ЗВІТ З ЛАБОРАТОРНОЇ РОБОТИ №4
З ДИСЦИПЛІНИ «СТАТИСТИЧНІ АЛГОРИТМИ НАВЧАННЯ»
РЕКУРЕНТНІ НЕЙРОННІ МЕРЕЖІ

Виконав студент 2к. маг. Ломако Олександр

Розглянемо датасет imdb бібліотеки keras, що містить 50 000 відгуків з популярного однойменного сайту кіно. Відгуки можуть мати лише два значення: позитивні і негативні. Нашею задачею є побудувати таку нейронну мережу, яка би змогла за відгуком «розпізнати» який це саме відгук на фільм: позитивний чи негативний.

Почнемо з підключення бібліотек і завантаження датасету.

```
> # підключаємо бібліотеки
> library(keras)
> library(tensorflow)
```

Одразу варто зауважити, що в даній бібліотеці датасет повністю готовий до використання: всі відгуки, а точніше всі слова в них є закодованими в певному словнику (який можна подивитись в datastet_imbd_word_index), і нам для коректної роботи нейронних мереж потрібно обмежити, що ми для аналізу використовуватимемо топ-10 000 слів, і максимальну довжину відгуку покладемо 1 000.

```
> num_words <- 10000
> max_length <- 1000
> data <- dataset_imdb(num_words = num_words) # зчитуємо дані
> c(c(x_train, y_train), c(x_test, y_test)) %<-% data
```

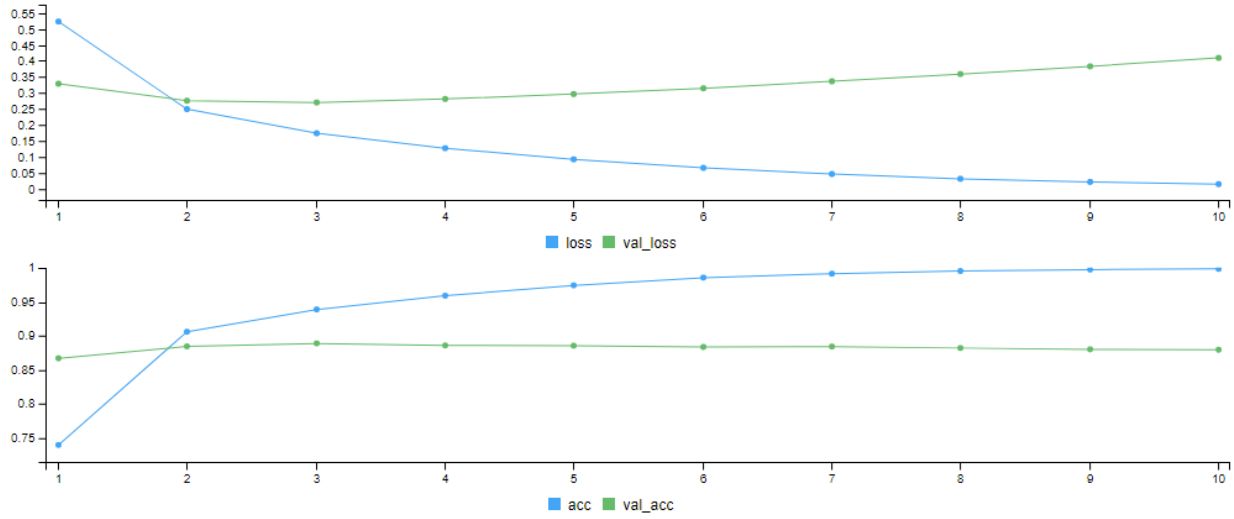
Далі відповідно розбиваємо датасет на тренувальну і тестову частини, і виходячи з нашого обмеження max_length, маємо привести всі повідомлення до одного формату.

```
> x_train <- pad_sequences(x_train, maxlen = max_length)
> x_test <- pad_sequences(x_test, maxlen = max_length)
```

