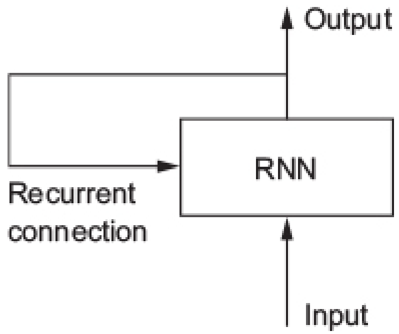
Recurrent NNs (RNNs)

PUBH 7475/8475

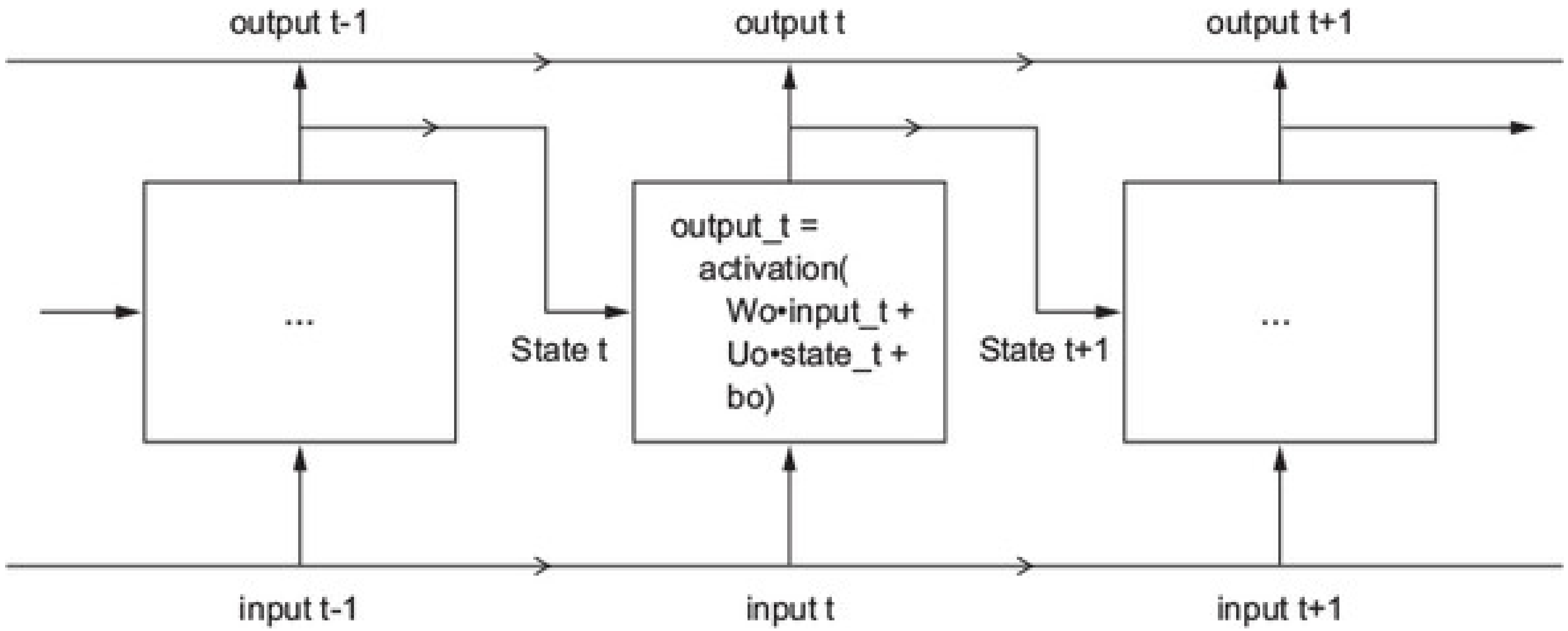
Based on “Deep Learning wth R”
https://www.manning.com/books/deep-learning-with-r
(including source code)




```
state_t <- 0
for (input_t in input_sequence) {
  output_t <- activation(dot(W, input_t) + dot(U, state_t) + b)
  state_t <- output_t
}
```

```
timesteps <- 100
input_features <- 32
output_features <- 64
random_array <- function(dim) {
  array(runif(prod(dim)), dim = dim)
}
inputs <- random_array(dim = c(timesteps, input_features))
state_t <- rep_len(0, length = c(output_features))
W <- random_array(dim = c(output_features, input_features))
U <- random_array(dim = c(output_features, output_features))
b <- random_array(dim = c(output_features, 1))
output_sequence <- array(0, dim = c(timesteps, output_features))
for (i in 1:nrow(inputs)) {
  input_t <- inputs[i,]
  output_t <- tanh(as.numeric((W %*% input_t) + (U %*% state_t) + b))
  output_sequence[i,] <- as.numeric(output_t)
  state_t <- output_t
}
```

Figure 6.10. The starting point of an LSTM layer: a simple RNN



Simple RNN:

