Chapter 12
Artificial Neural Networks and Deep Learning for Genomic Prediction of Binary, Ordinal, and Mixed Outcomes

12.1 Training DNN with Binary Outcomes

Before starting with the examples, we explain in general terms the process to follow to train DNN for binary outcomes. When training binary outcomes, it is important to denote the values of the response variable as 0 and 1, where 0 denotes the absence of the disease and 1 its presence; any other two types of interest should be denoted as 0 and 1. Below we provide some key elements to train this type of DNNs more efficiently:

(a) *Define the DNN model in Keras.* As in the previous chapters, we again focus only on fully connected neural networks that consist of stacking fully connected networks to all neurons (units). We suggest using the RELU activation function or some of the following activation functions (leaky RELU, tanh, exponential linear unit, etc.) for hidden layers, but for the output layer we suggest using a single-unit layer with the sigmoid activation function to guarantee that the output is a probability between 0 and 1. Also, the first layer requires the input shape (features) information; however, this is not required for the following layers since they automatically infer the shape from the previous layer.

    The construction of a DNN in Keras for binary outcomes again needs to start by initializing a sequential model using the keras_model_sequential() function which allows implementing a series of layer functions that create a linear stacking of layers. The summary () can also be used to print a summary of our DNN model. The number of neurons in the output layer is 1 since we are dealing with a univariate binary outcome with two categories in the outcome variable.

(b) *Configuring and compiling the model.* At this stage of the training process, the loss function, the optimizer, and the metric for evaluating the prediction performance should be defined. Regarding the loss function, most of the time binary_crossentropy is suggested for binary outcomes, while categorical_crossentropy is suggested for categorical or ordinal data. As it was studied in the previous chapter, the mean_squared_error loss is the most popular for continuous outcomes. However, it is also possible to use the mean_squared_error loss for binary and categorical outcomes, but binary_crossentropy and categorical_crossentropy are the best choices when we are dealing with DNN models that output probabilities because they measure the distance between probability distributions, that is, between the ground truth distribution and the predictions obtained. As was mentioned in the previous chapter, the optimizer plays a really important role when updating model parameters (weights and biases). There is no specific optimizer for each type of response variable, and as we studied in the previous chapter, there are at least seven optimizers available in the Keras library. However, we usually use the Adam optimizer since it performs well in many cases. Finally, regarding the type of metric for evaluating the prediction performance for binary and categorical data, we use the accuracy metric which measures the proportion of cases that are correctly classified.

(c) *Fitting the model.* At this stage, we need to specify the number of epochs (i.e., the number of times the algorithm uses the entire training data set) and the batch size (size of the sample to be passed through the entire algorithm in each epoch) because if the training data consist of 1000 observations and we use a batch size = 50, we will need 20 iterations per epoch. Here, we should specify the validation split when you are in the tuning process (the value of the validation split is between 0 and 1) or specify the validation data set that should be used. For example, if you specified a validation_split = 0.3, this means that 30% of the original training data should be used as the validation set, and the remaining 70% of the observations will be used for training the model; the prediction performance of the model is evaluated with the validation set. Also, if you want to use the early stopping method, this should be specified in callbacks exactly as was done in the previous chapter for continuous outcomes.

(d) *Evaluating the prediction performance.* For binary outcomes, we suggest using the predict_classes() function that requires the information of the independent variables of the testing set as input for which the predictions are required. The predict_classes() function gives binary results (0 or 1) as output. However, practitioners can also use the predict() function, which provides probabilities for each category as outputs that need to be converted to 0 and 1 using a threshold value, for example, observations with probabilities larger or equal to 0.5 should be classified as 1 and observations with probabilities smaller than 0.5 should be classified as 0. Next, we provide the first illustrative example for training DNN with binary outcomes.

Example 12.1

Binary outcomes. This toy data set is called EYT and is composed of four environments (Bed5IR, EHT, Flat5IR, and LHT), 40 lines in each environment, and contains four traits (DTHD, DTMT, GY, and Height). Traits DTHD and DTMT are ordinal traits, GY is a continuous trait, and Height is a binary trait. This data set contains a genomic relationship matrix of 40 × 40 that corresponds to the similarity between lines.

The first eight observations of this data set are given below.

```
> head(Data_Pheno,8)
       GID   Env DTHD DTMT     GY Height
1 GID6569128 Bed5IR   1   1 6.119272    0
2 GID6688880 Bed5IR   2   2 5.855879    0
3 GID6688916 Bed5IR   2   2 6.434748    0
4 GID6688933 Bed5IR   2   2 6.350670    0
5 GID6688934 Bed5IR   1   2 6.523289    0
6 GID6688949 Bed5IR   1   2 5.984599    0
7 GID6689407 Bed5IR   1   2 6.436980    0
8 GID6689482 Bed5IR   3   3 6.052307    1
```

We can see that the ordinal traits (DTHD and DTMT) have three levels denoted as 1, 2, and 3, and the binary trait (Height) has two levels denoted as 0 and 1. We will create flags for the tuning process. The following code is used to create the flags; it is called Code_Tuning_With_Flags_Bin.R.

```
####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 ="sigmoid")

#####c) Compiling the DNN model
model %>% compile(
 loss = "binary_crossentropy",
 optimizer =optimizer_adam(lr=FLAGS$learning_rate),
 metrics =c('accuracy'))
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 a) are given the default flag values for % of dropout, number of units, activation function for hidden layers, batch size, number of epochs, learning rate, and validation split. In b) the DNN model is defined and the flag parameters are incorporated within our DNN model. It should be pointed out that the RELU activation function is used for the hidden layers, but the sigmoid activation function is used for the output layer to guarantee a probability as output between 0 and 1. Only one unit is specified for the output layer since we are interested in predicting only a univariate binary outcome. It is clear that our DNN model contains four hidden layers since the layer_dense() function was specified five times, but the last one corresponds to the output layer.

The model is compiled in part c) of the code, and the important things to note are that (a) now the loss function is binary_crossentropy, which is appropriate for binary response variables, (b) the optimizer specified was the Adam optimizer, optimizer_adam(), which is not specific for binary data, and (c) the metrics specified for evaluating the prediction performance is the accuracy that measures the proportion of correctly classified cases. In part d) the model is fitted using the number of epochs, the batch size, and the validation split specified in the flags (part a). In this case, the fitting process was done using the early stopping method.

Then, the above codes named "Code_Tuning_With_Flags_Bin.R" are called in the code given in Appendix 1. The code given in Appendix 1 executes the grid search using the library tfruns (Allaire 2018) with the tuning_run() function. The implemented grid search is shown below:

```
runs.sp<-tuning_run("Code_Tuning_With_Flags_Bin_4HL.R",runs_dir =
'_tuningE1',
flags=list(dropout1= c(0,0.05),
     units = c(67,100),
     activation1=("relu"),
     batchsize1=c(28),
     Epoch1=c(1000),
     learning_rate=c(Learn_val[e]),
     val_split=c(0.2)),sample=1,confirm =FALSE,echo =F)
```

The grid search is composed of only four combinations of hyperparameters that resulted from using two values of dropout (0, 0.05) and two units (67, 100). We used this small grid search because it is often not possible to do a full cartesian grid search with many values of each hyperparameter due to time and computational constraints. The code given in Appendix 1 was run five times, and each time it was run for a specific value of learning rate. It is important to note that the prediction performance is reported using not only the PCCC but also the Kappa coefficient, the sensitivity, and the specificity. Table 12.1 indicates that the best prediction performance was obtained using a learning rate (learn_val) of 0.01 across environments.

Table 12.1 Prediction performance for binary outcomes for five different values of learning rate using four hidden layers with the RELU activation function with five outer fold cross-validations and five inner fold cross-validations

learn_val	Env	PCCC	SE_PCCC	Kappa	SE_Kappa	Sensitivity	Specificity
0.001	Bed5IR	0.724	0.091	0.419	0.203	0.683	0.703
0.001	EHT	0.900	0.045	0.800	0.090	0.883	0.927
0.001	Flat5IR	0.654	0.043	0.191	0.122	0.704	0.500
0.001	LHT	0.513	0.058	0.010	0.127	0.523	0.510
0.01	**Bed5IR**	**0.792**	**0.072**	**0.590**	**0.138**	**0.827**	**0.767**
0.01	**EHT**	**0.900**	**0.063**	**0.779**	**0.139**	**0.943**	**0.883**
0.01	**Flat5IR**	**0.668**	**0.033**	**0.323**	**0.073**	**0.780**	**0.600**
0.01	**LHT**	**0.584**	**0.097**	**0.234**	**0.137**	**0.650**	**0.607**
0.1	Bed5IR	0.577	0.047	0.184	0.088	0.567	0.633
0.1	EHT	0.721	0.101	0.481	0.167	0.653	0.860
0.1	Flat5IR	0.604	0.044	0.146	0.117	0.860	0.372
0.1	LHT	0.576	0.070	0.189	0.131	0.750	0.574
0.5	Bed5IR	0.445	0.070	−0.013	0.013	0.429	0.470
0.5	EHT	0.484	0.094	0.164	0.109	0.513	0.717
0.5	Flat5IR	0.484	0.069	−0.032	0.032	0.608	0.220
0.5	LHT	0.392	0.089	0.080	0.080	0.519	0.413
1	Bed5IR	0.413	0.042	0.025	0.025	0.354	0.565
1	EHT	0.473	0.082	0.000	0.000	0.325	0.571
1	Flat5IR	0.462	0.078	0.000	0.000	0.625	0.353
1	LHT	0.402	0.092	0.000	0.000	0.374	0.421

12.2 Training DNN with Categorical (Ordinal) Outcomes

When training DNN for categorical or ordinal outcomes, the C levels of the response variable are 0, 1, 2, . . ., $C - 1$. For example, if the categorical or ordinal response variable has three levels (no infection, middle level of infection, and total level of infection), they should be denoted as 0, 1, and 2, where 0 denotes no infection, 1 middle level of infection, and 2 total level of infection. Another example is that assume you are interested in training a machine for classification with orange, mandarin, tangerine, and lemon as outcomes; you can denote orange with 0, mandarin with 1, tangerine with 2, and lemon with 3. Of course you can choose different values for each fruit, but because there are four categories, you will use 0, 1, 2, and 3 to denote the four fruits even though this is a nominal variable. Next, we provide some key elements to train this type of DNN models more efficiently.

Define the DNN model in Keras. The training process is equal to the training of binary response variables, except that in the output layer we suggest using the softmax activation function with a number of units equal to the number of categories; this guarantees that the output of each category is a probability between 0 and 1, and that the sum of these (all categories) probabilities is 1. Before starting the training process, you need to convert to dummy variables the categorical (or ordinal) response variable using the to_categorical() function that needs, as input, the vector of the categorical response variable and the number of classes that the response has. This way of coding the categorical and ordinal responses is called one-hot encoding or categorical encoding. It consists of embedding each level (label) of the categorical response variable as an all-zero vector with 1 in the place of the label index. For example, suppose that your response variable contains the following values: $y = (0, 2, 4, 1, 3, 0, 1, 3, 4, 2)$. Then the vector of response variable is transformed to $yf = to\_categorical(y, 5)$ that produces the following result:

y	*yf*				
0	1	0	0	0	0
2	0	0	1	0	0
4	0	0	0	0	1
1	0	1	0	0	0
3	0	0	0	1	0
0	1	0	0	0	0
1	0	1	0	0	0
3	0	0	0	1	0
4	0	0	0	0	1
2	0	0	1	0	0

This means that the dependent variable is no longer a vector, because it is a matrix of zeros and ones; for this reason, the number of units required for the output layer is equal to the number of classes. In this example, the number of units required for the output layer should be five.

(a) *Configuring and compiling the model*. The only difference when compiling binary response variables and ordinal (or categorical) outcomes is that now using the categorical_crossentroy loss as the loss function is recommended.
(b) *Fitting the model*. Everything that was explained for binary data also applies to ordinal and categorical outcomes.
(c) *Evaluating the prediction performance*. For ordinal and categorical outcomes, we suggest using the predict_classes() function because it produces, as output, values of the categorical or ordinal data in the scale of the response variable, that is, 0, 1, . . ., $C - 1$. However, you can also use the predict() function, which will provide you with probabilities for each category and the sum of all of them is equal to 1. These probabilities need to be converted to the original response variable (0, 1, . . ., $C - 1$). This conversion can be done by assigning each observation to the category with the largest probability (Allaire and Chollet 2019). Next, we provide one illustrative example for a DNN with an ordinal outcome.

Example 12.2
Ordinal outcome. This example uses the same data as Example 12.1 (Toy_EYT data set), but now we use the ordinal trait DTHD as the response variable. For the tuning process, we first created the flags which are given next and should be placed in a file called Code_Tuning_With_Flags_Ordinal_4HL2.R, as it is called in Appendix 2.

