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.

```r
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,] 0.2418842 0.2160256 0.3328353 0.2092548
## [2,] 0.2363858 0.2228332 0.3235803 0.2172006
## [3,] 0.2369523 0.2181571 0.3296198 0.2152708
```

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

```r
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: 0.1063
##      acc: 0.9934
##       f1: Inf
## val_loss: 0.5382
##  val_acc: 0.8145
##   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.

```r
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: 0.04586
##      acc: 0.9988
##       f1: Inf
## val_loss: 0.3642
##  val_acc: 0.873
##   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),
```

```r
  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
## ________________________________________________________________________________
```

```r
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 = 6,
             steps.per.epoch = 25,
             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 25 samples (batch_size=NULL, epochs=6)
## Final epoch (plot to see history):
##     loss: 0.004071
##      acc: 0.9991
##       f1: 0.9991
## val_loss: 0.05052
##  val_acc: 0.9852
##   val_f1: 0.9849
##       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,] 0.9995    5e-04
## [2,] 0.9996    4e-04
## [3,] 0.9997    3e-04
## [4,] 0.9993    7e-04
## [5,] 0.9995    5e-04
## [6,] 0.9995    5e-04
```

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.

Inference II

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)
```

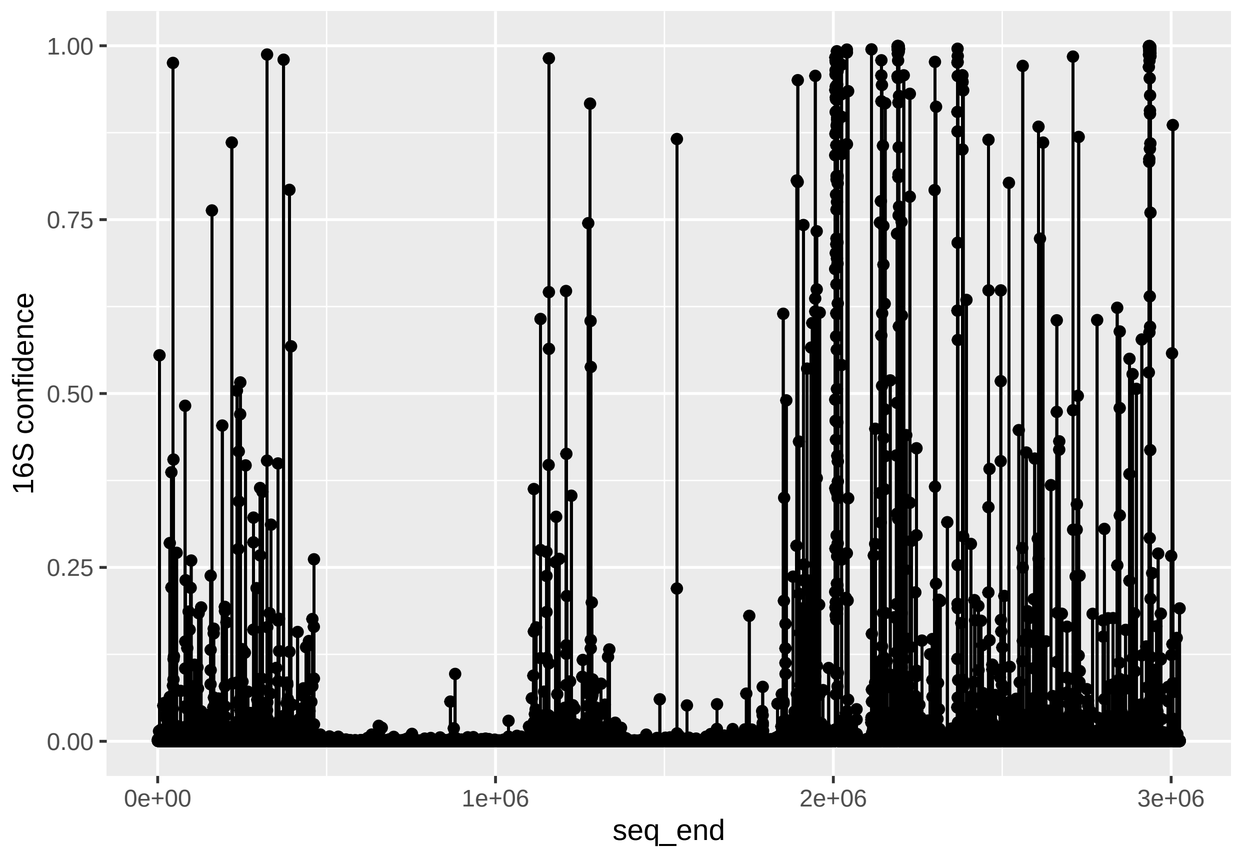
```
##   conf_16S conf_bacteria seq_end
## 1   0.0027        0.9973     500
## 2   0.0005        0.9995     600
## 3   0.0006        0.9994     700
## 4   0.0010        0.9990     800
## 5   0.0005        0.9995     900
## 6   0.0007        0.9993    1000
```