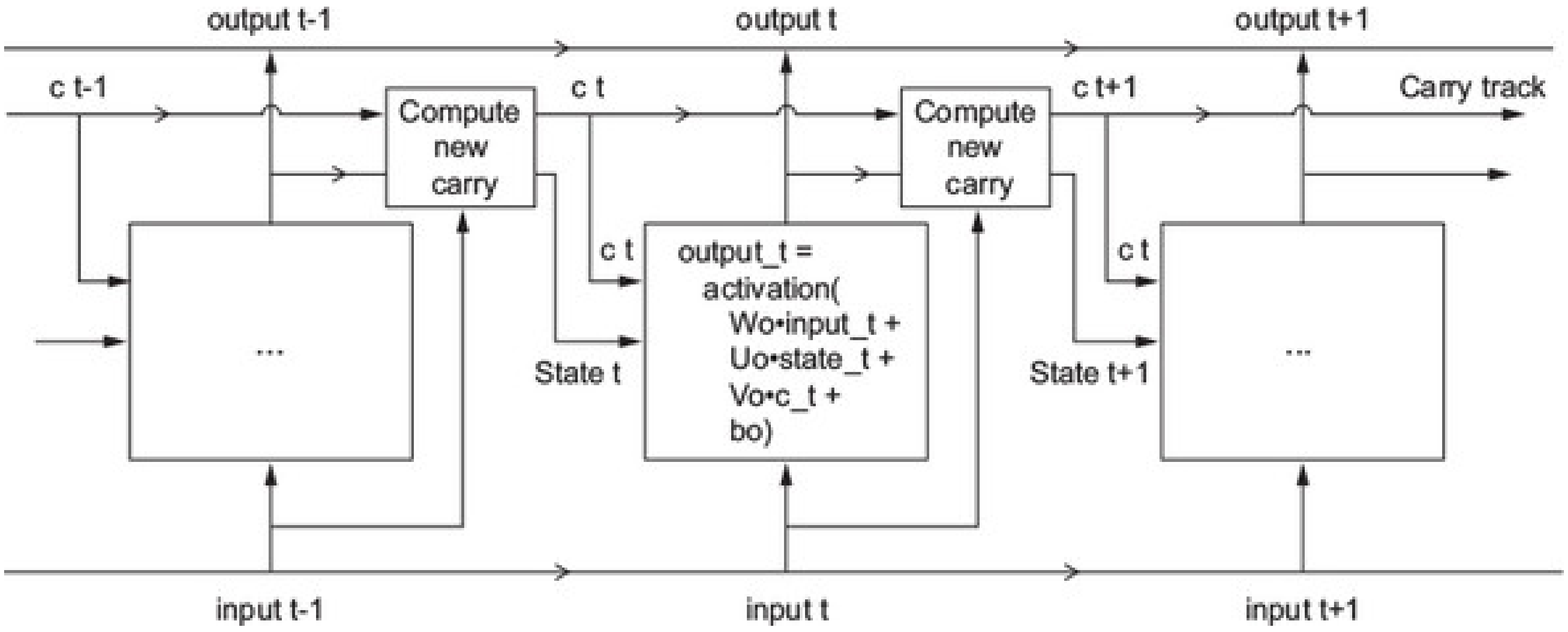
```
 output_t <- activation(dot(W, input_t) + dot(U, state_t) + b)
 state_t+1 <- output_t
```

Does not have memory of long-range dependence➔ Long Short-Term Memory (LSTM):

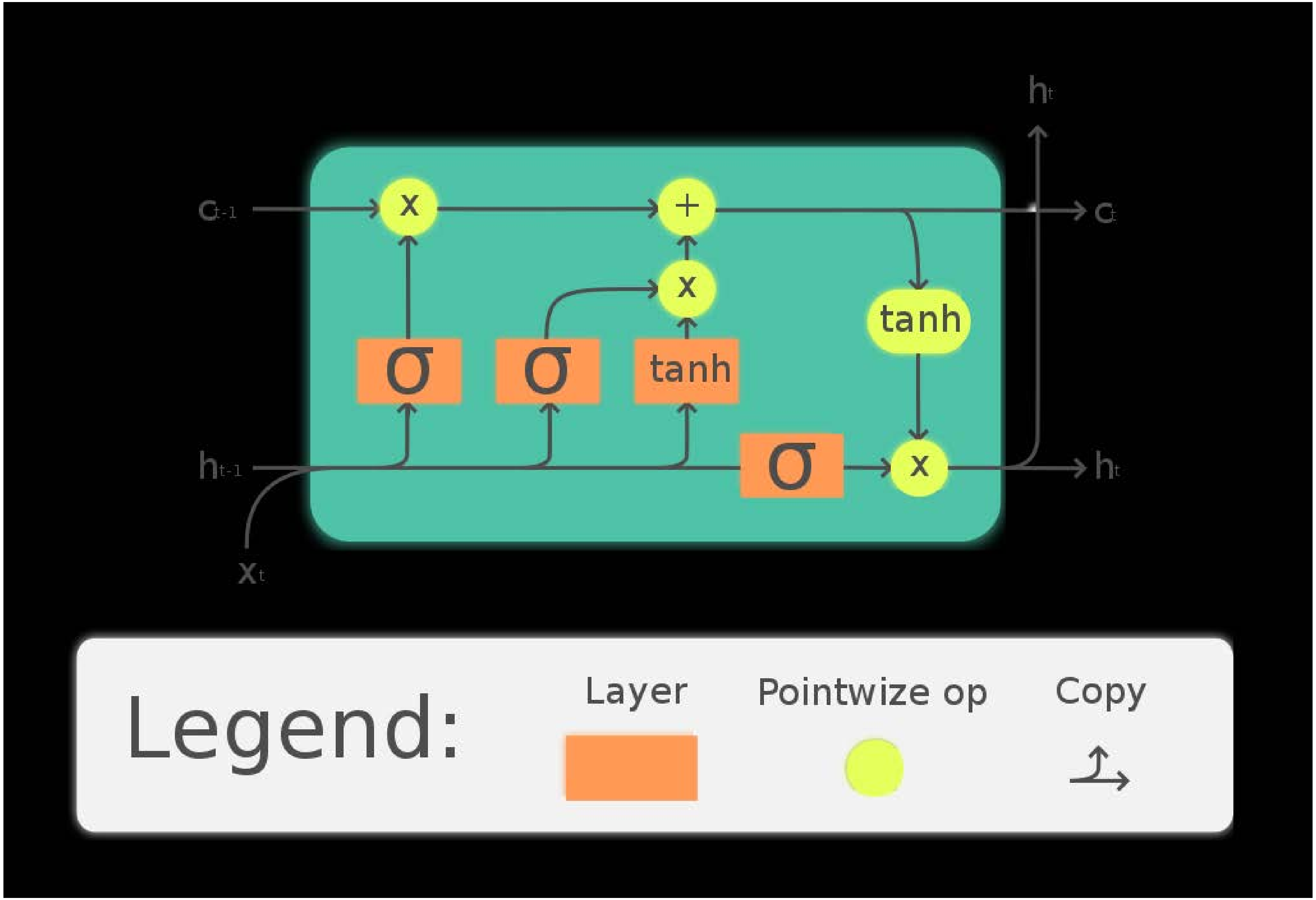
```
output_t = activation(dot(state_t, Uo) + dot(input_t, Wo) + bo)

i_t = activation(dot(state_t, Ui) + dot(input_t, Wi) + bi)
f_t = activation(dot(state_t, Uf) + dot(input_t, Wf) + bf)
cc_t = tanh(dot(state_t, Uk) + dot(input_t, Wk) + bc)

c_t = i_t * cc_t + c_t-1 * f_t
state_t+1=tanh(c_t)*output_t
```

