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 (runs input in two ways) or stateful (considers dependencies between batches).

The last dense layer layer has a softmax activation and determines how many targets we want to predict. This output gives a vector of probabilities, i.e. the sum of the vector is 1 and each entry is a probabilty for one class.

The following implementation creates a model with 3 CNN layer (+ batch normalization), 1 LSTM and 1 dense layer.

```
model <- create_model_lstm_cnn(
  maxlen = 500, # number of nucleotides processed in one sample
  layer_lstm = c(32), # number of LSTM cells
  layer_dense = c(4), # number of neurons in last layer (4 targets: A,C,G,T)
  vocabulary.size = 4, # input vocabulary has size 4 (A,C,G,T)
  kernel_size = c(12, 12, 12), # size of individual CNN windows for each layer
  filters = c(32, 64, 64), # number of CNN filters per layer
  pool_size = c(3, 3, 3) # size of max pooling per layer
)
```

```
## Model: "model"
## _________________________________________________________________________
## Layer (type)                         Output Shape                     Param #
## =========================================================================
## input_1 (InputLayer)                 [(None, 500, 4)]                 0
## _________________________________________________________________________
## conv1d (Conv1D)                      (None, 500, 32)                  1568
## _________________________________________________________________________
## max_pooling1d (MaxPooling1D)         (None, 166, 32)                  0
## _________________________________________________________________________
## batch_normalization (BatchNormaliza (None, 166, 32)                  128
## _________________________________________________________________________
## conv1d_1 (Conv1D)                    (None, 166, 64)                  24640
## _________________________________________________________________________
## batch_normalization_1 (BatchNormali (None, 166, 64)                  256
## _________________________________________________________________________
## max_pooling1d_1 (MaxPooling1D)       (None, 55, 64)                   0
## _________________________________________________________________________
## conv1d_2 (Conv1D)                    (None, 55, 64)                   49216
## _________________________________________________________________________
## batch_normalization_2 (BatchNormali (None, 55, 64)                   256
## _________________________________________________________________________
## max_pooling1d_2 (MaxPooling1D)       (None, 18, 64)                   0
## _________________________________________________________________________
## lstm (LSTM)                          (None, 32)                       12416
## _________________________________________________________________________
## dense (Dense)                        (None, 4)                        132
## =========================================================================
## Total params: 88,612
## Trainable params: 88,292
## Non-trainable params: 320
```

```
## ____________________________________________________________________________
```

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 <- 500
vocabulary.size <- 4
input <- array(rnorm(maxlen * batch_size * vocabulary.size),
               dim = c(batch_size, maxlen, vocabulary.size))
pred <- predict(model, input) # make a prediction with random data
dim(pred)
```

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

```
colnames(pred) <- c("A", "C", "G", "T")
pred # prediction for initial random weights
```

```
##              A         C         G         T
## [1,] 0.2602245 0.2536777 0.2337457 0.2523521
## [2,] 0.2549642 0.2419505 0.2287941 0.2742911
## [3,] 0.2485396 0.2442063 0.2315147 0.2757394
```

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 = 500,
  layer_lstm = c(32),
  layer_dense = c(4),
  vocabulary.size = 4,
  kernel_size = c(12, 12, 12),
  filters = c(32, 64, 64),
  pool_size = c(3, 3, 3)
)
```

```
## Model: "model_1"
## ____________________________________________________________________________
## Layer (type)              Output Shape      Param #  Connected to
## ============================================================================
## input_2 (InputLayer)      [(None, 250, 4)]  0
## ____________________________________________________________________________
## input_3 (InputLayer)      [(None, 250, 4)]  0
## ____________________________________________________________________________
## conv1d_3 (Conv1D)         (None, 250, 32)   1568     input_2[0][0]
## ____________________________________________________________________________
## conv1d_6 (Conv1D)         (None, 250, 32)   1568     input_3[0][0]
## ____________________________________________________________________________
## max_pooling1d_3 (MaxPooli (None, 83, 32)    0        conv1d_3[0][0]
```

```
## ____________________________________________________________________________________________
## max_pooling1d_6 (MaxPooli (None, 83, 32)    0         conv1d_6[0][0]
## ____________________________________________________________________________________________
## batch_normalization_3 (Ba (None, 83, 32)    128       max_pooling1d_3[0][0]
## ____________________________________________________________________________________________
## batch_normalization_6 (Ba (None, 83, 32)    128       max_pooling1d_6[0][0]
## ____________________________________________________________________________________________
## conv1d_4 (Conv1D)         (None, 83, 64)    24640     batch_normalization_3[0][0]
## ____________________________________________________________________________________________
## conv1d_7 (Conv1D)         (None, 83, 64)    24640     batch_normalization_6[0][0]
## ____________________________________________________________________________________________
## max_pooling1d_4 (MaxPooli (None, 27, 64)    0         conv1d_4[0][0]
## ____________________________________________________________________________________________
## max_pooling1d_7 (MaxPooli (None, 27, 64)    0         conv1d_7[0][0]
## ____________________________________________________________________________________________
## batch_normalization_4 (Ba (None, 27, 64)    256       max_pooling1d_4[0][0]
## ____________________________________________________________________________________________
## batch_normalization_7 (Ba (None, 27, 64)    256       max_pooling1d_7[0][0]
## ____________________________________________________________________________________________
## conv1d_5 (Conv1D)         (None, 27, 64)    49216     batch_normalization_4[0][0]
## ____________________________________________________________________________________________
## conv1d_8 (Conv1D)         (None, 27, 64)    49216     batch_normalization_7[0][0]
## ____________________________________________________________________________________________
## max_pooling1d_5 (MaxPooli (None, 9, 64)     0         conv1d_5[0][0]
## ____________________________________________________________________________________________
## max_pooling1d_8 (MaxPooli (None, 9, 64)     0         conv1d_8[0][0]
## ____________________________________________________________________________________________
## batch_normalization_5 (Ba (None, 9, 64)     256       max_pooling1d_5[0][0]
## ____________________________________________________________________________________________
## batch_normalization_8 (Ba (None, 9, 64)     256       max_pooling1d_8[0][0]
## ____________________________________________________________________________________________
## lstm_1 (LSTM)             (None, 32)        12416     batch_normalization_5[0][0]
## ____________________________________________________________________________________________
## lstm_2 (LSTM)             (None, 32)        12416     batch_normalization_8[0][0]
## ____________________________________________________________________________________________
## concatenate (Concatenate) (None, 64)        0         lstm_1[0][0]
##                                                       lstm_2[0][0]
## ____________________________________________________________________________________________
## dense_1 (Dense)           (None, 4)         260       concatenate[0][0]
## ============================================================================================
## Total params: 177,220
## Trainable params: 176,580
## Non-trainable params: 640
## ____________________________________________________________________________________________
```

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.

