deepG tutorial

Contents

Introduction

The deepG library can be used for applying deep learning on genomic data. The library supports creating neural network architecture, automation of data preprocesing (data generator), network training, inference and visualizing feature importances (integrated gradients).

Create a model

deepG supports three functions to create a keras model.

`create_model_lstm_cnn`

The architecture of this model is k * LSTM, m * CNN and n * dense layers, where $k, m \geq 0$ and $n \geq 1$.
The user can choose the size of the individual LSTM, CNN and Dense layers and add additional features to each layer; for example the LSTM layer may be bidirectional or stateful.
The last dense layer layer has a softmax activation and determines how many targets we want to predict.

The following implementation creates a model with 1 CNN layer (+ batch normalization), 2 LSTM and 2 dense layers.

```
library(deepG)
```

```
## Warning: replacing previous import 'data.table::last' by 'dplyr::last' when
## loading 'deepG'

## Warning: replacing previous import 'data.table::first' by 'dplyr::first' when
## loading 'deepG'

## Warning: replacing previous import 'data.table::between' by 'dplyr::between'
## when loading 'deepG'

## The deepG package has been successfully loaded.
```

```
model <- create_model_lstm_cnn(
  maxlen = 20,
  layer_lstm = c(8, 8),
  layer_dense = c(16, 4),
  vocabulary.size = 4,
  kernel_size = c(3),
  filters = c(4)
)
```

```
## Model: "model"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## input_1 (InputLayer)                [(None, 20, 4)]                 0
## ________________________________________________________________________________
## conv1d (Conv1D)                     (None, 20, 4)                   52
## ________________________________________________________________________________
## batch_normalization (BatchNormaliza (None, 20, 4)                   16
## ________________________________________________________________________________
## lstm (LSTM)                         (None, 20, 8)                   416
## ________________________________________________________________________________
## lstm_1 (LSTM)                       (None, 8)                       544
## ________________________________________________________________________________
## dense (Dense)                       (None, 16)                      144
## ________________________________________________________________________________
## dense_1 (Dense)                     (None, 4)                       68
## ================================================================================
## Total params: 1,240
## Trainable params: 1,232
## Non-trainable params: 8
## ________________________________________________________________________________
```

The model expects an input with dimensions (NULL (batch size), maxlen, vocabulary size) and a target with dimension (NULL (batch size), number of targets). Maxlen specifies the length of the input sequence.

```
batch_size <- 3
maxlen <- 20
vocabulary.size <- 4
input <- array(rnorm(maxlen * batch_size * vocabulary.size),
               dim = c(batch_size, maxlen, vocabulary.size))
pred <- predict(model, input)
dim(pred)
```

```
## [1] 3 4
```

`create_model_lstm_cnn_target_middle`

This architecture is closely related to `create_model_lstm_cnn_target` with the main difference that the model has two input layers (provided `label_input = NULL`).

```
model <- create_model_lstm_cnn_target_middle(
  maxlen = 20,
  layer_lstm = c(8, 8),
  layer_dense = c(16, 4),
  vocabulary.size = 4,
  kernel_size = c(3),
  filters = c(4)
)
```

```
## Model: "model_1"
## ________________________________________________________________________________
## Layer (type)                Output Shape      Param #   Connected to
## ================================================================================
## input_2 (InputLayer)        [(None, 10, 4)]   0
## ________________________________________________________________________________
## input_3 (InputLayer)        [(None, 10, 4)]   0
## ________________________________________________________________________________
## conv1d_1 (Conv1D)           (None, 10, 4)     52        input_2[0][0]
## ________________________________________________________________________________
## conv1d_2 (Conv1D)           (None, 10, 4)     52        input_3[0][0]
## ________________________________________________________________________________
## batch_normalization_1 (Ba (None, 10, 4)       16        conv1d_1[0][0]
## ________________________________________________________________________________
## batch_normalization_2 (Ba (None, 10, 4)       16        conv1d_2[0][0]
## ________________________________________________________________________________
## lstm_2 (LSTM)               (None, 10, 8)     416       batch_normalization_1[0][0]
## ________________________________________________________________________________
## lstm_4 (LSTM)               (None, 10, 8)     416       batch_normalization_2[0][0]
## ________________________________________________________________________________
## lstm_3 (LSTM)               (None, 8)         544       lstm_2[0][0]
## ________________________________________________________________________________
## lstm_5 (LSTM)               (None, 8)         544       lstm_4[0][0]
## ________________________________________________________________________________
## concatenate (Concatenate) (None, 16)          0         lstm_3[0][0]
##                                                      lstm_5[0][0]
## ________________________________________________________________________________
## dense_2 (Dense)            (None, 16)        272      concatenate[0][0]
## ________________________________________________________________________________
## dense_3 (Dense)            (None, 4)         68       dense_2[0][0]
## ================================================================================
## Total params: 2,396
## Trainable params: 2,380
## Non-trainable params: 16
## ________________________________________________________________________________
```

