Building Deep Learning Models

Jenna Reps, Egill Fridgeirsson, Chungsoo Kim, Henrik John, Seng Chan You, Xiaoyong Pan

2022-08-17

Contents

1 Introduction

1.1 DeepPatientLevelPrediction

Patient level prediction aims to use historic data to learn a function between an input (a patient's features such as age/gender/comorbidities at index) and an output (whether the patient experienced an outcome during some time-at-risk). Deep learning is example of the the current state-of-the-art classifiers that can be implemented to learn the function between inputs and outputs.

Deep Learning models are widely used to automatically learn high-level feature representations from the data, and have achieved remarkable results in image processing, speech recognition and computational biology. Recently, interesting results have been shown using large observational healthcare data (e.g., electronic healthcare data or claims data), but more extensive research is needed to assess the power of Deep Learning in this domain.

This vignette describes how you can use the Observational Health Data Sciences and Informatics (OHDSI) `PatientLevelPrediction` package and `DeepPatientLevelPrediction` package to build Deep Learning models. This vignette assumes you have read and are comfortable with building patient level prediction models as described in the `BuildingPredictiveModels` vignette. Furthermore, this vignette assumes you are familiar with Deep Learning methods.

1.2 Background

Deep Learning models are build by stacking an often large number of neural network layers that perform feature engineering steps, e.g embedding, and are collapsed in a final linear layer (equivalent to logistic regression). These algorithms need a lot of data to converge to a good representation, but currently the sizes of the large observational healthcare databases are growing fast which would make Deep Learning an interesting approach to test within OHDSI's Patient-Level Prediction Framework. The current implementation allows us to perform research at scale on the value and limitations of Deep Learning using observational healthcare data.

In the package we have used torch but we invite the community to add other backends.

Many network architectures have recently been proposed and we have implemented a number of them, however, this list will grow in the near future. It is important to understand that some of these architectures require a 2D data matrix, i.e. |patient|x|feature|, and others use a 3D data matrix |patient|x|feature|x|time|. The FeatureExtraction Package has been extended to enable the extraction of both data formats as will be described with examples below.

Note that training Deep Learning models is computationally intensive, our implementation therefore supports both GPU and CPU. It will automatically check whether there is GPU or not in your computer. A GPU is highly recommended for Deep Learning!

1.3 Requirements

Full details about the package requirements and instructions on installing the package can be found here.

1.4 Integration with PatientLevelPrediction

The `DeepPatientLevelPrediction` package provides additional model settings that can be used within the `PatientLevelPrediction` package `runPlp()` function. To use both packages you first need to pick the deep learning architecture you wish to fit (see below) and then you specify this as the modelSettings inside `runPlp()`.

```r
# load the data
plpData <- PatientLevelPrediction::loadPlpData('locationOfData')

# pick the set<Model> from  DeepPatientLevelPrediction
deepLearningModel <- DeepPatientLevelPrediction::setResNet()

# use PatientLevelPrediction to fit model
deepLearningResult <- PatientLevelPrediction::runPlp(
    plpData = plpData,
    outcomeId = 1230,
    modelSettings = deepLearningModel,
    analysisId = 'resNetTorch',
    ...
  )
```

2 Non-Temporal Architectures

We implemented the following non-temporal (2D data matrix) architectures:

2.1 Simple MLP

2.1.1 Overall concept

A multilayer perceptron (MLP) model is a directed graph consisting of an input layer, one or more hidden layers and an output layer. The model takes in the input feature values and feeds these forward through the graph to determine the output class. A process known as ‘backpropagation’ is used to train the model. Backpropagation requires labelled data and involves automatically calculating the derivative of the model parameters with respect to the the error between the model’s predictions and ground truth. Then the model learns how to adjust the model’s parameters to reduce the error.

2.1.2 Example

2.1.2.1 Set Function To use the package to fit a MLP model you can use the `setDeepNNTorch()` function to specify the hyper-parameter settings for the MLP.

2.1.2.2 Inputs The `units` input defines the network topology via the number of neurons per layer in the network’s hidden layers. A list of different topologies can be investigated. `list(c(10,63), 128)` means two different topologies will be fit, the first has two hidden layers with 10 neurons in the first hidden layer and 63 in the second hidden layer. The second just has one hidden layer with 128 neurons.

The `layer_dropout` input specifies the probability that a layer randomly sets input units to 0 at each step during training time. A value of `0.2` means that 20% of the time the layer input will be set to 0. This is used to reduce overfitting.

The `lr` input is the learning rate which is a hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated. The smaller the `lr` the longer it will take to fit the model and the model weights may get stuck, but if the `lr` is too large, the weights may sub-optimally converge too fast.

The `decay` input corresponds to the weight decay in the objective function. During model fitting the aim is to minimize the objective function. The objective function is made up of the prediction error (the difference between the prediction vs the truth) plus the square of the weights multiplied by the weight decay. The larger the weight decay, the more you penalize having large weights. If you set the weight decay too large, the model will never fit well enough, if you set it too low, you need to be careful of overfitting (so try to stop model fitting earlier).