create_model_wavenet

This model uses causal dilated convolution layers, which is suitable to handle long sequences. The original paper can be found here

```
model <- create_model_wavenet(filters = 16, kernel_size = 2, residual_blocks = 2^(2:4),
                              maxlen = 500, input_tensor = NULL, initial_kernel_size = 32,
                              initial_filters = 32, output_channels = 4,
                              output_activation = "softmax", solver = "adam",
                              learning.rate = 0.001, compile = TRUE)
model
```

```
## Model
## Model: "model_2"
## ____________________________________________________________________________
## Layer (type)              Output Shape      Param #  Connected to
## ============================================================================
## input_4 (InputLayer)      [(None, 500, 4)]  0
## ____________________________________________________________________________
## conv1d_9 (Conv1D)         (None, 500, 32)   4096     input_4[0][0]
## ____________________________________________________________________________
## r_layer (RLayer)          [(None, 500, 32), 0        conv1d_9[0][0]
## ____________________________________________________________________________
## r_layer_1 (RLayer)        [(None, 500, 32), 0        r_layer[0][0]
## ____________________________________________________________________________
## r_layer_2 (RLayer)        [(None, 500, 32), 0        r_layer_1[0][0]
## ____________________________________________________________________________
## add (Add)                 (None, 500, 32)   0        r_layer[0][1]
##                                                      r_layer_1[0][1]
##                                                      r_layer_2[0][1]
## ____________________________________________________________________________
## activation (Activation)   (None, 500, 32)   0        add[0][0]
## ____________________________________________________________________________
## conv1d_11 (Conv1D)        (None, 500, 16)   512      activation[0][0]
## ____________________________________________________________________________
## conv1d_10 (Conv1D)        (None, 500, 4)    68       conv1d_11[0][0]
## ============================================================================
## Total params: 4,676
## Trainable params: 4,676
## Non-trainable params: 0
## ____________________________________________________________________________
```

The model expects an input and output of dimension (batch size, maxlen, vocabulary.size). The target sequence should be equal to input sequence shifted by one position. For example, given a sequence `ACCGGTC` and maxlen = 6, the input should correspond to `ACCGGT` and target to `CCGGTC`.

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.

```
path <- "/home/rmreches/tutorial"
path_16S_train <- file.path(path, "16s/train")
path_16S_validation <- file.path(path, "16s/validation")
path_bacteria_train <- file.path(path, "bacteria/train")
path_bacteria_validation <- file.path(path, "bacteria/validation")

checkpoint_path <- file.path(path, "checkpoints")
tensorboard.log <- file.path(path, "tensorboard")
dir_path <- file.path(path, "outputs")
if (!dir.exists(checkpoint_path)) dir.create(checkpoint_path)
if (!dir.exists(tensorboard.log)) dir.create(tensorboard.log)
if (!dir.exists(dir_path)) dir.create(dir_path)
```

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 1 dense layer for predictions.

```
model <- create_model_lstm_cnn(
  maxlen = 500,
  layer_lstm = c(32),
  layer_dense = c(4),
  vocabulary.size = 4,
  kernel_size = c(12, 12, 12),
  filters = c(32, 64, 64),
  pool_size = c(3, 3, 3),
  learning.rate = 0.001
)
```

```
## Model: "model_3"
## ________________________________________________________________________________
```

```
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## input_5 (InputLayer)                [(None, 500, 4)]                0
## ________________________________________________________________________________
## conv1d_12 (Conv1D)                  (None, 500, 32)                 1568
## ________________________________________________________________________________
## max_pooling1d_9 (MaxPooling1D)      (None, 166, 32)                 0
## ________________________________________________________________________________
## batch_normalization_9 (BatchNormali (None, 166, 32)                 128
## ________________________________________________________________________________
## conv1d_13 (Conv1D)                  (None, 166, 64)                 24640
## ________________________________________________________________________________
## batch_normalization_10 (BatchNormal (None, 166, 64)                 256
## ________________________________________________________________________________
## max_pooling1d_10 (MaxPooling1D)     (None, 55, 64)                  0
## ________________________________________________________________________________
## conv1d_14 (Conv1D)                  (None, 55, 64)                  49216
## ________________________________________________________________________________
## batch_normalization_11 (BatchNormal (None, 55, 64)                  256
## ________________________________________________________________________________
## max_pooling1d_11 (MaxPooling1D)     (None, 18, 64)                  0
## ________________________________________________________________________________
## lstm_3 (LSTM)                       (None, 32)                      12416
## ________________________________________________________________________________
## dense_2 (Dense)                     (None, 4)                       132
## ================================================================================
## Total params: 88,612
## Trainable params: 88,292
## Non-trainable params: 320
## ________________________________________________________________________________
```

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", # train a language model
             model = model,
             path = path_16S_train, # location of trainig data
             path.val = path_16S_validation, # location of validation data
             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 = 4,
             steps.per.epoch = 10, # 1 epoch = 10 batches
             step = 500, # take a sample every 500 steps
             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 10 samples (batch_size=NULL, epochs=4)
## Final epoch (plot to see history):
##     loss: 0.08382
##      acc: 0.9937
##       f1: Inf
## val_loss: 0.5738
##  val_acc: 0.8125
##   val_f1: Inf
##       lr: 0.001
```