activation: usually sigmoid

Figure 6.12. Anatomy of an LSTM



Wikipedia:



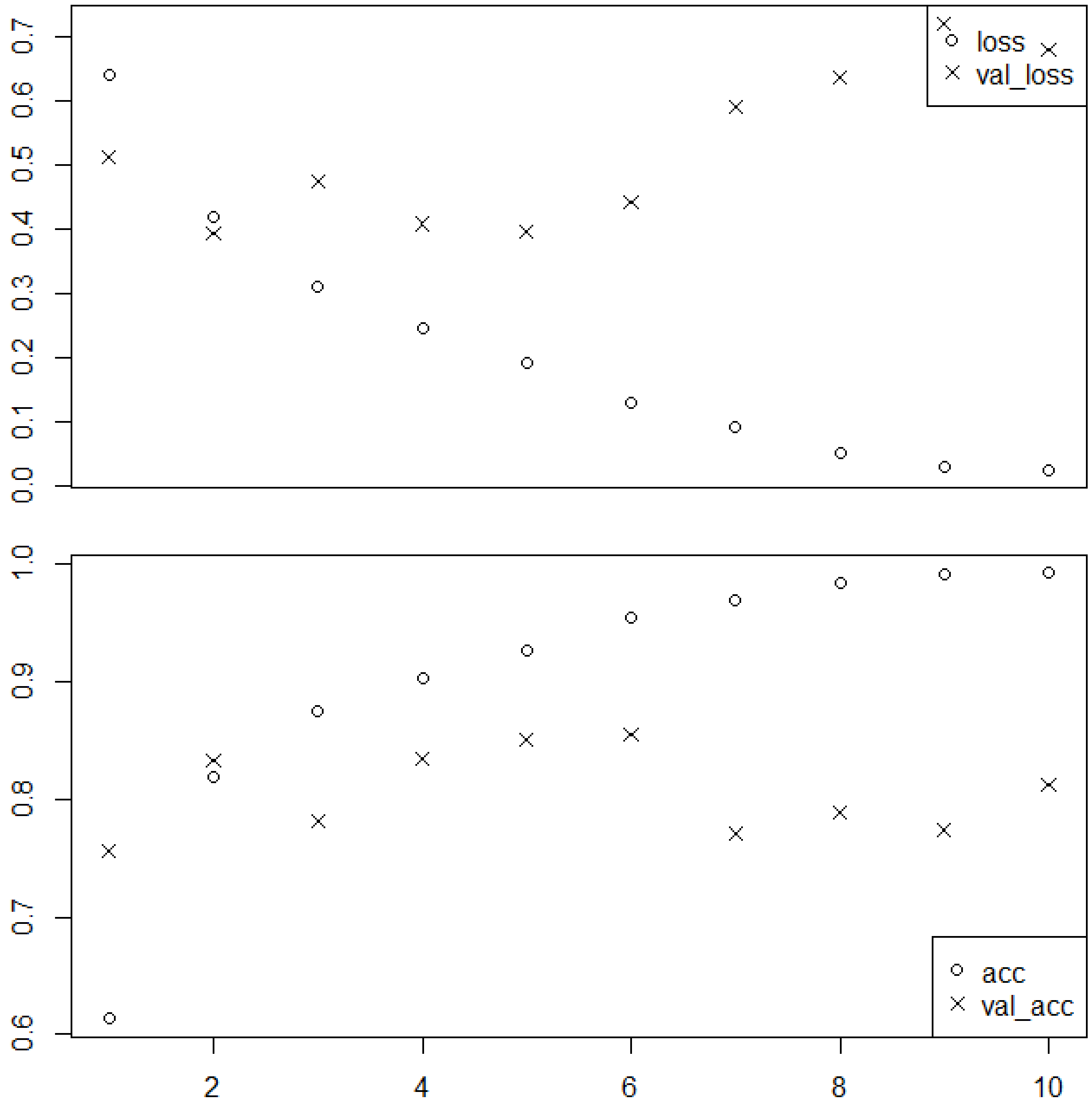
c_{t-1}
x
+
x
tanh
σ
σ
tanh
h_t
c_t
h_{t-1}
σ
x
h_t
x_t
Legend:
Layer
Pointwize op
Copy

```
library(keras)
model <- keras_model_sequential() %>%
  layer_embedding(input_dim = 10000, output_dim = 32) %>%
  layer_simple_rnn(units = 32)

> summary(model)
________________________________________________________________________
Layer (type)                 Output Shape        Param #
========================================================================
embedding_22 (Embedding)       (None, None, 32)     320000
________________________________________________________________________
simplernn_10 (SimpleRNN)       (None, 32)           2080
========================================================================
Total params: 322,080
Trainable params: 322,080
Non-trainable params: 0

# 320000 =10000*32
#Q: why 2080?  = (32+32+1)*32
```

```r
library(keras)
max_features <- 10000
maxlen <- 500

cat("Loading data...\n")
imdb <- dataset_imdb(num_words = max_features)
c(c(input_train, y_train), c(input_test, y_test)) %<-% imdb
cat(length(input_train), "train sequences\n")
#25000 train sequences
cat(length(input_test), "test sequences")
#25000 test sequences
cat("Pad sequences (samples x time)\n")
input_train <- pad_sequences(input_train, maxlen = maxlen)
input_test <- pad_sequences(input_test, maxlen = maxlen)
cat("input_train shape:", dim(input_train), "\n")
#input_train shape: 25000 500
cat("input_test shape:", dim(input_test), "\n")
#input_test shape: 25000 500
```

```r
model <- keras_model_sequential() %>%
  layer_embedding(input_dim = max_features, output_dim = 32) %>%
  layer_simple_rnn(units = 32) %>%
  layer_dense(units = 1, activation = "sigmoid")

model %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("acc")
)

history <- model %>% fit(
  input_train, y_train,
  epochs = 10,
  batch_size = 128,
  validation_split = 0.2
)
```

```
> model
Model
Model: "sequential_5"
________________________________________________________________________________
Layer (type)                        Output Shape                   Param #
================================================================================
embedding_2 (Embedding)              (None, None, 32)               320000
________________________________________________________________________________
simple_rnn_1 (SimpleRNN)            (None, 32)                     2080
________________________________________________________________________________
dense_9 (Dense)                     (None, 1)                      33
================================================================================
Total params: 322,113
Trainable params: 322,113
Non-trainable params: 0
```