```
####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_batch_normalization() %>%
 layer_dropout(rate=FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, activation =FLAGS$activation1) %>%
 layer_batch_normalization() %>%
 layer_dropout(rate=FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, activation =FLAGS$activation1) %>%
 layer_batch_normalization() %>%
 layer_dropout(rate=FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, activation =FLAGS$activation1) %>%
 layer_batch_normalization() %>%
 layer_dropout(rate=FLAGS$dropout1) %>%
 layer_dense(units=3, activation ="softmax")

#####c) Compiling the DNN model
model %>% compile(
 loss = "categorical_crossentropy",
 optimizer =optimizer_adam(lr=FLAGS$learning_rate),
 metrics =c('accuracy'))
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 a) are given the default flag values for some hyperparameters; the DNN is defined in b) and is very similar to the definition of the DNN for binary outcomes, except that in the output layer the softmax activation function is used, which is appropriate for categorical or ordinal data, and now instead of one unit, three are used in the output layer since this is the number of classes of the DTHD ordinal response variable. Another important difference in the definition of the DNN model is that now we used the layer_batch_normalization() function just after specifying each hidden layer. The layer_batch_normalization() function is used to help with a problem called internal covariate shift that consists of changing the distribution of network activations due to the change in network parameters during training. Therefore, the layer_batch_normalization() function improves the training process by reducing the internal covariate shift by fixing the distribution of the layer inputs x as the training progresses (Ioffe and Szegedy 2015; LeCun et al. 1998; Wiesler and Ney 2011), and the internal covariate shift is reduced by linearly transforming the input to have zero means and unit variances and decorrelating the input information. This process is done in the input of each layer to fix the distributions of inputs that would remove the effects of the internal covariate shift (Ioffe and Szegedy 2015).

In part c) of the code, the DNN model is compiled; this is the same as the compilation process of binary outcomes, except that now a categorical_crossentropy loss function should be used because the response variable is ordinal or categorical. The fitting process, part d), is the same as that for binary outcomes.

Before using these flags, the response variable was converted to dummy variables using the to_categorical() function. Next, we show the first eight observations of the testing set of the first partition used in the code given in Appendix 2.

```
> y_tst= to_categorical(y[tst_set],nclas)
> cbind(y[tst_set],y_tst)
     [,1] [,2] [,3] [,4]
 [1,]   2    0    0    1
 [2,]   0    1    0    0
 [3,]   2    0    0    1
 [4,]   2    0    0    1
 [5,]   1    0    1    0
 [6,]   0    1    0    0
 [7,]   0    1    0    0
 [8,]   2    0    0    1
```

Here we observe that the first column corresponds to the original ordinal score of the response variable with levels 0, 1, and 2, that is, nclas = 3, while the remaining three columns are the three dummy variables created using the to_categorical() function since the original response variable has three types. From the output produced using the to_categorical() function, we can see that the first observation in the last column is a 1, while columns 2 and 3 have values of 0, since the original categorical response variable is 2. In the second observation, we can see that since the original categorical score is 0, the 1 appears in the second column, and values of 0 in the remaining columns. In the fifth observation, we can see that the original categorical score is 1; for this reason, in the third column, there is a 1 and in the remaining columns there is a value of 0.

The code given above is called Code_Tuning_With_Flags_Ordinal_4HL2.R in the code given in Appendix 2. The code given in Appendix 2 does the grid search using the library tfruns (Allaire 2018) and the tuning_run() function. The implemented grid search is shown below:

```
runs.sp<-tuning_run("Code_Tuning_With_Flags_Ordinal_4HL2.R",
runs_dir = '_tuningE1',
         flags=list(dropout1= c(0,0.05),
              units = c(67,100),
              activation1=("relu"),
              batchsize1=c(28),
              Epoch1=c(1000),
              learning_rate=c(Learn_val[e]),
              val_split=c(0.2)),sample=1,confirm =FALSE,echo =F)
```

Table 12.2 Prediction performance for ordinal outcomes for different values of learning rate with four hidden layers

Learn_val	Env	PCCC	SE_PCCC	Kappa	SE_Kappa	Sensitivity	Specificity
0.005	Bed5IR	0.692	0.085	0.535	0.109	0.783	0.625
0.005	EHT	0.693	0.075	0.439	0.082	0.542	0.167
0.005	Flat5IR	0.757	0.071	0.617	0.101	0.854	0.542
0.005	LHT	0.702	0.056	0.517	0.084	0.810	0.333
0.01	**Bed5IR**	**0.734**	**0.097**	**0.585**	**0.145**	**0.750**	**0.556**
0.01	**EHT**	**0.743**	**0.075**	**0.537**	**0.139**	**0.660**	**0.250**
0.01	**Flat5IR**	**0.702**	**0.089**	**0.529**	**0.137**	**0.850**	**0.567**
0.01	**LHT**	**0.718**	**0.055**	**0.519**	**0.084**	**0.658**	**0.750**
0.015	Bed5IR	0.691	0.111	0.502	0.169	0.710	0.333
0.015	EHT	0.689	0.053	0.462	0.103	0.625	0.167
0.015	Flat5IR	0.685	0.087	0.523	0.131	0.767	0.367
0.015	LHT	0.725	0.097	0.544	0.158	0.906	0.313
0.03	Bed5IR	0.684	0.087	0.500	0.118	0.883	0.542
0.03	EHT	0.733	0.037	0.373	0.167	0.900	0.167
0.03	Flat5IR	0.633	0.091	0.417	0.132	0.883	0.400
0.03	LHT	0.676	0.126	0.494	0.184	0.833	0.250
0.06	Bed5IR	0.720	0.097	0.503	0.177	0.704	0.500
0.06	EHT	0.713	0.033	0.425	0.078	0.733	0.333
0.06	Flat5IR	0.695	0.062	0.519	0.091	0.817	0.250
0.06	LHT	0.658	0.068	0.442	0.104	0.761	0.125

The grid search was used (sample = 1) for the tuning process and, for the four hyperparameter combinations (two values of dropout and two values of units), were evaluated.

Table 12.2 gives the results of implementing the code given in Appendix 2 for five different values of learning rate (0.005, 0.01, 0.015, 0.03, and 0.06), where the best predictions were obtained with a learning rate value of 0.01 across environments. These results give evidence that the prediction performance depends considerably on the value of the hyperparameter called learning rate.

12.3 Training DNN with Count Outcomes

Remember that count data (0, 1, 2, . . .) are usually modeled with Poisson regression or negative binomial regression in the statistical world. In the world of DNN, only the Poisson DNN model has been available until now in Keras and its key elements imitate those of the generalized linear models of the statistical world. For this reason, its construction in Keras uses as loss function the minus log-likelihood of a Poisson distribution; for the output layer, you can use the exponential activation function (inverse of log link in generalized linear models) that is available in Keras, which guarantees only positive outcomes. Also a RELU activation function can be used to guarantee a positive outcome.

In general, the definition of a DNN model for count outcomes in Keras is very similar to what we have studied before for continuous, binary, and categorical data for univariate responses. If we are interested in predicting only one response variable, we only need to specify one unit in the output layer, but if we are interested in predicting five count response variables, we need to specify a unit for each response we wish to predict. Also, the process of adding the hidden layers is exactly the same as was done for continuous, binary, and categorical (or ordinal) data with the same activation functions, for example, RELU for all the hidden layers, since the fitting process of the model is exactly the same as the fitting process of continuous, binary, and ordinal univariate outcomes. However, some of the key differences are in the compilation and prediction process. In the compilation process, we need to specify the "poisson" loss function that was created as the minus log-likelihood of the Poisson distribution, and for the metric, we can still use the mean squared error metric; however, for the prediction process, we should use the predict() function that will always produce positive values only if we specify an appropriate activation function in the output layer like the exponential activation function. Next, we provide an illustrative example for training DNN with count outcomes.

Example 12.3
Count data. This toy data set contains 115 lines, evaluated in three environments (Batan2012, Batan2014, and Chunchi2014) and in each environment two blocks were created. The total number of observations of this data set is 649. The data set is denoted as Data_Count_Toy.RData. The count response variable has a minimum value of 0 and a maximum value of 17.

Modeling and predicting count data is not only important in plant breeding, but also very common in areas such as health, finance, social science, etc. Generalized linear models have been widely used for modeling count response variables, but many times fail to capture complex data patterns. For this reason, nonlinear Poisson regressions under the umbrella of deep artificial neural networks are of paramount importance for modeling count data and improving the prediction accuracy. The details for implementing DNN models for count data are given below.

The tuning process was done by creating flags which are given below; they should be placed in a file named: Code_Tuning_With_Flags_Count_Lasso.R, that is used in Appendix 3.

```
####a) Declaring the flags for hyperparameters
FLAGS = flags(
 flag_numeric("dropout1", 0.0),
 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),
 flag_numeric("Lasso_par",0.001))

####b) Defining the DNN model
build_model<-function() {
model <- keras_model_sequential()
model %>%
 layer_dense(units =FLAGS$units, kernel_regularizer=regularizer_l1
(FLAGS$Lasso_par),activation =FLAGS$activation1, input_shape = c(dim
(X_trII)[2])) %>%
 layer_dropout(rate=FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, kernel_regularizer=regularizer_l1
(FLAGS$Lasso_par),activation =FLAGS$activation1) %>%
 layer_dropout(rate=FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, kernel_regularizer=regularizer_l1
(FLAGS$Lasso_par),activation =FLAGS$activation1) %>%
 layer_dropout(rate=FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, kernel_regularizer=regularizer_l1
(FLAGS$Lasso_par),activation =FLAGS$activation1) %>%
 layer_dropout(rate=FLAGS$dropout1) %>%
 layer_dense(units=1, activation ="exponential")

#####c) Compiling the DNN model
model %>% compile(
 loss = "poisson",
 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 =30)

###########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=T,
 validation_split =FLAGS$val_split,
 verbose=0,callbacks = list(early_stop,print_dot_callback))
```

Again, in part a) are defined the default flags, in part b) the DNN for count data is defined, where the significant difference is that the exponential activation function is

Table 12.3 Prediction performance of count data for different values of regularization with four hidden layers and Ridge regularization

relularization value	Environment	Trait	MSE	SE_MSE	MAAPE	SE_MAAPE
0.005	Batan2012	Count	1.787	0.2466	0.8752	0.0276
0.005	Batan2014	Count	1.7613	0.1572	0.8864	0.0171
0.005	Chunchi2014	Count	15.0395	2.2348	0.6663	0.0122
0.01	Batan2012	Count	2.5557	0.1534	0.9026	0.0467
0.01	Batan2014	Count	2.2408	0.0878	0.9093	0.0109
0.01	Chunchi2014	Count	13.9118	2.176	0.5964	0.0227
0.015	Batan2012	Count	1.8293	0.3306	0.8662	0.0378
0.015	Batan2014	Count	1.7998	0.2559	0.8874	0.0226
0.015	Chunchi2014	Count	15.1706	1.8295	0.6334	0.0153
0.03	Batan2012	Count	2.2634	0.2003	0.9072	0.0295
0.03	Batan2014	Count	2.2277	0.2537	0.9055	0.0174
0.03	Chunchi2014	Count	13.0655	1.7221	0.5856	0.0199
0.06	Batan2012	Count	2.2384	0.2008	0.8975	0.016
0.06	Batan2014	Count	2.0308	0.2055	0.9078	0.0275
0.06	Chunchi2014	Count	13.8051	1.3726	0.6511	0.0391

used for the output layer; we should point out that in each hidden layer, the Lasso (L1) regularization is also used in addition to dropout. In part c) the relevant parts are (1) the specification of the Poisson loss function and (2) the use of the mean squared error as a metric for evaluating the prediction performance, while the fitting process is exactly the same as was done for continuous, binary, and categorial or ordinal outcomes.

