, momentum, and decay. The saddle point and vanishing gradient are two problems often faced in DL. These problems can be explored in more detail while choosing the best optimizer for your problem. But Adam always works fine in most cases (Géron 2019).

## 11.4 Illustrative Examples

### Example 11.1 MaizeToy

In this example, we use the MaizeToy data set. This data set is preloaded in the BMTME library, and for this reason it is important to first install this library and also the Keras library. To see the information of this data set, we can use:

```
rm(list=ls())
library(keras)
library(BMTME)
#####Set seed for reproducible results#####
use_session_with_seed(45)
#####Loading the MaizeToy data sets#####
data("MaizeToy")
ls()
head(phenoMaizeToy)
```

Using this code, we can see in the console that this data set has two objects: “genoMaizeToy” and “phenoMaizeToy.” The first contains the genomic relationship matrix (GRM) of the lines, while the second contains the phenotypic information for which the first six observations resulting from using the head() command are shown:

```
> head(phenoMaizeToy)
  Line Env   Yield  ASI  PH
1 CKDHL0008 EBU   6.88 2.7 226
2 CKDHL0008 KAK   4.83 1.7 227
3 CKDHL0008 KTI   4.84 3.0 240
4 CKDHL0039 EBU   6.85 1.3 239
5 CKDHL0039 KAK   5.23 0.7 212
6 CKDHL0039 KTI   6.18 2.0 238
```

Here we can observe that in this data set, the first column contains the names of the lines, the second contains the name of the environments, while columns 3–5 contain the measured traits: grain yield (Yield), anthesis-silking interval (ASI), and plant height (PH).

Next we order the data, first in terms of environments and then in terms of lines inside each environment, using the following commands:

```
#####Ordering the data #####
phenoMaizeToy<- (phenoMaizeToy[order(phenoMaizeToy$Env,
phenoMaizeToy$Line),])
rownames(phenoMaizeToy)=1:nrow(phenoMaizeToy)
head(phenoMaizeToy,5)
```

The output of five observations using the head() function is equal to:

```
> head(phenoMaizeToy, 5)
  Line Env  Yield ASI  PH
1 CKDHL0008 EBU  6.88 2.7 226
2 CKDHL0039 EBU  6.85 1.3 239
3 CKDHL0042 EBU  6.37 2.3 238
4 CKDHL0050 EBU  4.98 3.1 239
5 CKDHL0060 EBU  7.07 1.4 242
```

Here it is clear that the data are ordered first in terms of environments and then in terms of lines inside each environment. Next, we build the design matrices required to implement a model with a predictor with information of environments, lines, and genotype by environment interaction. To do this, the following code can be used:

```
#####Design
matrices#####
LG <- cholesky(genoMaizeToy)
ZG <- model.matrix(~0 + as.factor(phenoMaizeToy$Line))
Z.G <- ZG %*% LG
Z.E <- model.matrix(~0 + as.factor(phenoMaizeToy$Env))
ZEG <- model.matrix(~0 + as.factor(phenoMaizeToy$Line):as.factor
(phenoMaizeToy$Env))
G2 <- kronecker(diag(length(unique(phenoMaizeToy$Env))), data.matrix
(genoMaizeToy))
LG2 <- cholesky(G2)
Z.EG <- ZEG %*% LG2
```

It is important to point out that the design matrices are built using the model matrix command of R that is specialized for this task, but the resulting design matrices of lines and genotype by environment interaction are post-multiplied by the Cholesky decomposition of the GRM of lines or genotype  $\times$  environment. In the next code, we select the available response variables and create the training-testing set using a CV strategy with five-fold.

```
#####Selecting the response variable#####
Y <- as.matrix(phenoMaizeToy[, -c(1, 2)])
###Training-testing sets using the BMTME package#####
pheno <- data.frame(GID = phenoMaizeToy[, 1], Env = phenoMaizeToy[, 2],
  Response = phenoMaizeToy[, 3])
CrossV <- CV.KFold(pheno, DataSetID = 'GID', K = 5, set_seed = 123)
```

The Y matrix contains information of the three traits available in the MaizeToy data sets. The pheno variable is an auxiliary object that contains information of lines, information of environments, and the response variable of only one trait to be able to create a training-testing partition according to a five-fold cross-validation strategy where information of some lines is missing in some environments but not in all. The

CrossV\$CrossValidation\_list argument was used to print the testing observations for each fold.

```
> CrossV$CrossValidation_list
$partition1
 [1] 88 79 59 67  3 36 90 14 23 78 10 27 26 24 74  9 13 72

$partition2
 [1] 12 39 30 69 41 65 83 63 87 57 48 53  2 38 20 31 43 29

$partition3
 [1] 89 80 75 17 11 54 33 52 28 84 56  7 35 37 42 21 34 70

$partition4
 [1] 81 58 44  8 73 19  4 66 51  6 86 60 40 76 61 16 15 64

$partition5
 [1] 45 82 18  5 71 47 46 49 85 50 25 55 62 22 32 68  1 77
```

We used the following code to select only one trait to be our response variable (Yield), and we also collected in an X matrix the information of environments, lines, and genotype by environment that will be the input of our artificial deep neural network model:

```
#####Part 0: Final X and y before splitting#####
y=(phenoMaizeToy[, 3])
length(y)
X=cbind(Z.E,Z.G,Z.EG)
dim(X)
```

The output of this code shows that the total number of observations is 90, that is, 30 for each environment, and the number of input variables (predictors) is 123, which corresponds to the number of columns in matrix X. For the moment, we will only work with one-fold of the five training–testing sets, and we will also specify the number of epochs and units (neurons) used to implement our first artificial deep neural network model in Keras.

```
#####One holdout cross-validation#####
tst_set=CrossV$CrossValidation_list[[2]]
No_Epoch=1000
N_Units=33
X_trn=X[-tst_set,]
X_tst=X[tst_set,]
y_trn=y[-tst_set]
y_tst=y[tst_set]
```

Before specifying the model, we will explain the pipe operator (`%>%`) that connects functions or operations together in the Keras library. We will be using

the pipe operator (`%>%`) to add layers to the network. The pipe operator allows us to pass the value on its left as the first argument to the function on its right.

Next, we define the *artificial neural network model* and *compile* it in Keras:

```
build_model <- function() {
  model <- keras_model_sequential()
  model %>%
    layer_dense(units = N_Units, activation = "relu", input_shape = c(dim
  (X_trn)[2])) %>%
    layer_dropout(rate = 0.0) %>%
    layer_dense(units = 1, activation = "linear")

  model %>% compile(
    loss = "mse",
    optimizer = "rmsprop",
    metrics = c("mse"))
  model}

model <- build_model()
model %>% summary()
```

The first part of this code specifies the type and topology of the model. With `keras_model_sequential` we are using Keras to implement a feedforward (or densely connected) network. The sequential model is composed of a linear stack of layers. Layers consist of the input layer, hidden layers, and an output layer. Sequential models are appropriate when the information available is in two dimensions: one dimension for samples (observations or individuals) and the other for predictors (features), since the available data set is a matrix where the rows correspond to observations and the columns correspond to the features at hand. The first `layer_dense` is used to specify the first hidden layer and also the required number of neurons (units), and in the `input_shape` is specified the dimension of the input variables (in this case, it is equal to the number of columns in the training set of the predictors, that is, 123). Also specified here is the linear activation function which gives as output positive values. Next in `layer_dropout()` we are instructing Keras that any of the weights of the neurons of the first hidden layer will be set to zero, that is, all specified neurons will be used since `rate = 0`. Then the second `layer_dense()` is used to specify the output layer, which only used one neuron since the response variable under study is continuous and unique (univariate); in this layer we used the linear activation function, which is a good choice for continuous outputs. An input shape argument is not required in the second or other hidden layers since its input shape is automatically inferred as being the output of the previous layer.

Next, the compilation step is done in which the learning process of the network is configured and where you should specify the optimizer, the loss function(s), and the metric used for monitoring the match between observed and predicted values. In this case, for the optimizer we specified “rmsprop,” which is a gradient-descent-based algorithm that combines Adagrad and Adadelta adaptive learning ability. Adagrad is a gradient-descent-based algorithm that accumulates previous costs to do adaptive

learning. Adadelta is a gradient-descent-based algorithm that uses Hessian approximation to do adaptive learning. The loss function and metric used in this case is the mean squared error (mse) since the response variable is quantitative. The last two lines of this code provide a summary of the model that will be implemented.

```
> model <- build_model()
> model %>% summary()
```

---

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 33)	4092
dropout_1 (Dropout)	(None, 33)	0
dense_2 (Dense)	(None, 1)	34

---

```
Total params: 4,126
Trainable params: 4,126
Non-trainable params: 0
```

---

From this summary output of the neural model that will be implemented, we can see that there are 33 neurons (since we decided to use 33 neurons) in the hidden layer, and any of their weights will be fixed at zero since the dropout percentage is zero. For this reason, the number of parameters (weights) in the first hidden layer is equal to  $123 \times 33 + 33 = 4092$ , and to 34 in the second layer since there are 34 weights that connect from the hidden layer to the output layer; this corresponds to 33 units plus an intercept (bias) term. For this reason, the total number of parameters to estimate is equal to 4126.

We then provide the code for training (fitting) the model and also specify a plot to visualize training and validation metrics by epoch:

```
print_dot_callback <- callback_lambda(
  on_epoch_end = function(epoch, logs) {
    if (epoch %% 20 == 0) cat ("\n")
    cat (". ")
  })
#####Internally this part of the code split the training set in
training-inner and validation###
model_fit <- model %>% fit(
  X_trn, y_trn,
  shuffle=F,
  epochs=No_Epoch, batch_size=72,
  validation_split=0.2,
  verbose = 0,
  callbacks = list(print_dot_callback))
#####Plot history#####
plot(model_fit)
```

(continued)



(continued)

The first part of the code is a custom callback function for replacing the default training output with a single dot per epoch, while the second part is for fitting the artificial neural model and storing training stats. Also, in this second part of the code is specified the input and output training information, the number of epochs and batch size, and the size of the inner validation set, which in this case was 0.2 or 20%. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on this data at the end of each epoch. The validation data are selected from the last samples in the X and y data provided, before shuffling (permuting the data). The goal is to shuffle the training data before each epoch when TRUE is specified, but not when FALSE is specified. The plot of model\_fit is for visualizing the model's training progress using the metrics stored in the model\_fit variable. We want to use these data to determine how long to train before the model stops making progress.