Почнемо з тренування власного вкладення, тобто embedding. Оберемо розмірність латентного простору рівною для початку 8. Мережу задамо з трьох шарів: шару ембедінгу, шару так званого згладжування і щільного шару, який і видаватиме нам відгук.

```
> model <- keras_model_sequential() %>%
+   layer_embedding(input_dim = num_words, output_dim = 8, input_length =  ma
x_length) %>%
+   layer_flatten() %>%
+   layer_dense(units = 1, activation = 'sigmoid')
> model %>% compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics
= c('acc'))
> model %>% fit(x_train, y_train, epochs = 10, batch_size = 32, validation_sp
lit = 0.2)
Epoch 1/10
625/625 [==============================] - 3s 5ms/step - loss: 0.5234 - acc:
0.7391 - val_loss: 0.3289 - val_acc: 0.8672
Epoch 2/10
625/625 [==============================] - 2s 3ms/step - loss: 0.2494 - acc:
0.9064 - val_loss: 0.2757 - val_acc: 0.8848
Epoch 3/10
```

```
625/625 [==============================] - 2s 3ms/step - loss: 0.1740 - acc:
0.9394 - val_loss: 0.2702 - val_acc: 0.8892
Epoch 4/10
625/625 [==============================] - 2s 3ms/step - loss: 0.1272 - acc:
0.9597 - val_loss: 0.2812 - val_acc: 0.8862
Epoch 5/10
625/625 [==============================] - 2s 4ms/step - loss: 0.0924 - acc:
0.9750 - val_loss: 0.2966 - val_acc: 0.8858
Epoch 6/10
625/625 [==============================] - 2s 3ms/step - loss: 0.0657 - acc:
0.9865 - val_loss: 0.3146 - val_acc: 0.8840
Epoch 7/10
625/625 [==============================] - 2s 3ms/step - loss: 0.0465 - acc:
0.9922 - val_loss: 0.3365 - val_acc: 0.8846
Epoch 8/10
625/625 [==============================] - 2s 3ms/step - loss: 0.0315 - acc:
0.9962 - val_loss: 0.3592 - val_acc: 0.8824
Epoch 9/10
625/625 [==============================] - 2s 3ms/step - loss: 0.0217 - acc:
0.9983 - val_loss: 0.3833 - val_acc: 0.8804
Epoch 10/10
625/625 [==============================] - 2s 3ms/step - loss: 0.0148 - acc:
0.9993 - val_loss: 0.4097 - val_acc: 0.8798
```



І подивимось на точність на тренувальних даних.

```
> model %>% evaluate(x_test, y_test)
782/782 [==============================] - 1s 753us/step - loss: 0.4206 - acc
: 0.8672
     loss       acc
0.4206299 0.8671600
```

Отримали точність в 86%, що не є гарним результатом роботи моделі. Також не на користь моделі може свідчити поведінка валідаційної точності, і вигляд валідаційної функції втрат (ми при тренуванні моделі задали параметр validation_split = 0.2). Беззаперечно, можна пошукати модель покраще.

Спробуємо для початку змінити розмірність латетного простору, а саме збільшити, наприклад, в два рази до 16.

```
> model <- keras_model_sequential() %>%
+   layer_embedding(input_dim = num_words, output_dim = 16, input_length =  m
ax_length) %>%
+   layer_flatten() %>%
+   layer_dense(units = 1, activation = 'sigmoid')
```

