Using ann_tab_cv() and transfer learning

Walter K. Kremers, Mayo Clinic, Rochester MN

22 March 2024

The functions

The ann_cv_lin() function fits a neural network (NN) model to tabular data. The network has a simple "sequential" structure with linear components and by default ReLU activations. It has two hidden layers where the number of terms in the two layers can be specified by the user. Data for input are in a format as with the nested.glmnetr() function. Models can be fit as generalizations to the Cox, logistic or linear regression models.

We originally wrote the ann_cv_lin() function to better understand how a NN model which begins its "numerical optimization" near a model informed by a linear model might perform better than a standardly fit NN model. The program is not optimized to find the "best algorithm" for model fitting so other NN programs may perform better. Still, it does show how this "transfer learning" from a linear model to the NN can dramatically improve model fit. Here we use the relaxed lasso regression model (tuned on lambda and gamma) as the linear model for the transfer learning. In that both the relaxed lasso and NN models are fit in nested.glmneter() we use this function to fit these transfer learning NN models. As nested cross validations can have long run times, as an option, one may fit these models without actually performing the nested part.

The nested.glmnetr() function also fits lasso, XGBoost and p-value tuned stepwise regression models while using one level of cross validation to inform hyper parameters and another level of cross validation to evaluate model performance, that is it does nested cross validation. Measures of model performance include, as often done, deviance and agreement (concordance or R-square). We also provide linear calibration coefficients, the values obtained by regressing the original outcome variable on the predicteds from the different models, including the NNs. In this formulation the predicteds are taken before any "final" activation so they are analogous to the X * Beta term from a Cox, logistic or linear model. A regression coefficient from this refit greater than 1 suggests a bias in the model under fitting and a coefficient less than 1 suggests a bias in over fitting.

Data requirements

The data elements for input to the ann_tab_cv() function are basically the same as for the other *glmnetr* functions like nested.glmenter() and cv.glmneter(). Input data should comprise of 1) a (numerical) matrix of predictors and 2) an outcome variable or variables in (numerical) vector form. NULL and Na's are not allowed in input matrices or vectors. For the estimation of the "fully" relaxed parts (where gamma=0) of the relaxed lasso models the package is set up to fit the "gaussian" and "binomial" models using the *stats* glm() function and Cox survival models using the the coxph() function of the *survival* pacakge. When fitting the Cox model or an extension like with NNs the "outcome" variable is interpreted as the "time" variable in the Cox model, and one must also specify 3) an indicator variable for event, again in vector form. Start times are not at this time accounted for in ann_tab_cv(). Row i of the predictor matrix and element i of the outcome vector(s) are to include the data for the same sampling unit.

For fitting the NNs we use the R torch package. We refer to the package reference manual and the book https://skeydan.github.io/Deep-Learning-and-Scientific-Computing-with-R-torch/ for general information on this package. To run the NN models one needs to install the R torch package. When first using the package one may also be prompted to allow torch to download some tensor libraries. The NN torch library generally requires input data to be in torch tensor format, and provides output in torch tensor format as well. Here we convert data from the usual R format to the torch format so the user does not have to. In some functions, too, we convert outputs from torch format back to standard R format for the user.

An example dataset

To demonstrate usage of ann_tab_cv() we first generate a data set for analysis. The code

```
# Simulate data for use in an example relaxed lasso fit of survival data
# first, optionally, assign seeds for random number generation to get replicable results
set.seed(116291949)
torch_manual_seed(77421177)
simdata=glmnetr.simdata(nrows=1000, ncols=100, beta=NULL, intr=c(1,0,1,0))
```

generates simulated data with interactions (specified by the intr=c(1,0,1,0) term) for analysis. We extract data in the format required for input to the ann_tab_cv() program.

```
# Extract simulated survival data
xs = simdata$xs         # matrix of predictors
yt = simdata$yt         # vector of survival times
event = simdata$event   # indicator of event vs. censoring
y_ = simdata$y_         # vector of quantitative values
yb = simdata$yb         # vector of  0 and 1 values
```

Inspecting the predictor matrix we see

```
# Check the sample size and number of predictors
print(dim(xs))
```

```
## [1] 1000  100
```

```
# Check the rank of the design matrix, i.e. the degrees of freedom in the predictors
rankMatrix(xs)[[1]]
```

```
## [1] 94
```

```
# Inspect the first few rows and some select columns
print(xs[1:10,c(1:12,18:20)])
```

```
##       X1 X2 X3 X4 X5 X6 X7 X8 X9 X10 X11 X12        X18        X19         X20
##  [1,]  1  1  0  0  0  0  0  0  0   1   0   1  0.1513225 -0.4034383  0.35250844
##  [2,]  1  0  0  0  1  0  0  1  0   0   0   0 -1.1610480  0.5533030  0.14578868
##  [3,]  1  0  0  1  0  0  1  0  0   0   0   0 -0.3292269  0.3086399 -0.48443836
##  [4,]  1  1  0  0  0  0  0  0  0   1   0   0  2.0635214 -0.5500741 -0.02173104
##  [5,]  1  0  0  0  1  0  0  1  0   0   0   0  0.3905722 -0.6836452 -0.37643201
##  [6,]  1  0  1  0  0  0  0  0  1   0   0   0 -0.2397597  1.6909447  0.49599945
```

```
##  [7,]  1  0  1  0  0  0  0  1  0   0   0   0 -0.5592424  0.2314638 -0.53198341
##  [8,]  1  0  0  1  0  0  0  0  0   0   1   0 -1.0050514  0.5319574  0.54287646
##  [9,]  1  0  0  1  0  0  0  0  0   0   1   0  1.2548034  0.8213164  0.17067691
## [10,]  1  0  0  0  1  0  0  0  1   0   0   0 -0.3079151 -0.6105910 -0.88711869
```