Then, the flags above are called in Appendix 3 that used the following grid search:

```
runs.sp<-tuning_run("Code_Tuning_With_Flags_Count_Lasso.R",
            runs_dir = '_tuningE1',
            flags=list(dropout1= c(0),
            units = c(67,150),
            activation1=("relu"),
            batchsize1=c(28),
            Epoch1=c(1000),
            learning_rate=c(Learn_val[e]),
            val_split=c(0.2),
          Lasso_par=c(0.001,0.01)), sample=1,confirm =FAL SE,echo =F)
```

From the above random grid search, we observe that four is the total number of hyperparameters that form the grid (two regularization parameters and two units) and should be evaluated. The code given in Appendix 3 was used to get the results given in Table 12.3, as well as for evaluating the prediction performance with five regularization values (0.005, 0.01, 0.015, 0.03, and 0.06) and with five outer fold and five inner fold cross-validations and with four hidden layers.

12.4 Training DNN with Multivariate Outcomes

Training models with multivariate outcomes are very important for plant breeders since they are interested in predicting more than one trait. In the context of plant breeding, multivariate models for the prediction of more than one trait simultaneously are called multi-trait models. There is evidence that multi-trait models capture the complex relationships between traits more efficiently and, for this reason, many times they improve the prediction performance when compared with univariate models. Statistical multi-trait models capture the correlation between traits and also the correlation between lines.

There is evidence, and not only in genomic selection, that the larger the correlation between traits, the better the prediction performance of multi-trait analysis (Jia and Jannink 2012; Jiang et al. 2015). Authors like He et al. (2016) and Schulthess et al. (2017) found a significant improvement of multi-trait analysis with regard to univariate analysis, while Calus and Veerkamp (2011), Montesinos-López et al. (2016), Montesinos-López et al. (2018a, b), and Montesinos-López et al. (2019) found modest improvement of multivariate analysis when compared to univariate analysis. Also in the context of multi-trait models, it helps to clarify the relationship and the effect of each studied independent variable on the dependent multivariate variables (Castro et al. 2013; Huang et al. 2015).

Despite the positive advantages of statistical multi-trait models mostly for continuous outcomes, it has not been possible to develop efficient models for other types of multivariate response variables (binary, categorical, and count) and efficient models for mixed outcomes (continuous, binary, categorical, and count) are still lacking. There have been some developments in the statistical literature, but most of them are not efficient for large data sets. However, as shown in two publications by Montesinos-López et al. (2018b, c) and Montesinos-López et al. (2019), the deep learning methodology has the power to efficiently implement univariate and multivariate models for each type of response variables, and even for mixed outcomes (Chollet and Alliare 2017). For this reason, in this section, we will show how to implement multi-trait analysis for continuous outcomes, multi-trait binary outcomes, multi-trait categorical outcomes, multi-trait count outcomes, and multi-trait mixed outcomes with a combination of at least two types of response variables (continuous and binary; continuous and categorical; continuous and count; categorical and count; etc.) and even with four types of response variables for continuous, binary, categorical, and count outcomes. Next, we will illustrate the DNN training process with multi-trait outcomes with all traits as continuous.

12.4.1 DNN with Multivariate Continuous Outcomes

The data set used for illustrating how to train multivariate continuous outcomes is called MaizeToy data set. This data set contains 30 lines that were measured in three environments (EBU, KAT, and KTI); for this reason, the total number of observations is 90. Also, three continuous traits were measured for each observation, and these traits were Yield, ASI, and PH. The genomic information is contained in the genomic relationship matrix denoted as genoMaizeToy.R.

Next, we provide the default flags for training continuous outcomes. These flags should be placed in the R (R Core Team 2019) code called Code_Tuning_With_Flags_MT_normal.R.

```
####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 multi-trait DNN model
input <- layer_input(shape=dim(X_trII)[2],name="covars")

# add hidden layers
base_model <- input %>%
 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)

# add output 1
yhat1 <- base_model %>%
 layer_dense(units=1, name="response_1")

# add output 2
yhat2 <- base_model %>%
 layer_dense(units= 1, name="response_2")

# add output 3
yhat3 <- base_model %>%
 layer_dense(units= 1,name="response_3")

#c) Compiling the multi-trait model
model <- keras_model(input,list(response_1=yhat1,response_2=yhat2,
response_3=yhat3)) %>% compile(optimizer =optimizer_adam
(lr=FLAGS$learning_rate),
    loss=list(response_1="mse",response_2="mse", response_3="mse"),
     metrics=list(response_1="mse",response_2="mse",
response_3="mse"),
     loss_weights=list(response_1=0.99,response_2=1.8,
response_3=0.069))
```

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

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

# d) Fitting multi-trait model
model_fit <- model %>%
 fit(x=X_trII,
   y=list(response_1=y_trII[,1],response_2=y_trII[,2],
response_3=y_trII[,3]),
   epochs=FLAGS$Epoch1,
   batch_size =FLAGS$batchsize1,
   validation_split=FLAGS$val_split,
   verbose=0, callbacks = list(early_stop,print_dot_callback))
```

In part a) the default flags which are defined exactly as for univariate outcomes are defined. In part b) the multi-trait DNN model is defined, with layer_input(), the dimension (number of independent variables) of the input information is then provided and the hidden layers are added. In this case, only three hidden layers were specified and each layer had dropout that was added with layer_dropout(). Then three output layers were added that correspond to each of the three traits of the mult-trait DNN model. The three output layers use one unit and the linear activation function (not necessary to specify which) since the three traits are assumed to be continuous. Next, in part c), the compilation process is done; here, as in univariate DNN models, we need to specify the loss function, the optimizer, and the metrics used to evaluate the prediction performance. The specified optimizer is exactly the same as in the univariate DNN models. However, for the specification of the loss function and metrics we need to specify a loss function and metrics for each trait (outcome), that need to be in agreement with the type of response of each trait. Since in this example we have three continuous traits, we specified as a loss function and metric for each trait the mean_squared error (mse); however, for traits with different types of outcome, the practitioner should specify a loss function and metric appropriate for each type of trait. Two differences in the compilation process of multi-trait DNN models with regard to univariate DNN models are found. The first one is that in the Keras we need to specify the traits under study using a list() where, separated by a comma, the names of the traits under study are provided. The second one is that we need to specify the loss_weights for each trait. One simple approach is to use the same weights for all traits: for example, (1,1,1), if we have three traits or (0.3333,0.3333, 0.3333), this approach is valid when all traits are on the same scale, but when traits are on different scales, we suggest using different weights for each continuous trait. In this example, different weights were used, since each trait has a different scale and the weights were built as follows: (1) first we calculated the median of each trait, (2) then we calculated the 0.25 and 0.75 quantiles for each trait, (3) then we calculated the maximum distance in terms of absolute value between the median and both quantiles, (4) then we used as the weight for the first trait (GY) its calculated distance, and (5) then we used as weight for the second trait the value obtained by dividing the distance of the first trait by the distance of the second trait, and finally the weight for the third trait was also obtained by dividing the distance of the first trait by the distance of the third trait. Finally, the fitting process is done in part d), and it is the same as the fitting process for the univariate DNN model, except that the training set of the response variables is provided as a list. These steps are only suggestions that can work for some data sets, but there is no guarantee that they can work for all data sets.

Next, we called the flag Code_Tuning_With_Flags_MT_normal.R. In Appendix 4, it is used to implement the tuning process for selecting the best combination of hyperparameters, and after selecting the best combination of hyperparameters, the multi-trait DNN with the optimum hyperparameters is refitted, and the prediction performance using cross-validation is evaluated with this refitted model. The grid search implemented in this code is given next:

```
 runs.sp<-tuning_run("Code_Tuning_With_Flags_MT_normal.R",runs_dir
= '_tuningE1',
            flags=list(dropout1= c(0,0.05),
                 units = c(56,97),
                 activation1=("relu"),
                 batchsize1=c(22),
                 Epoch1=c(1000),
                 learning_rate=c(0.001),
               val_split=c(0.25)), sample=0.5, confirm =FALSE,echo =F)
```

The results of implementing the last random grid search (sample = 0.5) that consists of four combinations of hyperparameters resulting from two dropout values (0, 0.05) and two values of units (56, 97) are given in Table 12.4, which shows that the best predictions were obtained for trait PH in terms of Pearson's correlation and MAAPE.

12.4.2 DNN with Multivariate Binary Outcomes

For illustrating the process of training multivariate binary DNN models, we used the same data set (Data_Toy_EYT.RData) as in Example 12.1 in this chapter. As was explained in Example 12.1, this data set contains four environments (Bed5IR, EHT, Flat5IR, and LHT), 40 lines in each environment, a genomic relationship matrix of order 40 × 40, four traits (DTHD, DTMT, GY, and Height) of which DTHD and DTMT are ordinal traits, trait GY is continuous, and trait Height is binary. However, for the implementation of the multivariate binary DNN model, we converted ordinal traits DTHD and DTMT into binary traits, making 0 the levels of 1, and 1 the levels

Table 12.4 Prediction performance of three continuous traits with three hidden layers without genotype × environment interaction

Environment	Trait	Pearson	SE_Pearson	MAAPE	SE_MAAPE	MSE	SE_MSE
EBU	Yield	0.2643	0.1593	0.1797	0.0214	1.871	0.3754
KAK	Yield	0.2834	0.1287	0.0835	0.0134	0.3899	0.1735
KTI	Yield	0.088	0.1992	0.1756	0.0158	1.5099	0.3677
EBU	ASI	0.0924	0.2167	0.2936	0.0324	0.5816	0.1267
KAK	ASI	−0.0234	0.2562	0.6847	0.0597	1.2333	0.2179
KTI	ASI	−0.1824	0.2311	0.2631	0.0314	1.1295	0.4023
EBU	PH	0.4658	0.149	0.0628	0.0122	351.9016	158.214
KAK	PH	0.5608	0.0711	0.0346	0.0028	87.6928	9.4417
KTI	PH	0.4839	0.1397	0.0613	0.0106	289.4436	83.5068

of 2 and 3. For this reason, the illustration of the multivariate binary DNN model was done using the following three traits: DTHD, DTMT, and Height.

The default flags for training multivariate binary outcomes are given next. They should be put in the R code (R Core Team 2019) called Code_Tuning_With_Flags_MT_Binary.R.

