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 preprocessing (data generator), network training, inference and visualizing feature importance (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 $n \geq 1$.
LSTM (long short-term memory) layers are specifically designed to processes sequential data (order of data is important) by using feedback connections. CNN (convolutional neural network) layers are usually applied to images or audio data but can also be used for natural language processing or genomic sequences. Contrary to vanilla feedforward networks, they are able to process spatial relations in the data.
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 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 probability for one class.

The following implementation creates a model with 3 CNN layer (+ batch normalization and max pooling), 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 of dimensions (NULL (batch size), maxlen, vocabulary size) and a target of 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,] 0.3061384 0.2042683 0.2196189 0.2699745
## [2,] 0.3181809 0.1918129 0.2106417 0.2793645
## [3,] 0.3189375 0.1903433 0.2187358 0.2719834
```

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]
```

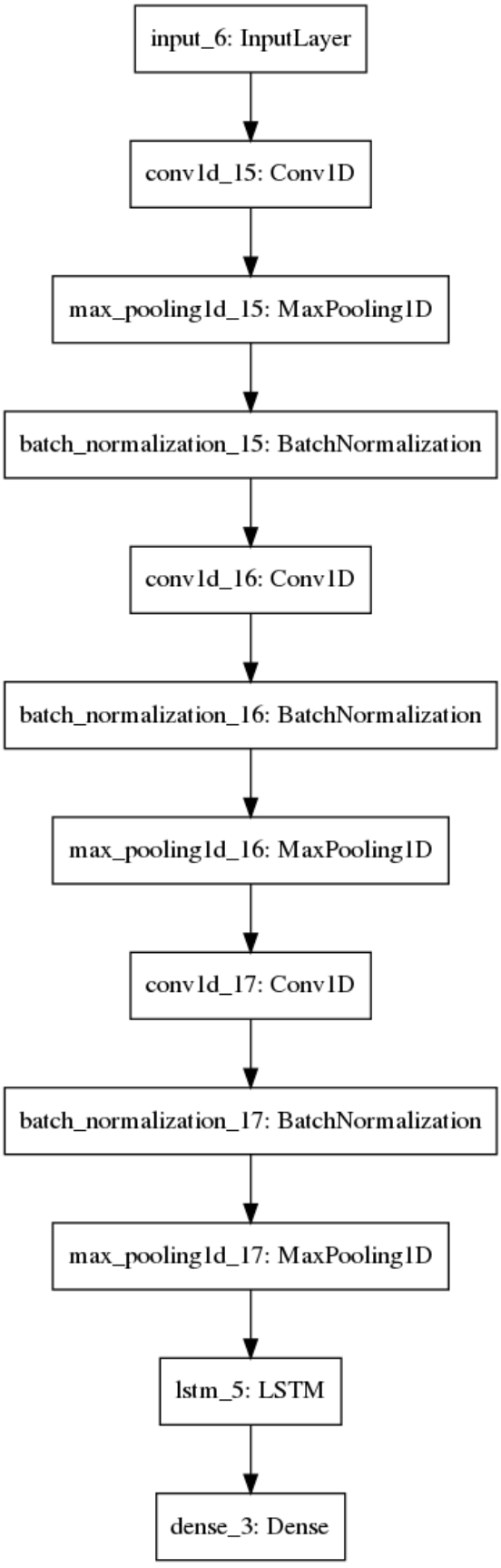


Figure 1: model with 3 CNN and 1 LSTM and 1 Dense layer

```
## ____________________________________________________________________________________________
## 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

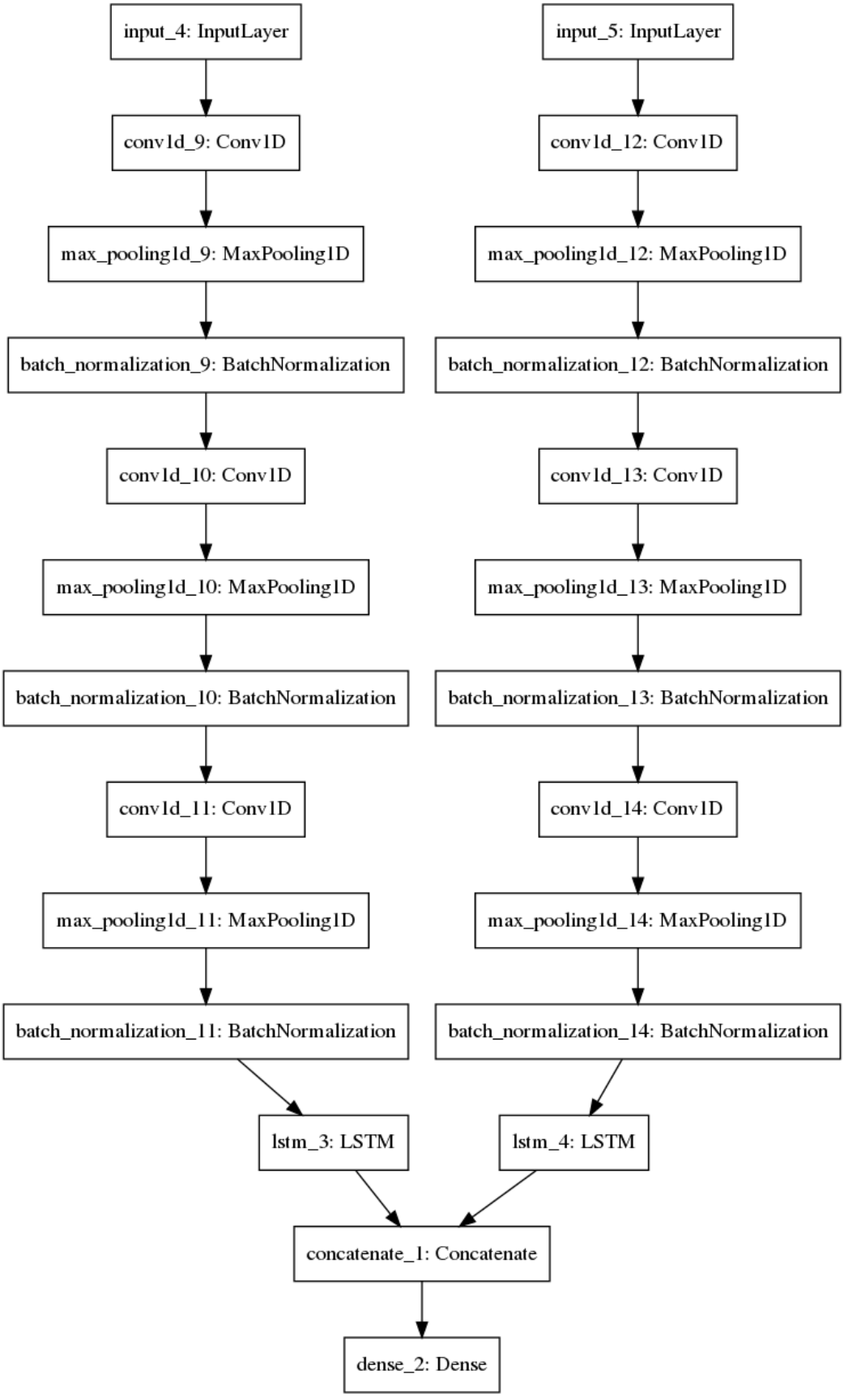


Figure 2: model with two input layers

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 objective 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)
```