Fitting a basic neural network model to "tabular" data

To fit a NN model we can use a simple function call as in

```
## fit a model with some monitoring to the R console
set.seed( 67213041 )
torch_manual_seed( 7221250 )
ann_tab_cox_ex1 = ann_tab_cv(myxs=xs, myy=yt, myevent=event, family="cox",
                             fold_n=10, eppr=-2)
```

```
##  Epoch:   0 Full data Loss:  6.205775 Train Concordance: 0.5323026
##   imax= 40   minloss= 1   which_loss= 40   which_agree= 51   gotoend= 0   l1= 0
##  Epoch: 40 Train Loss:  5.267484 Train Concordance: 0.8725083
```

We see there is little the user needs to specify, which is basically the data one includes when fitting a regular Cox model. The one piece of input which is unusual is the eppr=-2. The eppr term has no influence on the model but instructs the program to send some fit information to the R console for monitoring the model fitting process. Here we see the model loss and concordance calculated using the training data. From the output generated during model fit we see that for the "starting point" of the model fit the loss function is about 6.2 and concordance 0.5. The concordance of about 0.5 is what we expect by chance alone, consistent with the fact that when we begin the model fit we are taking a model chosen at random. After the gradient descent goes through 0 iterations, as suggested by cross validation, the loss based upon the training data goes down to about 3.55 and the concordance increases to about 0.787, a marked increase. This minimal amount of information sent to the R console is achieved by setting eppr=-2. To avoid any information being sent to the console one may set eppr = -3, or any number less than -2. Numbers larger than -2 will provide more updates during the model fitting process.

To see more information on the model fit we submit

```
## simple model summary
ann_tab_cox_ex1$modelsum
```

```
##        n folds         epochs      length Z1      length Z2           actv
##     10.0000000    200.0000000     16.0000000      8.0000000      1.0000000
##          drpot           mylr             wd             l1          lasso
##      0.0000000      0.0050000      0.0000000      0.0000000      0.0000000
##         lscale          scale     which loss    which agree        CV loss
##      5.0000000      1.0000000     40.0000000     51.0000000      3.5544941
##       CV Agree    CV accuracy     naive loss    naive agree naive accuracy
##      0.7871981      0.0000000      5.2674842      0.8725083      0.0000000
##     agree i_=0
##      0.5323026
```

Here we see that the cross validation is based upon a 10 fold split of the data, and the gradient descent algorithm goes through 200 iterations or epochs for each fold of the data. For this simple model fit we are tuning on number of iterations in the gradient descent fitting. This is not necessarily the most robust way to fit a model but 1) this is meant as simpler example of how to fit NN models and 2) the original purpose of

the ann_tab_cv() function was to see how we can improve NN fits by using a transfer learning form a linear model to the NN (and discuss below). Next we see the first hidden layer is a vector of length 16, and the second hidden layer a vector of length 8. The actv of 1 specifies a ReLU activation function was used, drpot of 0 that no drop out was used, mylr of 0.005 indicates the learning rate used in the optimization. wd and l1 indicate the L2 and L1 penalties used when model fitting, corresponding to ridge and lasso regression. Both are 0 here indicating these penalties were not employed. The "which loss" and "which agree" values indicate the number of epochs when loss is minimized and agreement (concordance or R-square) is maximized in the cross validation and inform the number of epochs to be used in the final model fit using the whole data set. The CV loss and CV concordance based upon these numbers of fits were about 1 and 40, while the "naive" loss and concordances calculated using training data were much greater at about 3.55 and 0.787. This is as we expect for concordance as the values based upon the training data are associated with a lot of over fitting due to the number of free parameters in the NN model. The naive deviance of 3.55 being larger than the cv deviance of 1 may derive from the larger sample size when using the whole data set for calculations. CV accuracy, the fraction of "correctly classified" is not calculated here and 0 is displayed. The concordance for the random starting model was about 0.5, as sent to the R console in the example above.