The `outcome_weight` specifies whether to add more weight to misclassifying one class (e.g., with outcome during TAR) vs the other (e.g., without outcome during TAR). This can be useful if there is imbalance between the classes (e.g., the outcome rarely occurs during TAR). However be careful since this will also result in miscalibrated models which need to be recalibrated.

The `batch_size` corresponds to the number of data points (patients) used per iteration to estimate the network error during model fitting.

The epochs corresponds to how many time to run through the entire training data while fitting the model.

The seed lets the user reproduce the same network given the same training data and hyper-parameter settings if they use the same seed.

2.1.2.3 Example Code For example, the following code will try two different network topologies and pick the topology that obtains the greatest AUROC via cross validation in the training data and then fit the model with that topology using all the training data. The standard output of runPlp() will be returned - this contains the MLP model along with the performance details and settings.

```r
#singleLayerNN(inputN = 10, layer1 = 100, outputN = 2, layer_dropout = 0.1)
deepset <- setDeepNNTorch(
  units = list(c(10,63), 128),
  layer_dropout = c(0.2),
  lr = c(1e-4),
  decay = c(1e-5),
  outcome_weight = c(1.0),
  batch_size = c(100),
  epochs = c(5),
  seed = 12
  )

mlpResult <- PatientLevelPrediction::runPlp(
    plpData = plpData,
    outcomeId = 3,
    modelSettings = deepset,
    analysisId = 'DeepNNTorch',
    analysisName = 'Testing Deep Learning',
    populationSettings = populationSet,
    splitSettings = PatientLevelPrediction::createDefaultSplitSetting(),
    sampleSettings = PatientLevelPrediction::createSampleSettings(),  # none
    featureEngineeringSettings = PatientLevelPrediction::createFeatureEngineeringSettings(), # none
    preprocessSettings = PatientLevelPrediction::createPreprocessSettings(),
    executeSettings = PatientLevelPrediction::createExecuteSettings(
      runSplitData = T,
      runSampleData = F,
      runfeatureEngineering = F,
      runPreprocessData = T,
      runModelDevelopment = T,
      runCovariateSummary = F
    ),
    saveDirectory = file.path(testLoc, 'DeepNNTorch')
  )
```

2.2 ResNet

2.2.1 Overall concept

Deep learning models are often trained via a process known as gradient descent during backpropogation. During this process the network weights are updated based on the gradient of the error function for the current weights. However, as the number of layers in the network increase, there is a greater chance of experiencing an issue known as the vanishing or exploding gradient during this process. The vanishing or exploding gradient is when the gradient goes to 0 or infinity, which negatively impacts the model fitting.

The residual network (ResNet) was introduced to address the vanishing or exploding gradient issue. It works by adding connections between non-adjacent layers, termed a ‘skip connection’. Using some form of regularization with these ‘skip connections’ enables the network to ignore any problematic layer that resulted due to gradient issues.

The ResNet calculates embeddings for every feature and then averages them to compute an embedding per patient.

This implementation of a ResNet for tabular data is based on this paper.

2.2.2 Example

2.2.2.1 Set Function To use the package to fit a ResNet model you can use the `setResNet()` function to specify the hyperparameter settings for the network.

2.2.2.2 Inputs

2.2.2.2.1 Model inputs: `numLayers`: How many layers to use in the model.

`sizeHidden`: How many neurons in each hidden layer

`hiddenFactor`: How much to increase number of neurons in each layer

`residualDropout` and`hiddenDropout` : How much dropout to apply in hidden layer or residual connection

`sizeEmbedding` : The size of the initial embedding layer

2.2.2.2.2 Training process inputs: `weightDecay` : How much weight decay to apply, which penalizes bigger weights

`learningRate` : Which learning rate to use

`seed` : Use a seed for reproducibility

`device` : Which device to use, such as a cpu or a gpu

`batchSize` : Size of batch of data used per iteration during training

`epochs` : How many runs through the data

2.2.2.2.3 Hyperparameter tuning inputs: `hyperParamSearch` : Which type of hyperparameter search to use, either random sampling or exhaustive (grid) search

`randomSample`: If doing a random search for hyperparameters, how many random samples to use

2.2.2.3 Example Code For example, the following code will fit a two layer ResNet where each layer has 32 neurons which increases by a factor of two before decreasing againg (hiddenFactor). 10% of inputs to each layer and residual connection within the layer are randomly zeroed. The embedding layer has 32 neurons. Learning rate of 3e-4 with weight decay of 1e-6 is used for the optimizer. No hyperparameter search is done since each input only includes one option.

```r
resset <- setResNet(
  numLayers = c(2),
  sizeHidden = c(32),
  hiddenFactor = c(2),
  residualDropout = c(0.1),
  hiddenDropout = c(0.1),
  sizeEmbedding = c(32),
  weightDecay = c(1e-6),
  learningRate = c(3e-4),
  seed = 42,
  hyperParamSearch = 'random',
  randomSample = 1,
  #device='cuda:0', # uncomment to use GPU
  batchSize = 128,
  epochs = 3
)

resResult <- PatientLevelPrediction::runPlp(
    plpData = plpData,
    outcomeId = 3,
    modelSettings = resset,
    analysisId = 'ResNet',
    analysisName = 'Testing ResNet',
    populationSettings = populationSet,
    splitSettings = PatientLevelPrediction::createDefaultSplitSetting(),
    sampleSettings = PatientLevelPrediction::createSampleSettings(),  # none
    featureEngineeringSettings = PatientLevelPrediction::createFeatureEngineeringSettings(), # none
    preprocessSettings = PatientLevelPrediction::createPreprocessSettings(),
    executeSettings = PatientLevelPrediction::createExecuteSettings(
      runSplitData = T,
      runSampleData = F,
      runfeatureEngineering = F,
      runPreprocessData = T,
      runModelDevelopment = T,
      runCovariateSummary = F
    ),
    saveDirectory = file.path(getwd(), 'ResNet') # change to save elsewhere
  )
```