```
####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 multi-trait DNN model
input <- layer_input(shape=dim(X_trII)[2],name="covars")

# add hidden layers
base_model <- input %>%
 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)

# add output 1
yhat1 <- base_model %>%
 layer_dense(units=1,activation="sigmoid", name="response_1")

# add output 2
yhat2 <- base_model %>%
 layer_dense(units= 1, activation="sigmoid",name="response_2")

# add output 3
yhat3 <- base_model %>%
 layer_dense(units= 1, activation="sigmoid",name="response_3")

#c) Compiling the multi-trait model
model <- keras_model(input,list(response_1=yhat1,response_2=yhat2,
response_3=yhat3)) %>%
 compile(optimizer =optimizer_adam(lr=FLAGS$learning_rate),
     loss=list(response_1="binary_crossentropy",
response_2="binary_crossentropy",response_3=
"binary_crossentropy"),
     metrics=list(response_1="accuracy",response_2="accuracy",
response_3="accuracy"),
     loss_weights=list(response_1=1,response_2=1,response_3=1))
###1,3.2,0.024
```

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

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

# d) Fitting multi-trait model
model_fit <- model %>%
 fit(x=X_trII,
   y=list(response_1=y_trII[,1],response_2=y_trII[,2],
response_3=y_trII[,3]),
   epochs=FLAGS$Epoch1,
   batch_size =FLAGS$batchsize1,
   validation_split=FLAGS$val_split,
   verbose=0, callbacks = list(early_stop,print_dot_callback))
```

Also in part a) are given the default flags for implementing the multi-trait binary DNN model. In part b) is defined the multi-trait binary DNN model where we can see that the process of adding the dimension of the input and the hidden layers is exactly the same as for the multi-trait continuous DNN model. Also, the number of output layers depends on the number of traits under study, that is, if there are three traits under study, we need to specify three output layers with one unit for each output layer, as was done in continuous multivariate outcomes, but the key difference is that now for binary multivariate outcomes, instead of using the linear activation function in the output layer, we use the sigmoid activation function for each of the output layers, since we are dealing with binary multivariate outcomes. The multi-trait binary DNN model is compiled in part c); this process is very similar to the compilation process of a multi-trait continuous DNN model, except that now we use the binary_crossentropy loss function for each trait under study, we also use accuracy as the metric for each of the binary traits and we use the same weight (in this case, 1, 1, and 1) for each of the three traits. We now use the same weights because the three traits under study are in the same type of response variable and scale. Finally, in part d) is given the code for the fitting process, which is exactly the same as for multi-trait continuous DNN models.

These flags (Code_Tuning_With_Flags_MT_Binary.R) can be used with Appendix 4 with some small modifications such as those just described above, which are related to (1) the activation function used for the output layers, (2) the loss function used for each trait since now we should use binary_crossentropy, (3) the metrics used for each trait since now we should use accuracy for each trait, (4) the weights used in the compilation process since now we will use the same weight for each trait, and (5) the prediction process since we will replace the following code of Appendix 4:

```
# predict values for test set
Yhat<-predict(model,X_tst)%>%
 data.frame()%>%
 setNames(colnames(y_trn))
YP=Yhat
```

With the following code to guarantee binary predictions:

```
# predict values for test set
Yhat<-predict(model,X_tst)%>%
 data.frame()%>%
 setNames(colnames(y_trn))
YP=matrix(NA,ncol=ncol(y2),nrow=nrow(Yhat))
head(Yhat)
P_T1=ifelse(Yhat[,1]>0.5,1,0)

P_T2=ifelse(Yhat[,2]>0.5,1,0)
P_T3=ifelse(Yhat[,3]>0.5,1,0)
YP[,1]=P_T1
YP[,2]=P_T2
YP[,3]=P_T3
```

The random grid search was used since sample = 0.5, for the prediction under a multi-trait binary DNN model, as is shown next:

```
runs.sp<-tuning_run("Code_Tuning_With_Flags_MT_Binary.R",runs_dir
= '_tuningE1',
          flags=list(dropout1= c(0,0.05),
                units = c(56,97),
                activation1=("relu"),
                batchsize1=c(22),
                Epoch1=c(1000),
                learning_rate=c(0.001),
               val_split=c(0.25)), sample=0.5,confirm =FALSE,echo =F)
```

The important thing about this random grid search is that the tuning_run() function calls the flags developed above for the training process of multi-trait binary DDN models. Also, the total number of hyperparameters of the grid search is four, since we set a unique value for all the hyperparameters, except for the hyperparameters dropout with two values (0, 0.05) and the units with another two values (56, 97). It is important to point out that of the four total hyperparameter combinations, only two were evaluated, since we implemented a random grid search by specifying sample = 0.5.

In Table 12.5 is given the prediction performance for the average of the five outer fold cross-validations used for training the multi-trait binary DNN model, the prediction performance reported as metrics, the PCCC, the Kappa coefficient, the sensitivity, and the specificity. From these results, it is evident that the best predictions belong to trait DTHD and trait DTMT. For all trait-environment

Table 12.5 Prediction performance of multivariate binary data for three hidden layers with genotype × environment interaction and with five outer fold cross-validations

Trait	Env	PCCC	SE_PCCC	Kappa	SE_Kappa	Sensitivity	Specificity
DTHD	Bed5IR	0.800	0.109	0.533	0.209	0.700	0.888
DTHD	EHT	0.775	0.047	0.452	0.104	0.767	0.808
DTHD	Flat5IR	0.775	0.061	0.437	0.192	0.700	0.819
DTHD	LHT	0.750	0.040	0.344	0.140	0.693	0.751
DTMT	Bed5IR	0.800	0.109	0.533	0.209	0.888	0.700
DTMT	EHT	0.775	0.047	0.452	0.104	0.808	0.767
DTMT	Flat5IR	0.775	0.061	0.437	0.192	0.819	0.700
DTMT	LHT	0.750	0.040	0.344	0.140	0.751	0.693
Height	Bed5IR	0.575	0.031	0.115	0.065	0.520	0.607
Height	EHT	0.850	0.061	0.670	0.123	0.760	0.910
Height	Flat5IR	0.725	0.073	0.363	0.152	0.850	0.550
Height	LHT	0.750	0.040	0.492	0.074	0.800	0.767

combinations, the PCCC was larger than 0.5, that is, larger than the probability of a correct classification by chance since we are dealing with binary outcomes.

12.4.3 DNN with Multivariate Ordinal Outcomes

To illustrate the training of multivariate ordinal or categorical DNN models, we used the same data set (Data_Toy_EYT.RData) used in this chapter in Example 12.1 and before for training multivariate binary DNN models. However, now the implementation of the multivariate ordinal DNN model was done with traits DTHD, DTMT, and GY, but since GY is a continuous trait, this was converted to an ordinal outcome in the following way: if GY is less than 3.2, then the outcome was set to 0, but if $3.2 < GY < 5.8$, the outcome was set to 1, while if $5.8 < GY < 6.2$, the ordinal outcome was set to 2, and finally, if $GY > 6.2$, the outcome was set to 3. This means the training of the multivariate ordinal DNN model was done with three traits (DTHD, DTMT, and GY) where the first two had three levels and the last one had four levels.

Next are provided the default flags that we suggest placing in an R (R Core Team 2019) file called Code_Tuning_With_Flags_MT_Ordinal.R.

```
####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 multi-trait ordinal DNN model
input <- layer_input(shape=dim(X_trII)[2],name="covars")

# add hidden layers
base_model <- input %>%
 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)

# add output 1
yhat1 <- base_model %>%
 layer_dense(units=3,activation="softmax", name="response_1")

# add output 2
yhat2 <- base_model %>%
 layer_dense(units= 3, activation="softmax",name="response_2")

# add output 3
yhat3 <- base_model %>%
 layer_dense(units= 4, activation="softmax",name="response_3")

#c) Compiling the multi-trait ordinal DNN model
model <- keras_model(input,list(response_1=yhat1,response_2=yhat2,
response_3=yhat3)) %>%compile(optimizer =optimizer_adam
(lr=FLAGS$learning_rate),
loss=list(response_1="categorical_crossentropy",
response_2="categorical_crossentropy",
response_3="categorical_crossentropy"),
metrics=list(response_1="accuracy",response_2="accuracy",
response_3="accuracy"),
loss_weights=list(response_1=1,response_2=1,response_3=1))

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

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

# d) Fitting multi-trait model
model_fit <- model %>%
 fit(x=X_trII,
   y=list(response_1=y_trII[,1],response_2=y_trII[,2],
response_3=y_trII[,3]),
   epochs=FLAGS$Epoch1,
   batch_size =FLAGS$batchsize1,
   validation_split=FLAGS$val_split,
   verbose=0, callbacks = list(early_stop,print_dot_callback))
```

The default flags for implementing the multi-trait ordinal DNN model are in part a). In part b) is defined the multi-trait categorical DNN model, for which its implementation is equal to binary outcomes, with two exceptions: (1) now the softmax activation function is used for each trait in the output layer and (2) the number of units in each output layer depends on the number of categories of each trait under study, in this case, 3, 3, and 4 for the first, second, and third traits, respectively. With regard to the compilation process (part c), it is the same as the compilation of the multi-trait binary DNN model, except that now the categorical_crossentropy loss function is used for each trait under study. Finally, the code for the fitting process (part d) is exactly the same as for multi-trait binary DNN models.

The flags given above for training multivariate ordinal outcomes (Code_Tuning_With_Flags_MT_Ordinal.R) can be used with Appendix 4 with the following modifications: (1) use the softmax activation function for each output layer, (2) use the categorical_crossentropy for each trait, (3) use accuracy as the metric for each trait, (4) use the same weights for each trait, and (5) replace the following code of Appendix 4:

```
# predict values for test set
Yhat<-predict(model,X_tst)%>%
 data.frame()%>%
 setNames(colnames(y_trn))
YP=Yhat
```

Use the following code to obtain categorical outcomes as predictions:

```
Yhat<-predict(model,X_tst)%>%
 data.frame()%>%
 setNames(colnames(y_trn))
YP=matrix(NA,ncol=ncol(y),nrow=nrow(Yhat))
head(Yhat)
P_T1=(apply(data.matrix(Yhat[,1:3]),1,which.max)-1)
P_T2=(apply(data.matrix(Yhat[,4:6]),1,which.max)-1)
P_T3=(apply(data.matrix(Yhat[,7:10]),1,which.max)-1)
YP[,1]=P_T1
YP[,2]=P_T2
YP[,3]=P_T3
```

The random grid search (since sample = 0.5) for the prediction of a multi-trait categorical DNN model is given next:

```
 runs.sp<-tuning_run("Code_Tuning_With_Flags_MT_Ordinal.R",
runs_dir = '_tuningE1',
           flags=list(dropout1= c(0,0.05),
                 units = c(56,97),
                 activation1=("relu"),
                 batchsize1=c(22),
                 Epoch1=c(1000),
                 learning_rate=c(0.001),
                val_split=c(0.25)), sample=0.5,confirm =FALSE,echo =F)
```

Table 12.6 Prediction performance of multivariate ordinal data for three hidden layers with genotype × environment interaction and five outer fold cross-validation

Trait	Env	PCCC	SE_PCCC	Kappa	SE_Kappa	Sensitivity	Specificity
DTHD	**Bed5IR**	**0.625**	**0.0685**	**0.408**	**0.0922**	**0.6**	**0.5**
DTHD	**EHT**	**0.625**	**0.0884**	**0.398**	**0.1414**	**0.6833**	**0.25**
DTHD	**Flat5IR**	**0.6**	**0.0919**	**0.426**	**0.1224**	**0.7467**	**0.55**
DTHD	LHT	0.575	0.0637	0.287	0.0855	0.6167	0.0833
DTMT	Bed5IR	0.525	0.0612	0.246	0.1179	0.8	0.3917
DTMT	EHT	0.65	0.0729	0.444	0.1136	0.4583	0.6533
DTMT	Flat5IR	0.575	0.05	0.391	0.0448	0.78	0.3083
DTMT	LHT	0.55	0.0848	0.24	0.1589	0.7917	0.3958
GY	Bed5IR	0.4	0.1275	0.141	0.1666	NaN	0.3
GY	EHT	0.625	0.0395	0.23	0.0949	NaN	0.2
GY	Flat5IR	0.45	0.075	0.156	0.0846	NaN	0.4333
GY	LHT	0.425	0.0306	−0.06	0.0579	0.5667	0.3733

This tuning process was implemented using a random grid search since sample = 0.5 was specified inside the running_run() function, which means that only two of the four hyperparameter combinations should be evaluated at each iteration of the deep learning algorithm.

The results of implementing the multi-trait ordinal DNN model using the best combination of hyperparameters resulting from the above research grid are given in Table 12.6, where we can see that the metrics used for evaluating the prediction performance were the same as those used for the multi-trait binary outcomes, but now the best predictions were observed in trait DTHD and the worst in trait GY. However, note that by chance alone now for traits DTHD and DTMT the probability is 1/3 while for trait GY this probability is ¼ since for the first two traits there are three levels, while for the third trait there are four response options.