Our data set for this tutorial consists of 16S sequences and bacterial genomes.

```
cat("number of files in 16S train:",  length(list.files(path_16S_train)), "\n")
```

```
## number of files in 16S train: 498
```

```
cat("number of files in 16S validation:", length(list.files(path_16S_validation)), "\n")
```

```
## number of files in 16S validation: 100
```

```
cat("number of files in bacteria train:", length(list.files(path_bacteria_train)), "\n")
```

```
## number of files in bacteria train: 20
```

```
cat("number of files in bacteria validation:",  length(list.files(path_bacteria_validation)), "\n")
```

```
## number of files in bacteria validation: 5
```

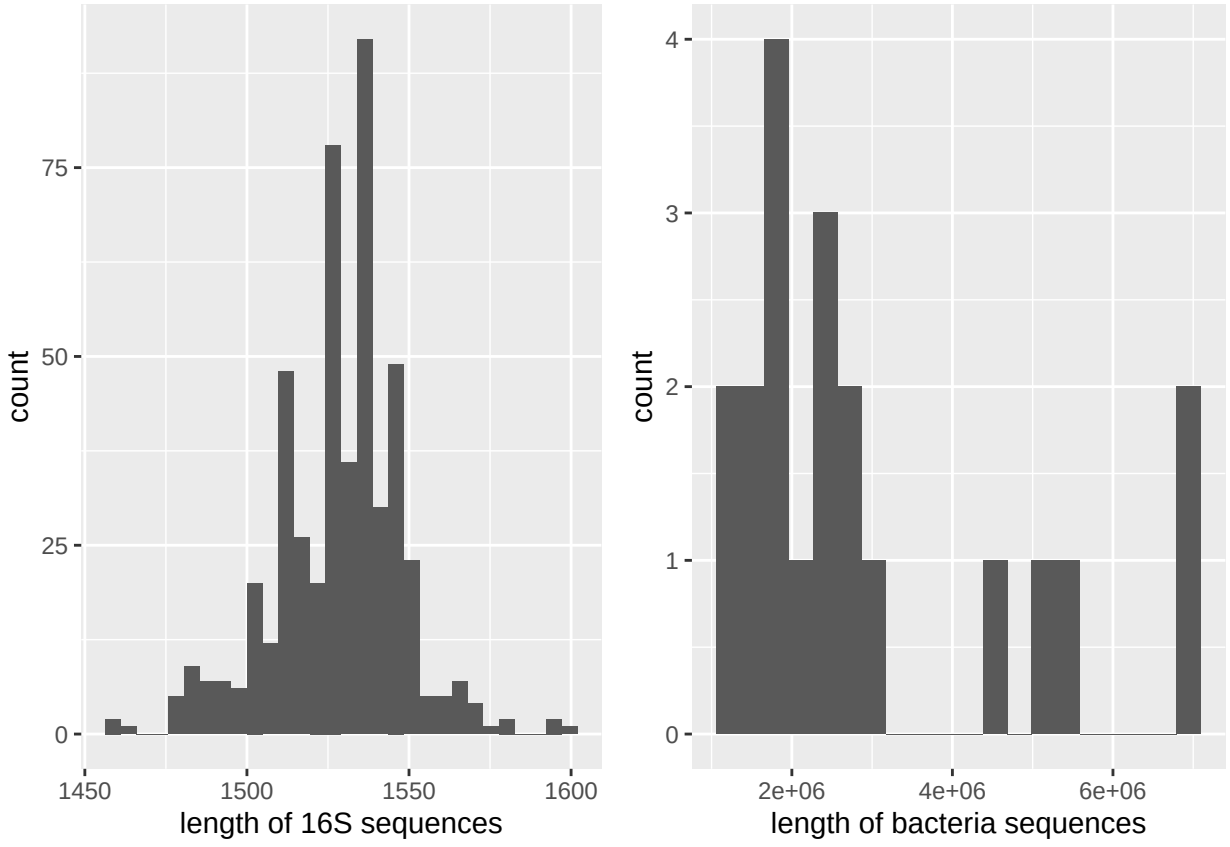
```
# print first 16S file
print(microseq::readFasta(list.files(path_16S_train, full.names = TRUE)[1]))
```

```
## # A tibble: 1 x 2
##   Header                     Sequence
##   <chr>                      <chr>
## 1 16S_rRNA::AM295250.1:1217~ TATGGAGAGTTTGATCCTGGCTCAGGATGAACGCTGGCGGCGTGCCTAAT~
```

```r
seq_length_16_train <- vector("integer", length(path_16S_train))
i <- 1
for (file_name in list.files(path_16S_train, full.names = TRUE)) {
  fasta.file <- microseq::readFasta(file_name)
  seq_length_16_train[i] <- nchar(fasta.file$Sequence)
  i <- i + 1
}
df_16S <- as.data.frame(seq_length_16_train)

seq_length_bacteria_train <- vector("integer", length(path_bacteria_train))
i <- 1
for (file_name in list.files(path_bacteria_train, full.names = TRUE)) {
  fasta.file <- microseq::readFasta(file_name)
  seq_length_bacteria_train[i] <- nchar(fasta.file$Sequence)
  i <- i + 1
}
df_bact <- as.data.frame(seq_length_bacteria_train)

p1 <- ggplot(df_16S, aes(x=seq_length_16_train)) +
  xlab("length of 16S sequences") +
  geom_histogram(bins=30)
p2 <- ggplot(df_bact, aes(x=seq_length_bacteria_train)) +
  xlab("length of bacteria sequences") +
  geom_histogram(bins=20)
ggpubr::ggarrange(p1, p2, ncol = 2, nrow = 1)
```



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, 128),
  pool_size = c(3, 3, 3),
  learning.rate = 0.001
)
```

Next we have to specify the location of our training and validation data and the output format of the data generator. We randomly select 95% from each file so the generator does not take samples from the same position repeatedly.