```
tensorflow::tensorboard(tensorboard.log)
```

```
## Started TensorBoard at http://127.0.0.1:4527
```

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 = 500,
  layer_lstm = c(32),
  layer_dense = c(4),
  vocabulary.size = 4,
  kernel_size = c(12, 12, 12),
  filters = c(32, 64, 64),
  pool_size = c(3, 3, 3),
  learning.rate = 0.001
)
```

```
## Model: "model_4"
## ____________________________________________________________________________
## Layer (type)              Output Shape      Param #  Connected to
## ============================================================================
## input_6 (InputLayer)      [(None, 250, 4)]  0
## ____________________________________________________________________________
## input_7 (InputLayer)      [(None, 250, 4)]  0
## ____________________________________________________________________________
## conv1d_15 (Conv1D)        (None, 250, 32)   1568     input_6[0][0]
## ____________________________________________________________________________
## conv1d_18 (Conv1D)        (None, 250, 32)   1568     input_7[0][0]
## ____________________________________________________________________________
## max_pooling1d_12 (MaxPool (None, 83, 32)    0        conv1d_15[0][0]
## ____________________________________________________________________________
```

```
## max_pooling1d_15 (MaxPool (None, 83, 32)    0        conv1d_18[0][0]
## ____________________________________________________________________________
## batch_normalization_12 (B (None, 83, 32)    128      max_pooling1d_12[0][0]
## ____________________________________________________________________________
## batch_normalization_15 (B (None, 83, 32)    128      max_pooling1d_15[0][0]
## ____________________________________________________________________________
## conv1d_16 (Conv1D)        (None, 83, 64)    24640    batch_normalization_12[0][0
## ____________________________________________________________________________
## conv1d_19 (Conv1D)        (None, 83, 64)    24640    batch_normalization_15[0][0
## ____________________________________________________________________________
## max_pooling1d_13 (MaxPool (None, 27, 64)    0        conv1d_16[0][0]
## ____________________________________________________________________________
## max_pooling1d_16 (MaxPool (None, 27, 64)    0        conv1d_19[0][0]
## ____________________________________________________________________________
## batch_normalization_13 (B (None, 27, 64)    256      max_pooling1d_13[0][0]
## ____________________________________________________________________________
## batch_normalization_16 (B (None, 27, 64)    256      max_pooling1d_16[0][0]
## ____________________________________________________________________________
## conv1d_17 (Conv1D)        (None, 27, 64)    49216    batch_normalization_13[0][0
## ____________________________________________________________________________
## conv1d_20 (Conv1D)        (None, 27, 64)    49216    batch_normalization_16[0][0
## ____________________________________________________________________________
## max_pooling1d_14 (MaxPool (None, 9, 64)     0        conv1d_17[0][0]
## ____________________________________________________________________________
## max_pooling1d_17 (MaxPool (None, 9, 64)     0        conv1d_20[0][0]
## ____________________________________________________________________________
## batch_normalization_14 (B (None, 9, 64)     256      max_pooling1d_14[0][0]
## ____________________________________________________________________________
## batch_normalization_17 (B (None, 9, 64)     256      max_pooling1d_17[0][0]
## ____________________________________________________________________________
## lstm_4 (LSTM)             (None, 32)        12416    batch_normalization_14[0][0
## ____________________________________________________________________________
## lstm_5 (LSTM)             (None, 32)        12416    batch_normalization_17[0][0
## ____________________________________________________________________________
## concatenate_1 (Concatenat (None, 64)        0        lstm_4[0][0]
##                                                      lstm_5[0][0]
## ____________________________________________________________________________
## dense_3 (Dense)           (None, 4)         260      concatenate_1[0][0]
## ============================================================================
## Total params: 177,220
## Trainable params: 176,580
## Non-trainable params: 640
## ____________________________________________________________________________
```

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", # train a language model
             model = model,
             path = path_16S_train, # location of trainig data
             path.val = path_16S_validation, # location of validation data
             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_middle_lstm",
            batch.size = 256,
            epochs = 4,
            steps.per.epoch = 10, # 1 epoch = 10 batches
            step = 500, # take a sample every 500 steps
            output = list(none = FALSE,
                          checkpoints = TRUE,
                          tensorboard = TRUE,
                          log = FALSE,
                          serialize_model = FALSE,
                          full_model = FALSE
            ),
            tb_images = TRUE,
            output_format = "target_middle_lstm" # predict character in middle of sequence
)
```

```
## Trained on 10 samples (batch_size=NULL, epochs=4)
## Final epoch (plot to see history):
##     loss: 0.04662
##      acc: 0.9984
##       f1: Inf
## val_loss: 0.3665
##  val_acc: 0.8633
##   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. In the following example, we want to classify if a sequence belongs to 16s or bacterial genome. We have to put all 16s/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 = 500,
  layer_lstm = c(32),
  layer_dense = c(2), # predict two classes
  vocabulary.size = 4,
  kernel_size = c(12, 12, 12),
  filters = c(32, 64, 64),
  pool_size = c(3, 3, 3),
```

```
  learning.rate = 0.001
)
```

```
## Model: "model_5"
## _________________________________________________________________________
## Layer (type)                    Output Shape                  Param #
## =========================================================================
## input_8 (InputLayer)            [(None, 500, 4)]              0
## _________________________________________________________________________
## conv1d_21 (Conv1D)              (None, 500, 32)               1568
## _________________________________________________________________________
## max_pooling1d_18 (MaxPooling1D) (None, 166, 32)               0
## _________________________________________________________________________
## batch_normalization_18 (BatchNormal (None, 166, 32)           128
## _________________________________________________________________________
## conv1d_22 (Conv1D)              (None, 166, 64)               24640
## _________________________________________________________________________
## batch_normalization_19 (BatchNormal (None, 166, 64)           256
## _________________________________________________________________________
## max_pooling1d_19 (MaxPooling1D) (None, 55, 64)                0
## _________________________________________________________________________
## conv1d_23 (Conv1D)              (None, 55, 64)                49216
## _________________________________________________________________________
## batch_normalization_20 (BatchNormal (None, 55, 64)            256
## _________________________________________________________________________
## max_pooling1d_20 (MaxPooling1D) (None, 18, 64)                0
## _________________________________________________________________________
## lstm_6 (LSTM)                   (None, 32)                    12416
## _________________________________________________________________________
## dense_4 (Dense)                 (None, 2)                     66
## =========================================================================
## Total params: 88,546
## Trainable params: 88,226
## Non-trainable params: 320
## _________________________________________________________________________
```

```
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,
             validation.split = 0.2,
             run.name = "16S_vs_bacteria",
             batch.size = 512, # half of batch is 16s and other half bacteria data
             epochs = 5,
             steps.per.epoch = 15,
             step = 500,
             labelVocabulary = c("16s", "bacteria"), # label names
             output = list(none = FALSE,
                           checkpoints = TRUE,
```

```
                            tensorboard = TRUE,
                            log = FALSE,
                            serialize_model = FALSE,
                            full_model = FALSE
                ),
                tb_images = TRUE,
                proportion_per_file = c(1, 0.05) # randomly select 5% of bacteria file
)
```