We can get more information on the NN model, for example with the command

```
## This shows the tensor (matrix) structure used in the model
str(ann_tab_cox_ex1$model$parameters)
```

```
## List of 6
##  $ 0.weight:Float [1:16, 1:100]
##  $ 0.bias  :Float [1:16]
##  $ 3.weight:Float [1:8, 1:16]
##  $ 3.bias  :Float [1:8]
##  $ 6.weight:Float [1:1, 1:8]
##  $ 6.bias  :Float [1:1]
```

The first item here, 0.wieght, describes the dimensions of the tensor (matrix) used in transforming the original data to the fist hidden layer, here 100 and 16. While we would usually expect this tensor to be 100 rows tall and 16 columns wide, tensors are often transposed to take greater advantage of machine architecture to speed calculations. Next we see 0.bias, a tensor with 1 dimension and thus essentially a vector, which contains the intercept terms when transforming from the input data to the first hidden layer. The 3.weight and 3.bias terms determine the transformation from the first hidden to the second hidden layer. Here they transform a vector with 16 terms to a vector or 8 terms. Finally the 6.weight and 6.bias terms determine the transformation from the second hidden layer to the model output of 1 dimension. To get more model information we can use the command

```
## This shows the general structure of the neural network model
ann_tab_cox_ex1$model$modules[[1]]
```

```
## An `nn_module` containing 1,761 parameters.
##
## -- Modules ---------------------------------------------------------------------
## * 0: <nn_module> #1,616 parameters
## * 1: <nn_relu> #0 parameters
## * 2: <nn_dropout> #0 parameters
## * 3: <nn_module> #136 parameters
## * 4: <nn_relu> #0 parameters
## * 5: <nn_dropout> #0 parameters
## * 6: <nn_module> #9 parameters
## * 7: <nn_identity> #0 parameters
```

This shows the model structure. The first term nn_module, involves a linear transformation from the inputs to a hidden layer. This module is based upon the "nn_linear" tensor module which we modified to allow us to more easily set or update model weights and biases. This is followed by a ReLU activation and then a "dropout" with probability 0. The activation function is a nonlinear transformation that allows the NN to fit more general response surfaces than linear models. Without this the NN would simply involve matrix multiplication which would result in another matrix, and reduce to a linear model. The dropout can be used to randomly set terms to 0, i.e. to drop them out, with a specified probability. This can sometimes improve model fit. Here though we set this probability to 0 so this is not impacting our model fit. The final element "nn_identity" unsurprisingly means to apply the identity function. This could be set to "sigmoid", i.e. (1/(1+exp(-x)), to transform the input values to the interval (0,1), as can be done when fitting generalizations of the logistic regression model.

See here how we have been inspecting elements of lists of lists. The basic flow of model fitting using R torch is to first define a model object and then use numerical routines to update the contents of this model object using numerical algorithms. Here we combine this model object with other information in a list to organize in one place the model, information about the model fit and how the model was derived.

Though we can, we do not typically inspect individual parameters (weights and biases) in a NN. The value of NN models lies instead in their ability to predict. In general predicted values can be gotten in R torch by a command like model(newdata). We can get predicteds form an ann_tab_cv() output object, for example, with a command like

```
# ann_tab_cv() predicteds in torch tensor format
preds = ann_tab_cox_ex1$model(xs)
preds[1:8]
```

```
## torch_tensor
## -1.3531
## -2.5461
##  2.0564
## -0.2299
## -0.2578
##  0.8483
## -1.9271
## -1.3005
## [ CPUFloatType{8,1} ][ grad_fn = <SliceBackward0> ]
```

Since R torch models work with tensors the model predicteds, too, are put in a torch tensor format. We can easily change the output to a usual R numerical format by wrapping the output in the as.numeric() function, for example, as in

```
# ann_tab_cv() predicteds in standard R numerical format
preds = as.numeric( ann_tab_cox_ex1$model(xs) )
preds[1:8]
```

```
## [1] -1.3530899 -2.5460956  2.0564485 -0.2298792 -0.2578476  0.8482994 -1.9271022
## [8] -1.3005327
```

Working with torch tensors

When managing tensors torch does not always make actual copies of newly crated tensors. Instead it will often store a pointer to wahre a tensor may be found and opdrations that need to be performed to obtain

the tensor identified by its name. A difficulty with this is when one leaves the R session teh actual tensors may be lost. To be able to store and retreive the NN models we store the tensors used in the models as R vectors and matrices. We can then save object with code like

```
save( ann_tab_cox_ex1, file="~data/ann_tab_cox_ex1.RLIST")
```

and read a stored output object with code like

```
load( file="~data/ann_tab_cox_ex1.RLIST" )
```

Because the actual tensors can be lost when doing this the functions above taking torch objects including tensors may not work. We have written a predict function, predict_ann_tab(), which allows the user to get predicteds from a load()ed model. This is done in an example below.