12.4.4 DNN with Multivariate Count Outcomes

To illustrate the training process of the multivariate count DNN model, we used the data called Data_Multi_Count_Toy.RData, which is a modified version of the data called Data_ Count_Toy.RData. The basic modification is that the modified version has two traits instead of one, which are denoted as y1 and y2. For this reason, again this data set contains 115 lines, evaluated in three environments (Batan2012, Batan2014, and Chunchi2014) and the total number of observations of this data set is 298. Next, we provide the default flags that we suggest placing in an R file called Code_Tuning_With_Flags_MT_Count.R.

```
####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 multi-trait count DNN model
input <- layer_input(shape=dim(X_trII)[2],name="covars")

# add hidden layers
base_model <- input %>%
 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)

# add output 1
yhat1 <- base_model %>%
 layer_dense(units=1,activation="exponential", name="response_1")

# add output 2
yhat2 <- base_model %>%
 layer_dense(units=1, activation="exponential",name="response_2")

#c) Compiling the multi-trait ordinal DNN model
model <- keras_model(input,list(response_1=yhat1,response_2=yhat2))
%>%
 compile(optimizer =optimizer_adam(lr=FLAGS$learning_rate),
     loss=list(response_1="poisson",response_2="poisson"),
     metrics=list(response_1="mse",response_2="mse"),
     loss_weights=list(response_1=1,response_2=1))

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

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

# d) Fitting multi-trait count DNN model
model_fit <- model %>%
 fit(x=X_trII,
   y=list(response_1=y_trII[,1],response_2=y_trII[,2]),
   epochs=FLAGS$Epoch1,
   batch_size =FLAGS$batchsize1,
   validation_split=FLAGS$val_split,
   verbose=0, callbacks = list(early_stop,print_dot_callback))
```

The default flags for the multi-trait count DNN model are in part a). The process of building the multi-trait count DNN model (part b) is very similar to the building of continuous multivariate outcomes, except that now the exponential activation function should be used. The compilation process (part c) is the same as the compilation of the multi-trait continuous DNN model, except that now the Poisson loss function is used for each trait under study, and the fitting process (part d) is exactly the same as for multi-trait continuous DNN models.

The flags given above for training multivariate count outcomes (Code_Tuning_With_Flags_MT_Count. R) can be used with Appendix 4 with the following modifications: (1) use the exponential activation function for each output layer, (2) use the Poisson loss function for each trait, and (3) use the same weights for each trait.

Next, we give the random grid search used for tuning a multi-trait count DNN model:

```
  runs.sp<-tuning_run("Code_Tuning_With_Flags_MT_Count.R",runs_dir
= '_tuningE1',
           flags=list(dropout1= c(0,0.05),
                units = c(56,97),
                activation1=("relu"),
                batchsize1=c(22),
                Epoch1=c(1000),
                learning_rate=c(0.001),
               val_split=c(0.2)),sample=0.5,confirm =FALSE,echo =F)
```

Again, we used the tuning_run() function to perform the tuning process which now was done with a random grid search since we specified sample = 0.5. Here the total number of hyperparameters is four, since only hyperparameters % of dropout and number of units have two values.

The prediction performance of training the multi-trait count data is given in Table 12.7. Now the metrics used for evaluating the accuracy were the mean square error of prediction and MAAPE. The best predictions across environments belong to

Table 12.7 Prediction performance of multivariate count data for three hidden layers with genotype × environment interaction and five outer fold cross-validation

Trait	Environment	MAAPE	SE_MAAPE	MSE	SE_MSE
y1	Batan2012	0.7975	0.0498	1.6228	0.1696
y1	Batan2014	0.6739	0.0648	1.175	0.3795
y1	Chunchi2014	0.7933	0.0517	22.2328	3.3194
y2	Batan2012	0.9184	0.0534	1.0801	0.3943
y2	Batan2014	0.7897	0.0619	1.0169	0.5013
y2	Chunchi2014	0.5128	0.0254	9.5645	0.7366

trait y1 in environments Batan2012 and Batan2014, and to trait y2 in environment Chunchi2014. Results given in Table 12.7 belong to the best combination of hyperparameters of the above grid. The comparison of prediction performance between traits in MSE terms is not valid when the traits are on different scales.

12.4.5 DNN with Multivariate Mixed Outcomes

Finally, in this chapter, we provided key elements for training DNN models for binary, categorical, count, and continuous outcomes. However, with the implementation of these DNN models, it is clear that DNN models are a novel tool for training univariate and multivariate genomic prediction models for binary, categorical, count, and mixed outcomes. The power to train univariate and multivariate nonlinear regression with count data is unique to deep learning models since similar tools in conventional statistical learning are not efficient and only a few are available. But the gain in training multivariate DNN models is only due to the increase in the sample size by modeling more than one trait simultaneously since the DNN models just studied do not take into account a variance–covariance matrix of traits to capture the correlation between traits. However, we also emphasize that the training process of DNN models is very challenging since more time, thought, experimentation, and resources are required for training these models than other statistical machine learning models studied in this book, since for most of the statistical machine learning algorithms, the search space for finding the optimum combination of hyperparameters is small compared to DNN models. For these reasons, we strongly suggest using the random grid search (or other new approaches like Bayesian optimization or genetic algorithms) since the larger the number of hyperparameters, the bigger the number of hyperparameter combinations that need to be evaluated due to the quick explosion in the number of hyperparameter combinations. For this reason, the use of the random grid search that explores only a fraction of the total combination of hyperparameters is more efficient.

Next the default flags are given in an R file called Code_Tuning_With_Flags_MT_Mixed.R.

```
####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
input <- layer_input(shape=dim(X_trII)[2],name="covars")
```

```
# add hidden layers
base_model <- input %>%
 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)

# add output 1
yhat1 <- base_model %>%
 layer_dense(units = 3,activation="softmax", name="response_1")

# add output 2
yhat2 <- base_model %>%
 layer_dense(units = 1, activation="exponential",name="response_2")

# add output 3
yhat3 <- base_model %>%
 layer_dense(units = 1, name="response_3")

# add output 3
yhat4 <- base_model %>%
 layer_dense(units =1, activation="sigmoid",name="response_4")

#c) Compiling the multi-trait mixed DNN model
Model=keras_model(input,list(response_1=yhat1,response_2=yhat2,
response_3=yhat3,response_4=yhat4)) %>%
 compile(optimizer =optimizer_adam(lr=FLAGS$learning_rate),
loss=list(response_1="categorical_crossentropy",response_2="mse",
response_3="mse",response_4="binary_crossentropy"), metrics=list
(response_1="accuracy",response_2="mse",response_3="mse",
response_4="accuracy"), loss_weights=list(response_1=1,
response_2=1,response_3=1,response_4=1))

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

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

# d) Fitting multi-trait mixed DNN model
model_fit <- model %>%
fit(x=X_trII,y=list(response_1=y_trII[,1],response_2=y_trII[,2],
response_3=y_trII[,3],response_4=y_trII[,4]), epochs=FLAGS$Epoch1,
batch_size =FLAGS$batchsize1, validation_split=FLAGS$val_split,
verbose=0, callbacks =list(early_stop,print_dot_callback))
```

The default flags for the multi-trait mixed outcome DNN model are in part a). The building process of the multi-trait mixed outcome DNN model (part b) is different only in the specification of the outputs. Since now we have specified an activation function for each response variable, the first response variable is an ordinal output with three categories. For this reason, we specified three neurons and the softmax activation function. The second response variable is a count, and for this reason, we specified one unit (neuron) and the exponential activation function. The third response variable is continuous, and for this reason, we specified only one neuron and the linear activation function (default activation function). Finally, the last response variable is binary, and for this reason, we specified only one neuron and the sigmoid activation function. With regard to the compilation process (part c), we had to specify a mixture loss function due to the fact that we had multivariate mixed outcomes. For this reason, the mixture loss function in this example is composed of four types of losses: "categorical_crossentropy," "mse," "mse," and "binary_crossentropy," which are appropriate for a mixed outcome of ordinal, count, continuous, and binary response variables. The same type of mixture is required for specifying the metrics to evaluate the prediction accuracy to guarantee that they are in agreement with the response variables. For this reason, in this case, the specification of the metrics contains: "accuracy," "mse," "mse," and "accuracy." The first metric is for the ordinal response variable, the second for the count output, the third for the continuous outcome, and the last one for the binary outcome. The fitting process (part d) is exactly the same as the fitting process of the previous multivariate deep learning models with only one type of scale in the output.

The flags given above for training multivariate mixed outcomes (Code_Tuning_With_Flags_MT_Mixed. R) can be used with Appendix 5.

Next is given the grid search used for tuning a multi-trait mixed outcome DNN model:

```
runs.sp<-tuning_run("Code_Tuning_With_Flags_MT_Mixed.R",runs_dir =
'_tuningE1',
          flags=list(dropout1= c(0,0.05),
               units = c(56,97),
               activation1=("relu"),
               batchsize1=c(30),
               Epoch1=c(1000),
               learning_rate=c(0.01),
              val_split=c(0.10)),sample=0.5,confirm =FALSE,echo =F)
```

Again, we used the tuning_run() function to perform the tuning process, which now was done with the full grid search since we specified sample = 1. Here the total number of hyperparameters is four, since hyperparameters % of dropout and number of units have only two values.

The prediction performance of training multi-trait mixed outcome data is given in Table 12.8. The metrics used for evaluating the accuracy were MAAPE for the continuous and count response variables, and the proportion of cases correctly classified (PCCC) for the ordinal and binary traits. The best predictions for the two

Table 12.8 Prediction performance of multivariate mixed outcome data for three hidden layers with genotype × environment interaction and five outer fold cross-validations

Trait	Env	MAAPE	SE_MAAPE	PCCC	SE_PCCC
DTHD	Bed5IR	–	–	0.600	0.073
DTHD	EHT	–	–	0.675	0.064
DTHD	Flat5IR	–	–	0.550	0.050
DTHD	LHT	–	–	0.650	0.073
DTMT	Bed5IR	0.628	0.065	–	–
DTMT	EHT	0.609	0.059	–	–
DTMT	Flat5IR	0.669	0.038	–	–
DTMT	LHT	0.678	0.092	–	–
GY	Bed5IR	0.106	0.010	–	–
GY	EHT	0.124	0.016	–	–
GY	Flat5IR	0.169	0.032	–	–
GY	LHT	0.154	0.005	–	–
Height	Bed5IR	–	–	0.575	0.050
Height	EHT	–	–	0.850	0.073
Height	Flat5IR	–	–	0.725	0.047
Height	LHT	–	–	0.675	0.050

traits evaluated with MAAPE were those of trait GY, while for the ordinal and binary traits we can see that the PCCC was better for the trait Height that includes only two categories (Table 12.8). Again, these predictions belong to the best combination of hyperparameters of the above grid. Practitioners should remember that we only reported the MAAPE for the count and continuous traits, since it made no sense for the ordinal and binary outcomes. Similarly, we only reported the PCCC for the binary and ordinal traits since it made no sense for the count and continuous outcomes.