```
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  train size for validation
             run.name = "lm_16S_target_right",
             batch.size = 256,
             epochs = 5,
             steps.per.epoch = 15, # 1 epoch = 15 batches
             step = 100, # take a sample every 100 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
             proportion_per_file = c(0.95)
```

```
            # randomly select 95% of file
)
```

```
## Trained on 15 samples (batch_size=NULL, epochs=5)
## Final epoch (plot to see history):
##     loss: 1.112
##      acc: 0.5422
##       f1: Inf
## val_loss: 1.096
##  val_acc: 0.543
##   val_f1: Inf
##       lr: 0.001
```

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

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, 128),
  pool_size = c(3, 3, 3),
  learning.rate = 0.001
)
```

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 train size for validation
             run.name = "lm_16S_target_middle_lstm",
             batch.size = 256,
             epochs = 5,
             steps.per.epoch = 15, # 1 epoch = 15 batches
             step = 100, # take a sample every 100 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
            proportion_per_file = c(0.95)
            # randomly select 95% of file
)
```

```
## Trained on 15 samples (batch_size=NULL, epochs=5)
## Final epoch (plot to see history):
##     loss: 0.8762
##      acc: 0.6826
##       f1: Inf
## val_loss: 0.8886
##  val_acc: 0.6602
##   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 separate folders

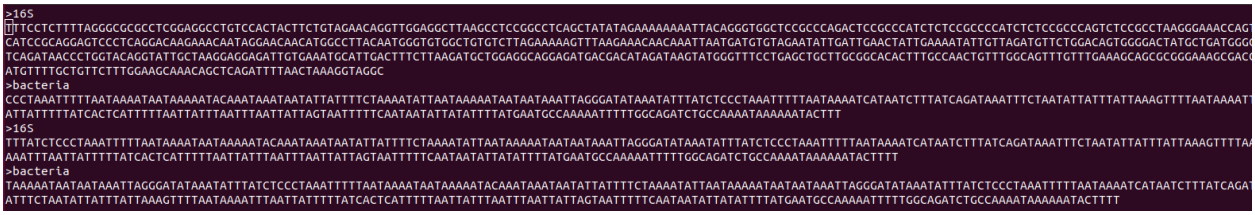


Figure 3: Example: read class label from header

Label by folder

We put all data from each 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 = NULL,
  layer_dense = c(2), # predict 2 classes
  vocabulary.size = 4,
  kernel_size = c(12, 12, 12),
  filters = c(32, 64, 128),
  pool_size = c(3, 3, 3),
  learning.rate = 0.001
)
```

```
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 = 256, # half of batch is 16S and other half bacteria data
             epochs = 5,
             save_best_only = FALSE,
             steps.per.epoch = 15,
             step = c(100, 500), # smaller step size for 16S
             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(0.95, 0.05)
             # randomly select 95% of 16S and 5% of bacteria files,
             # since bacteria files are much larger
)
```

```
## Trained on 15 samples (batch_size=NULL, epochs=5)
## Final epoch (plot to see history):
##     loss: 0.009258
##      acc: 0.9987
##       f1: 0.9987
## val_loss: 0.002073
##  val_acc: 1
##   val_f1: 1
##       lr: 0.001
```

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.

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

```
## [1] "Ep.001-val_loss0.08-val_acc0.975.hdf5"
## [2] "Ep.002-val_loss0.05-val_acc0.982.hdf5"
## [3] "Ep.003-val_loss0.01-val_acc0.996.hdf5"
## [4] "Ep.004-val_loss0.02-val_acc0.993.hdf5"
## [5] "Ep.005-val_loss0.00-val_acc1.000.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.

```
model <- create_model_lstm_cnn(
  maxlen = 500,
  layer_lstm = NULL,
  layer_dense = c(2),
  vocabulary.size = 4,
  kernel_size = c(12, 12, 12),
  filters = c(32, 64, 128),
  pool_size = c(3, 3, 3),
  learning.rate = 0.001
)

# evaluate 1000 samples, 500 from each class
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",
                            verbose = FALSE)
```

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

```
## [1] 0.499
```

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

```
##            Truth
## Prediction 16s bacteria
##   16s        0        1
##   bacteria 500      499
```

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",
                            verbose = FALSE)
```

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

```
## [1] 0.995
```

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

```
##            Truth
## Prediction 16s bacteria
##   16s      495        0
##   bacteria   5      500
```

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. First, we apply our model to a file from our 16S validation set.

```
model <- keras::load_model_hdf5(weight_path, compile = FALSE)
model <- keras::load_model_weights_hdf5(model, weight_path)
```

```
maxlen <- model$input$shape[[2]]
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, # make predictions for 16S file
                         full.names = TRUE)[1]
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")
}
```

We can access the h5 file as follows

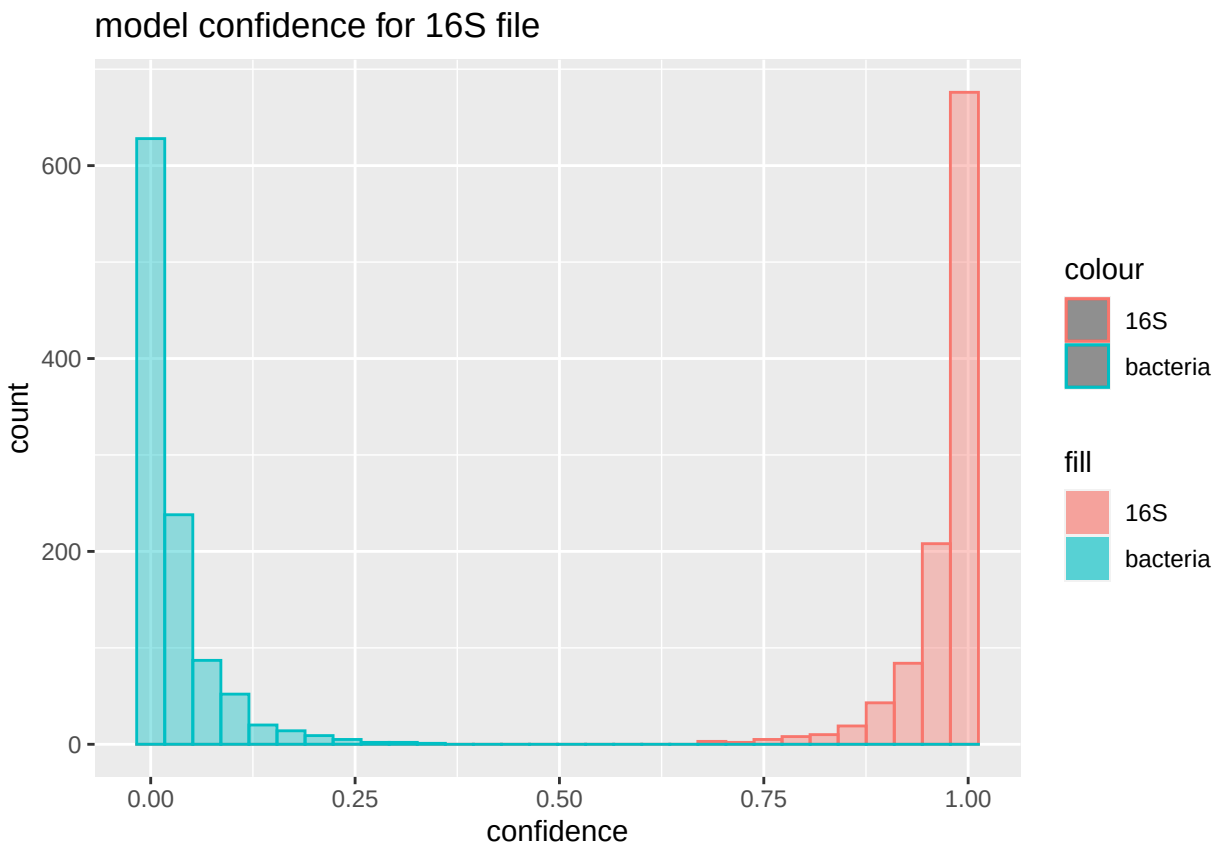
```
states <- readRowsFromH5(h5_path = filename, complete = TRUE)
```

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