```
## Trained on 15 samples (batch_size=NULL, epochs=5)
## Final epoch (plot to see history):
##     loss: 0.004327
##      acc: 0.9992
##       f1: 0.9992
## val_loss: 0.005792
##  val_acc: 0.9987
##   val_f1: 0.9987
##       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. In the following example we use the binary model trained to classify 16S/bacteria data.

```
print(model)
```

```
## Model
## Model: "model_5"
## _________________________________________________________________________
## Layer (type)                    Output Shape                  Param #
## =========================================================================
## input_8 (InputLayer)            [(None, 500, 4)]              0
## _________________________________________________________________________
## conv1d_21 (Conv1D)              (None, 500, 32)               1568
## _________________________________________________________________________
## max_pooling1d_18 (MaxPooling1D) (None, 166, 32)               0
## _________________________________________________________________________
## batch_normalization_18 (BatchNormal (None, 166, 32)           128
## _________________________________________________________________________
## conv1d_22 (Conv1D)              (None, 166, 64)               24640
## _________________________________________________________________________
## batch_normalization_19 (BatchNormal (None, 166, 64)           256
## _________________________________________________________________________
## max_pooling1d_19 (MaxPooling1D) (None, 55, 64)                0
## _________________________________________________________________________
## conv1d_23 (Conv1D)              (None, 55, 64)                49216
## _________________________________________________________________________
## batch_normalization_20 (BatchNormal (None, 55, 64)            256
## _________________________________________________________________________
## max_pooling1d_20 (MaxPooling1D) (None, 18, 64)                0
## _________________________________________________________________________
```

```
## lstm_6 (LSTM)                       (None, 32)                      12416
## ________________________________________________________________________________
## dense_4 (Dense)                     (None, 2)                       66
## ================================================================================
## Total params: 88,546
## Trainable params: 88,226
## Non-trainable params: 320
## ________________________________________________________________________________
```

```
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_4
```

```
fasta.path <- list.files(path_16S_validation, full.names = TRUE)[1] # make predictions for 16S file
fasta.file <- microseq::readFasta(fasta.path)
head(fasta.file)
```

```
## # A tibble: 1 x 2
##   Header                      Sequence
##   <chr>                       <chr>
## 1 16S_rRNA::CP015410.2:15033~ TATGAGAGTTTGATCCTGGCTCAGGACGAACGCTGGCGGCGTGCCTAAT~
```

```
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")
}
```

```
## Computing output for model at layer dense_4
## Model
## Model: "model_6"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## input_8 (InputLayer)                [(None, 500, 4)]                0
## ________________________________________________________________________________
## conv1d_21 (Conv1D)                  (None, 500, 32)                 1568
## ________________________________________________________________________________
## max_pooling1d_18 (MaxPooling1D)     (None, 166, 32)                 0
## ________________________________________________________________________________
## batch_normalization_18 (BatchNormal (None, 166, 32)                 128
## ________________________________________________________________________________
```

```
## conv1d_22 (Conv1D)                  (None, 166, 64)                24640
## ____________________________________________________________________________
## batch_normalization_19 (BatchNormal (None, 166, 64)                256
## ____________________________________________________________________________
## max_pooling1d_19 (MaxPooling1D)     (None, 55, 64)                 0
## ____________________________________________________________________________
## conv1d_23 (Conv1D)                  (None, 55, 64)                 49216
## ____________________________________________________________________________
## batch_normalization_20 (BatchNormal (None, 55, 64)                 256
## ____________________________________________________________________________
## max_pooling1d_20 (MaxPooling1D)     (None, 18, 64)                 0
## ____________________________________________________________________________
## lstm_6 (LSTM)                       (None, 32)                     12416
## ____________________________________________________________________________
## dense_4 (Dense)                     (None, 2)                      66
## ============================================================================
## Total params: 88,546
## Trainable params: 88,226
## Non-trainable params: 320
## ____________________________________________________________________________
```

We can access the h5 file as follows

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

```
## states matrix has 1058 rows and  2 columns
```

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

```
##         16S bacteria
## [1,] 0.9988   0.0012
## [2,] 0.9983   0.0017
## [3,] 0.9965   0.0035
## [4,] 0.9968   0.0032
## [5,] 0.9960   0.0040
## [6,] 0.9966   0.0034
```

The matrix shows the models confidence in its predictions. Every row corresponds to one sample. If the value in the 16s column is > 0.500, the model will classify the sample as 16s.

Detect 16S region

We can use or trained model to detect 16S sequences in a bacterial genome. First, we search for the true rRNA region in the corresponding gff file.

```
fasta.path<- file.path(path, "E_faecalis.fasta")
gff.file <- file.path(path, "E_faecalis.gff")
gff.data <- rtracklayer::readGFF(gff.file, version = 0,
                                 columns = NULL, tags = NULL, filter = NULL, nrows = -1,
                                 raw_data = FALSE)
```

```
rRNA_index <- stringr::str_detect(gff.data$product, "^16S ribosomal") & (gff.data$strand == "+")
start <- gff.data[rRNA_index, "start"]
end <- gff.data[rRNA_index, "end"]
start; end
```

```
## [1] 2189670 2933745
```

```
## [1] 2191227 2935302
```

We iterate over the bacteria file and make a predictions every 100 steps

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

if (!file.exists(filename)) {
  writeStates(
    model = model,
    layer_name = layer_name,
    sequence = sequence,
    round_digits = 4,
    filename = filename,
    batch.size = 500,
    step = 100)
}
```

```
## Computing output for model at layer dense_4
## Model
## Model: "model_7"
## _________________________________________________________________________
## Layer (type)                    Output Shape                  Param #
## =========================================================================
## input_8 (InputLayer)            [(None, 500, 4)]              0
## _________________________________________________________________________
## conv1d_21 (Conv1D)              (None, 500, 32)               1568
## _________________________________________________________________________
## max_pooling1d_18 (MaxPooling1D) (None, 166, 32)               0
## _________________________________________________________________________
## batch_normalization_18 (BatchNormal (None, 166, 32)           128
## _________________________________________________________________________
## conv1d_22 (Conv1D)              (None, 166, 64)               24640
## _________________________________________________________________________
## batch_normalization_19 (BatchNormal (None, 166, 64)           256
## _________________________________________________________________________
## max_pooling1d_19 (MaxPooling1D) (None, 55, 64)                0
## _________________________________________________________________________
## conv1d_23 (Conv1D)              (None, 55, 64)                49216
## _________________________________________________________________________
## batch_normalization_20 (BatchNormal (None, 55, 64)            256
## _________________________________________________________________________
## max_pooling1d_20 (MaxPooling1D) (None, 18, 64)                0
## _________________________________________________________________________
```

```
## lstm_6 (LSTM)                       (None, 32)                        12416
## _________________________________________________________________________
## dense_4 (Dense)                     (None, 2)                         66
## =========================================================================
## Total params: 88,546
## Trainable params: 88,226
## Non-trainable params: 320
## _________________________________________________________________________
```

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