In Fig. 11.7, the mean square error is plotted in the bottom panel, and the loss in the top panel. In this plot you can see that after 250 epochs, there is no relevant improvement in the prediction performance of the validation set. Next, we calculate the MSE for the training set; the predictions and MSE of the testing set were obtained using the following code:

```
#####MSE for No_epochs=1000 in the outer testing set#####  
model_fit$metrics$val_mse [No_Epoch]  
pred=model%>% predict(X_tst)  
Pred=c(pred)  
Obs=y_tst  
MSE_First=mean((Obs-Pred)^2)  
MSE_First
```

The first part of the code extracts the MSE for the training set implemented with 1000 epochs, which is equal to 4.5429; the second manually calculates the MSE for the testing set, which is equal to 1.8688. Next, we refit the model but using the *early*

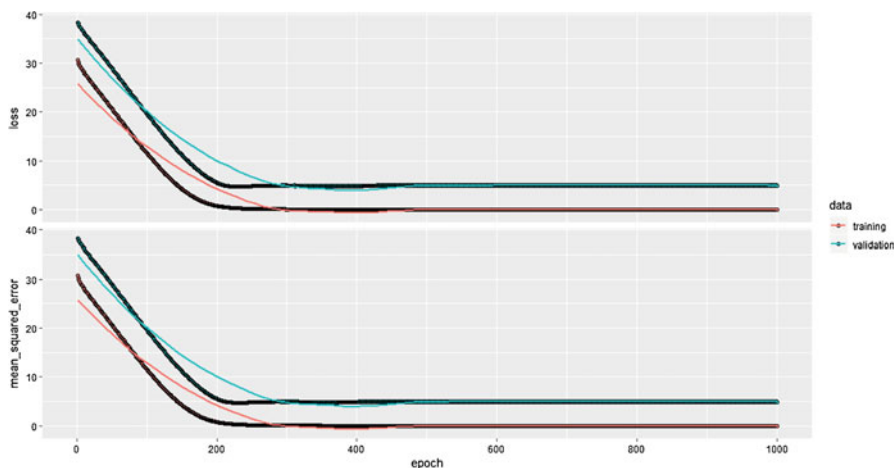


Fig. 11.7 Training and validation metrics

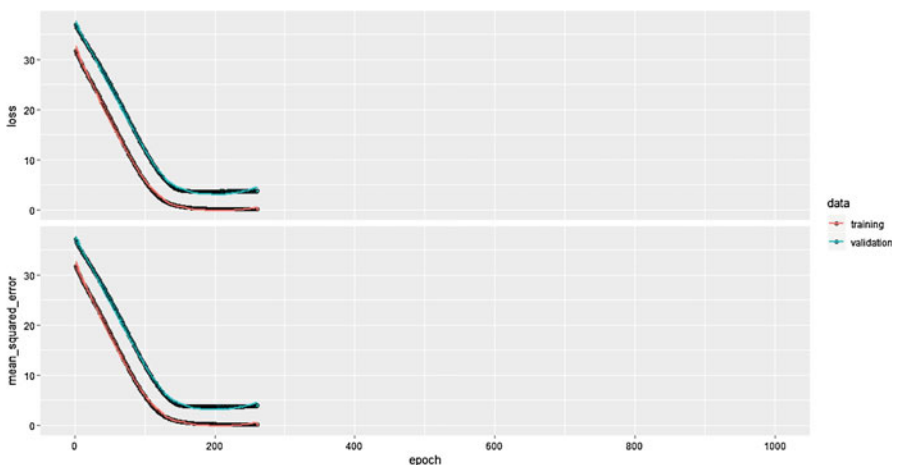
*stopping approach* to stop the training process when there is no more improvement in terms of prediction accuracy.

```
#####Fitting the model with early stopping#####
##### The patience parameter is the amount of epochs to check for
improvement.
early_stop <- callback_early_stopping(monitor = "val_loss",
mode='min', patience =50)

model_Final<-build_model()
model_fit_Final<-model_Final %>% fit (
  X_trn, y_trn,
  shuffle=F,
  epochs =No_Epoch, batch_size =72,
  validation_split = 0.2,
  verbose=0, callbacks = list(early_stop,print_dot_callback))

#####Plot history #####
length(model_fit_Final$metrics$mean_squared_error)
plot(model_fit_Final)
```

The first part of this code is a function that will help to implement the early stopping process in Keras, and the function to monitor is the validation loss score. The patience parameter is the number of epochs with no improvement, after which training will be stopped. The second part is exactly the same as the one used before for fitting the model, with the difference that now the training process will stop according to the information provided in the early stopping function. The code `length(model_fit_Final$metrics$mse)` prints the number of epochs during the stopping process. Finally, the history of the training and validation process is depicted in this plot (Fig. 11.8).



**Fig. 11.8** Training and validation metrics under the stopping approach

In the plot it is clear that the stop occurred in epoch 282, which means that the training process was done with this number of epochs and not with 1000 as before. Finally, the following code was used to obtain the predicted values and MSE for the testing set using the training information with 282 epochs. Two MSE were also printed, one using the original 1000 epochs (MSE\_First) and the last one with 282 epochs.

```
#####Prediction of testing set#####
predicted=model_Final %>% predict(X_tst)
Predicted=c(predicted)
Observed=y_tst
plot(Observed,Predicted)
MSE=mean((Observed-predicted)^2)
MSE
Obs_Pred=cbind(Observed,predicted)
colnames(Obs_Pred)=c("Observed","Predicted")
Obs_Pred

#####Results with and without early stopping MSE and MSE_First##
MSE_First
MSE
```

Finally, this code, in addition to providing the observed and predicted values, of which we show the first six observations, also provides the MSE of both training processes:

```
> head(Obs_Pred)
      Observed Predicted
[1,]  7.55      5.824002
[2,]  5.91      4.806885
[3,]  5.62      4.344528
[4,]  5.39      4.817317
[5,]  5.23      4.776768
[6,]  6.39      5.039447
>
>
> #####Results with and without early stopping are printed as MSE and
MSE_Firt, respectively ##
> MSE_First
[1] 1.868842
> MSE
[1] 1.806609
```

From this output it is clear that when the training process was done with the *early stopping approach*, the MSE of the testing set was better by 3.4% (even though maybe it is not so relevant since the predictions are still quite far from the true values), than when using all the epochs originally used. However, it is important to point out that the early stopping method did not always improve the prediction performance.

### Example 11.2 MaizeToy Example with a Tuning Process

The last example illustrated how to use the early stopping rule to train ANN or DL models. However, that example only used one partition of the data sets (one TRN with 80% and one TST with 20%) and one value of neurons. However, if another TRN–TST split and number of neurons are used, the expected MSE will be different. For this reason, next we split the training into five TRN–TST inner subsets and also train the ANN model with a grid of two values for the neurons and two dropout values.

Tuning a DNN model often requires exploring many hyperparameters to find the best combination that produces the best out-of-sample predictions. The best way to approach this in Keras is by defining external flags (a map of the model parameter name and an arrangement of values to try) for key parameters, which you may want to vary instead of progressively changing your source code of the training script each time. In other words, this is a way in Keras for specifying a grid search. The `flags()` function provides a flexible mechanism for defining flags and varying them across training runs. For this reason, for tuning hyperparameters in Keras, we will now use the `tfruns` package (Allaire 2018), which uses flags and provides a lot of flexibility for tracking, visualizing, and managing training runs and experiments in R. This package allows tracking the metrics, hyperparameters, source code of every training run, and output, and visualizes the results of individual runs and comparisons between runs. For this reason, first we created the Keras script with flags called `Code_Tuning_With_Flags_00.R`, as follows:

```
####a) Declaring the flags for hyperparameters
FLAGS = flags(
  flag_numeric("dropout1", 0.05),
  flag_integer("units", 33),
  flag_string("activation1", "relu"),
  flag_integer("batchsize1", 56),
  flag_integer("Epoch1", 1000),
  flag_numeric("learning_rate", 0.01),
  flag_numeric("val_split", 0.2),
  flag_numeric("reg_l1", 0.001)

####b) Defining the DNN model
build_model<-function() {
model <- keras_model_sequential()
model %>%
  layer_dense(units =FLAGS$units, activation =FLAGS$activation1,
  input_shape = c(dim(X_trII) [2])) %>%
  layer_dropout (rate=FLAGS$dropout1) %>%
  layer_dense (units=1, activation ="linear")

####c) Compiling the DNN model
model %>% compile(
  loss = "mse",
  optimizer =optimizer_adam(FLAGS$learning_rate),
  metrics = c("mse"))
model}
```

```

model<-build_model()
model %>% summary()

print_dot_callback <- callback_lambda(
  on_epoch_end = function(epoch, logs) {
    if (epoch %% 20 == 0) cat ("\n")
    cat (".") })

early_stop <- callback_early_stopping(monitor = "val_loss",
mode='min',patience =50)

#####d) Fitting the DNN model#####
model_Final<-build_model()
model_fit_Final<-model_Final %>% fit (
  X_trII, y_trII,
  epochs =FLAGS$Epoch1, batch_size =FLAGS$batchsize1,
  shuffled=F,
  validation_split = FLAGS$val_split,
  verbose=0,callbacks = list(early_stop,print_dot_callback))

```

In part a) of the above code, we declare the flags for each of the candidate hyperparameters to be tuned. It is important to point out that the type of flag that is used depends on the type of hyperparameter, and there are four types of flags: a) `flag_integer()` for integer values like 1, 2, 3, 4, etc., b) `flag_numeric()` for floating point values like 2.5, 4.4, -1.23, 1.35, etc., c) `flag_boolean()` for logical values (e.g., true, false, 1, 0), and d) `flag_string()`, for example, character values (e.g., “RELU”).

Next we incorporate the flag parameters within our model. In part b) of the code, a feedforward neural network is defined, but incorporating the flags for units and hidden activation function allows us to vary the values of these hyperparameters and not only work with the literal values of these hyperparameters. The definition of the DNN model differs from the previous DNN model in the use of flags. In part c), the compilation of the DNN model is done using flags and in this part the only flag used was for the learning rate. Finally, in part d), the model is fitted; flags were also used here but now for epochs, batch\_size, and validation\_split.