Appendix 1

R code for training a univariate binary outcome with four hidden layers

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

```
##########Set seed for reproducible results###################
use_session_with_seed(64)

###########Loading the EYT_Toy data set#######################
load("Data_Toy_EYT.RData")

#############Genomic relationship matrix (GRM)#################
Gg=data.matrix(G_Toy_EYT)
G=Gg
dim(G)

############Phenotypic data ###################################
Data_Pheno=Pheno_Toy_EYT
head(Data_Pheno)

summary.BMTMECV <- function(results, information = 'compact', digits =
4, ...) {
 # if (!inherits(object, "BMTMECV")) stop("This function only works for
objects of class 'BMTMECV'")
 results$Observed=as.factor(results$Observed)
 results$Predicted=as.factor(results$Predicted)
 results %>%
  group_by(Env, Partition) %>%
  summarise(PCCC=confusionMatrix(table(Observed,Predicted))$
overall[1],
      PCCC_Lower=confusionMatrix(table(Observed,Predicted))$overall
[3],
      PCCC_Upper=confusionMatrix(table(Observed,Predicted))$overall
[4],
       Kappa=confusionMatrix(table(Observed,Predicted))$overall[2],
      Sensitivity=confusionMatrix(table(Observed,Predicted))$byClas
[1],
       Specificity=confusionMatrix(table(Observed,Predicted))$byClas
[2]) %>%
  select(Env, Partition, PCCC,PCCC_Lower,PCCC_Upper,Kappa,
Sensitivity,Specificity) %>%
  mutate_if(is.numeric, funs(round(., digits))) %>%
  as.data.frame() -> presum

 presum %>% group_by(Env) %>%
  summarise(SE_PCCC= sd(PCCC, na.rm = T)/sqrt(n()), PCCC = mean(PCCC,
na.rm = T),
      SE_PCCC_Lower= sd(PCCC_Lower, na.rm = T)/sqrt(n()), PCCC_Lower =
mean(PCCC_Lower, na.rm = T),
      SE_PCCC_Upper= sd(PCCC_Upper, na.rm = T)/sqrt(n()), PCCC_Upper=
mean(PCCC_Upper, na.rm = T),
      SE_Kappa= sd(Kappa, na.rm = T)/sqrt(n()), Kappa = mean(Kappa, na.
rm = T),
       SE_Sensitivity= sd(Sensitivity, na.rm = T)/sqrt(n()),
Sensitivity = mean(Sensitivity, na.rm = T),
      SE_Specificity= sd(Specificity, na.rm = T)/sqrt(n()), Specificity =
mean(Specificity, na.rm = T)) %>%
  select(Env,PCCC, SE_PCCC,PCCC_Lower, SE_PCCC_Lower,
      PCCC_Upper,SE_PCCC_Upper,Kappa,SE_Kappa, Sensitivity,
SE_Sensitivity,Specificity,SE_Specificity ) %>%
  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_PCCC <- NA
         presum$SE_PCCC_Lower <- NA
         presum$SE_PCCC_Upper <- NA
         presum$SE_Kappa <- NA
         presum$Sensitivity<- NA
         presum$Specificity<- NA
         rbind(presum, finalSum)
        }
 )
 return(out)
}

########Creating the design matrix of lines ##################
Z1G=model.matrix(~0+as.factor(Data_Pheno$GID))
L=t(chol(Gg))
Z1G=Z1G%*%L
ZE=model.matrix(~0+as.factor(Data_Pheno$Env))
Z2GE=model.matrix(~0+Z1G:as.factor(Data_Pheno$Env))
nCV=5  ###Number of outer Cross-validation

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

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

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

########Here are printed the testing observations of each fold#####
CrossV$CrossValidation_list

#######Final X and y=Height to use for training the model#############
y=(Data_Pheno[, 6])
length(y)
y
X=cbind(ZE,Z1G)
dim(X)
Learn_val=c(0.001,0.01,0.1,0.5,1)
Final_results=data.frame()
```

```
for (e in 1:5){
#e=1

 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){
 #  i=1
   Sam_per=sample(1:nrow(X_trn),nrow(X_trn))
   X_trII=X_trn[Sam_per,]
   y_trII=y_trn[Sam_per]

   #####a) Grid search using the tuning_run() function of tfruns
package########
   runs.sp<-tuning_run("Code_Tuning_With_Flags_Bin_4HL.R",runs_dir
= '_tuningE1',
            flags=list(dropout1= c(0,0.05),
                 units = c(67,100),
                 activation1=("relu"),
                 batchsize1=c(28),
                 Epoch1=c(1000),
                 learning_rate=c(Learn_val[e]),
                 val_split=c(0.2)),sample=1,confirm =FALSE,echo =F)
   runs.sp[,2:5]
   ###b) ###### 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_acc=runs.sp$metric_val_acc,flag_dropout1=runs.
sp$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$flag_activation1)
   Hyperpar=rbind(Hyperpar,data.frame(Parameters))
  }
  Hyperpar %>%
   group_by(grid_length) %>%
   summarise(val_acc=mean(metric_val_acc),
        dropout1=mean(flag_dropout1),
        units=mean(flag_units),
        batchsize1=mean(flag_batchsize1),
        learning_rate=mean(flag_learning_rate),
        epochs=mean( epochs_completed)) %>%
   select(grid_length,val_acc,dropout1,units,batchsize1,
learning_rate, epochs) %>%
   mutate_if(is.numeric, funs(round(., 3))) %>%
   as.data.frame() -> Hyperpar_Opt
  ############Optimal hyperparameters############
  Max=max(Hyperpar_Opt$val_acc)
  pos_opt=which(Hyperpar_Opt$val_acc==Max)
  pos_opt=pos_opt[1]
  Optimal_Hyper=Hyperpar_Opt[pos_opt,]
  #####Selectiong the best hyperparameters
  Drop_O=Optimal_Hyper$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, activation ="sigmoid")

  model_Sec %>% compile(
   loss = "binary_crossentropy",
   optimizer = optimizer_adam(lr_O),
   metrics = c('accuracy'))

  ModelFinal <-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))

  ####e) Prediction of testing set ##########################
  predicted=model_Sec %>% predict_classes(X_tst)
  Predicted=predicted
  Observed=y[tst_set]
  results<-rbind(results, data.frame(Position=tst_set,
                    Env=CrossV$Environments[tst_set],
                    Partition=o,
                    Units=Units_O,
                    Epochs=Epoch_O,
                Observed=round(Observed, digits), #$response, digits),
                    Predicted=round(Predicted, digits),
                    Trait=Names_Traits[t]))
  cat("CV=",o,"\n")
 }
 results

 Pred_Summary=summary.BMTMECV(results=results, information =
'compact', digits = 4)
 Pred_Summary

 Final_results=rbind(Final_results,data.frame(learn_val=Learn_val
[e],Pred_Summary))
}
Final_results
write.csv(Final_results,file="Appendix1_Bin_Chapter12_modified.
csv")
```

Appendix 2

R code for training a univariate categorical or ordinal outcome with four hidden layers

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

##########Set seed for reproducible results##################
use_session_with_seed(64)

###########Loading the EYT_Toy data set######################
load("Data_Toy_EYT.RData")

#############Genomic relationship matrix (GRM)################
Gg=data.matrix(G_Toy_EYT)
```

```
G=Gg
dim(G)

###########Phenotypic data ##################################
Data_Pheno=Pheno_Toy_EYT
head(Data_Pheno)

summary.BMTMECV <- function(results, information = 'compact', digits =
4, ...) {
 # if (!inherits(object, "BMTMECV")) stop("This function only works for
objects of class 'BMTMECV'")
 results$Observed=as.factor(results$Observed)
 results$Predicted=as.factor(results$Predicted)
 results %>%
  group_by(Env, Partition) %>%
  summarise(PCCC=as.numeric(confusionMatrix(table(Observed,
Predicted))$overall[1]),
       PCCC_Lower=as.numeric(confusionMatrix(table(Observed,
Predicted))$overall[3]),
       PCCC_Upper=as.numeric(confusionMatrix(table(Observed,
Predicted))$overall[4]),
       Kappa=as.numeric(confusionMatrix(table(Observed,Predicted))$
overall[2]),
       Sensitivity=as.numeric(confusionMatrix(table(Observed,
Predicted))$byClas[1]),
       Specificity=as.numeric(confusionMatrix(table(Observed,
Predicted))$byClas[2])) %>%
  select(Env, Partition, PCCC,PCCC_Lower,PCCC_Upper,Kappa,
Sensitivity,Specificity) %>%
  mutate_if(is.numeric, funs(round(., digits))) %>%
  as.data.frame() -> presum

 presum %>% group_by(Env) %>%
  summarise(SE_PCCC= sd(PCCC, na.rm = T)/sqrt(n()), PCCC = mean(PCCC,
na.rm = T),
      SE_PCCC_Lower= sd(PCCC_Lower, na.rm = T)/sqrt(n()), PCCC_Lower =
mean(PCCC_Lower, na.rm = T),
      SE_PCCC_Upper= sd(PCCC_Upper, na.rm = T)/sqrt(n()), PCCC_Upper=
mean(PCCC_Upper, na.rm = T),
      SE_Kappa= sd(Kappa, na.rm = T)/sqrt(n()), Kappa = mean(Kappa, na.
rm = T),
       SE_Sensitivity= sd(Sensitivity, na.rm = T)/sqrt(n()),
Sensitivity = mean(Sensitivity, na.rm = T),
      SE_Specificity= sd(Specificity, na.rm = T)/sqrt(n()), Specificity =
mean(Specificity, na.rm = T)) %>%
  select(Env,PCCC, SE_PCCC,PCCC_Lower,SE_PCCC_Lower,PCCC_Upper,
SE_PCCC_Upper,Kappa,SE_Kappa, Sensitivity,SE_Sensitivity,
Specificity,SE_Specificity ) %>%
  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_PCCC <- NA
         presum$SE_PCCC_Lower <- NA
         presum$SE_PCCC_Upper <- NA
         presum$SE_Kappa <- NA
         presum$SE_Sensitivity<- NA
         presum$SE_Specificity<- NA
         rbind(presum, finalSum)
        }
 )
 return(out)
}
##############Creating the design matrix of lines ##################
Z1G=model.matrix(~0+as.factor(Data_Pheno$GID))
L=t(chol(Gg))
Z1G=Z1G%*%L
ZE=model.matrix(~0+as.factor(Data_Pheno$Env))
Z2GE=model.matrix(~0+Z1G:as.factor(Data_Pheno$Env))
nCV=5  ###Number of outer Cross-validation

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

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

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

########Here are printed the testing observations of each fold#####
CrossV$CrossValidation_list

#######Final X and y=DTHD to use for training the model################
y=c(Data_Pheno[, 3])-1
nclas=length(unique(y))
length(y)
X=cbind(ZE,Z1G)
dim(X)
Learn_val=c(0.005,0.01,0.015,0.03,0.06)
Final_results=data.frame()
for (e in 1:5){
 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= to_categorical(y[-tst_set],nclas)
  y_tst= to_categorical(y[tst_set],nclas)

  ################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,]

   #####a) Grid search using the tuning_run() function of tfruns
package########
   runs.sp<-tuning_run("Code_Tuning_With_Flags_Ordinal_4HL2.R",
runs_dir = '_tuningEO',
             flags=list(dropout1= c(0,0.05),
                   units = c(67,100),
                   activation1=("relu"),
                   batchsize1=c(28),
                   Epoch1=c(1000),
                   learning_rate=c(Learn_val[e]),
                   val_split=c(0.2)),sample=1,confirm =FALSE,echo =F)
   runs.sp[,2:5]
   ###b) ###### 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_acc=runs.sp$metric_val_acc,flag_dropout1=runs.
sp$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$flag_activation1)
   Hyperpar=rbind(Hyperpar,data.frame(Parameters))
  }
  Hyperpar %>%
   group_by(grid_length) %>%
   summarise(val_acc=mean(metric_val_acc),
        dropout1=mean(flag_dropout1),
        units=mean(flag_units),
        batchsize1=mean(flag_batchsize1),
        learning_rate=mean(flag_learning_rate),
        epochs=mean( epochs_completed)) %>%
   select(grid_length,val_acc,dropout1,units,batchsize1,
learning_rate, epochs) %>%
   mutate_if(is.numeric, funs(round(., 3))) %>%
   as.data.frame() -> Hyperpar_Opt
  ###########Optimal hyperparameters###########
  Max=max(Hyperpar_Opt$val_acc)
  pos_opt=which(Hyperpar_Opt$val_acc==Max)
  pos_opt=pos_opt[1]
  Optimal_Hyper=Hyperpar_Opt[pos_opt,]
  #####Selectiong the best hyperparameters
  Drop_O=Optimal_Hyper$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_batch_normalization() %>%
   layer_dropout(rate =Drop_O) %>%
   layer_dense(units =Units_O, activation =activation_O) %>%
   layer_batch_normalization() %>%
   layer_dropout(rate=Drop_O) %>%
   layer_dense(units =Units_O, activation =activation_O) %>%
   layer_batch_normalization() %>%
   layer_dropout(rate=Drop_O) %>%
   layer_dense(units =Units_O, activation =activation_O) %>%
   layer_batch_normalization() %>%
   layer_dropout(rate=Drop_O) %>%
   layer_dense(units =nclas, activation ="softmax")

  model_Sec %>% compile(
   loss = "categorical_crossentropy",
   optimizer = optimizer_adam(lr_O),
   metrics = c('accuracy'))