```
colnames(states) <- c("conf_16S", "conf_bacteria")
rownames(states) <- paste0("sample_", 1:nrow(states))
head(states)
```

```
##          conf_16S conf_bacteria
## sample_1   0.9960        0.0040
## sample_2   0.9919        0.0081
## sample_3   0.9951        0.0049
## sample_4   0.9956        0.0044
## sample_5   0.9953        0.0047
## sample_6   0.9961        0.0039
```

```
ggplot(as.data.frame(states)) +
  geom_histogram(aes(conf_16S, color = "16S", fill = "16S"), alpha = 0.4) +
  geom_histogram(aes(conf_bacteria, color = "bacteria", fill = "bacteria"), alpha = 0.4) +
  xlab("confidence") + ggtitle("model confidence for 16S file")
```



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

In the following example we use the binary model trained to classify 16S/bacteria to predict the location of 16S sequences (similar to https://github.com/tseemann/barrnap). We use the same model architecture as before, but load the weights of a model that was trained on more data and for a longer time. We iterate over a new bacteria file (not present in train or validation data) and make predictions every 100 steps

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

```
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   0.0018        0.9982     500
## 2   0.0027        0.9973     600
## 3   0.0029        0.9971     700
## 4   0.0028        0.9972     800
## 5   0.0012        0.9988     900
## 6   0.0014        0.9986    1000
```

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

```
##       conf_16S conf_bacteria seq_end
## 20241   0.6726        0.3274 2024500
## 20368   0.6233        0.3767 2037200
## 21125   0.6095        0.3905 2112900
## 21897   0.9963        0.0037 2190100
## 21898   0.9946        0.0054 2190200
## 21899   0.9627        0.0373 2190300
```

Next, we search for the true rRNA region in the corresponding gff file to validate our models predictions.

```
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"]
cat("start positions of 16S sequences:", start, "\n")
```

```
## start positions of 16S sequences: 2189670 2933745
```

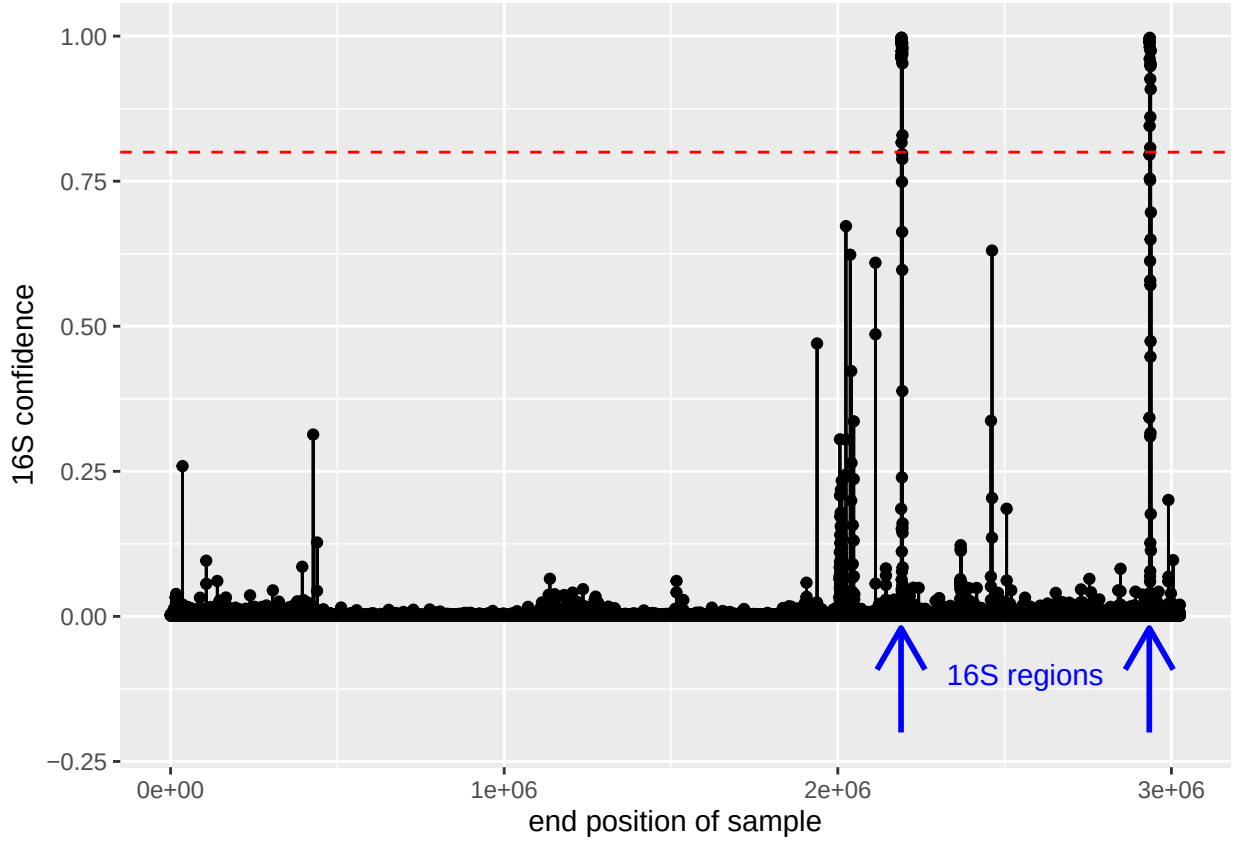
```
cat("end positions of 16S sequences:", end, "\n")
```