Next we created a grid of hyperparameters and we called the last R code (Code\_Tuning\_With\_Flags\_00.R) to be able to vary each flag with a grid of values using the `tfruns::tuning_run()` function where the grid of values for each of the hyperparameters is specified. The following code performs a tuning process for choosing the optimal hyperparameters. The data used here are the same as before (MaizeToy data sets), but it is important to point out that to be able to use this code, first all the codes that were used in Example 11.1 should be used, until part 0: Part 0: Final X and y before splitting.

```

#####a) Inner cross-validation#####
nCVI=5 ###Number of folds for inner CV
Hyperpar=data.frame()
for (i in 1:nCVI) {
  Sam_per=sample(1:nrow(X_trn),nrow(X_trn))

```

```
X_trII=X_trn[Sam_per,]
y_trII=y_trn[Sam_per]
```

```
#####b) Grid search using the tuning_run() function of tfruns
package#####
```

```
runs.sp<-tuning_run("Code_Tuning_With_Flags_00.R",runs_dir =
'_tuningE1',flags=list(dropout1=c(0,0.05),
                        units = c(33,67),
                        activation1=("relu"),
                        batchsize1=c(28),
                        Epoch1=c(1000),
                        learning_rate=c(0.001),
                        val_split=c(0.2)),sample=1,confirm =
FALSE,echo =F)
```

```
#####c) Saving each combination of hyperparameters in the Hyperpar
data.frame
```

```
#### Ordering in the same way all grids
runs.sp=runs.sp[order(runs.sp$flag_units,runs.sp$flag_dropout1),]
runs.sp$grid_length=1:nrow(runs.sp)
Parameters=data.frame(grid_length=runs.sp$grid_length,
metric_val_mse=runs.sp$metric_val_mse,flag_dropout1=runs.
sp$flag_dropout1,flag_units=runs.sp$flag_units,flag_batchsize1=runs.
sp$flag_batchsize1,epochs_completed=runs.sp$epochs_completed,
flag_learning_rate=runs.sp$flag_learning_rate,flag_activation1=runs.
sp$flag_activation1)
Hyperpar=rbind(Hyperpar,data.frame(Parameters))
}
```

```
#####d) Summarizing the five inner fold by hyperparameter combination
Hyperpar %>%
```

```
group_by(grid_length) %>%
summarise(val_mse=mean(metric_val_mse),
dropout1=mean(flag_dropout1),
units=mean(flag_units),
batchsize1=mean(flag_batchsize1),
learning_rate=mean(flag_learning_rate),
epochs=mean(epochs_completed)) %>%
select(grid_length,val_mse,dropout1,units,batchsize1,
learning_rate,epochs) %>%
mutate_if(is.numeric, funs(round(., 3))) %>%
as.data.frame() -> Hyperpar_Opt
```

```
#####e) Selecting the optimal hyperparameters#####
```

```
Min=min(Hyperpar_Opt$val_mse)
pos_opt=which(Hyperpar_Opt$val_mse==Min)
pos_opt=pos_opt[1]
Optimal_Hyper=Hyperpar_Opt[pos_opt,]
#####Selecting the best hyperparameters
Drop_O=Optimal_Hyper$dropout1
Epoch_O=round(Optimal_Hyper$epochs,0)
Units_O=round(Optimal_Hyper$units,0)
activation_O=unique(Hyperpar$flag_activation1)
batchsize_O=round(Optimal_Hyper$batchsize1,0)
lr_O=Optimal_Hyper$learning_rate
```

```

print_dot_callback <- callback_lambda(
  on_epoch_end = function(epoch, logs) {
    if (epoch %% 20 == 0) cat ("\n")
    cat (".") })

#####f) Refitting the model with the optimal values#####
model_Sec<-keras_model_sequential()
model_Sec %>%
  layer_dense(units =Units_O , activation =activation_O, input_shape =
c(dim(X_trn) [2])) %>%
  layer_dropout(rate =Drop_O) %>%
  layer_dense(units =1, activation =activation_O)

model_Sec %>% compile(
  loss = "mean_squared_error",
  optimizer = optimizer_adam(lr=lr_O),
  metrics = c("mean_squared_error"))

ModelFited <-model_Sec %>% fit (
  X_trn, y_trn,
  epochs=Epoch_O, batch_size =batchsize_O, #####validation_split=0.2,
  early_stop,
  verbose=0, callbacks=list(print_dot_callback))

#####g) Prediction of testing set #####
Yhat=model_Sec%>% predict(X_tst)
y_p=Yhat
y_p_tst=as.numeric(y_p)
y_tst=y[tst_set]
plot(y_tst,y_p_tst)
MSE=mean((y_tst-y_p_tst)^2)
MSE

```

In part a) of the above code, the number of inner TRN–TST partitions that will be implemented is specified; in this case, there are five. Then a data.frame called Hyperpar is defined in which the results of each combination of hyperparameters resulting from each inner partition will be saved. Then we start the loop for implementing the five inner CV strategies for selecting the best hyperparameters. Then the outer training set is shuffled (permuted; using sample without replacement) in such a way that the TRN–TST partition constructed internally in Keras using the validation\_set function does not use the same validation and training set in each inner partition.

Part b) of this code specifies the grid for each hyperparameter. It is important to point out that two values were used, 0 (0%) and 0.05 (5%), for dropout. Also, two values, 33 and 67, were used for units, while for the rest of the hyperparameters only one value was used. For this reason, a total of four hyperparameter combinations should be evaluated. It is important to point out that you can put many more values for each hyperparameter, for example, using a grid of 4 values for each hyperparameter implies a total of  $4^7 = 16,384$  combinations since 7 hyperparameters

can be tuned with the flags specified in the above code, which illustrates that the hyperparameter search space explodes quickly with DL models since there are many hyperparameters that need to be tuned. For this reason, the `tuning_run()` function, in addition to implementing by default a full Cartesian grid search (with `sample = 1`), also allows implementing a random grid search by specifying a value between 0 and 1 in the `sample` command inside the `tuning_run()` function. For example, by specifying `sample = 0.25`, only 25% of the total combinations of hyperparameters should be evaluated.

In part c) of the code, first the outputs of the `tuning_run` function are ordered, that were saved in the `runs.sp` object. The ordering should be done for all those hyperparameters that have more than one value for evaluation. We did the ordering process only for `dropout1` and `units` (hyperparameter neurons) that have two values each to be evaluated. However, if more hyperparameters contain more than one value to be evaluated, they should also be ordered. Then we added to the `runs.sp` data frame the unique consecutive number for each unique combination of hyperparameters under study. Then in `parameters`, in addition to the grid length, the important hyperparameters are saved for each inner fold in the `Hyperpar` data frame. In part d) the average prediction performance is obtained for each of the four hyperparameters under study in this problem. In e) the best combination of hyperparameters is selected. Then in part f), the model with the optimal hyperparameters is refitted, and finally in part g), the predictions of the second-order cross-validation are obtained.

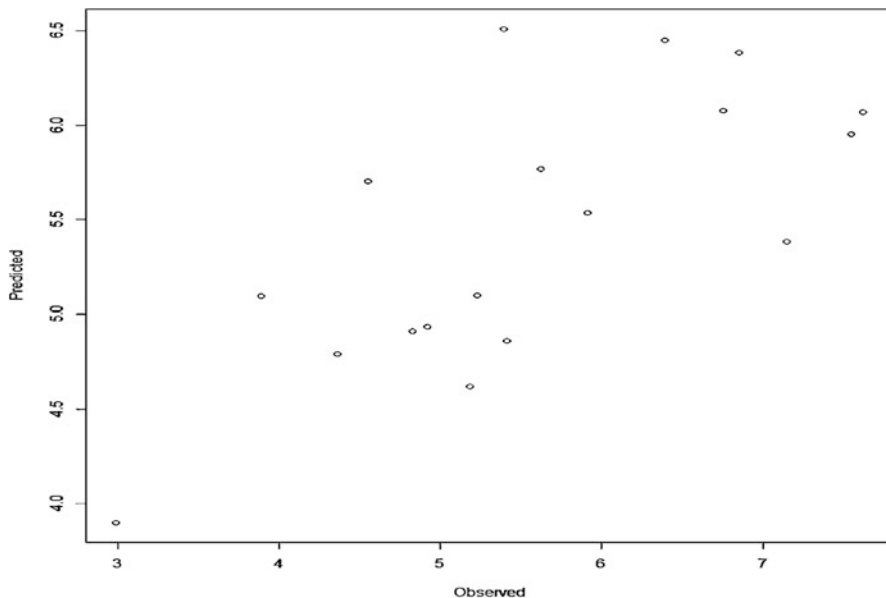
Before giving the output of the refitted model, we printed the average prediction performance of the five inner cross-validations of the four combinations of hyperparameters in the grid search for the outer fold cross-validation of the second partition:

```
> Hyperpar_Opt
  grid_length val_mse dropout1 units batchsize1 learning_rate epochs
1 1          8.093  0.00     33    28          0.001      298.0
2 2          1.008  0.05     33    28          0.001      299.4
3 3          13.309  0.00     67    28          0.001      177.2
4 4           0.971  0.05     67    28          0.001      247.6
```

We can see that the best combination of hyperparameters, in partition 2, of the inner cross-validation is combination 4 (with `dropout = 0.05`, `units = 67`, and `epochs = 247.6`) since it has the lowest MSE (0.971) in the testing set (validation set). Then with the optimal values of `dropout`, `epochs`, and `neurons`, the ANN was refitted, but using the whole second outer training set, and then, with the final trained ANN, the predictions for the outer testing set were obtained. Finally, the plot of the observed and predicted values of the outer testing set was done, as shown in Fig. 11.9. The MSE was calculated and was equal to (0.810), which is lower than the MSE obtained (1.222) using only one split (inner TRN-TST) for partition 2.

Finally, using the code given in Appendix 1, we implemented the above code using the five outer folds and five inner folds, and the MSE resulting from the





**Fig. 11.9** Observed versus predicted values for fold 2 trained with a grid search

average of the five outer CV is equal to 0.892; it is slightly larger than when using only fold 2, which means that the variability of fold to fold is considerably large.

### Example 11.3 MaizeToy Example with Five-fold CV with Dropout

Next, we perform a five-fold CV for the MaizeToy data sets but with different percentages of dropout, a type of regularization. It is important to point out that now, in addition to reporting the MSE by fold, we also provide the MSE for each environment. First, we provide the MSE for each environment, then we average the five-folds, using as input the information of environments + the information of lines (taking into account the marker information). The code used to obtain these results is given in Appendix 2. This code also uses the Code\_Tuning\_With\_Flags\_00.R, but now the grid search is performed not only for two hyperparameters since we use two values for the following hyperparameters: units, batchsize, and learning\_rate, which means that the total combinations to be evaluated are equal to  $2^3 = 8$ , while the remaining four hyperparameters, validation split, dropout, activation function, and number of epochs, were fixed at 0.20, 0, relu, and 1000, respectively. Next we provide the key part of the Appendix 2 code:

```
runs.sp<-tuning_run("Code_Tuning_With_Flags_00.R", flags=list
(dropout1= c(0),
              units = c(33,67),
              activation1=("relu"),
              batchsize1=c(28,56),
```

```
Epoch1=c(1000),
learning_rate=c(0.001,0.01),
val_split=c(0.2)), sample=1, confirm=FALSE, echo =F),
```

where it is clear that the % of dropout value was fixed at zero. Of course, when you want to use other dropout percentages, you only need to specify in `dropout1` the new value in the above code for tuning and it is modified in the `Code_Tuning_With_Flags_00.R` the following part of the code

```
layer_dropout(rate = 0.05) %>%
```

This means that when the `Code_Tuning_With_Flags_00.R` is executed with 5% dropout, in this code, the line above is modified with a `rate = 0.05`, but if you want to run with 25% dropout, the `rate` parameter is changed to 0.25. This code in Appendix 2 is run for the following dropout percentages: 0%, 5%, 15%, 25%, and 35%. This code was run once for a grid of the five dropout percentage values, but first without taking into account the genotype by environment interaction and then with the genotype by environment interaction. It is important to point out that by changing the value of the `sample` argument in the `tuning_run()` function, we can implement the random search grid that tests only the proportion of hyperparameter combinations that is specified in the `sample`. For example, if we specify `sample = 0.5`, this means that only half of the total number of hyperparameter combinations in the grid will be evaluated, while if we specify `sample = 0.10`, only 10% of the total number of hyperparameter combinations should be evaluated. However, to use the code in Appendix 2 with a value of `sample` less than one, it is also important to modify the number of inner testing CV to 1 (`nCVI = 1`, #####Number of folds for inner CV) since with more than two (`nCVI = 2, 3, 4, ...`), the R code given in Appendix 2 is not valid.