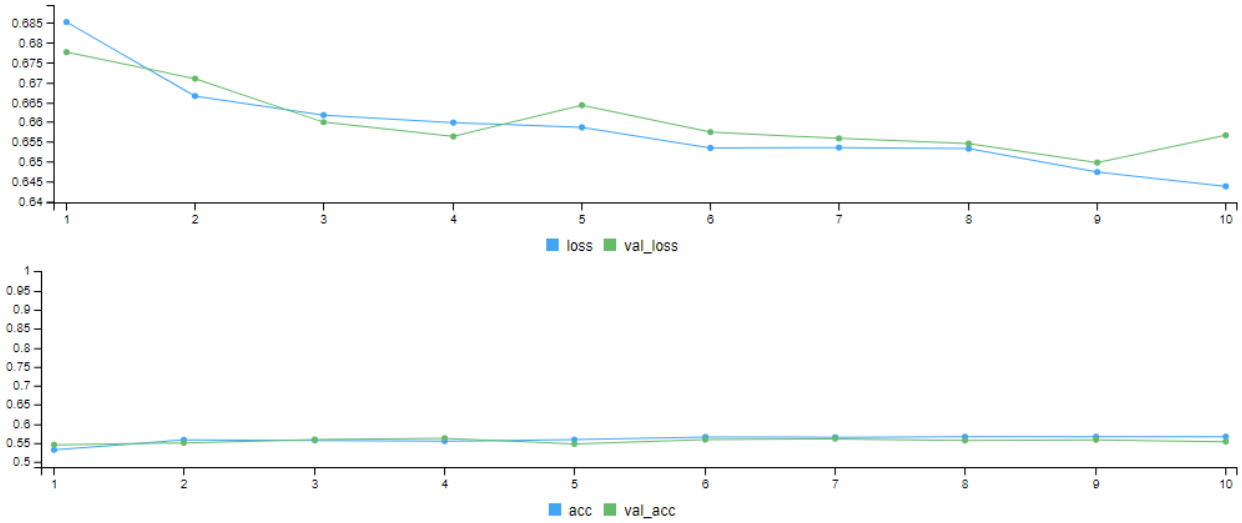
```
> model %>% compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics
= c('acc'))
> model %>% fit(x_train, y_train, epochs = 10, batch_size = 32, validation_sp
lit = 0.2)
Epoch 1/10
625/625 [==============================] - 3s 5ms/step - loss: 0.4867 - acc:
0.7556 - val_loss: 0.3035 - val_acc: 0.8740
Epoch 2/10
625/625 [==============================] - 3s 5ms/step - loss: 0.2224 - acc:
0.9163 - val_loss: 0.2717 - val_acc: 0.8888
Epoch 3/10
625/625 [==============================] - 3s 5ms/step - loss: 0.1462 - acc:
0.9519 - val_loss: 0.2808 - val_acc: 0.8860
Epoch 4/10
625/625 [==============================] - 3s 5ms/step - loss: 0.0964 - acc:
0.9735 - val_loss: 0.2989 - val_acc: 0.8868
Epoch 5/10
625/625 [==============================] - 3s 5ms/step - loss: 0.0610 - acc:
0.9873 - val_loss: 0.3299 - val_acc: 0.8818
Epoch 6/10
625/625 [==============================] - 3s 5ms/step - loss: 0.0369 - acc:
0.9941 - val_loss: 0.3542 - val_acc: 0.8802
Epoch 7/10
625/625 [==============================] - 3s 5ms/step - loss: 0.0218 - acc:
0.9980 - val_loss: 0.3827 - val_acc: 0.8790
Epoch 8/10
625/625 [==============================] - 3s 5ms/step - loss: 0.0131 - acc:
0.9992 - val_loss: 0.4110 - val_acc: 0.8764
Epoch 9/10
625/625 [==============================] - 3s 5ms/step - loss: 0.0082 - acc:
0.9998 - val_loss: 0.4421 - val_acc: 0.8776
Epoch 10/10
625/625 [==============================] - 3s 5ms/step - loss: 0.0050 - acc:
0.9999 - val_loss: 0.4653 - val_acc: 0.8762
> model %>% evaluate(x_test, y_test)
782/782 [==============================] - 1s 1ms/step - loss: 0.4579 - acc:
0.8694
     loss       acc
0.4578718 0.8693600
```

На тренувальних даних точність досягнулася чи не в 100%, а валідаційна, і, власне кажучи, точність на тестових даних коливається в районі 87%. Ні, такій моделі властиве перенавчання, і хоч ми трішки за рахунок збільшення розмірності латетного простору і збільшили точність, але це явно не та модель, яку ми чекали.