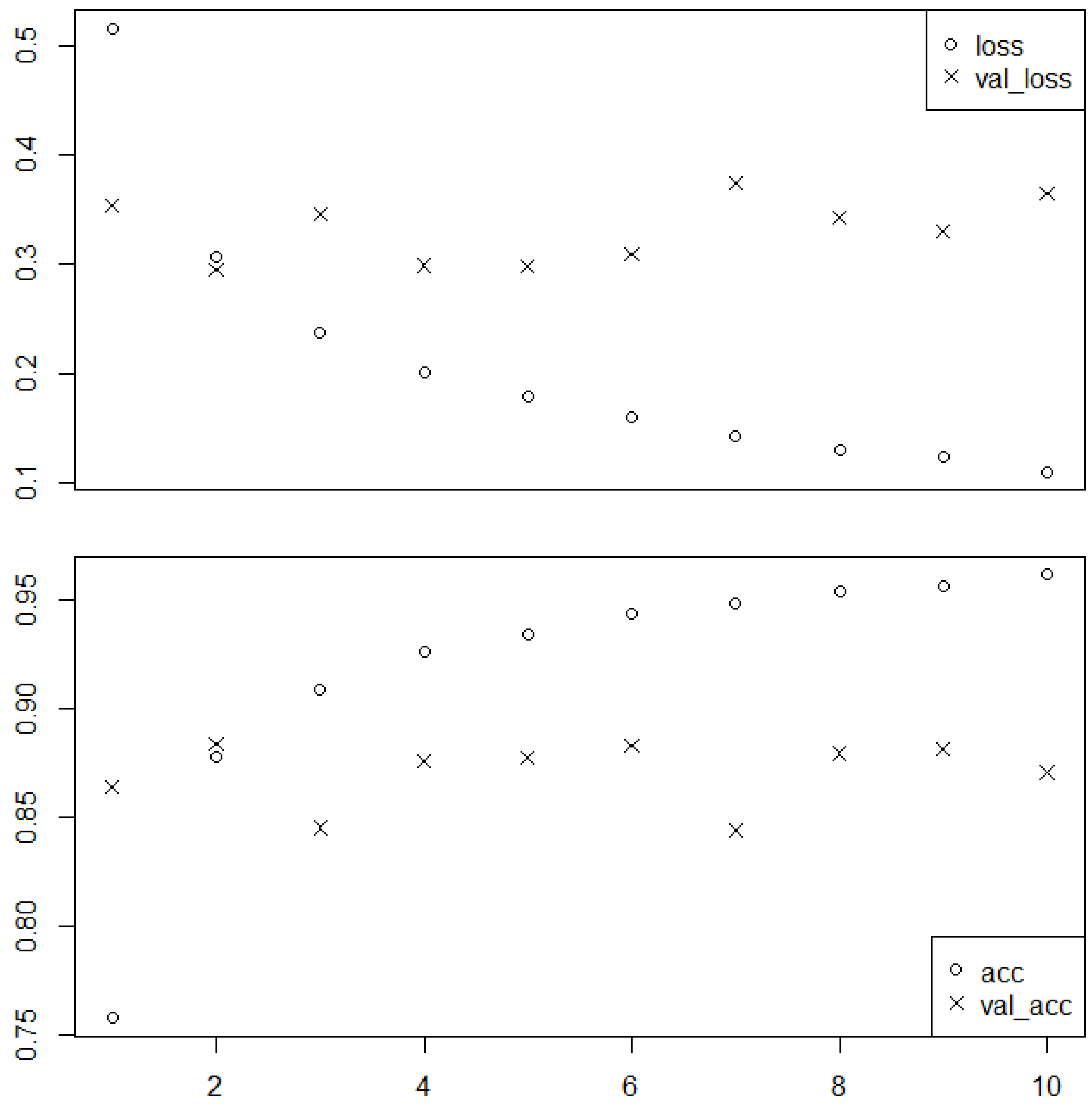


0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
loss
val_loss
0.6
0.7
0.8
0.9
1.0
acc
val_acc
2
4
6
8
10

```
model <- keras_model_sequential() %>%
  layer_embedding(input_dim = max_features, output_dim = 32) %>%
  layer_lstm(units = 32) %>%
  layer_dense(units = 1, activation = "sigmoid")

model %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("acc")
)

history <- model %>% fit(
  input_train, y_train,
  epochs = 10,
  batch_size = 128,
  validation_split = 0.2
)
```

```
> model
Model
Model: "sequential_6"
________________________________________________________________________________
Layer (type)                    Output Shape                 Param #
================================================================================
embedding_3 (Embedding)         (None, None, 32)             320000
________________________________________________________________________________
lstm_1 (LSTM)                   (None, 32)                   8320
________________________________________________________________________________
dense_10 (Dense)                (None, 1)                    33
================================================================================
Total params: 328,353
Trainable params: 328,353
Non-trainable params: 0
```