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

```
##       conf_16S conf_bacteria seq_end
## 49      0.5550        0.4450    5300
## 448     0.9753        0.0247   45200
## 1601    0.7632        0.2368  160500
## 2189    0.8609        0.1391  219300
## 2342    0.5041        0.4959  234600
## 2441    0.5160        0.4840  244500
## 3234    0.9874        0.0126  323800
## 3723    0.9799        0.0201  372700
## 3895    0.7928        0.2072  389900
## 3943    0.5679        0.4321  394700
## 11328   0.6072        0.3928 1133200
## 11576   0.9819        0.0181 1158000
## 11577   0.6458        0.3542 1158100
## 11578   0.5642        0.4358 1158200
## 12083   0.6474        0.3526 1208700
## 12741   0.7449        0.2551 1274500
## 12795   0.9168        0.0832 1279900
## 12810   0.6043        0.3957 1281400
## 12817   0.5382        0.4618 1282100
## 15366   0.8659        0.1341 1537000
## 18514   0.6147        0.3853 1851800
## 18915   0.8063        0.1937 1891900
## 18942   0.8041        0.1959 1894600
## 18944   0.9505        0.0495 1894800
## 19111   0.7423        0.2577 1911500
## 19220   0.5358        0.4642 1922400
## 19340   0.5661        0.4339 1934400
## 19373   0.6013        0.3987 1937700
## 19460   0.6366        0.3634 1946400
## 19461   0.6180        0.3820 1946500
## 19462   0.9567        0.0433 1946600
## 19504   0.7333        0.2667 1950800
## 19505   0.6498        0.3502 1950900
## 19593   0.6163        0.3837 1959700
## 20050   0.6790        0.3210 2005400
## 20052   0.8426        0.1574 2005600
## 20055   0.9363        0.0637 2005900
## 20056   0.9832        0.0168 2006000
## 20059   0.8733        0.1267 2006300
## 20061   0.9587        0.0413 2006500
## 20062   0.9788        0.0212 2006600
## 20063   0.9648        0.0352 2006700
## 20065   0.9047        0.0953 2006900
## 20067   0.9768        0.0232 2007100
## 20068   0.7019        0.2981 2007200
## 20070   0.9415        0.0585 2007400
## 20071   0.9259        0.0741 2007500
## 20074   0.9231        0.0769 2007800
## 20076   0.7860        0.2140 2008000
## 20077   0.8755        0.1245 2008100
## 20078   0.9354        0.0646 2008200
## 20079   0.5822        0.4178 2008300
## 20080   0.9389        0.0611 2008400
## 20082   0.9058        0.0942 2008600
## 20083   0.6154        0.3846 2008700
## 20084   0.7146        0.2854 2008800
## 20085   0.8570        0.1430 2008900
## 20086   0.9619        0.0381 2009000
## 20088   0.6570        0.3430 2009200
## 20089   0.7229        0.2771 2009300
## 20090   0.7644        0.2356 2009400
## 20091   0.8078        0.1922 2009500
## 20092   0.8127        0.1873 2009600
## 20093   0.8853        0.1147 2009700
## 20095   0.9585        0.0415 2009900
## 20097   0.7148        0.2852 2010100
## 20098   0.5631        0.4369 2010200
## 20099   0.6140        0.3860 2010300
## 20100   0.9221        0.0779 2010400
## 20101   0.9921        0.0079 2010500
## 20103   0.5066        0.4934 2010700
## 20104   0.6941        0.3059 2010800
## 20105   0.7754        0.2246 2010900
## 20107   0.8940        0.1060 2011100
## 20108   0.8964         0.1036 2011200
## 20110   0.9441         0.0559 2011400
## 20115   0.8897         0.1103 2011900
## 20116   0.9839         0.0161 2012000
## 20118   0.7172         0.2828 2012200
## 20119   0.8098         0.1902 2012300
## 20120   0.6868         0.3132 2012400
## 20122   0.9634         0.0366 2012600
## 20123   0.8128         0.1872 2012700
## 20125   0.8023         0.1977 2012900
## 20128   0.6296         0.3704 2013200
## 20239   0.9303         0.0697 2024300
## 20240   0.5410         0.4590 2024400
## 20241   0.9728         0.0272 2024500
## 20242   0.8442         0.1558 2024600
## 20243   0.8978         0.1022 2024700
## 20397   0.9945         0.0055 2040100
## 20398   0.9903         0.0097 2040200
## 20399   0.8585         0.1415 2040300
## 20435   0.9347         0.0653 2043900
## 21126   0.9948         0.0052 2113000
## 21376   0.7458         0.2542 2138000
## 21404   0.7766         0.2234 2140800
## 21417   0.5835         0.4165 2142100
## 21418   0.9573         0.0427 2142200
## 21419   0.9791         0.0209 2142300
## 21420   0.9201         0.0799 2142400
## 21436   0.9436         0.0564 2144000
## 21439   0.5111         0.4889 2144300
## 21442   0.6153         0.3847 2144600
## 21465   0.8563         0.1437 2146900
## 21478   0.7410         0.2590 2148200
## 21480   0.6853         0.3147 2148400
## 21517   0.6290         0.3710 2152100
## 21532   0.9173         0.0827 2153600
## 21681   0.5189         0.4811 2168500
## 21882   0.7297         0.2703 2188600
## 21896   0.9561         0.0439 2190000
## 21897   0.9985         0.0015 2190100
## 21898   0.9994         0.0006 2190200
## 21899   0.9952         0.0048 2190300
## 21900   0.9973         0.0027 2190400
## 21901   0.9991         0.0009 2190500
## 21902   0.9962         0.0038 2190600
## 21903   0.9997         0.0003 2190700
## 21904   0.9971         0.0029 2190800
## 21905   0.9946         0.0054 