Далі побудуємо рекурентну нейронну мережу. Архітектура буде схожою з минулою, але тепер замість шару згладжування додамо шар рекуретної нейронної мережі з 32 нейронами.

```
> model <- keras_model_sequential() %>%
+   layer_embedding(input_dim = num_words, output_dim = 8, input_length =  ma
x_length) %>%
+   layer_simple_rnn(units = 32, return_sequences = TRUE) %>%
+   layer_dense(units = 1, activation = 'sigmoid')
> model %>% compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics
= c('acc'))
> model %>% fit(x_train, y_train, epochs = 10, batch_size = 32, validation_sp
lit = 0.2)
Epoch 1/10
625/625 [==============================] - 87s 140ms/step - loss: 0.6852 - ac
c: 0.5307 - val_loss: 0.6776 - val_acc: 0.5444
Epoch 2/10
625/625 [==============================] - 88s 140ms/step - loss: 0.6665 - ac
c: 0.5573 - val_loss: 0.6709 - val_acc: 0.5493
Epoch 3/10
```

```
625/625 [==============================] - 87s 139ms/step - loss: 0.6617 - ac
c: 0.5560 - val_loss: 0.6599 - val_acc: 0.5583
Epoch 4/10
625/625 [==============================] - 86s 138ms/step - loss: 0.6598 - ac
c: 0.5538 - val_loss: 0.6563 - val_acc: 0.5615
Epoch 5/10
625/625 [==============================] - 86s 137ms/step - loss: 0.6586 - ac
c: 0.5578 - val_loss: 0.6642 - val_acc: 0.5464
Epoch 6/10
625/625 [==============================] - 88s 140ms/step - loss: 0.6534 - ac
c: 0.5648 - val_loss: 0.6574 - val_acc: 0.5580
Epoch 7/10
625/625 [==============================] - 86s 138ms/step - loss: 0.6535 - ac
c: 0.5641 - val_loss: 0.6558 - val_acc: 0.5601
Epoch 8/10
625/625 [==============================] - 89s 142ms/step - loss: 0.6532 - ac
c: 0.5653 - val_loss: 0.6545 - val_acc: 0.5558
Epoch 9/10
625/625 [==============================] - 88s 140ms/step - loss: 0.6473 - ac
c: 0.5654 - val_loss: 0.6497 - val_acc: 0.5573
Epoch 10/10
625/625 [==============================] - 89s 142ms/step - loss: 0.6437 - ac
c: 0.5656 - val_loss: 0.6566 - val_acc: 0.5525
```



Отримали точність лише на тренувальних даних 55-56%! Це поганий результат, про що і свідчать в тому числі і валідаційні втрата і точність. Вочевидь, очікувати високої точності на тестових даних не варто.