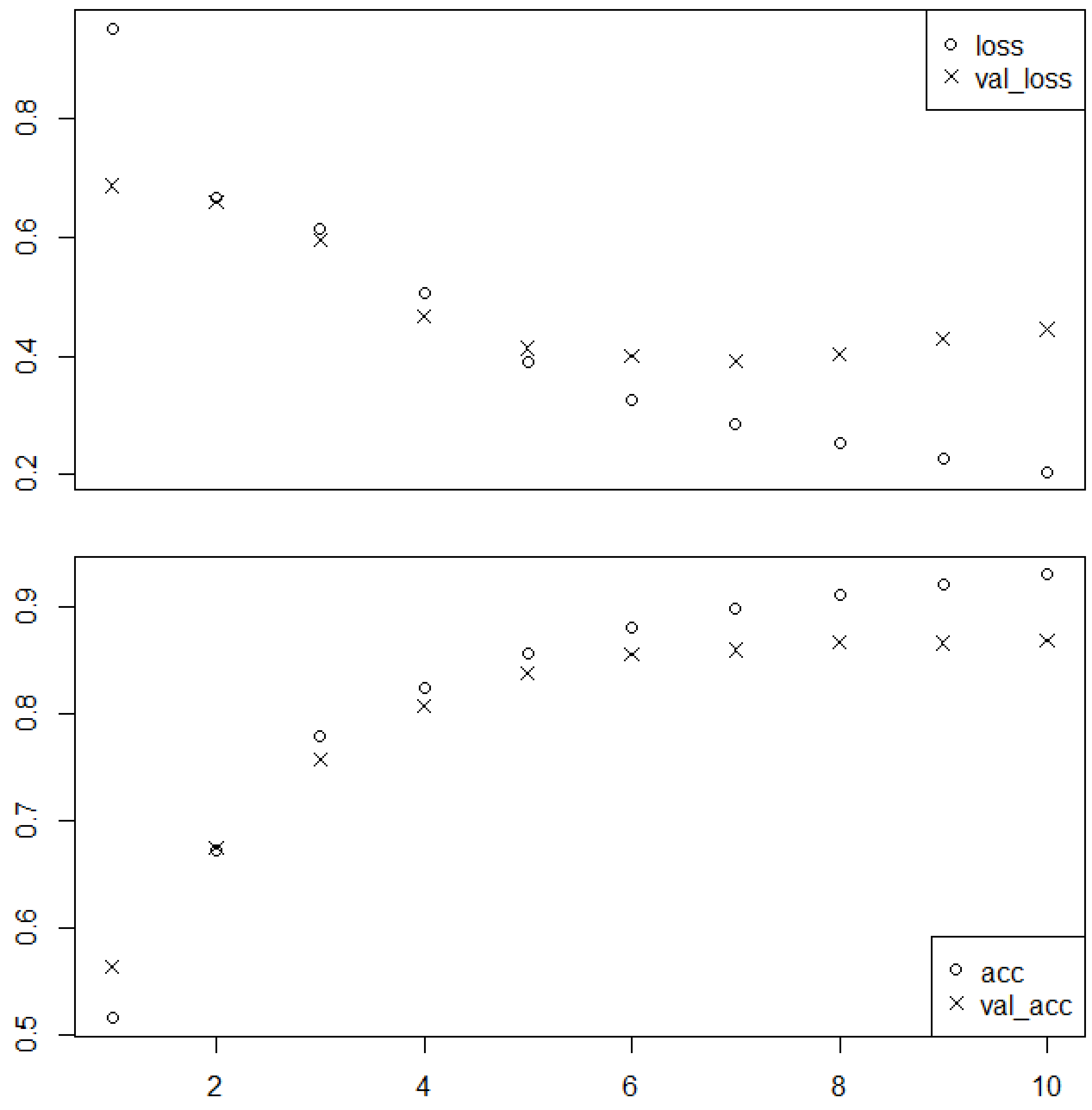
#Q: why 8320? =(32+32+1)*4*32



loss
val_loss
acc
val_acc
0.1
0.2
0.3
0.4
0.5
0.75
0.80
0.85
0.90
0.95
2
4
6
8
10

```
##1D CNN:
model <- keras_model_sequential() %>%
  layer_embedding(input_dim = max_features, output_dim = 128,
                  input_length = maxlen) %>%
  layer_conv_1d(filters = 32, kernel_size = 7, activation = "relu") %>%
  layer_max_pooling_1d(pool_size = 5) %>%
  layer_conv_1d(filters = 32, kernel_size = 7, activation = "relu") %>%
  layer_global_max_pooling_1d() %>%
  layer_dense(units = 1)
summary(model)
model %>% compile(
  optimizer = optimizer_rmsprop(lr = 1e-4),
  loss = "binary_crossentropy",
  metrics = c("acc")
)
history <- model %>% fit(
  input_train, y_train,
  epochs = 10,
  batch_size = 128,
  validation_split = 0.2
)
```