2190900
## 21906   0.9538         0.0462 2191000
## 21907   0.9883         0.0117 2191100
## 21908   0.9997         0.0003 2191200
## 21909   0.9891         0.0109 2191300
## 21914   0.9787         0.0213 2191800
## 21915   0.9914         0.0086 2191900
## 21916   0.9925         0.0075 2192000
## 21917   0.9929         0.0071 2192100
## 21918   0.9970         0.0030 2192200
## 21919   0.9994         0.0006 2192300
## 21920   0.9964         0.0036 2192400
## 21922   0.8111         0.1889 2192600
## 21923   0.9183         0.0817 2192700
## 21924   0.9982         0.0018 2192800
## 21925   0.9935         0.0065 2192900
## 21926   0.9538         0.0462 2193000
## 21927   0.8153         0.1847 2193100
## 21928   0.9978         0.0022 2193200
## 21929   0.7561         0.2439 2193300
## 21930   0.8540         0.1460 2193400
## 21931   0.9986         0.0014 2193500
## 21932   0.9974         0.0026 2193600
## 21933   0.9994         0.0006 2193700
## 21934   0.9914         0.0086 2193800
## 21935   0.9940         0.0060 2193900
## 21936   0.5965         0.4035 2194000
## 21937   0.9959         0.0041 2194100
## 21938   0.7306         0.2694 2194200
## 21939   0.9948         0.0052 2194300
## 21940   0.9957         0.0043 2194400
## 21941   0.9278         0.0722 2194500
## 21942   0.7687         0.2313 2194600
## 22017   0.7469         0.2531 2202100
## 22027   0.6121         0.3879 2203100
## 22081   0.9575         0.0425 2208500
## 22260   0.7830         0.2170 2226400
## 22261   0.9309         0.0691 2226500
## 22995   0.7925         0.2075 2299900
## 23005   0.9768         0.0232 2300900
## 23037   0.9123         0.0877 2304100
## 23669   0.9049         0.0951 2367300
## 23670   0.6192         0.3808 2367400
## 23671   0.9759         0.0241 2367500
## 23672   0.8768         0.1232 2367600
## 23674   0.9957         0.0043 2367800
## 23676   0.7168         0.2832 2368000
## 23677   0.9766         0.0234 2368100
## 23679   0.9565         0.0435 2368300
## 23680   0.9852         0.0148 2368400
## 23682   0.5770         0.4230 2368600
## 23818   0.9488         0.0512 2382200
## 23819   0.9576         0.0424 2382300
## 23820   0.8508         0.1492 2382400
## 23821   0.9476         0.0524 2382500
## 23842   0.9358         0.0642 2384600
## 23937   0.6346         0.3654 2394100
## 24589   0.8648         0.1352 2459300
## 24591   0.6483         0.3517 2459500
## 24953   0.5178         0.4822 2495700
## 24954   0.6483         0.3517 2495800
## 25194   0.8029         0.1971 2519800
## 25603   0.9710         0.0290 2560700
## 26070   0.8836         0.1164 2607400
## 26114   0.7227         0.2773 2611800
## 26206   0.8608         0.1392 2621000
## 26611   0.6053         0.3947 2661500
## 27090   0.9844         0.0156 2709400
## 27260   0.8689         0.1311 2726400
## 27806   0.6056         0.3944 2781000
## 28400   0.6233         0.3767 2840400
## 28475   0.5892         0.4108 2847900
## 28763   0.5499         0.4501 2876700
## 28855   0.5279         0.4721 2885900
## 28966   0.5067         0.4933 2897000
## 29127   0.5777         0.4223 2913100
## 29336   0.5303         0.4697 2934000
## 29337   0.9695         0.0305 2934100
## 29338   0.9992         0.0008 2934200
## 29339   0.9994         0.0006 2934300
## 29340   0.9871         0.0129 2934400
## 29341   0.9990         0.0010 2934500
## 29342   0.9986         0.0014 2934600
## 29343   0.9964         0.0036 2934700
## 29344   0.9975         0.0025 2934800
## 29345   0.9876         0.0124 2934900
## 29346   0.9987         0.0013 2935000
## 29347   0.8332         0.1668 2935100
## 29348   0.9925         0.0075 2935200
## 29349   0.9997         0.0003 2935300
## 29350   0.8373         0.1627 2935400
## 29353   0.5879         0.4121 2935700
## 29356   0.9786         0.0214 2936000
## 29357   0.9901         0.0099 2936100
## 29358   0.9958         0.0042 2936200
## 29359   0.9968         0.0032 2936300
## 29360   0.9969         0.0031 2936400
## 29361   0.9995         0.0005 2936500
## 29362   0.9532         0.0468 2936600
## 29364   0.6397         0.3603 2936800
## 29365   0.9947         0.0053 2936900
## 29366   0.9985         0.0015 2937000
## 29367   0.9957         0.0043 2937100
## 29368   0.8518         0.1482 2937200
## 29369   0.9069         0.0931 2937300
## 29370   0.9975         0.0025 2937400
## 29371   0.9022         0.0978 2937500
## 29372   0.5960         0.4040 2937600
## 29373   0.9286         0.0714 2937700
## 29374   0.9967         0.0033 2937800
## 29375   0.9986         0.0014 2937900
## 29376   0.9917         0.0083 2938000
## 29377   0.9892         0.0108 2938100
## 29379   0.9937         0.0063 2938300
## 29380   0.9958         0.0042 2938400
## 29381   0.9982         0.0018 2938500
## 29382   0.8596         0.1404 2938600
## 29383   0.9893         0.0107 2938700
## 29384   0.9850         0.0150 2938800
## 29385   0.7600         0.2400 2938900
## 30023   0.5578         0.4422 3002700
## 30046   0.8860         0.1140 3005000
```