```
## end positions of 16S sequences: 2191227 2935302
```

We can now plot the models confidence in 16S over the whole genome.

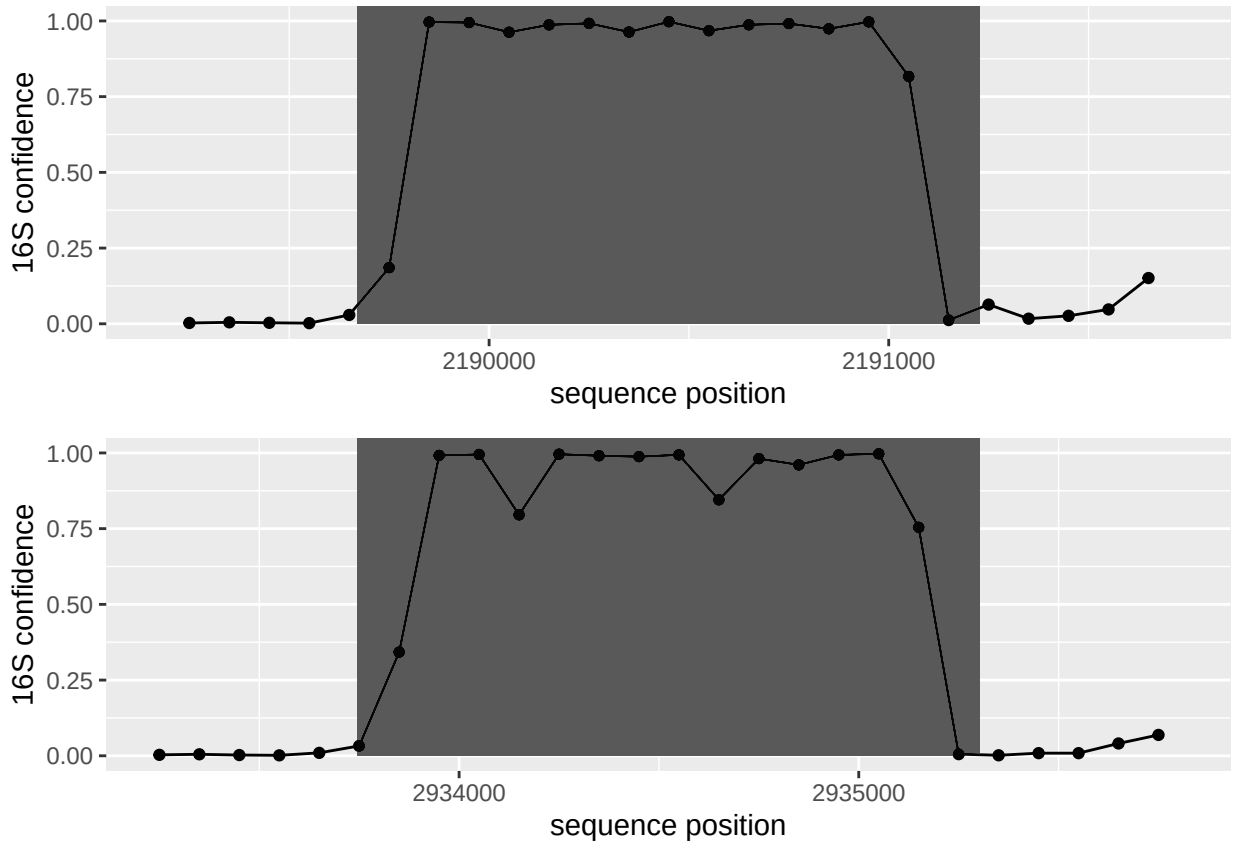
```
ggplot(df, aes(x = seq_end, y = conf_16S)) + geom_point() +
  geom_line() + ylab("16S confidence") + xlab("end position of sample") +
  geom_hline(yintercept=0.8, linetype="dashed", color = "red") +
```

```
  annotate("segment", xend = start[1], x = start[1], y = -0.2, yend = -0.02,
           colour = "blue", size = 1, arrow = arrow()) +
  annotate("segment", xend = start[2], x = start[2], y = -0.2, yend = -0.02,
           colour = "blue", size = 1, arrow = arrow()) +
  annotate("text", x = mean(c(start[1], start[2])), y = -0.1, label = "16S regions", color = "blue")
```



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

```
p1 <- ggplot(df, aes(x = seq_end - 250, 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 - 250, 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)
```



Finally, let's decrease our step size parameter to 1 and make a predictions for every possible sample around the true 16S regions (add 500 nucleotides to start/end)

```
fasta.file <- microseq::readFasta(fasta.path)
buffer <- 500
sequence_16S_1 <- fasta.file$Sequence[1] %>%
  substr(start = start[1] - buffer, stop = end[1] + buffer)
sequence_16S_2 <- fasta.file$Sequence[1] %>%
  substr(start = start[2] - buffer, stop = end[2] + buffer)

filename_16S_1 <- file.path(dir_path, "states_16S_1.h5")
filename_16S_2 <- file.path(dir_path, "states_16S_2.h5")

if (!file.exists(filename_16S_1)) {
  writeStates(
    model = model,
    layer_name = layer_name,
    sequence = sequence_16S_1,
    round_digits = 4,
    filename = filename_16S_1,
    batch.size = 500,
    step = 1)
}

if (!file.exists(filename_16S_2)) {
  writeStates(
```

```
    model = model,
    layer_name = layer_name,
    sequence = sequence_16S_2,
    round_digits = 4,
    filename = filename_16S_2,
    batch.size = 500,
    step = 1)
}

states_1 <- readRowsFromH5(h5_path = filename_16S_1, complete = TRUE, getTargetPositions = TRUE)
```

```
## states matrix has 2058 rows and  2 columns
```

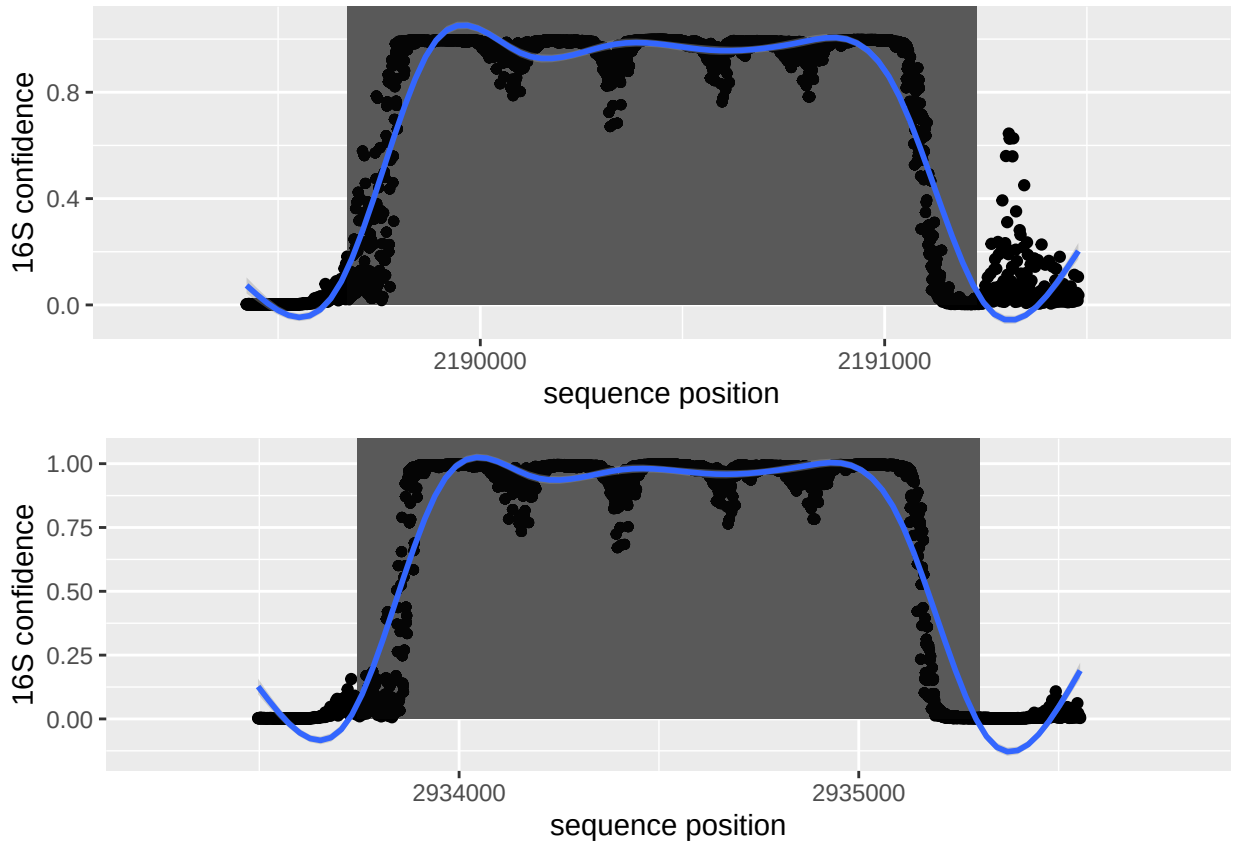
```
pred_1 <- states_1[[1]]
position_1 <- start[1] - buffer + 1 + states_1[[2]] - floor(maxlen/2)
df_1 <- cbind(pred_1, position_1) %>% as.data.frame()
colnames(df_1) <- c("conf_16S", "conf_bacteria", "seq_end")
states_2 <- readRowsFromH5(h5_path = filename_16S_2, complete = TRUE, getTargetPositions = TRUE)
```