This architecture can be used to predict a character in the middle of a sequence. For example

sequence: `ACCGTGGAA`

then the first input should correspond to `ACCG`, the second input to `GGAA` and `T` to the target.
This can be used to combine the 2 tasks

1. predict `T` given `ACCG`
2. predict `T` given `AAGG` (note reversed order of input)

in one model.

Training

Preparing the data

Input data must be files in FASTA or FASTQ format and file names must have .fasta or .fastq ending; otherwise files will be ignored. All training and validation data should each be in one folder. deepG uses a data generator to iterate over files in train/validation folder.
Before we train our model, we have to decide what our training objetive is. It can be either a language model or label classification.

```r
#### dummy data for now ####
path_16S_train <- "/home/rmreches/testData/labelDataFolder/train1234"
path_16S_validation <- "/home/rmreches/testData/labelDataFolder/val1234"
path_bacteria_train <- "/home/rmreches/testData/labelDataFolder/train4321"
path_bacteria_validation <- "/home/rmreches/testData/labelDataFolder/val4321"

checkpoint_path <- "/home/rmreches/checkpoints"
tensorboard.log <- "/home/rmreches/tensorboard"
dir_path <- "/home/rmreches/outputs"
if (!dir.exists("/home/rmreches/outputs")) dir_path <- dir.create("/home/rmreches/outputs")
```

Language model

With language model, we mean a model that predicts a character in a sequence. The target can be at the end of the sequence, for example

`ACGTCAG`

or in the middle

`ACGTCAG`

Language model for 16S (predict next character)

Say we want to predict the next character in a sequence given the last 500 characters and our text consists of the letters `A,C,G,T`. First we have to create a model. We may use a model with 1 LSTM, 3 CNN and one dense layer for predictions.

```
model <- create_model_lstm_cnn(
               maxlen = 50, ### change later
               kernel_size = c(12, 12, 12),
               filters = c(64, 64, 64),
               pool_size = c(3, 3, 3),
               layer_lstm = c(32),
               layer_dense = c(4),
               learning.rate = 0.001,
               vocabulary.size = 4)
```

```
## Model: "model_2"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## input_4 (InputLayer)                [(None, 50, 4)]                 0
## ________________________________________________________________________________
## conv1d_3 (Conv1D)                   (None, 50, 64)                  3136
## ________________________________________________________________________________
## max_pooling1d (MaxPooling1D)        (None, 16, 64)                  0
## ________________________________________________________________________________
## batch_normalization_3 (BatchNormali (None, 16, 64)                  256
## ________________________________________________________________________________
## conv1d_4 (Conv1D)                   (None, 16, 64)                  49216
## ________________________________________________________________________________
## batch_normalization_4 (BatchNormali (None, 16, 64)                  256
## ________________________________________________________________________________
## max_pooling1d_1 (MaxPooling1D)      (None, 5, 64)                   0
## ________________________________________________________________________________
## conv1d_5 (Conv1D)                   (None, 5, 64)                   49216
## ________________________________________________________________________________
## batch_normalization_5 (BatchNormali (None, 5, 64)                   256
## ________________________________________________________________________________
## max_pooling1d_2 (MaxPooling1D)      (None, 1, 64)                   0
## ________________________________________________________________________________
## lstm_6 (LSTM)                       (None, 32)                      12416
## ________________________________________________________________________________
## dense_4 (Dense)                     (None, 4)                       132
## ================================================================================
## Total params: 114,884
## Trainable params: 114,500
## Non-trainable params: 384
## ________________________________________________________________________________
```