Table 11.2 shows that in general, the prediction performance in terms of MSE and MAAPE improved using different dropout percentages, since the best prediction performance was observed using 15% of dropout (rows in bold).

Table 11.3 gives the prediction performance using the same dropout percentages using the code given in Appendix 2, but now taking into account as predictors the information corresponding to the  $G \times E$  interaction term which is obtained by replacing, in Appendix 2, `X = cbind(Z.E, Z.G)` with `X = cbind(Z.E, Z.G, Z.EG)`. This table shows that there is a gain in prediction performance with a dropout percentage larger than zero and the best predictions are obtained with 15% dropout. However, it is clear that the predictions are worse when the  $G \times E$  interaction term is ignored, which can be attributed in part to the fact that the number of data sets used is small and that ANN with at least one hidden layer can capture complex interactions by the nonlinear transformations performed in the hidden layer. Perhaps for this reason, in many cases providing as input the information corresponding to the  $G \times E$  interaction term is not required. It is possible to increase the prediction accuracy of this model with the  $G \times E$  interaction term by increasing the number of neurons in the hidden layer, but this also increases the computational resources required.

**Table 11.2** Prediction performance of trait yield with five-fold CV with different dropout percentages without the genotype  $\times$  environment interaction term

% Dropout	Environment	Trait	MSE	SE_MSE	MAAPE	SE_MAAPE
0	EBU	Yield	1.538	0.350	0.174	0.018
0	KAK	Yield	0.443	0.097	0.103	0.016
0	KTI	Yield	1.222	0.185	0.154	0.021
5	EBU	Yield	1.371	0.315	0.163	0.016
5	KAK	Yield	0.407	0.091	0.105	0.012
5	KTI	Yield	1.140	0.206	0.144	0.021
<b>15</b>	<b>EBU</b>	<b>Yield</b>	<b>1.465</b>	<b>0.278</b>	<b>0.168</b>	<b>0.018</b>
<b>15</b>	<b>KAK</b>	<b>Yield</b>	<b>0.380</b>	<b>0.066</b>	<b>0.101</b>	<b>0.009</b>
<b>15</b>	<b>KTI</b>	<b>Yield</b>	<b>1.055</b>	<b>0.214</b>	<b>0.138</b>	<b>0.018</b>
25	EBU	Yield	1.540	0.332	0.174	0.020
25	KAK	Yield	0.473	0.112	0.112	0.016
25	KTI	Yield	1.194	0.184	0.144	0.018
35	EBU	Yield	1.348	0.227	0.169	0.018
35	KAK	Yield	0.411	0.110	0.100	0.016
35	KTI	Yield	1.280	0.183	0.150	0.019

**Table 11.3** Prediction performance of trait yield with five-fold CV with different dropout percentages with the  $G \times E$  interaction term

% Dropout	Environment	Trait	MSE	SE_MSE	MAAPE	SE_MAAPE
0	EBU	Yield	1.772	0.137	0.176	0.012
0	KAK	Yield	0.696	0.212	0.126	0.020
0	KTI	Yield	1.422	0.255	0.150	0.015
5	EBU	Yield	10.849	9.412	0.289	0.125
5	KAK	Yield	5.518	5.003	0.246	0.135
5	KTI	Yield	6.801	5.397	0.279	0.128
<b>15</b>	<b>EBU</b>	<b>Yield</b>	<b>1.680</b>	<b>0.235</b>	<b>0.168</b>	<b>0.012</b>
<b>15</b>	<b>KAK</b>	<b>Yield</b>	<b>0.536</b>	<b>0.094</b>	<b>0.116</b>	<b>0.011</b>
<b>15</b>	<b>KTI</b>	<b>Yield</b>	<b>1.426</b>	<b>0.243</b>	<b>0.153</b>	<b>0.015</b>
25	EBU	Yield	1.867	0.259	0.185	0.010
25	KAK	Yield	0.592	0.162	0.116	0.017
25	KTI	Yield	1.393	0.285	0.150	0.013
35	EBU	Yield	1.771	0.289	0.175	0.016
35	KAK	Yield	0.581	0.128	0.113	0.015
35	KTI	Yield	1.413	0.294	0.147	0.015

**Example 11.4 MaizeToy Example with More Than One Hidden Layer with Inner CV**

The goal of this section is to show how to train DL models with different number of layers and how they impact the prediction performance. Now the grid contains the number of neurons (33, 67), three dropout percentages (0, 5, and 10%) and two batch

size values (28, 56), and again the number of epochs is chosen using the early stopping approach. All 12 combinations are evaluated, since they were implemented using the full grid search with  $\text{sample} = 1$ . It is important to point out that this grid was implemented with an outer 5FCV and inner 5FCV. The code used for implementing this grid with four hidden layers is given in Appendix 4 (this code call the code given in Appendix 3 that contains the flags). It is very similar to the code used in Appendix 2, but one of the most important parts is how the layers are added to this code. Next we provide the part where the layers are added.

```

model %>%
  layer_dense(units =Units_O , activation =activation_O, input_shape =
c(dim(X_trn) [2])) %>%
  layer_dropout (rate =Drop_O) %>%
  layer_dense(units =Units_O , activation =activation_O) %>%
  layer_dropout (rate =Drop_O) %>%
  layer_dense(units =Units_O , activation =activation_O) %>%
  layer_dropout (rate =Drop_O) %>%
  layer_dense(units =Units_O , activation =activation_O) %>%
  layer_dropout (rate =Drop_O) %>%
  layer_dense(units =1)

```

This part of the code that forms part of Appendix 4 is for a DL model with four hidden layers; the lines that start with `layer_dense()` specified all the layers, but the first four correspond to the hidden layers and the last one to the output layer. Also, this DL model corresponds to a neural network with five layers.

Table 11.4 shows the prediction performance with one to four different hidden layers; the prediction performance is affected by the number of hidden layers used, but in this case, the best predictions across environments were obtained with three hidden layers.

**Table 11.4** Prediction performance of trait yield with five outer fold CV, five inner fold CV, with different dropout percentages without the  $G \times E$  interaction term. Env denotes environments

No. layers	Environment	Trait	MSE	SE_MSE	MAAPE	SE_MAAPE
1	EBU	Yield	1.489	0.316	0.170	0.018
1	KAK	Yield	0.426	0.088	0.102	0.015
1	KTI	Yield	1.251	0.147	0.153	0.020
2	EBU	Yield	1.456	0.260	0.168	0.016
2	KAK	Yield	0.427	0.110	0.103	0.015
2	KTI	Yield	1.118	0.115	0.151	0.016
<b>3</b>	<b>EBU</b>	<b>Yield</b>	<b>1.372</b>	<b>0.212</b>	<b>0.157</b>	<b>0.015</b>
<b>3</b>	<b>KAK</b>	<b>Yield</b>	<b>0.412</b>	<b>0.062</b>	<b>0.101</b>	<b>0.007</b>
<b>3</b>	<b>KTI</b>	<b>Yield</b>	<b>1.134</b>	<b>0.203</b>	<b>0.148</b>	<b>0.022</b>
4	EBU	Yield	1.724	0.291	0.184	0.017
4	KAK	Yield	0.350	0.060	0.092	0.008
4	KTI	Yield	1.320	0.298	0.162	0.024

**Table 11.5** Prediction performance of trait yield with five outer fold CV, one inner fold CV, with different hidden layers without the  $G \times E$  interaction term and the inner CV

No. layers	Env	Trait	MSE	SE_MSE	MAAPE	SE_MAAPE
1	EBU	Yield	1.2767	0.1807	0.1469	0.0131
1	KAK	Yield	0.3508	0.0613	0.0962	0.0123
1	KTI	Yield	1.0319	0.1332	0.138	0.0135
2	EBU	Yield	1.1731	0.1579	0.1408	0.012
2	KAK	Yield	0.4397	0.1274	0.1049	0.0161
2	KTI	Yield	1.2271	0.1211	0.151	0.0134
<b>3</b>	<b>EBU</b>	<b>Yield</b>	<b>1.1565</b>	<b>0.1844</b>	<b>0.1435</b>	<b>0.007</b>
<b>3</b>	<b>KAK</b>	<b>Yield</b>	<b>0.4852</b>	<b>0.1157</b>	<b>0.1138</b>	<b>0.0186</b>
<b>3</b>	<b>KTI</b>	<b>Yield</b>	<b>1.1017</b>	<b>0.1213</b>	<b>0.1376</b>	<b>0.0155</b>
4	EBU	Yield	1.3575	0.1894	0.1602	0.0077
4	KAK	Yield	0.4864	0.1152	0.1238	0.0189
4	KTI	Yield	1.2323	0.2545	0.1481	0.0092

**Example 11.5 MaizeToy Example with More Than One Hidden Layer Without Inner CV**

The goal of this example is to implement exactly the same grid used in Example 11.4, but instead of implementing the five inner fold CV, we implemented this with only one inner random partition. The code used here is the one given in Appendix 5 with only one inner cross-validation.

Table 11.5 shows that the best predictions are observed with three hidden layers, but it is important to point out that in general the prediction performances using only one inner random partition were not quite similar to performances using five inner fold CV, as can be observed in Table 11.4 in Example 11.4.

**Example 11.6 MaizeToy Example with One Hidden Layer with Inner CV and Ridge, Lasso, and Elastic Net (Ridge-Lasso) Regularization**

The code used for implementing a grid search with an outer and inner CV with Ridge regularization (L2) is similar to the one given in Appendix 2, with the difference that now for specifying the model the following part of the code is used:

```

model_Sec %>%
  layer_dense(units = Units_O , activation = activation_O,
kernel_regularizer = regularizer_l2(Reg_l2_O) , input_shape = c(dim
(X_trn) [2])) %>%
  layer_dropout(rate = Drop_O) %>%
  layer_dense(units = 1, activation = "linear")
    
```

The only new part of this code is the part that contains `kernel_regularizer = regularizer_l2(Reg_l2_O)` that specifies Ridge regularization, and in `Ridge_values`, the following values were used: 0.001, 0.01, 0.1, 0.5, 1. The grid here is composed of four values due to two unit values (33, 67) and two batch size values (28, 56); we used `sample = 1`, which means that the four hyperparameters

**Table 11.6** Prediction performance of trait yield with five outer fold CV and five-fold inner CV with different values of L2 regularization (0.001, 0.01, 0.1, 0.5, 1) without the  $G \times E$  interaction term

L2 value	Environment	Trait	MSE	SE_MSE	MAAPE	SE_MAAPE
0.001	EBU	Yield	1.639	0.231	0.166	0.011
0.001	KAK	Yield	0.495	0.118	0.105	0.008
0.001	KTI	Yield	1.366	0.268	0.150	0.017
<b>0.01</b>	<b>EBU</b>	<b>Yield</b>	<b>1.387</b>	<b>0.119</b>	<b>0.169</b>	<b>0.013</b>
<b>0.01</b>	<b>KAK</b>	<b>Yield</b>	<b>0.338</b>	<b>0.080</b>	<b>0.085</b>	<b>0.008</b>
<b>0.01</b>	<b>KTI</b>	<b>Yield</b>	<b>1.132</b>	<b>0.217</b>	<b>0.149</b>	<b>0.023</b>
0.1	EBU	Yield	1.371	0.252	0.168	0.022
0.1	KAK	Yield	0.384	0.090	0.093	0.010
0.1	KTI	Yield	1.165	0.247	0.144	0.026
0.5	EBU	Yield	1.372	0.315	0.167	0.026
0.5	KAK	Yield	0.428	0.132	0.101	0.015
0.5	KTI	Yield	1.194	0.234	0.141	0.025
1	EBU	Yield	1.413	0.342	0.168	0.027
1	KAK	Yield	0.461	0.151	0.107	0.019
1	KTI	Yield	1.215	0.233	0.142	0.025

were evaluated. Each of these values will add  $\text{Ridge\_val} \times \text{weight\_coefficient\_value}$  to the total loss of the network. Since this penalty is only added at training time, the loss for this network will be much bigger at training than at testing time. To reproduce these results, you can replace in Appendices A3 and A4 the specification of the model provided above each time you need to modify in `kernel_regularizer = regularizer_l2(Reg_l2_O)` the corresponding value of the Ridge regularization parameter.