```
## states matrix has 2058 rows and  2 columns
```

```
pred_2 <- states_2[[1]]
position_2 <-  start[2] - buffer + 1 + states_2[[2]] - floor(maxlen/2)
df_2 <- cbind(pred_2, position_2) %>% as.data.frame()
colnames(df_2) <- c("conf_16S", "conf_bacteria", "seq_end")
```

and plot the results again

```
p1 <- ggplot(df_1, aes(x = seq_end, y = conf_16S)) + geom_point() +
  geom_rect(aes(xmin=start[1], xmax=end[1], ymin=0, ymax=Inf),  alpha = 0.01) +
  xlim(c(start[1] - buffer, end[1] + buffer)) +
  ylab("16S confidence") + xlab("sequence position") +
  geom_smooth()
p2 <- ggplot(df_2, aes(x = seq_end, y = conf_16S)) + geom_point() +
  geom_rect(aes(xmin=start[2], xmax=end[2], ymin=0, ymax=Inf),  alpha = 0.01) +
  xlim(c(start[2] - buffer, end[2] + buffer)) +
  ylab("16S confidence") + xlab("sequence position") +
  geom_smooth()
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 our 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 misclassified more bacteria sequences as 16S than vice versa in one of the epochs.

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

The "HPARAM" tab tracks the hyper parameters of the different runs (maxlen, batch size etc.). This can helpful to find the optimal hyper parameter settings for a given task

Further tensorboard documentation can be found here.