Next we have to specify the location of our training and validation data and the output format of the data generator

```
trainNetwork(train_type = "lm",
             model = model,
             path = path_16S_train,
             path.val = path_16S_validation,
             checkpoint_path = checkpoint_path,
             tensorboard.log = tensorboard.log,
             validation.split = 0.2, # use 20% of samples for validation compared to train size
             run.name = "lm_16S_target_right",
             batch.size = 256,
             epochs = 10, ####
             steps.per.epoch = 2, ####
             step = 50,
             output = list(none = FALSE,
                           checkpoints = TRUE,
                           tensorboard = TRUE,
                           log = FALSE,
                           serialize_model = FALSE,
                           full_model = FALSE
             ),
             tb_images = TRUE,
             output_format = "target_right" # predict target at end of sequence
             )
```

```
## Trained on 2 samples (batch_size=NULL, epochs=10)
## Final epoch (plot to see history):
##     loss: 0.2115
##      acc: 1
##       f1: Inf
## val_loss: 0.8059
##  val_acc: 0.6992
##   val_f1: Inf
##       lr: 0.001
```

Predict character in middle of sequence

If we want to predict a character in the middle of a sequence and use LSTM layers, we should split our input into two layers. One layer handles the sequence before and one the input after the target. If, for example

sequence: `ACCGTGGAA`

then first input corresponds to `ACCG` and second to `AAGG`. We may create a model with two input layers using the `create_model_cnn_lstm_target_middle`

```
model <- create_model_lstm_cnn_target_middle(
  maxlen = 50,
  layer_lstm = c(8, 8),
  layer_dense = c(4),
  vocabulary.size = 4 # text consists of A,C,G,T
)
```

```
## Model: "model_3"
## ________________________________________________________________________________
## Layer (type)                 Output Shape      Param #   Connected to
## ================================================================================
## input_5 (InputLayer)      [(None, 25, 4)]   0
## ________________________________________________________________________________
## input_6 (InputLayer)      [(None, 25, 4)]   0
## ________________________________________________________________________________
## lstm_7 (LSTM)             (None, 25, 8)     416      input_5[0][0]
## ________________________________________________________________________________
## lstm_9 (LSTM)             (None, 25, 8)     416      input_6[0][0]
## ________________________________________________________________________________
## lstm_8 (LSTM)             (None, 8)         544      lstm_7[0][0]
## ________________________________________________________________________________
## lstm_10 (LSTM)            (None, 8)         544      lstm_9[0][0]
## ________________________________________________________________________________
## concatenate_1 (Concatenat (None, 16)        0        lstm_8[0][0]
##                                                      lstm_10[0][0]
## ________________________________________________________________________________
## dense_5 (Dense)           (None, 4)         68       concatenate_1[0][0]
## ================================================================================
## Total params: 1,988
## Trainable params: 1,988
## Non-trainable params: 0
## ________________________________________________________________________________
```

The `trainNetwork` call is identical to the previous model, except we have to change the output format of the generator by setting `output_format = "target_middle_lstm"`. This reverses the order of the sequence after the target.

```
trainNetwork(train_type = "lm", # running a language model
             model = model,
             path = path_16S_train,
             path.val = path_16S_validation,
             checkpoint_path = checkpoint_path,
             tensorboard.log = tensorboard.log,
             run.name = "lm_16S_target_middle",
             steps.per.epoch = 5, # use 5 batches per epoch
             validation.split = 0.2, # use 20% of samples for validation compared to train
             batch.size = 8,
             epochs = 2,
             output_format = "target_middle_lstm" # predict target in middle of sequence
)
```

```
## Trained on 5 samples (batch_size=NULL, epochs=2)
## Final epoch (plot to see history):
##     loss: 1.384
##      acc: 0.225
##       f1: Inf
## val_loss: 1.302
##  val_acc: 0.625
##   val_f1: Inf
##       lr: 0.001
```