  ModelFinal <-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))

  ####e) Prediction of testing set ##########################
  predicted=model_Sec %>% predict_classes(X_tst)
  Predicted=predicted
  Observed=y[tst_set]
  results<-rbind(results, data.frame(Position=tst_set,
                   Env=CrossV$Environments[tst_set],
                   Partition=o,
                   Units=Units_O,
                   Epochs=Epoch_O,
              Observed=round(Observed, digits), #$response, digits),
                  Predicted=round(Predicted, digits),
                  Trait=Names_Traits[t]))
  cat("CV=",o,"\n")
 }
 results

 Pred_Summary=summary.BMTMECV(results=results, information =
'compact', digits = 4)
 Pred_Summary
 Final_results=rbind(Final_results,data.frame(learn_val=Learn_val
[e],Pred_Summary))
}
Final_results
write.csv(Final_results,file="Appendix1_Ord_Chapter12_modifiedV2.
csv")
```

Appendix 3

R code for training a univariate count outcome with four hidden layers and Ridge regularization

```
rm(list=ls(all=TRUE))
library(plyr)
library(tidyr)
library(dplyr)
library(BMTME)
library(tensorflow)
library(keras)
library(tfruns)

#Loading Data_Count_Toy
load('Data_Count_Toy.RData',verbose=TRUE)
ls()

####Phenotipic data
dat_F=Pheno_Toy_Count
head(dat_F)
dim(dat_F)

######Genomic relationship matrix###
G =G_Toy_Count
dim(G)
G_0.5 = cholesky(G, tolerance = 1e-9)

#####Design matrices#####
ZL = model.matrix(~0+as.factor(GID),data=dat_F)
ZL = ZL%*%G_0.5
X_L = model.matrix(~0+Loc,data=dat_F)
```

```
dim(X_L)
X_LM = model.matrix(~0+ZL:Loc,data=dat_F)
X_Block = model.matrix(~0+as.factor(Block):Loc,data=dat_F)
dim(X_Block )

#################
#########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)
}

##Number of fold cross-validation and TRN and TST sets
nCV=5
Data.Final_1=dat_F[,c(1,2,6)]
colnames(Data.Final_1)=c("Line","Env","Response")
Env=unique(Data.Final_1$Env)
nI=length(unique(Data.Final_1$Env))

############Training testing partitions####################
CrossV<-CV.KFold(Data.Final_1, K =nCV, set_seed=123)

X = cbind(X_L,ZL,X_Block,X_LM)
y=dat_F$y
```

```
Learn_val=c(0.005,0.01,0.015,0.03,0.06)
Final_results=data.frame()
digits=4
for (e in 1:5){
#e=5
 # 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){
   #i=1
   Sam_per=sample(1:nrow(X_trn),nrow(X_trn))
   X_trII=X_trn[Sam_per,]
   y_trII=y_trn[Sam_per]

   #####a) Grid search using the tuning_run() function of tfruns
package########
   runs.sp<-tuning_run("Code_Tuning_With_Flags_Count_Lasso.R",
             flags=list(dropout1= c(0),
                   units = c(67,150),
                   activation1=("relu"),
                   batchsize1=c(28),
                   Epoch1=c(1000),
                   learning_rate=c(Learn_val[e]),
                   val_split=c(0.2),
                Lasso_par=c(0.001,0.01)),sample=1,confirm =FALSE,echo
=F)
   runs.sp[,2:5]
   ###b) ###### Ordering in the same way all grids
  runs.sp=runs.sp[order(runs.sp$flag_units,runs.sp$flag_Lasso_par),]

   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_Lasso_par=runs.
sp$flag_Lasso_par, flag_activation1=runs.sp$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),
        Lasso_par=mean(flag_Lasso_par),
        epochs=mean( epochs_completed)) %>%
   select(grid_length,val_mse,dropout1,units,batchsize1,
learning_rate, Lasso_par,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
  Lasso_par_O=Optimal_Hyper$Lasso_par

  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, kernel_regularizer=regularizer_l1
(Lasso_par_O),activation =activation_O, input_shape = c(dim
(X_trII)[2])) %>%
   layer_dropout(rate=Drop_O) %>%
   layer_dense(units =Units_O, kernel_regularizer=regularizer_l1
(Lasso_par_O),activation =activation_O) %>%
   layer_dropout(rate=Drop_O) %>%
   layer_dense(units =Units_O, kernel_regularizer=regularizer_l1
(Lasso_par_O),activation =activation_O) %>%
   layer_dropout(rate=Drop_O) %>%
   layer_dense(units =Units_O, kernel_regularizer=regularizer_l1
(Lasso_par_O),activation =activation_O) %>%
   layer_dropout(rate=Drop_O) %>%
   layer_dense(units =1, activation ="exponential")

  model_Sec %>% compile(
   loss = "poisson",
   optimizer = optimizer_adam(lr_O),
   metrics = c('mse'))
  ModelFinal <-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))

  ####e) Prediction of testing set ##########################
  predicted=model_Sec %>% predict(X_tst)
  Predicted=predicted
  Observed=y[tst_set]
  results<-rbind(results, data.frame(Position=tst_set,
                     Environment=CrossV$Environments[tst_set],
                     Partition=o,
                     Units=Units_O,
                     Epochs=Epoch_O,
                Observed=round(Observed, digits), #$response, digits),
                     Predicted=round(Predicted, digits),
                     Trait="Count"))
  cat("CV=",o,"\n")
 }
 results

 Pred_Summary=summary.BMTMECV(results=results, information =
'compact', digits = 4)
 Pred_Summary

 Final_results=rbind(Final_results,data.frame(learn_val=Learn_val
[e],Pred_Summary))
}
Final_results
write.csv(Final_results,file="Appendix3_Count_Chapter12_modifiedv2.
csv")
```

Appendix 4

R code for training a multi-trait normal outcome with three hidden layers

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

##########Set seed for reproducible results###################
use_session_with_seed(64)
```

```
################Loading the MaizeToy Data set###############
data("MaizeToy")
ls()
head(phenoMaizeToy)

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

################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#######################
Data_Pheno=phenoMaizeToy
Y <- as.matrix(Data_Pheno[, -c(1, 2)])

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

#CrossV <- CV.KFold(pheno, DataSetID = 'GID', K = 5, set_seed = 123)
CrossV <- CV.RandomPart(pheno, NPartitions = 5, PTesting = 0.2,
set_seed = 123)

summary.BMTMECV <- function(results, information='compact',
digits=4, ...) {
# if (!inherits(object, "BMTMECV")) stop("This function only works for
objects of class 'BMTMECV'")
  results %>%
  group_by(Environment, Trait, Partition) %>%
  summarise(Pearson = cor(Predicted, Observed, use = 'pairwise.
complete.obs'),
  MAAPE = mean(atan(abs(Observed-Predicted)/abs(Observed))),
  MSE= mean((Observed-Predicted)^2)) %>%
  select(Environment, Trait, Partition, Pearson, MAAPE,MSE) %>%
  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_Pearson = sd(Pearson, na.rm = T)/sqrt(n()), Pearson = mean
(Pearson, na.rm = T),
  SE_MSE = sd(MSE, na.rm = T)/sqrt(n()), MSE = mean(MSE, na.rm = T)) %>%
  select(Environment, Trait, Pearson, SE_Pearson, MAAPE, SE_MAAPE,
MSE, SE_MSE ) %>%
  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_Pearson <- NA
         presum$SE_MAAPE <- NA
         presum$SE_MSE <- NA
         rbind(presum, finalSum)
        }
 )
 return(out)
}

#######Final X and y=Height to use for training the model#############
tst_set=CrossV$CrossValidation_list[[1]]

#####X training and testing####
X=cbind(Z.E,Z.G)
dim(X)

y=(Data_Pheno[, 3:5])
y2=y

####Weights calculation
max_Dif_q1=max(quantile(y[,1],p=0.75)-quantile(y[,1],p=0.5),
quantile(y[,1],p=0.5)-quantile(y[,1],p=0.25))
max_Dif_q2=(max(quantile(y[,2],p=0.75)-quantile(y[,2],p=0.5),
quantile(y[,2],p=0.5)-quantile(y[,2],p=0.25)))
max_Dif_q3=(max(quantile(y[,3],p=0.75)-quantile(y[,3],p=0.5),
quantile(y[,3],p=0.5)-quantile(y[,3],p=0.25)))
w1=max_Dif_q1
w2=max_Dif_q1/max_Dif_q2
w3=max_Dif_q1/max_Dif_q3

###################Outer cross-validation######################
digits=4
n=dim(X)[1]
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############################
  X_trII=X_trn
  y_trII=y_trn

  #####a) Grid search using the tuning_run() function of tfruns
package########
  runs.sp<-tuning_run("Code_Tuning_With_Flags_MT_normal.R",
runs_dir = '_tuningE1',
           flags=list(dropout1= c(0,0.05),
                units = c(56,97),
                activation1=("relu"),
                batchsize1=c(22),
                Epoch1=c(1000),
                learning_rate=c(0.001),
               val_split=c(0.25)),sample=0.5,confirm =FALSE,echo =F)
  runs.sp[,23:27]

  ###b) Decreasing order of prediction performance of each combination
of the grid
  runs=runs.sp[order(runs.sp$metric_val_loss , decreasing = T), ]
  runs
  runs[,23:27]
  dim(runs)[1]

  ####c) Selecting the best combination of hyperparameters ####
  pos_opt=dim(runs)[1]
  opt_runs=runs[pos_opt,]

  ####d) Renaming the optimal hyperparametes
  Drop_O=opt_runs$flag_dropout1
  Epoch_O=opt_runs$epochs_completed
  Units_O=opt_runs$flag_units
  activation_O=opt_runs$flag_activation1
  batchsize_O=opt_runs$flag_batchsize1
  lr_O=opt_runs$flag_learning_rate

 ###########Refitting the model with the optimal values#############
 ### add covariates
 input <- layer_input(shape=dim(X_trn)[2],name="covars")

 ### add hidden layers
 base_model <- input %>%
  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)

 # add output 1
 yhat1 <- base_model %>%
  layer_dense(units =1, name="response_1")

 # add output 2
 yhat2 <- base_model %>%
  layer_dense(units = 1, name="response_2")

 # add output 3
 yhat3 <- base_model %>%
  layer_dense(units = 1, name="response_3")

 # build multi-output model
 model <- keras_model(input,list(response_1=yhat1,response_2=yhat2,
response_3=yhat3)) %>%
  compile(optimizer =optimizer_adam(lr=lr_O),
      loss=list(response_1="mse",response_2="mse",
response_3="mse"),
      metrics=list(response_1="mse",response_2="mse",
response_3="mse"),
     loss_weights=list(response_1=w1,response_2=w2,response_3=w3))

 print_dot_callback <- callback_lambda(
  on_epoch_end = function(epoch, logs) {
   if (epoch %% 20 == 0) cat("\n")
   cat(".")
  })
 early_stop <- callback_early_stopping(monitor = c("val_loss"),
mode='min', patience =50)

 # fitting the model
 model_fit <- model %>%
  fit(x=X_trn,
    y=list(response_1=y_trn[,1],response_2=y_trn[,2],
response_3=y_trn[,3]),
    epochs=Epoch_O,
    batch_size =batchsize_O,
    verbose=0, callbacks = list(print_dot_callback))

 # predict values for test set
 Yhat<-predict(model,X_tst)%>%
  data.frame()%>%
  setNames(colnames(y_trn))
 YP=Yhat
 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(YP, digits)))
 cat("CV=",o,"\n")
}
results
nt=3

Pred_all_traits=data.frame()
for (i in 1:nt){
 #i=1
 pos_i_obs=5+i
 pos_i_pred=8+i
 results_i=results[,c(1:5,pos_i_obs,pos_i_pred)]
 results_i$Trait=Names_Traits[i]
 Names_results_i=colnames(results_i)
 colnames(results_i)=c(Names_results_i
[1:5],"Observed","Predicted","Trait")

 Pred_Summary=summary.BMTMECV(results=results_i, information =
'compact', digits = 4)
 Pred_Summary
 Pred_all_traits=rbind(Pred_all_traits,Pred_Summary)
}
Pred_all_traits
write.csv(Pred_all_traits,file="Multi-trait-predictions_Normal2.
csv")
```