```
## states matrix has 30252 rows and  2 columns
```

```
pred <- states[[1]]
position <- states[[2]] - 1
df <- cbind(pred, position) %>% as.data.frame()
colnames(df) <- c("conf_16S", "conf_bacteria", "seq_end")
head(df)
```

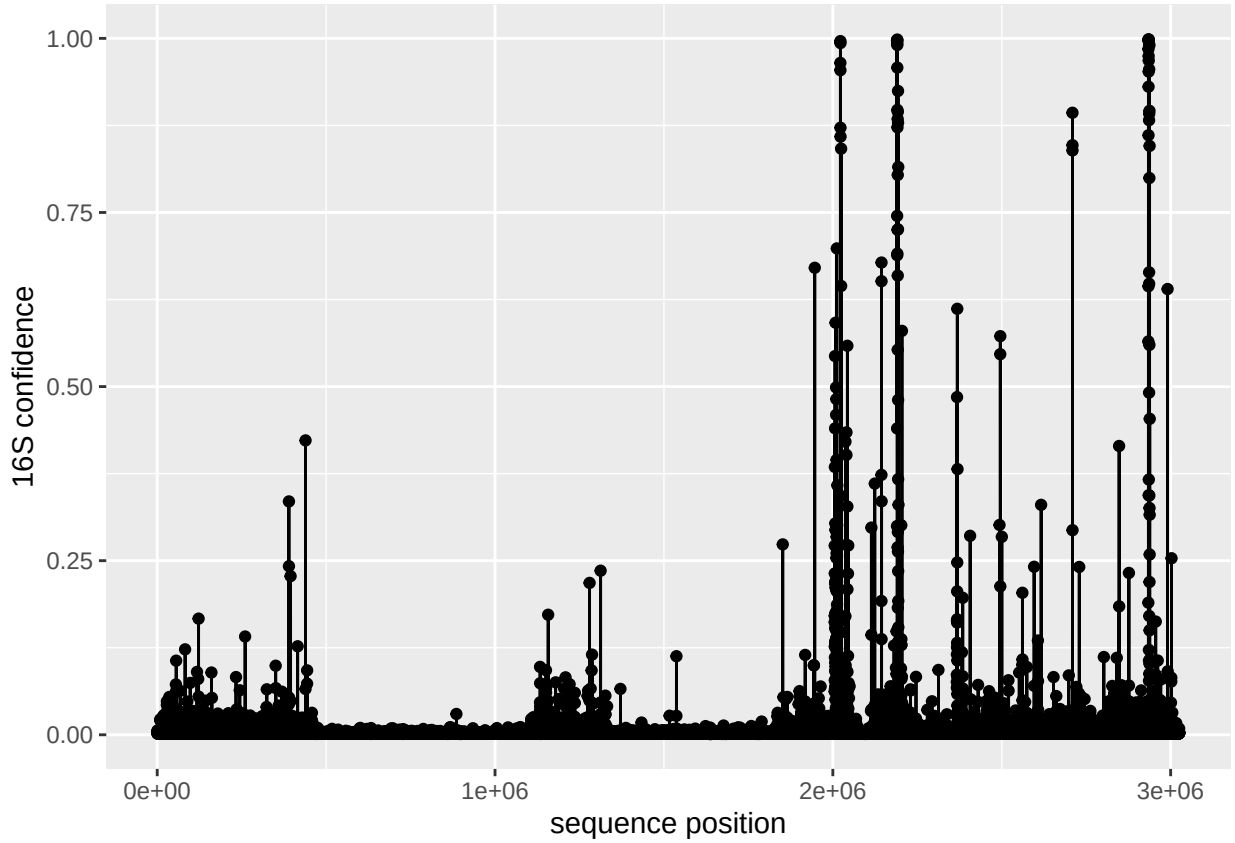
```
##   conf_16S conf_bacteria seq_end
## 1   0.0059        0.9941     500
## 2   0.0032        0.9968     600
## 3   0.0030        0.9970     700
## 4   0.0048        0.9952     800
## 5   0.0025        0.9975     900
## 6   0.0017        0.9983    1000
```

```
index_16S_pred <- df[ , 1] > 0.5
df_16S <- df[index_16S_pred, ]
head(df_16S)
```

```
##       conf_16S conf_bacteria seq_end
## 19461   0.6706        0.3294 1946500
## 20061   0.5438        0.4562 2006500
## 20076   0.5917        0.4083 2008000
## 20114   0.6984        0.3016 2011800
## 20219   0.9648        0.0352 2022300
## 20220   0.9961        0.0039 2022400
```

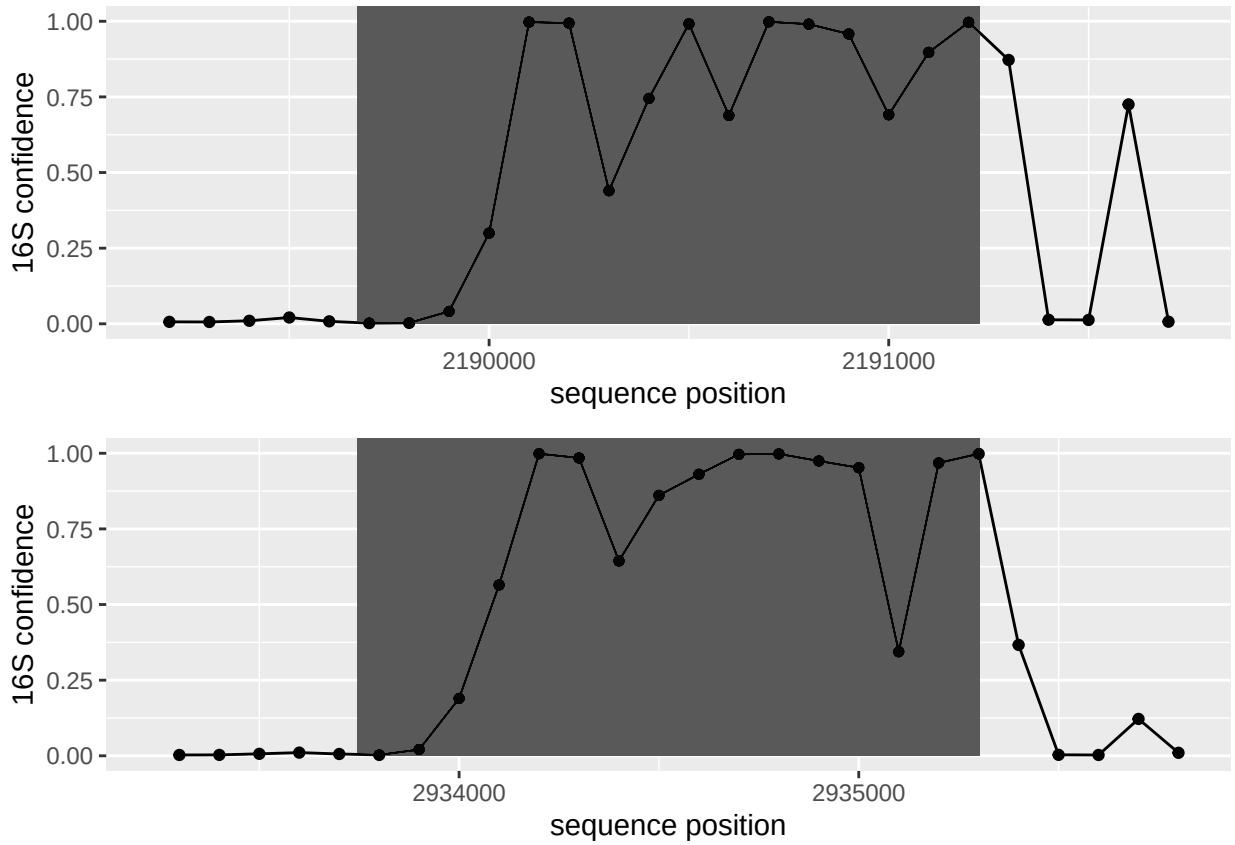
Let's visualize or models predictions and compare them to the true areas. First we look at the confidence in 16S over the whole genome.

```
ggplot(df, aes(x = seq_end, y = conf_16S)) + geom_point() + geom_line() + ylab("16S confidence") + xlab
```



Next we may zoom into areas with high 16S confidence, the true 16S regions are shaded grey.