Label classification

With label classification, we describe the task of mapping a label to a sequence. For example: given the sequence `ACGACCG`, does the sequence belong to a viral or bacterial genome?

deepG offers two options to map a label to a sequence

1. the label gets read from the fasta header
2. files from every class are in seperate folders

Label by folder

We put all data from one class into separate folders. Say we want to classify if a sequence belongs to viral or bacterial genomen. Then we have to put all virus/bacteria files into their own folder. In this case the `path` and `path.val` arguments should be vectors, where each entry is the path to one class.

```
model <- create_model_lstm_cnn(
               maxlen = 50,  #######
               kernel_size = c(12, 12, 12),
               filters = c(64, 64, 64),
               pool_size = c(3, 3, 3),
               layer_lstm = c(32),
               layer_dense = c(2),
               learning.rate = 0.001,
               vocabulary.size = 4)
```

```
## Model: "model_4"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## input_7 (InputLayer)                [(None, 50, 4)]                 0
## ________________________________________________________________________________
## conv1d_6 (Conv1D)                   (None, 50, 64)                  3136
## ________________________________________________________________________________
## max_pooling1d_3 (MaxPooling1D)      (None, 16, 64)                  0
## ________________________________________________________________________________
## batch_normalization_6 (BatchNormali (None, 16, 64)                  256
## ________________________________________________________________________________
## conv1d_7 (Conv1D)                   (None, 16, 64)                  49216
## ________________________________________________________________________________
## batch_normalization_7 (BatchNormali (None, 16, 64)                  256
## ________________________________________________________________________________
## max_pooling1d_4 (MaxPooling1D)      (None, 5, 64)                   0
## ________________________________________________________________________________
## conv1d_8 (Conv1D)                   (None, 5, 64)                   49216
## ________________________________________________________________________________
## batch_normalization_8 (BatchNormali (None, 5, 64)                   256
## ________________________________________________________________________________
## max_pooling1d_5 (MaxPooling1D)      (None, 1, 64)                   0
## ________________________________________________________________________________
## lstm_11 (LSTM)                      (None, 32)                      12416
## ________________________________________________________________________________
## dense_6 (Dense)                     (None, 2)                       66
## ================================================================================
## Total params: 114,818
## Trainable params: 114,434
## Non-trainable params: 384
## ________________________________________________________________________________
```

```
trainNetwork(train_type = "label_folder", # reading label from folder
             model = model,
             path = c(path_16S_train, # note that path has two entries
                      path_bacteria_train),
             path.val = c(path_16S_validation,
                          path_bacteria_validation),
             checkpoint_path = checkpoint_path,
             tensorboard.log = tensorboard.log,
             steps.per.epoch = 5, # use 5 batches per epoch
             validation.split = 0.2, # use 20% of samples for validation compared to training
             batch.size = 8,
             run.name = "16S_vs_bacteria",
             epochs = 2,
             step = 500,
             labelVocabulary = c("16S", "bacteria"), # names of classes
             output = list(none = FALSE,
                           checkpoints = TRUE,
                           tensorboard = TRUE,
                           log = FALSE,
                           serialize_model = FALSE,
                           full_model = FALSE
                          )
)
```

```
## Trained on 5 samples (batch_size=NULL, epochs=2)
## Final epoch (plot to see history):
##     loss: 0.1261
##      acc: 1
##       f1: 1
## val_loss: 0.4159
##  val_acc: 1
##   val_f1: 1
##       lr: 0.001
```