Performing calibrations with the neural network models

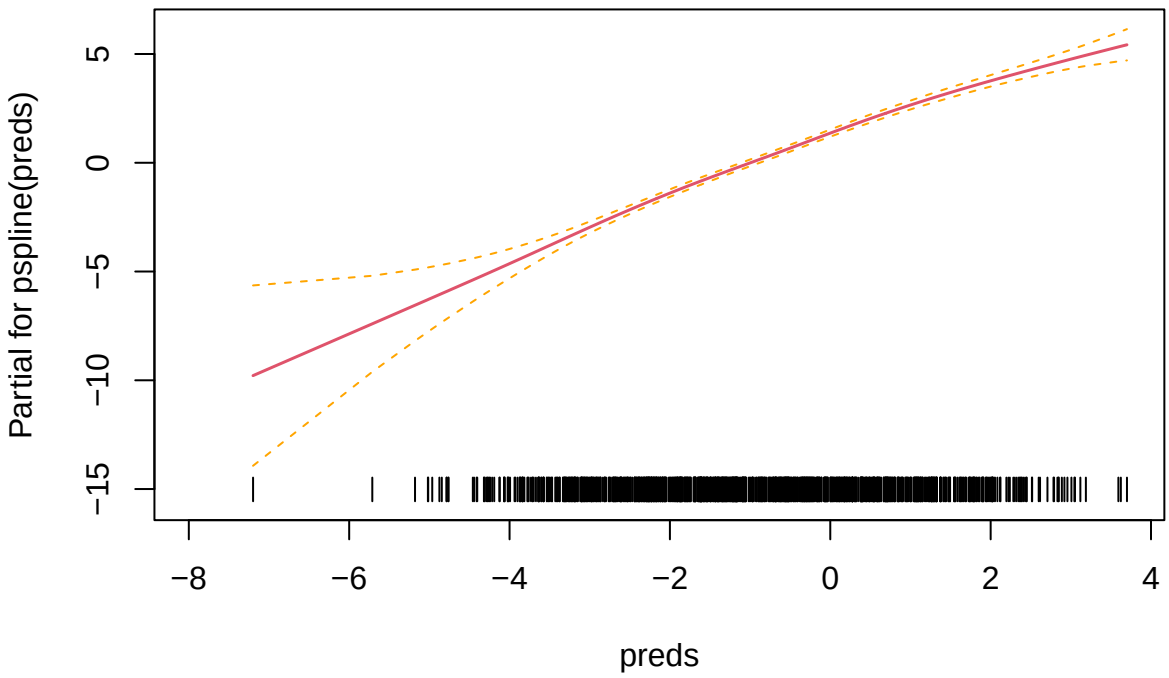
We can then use the predicteds in this R numerical format for evaluation using other R functions and packages. For example for calibration of the NN model one could begin with the code

```
# ann_tab_cv() linear calibration
cox_fit1 = coxph(Surv(yt, event) ~ preds)
summary(cox_fit1)
```

```
## Call:
## coxph(formula = Surv(yt, event) ~ preds)
##
##   n= 1000, number of events= 699
##
##          coef exp(coef) se(coef)     z Pr(>|z|)
## preds 1.30393   3.68375  0.03954 32.98   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
##       exp(coef) exp(-coef) lower .95 upper .95
## preds     3.684     0.2715     3.409     3.981
##
## Concordance= 0.876  (se = 0.006 )
## Likelihood ratio test= 1427  on 1 df,   p=<2e-16
## Wald test            = 1087  on 1 df,   p=<2e-16
## Score (logrank) test = 1216  on 1 df,   p=<2e-16
```

Here the coefficient of about 1.3 being greater than 1 suggests the NN model may be underestimating the hazard ratios between sample elements. To further evaluate the need for calibration we can fit a spline on the NN predicteds as in

```
# Fit a spline to preds using coxph, and plot
cox_fit2 = coxph(Surv(yt, event) ~ pspline(preds))
termplot(cox_fit2,term=1,se=T,rug=T)
```



The spline fit depicted in this graph does not appear to be consistent with a strignt line suggesting the NN is not, in its current form, well calibrated.

So, why the ann in ann_tab_cv? Many of the torch functions begin with nn_. To not confuse our function with a native torch function we begin the name with a different, yet we hope recognizable, letter sequence. The ann can be thought of as denoting "artificial NN".

Transfer learning from linear models to neural network models

It is very common to start model fits informed by previously fit NNs. Recognizing that NNs are a generalization of linear models one can start an NN model fit informed by a linear model. Basically one can chose some of the weights (like betas in a linear regression) and biases (like intercepts) to replicate the informing linear model, and let the other weights and biases be chosen at random as is common, in many contexts usual, when fitting a NN.

The demonstrate such a NN model fit informed by a linear model we use the nested.glmnetr() function. This is out of convenience in that the nested.glmenter() function already fits lasso linear and NN models and so contains the pieces to combine the two for transfer learning.