```
p1 <- ggplot(df, aes(x = seq_end, y = conf_16S)) + geom_point() + geom_line() +
  geom_rect(aes(xmin=start[1], xmax=end[1], ymin=0, ymax=Inf),  alpha = 0.01) +
  xlim(c(start[1] - 500, end[1] + 500)) +
  ylab("16S confidence") + xlab("sequence position")
p2 <- ggplot(df, aes(x = seq_end, y = conf_16S)) + geom_point() + geom_line() +
  geom_rect(aes(xmin=start[2], xmax=end[2], ymin=0, ymax=Inf),  alpha = 0.01) +
  xlim(c(start[2] - 500, end[2] + 500)) +
  ylab("16S confidence") + xlab("sequence position")
ggpubr::ggarrange(p1, p2, ncol = 1, nrow = 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.

```
# trainNetwork(run.name = "unique_run_name",
#              tensorboard.log = "tensorboard_path",
#              ...
# )
```

We can inspect out previous training runs in tensorboard

```
## open tensorboard in browser
# tensorflow::tensorboard(tensorboard.log)
```

The "SCALARS" tab displays accuracy,

loss

and percentage of files seen for each epoch

In the "IMAGES" tab, we implemented a display of train and validation confusion matrices after every epoch. We can see for our binary classification of bacteria/16S sequences, that the model misclassifies more bacteria sequences as 16S than vice versa.

The "TEXT" tab shows the `trainNetwork` call as text.