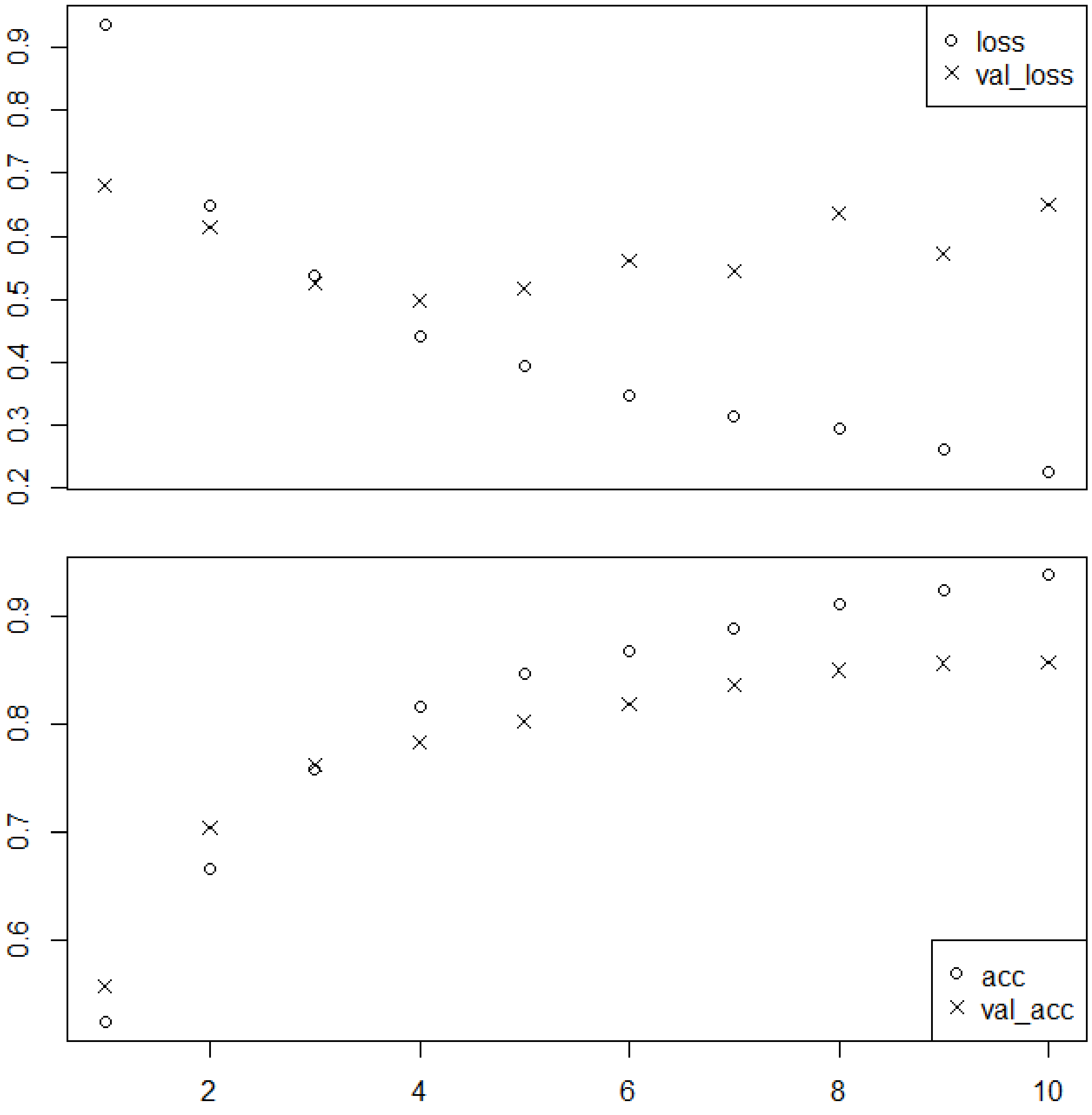
```
> summary(model)
Model: "sequential_6"
________________________________________________________________________________
Layer (type)                         Output Shape                   Param #
================================================================================
embedding_6 (Embedding)              (None, 500, 128)               1280000
________________________________________________________________________________
conv1d_3 (Conv1D)                    (None, 494, 32)                28704
________________________________________________________________________________
max_pooling1d_1 (MaxPooling1D)       (None, 98, 32)                 0
________________________________________________________________________________
conv1d_2 (Conv1D)                    (None, 92, 32)                 7200
________________________________________________________________________________
global_max_pooling1d_1 (GlobalMaxPooling1D)   (None, 32)            0
________________________________________________________________________________
dense_15 (Dense)                     (None, 1)                      33
================================================================================
Total params: 1,315,937
Trainable params: 1,315,937
Non-trainable params: 0
```



loss
val_loss
acc
val_acc
0.2
0.4
0.6
0.8
0.5
0.6
0.7
0.8
0.9
2
4
6
8
10

```
##1D CNN + RNN:
model <- keras_model_sequential() %>%
  layer_embedding(input_dim = max_features, output_dim = 128,
                  input_length = maxlen) %>%
  layer_conv_1d(filters = 32, kernel_size = 7, activation = "relu") %>%
  layer_max_pooling_1d(pool_size = 5) %>%
  layer_conv_1d(filters = 32, kernel_size = 7, activation = "relu") %>%
  layer_gru(units=32) %>%
  layer_dense(units = 1)
summary(model)
model %>% compile(
  optimizer = optimizer_rmsprop(lr = 1e-4),
  loss = "binary_crossentropy",
  metrics = c("acc")
)
history <- model %>% fit(
  input_train, y_train,
  epochs = 10,
  batch_size = 128,
  validation_split = 0.2
)
```

```
> summary(model)
Model: "sequential_7"
________________________________________________________________________________
Layer (type)                          Output Shape                    Param #
================================================================================
embedding_7 (Embedding)               (None, 500, 128)                1280000
________________________________________________________________________________
conv1d_5 (Conv1D)                     (None, 494, 32)                 28704
________________________________________________________________________________
max_pooling1d_2 (MaxPooling1D)        (None, 98, 32)                  0
________________________________________________________________________________
conv1d_4 (Conv1D)                     (None, 92, 32)                  7200
________________________________________________________________________________
gru (GRU)                             (None, 32)                      6240
________________________________________________________________________________
dense_16 (Dense)                      (None, 1)                       33
================================================================================
Total params: 1,322,177
Trainable params: 1,322,177
Non-trainable params: 0
```



loss
val_loss
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
acc
val_acc
0.6
0.7
0.8
0.9
2
4
6
8
10