Table 11.6 gives the output without taking into account the  $G \times E$  interaction using the following L2 regularization values of lambda, 0.001, 0.01, 0.1, 0.5, 1, implemented with the early stopping method. We can see in Table 11.6 that the best performance was obtained using the value of 0.01 for the L2 regularization hyperparameter.

Table 11.7 provides the output without taking into account the  $G \times E$  interaction for the L1 regularization for the same grid containing the following values of lambda, 0.001, 0.01, 0.1, 0.5, 1, two neuron values (33, 67), two batch size values (28, 56), and 1000 epochs implemented with early stopping. The performance was very similar using different values of L1 for regularization, but the performance under the L1 regularization using a value of 0.01 was slightly better.

In similar fashion, Table 11.8 gives the output without taking into account the  $G \times E$  interaction for five regularization parameters (0.001, 0.01, 0.1, 0.5, 1) using the Elastic Net regularization (L1-L2) for two neuron values (33, 67), two batch size values (28, 56), and 1000 epochs implemented with early stopping. We can see in Table 11.8 that the best performance was obtained using the value of 0.001 of the L1-L2 regularization parameter across environments under the MAAPE and a value of 0.01 of the L1-L2 regularization parameter under the MSE.

**Table 11.7** Prediction performance of trait yield with five outer fold CV, five inner fold CV with different values of L1 regularization (0.001, 0.01, 0.1, 0.5, 1) in the grid without the  $G \times E$  interaction term

L1 value	Environment	Trait	MSE	SE_MSE	MAAPE	SE_MAAPE
0.001	EBU	Yield	1.5623	0.2358	0.1668	0.0154
0.001	KAK	Yield	0.4312	0.099	0.0959	0.0038
0.001	KTI	Yield	1.2691	0.2402	0.1459	0.0178
<b>0.01</b>	<b>EBU</b>	<b>Yield</b>	<b>1.4894</b>	<b>0.2366</b>	<b>0.1762</b>	<b>0.0186</b>
<b>0.01</b>	<b>KAK</b>	<b>Yield</b>	<b>0.4366</b>	<b>0.0717</b>	<b>0.1055</b>	<b>0.0113</b>
<b>0.01</b>	<b>KTI</b>	<b>Yield</b>	<b>1.1368</b>	<b>0.2443</b>	<b>0.1365</b>	<b>0.0285</b>
0.1	EBU	Yield	1.4492	0.3605	0.1679	0.0275
0.1	KAK	Yield	0.4813	0.1466	0.1138	0.0203
0.1	KTI	Yield	1.2664	0.2405	0.1439	0.0236
0.5	EBU	Yield	1.6529	0.389	0.1669	0.0288
0.5	KAK	Yield	1.0277	0.2506	0.183	0.025
0.5	KTI	Yield	1.2195	0.2598	0.1403	0.0232
1	EBU	Yield	1.6669	0.3923	0.1675	0.029
1	KAK	Yield	1.0326	0.2575	0.1829	0.026
1	KTI	Yield	1.2108	0.2631	0.1398	0.0233

**Table 11.8** Prediction performance of trait yield with five outer fold CV, five inner fold CV, with different values of L1-L2 regularization (0.001, 0.01, 0.1, 0.5, 1) in the grid without the  $G \times E$  interaction term

L1_L2 value	Environment	Trait	MSE	SE_MSE	MAAPE	SE_MAAPE
0.001	EBU	Yield	1.4893	0.1938	<b>0.167</b>	0.0154
0.001	KAK	Yield	0.3991	0.1053	<b>0.0909</b>	0.0067
0.001	KTI	Yield	1.2316	0.2226	<b>0.1459</b>	0.0181
0.01	EBU	Yield	<b>1.4719</b>	0.2554	0.1761	0.0207
0.01	KAK	Yield	<b>0.4286</b>	0.0767	0.1038	0.0096
0.01	KTI	Yield	<b>1.1693</b>	0.2441	0.139	0.0276
0.1	EBU	Yield	1.4484	0.3632	0.1677	0.0277
0.1	KAK	Yield	0.5088	0.164	0.1141	0.0213
0.1	KTI	Yield	1.2798	0.2451	0.1436	0.0236
0.5	EBU	Yield	1.6546	0.3837	0.1672	0.0289
0.5	KAK	Yield	1.0238	0.2445	0.1825	0.0247
0.5	KTI	Yield	1.2159	0.2599	0.1398	0.023
1	EBU	Yield	1.6574	0.3836	0.1674	0.0288
1	KAK	Yield	1.0372	0.256	0.1833	0.026
1	KTI	Yield	1.2208	0.2609	0.1402	0.0234

**Table 11.9** Prediction performance of trait yield with five outer and five inner fold CV with different different optimizers without the genotype  $\times$  environment interaction term

Optimizer	Environment	Trait	MSE	SE_MSE	MAAPE	SE_MAAPE
optimizer_adam	EBU	Yield	1.7009	0.2456	0.1679	0.0098
optimizer_adam	KAK	Yield	0.6735	0.1827	0.1254	0.019
optimizer_adam	KTI	Yield	1.4192	0.295	0.1505	0.0157
optimizer_sgd	EBU	Yield	1.457	0.3499	0.1702	0.0264
optimizer_sgd	KAK	Yield	0.4953	0.1385	0.1074	0.014
optimizer_sgd	KTI	Yield	1.2861	0.2312	0.1457	0.0236
optimizer_rmsprop	EBU	Yield	1.7165	0.2187	0.1682	0.0093
optimizer_rmsprop	KAK	Yield	0.862	0.2246	0.1484	0.0242
optimizer_rmsprop	KTI	Yield	1.5172	0.2486	0.1582	0.0103
optimizer_adagrad	EBU	Yield	32.0716	2.8222	0.7224	0.009
optimizer_adagrad	KAK	Yield	19.0737	1.3223	0.7157	0.0146
optimizer_adagrad	KTI	Yield	26.0565	1.9791	0.7131	0.0079
optimizer_adadelat	EBU	Yield	39.5482	2.2787	0.7805	0.0018
optimizer_adadelat	KAK	Yield	24.0182	0.9201	0.7783	0.007
optimizer_adadelat	KTI	Yield	33.2928	2.8686	0.7779	0.0041
<b>optimizer_adamax</b>	<b>EBU</b>	<b>Yield</b>	<b>1.4955</b>	<b>0.1743</b>	<b>0.1664</b>	<b>0.0117</b>
<b>optimizer_adamax</b>	<b>KAK</b>	<b>Yield</b>	<b>0.427</b>	<b>0.117</b>	<b>0.0932</b>	<b>0.0093</b>
<b>optimizer_adamax</b>	<b>KTI</b>	<b>Yield</b>	<b>1.2982</b>	<b>0.2195</b>	<b>0.1522</b>	<b>0.0229</b>
optimizer_nadam	EBU	Yield	1.709	0.2438	0.168	0.0098
optimizer_nadam	KAK	Yield	0.6759	0.1839	0.1255	0.0191
optimizer_nadam	KTI	Yield	1.4208	0.2942	0.1506	0.0155

**Example 11.7 MaizeToy Example with One Hidden Layer with Inner CV, a Dropout Rate of 5%, and Different Optimizers**

In Table 11.9 are given the results of evaluating seven types of optimizers for a five outer and five inner fold cross-validations without taking into account the genotype $\times$ environment interaction. The code given in Appendix 2 was used in this implementation, but with a 0% dropout rate and with only one hidden layer. The neuron grid consists of two neurons (33 and 67), two batch sizes (28, 56) and they were implemented with early stopping to find the optimal number of epochs, which was fixed at 1000. Since the code given in Appendix 2 was used each time, this code was run by replacing in the optimizer each of the seven types of optimizers. The results in Table 11.9 show that the best performance in terms of prediction accuracy was found under the **optimizer\_adamax** and the worst under the **optimizer\_adagrad**.

Finally, with these examples we illustrated how to use DNN models for implementing genomic prediction models for continuous outcomes in Keras. Most of the key elements for implementing DNN models were explained to enable you to implement these models on your own data without any complications. However, as mentioned before, the DNN training process is challenging due to the complexity of tuning hyperparameters, but thanks to functions like the `tfruns`, this work is not as hard. However, more research and automatization of all these activities are required to be able to implement DNN models more quickly and efficiently since the tuning process still requires a lot of time.