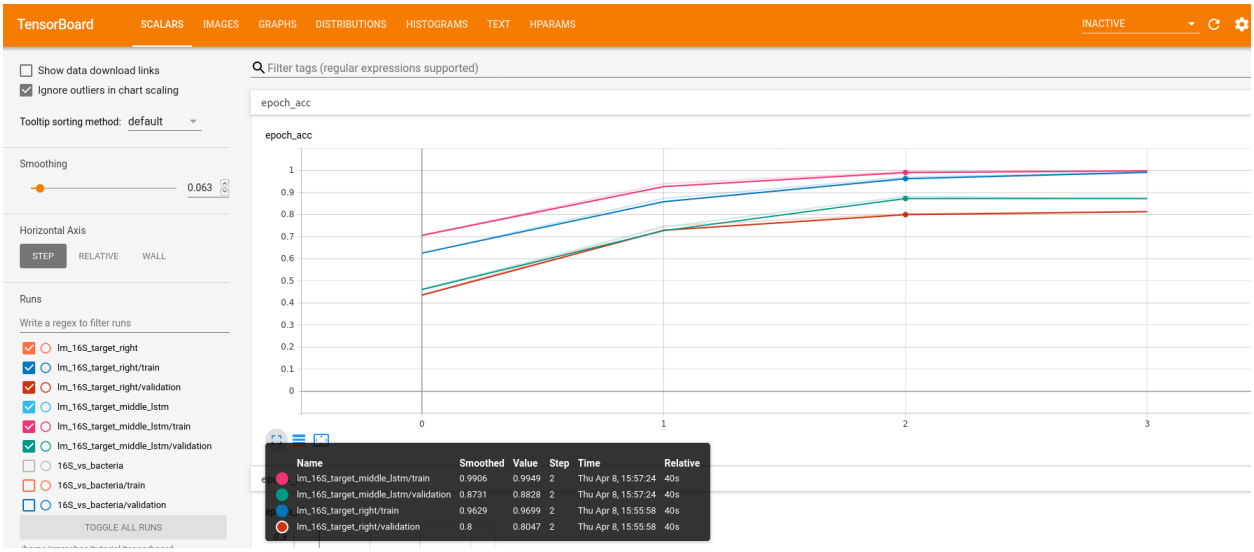


Figure 1: accuracy

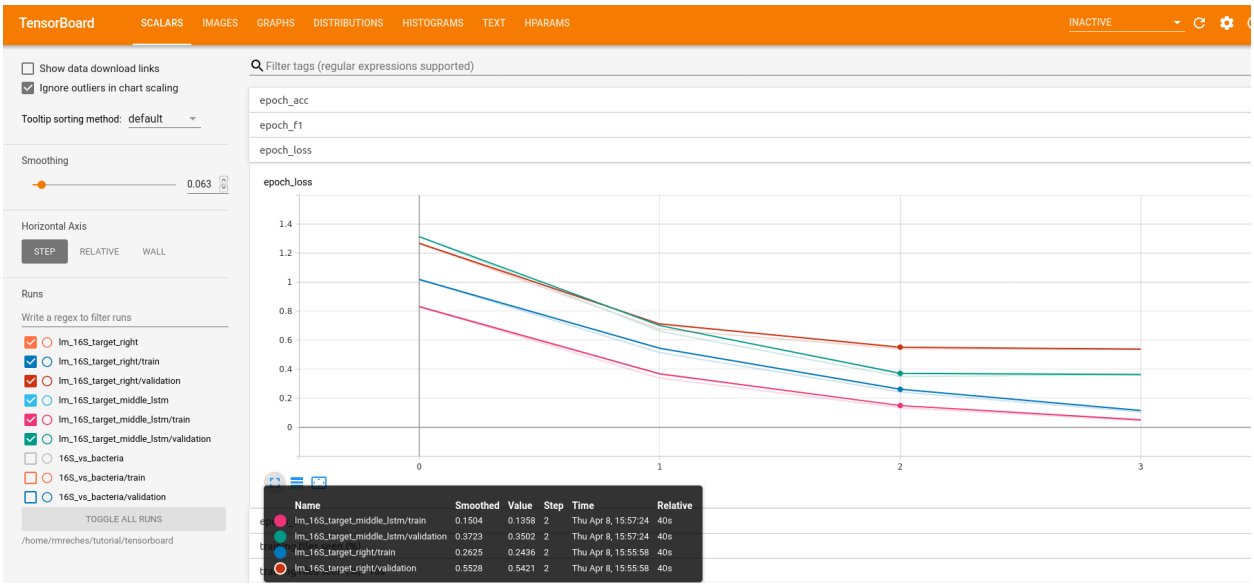


Figure 2: loss

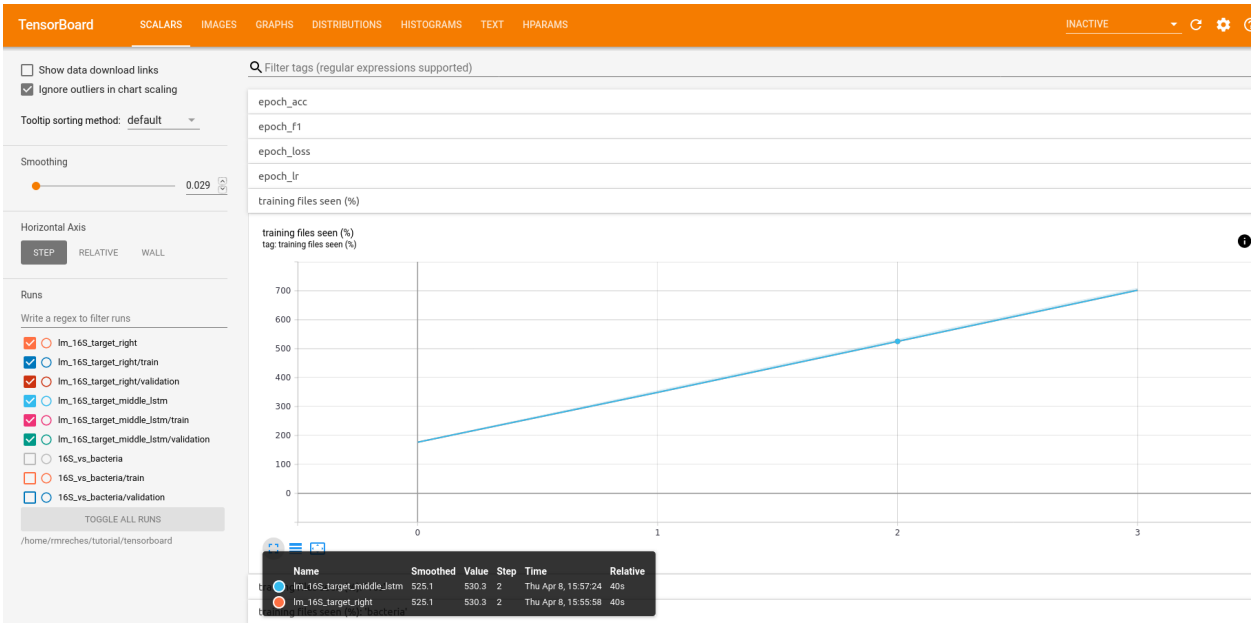


Figure 3: percentage of seen training files

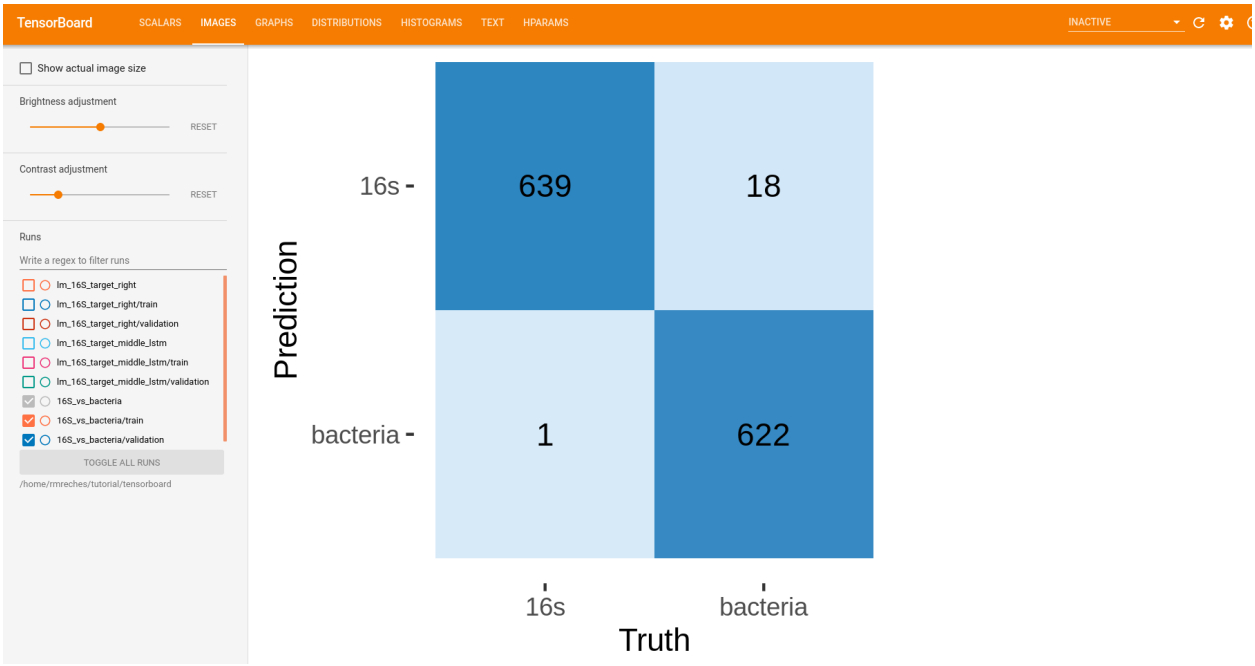


Figure 4: confusion matrix

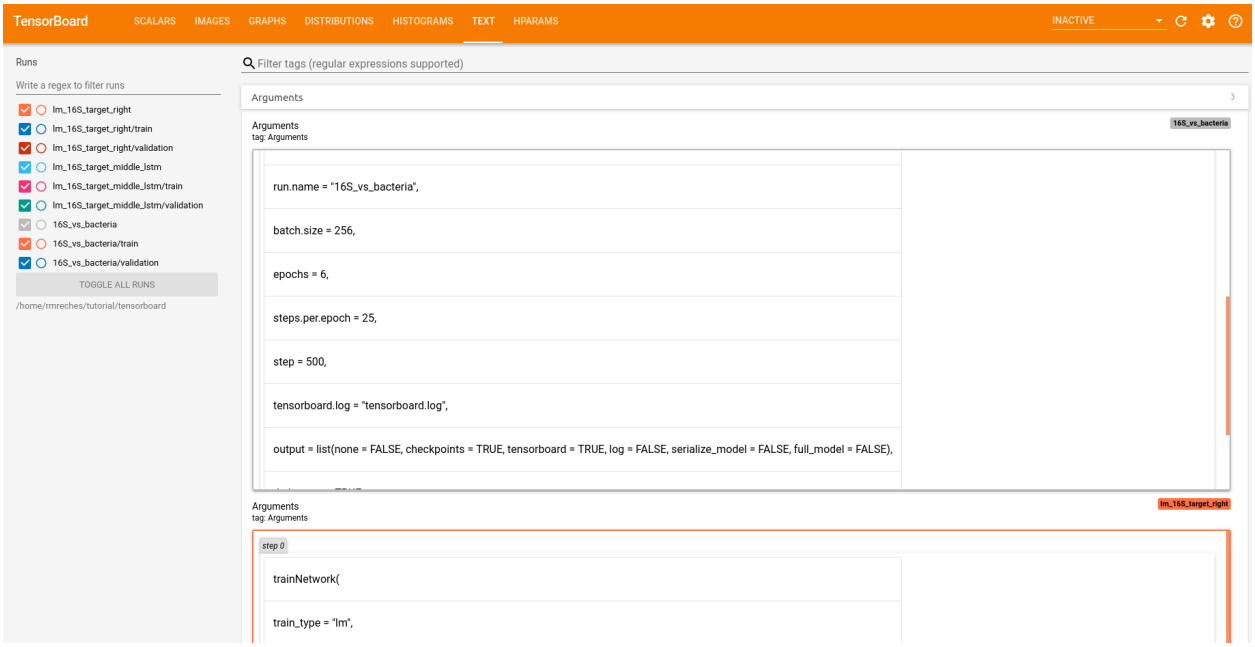


Figure 5: trainNetwork call

The "HPARAM" tab tracks the hyperparameters of the different runs (maxlen, batch size etc.). This can be used to find the hyperparameter settings for a given task