- GRU (Gated Recurrent Unit): similar to LSTM; less computationally demanding and less powerful?
  - R: layer_gru(units = 32, …)
- Can use dropout

```
layer_gru(units = 32, dropout = 0.1, recurrent_dropout = 0.5)
```

- Using bidirectional RNNs:

```
bidirectional(
    layer_lstm(units = 32)
    )
```

- RNNs: expensive/difficult to train
  - Vanishing/exploding gradients for simple RNNs
- For short seqs, CNNs or FFNs may work better (and being cheaper)!
- Applications:
  - article/music/image generation,
  - Google Neural Machine Translation (GNMT),

    Microsoft: Awadalla et al (2018). Achieving Human Parity on Automatic Chinese to English News Translation. arXiv:1803.05567
  - image captioning, …
- YouTube: "LSTM is dead. Long Live Transformers!" (for NLP)
- Generative Pre-trained Transformer:

  Oct 28, 2019: "OpenAI's GPT2 Now Writes Scientific Paper Abstracts"

  https://interestingengineering.com/openais-gpt2-now-writes-scientific-paper-abstracts

  Now GPT3: …175 billion parameters… "warned of GPT-3's potential dangers"…check out at Wikipedia

Quantitative Biology 2019, 7(2): 122–137 https://doi.org/10.1007/s40484-019-0154-0

RESEARCH ARTICLE

Predicting enhancer-promoter interaction from genomic sequence with deep neural networks

Shashank Singh[1], Yang Yang[2], Barnabás Póczos[1], Jian Ma[2,*]

[1] Machine Learning Department, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA
[2] Computational Biology Department, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA
* Correspondence: jianma@cs.cmu.edu

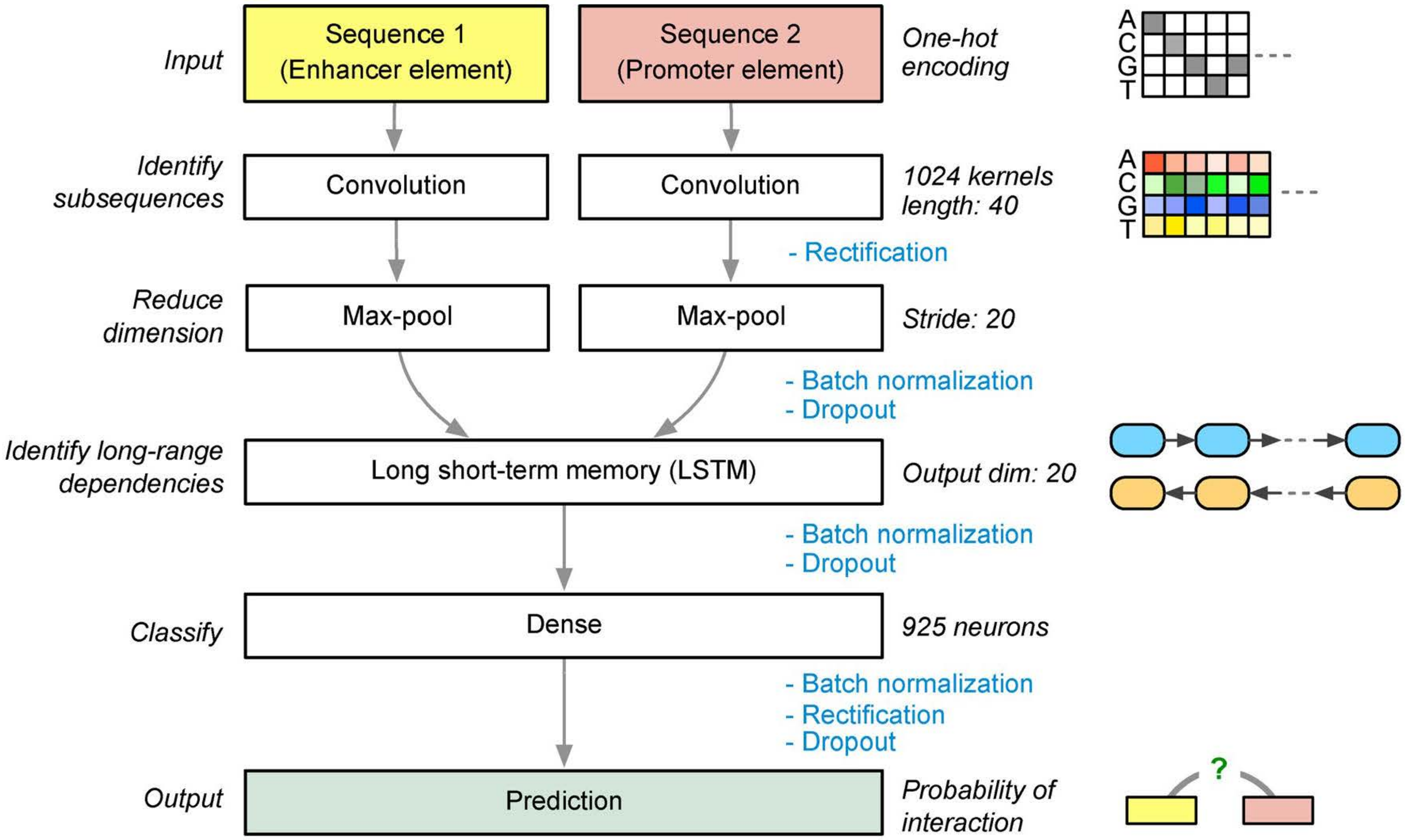


Figure 1. Diagram of our deep learning model SPEID to predict enhancer-promoter interactions based on sequences only. Key steps involving rectification, batch normalization, and dropout are annotated. Note that the final output step is essentially a logistic regression in SPEID which provides a probability to indicate whether the input enhancer element and promoter element would interact.