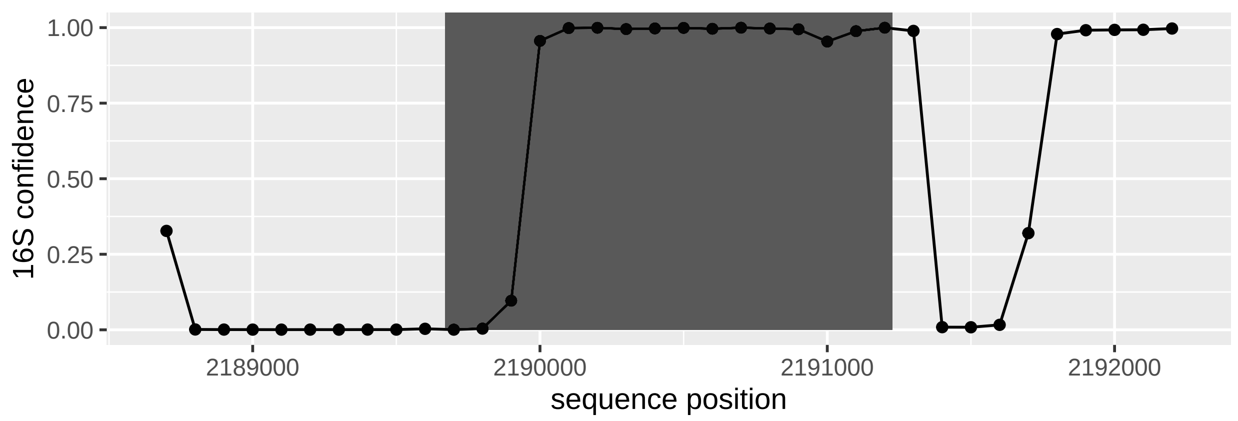
Let's visualize or models predictions and compare them to the true areas. First we look at the confidence in 16S over the whole genome.