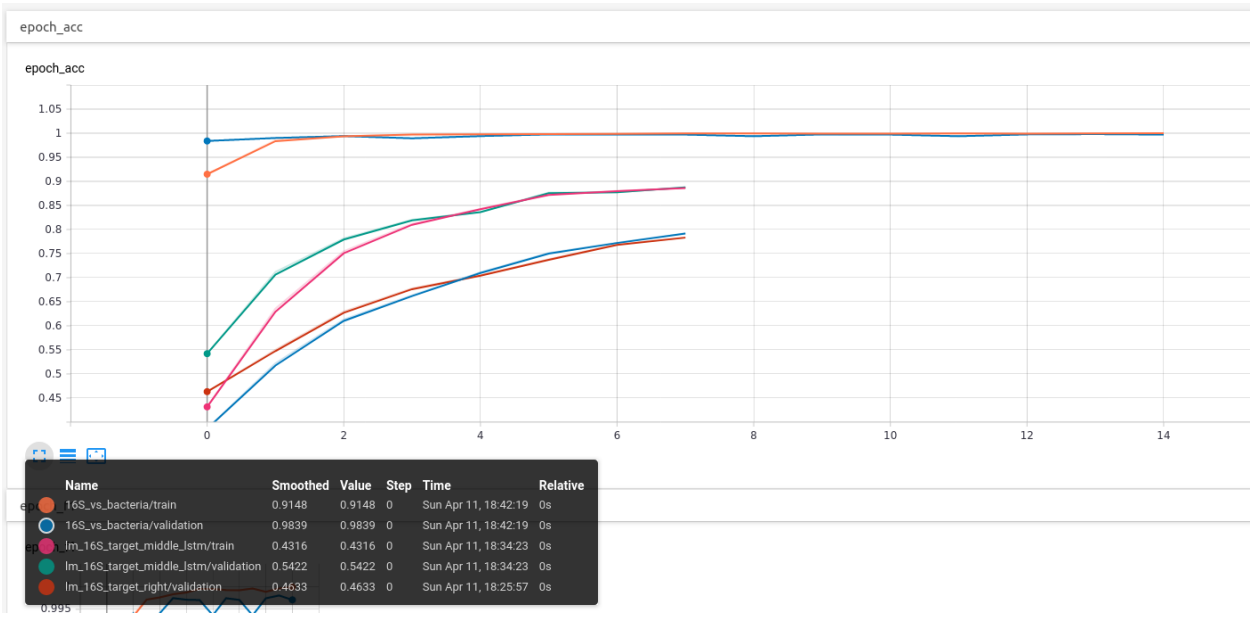


Figure 4: accuracy

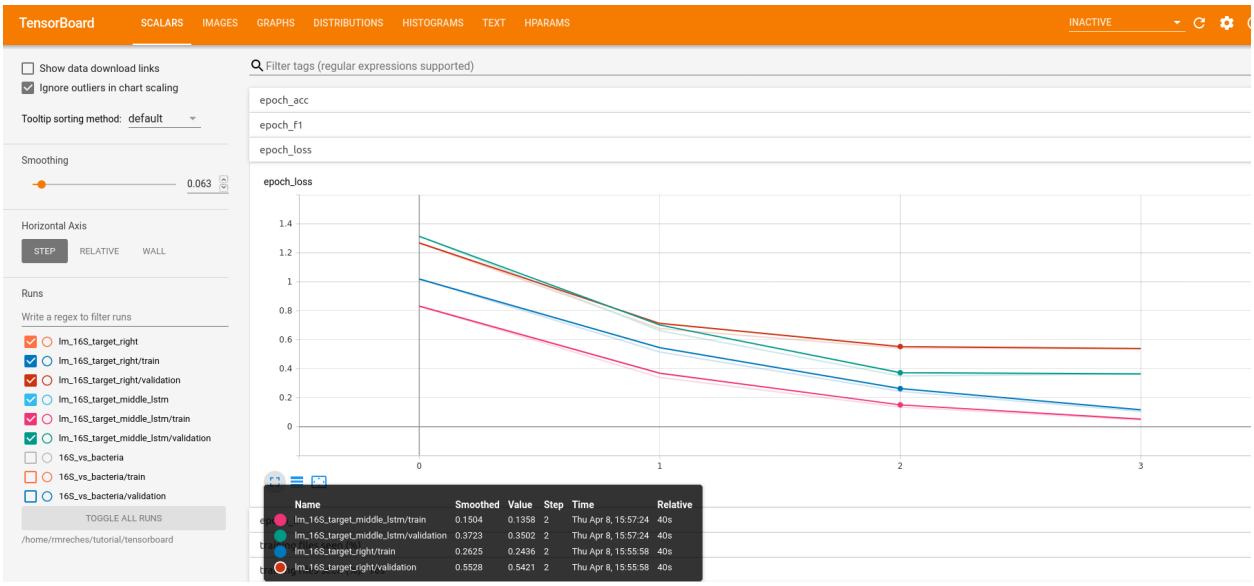


Figure 5: loss

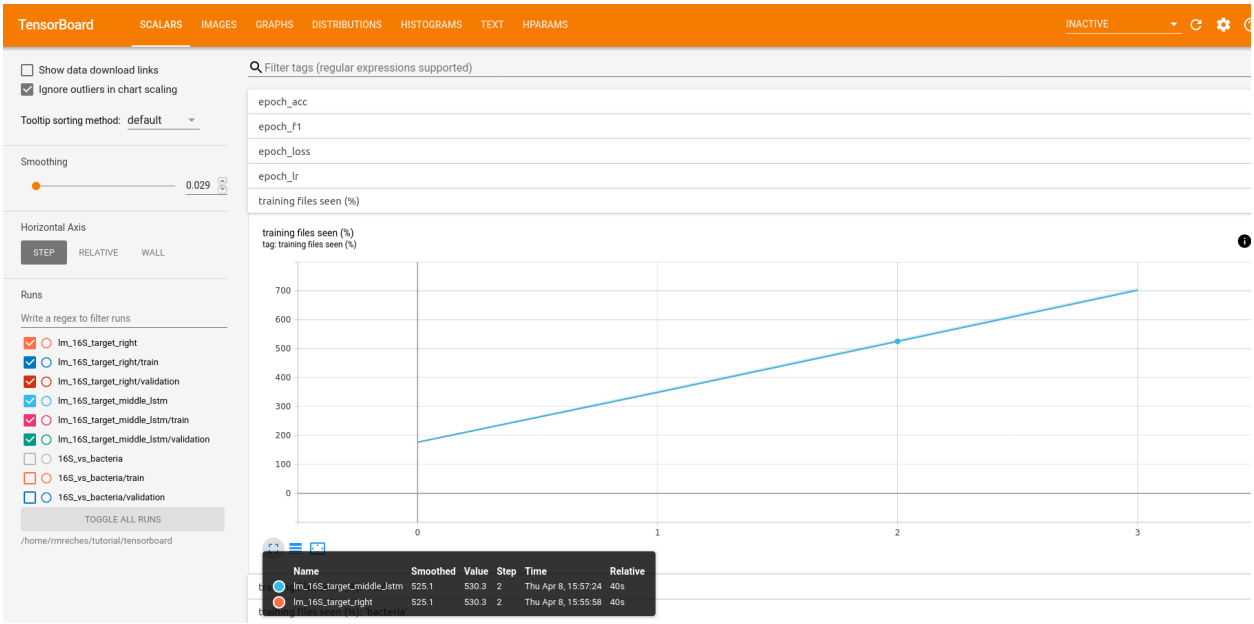


Figure 6: percentage of seen training files

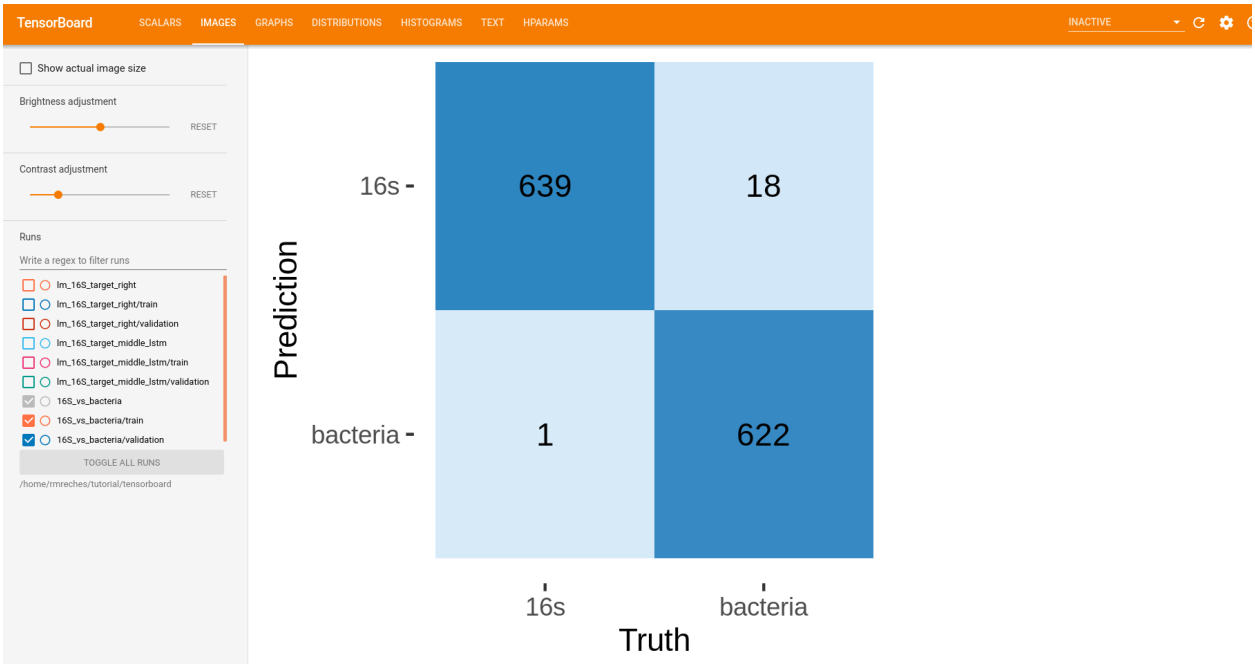


Figure 7: confusion matrix

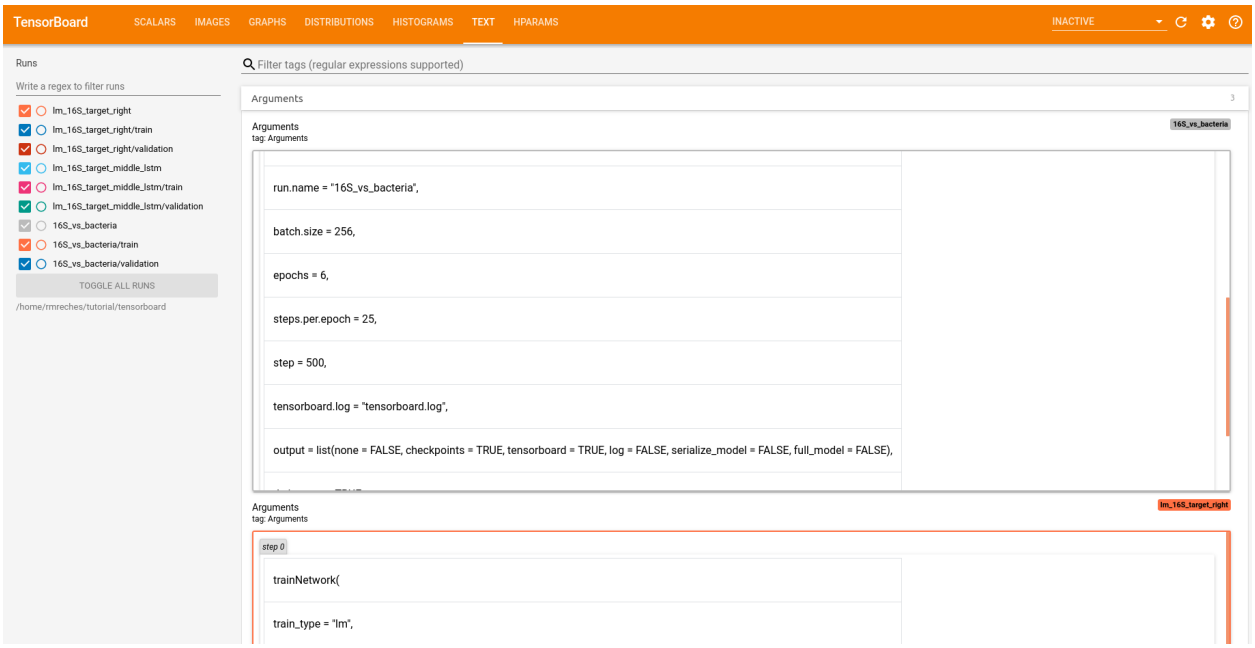


Figure 8: trainNetwork call

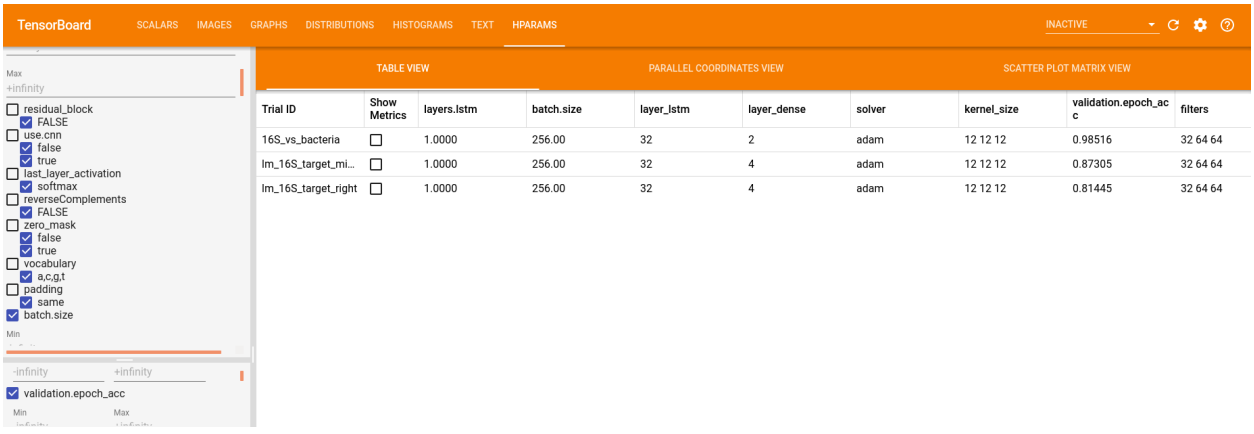


Figure 9: hyperparameters

Integrated gradient

To visualize which parts of an input sequence is important for the models decision, we may use a method called Integrated Gradient (paper). This compares the predictions for a given sequence to a baseline sequence (usually a zero tensor or shuffling of original sequence) to determine which features were important for the models decision.

```
# load trained model
model_path <- "pretrained_models/bact_16S_model.hdf5"
model <- keras::load_model_hdf5(model_path, compile = FALSE)
model <- keras::load_model_weights_hdf5(model, model_path)
```

```
fasta_path <- file.path(path_16S_validation,
                        "GCF_000427035.1_09mas018883_genomic.16s.fasta.fasta")
fasta_file <- microseq::readFasta(fasta_path)

# extract input tensors with data generator
gen <- labelByFolderGenerator(corpus.dir = fasta_path,
                              batch.size = 1,
                              maxlen = 500,
                              reverseComplements = FALSE,
                              numTargets = 2,