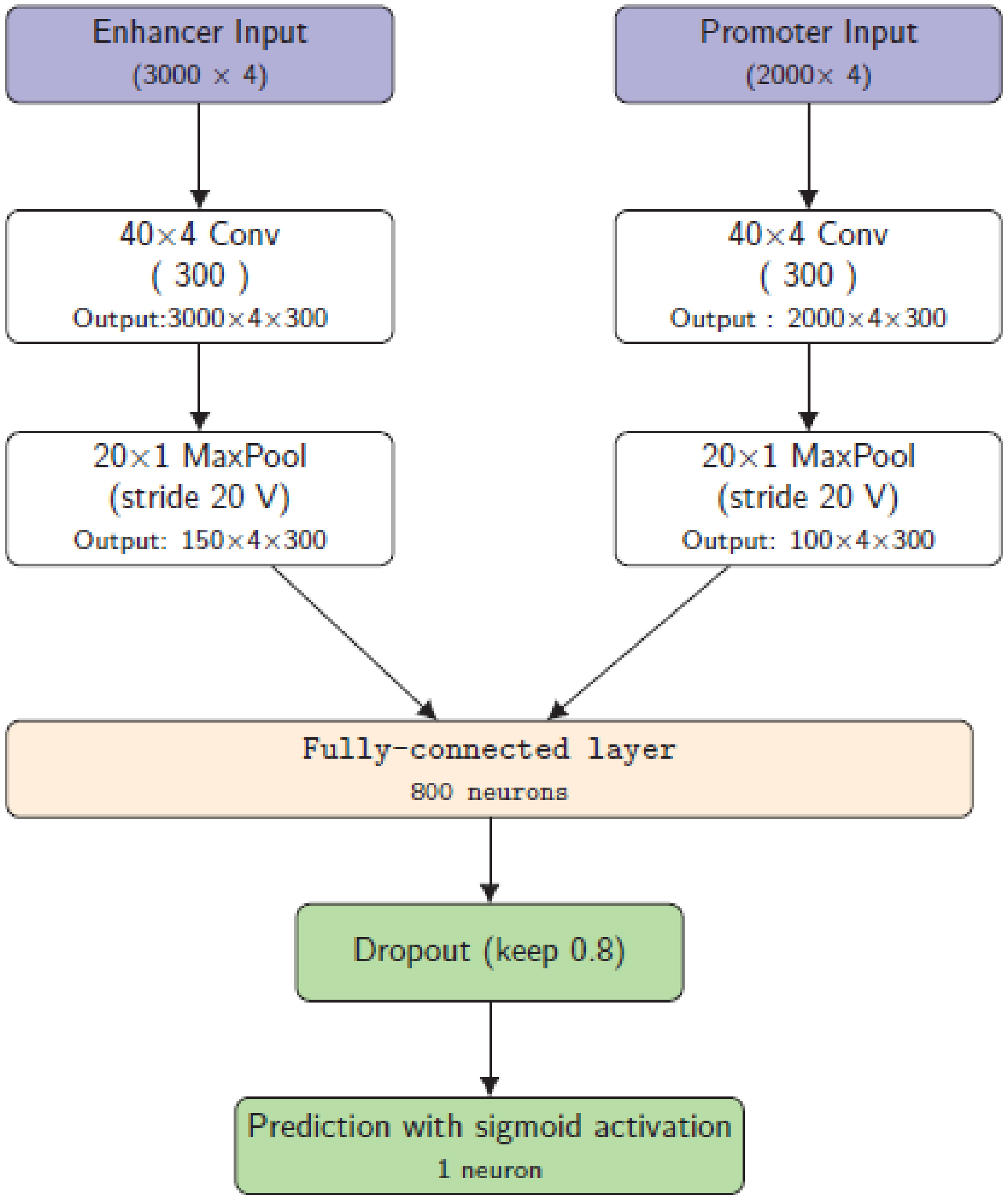
Bioinformatics, 35(17), 2019, 2899–
2906 doi:
10.1093/bioinformatics/bty1050
Advance Access Publication Date: 14 January 2019
Original Paper

Genome analysis

A simple convolutional neural network for prediction of enhancer–promoter interactions with DNA sequence data

Zhong Zhuang[1], Xiaotong Shen[2] and Wei Pan[3,*]

[1]Department of Electrical and Computer Engineering, [2]School of Statistics and [3]Division of Biostatistics, University of Minnesota, Minneapolis, MN 55455, USA



Enhancer Input
(3000 × 4)
Promoter Input
(2000× 4)
40×4 Conv
(300)
Output:3000×4×300
40×4 Conv
(300)
Output : 2000×4×300
20×1 MaxPool
(stride 20 V)
Output: 150×4×300
20×1 MaxPool
(stride 20 V)
Output: 100×4×300
Fully-connected layer
800 neurons
Dropout (keep 0.8)
Prediction with sigmoid activation
1 neuron

genes

Article

Local Epigenomic Data are more Informative than Local Genome Sequence Data in Predicting Enhancer-Promoter Interactions Using Neural Networks

Mengli Xiao [1], Zhong Zhuang [2] and Wei Pan [1,*]

1 Division of Biostatistics, University of Minnesota, Minneapolis, MN 55455, USA; xiaox345@umn.edu
2 Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, MN 55455, USA; zhuan143@umn.edu
* Correspondence: panxx014@umn.edu; Tel.: +01-612-626-2705

Received: 29 November 2019; Accepted: 26 December 2019; Published: 29 December 2019