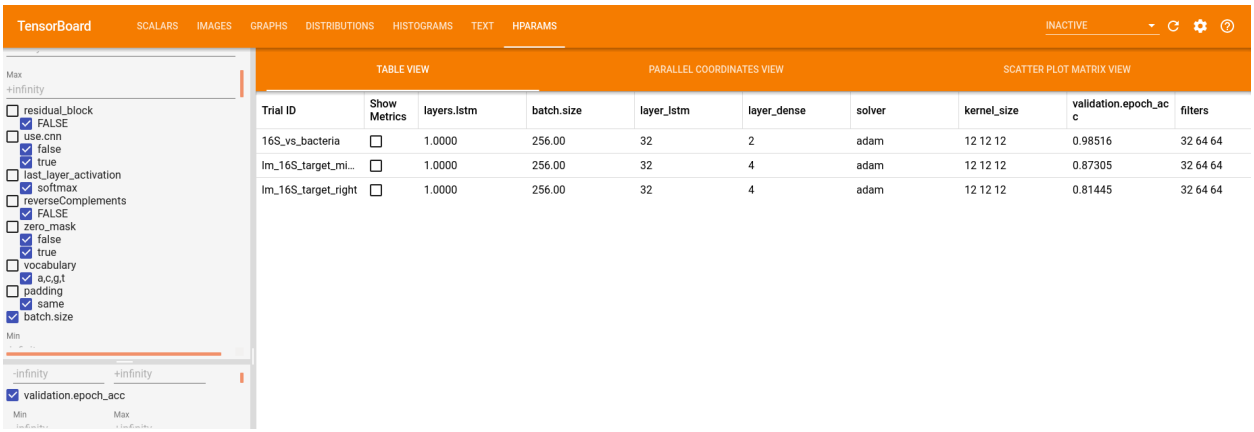


Figure 6: hyperparameters

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. For example, we can display the checkpoints from binary classification model for 16S/bacteria.

```r
cp <- list.files(file.path(checkpoint_path, "16S_vs_bacteria_checkpoints"), full.names = TRUE)
print(basename(cp))
```

```
## [1] "Ep.001-val_loss0.44-val_acc0.794.hdf5"
## [2] "Ep.002-val_loss0.10-val_acc0.959.hdf5"
## [3] "Ep.003-val_loss0.04-val_acc0.990.hdf5"
## [4] "Ep.004-val_loss0.01-val_acc0.999.hdf5"
## [5] "Ep.005-val_loss0.01-val_acc0.999.hdf5"
```

After training, we can load a trained model and continue training or use the model for predictions/inference. Let's create a model with random weights identical to our 16S/bacteria classifier and make some predictions.

```r
model <- create_model_lstm_cnn(
  maxlen = 500,
  layer_lstm = c(32),
  layer_dense = c(2),
  vocabulary.size = 4,
  kernel_size = c(12, 12, 12),
  filters = c(32, 64, 64),
  pool_size = c(3, 3, 3),
  learning.rate = 0.001
)
```

```
## Model: "model_8"
## _________________________________________________________________________
## Layer (type)                         Output Shape                     Param #
## =========================================================================
## input_9 (InputLayer)                 [(None, 500, 4)]                 0
## _________________________________________________________________________
## conv1d_24 (Conv1D)                   (None, 500, 32)                  1568
## _________________________________________________________________________
## max_pooling1d_21 (MaxPooling1D)      (None, 166, 32)                  0
## _________________________________________________________________________
## batch_normalization_21 (BatchNormal (None, 166, 32)                  128
## _________________________________________________________________________
## conv1d_25 (Conv1D)                   (None, 166, 64)                  24640
## _________________________________________________________________________
## batch_normalization_22 (BatchNormal (None, 166, 64)                  256
## _________________________________________________________________________
## max_pooling1d_22 (MaxPooling1D)      (None, 55, 64)                   0
## _________________________________________________________________________
## conv1d_26 (Conv1D)                   (None, 55, 64)                   49216
## _________________________________________________________________________
## batch_normalization_23 (BatchNormal (None, 55, 64)                   256
## _________________________________________________________________________
## max_pooling1d_23 (MaxPooling1D)      (None, 18, 64)                   0
## _________________________________________________________________________
## lstm_7 (LSTM)                        (None, 32)                       12416
## _________________________________________________________________________
## dense_5 (Dense)                      (None, 2)                        66
## =========================================================================
## Total params: 88,546
```

```
## Trainable params: 88,226
## Non-trainable params: 320
## _________________________________________________________________________________
```

```
eval_model <- evaluateFasta(fasta.path = c(path_16S_validation,
                                           path_bacteria_validation),
              model = model,
              batch.size = 100,
              step = 100,
              label_vocabulary = c("16s", "bacteria"),
              numberOfBatches = 10,
              mode = "label_folder")
```

```
## Progress:  10 %
## Progress:  20 %
## Progress:  30 %
## Progress:  40 %
## Progress:  50 %
## Progress:  60 %
## Progress:  70 %
## Progress:  80 %
## Evaluation will take approximately 0.0007769005 hours
## Progress:  90 %
## Progress:  100 %
```

```
eval_model[["accuracy"]]
```

```
## [1] 0.509
```

```
eval_model[["confusion_matrix"]]
```

```
##           Truth
## Prediction 16s bacteria
##   16s      484      475
##   bacteria  16       25
```

As expected, the performance is not better than random guessing. Let's repeat evaluation but load the weights of our pretrained model

```
weight_path <- cp[length(cp)]
model <- keras::load_model_weights_hdf5(model, weight_path)

eval_model <- evaluateFasta(fasta.path = c(path_16S_validation,
                                           path_bacteria_validation),
              model = model,
              batch.size = 100,
              step = 100,
              label_vocabulary = c("16s", "bacteria"),
              numberOfBatches = 10,
              mode = "label_folder")
```

```
## Progress:  10 %
## Progress:  20 %
## Progress:  30 %
## Progress:  40 %
## Progress:  50 %
## Progress:  60 %
## Progress:  70 %
## Progress:  80 %
## Evaluation will take approximately 0.0006256471 hours
## Progress:  90 %
## Progress:  100 %
```

```
eval_model[["accuracy"]]
```

```
## [1] 0.967
```

```
eval_model[["confusion_matrix"]]
```

```
##            Truth
## Prediction 16s bacteria
##   16s      467        0
##   bacteria  33      500
```

Integrated gradient