A NN model based upon the Cox regression informed by a lasso model

A NN for "tabular" data informed by a linear model can be fit using nested.glmnetr()

```
## Fit an AN informed with staring point for iterative fit by a lasso fit
time_start = diff_time()
```

```
## Start at Sys.time = 2024-03-23 00:26:19.880513
```

```
nested_cox_fit_ex2 = nested.glmnetr(xs=xs, y_=yt, event=event, family="cox",
                    dolasso=1, doann=1, ensemble=c(0,0,0,1), folds_n=10, do_ncv=0,
                    seed=c(101844880,882560297),track=0)
time_last = diff_time(time_start)
```

```
## Sys.time = 2024-03-23 00:28:42.851677,  elapsed time = 0:2:22 h:m:s
```

Here we see that to fit a NN informed by a lasso model the user need do very little beyond specifying the data for the fit. In addition to providing the usual data for a Cox model, one specifies doann=1 to "do an ann" model, and ensemble=c(0,0,1) where the ensemble[3] = 1 indicates the NN model is to be fit informed by the lasso model. The folds_n=10 specifies there are to be 10 folds for CV, and the do_ncv = 0 indicates to not do a nested CV, that is to just do the one layer of CV. If unspecified, do_ncv defaults to 1 and a nested CV will be performed allowing one to assess and compare model performances. How to specify these terms is described in the reference manual. Here we used diff_time() to track how long the program took (Intel(R) i7) to fit the NN model.

One can get model information about the model fit similar to when using the ann_tab_cv() function, for example

```
## The lasso informed model fit is saved to "object"$ann_fit_4
## The object that provides the lasso information to ann_fit_4 is "object"$cv_glment_fit
nested_cox_fit_ex2$ann_fit_4$modelsum
```

```
##        n folds         epochs      length Z1      length Z2           actv
##     10.0000000    200.0000000     18.0000000     10.0000000      1.0000000
##          drpot           mylr             wd             l1          lasso
##      0.0000000      0.0010000      0.0000000      0.0000000      1.0000000
##         lscale          scale     which loss    which agree        CV loss
##      5.0000000      1.0000000      1.0000000      1.0000000      3.2263818
##       CV Agree    CV accuracy     naive loss    naive agree naive accuracy
##      0.8386678      0.0000000      5.5096259      0.8398991      0.0000000
##     agree i_=0         bestof
##      0.8398991      1.0000000
```

```
## This shows the number of terms, weights and biases, for the the two hidden layers
str(nested_cox_fit_ex2$ann_fit_4$model$parameters)
```

```
## List of 6
##  $ 0.weight:Float [1:18, 1:100]
##  $ 0.bias  :Float [1:18]
##  $ 3.weight:Float [1:10, 1:18]
##  $ 3.bias  :Float [1:10]
##  $ 6.weight:Float [1:1, 1:10]
##  $ 6.bias  :Float [1:1]
```

From this we also see the number of predictors the last hidden layer to be 10. As we have implemented this lasso informed fit we only include those terms that had non zero coefficients in the tuned relaxed lasso model. We can inspect the model structure as in

```
# view more information on the NN model structure
nested_cox_fit_ex2$ann_fit_4$model$modules[[1]]
```

```
## An 'nn_module' containing 2,019 parameters.
##
## -- Modules ---------------------------------------------------------------------
## * 0: <nn_module> #1,818 parameters
## * 1: <nn_relu> #0 parameters
## * 2: <nn_dropout> #0 parameters
## * 3: <nn_module> #190 parameters
## * 4: <nn_relu> #0 parameters
## * 5: <nn_dropout> #0 parameters
## * 6: <nn_module> #11 parameters
## * 7: <nn_identity> #0 parameters
```

We can obtain predicteds from the lasso informed NN models. Since this involves combining information from the lasso and NN fits we wrap this in the function predict_ann_tab() with an example usage of

```
#print(ann_cv_lin_fit1$model)
preds = predict_ann_tab(nested_cox_fit_ex2, xs, modl=4)
preds[1:10]
```

```
##  [1] -0.20432752 -2.54530740  2.51898217  0.92521775 -0.37982887  0.68683004
##  [7] -3.08902073  0.06633738  3.85693717  0.62727892
```

Here the first input is the nested.glmnetr() output object, the second input the data for which we want the predicteds and finally modl=3 means we want the predicted from the model indicated by ensemble[3] = 1 in the call to nested.glmentr(). Note, the predictds are output as a numerical vector value rathe thatn as a torch tensor generally allowing easier processing of the predicteds in the R environment.

```
# ann_tab_cv() linear calibration
cox_fit3 = coxph(Surv(yt, event) ~ preds)
summary(cox_fit3)
```