## Appendix 1

MaizeToy example with five-fold CV.

```

rm(list=ls())
library(lsa)
library(keras)
library(BMTME)
library(plyr)
library(tidyr)
library(dplyr)
library(tfruns)
options(bitmapType='cairo')

#####Set seed for reproducible results#####
use_session_with_seed(45)

#####Loading the MaizeToy data sets#####
data("MaizeToy")
head(phenoMaizeToy)

#####Ordering the data #####
phenoMaizeToy<- (phenoMaizeToy[order(phenoMaizeToy$Env,
phenoMaizeToy$Line),])
rownames(phenoMaizeToy)=1:nrow(phenoMaizeToy)
head(phenoMaizeToy, 8)

#####Design matrices#####
LG <- cholesky(genoMaizeToy)
ZG <- model.matrix(~0 + as.factor(phenoMaizeToy$Line))
Z.G <- ZG %*%LG
Z.E <- model.matrix(~0 + as.factor(phenoMaizeToy$Env))
ZEG <- model.matrix(~0 + as.factor(phenoMaizeToy$Line):as.factor
(phenoMaizeToy$Env))
G2 <- kronecker(diag(length(unique(phenoMaizeToy$Env))), data.matrix
(genoMaizeToy))
LG2 <- cholesky(G2)
Z.EG <- ZEG %*% LG2

#####Selecting the response variable#####
Y <- as.matrix(phenoMaizeToy[, -c(1, 2)])

#####Training-testing sets using the BMTME package#####
pheno <- data.frame(GID = phenoMaizeToy[, 1], Env = phenoMaizeToy[, 2],
Response = phenoMaizeToy[, 3])

CrossV <- CV.KFold(pheno, DataSetID = 'GID', K = 5, set_seed = 123)

```

```
#####Final X and y for fitting the model#####
y=(phenoMaizeToy[, 3])
X=cbind(Z.E,Z.G)

#####Outer cross-validation#####
digits=4
Names_Traits=colnames(Y)
results=data.frame()
t=1
for (o in 1:5){
#o=2
  tst_set=CrossV$CrossValidation_list[[o]]
  X_trn=(X[-tst_set,])
  X_tst=(X[tst_set,])
  y_trn=(y[-tst_set])
  y_tst=(y[tst_set])

#####Inner cross-validation#####
nCVI=5 ####Number of folds for inner CV
Hyperpar=data.frame()
for (i in 1:nCVI){
  set.seed(i+100)
  Sam_per=sample(1:nrow(X_trn),nrow(X_trn))
  X_trII=X_trn[Sam_per,]
  y_trII=y_trn[Sam_per]

#####Using the tuning_run function for tuning
hyperparameters#####
  runs.sp<-tuning_run("Code_Tuning_With_Flags_00.R",flags=list
(dropout1=c(0,0.05), units=c(33,67), activation1=("relu"),
batchsize1=c(28), Epoch1=c(1000), learning_rate=c(0.001),
val_split=c(0.2)), sample=1,confirm=FALSE,echo=F)

##### Ordering in the same way all grids
  runs.sp=runs.sp[order(runs.sp$flag_units,runs.sp$flag_dropout1),]

  runs.sp$grid_length=1:nrow(runs.sp)
  Parameters=data.frame(grid_length=runs.sp$grid_length,
metric_val_mse=runs.sp$metric_val_mse,flag_dropout1=runs.
sp$flag_dropout1,flag_units=runs.sp$flag_units, flag_batchsize1=runs.
sp$flag_batchsize1,epochs_completed=runs.sp$epochs_completed,
flag_learning_rate=runs.sp$flag_learning_rate, flag_activation1=runs.
sp$flag_activation1)
  Hyperpar=rbind(Hyperpar,data.frame(Parameters))
}
Hyperpar %>%
  group_by(grid_length) %>%
  summarise(val_mse=mean(metric_val_mse),
dropout1=mean(flag_dropout1),
units=mean(flag_units),
batchsize1=mean(flag_batchsize1),
learning_rate=mean(flag_learning_rate),
```

```

epochs=mean(epochs_completed) %>%
  select(grid_length, val_mse, dropout1, units, batchsize1,
learning_rate, epochs) %>%
  mutate_if(is.numeric, funs(round(., 3))) %>%
  as.data.frame() -> Hyperpar_Opt
#####Optimal hyperparameters#####
Min=min(Hyperpar_Opt$val_mse)
pos_opt=which(Hyperpar_Opt$val_mse==Min)
pos_opt=pos_opt[1]
Optimal_Hyper=Hyperpar_Opt[pos_opt,]
####Selecting the best hyperparameters
Drop_O=Optimal_Hyper$dropout1
Epoch_O=round(Optimal_Hyper$epochs,0)
Units_O=round(Optimal_Hyper$units,0)
activation_O=unique(Hyperpar$flag_activation1)
batchsize_O=round(Optimal_Hyper$batchsize1,0)
lr_O=Optimal_Hyper$learning_rate

print_dot_callback <- callback_lambda(
  on_epoch_end = function(epoch, logs) {
    if (epoch%% 20 == 0) cat("\n")
    cat(".")})

#####Refitting the model with the optimal values#####
model_Sec<-keras_model_sequential()

model_Sec %>%
  layer_dense(units =Units_O , activation =activation_O, input_shape =
c(dim(X_trn) [2])) %>%
  layer_dropout (rate =Drop_O) %>%
  layer_dense(units =1)

model_Sec %>% compile(
  loss = "mean_squared_error",
  optimizer = optimizer_adam(lr=lr_O),
  metrics = c("mean_squared_error"))

ModelFited <-model_Sec %>% fit (
  X_trn, y_trn,
  epochs=Epoch_O, batch_size =batchsize_O,
#####validation_split=0.2,early_stop,
  verbose=0, callbacks=list(print_dot_callback))

#####Prediction of testing set #####
Yhat=model_Sec%>% predict(X_tst)
y_p=Yhat
y_p_tst=as.numeric(y_p)
y_tst=y[tst_set]
MSE=mean((y_tst-y_p_tst)^2)
MSE

```

```
#####Saving the predictions of each outer testing set#####
results<-rbind(results, data.frame(Partition=o,MSE_Env=MSE))
cat ("CV=",o, "\n")
}

####Average prediction performance
MSE_global=mean(results$MSE_Env)
MSE_global
```

## Appendix 2

MaizeToy example with five-fold CV with dropout.

```
rm(list=ls())
library(lsa)
library(keras)
library(BMTME)
library(plyr)
library(tidyr)
library(dplyr)
library(tfruns)
options(bitmapType='cairo')

#####Set seed for reproducible results#####
use_session_with_seed(45)

#####Loading the MaizeToy data sets#####
data("MaizeToy")
head(phenoMaizeToy)

#####Ordering the data #####
phenoMaizeToy<- (phenoMaizeToy[order (phenoMaizeToy$Env,
phenoMaizeToy$Line),])
rownames (phenoMaizeToy) =1:nrow (phenoMaizeToy)
head (phenoMaizeToy, 8)

#####Design matrices#####
LG <- cholesky (genoMaizeToy)
ZG <- model.matrix (~0 + as.factor (phenoMaizeToy$Line))
Z.G <- ZG %*%LG
Z.E <- model.matrix (~0 + as.factor (phenoMaizeToy$Env))
ZEG <- model.matrix (~0 + as.factor (phenoMaizeToy$Line) :as.factor
(phenoMaizeToy$Env))
G2 <- kronecker (diag (length (unique (phenoMaizeToy$Env))), data.matrix
(genoMaizeToy))
LG2 <- cholesky (G2)
Z.EG <- ZEG %*% LG2
```

```
#####Selecting the response variable#####
Y <- as.matrix(phenoMaizeToy[, -c(1, 2)])

#####Training-testing sets using the BMTME package#####
pheno <- data.frame(GID = phenoMaizeToy[, 1], Env = phenoMaizeToy[, 2],
  Response = phenoMaizeToy[, 3])

CrossV <- CV.KFold(pheno, DataSetID = 'GID', K = 5, set_seed = 123)

#####Function for averaging the predictions#####
summary.BMTMECV <- function(results, information = 'compact', digits =
4, ...) {
  results %>%
    group_by(Environment, Trait, Partition) %>%
    summarise(MSE = mean((Predicted-Observed)^2),
      MAAPE = mean(atan(abs(Observed-Predicted)/abs(Observed))) %>%
    select(Environment, Trait, Partition, MSE, MAAPE) %>%
    mutate_if(is.numeric, funs(round(., digits))) %>%
    as.data.frame() -> presum

  presum %>% group_by(Environment, Trait) %>%
    summarise(SE_MAAPE = sd(MAAPE, na.rm = T)/sqrt(n()), MAAPE = mean
(MAAPE, na.rm = T),
      SE_MSE = sd(MSE, na.rm = T)/sqrt(n()), MSE = mean(MSE, na.rm = T)) %>%
  %
  select(Environment, Trait, MSE, SE_MSE, MAAPE, SE_MAAPE) %>%
  mutate_if(is.numeric, funs(round(., digits))) %>%
  as.data.frame() -> finalSum

  out <- switch(information,
    compact = finalSum,
    complete = presum,
    extended = {
      finalSum$Partition <- 'All'
      presum$Partition <- as.character(presum$Partition)
      presum$SE_MSE <- NA
      presum$SE_MAAPE <- NA
      rbind(presum, finalSum)
    }
  )
  return(out)
}

#####Final X and y for fitting the model#####
y = (phenoMaizeToy[, 3])
X = cbind(Z.E, Z.G)
Drop_per = c(0, 0.05, 0.15, 0.25, 0.35)
Final_results = data.frame()
for (e in 1:5) {
  #e=1
  #####Outer cross-validation#####
  digits=4
```

```

Names_Traits=colnames(Y)
results=data.frame()
t=1
for (o in 1:5){
  #o=1
  tst_set=CrossV$CrossValidation_list[[o]]
  X_trn=(X[-tst_set,])
  X_tst=(X[tst_set,])
  y_trn=(y[-tst_set])
  y_tst=(y[tst_set])

#####Inner cross-validation#####
nCVI=5 #####Number of folds for inner CV
Hyperpar=data.frame()
for (i in 1:nCVI){
  #i=1
  Sam_per=sample(1:nrow(X_trn),nrow(X_trn))
  X_trII=X_trn[Sam_per,]
  y_trII=y_trn[Sam_per]

#####Using the tuning_fun function for tuning
hyperparameters#####
  runs.sp<-tuning_run("Code_Tuning_With_Flags_00.R",flags=list
(dropout1= Drop_per[e],
                                units = c(33,67),
                                activation1="relu"),
                                batchsize1=c(28,56),
                                Epoch1=c(1000),
                                learning_rate=c(0.001,0.01),
                                val_split=c(0.2)),sample=1,
                                confirm =FALSE,echo =F)

##### Ordering in the same way all grids
  runs.sp=runs.sp[order(runs.sp$flag_units,runs.sp$flag_batchsize1,
runs.sp$flag_learning_rate),]

  runs.sp$grid_length=1:nrow(runs.sp)
  Parameters=data.frame(grid_length=runs.sp$grid_length,
metric_val_mse=runs.sp$metric_val_mse,flag_dropout1=runs.
sp$flag_dropout1,flag_units=runs.sp$flag_units, flag_batchsize1=runs.
sp$flag_batchsize1,epochs_completed=runs.sp$epochs_completed,
flag_learning_rate=runs.sp$flag_learning_rate, flag_activation1=runs.
sp$flag_activation1)
  Hyperpar=rbind(Hyperpar,data.frame(Parameters))
}
Hyperpar %>%
group_by(grid_length) %>%
summarise(val_mse=mean(metric_val_mse),
          dropout1=mean(flag_dropout1),
          units=mean(flag_units),
          batchsize1=mean(flag_batchsize1),
          learning_rate=mean(flag_learning_rate),
          epochs=mean(epochs_completed)) %>%