Appendix 5

R code for training a multi-trait mixed outcome with three hidden layers

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

```
##########Set seed for reproducible results###################
use_session_with_seed(64)

###########Loading the EYT_Toy data set#######################
load("Data_Toy_EYT.RData")

#############Genomic relationship matrix (GRM)################
Gg=data.matrix(G_Toy_EYT)
G=Gg

############Phenotypic data ##################################
Data_Pheno=Pheno_Toy_EYT

########Creating the design matrix of lines ##################
Z1G=model.matrix(~0+as.factor(Data_Pheno$GID))
L=t(chol(Gg))
Z1G=Z1G%*%L
ZE=model.matrix(~0+as.factor(Data_Pheno$Env))
Z2GE=model.matrix(~0+Z1G:as.factor(Data_Pheno$Env))
nCV=5

summary.BMTMECV <- function(results, information='compact',
digits=4, ...) {
 # if (!inherits(object, "BMTMECV")) stop("This function only works for
objects of class 'BMTMECV'")

 results %>%
  group_by(Environment, Trait, Partition) %>%
  summarise(Pearson = cor(Predicted, Observed, use = 'pairwise.
complete.obs'),
       MAAPE = mean(atan(abs(Observed-Predicted)/abs(Observed))),
       PCCC = 1-sum(Observed!=Predicted)/length(Observed)) %>%
  select(Environment, Trait, Partition, Pearson, MAAPE,PCCC) %>%
  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_Pearson = sd(Pearson, na.rm = T)/sqrt(n()), Pearson = mean
(Pearson, na.rm = T),
      SE_PCCC = sd(PCCC, na.rm = T)/sqrt(n()), PCCC = mean(PCCC, na.rm =
T)) %>%
  select(Environment, Trait, Pearson, SE_Pearson, MAAPE, SE_MAAPE,
PCCC, SE_PCCC ) %>%
  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_Pearson <- NA
         presum$SE_MAAPE <- NA
         presum$SE_PCCC <- NA
         rbind(presum, finalSum)
        }
 )
 return(out)
}

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

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

#CrossV <- CV.KFold(pheno, DataSetID = 'GID', K = 5, set_seed = 123)
CrossV <- CV.RandomPart(pheno, NPartitions =5, PTesting = 0.2, set_seed
= 123)

#####X training and testing####
X=cbind(ZE,Z1G,Z2GE)
dim(X)

y=Data_Pheno[, c(3,4,5,6)]
summary(y[,3])
y[,1]=y[,1]-1
y[,2]=y[,2]-1

############Outer cross-validation#######################
digits=4
n=dim(X)[1]
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,])

 y_trn[,1]=to_categorical(y_trn[,1], 3)
 y_tst[,1]=to_categorical(y_tst[,1], 3)

 ##################Inner cross-validation#########################
 X_trII=X_trn
 y_trII=y_trn
 #####a) Grid search using the tuning_run() function of tfruns
package########
 runs.sp<-tuning_run("Code_Tuning_With_Flags_MT_Mixed.R",runs_dir
= '_tuningE1',
           flags=list(dropout1= c(0,0.05),
                 units = c(56,97),
                 activation1=("relu"),
                 batchsize1=c(30),
                 Epoch1=c(1000),
                 learning_rate=c(0.01),
                 val_split=c(0.10)),sample=1,confirm =FALSE,echo =F)
 runs.sp[,23:27]

 ###b) Decreasing order of prediction performance of each combination of
the grid
 runs=runs.sp[order(runs.sp$metric_val_loss , decreasing = T), ]

 ####c) Selecting the best combination of hyperparameters ####
 pos_opt=dim(runs)[1]
 opt_runs=runs[pos_opt,]

 ####d) Renaming the optimal hyperparametes
 Drop_O=opt_runs$flag_dropout1
 Epoch_O=opt_runs$epochs_completed
 Units_O=opt_runs$flag_units
 activation_O=opt_runs$flag_activation1
 batchsize_O=opt_runs$flag_batchsize1
 lr_O=opt_runs$flag_learning_rate

 ###########Refitting the model with the optimal values###############
 ### add covariates
 input <- layer_input(shape=dim(X_trn)[2],name="covars")

 ### add hidden layers
 base_model <- input %>%
  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)

 # add output 1
 yhat1 <- base_model %>%
  layer_dense(units =3,activation="softmax", name="response_1")

 # add output 2
 yhat2 <- base_model %>%
  layer_dense(units = 1, activation="exponential",name="response_2")
 # add output 3
 yhat3 <- base_model %>%
  layer_dense(units = 1, name="response_3")

 # add output 4
 yhat4 <- base_model %>%
  layer_dense(units = 1, activation="sigmoid", name="response_4")

 # build multi-output model
 model <- keras_model(input,list(response_1=yhat1,response_2=yhat2,
response_3=yhat3,response_4=yhat4)) %>%
  compile(optimizer =optimizer_adam(lr=lr_O),
      loss=list(response_1="categorical_crossentropy",
response_2="mse",response_3="mse",
response_4="binary_crossentropy"),
      metrics=list(response_1="accuracy",response_2="mse",
response_3="mse",response_4="accuracy"),
      loss_weights=c(response_1=1,response_2=1,response_3=1,
response_4=1))

 print_dot_callback <- callback_lambda(
  on_epoch_end = function(epoch, logs) {
   if (epoch %% 20 == 0) cat("\n")
   cat(".")
  })
 early_stop <- callback_early_stopping(monitor = c("val_loss"),
mode='min', patience =50)
 #,early_stop

 # fitting the model
 model_fit <- model %>%
  fit(x=X_trn,
    y=list(response_1=y_trn[,1],response_2=y_trn[,2],
response_3=y_trn[,3],response_4=y_trn[,4]),
    epochs=Epoch_O,
    batch_size =batchsize_O,
    verbose=0, callbacks = list(print_dot_callback))

 # predict values for test set
 Yhat<-predict(model,X_tst)%>%
  data.frame()%>%
  setNames(colnames(y_trn))
 YP=matrix(NA,ncol=ncol(y),nrow=nrow(Yhat))
 head(Yhat)
 P_T1=(apply(data.matrix(Yhat[,1:3]),1,which.max)-1)
 P_T4=ifelse(c(Yhat[,6])>0.5,1,0)
 YP[,1]=P_T1
 YP[,2]=c(Yhat[,4])
 YP[,3]=c(Yhat[,5])
 YP[,4]=P_T4
 head(YP)
 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(YP, digits)))
 cat("CV=",o,"\n")
}
results
nt=4

Pred_all_traits=data.frame()
for (i in 1:nt){
 #i=1
 Names_Traits
 pos_i_obs=5+i
 pos_i_pred=9+i
 results_i=results[,c(1:5,pos_i_obs,pos_i_pred)]
 results_i$Trait=Names_Traits[i]
 Names_results_i=colnames(results_i)
 colnames(results_i)=c(Names_results_i
[1:5],"Observed","Predicted","Trait")

 Pred_Summary=summary.BMTMECV(results=results_i, information =
'compact', digits = 4)
 Pred_Summary
 Pred_all_traits=rbind(Pred_all_traits,data.frame
(Trait=Names_Traits[i],Pred_Summary))
}
Pred_all_traits
Res_Sum=Pred_all_traits[,-c(3:5)]
Res_Sum
write.csv(Res_Sum,file="Multi-trait-predictions_Mixed5.csv")
```

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

Calus MP, Veerkamp RF (2011) Accuracy of multi-trait genomic selection using different methods. Genetics Selection Evolution 43(1):26. https://doi.org/10.1186/1297-9686-43-26

Castro AFNM, Castro RV, Oliveira CAO, Lima JE, Santos RC, Pereira BLC, Alves ICN (2013) Multivariate analysis for the selection of eucalyptus clones destined for charcoal production. Pesq Agrop Brasileira 48(6):627–635

Chollet F, Allaire JJ (2017) Deep learning with R. Manning Publications, Manning Early Access Program (MEA), 1st edn

He D, Kuhn D, Parida L (2016) Novel applications of multitask learning and multiple output regression to multiple genetic trait prediction. Bioinformatics 32(12):i37–i43. https://doi.org/10.1093/bioinformatics/btw249

Huang M, Chen L, Chen Z (2015) Diallel analysis of combining ability and heterosis for yield and yield components in rice by using positive loci. Euphytica 205(1):37–50

Ioffe S, Szegedy C (2015) Batch normalization: accelerating deep network training by reducing internal covariate shift. arXiv Preprint arXiv:1502.03167

Jia Y, Jannink J-L (2012) Multiple-trait genomic selection methods increase genetic value prediction accuracy. Genetics 192(4):1513–1522. https://doi.org/10.1534/genetics.112.144246

Jiang J, Zhang Q, Ma L, Li J, Wang Z, Liu JF (2015) Joint prediction of multiple quantitative traits using a Bayesian multivariate antedependence model. Heredity 115(1):29–36

LeCun Y, Bottou L, Orr G, Muller K (1998) Efficient backprop. In: Orr G, Muller K (eds) Neural networks: tricks of the trade. Springer

Montesinos-López OA, Montesinos-López A, Crossa J, Toledo F, Pérez-Hernández O, Eskridge KM, Rutkoski J (2016) A genomic Bayesian multi-trait and multi-environment model. G3: Genes, Genomes, Genetics 6(9):2725–2744

Montesinos-López A, Montesinos-López OA, Gianola D, Crossa J, Hernández-Suárez CM (2018a) Multivariate Bayesian analysis of on-farm trials with multiple-trait and multiple-environment data. Agron J 111(6):2658–2669. https://doi.org/10.2134/agronj2018.06.0362

Montesinos-López OA, Montesinos-López A, Gianola D, Crossa J, Hernández-Suárez CM (2018b) Multi-trait, multi-environment deep learning modeling for genomic-enabled prediction of plant. G3: Genes, Genomes, Genetics 8(12):3829–3840

Montesinos-López A, Montesinos-López OA, Gianola D, Crossa J, Hernández-Suárez CM (2018c) Multi-environment genomic prediction of plant traits using deep learners with a dense architecture. G3: Genes, Genomes, Genetics 8(12):3813–3828. https://doi.org/10.1534/g3.118.200740

Montesinos-López OA, Martín-Vallejo J, Crossa J, Gianola D, Hernández-Suárez CM, Montesinos-López A, Juliana P, Singh R (2019) New deep learning genomic prediction model for multi-traits with mixed binary, ordinal, and continuous phenotypes. G3: Genes, Genomes, Genetics 9(5):1545–1556

R Core Team (2019) R: a language and environment for statistical computing. R Foundation for Statistical Computing, Vienna. ISBN 3-900051-07-0. http://www.R-project.org/

Schulthess AW, Zhao Y, Longin CFH, Reif JC (2017) Advantages and limitations of multiple-trait genomic prediction for Fusarium head blight severity in hybrid wheat (*Triticum aestivum* L.). Theor Appl Genet 131(3):685–701. https://doi.org/10.1007/s00122-017-3029-7

Wiesler S, Ney H (2011) A convergence analysis of log-linear training. In: Shawe-Taylor J, Zemel RS, Bartlett P, Pereira FCN, Weinberger KQ (eds), Advances in neural information processing systems, vol 24. Granada, pp 657–665