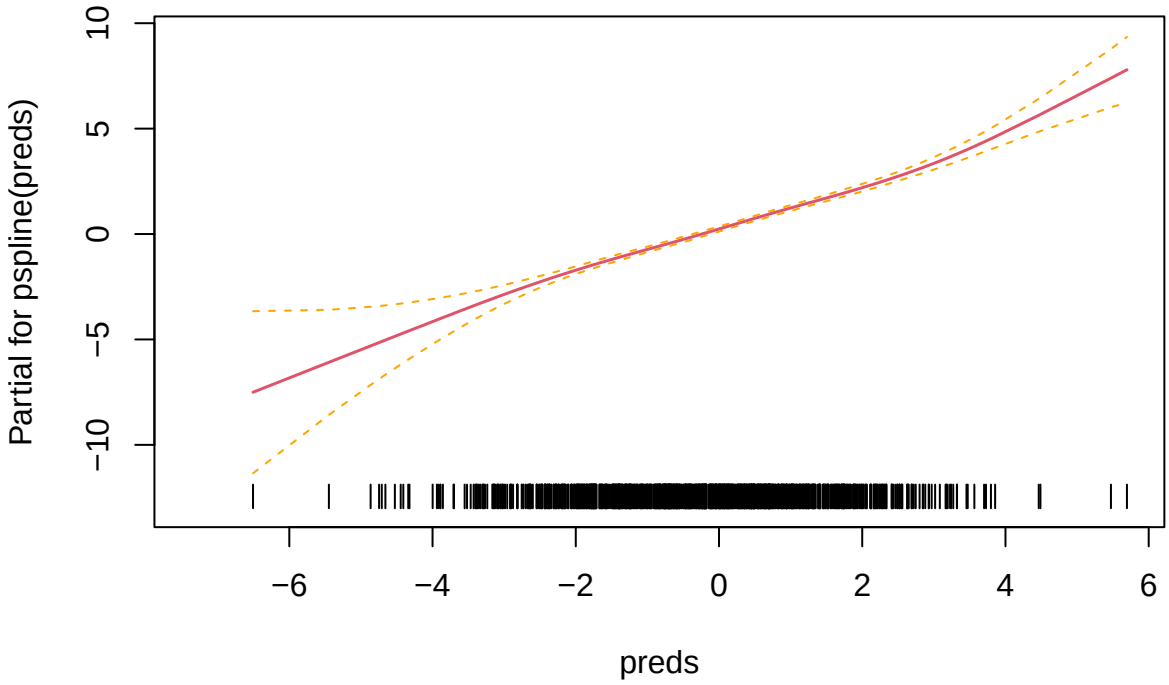
```
## Call:
## coxph(formula = Surv(yt, event) ~ preds)
##
##   n= 1000, number of events= 699
##
##          coef exp(coef) se(coef)     z Pr(>|z|)
## preds 1.00689   2.73706  0.03363 29.94   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
##       exp(coef) exp(-coef) lower .95 upper .95
## preds     2.737     0.3654     2.562     2.924
##
## Concordance= 0.84  (se = 0.007 )
## Likelihood ratio test= 980.5  on 1 df,   p=<2e-16
## Wald test            = 896.2  on 1 df,   p=<2e-16
## Score (logrank) test = 869.9  on 1 df,   p=<2e-16
```

Here the liner calibration coefficient is very near to 1, much nearer than for the model not informed by the lasso fit.

```
# Fit a spline to preds using coxph, and plot
cox_fit4 = coxph(Surv(yt, event) ~ pspline(preds))
summary(cox_fit4)
```

```
## Call:
## coxph(formula = Surv(yt, event) ~ pspline(preds))
##
##   n= 1000, number of events= 699
##
##                         coef  se(coef) se2     Chisq  DF   p
## pspline(preds), linear 1.009 0.03345  0.03345 910.43 1.00 5.3e-200
## pspline(preds), nonlin                          7.35 3.07  6.5e-02
##
##             exp(coef) exp(-coef) lower .95 upper .95
## ps(preds)3  5.125e+00  1.951e-01 6.434e-01 4.083e+01
## ps(preds)4  2.627e+01  3.807e-02 6.407e-01 1.077e+03
## ps(preds)5  1.345e+02  7.436e-03 1.014e+00 1.784e+04
## ps(preds)6  6.680e+02  1.497e-03 2.698e+00 1.654e+05
## ps(preds)7  2.567e+03  3.895e-04 9.279e+00 7.103e+05
## ps(preds)8  7.636e+03  1.310e-04 2.890e+01 2.017e+06
## ps(preds)9  2.770e+04  3.610e-05 1.046e+02 7.333e+06
## ps(preds)10 8.221e+04  1.216e-05 3.089e+02 2.188e+07
## ps(preds)11 3.398e+05  2.943e-06 1.268e+03 9.112e+07
## ps(preds)12 2.597e+06  3.851e-07 9.278e+03 7.270e+08
## ps(preds)13 2.242e+07  4.461e-08 6.914e+04 7.270e+09
## ps(preds)14 1.971e+08  5.073e-09 4.047e+05 9.602e+10
##
## Iterations: 4 outer, 17 Newton-Raphson
##      Theta= 0.7801373
## Degrees of freedom for terms= 4.1
## Concordance= 0.84  (se = 0.007 )
## Likelihood ratio test= 989.8  on 4.07 df,   p=<2e-16
```

```
termplot(cox_fit4,term=1,se=T,rug=T)
```



The spline fit is nearer a straight line than was the case for the uninformed NN fit, but still not as straight as we might want.