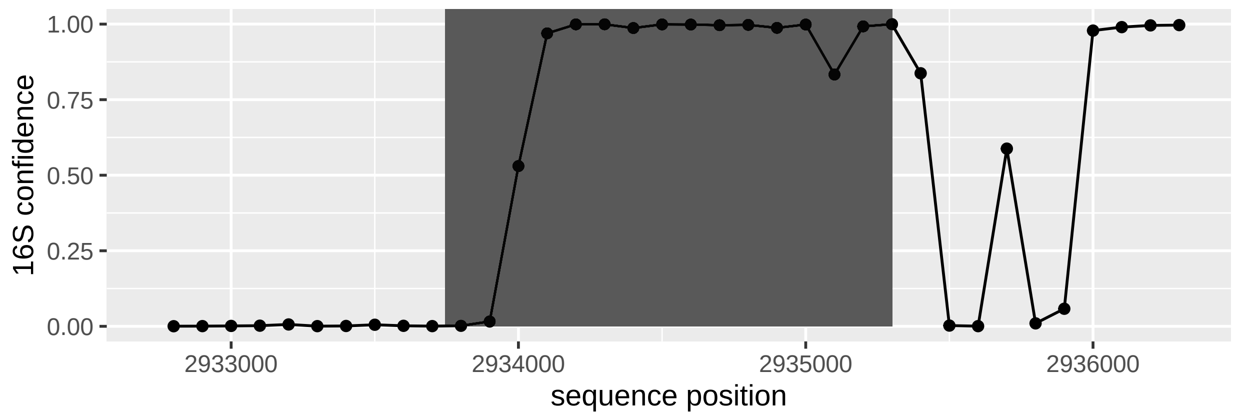
```r
ggplot(df, aes(x = seq_end, y = conf_16S)) + geom_point() + geom_line() + ylab("16S confidence")
```



Next we may zoom into areas with high 16S confidence.

```r
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] - 1000, end[1] + 1000)) +
  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] - 1000, end[2] + 1000)) +
  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.

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

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

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

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

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

Further tensorboard documentation can be found here.

Checkpoints

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

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

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

```
# cp_run_path <- file.path(checkpoint_path, paste0(run.name, "_checkpoints"))
# checkpoints <- list.files(cp_run_path)
# checkpoints
# last_checkpoint <- checkpoints[length(checkpoints)]
#
# # load trained model and compile
# model <- keras::load_model_hdf5(file.path(cp_run_path, last_checkpoint), compile = FALSE)
# model <- keras::load_weights_model_hdf5(model, file.path(cp_run_path, last_checkpoint))
# optimizer <-  keras::optimizer_adam(lr = 0.01)
# model %>% keras::compile(loss = "categorical_crossentropy", optimizer = optimizer, metrics = c("acc"))
#
# # continue training
# trainNetwork(train_type = "lm",
#              model = model,
#              path = train_path_1,
#              path.val = validation_path_1,
#              steps.per.epoch = 5,
#              batch.size = 8,
#              epochs = 2,
#              run.name = "continue_from_checkpoint",
#              output_format = "target_right"
# )
```