```

```

    select(grid_length, val_mse, dropout1, units, batchsize1,
learning_rate, epochs) %>%
  mutate_if(is.numeric, funs(round(., 3))) %>%
  as.data.frame() -> Hyperpar_Opt
#####Optimal hyperparameters#####
Min=min(Hyperpar_Opt$val_mse)
pos_opt=which(Hyperpar_Opt$val_mse==Min)
Optimal_Hyper=Hyperpar_Opt [pos_opt,]
pos_opt=pos_opt [1]
#####Selecting the best hyperparameters
Drop_O=Optimal_Hyper$dropout1
Epoch_O=round(Optimal_Hyper$epochs,0)
Units_O=round(Optimal_Hyper$units,0)
activation_O=unique(Hyperpar_$flag_activation1)
batchsize_O=round(Optimal_Hyper$batchsize1,0)
lr_O=Optimal_Hyper$learning_rate

print_dot_callback <- callback_lambda(
  on_epoch_end = function(epoch, logs) {
    if (epoch %% 20 == 0) cat ("\n")
    cat (".") })

#####Refitting the model with the optimal values#####
model_Sec<-keras_model_sequential()

model_Sec %>%
  layer_dense(units =Units_O , activation =activation_O, input_shape
= c(dim(X_trn) [2])) %>%
  layer_dropout(rate =Drop_O) %>%
  layer_dense(units =1)

model_Sec %>% compile(
  loss = "mean_squared_error",
  optimizer = optimizer_adam(lr=lr_O) ,
  metrics = c("mean_squared_error"))

ModelFited <-model_Sec %>% fit (
  X_trn, y_trn,
  epochs=Epoch_O, batch_size =batchsize_O,
#####validation_split=0.2,early_stop,
  verbose=0, callbacks=list(print_dot_callback))

####Prediction of testing set #####
Yhat=model_Sec%>% predict(X_tst)
y_p=Yhat
y_p_tst=as.numeric(y_p)

#####Saving the predictions of each outer testing
set#####
results<-rbind(results, data.frame(Position=tst_set,
  Environment=CrossV$Environments[tst_set] ,
  Partition=o,

```

```

        Units=Units_O,
        Epochs=Epoch_O,
        Observed=round(y[tst_set], digits), #s$response,
digits),
        Predicted=round(y_p_tst, digits),
        Trait=Names_Traits[t])
    cat("CV=", o, "\n")
}

results

#####Average of prediction performance#####
Pred_Summary=summary.BMTMECV(results=results, information =
'compact', digits = 4)
Pred_Summary
Final_results=rbind(Final_results, data.frame(Pred_Summary))
}
Final_results
write.csv(Final_results,
file="Example5_AppendixA2_modified_Table11.2.csv")

```

## Appendix 3

Flags with four hidden layers.

```

####a) Declaring the flags for hyperparameters
FLAGS = flags(
  flag_numeric("dropout1", 0.05),
  flag_integer("units", 33),
  flag_string("activation1", "relu"),
  flag_integer("batchsize1", 56),
  flag_integer("Epoch1", 1000),
  flag_numeric("learning_rate", 0.001),
  flag_numeric("val_split", 0.2))

####b) Defining the DNN model
build_model<-function() {
model <- keras_model_sequential()
model %>%
  layer_dense(units =FLAGS$units, activation =FLAGS$activation1,
input_shape = c(dim(X_trII) [2])) %>%
  layer_dropout(rate=FLAGS$dropout1) %>%
  layer_dense(units =FLAGS$units, activation =FLAGS$activation1) %>%
  layer_dropout(rate=FLAGS$dropout1) %>%
  layer_dense(units =FLAGS$units, activation =FLAGS$activation1) %>%
  layer_dropout(rate=FLAGS$dropout1) %>%
  layer_dense(units =FLAGS$units, activation =FLAGS$activation1) %>%
  layer_dropout(rate=FLAGS$dropout1) %>%
  layer_dense(units=1, activation ="linear")
}

```



```
#####c) Compiling the DNN model
model %>% compile(
  loss = "mse",
  optimizer = optimizer_adam(lr=FLAGS$learning_rate),
  metrics = c("mse"))
model}

model<-build_model()
model %>% summary()

print_dot_callback <- callback_lambda(
  on_epoch_end = function(epoch, logs) {
    if (epoch %% 20 == 0) cat("\n")
    cat(".")})

early_stop <- callback_early_stopping(monitor = "val_loss",
mode='min',patience =50)

#####d) Fitting the DNN model#####
model_Final<-build_model()
model_fit_Final<-model_Final %>% fit(
  X_trII, y_trII,
  epochs =FLAGS$Epoch1, batch_size =FLAGS$batchsize1,
  shuffled=F,
  validation_split =FLAGS$val_split,
  verbose=0, callbacks = list(early_stop,print_dot_callback))
```

## Appendix 4

MaizeToy example with five outer fold CV and five-folds inner CV without the  $G \times E$  interaction term and four hidden layers.

```
rm(list=ls())
library(lsa)
library(keras)
library(BMTME)
library(plyr)
library(tidyr)
library(dplyr)
library(tfruns)
library(tfestimators) # provides grid search & model training
interface

#####Set seed for reproducible results#####
use_session_with_seed(45)

#####Loading the MaizeToy data sets#####
data("MaizeToy")
head(phenoMaizeToy)
```

```
#####Ordering the data#####
phenoMaizeToy<- (phenoMaizeToy[order (phenoMaizeToy$Env,
phenoMaizeToy$Line),])
rownames (phenoMaizeToy)=1:nrow (phenoMaizeToy)
head (phenoMaizeToy, 8)

#####Design matrices#####
LG <- cholesky (genoMaizeToy)
ZG <- model.matrix (~0 + as.factor (phenoMaizeToy$Line))
Z.G <- ZG %*% LG
Z.E <- model.matrix (~0 + as.factor (phenoMaizeToy$Env))
ZEG <- model.matrix (~0 + as.factor (phenoMaizeToy$Line):as.factor
(phenoMaizeToy$Env))
G2 <- kronecker (diag (length (unique (phenoMaizeToy$Env))), data.matrix
(genoMaizeToy))
LG2 <- cholesky (G2)
Z.EG <- ZEG %*% LG2

#####Selecting the response variable#####
Y <- as.matrix (phenoMaizeToy[, -c (1, 2)])

####Training-testing sets using the BMTME package#####
pheno <- data.frame (GID = phenoMaizeToy[, 1], Env = phenoMaizeToy[, 2],
Response = phenoMaizeToy[, 3])

CrossV <- CV.KFold (pheno, DataSetID = 'GID', K = 5, set_seed = 123)

#####Function for averaging the predictions#####
summary.BMTMECV <- function (results, information = 'compact', digits =
4, ...) {
  results %>%
    group_by (Environment, Trait, Partition) %>%
    summarise (MSE = mean ((Predicted-Observed)^2),
      MAAPE = mean (atan (abs (Observed-Predicted) / abs (Observed)))) %>%
    select (Environment, Trait, Partition, MSE, MAAPE) %>%
    mutate_if (is.numeric, funs (round (., digits))) %>%
    as.data.frame () -> presum

  presum %>% group_by (Environment, Trait) %>%
    summarise (SE_MAAPE = sd (MAAPE, na.rm = T) / sqrt (n ()), MAAPE = mean
(MAAPE, na.rm = T),
      SE_MSE = sd (MSE, na.rm = T) / sqrt (n ()), MSE = mean (MSE, na.rm = T)) %>%
  %
  select (Environment, Trait, MSE, SE_MSE, MAAPE, SE_MAAPE) %>%
  mutate_if (is.numeric, funs (round (., digits))) %>%
  as.data.frame () -> finalSum

out <- switch (information,
  compact = finalSum,
  complete = presum,
  extended = {
    finalSum$Partition <- 'All'
  }
)
```

```

        presum$Partition <- as.character(presum$Partition)
        presum$SE_MSE <- NA
        presum$SE_MAAPE <- NA
        rbind(presum, finalSum)
    }
)
return(out)
}

#####Final X and y for fitting the model#####
y=(phenoMaizeToy[, 3])
X=cbind(Z.E,Z.G)

#####Outer cross-validation#####
digits=4
Names_Traits=colnames(Y)
results=data.frame()
t=1
for (o in 1:5){
  #o=1
  tst_set=CrossV$CrossValidation_list[[o]]
  X_trn=(X[-tst_set,])
  X_tst=(X[tst_set,])
  y_trn=(y[-tst_set])
  y_tst=(y[tst_set])

  #####Inner cross-validation#####
  nCVI=5 #####Number of folds for inner CV
  Hyperpar=data.frame()
  for (i in 1:nCVI){
    #i=1
    Sam_per=sample(1:nrow(X_trn),nrow(X_trn))
    X_trII=X_trn[Sam_per,]
    y_trII=y_trn[Sam_per]

    #####Using the tuning_fun function for tuning
hyperparameters#####
    runs.sp<-tuning_run("Code_Tuning_With_Flags_00_4HL.R",flags=list
(dropout1= c(0,0.05,0.1),
                                units = c(33,67),
                                activation1=("relu"),
                                batchsize1=c(28,56),
                                Epoch1=c(1000),
                                learning_rate=c(0.001),
                                val_split=c(0.2)),sample=1,confirm=FALSE,
echo =F)
    ##### Ordering in the same way all grids
    runs.sp=runs.sp[order(runs.sp$flag_units,runs.sp$flag_dropout1,
runs.sp$flag_batchsize1),]

    runs.sp$grid_length=1:nrow(runs.sp)
    Parameters=data.frame(grid_length=runs.sp$grid_length,

```

```

metric_val_mse=runs.sp$metric_val_mse,flag_dropout1=runs.
sp$flag_dropout1,flag_units=runs.sp$flag_units, flag_batchsize1=runs.
sp$flag_batchsize1,epochs_completed=runs.sp$epochs_completed,
flag_learning_rate=runs.sp$flag_learning_rate, flag_activation1=runs.
sp$flag_activation1)
  Hyperpar=rbind(Hyperpar,data.frame(Parameters))
}
Hyperpar %>%
  group_by(grid_length) %>%
  summarise(val_mse=mean(metric_val_mse),
            dropout1=mean(flag_dropout1),
            units=mean(flag_units),
            batchsize1=mean(flag_batchsize1),
            learning_rate=mean(flag_learning_rate),
            epochs=mean(epochs_completed)) %>%
  select(grid_length, val_mse, dropout1, units, batchsize1,
learning_rate, epochs) %>%
  mutate_if(is.numeric, funs(round(., 3))) %>%
  as.data.frame() -> Hyperpar_Opt
#####Optimal hyperparameters#####
Min=min(Hyperpar_Opt$val_mse)
pos_opt=which(Hyperpar_Opt$val_mse==Min)
pos_opt=pos_opt[1]
Optimal_Hyper=Hyperpar_Opt[pos_opt,]
#####Selecting the best hyperparameters
Drop_O=Optimal_Hyper$dropout1
Epoch_O=round(Optimal_Hyper$epochs,0)
Units_O=round(Optimal_Hyper$units,0)
activation_O=unique(Hyperpar$flag_activation1)
batchsize_O=round(Optimal_Hyper$batchsize1,0)
lr_O=Optimal_Hyper$learning_rate

print_dot_callback <- callback_lambda(
  on_epoch_end = function(epoch, logs) {
    if (epoch %% 20 == 0) cat ("\n")
    cat (".") })

#####Refitting the model with the optimal values#####
model_Sec<-keras_model_sequential()
model_Sec %>%
  layer_dense(units =Units_O , activation =activation_O, input_shape =
c(dim(X_trn) [2])) %>%
  layer_dropout (rate =Drop_O) %>%
  layer_dense(units =Units_O , activation =activation_O) %>%
  layer_dropout (rate =Drop_O) %>%
  layer_dense(units =Units_O , activation =activation_O) %>%
  layer_dropout (rate =Drop_O) %>%
  layer_dense(units =Units_O , activation =activation_O) %>%
  layer_dropout (rate =Drop_O) %>%
  layer_dense(units =1)

model_Sec %>% compile(
  loss = "mean_squared_error",