Inference

Once we have trained a model, we may use the model to get the activations of a certain layer and write the states to an h5 file.

```
print(model)
```

```
## Model
## Model: "model_4"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## input_7 (InputLayer)                [(None, 50, 4)]                 0
## ________________________________________________________________________________
## conv1d_6 (Conv1D)                   (None, 50, 64)                  3136
## ________________________________________________________________________________
## max_pooling1d_3 (MaxPooling1D)      (None, 16, 64)                  0
## ________________________________________________________________________________
## batch_normalization_6 (BatchNormali (None, 16, 64)                  256
## ________________________________________________________________________________
## conv1d_7 (Conv1D)                   (None, 16, 64)                  49216
## ________________________________________________________________________________
## batch_normalization_7 (BatchNormali (None, 16, 64)                  256
## ________________________________________________________________________________
## max_pooling1d_4 (MaxPooling1D)      (None, 5, 64)                   0
## ________________________________________________________________________________
## conv1d_8 (Conv1D)                   (None, 5, 64)                   49216
## ________________________________________________________________________________
## batch_normalization_8 (BatchNormali (None, 5, 64)                   256
## ________________________________________________________________________________
## max_pooling1d_5 (MaxPooling1D)      (None, 1, 64)                   0
## ________________________________________________________________________________
## lstm_11 (LSTM)                      (None, 32)                      12416
## ________________________________________________________________________________
## dense_6 (Dense)                     (None, 2)                       66
## ================================================================================
## Total params: 114,818
## Trainable params: 114,434
## Non-trainable params: 384
## ________________________________________________________________________________
```

```
num_layers <- length(model$get_config()$layers)
layer_name <- model$get_config()$layers[[num_layers]]$name
cat("get output at layer", layer_name)
```

```
## get output at layer dense_6
```

```
fasta.path <- list.files(path_16S_validation, full.names = TRUE)[1]
fasta.file <- microseq::readFasta(fasta.path)
head(fasta.file)
```

```
## # A tibble: 3 x 2
##   Header Sequence
##   <chr>  <chr>
## 1 1234   acgtaaccggttaaacccgggtttaaaaccccggggttttaaacccgggtttaaccggttacgt
## 2 1234   aaccggttaaacccgggtttaaaaccccggggttttaaaaacccccgggggtttttaaaaaaccccccgg~
## 3 1234   aaacccgggtttaaaaccccggggttttaaaaacccccgggggtttttaaaaaaccccccggggggtttt~
```

```
sequence <- fasta.file$Sequence[1]
filename <- file.path(dir_path, "states.h5")

if (!file.exists(filename)) {
  writeStates(
    model = model,
    layer_name = layer_name,
    sequence = sequence,
    round_digits = 4,
    filename = filename,
    batch.size = 10,
    mode = "lm")
}
```

We can access the h5 file as follows

```
states <- readRowsFromH5(h5_path = filename, complete = TRUE)
```

```
## states matrix has 14 rows and  2 columns
```

```
colnames(states) <- c("16S", "bacteria")
head(states)
```

```
##         16S bacteria
## [1,] 0.6134   0.3866
## [2,] 0.5785   0.4215
## [3,] 0.5777   0.4223
## [4,] 0.6654   0.3346
## [5,] 0.6815   0.3185
## [6,] 0.5787   0.4213
```

Features

Additional input vector

It is possible to feed a network additional information associated to a sequence. This information needs to be in a csv file. If all sequences in one file share the same label, the csv file should have one column named "file". If the label gets mapped to the header name, the csv file needs to have a column names "header".

We may add some additional input to our data

```
# dummy_files <- list.files(file.path(dir_path, "train_files_1"))
# dummy_files
#
# df <- data.frame(file = dummy_files,
#                    label_1 = c(0, 1), label_2 = c(1, 0), label_3 = c(1, 0))
# df
# write.csv(x = df, file = file.path(dir_path, "add_input.csv"), row.names = FALSE)
```

If we add the path to the csv file, the generator will map additional input to sequences:

```
# gen <- fastaFileGenerator(corpus.dir = file.path(dir_path, "train_files_1"),
#                             batch.size = 1,
#                             maxlen = 5,
#                              output_format = "target_right",
#                              added_label_path = file.path(dir_path, "add_input.csv"),
#                              add_input_as_seq = FALSE # don't treat input as sequence
# )
# z <- gen()
# added_label_input <- z[[1]][[1]]
# added_label_input
# sequence_input <- z[[1]][[2]]
# sequence_input[1, , ]
# target <- z[[2]]
# target
```