A NN model based upon least squares informed by a lasso model

This model is essentialy a generalization of linear regression and can be fit, for example, by

```
nested_nrm_fit_ex3 = nested.glmnetr(xs=xs, y_=y_, family="gaussian", dolasso=1, doann=1,
        seed=c(17820414,95337508), ensemble=c(1,0,0,1), folds_n=10, do_ncv=0, track=-1)
```

We can inspect numerically the fit by

```
preds = predict_ann_tab(nested_nrm_fit_ex3, xs)
```

```
##
##  modl not spedified, modl =  4  used
```

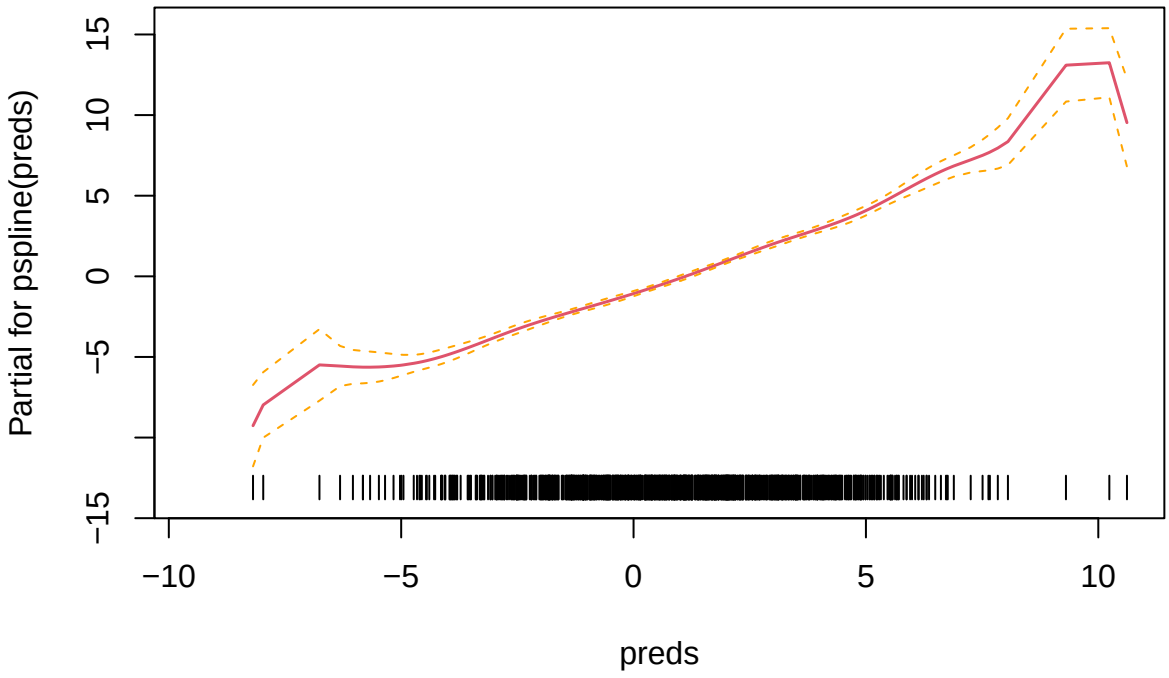
```
glm_fit1 <- glm(y_ ~ preds, family="gaussian")
summary(glm_fit1)
```

```
##
## Call:
## glm(formula = y_ ~ preds, family = "gaussian")
##
```

```
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.08006    0.04909  -1.631    0.103
## preds        0.99939    0.01729  57.817   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for gaussian family taken to be 2.143262)
##
##     Null deviance: 9303.4  on 999  degrees of freedom
## Residual deviance: 2139.0  on 998  degrees of freedom
## AIC: 3604.2
##
## Number of Fisher Scoring iterations: 2
```

and graphically by

```
glm_fit2 <- glm(y_ ~ pspline(preds), family="gaussian")
termplot(glm_fit2, rug=T, se=T)
```



Here the linear calibration coefficient is about 1 and the spline fit on the predicteds about linear with wiggle in the extremes where there are few data.

A NN model based upon logistic regression informed by a lasso model

An example NN fit based upon the logistic model frame work is

```
## fit a neural network based upon logistic regression framework
set.seed(  4695289  )
torch_manual_seed(  24260321  )
nested_bin_fit_ex4 = nested.glmnetr(xs=xs, y_=yb, family="binomial",
                     dolasso=1, doann=1, ensemble=c(1,0,1,0), folds_n=5, do_ncv=0,
                     track=0)
```

```
## print a short summary
summary( nested_bin_fit_ex4 )
```

```
##   Sample information including number of records, events, number of columns in
##   design (predictor, X) matrix, and df (rank) of design matrix:
##     family          n     nevent xs.columns      xs.df null.dev/n
##   binomial       1000        609        100         94       1.34
##
##  For LASSO, Artificial Neural Networks (ANN), naive performance measures
##  calculated using all data without cross validaiton are given
##
##                   Naive DevRat Naive Concordance Non Zero
## LASSO min               0.4411            0.9137       40
## LASSO minR              0.4169            0.8998       10
## LASSO minR.G0           0.4203            0.9000       10
## Ridge                   0.4723            0.9290       99
##
##                   Naive DevRat Naive Concordance Non Zero
## ANN Uninformed          0.8088            0.9911      100
## ANN lasso weights       0.4404            0.9076      100
```