2.3 Transformer

2.3.1 Overall concept

Recently there has been a surge of models in natural language processing and computer vision that utilize attention. This is a technique where the model learns where to look and what to focus on in the input data. This was first described in the attention is all you need paper. Here we have used an implementation that has shown good performance on non-temporal tabular data from this paper.

This architecture is computationally expensive and scales badly with longer sequence length. In this case the sequence is the amount of features each patient has. Users need to be aware of how many features they are feeding to the model since this will effect the computation time heavily. This is something you control in `FeatureExtraction` when you create your covariate settings.

2.3.2 Examples

2.3.2.1 Set Function To use the package to fit a Transformer model you can use the `setTransformer()` function to specify the hyperparameter settings for the network.

2.3.2.2 Inputs

The training and hyperparameter tuning inputs are the same as for the ResNet.

2.3.2.2.1 Model inputs:

`numBlocks` : How many Transformer blocks to use, each block includes a self-attention layer and a feedforward block with two linear layers.

`dimToken` : Dimension of the embedding for each feature's embedding

`dimOut` : Dimension of output, for binary problems this is 1.

`numHeads` : Number of attention heads for the self-attention

`attDropout` , `ffnDropout` and `resDropout` : How much dropout to apply on attentions, in feedforward block or in residual connections

`dimHidden` : How many neurons in linear layers inside the feedforward block

```r
modelSettings <- setTransformer(numBlocks = 3,
                                dimToken = 32,
                                dimOut = 1,
                                numHeads = 4,
                                attDropout = 0.25,
                                ffnDropout = 0.25,
                                resDropout = 0,
                                dimHidden = 128,
                                weightDecay = 1e-06,
                                learningRate = 3e-04
                                batchSize = 128,
                                epochs = 10,
                                device = 'cpu',  # or 'cuda' for GPU
                                randomSamples = 1)


TransformerResult <- PatientLevelPrediction::runPlp(
    plpData = plpData,
    outcomeId = 3,
    modelSettings = modelSettings,
    analysisId = 'Transformer',
    analysisName = 'Testing transformer',
    populationSettings = populationSet,
    splitSettings = PatientLevelPrediction::createDefaultSplitSetting(),
    sampleSettings = PatientLevelPrediction::createSampleSettings(),  # none
    featureEngineeringSettings = PatientLevelPrediction::createFeatureEngineeringSettings(), # none
    preprocessSettings = PatientLevelPrediction::createPreprocessSettings(),
    executeSettings = PatientLevelPrediction::createExecuteSettings(
      runSplitData = T,
      runSampleData = F,
      runfeatureEngineering = F,
      runPreprocessData = T,
      runModelDevelopment = T,
      runCovariateSummary = F
    ),
    saveDirectory = file.path(getwd(), 'Transformer') # change to save elsewhere
  )
```

2.3.2.3 Example Code

3 Acknowledgments

Considerable work has been dedicated to provide the `DeepPatientLevelPrediction` package.

```
citation("DeepPatientLevelPrediction")
```

```
##
## To cite package 'DeepPatientLevelPrediction' in publications use:
##
##   Reps J, Fridgeirsson E, Chan You S, Kim C, John H (2021). _DeepPatientLevelPrediction: Deep Learnin
##   For Patient Level Prediction Using Data In The OMOP Common Data Model_.
##   https://ohdsi.github.io/PatientLevelPrediction, https://github.com/OHDSI/DeepPatientLevelPredictio
##
## A BibTeX entry for LaTeX users is
##
##   @Manual{,
##     title = {DeepPatientLevelPrediction: Deep Learning For Patient Level Prediction Using Data In The
##     author = {Jenna Reps and Egill Fridgeirsson and Seng {Chan You} and Chungsoo Kim and Henrik John
##     year = {2021},
##     note = {https://ohdsi.github.io/PatientLevelPrediction, https://github.com/OHDSI/DeepPatientLevel
##   }
```

Please reference this paper if you use the PLP Package in your work:

Reps JM, Schuemie MJ, Suchard MA, Ryan PB, Rijnbeek PR. Design and implementation of a standardized framework to generate and evaluate patient-level prediction models using observational healthcare data. J Am Med Inform Assoc. 2018;25(8):969-975.