If we want to train a network with additional labels, we have to add an additional input layer.

```
# model <- create_model_lstm_cnn(
#   maxlen = 5,
#   layer_lstm = c(8, 8),
#   layer_dense = c(4),
#   label_input = 3 # additional input vector has length 3
# )
#
# trainNetwork(train_type = "lm",
#              model = model,
#              path = file.path(dir_path, "train_files_1"),
#              path.val = file.path(dir_path, "validation_files_1"),
#              added_label_path = file.path(dir_path, "add_input.csv"),
#              steps.per.epoch = 5,
#              batch.size = 8,
#              epochs = 2
# )
```

Tensorboard

We can use tensorboard to monitor our training runs. To track the runs, we have to specify a path for tensorboard files and give the run a unique name.

```
# tensorboard_path <- file.path(dir_path, "tensorboard")
# if (!dir.exists(tensorboard_path)) dir.create(tensorboard_path)
# model <- create_model_lstm_cnn()
# run.name <- "run_1"
# trainNetwork(train_type = "lm",
#              model = model,
#              path = train_path_1,
#              path.val = validation_path_1,
#              steps.per.epoch = 5,
#              batch.size = 8,
#              epochs = 10,
#              run.name = run.name,
#              tensorboard.log = tensorboard_path,
#              output = list(none = FALSE,
#                             checkpoints = FALSE,
#                             tensorboard = TRUE, # enable tensorboard
#                             log = FALSE,
#                             serialize_model = FALSE,
#                             full_model = FALSE),
#              output_format = "target_right"
# )

## open tensorboard in browser
# tensorflow::tensorboard(tensorboard_path)
```

The “SCALARS” tab displays accuracy, loss, learning rate and percentage of files seen for each epoch.

The “TEXT” tab shows the `trainNetwork` call as text.

The “HPARAM” tab tracks the hyperparameters of the different runs (maxlen, batch size etc.).

Further tensorboard documentation can be found here.

Checkpoints

We can save the architecture and weights of a model after every epoch using checkpoints. The checkpoints get stored in h5 format. The file names contain the corresponding epoch, loss and accuracy

```
# checkpoint_path <- file.path(dir_path, "checkpoints")
# if (!dir.exists(checkpoint_path)) dir.create(checkpoint_path)
# model <- create_model_lstm_cnn()
# run.name <- "run_2"
# trainNetwork(train_type = "lm",
#              model = model,
#              path = train_path_1,
#              path.val = validation_path_1,
#              steps.per.epoch = 5,
#              batch.size = 8,
#              epochs = 10,
#              run.name = run.name,
#              checkpoint_path = checkpoint_path,
#              save_best_only = TRUE, # only save model if loss improves
#              save_weights_only = FALSE, # save architecture and weights
#              output = list(none = FALSE,
#                            checkpoints = TRUE, # enable checkpoints
#                            tensorboard = FALSE,
#                            log = FALSE,
#                            serialize_model = FALSE,
#                            full_model = FALSE),
#              output_format = "target_right"
# )
```

After training, we can now load a trained model and continue training or use the model for predictions/inference.

```
# cp_run_path <- file.path(checkpoint_path, paste0(run.name, "_checkpoints"))
# checkpoints <- list.files(cp_run_path)
# checkpoints
# last_checkpoint <- checkpoints[length(checkpoints)]
```

```
#
# # load trained model and compile
# model <- keras::load_model_hdf5(file.path(cp_run_path, last_checkpoint), compile = FALSE)
# model <- keras::load_weights_model_hdf5(model, file.path(cp_run_path, last_checkpoint))
# optimizer <-  keras::optimizer_adam(lr = 0.01)
# model %>% keras::compile(loss = "categorical_crossentropy", optimizer = optimizer, metrics = c("acc")
#
# # continue training
# trainNetwork(train_type = "lm",
#              model = model,
#              path = train_path_1,
#              path.val = validation_path_1,
#              steps.per.epoch = 5,
#              batch.size = 8,
#              epochs = 2,
#              run.name = "continue_from_checkpoint",
#              output_format = "target_right"
# )
```