Comparison of lasso and neural network models

As the nested.glmnetr() function performs nested cross validation of the lasso and NN models we can use it to compare performances between these models. An example is

```
## print a summary
nested_nrm_fit_ex5
```

```
##   Sample information including number of records, number of columns in
##   design (predictor, X) matrix, and df (rank) of design matrix:
##     family          n xs.columns      xs.df null.dev/n
##   gaussian       1011        100         94       13.9
##
##  For LASSO, Artificial Neural Networks (ANN), average (Ave) model performance
##  measures from the 10-fold (nested) cross validation are given together with naive
##  summaries calculated using all data without cross validaiton
##
##                          Ave DevRat Ave Int Ave Slope Ave R-square Ave Non Zero
## LASSO min                    0.8280  0.0427    1.0519       0.8333         34.3
## LASSO minR                   0.8333  0.0112    0.9972       0.8373         18.2
## LASSO minR.G0                0.8333  0.0112    0.9972       0.8373         18.2
## Ridge                        0.8073  0.0827    1.1100       0.8189         99.0
##                          Naive DevRat Naive R-square Non Zero
```

```
## LASSO min                       0.8430         0.9195       33
## LASSO minR                      0.8433         0.9183       17
## LASSO minR.G0                   0.8433         0.9183       17
## Ridge                           0.8511         0.9231       99
##
##                            Ave DevRat Ave Int Ave Slope Ave R-square Ave Non Zero
## ANN Uninformed                 0.7088 -0.0590    0.9636       0.7144          100
## ANN lasso weights, reset       0.8295  0.0628    0.9975       0.8363          100
##                            Naive DevRat Naive R-square Non Zero
## ANN Uninformed                   0.9365         0.9366      100
## ANN lasso weights, reset         0.8421         0.8432      100
```

where the NN model with an R-square of 0.91 seems to perform similar to the lasso models with an R-squares of about 0.92. Further, the NN fit without this transfer of information from the lasso model with it's R-square of 0.85, did not perform nearly as well as the lasso informed NN or the lasso model itself. This shows the direct benefit of this transfer learning of the information from the linear model when fitting NN models. It is not the NN itself that it is performing competitivley with the lasso model but the NN model fit in conjunction with the lasso model information. We also see the cross validated linear calibration coefficients are about 1 for the lasso informed NN and the tuned relaxed lasso model suggesting these models may be reasonably well calibrated. The fully penalized lasso and the ridge regression model, with cross validated linear calibration coefficients deviating more form 1, are less well calibrated.

For this example we deliberately simulated data where there are interactions or product terms in the analysis data set. This we did by setting intr=c(1,0,0,1) in the call to glmnetr.simdata(). NNs can, if there is sufficient information in the data, pick up non-linear and interaction (or product) terms. In the absence of non-linearity or interactions a strict linear model should out perform a NN model because it more parsimoniously uses model parameters.

We do not show more simulation results recognizing that one can typically show one model to be better than the others by simulating the right data set. Instead others can run the nested.glmnetr() function on their own data sets, potentially historical data sets if at hand, and see which models tend to perform better in their setting.

Internal implementaiton of the "transfer learning"

The NN model informed by the relaxed lasso model when ensemble[3]=1 adds a column of predicteds to the input matrix in the nested.glmnetr() call, and then extends the hidden layers to carry the positive and negative part of the lasso predicteds through to the final model output. It also sets weights and biases for this new column so that there is at the beginning of the model fitting process no communication between the lasso predicteds and the other predictor variables. A second option set by ensemble[2] = 1 appends the lasso predicteds to the xs predictor matrix and treats this similar to the other input variables. The NN model informed by the relaxed lasso model when ensemble[4]=1 fits like with ensemble[3] = 1 but reassigns weights and biases at each optimization epoch. The models fit for ensemble[i+4]=1 are like those fit for ensemble[i]+1 except only those variables with non-zero coefficients in the lasso model are included in the input dataset.

Before fitting the NN models the predictor variables are standardized to have mean 0 and standard deviation of 1. This is accounted for when deriving predicteds. This is important when using L1 (lasso) or L2 (ridge or weight decay) penalties to assure the models are not scale dependent. This is also done in the *glmnet* package functions.

Further extensions of "transfer learning"

Just as one can assign a subset of the bias and weight (initial) values for the NN to replicate a linear model, and thereby improve model fit, one can also define another subset of the bias and weight values to replicate linear splines. One attractive feature of the NN numerical optimization is that the bias terms dictating where the spline "knots" are updated during model fit to improve fit, that is they need not be fixed in advance of the fitting.

Transfer learning and the gradient boosting machine

Just as we have informed the NN model with the results form a relaxed lasso model fit we an do the same with gradient boosting machines by either adding the lasso predicted as an additional predictor or including it as an offset. This is done by setting doxgb=1 in the nested.glmnetr() call. Results are similar but different from those for the NN models. For our data sets the GBMs took much longer to run since we used a search algorithm to find a "best" hyperparameter set.