```

```

optimizer = optimizer_adam(),
metrics = c("mean_squared_error")

ModelFited <-model_Sec %>% fit (
  X_trn, y_trn,
  epochs=Epoch_O, batch_size =batchsize_O,
#####validation_split=0.2,early_stop,
  verbose=0,callbacks=list(print_dot_callback))

#####Prediction of testing set #####
Yhat=model_Sec%>% predict (X_tst)
y_p=Yhat
y_p_tst=as.numeric(y_p)

#####Saving the predicctions of each outer testing
set#####
results<-rbind(results, data.frame(Position=tst_set,
  Environment=CrossV$Environments [tst_set],
  Partition=o,
  Units=Units_O,
  Epochs=Epoch_O,
  Observed=round(y [tst_set], digits),
  #response, digits),
  Predicted=round(y_p_tst, digits),
  Trait=Names_Traits [t]))

  cat ("CV=", o, "\n")
}

results

#####Average of prediction performance#####
Pred_Summary=summary.BMTMECV(results=results, information =
'compact', digits = 4)
Pred_Summary

```

## Appendix 5

MaizeToy example with five-fold CV with only one inner CV.

```

rm(list=ls())
library(lsa)
library(keras)
library(BMTME)
library(plyr)
library(tidyr)
library(dplyr)

#####Set seed for reproducible results#####
use_session_with_seed(45)

```

```
#####Loading the MaizeToy data sets#####
data("MaizeToy")
head(phenoMaizeToy)

#####Ordering the data #####
phenoMaizeToy<- (phenoMaizeToy[order(phenoMaizeToy$Env,
phenoMaizeToy$Line),])
rownames(phenoMaizeToy)=1:nrow(phenoMaizeToy)
head(phenoMaizeToy,8)

#####Design matrices#####
LG <- cholesky(genoMaizeToy)
ZG <- model.matrix(~0 + as.factor(phenoMaizeToy$Line))
Z.G <- ZG %*%LG
Z.E <- model.matrix(~0 + as.factor(phenoMaizeToy$Env))
ZEG <- model.matrix(~0 + as.factor(phenoMaizeToy$Line):as.factor
(phenoMaizeToy$Env))
G2 <- kronecker(diag(length(unique(phenoMaizeToy$Env))), data.matrix
(genoMaizeToy))
LG2 <- cholesky(G2)
Z.EG <- ZEG %*% LG2

#####Selecting the response variable#####
Y <- as.matrix(phenoMaizeToy[, -c(1, 2)])

#####Training-testing sets using the BMTME package#####
pheno <- data.frame(GID = phenoMaizeToy[, 1], Env = phenoMaizeToy[, 2],
Response = phenoMaizeToy[, 3])

CrossV <- CV.KFold(pheno, DataSetID = 'GID', K = 5, set_seed = 123)

#####Grid of hyperparameters#####
Stage <- expand.grid(units_M=seq(33,67,33), epochs_M=1000, Dropout=
(0.0,0.05,0.15, 0.25, 0.35))

#####Function for averaging the predictions#####
summary.BMTMECV <- function(results, information = 'compact', digits =
4, ...) {
  results %>%
  group_by(Environment, Trait, Partition) %>%
  summarise(MSE = mean((Predicted-Observed)^2),
    MAAPE = mean(atan(abs(Observed-Predicted)/abs(Observed)))) %>%
  select(Environment, Trait, Partition, MSE, MAAPE) %>%
  mutate_if(is.numeric, funs(round(., digits))) %>%
  as.data.frame() -> presum

  presum %>% group_by(Environment, Trait) %>%
  summarise(SE_MAAPE = sd(MAAPE, na.rm = T)/sqrt(n()), MAAPE = mean
(MAAPE, na.rm = T),
    SE_MSE = sd(MSE, na.rm = T)/sqrt(n()), MSE = mean(MSE, na.rm = T)) %>
%
  select(Environment, Trait, MSE, SE_MSE, MAAPE, SE_MAAPE) %>%
```

```

mutate_if(is.numeric, funs(round(., digits))) %>%
as.data.frame() -> finalSum

out <- switch(information,
  compact = finalSum,
  complete = presum,
  extended = {
    finalSum$Partition <- 'All'
    presum$Partition <- as.character(presum$Partition)
    presum$SE_MSE <- NA
    presum$SE_MAAPE <- NA
    rbind(presum, finalSum)
  }
)
return(out)
}

#####Final X and y for fitting the model#####
y=(phenoMaizeToy[, 3])
X=cbind(Z.E,Z.G)

#####Outer cross-validation#####
digits=4
Names_Traits=colnames(Y)
results=data.frame()
t=1
for (o in 1:5) {
  tst_set=CrossV$CrossValidation_list[[o]]
  X_trn=(X[-tst_set,])
  X_tst=(X[tst_set,])
  y_trn=(y[-tst_set])
  y_tst=(y[tst_set])

  nCVI=1 ####Number of folds for inner CV
  i=1
  #####Matrices for saving the output of inner CV#####
  Tab_pred_MSE=matrix(NA,ncol=length(Stage[,1]), nrow= nCVI)
  Tab_pred_Epoch=matrix(NA,ncol=length(Stage[,1]), nrow= nCVI)
  Tab_pred_Units=matrix(NA,ncol=length(Stage[,1]), nrow= nCVI)
  Tab_pred_Drop=matrix(NA,ncol=length(Stage[,1]), nrow= nCVI)
  X_trI=X_trn
  y_trI=y_trn

  for (stage in seq_len(dim(Stage)[1])) {
    X_trII=X_trI
    y_trII=y_trI
    units_M <- Stage[stage, 1]
    epochs_M <- Stage[stage, 2]
    Drop_per= Stage[stage, 3]

  build_model<-function() {
  model <- keras_model_sequential()

```

```

model %>%
  layer_dense(units=units_M, activation = "relu", input_shape = c(dim
(X_trII) [2])) %>%
  layer_dropout(rate=Drop_per) %>%
  layer_dense(units=1)

model %>% compile(
  loss = "mse",
  optimizer=optimizer_adam(),
  metrics = c("mse"))
model}

model<-build_model()
model %>% summary()

print_dot_callback <- callback_lambda(
  on_epoch_end = function(epoch, logs) {
    if (epoch %% 20 == 0) cat("\n")
    cat(".")
  })

#####Fitting the model with early stopping#####
early_stop <- callback_early_stopping(monitor = "val_loss",
mode='min', patience =50)

#####Fit of the model for each values of the grid#####
model_Final<-build_model()
model_fit_Final<-model_Final %>% fit(
  X_trII, y_trII,
  epochs =epochs_M, batch_size =72,
  ###shuffled=F,
  validation_split = 0.2,
  verbose=0, callbacks = list(early_stop, print_dot_callback))
#####Saving the output of each hyperparameter#####
No.Epoch_Min=length(model_fit_Final$metrics$val_mse)
Min_MSE=model_fit_Final$metrics$val_mean_squared_error[No.
Epoch_Min]

Tab_pred_MSE[i,stage]=Min_MSE
Tab_pred_Units[i,stage]=units_M
Tab_pred_Epoch[i,stage]=No.Epoch_Min[1]
Tab_pred_Drop[i,stage]=Drop_per
}

#####Selecting the optimal hyperparameters#####
Median_MSE_Inner=apply(Tab_pred_MSE,2,median)
Units_Inner=apply(Tab_pred_Units,2,max)
Drop_Inner=apply(Tab_pred_Drop,2,max)
Epoch_Inner=apply(Tab_pred_Epoch,2,median)
Pos_Min_MSE=which(Median_MSE_Inner==min(Median_MSE_Inner))
Units_O=Units_Inner[Pos_Min_MSE]

```



```

Epoch_O= Epoch_Inner[Pos_Min_MSE]
Drop_O= Drop_Inner[Pos_Min_MSE]

#####Refitting the model with the optimal values#####
model_Sec<-keras_model_sequential()
model_Sec %>%
  layer_dense(units =Units_O , activation ="relu", input_shape = c(dim
(X_trn) [2])) %>%
  layer_dropout(rate =Drop_O) %>%
  layer_dense(units =1)

model_Sec %>% compile(
  loss = "mean_squared_error",
  optimizer = optimizer_adam(),
  metrics = c("mean_squared_error"))

ModelFited <-model_Sec %>% fit (
  X_trn, y_trn,
  epochs=Epoch_O, batch_size =30,
  ##### validation_split=0.2,early_stop,
  verbose=0, callbacks=list(print_dot_callback))

#####Prediction of testing set #####
Yhat=model_Sec %>% predict (X_tst)
y_p=Yhat
y_p_tst=as.numeric(y_p)

#####Saving the predicctions of each outer testing set#####
results<-rbind(results, data.frame(Position=tst_set,
  Environment=CrossV$Environments [tst_set] ,
  Partition=o,
  Units=Units_O,
  Epochs=Epoch_O,
  Drop_Out=Drop_O,
  Observed=round(y [tst_set], digits) ,
  #$response, digits),
  Predicted=round(y_p_tst, digits),
  Trait=Names_Traits [t]))

cat ("CV=", o, "\n")
}
results

#####Average of prediction performance#####
Pred_Summary=summary.BMTMECV(results=results, information =
'compact', digits = 4)
Pred_Summary

```

## References

- Allaire JJ (2018) Tfruns: training run tools for 'tensorflow'. <https://CRAN.R-project.org/package=tfruns>
- Allaire JJ, Chollet F (2019) Keras: R interface to Keras. <https://CRAN.R-project.org/package=keras>
- Baysolow II T (2017) Introduction to deep learning using R. A step-by step guide to learning and implementing Deep learning models using R. Apress
- Chollet F, Allaire JJ (2017) Deep learning with R. Manning Publications, Manning Early Access Program (MEA), 1st edn
- Cybenko G (1989) Approximations by superpositions of sigmoidal functions. *Math Control Signal Syst* 2:303–314
- Géron A (2019) Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow. Concepts, tools, and techniques to build intelligent systems. O'Reilly Media, Inc., California
- Hecht-Nielsen R (1987) Counterpropagation networks. *Appl Opt* 26:4979–4984
- Lantz B (2015) Machine learning with R, 2nd edn. Packt Publishing Ltd, Birmingham
- Lippmann R (1987) An introduction to computing with neural nets. *IEEE ASSP Mag* 4(2):4–22
- Patterson J, Gibson A (2017) Deep learning: a practitioner's approach. O'Reilly Media
- Samarasinghe S (2007) Neural networks for applied sciences and engineering. From fundamentals to complex pattern recognition. Auerbach Publications, Taylor & Francis Group, Boca Raton, New York

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