                              onesColumn = 1,
                              step = 1,
                              padding = FALSE)

ig_list <- list()
seq_len <- nchar(fasta_file$Sequence)
# for (i in 1:(seq_len - 500)) {
#   z <- gen()
#   input <- z[[1]]
#   ig <- integrated_gradients(m_steps = 50,
#                              baseline_type = "shuffle",
#                              input_seq = input,
#                              target_class_idx = 1,
#                              model = model,
#                              num_baseline_repeats = 10)
#
#   py$integrated_grads <- ig
#   py_run_string("attribution_mask = tf.reduce_sum(tf.math.abs(integrated_grads), axis=-1)")
#   attribution_mask <- py$attribution_mask
#   attribution_mask <- as.matrix(attribution_mask, ncol = 1)
#   df <- data.frame(position = (1:(nrow(attribution_mask))) + i - 1,
#                    ig_sum = attribution_mask[ , 1])
#   ig_list[[i]] <- df
# }
# saveRDS(ig_list, paste0(dir_path, "/ig_list.rds"))
ig_list <- readRDS(paste0(dir_path, "/ig_list.rds"))

ig_df <- data.table::rbindlist(ig_list) %>% as.data.frame()
ig_df <- aggregate(x = ig_df$ig_sum,
                           by = list(ig_df$position),
                           FUN = mean)
```

```r
names(ig_df) <- c("position", "ig_sum")
ggplot(ig_df, aes(x = position, y = ig_sum, alpha = 0.001)) +
  #geom_point() +
  geom_smooth() + ylab("feature importance") +
  theme(legend.position = "none") +
  geom_line()
```