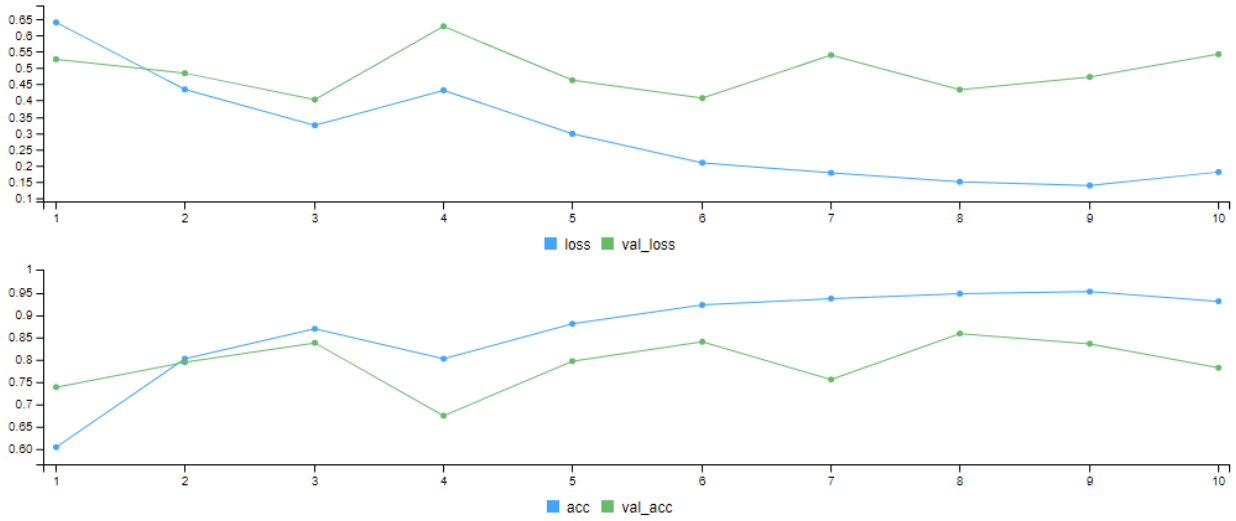
```
> model %>% evaluate(x_test, y_test)
782/782 [==============================] - 20s 25ms/step - loss: 0.6639 - acc
: 0.5484
     loss       acc
0.6638573 0.5483554
```

На тестових даних вдалось досягнути точності лише в 54.83%. Таку мережу беззаперечно треба покращувати.

Спробуємо додати ще один такий же прихований шар.

```
> model <- keras_model_sequential() %>%
+   layer_embedding(input_dim = num_words, output_dim = 8, input_length =  ma
x_length) %>%
+   layer_simple_rnn(units = 32, return_sequences = TRUE) %>%
+   layer_simple_rnn(units = 32) %>%
+   layer_dense(units = 1, activation = 'sigmoid')
> model %>% compile(optimizer = 'Adam', loss = 'binary_crossentropy', metrics
= c('acc'))
```

```
> model %>% fit(x_train, y_train, epochs = 10, batch_size = 32, validation_sp
lit = 0.2)
Epoch 1/10
625/625 [==============================] - 173s 277ms/step - loss: 0.6398 - a
cc: 0.6031 - val_loss: 0.5266 - val_acc: 0.7376
Epoch 2/10
625/625 [==============================] - 173s 277ms/step - loss: 0.4341 - a
cc: 0.8013 - val_loss: 0.4839 - val_acc: 0.7934
Epoch 3/10
625/625 [==============================] - 171s 273ms/step - loss: 0.3240 - a
cc: 0.8684 - val_loss: 0.4028 - val_acc: 0.8370
Epoch 4/10
625/625 [==============================] - 171s 273ms/step - loss: 0.4313 - a
cc: 0.8011 - val_loss: 0.6277 - val_acc: 0.6736
Epoch 5/10
625/625 [==============================] - 171s 273ms/step - loss: 0.2978 - a
cc: 0.8795 - val_loss: 0.4622 - val_acc: 0.7958
Epoch 6/10
625/625 [==============================] - 175s 280ms/step - loss: 0.2089 - a
cc: 0.9219 - val_loss: 0.4073 - val_acc: 0.8394
Epoch 7/10
625/625 [==============================] - 174s 278ms/step - loss: 0.1777 - a
cc: 0.9360 - val_loss: 0.5393 - val_acc: 0.7546
Epoch 8/10
625/625 [==============================] - 175s 279ms/step - loss: 0.1506 - a
cc: 0.9470 - val_loss: 0.4334 - val_acc: 0.8578
Epoch 9/10
625/625 [==============================] - 178s 285ms/step - loss: 0.1391 - a
cc: 0.9517 - val_loss: 0.4724 - val_acc: 0.8348
Epoch 10/10
625/625 [==============================] - 178s 285ms/step - loss: 0.1804 - a
cc: 0.9301 - val_loss: 0.5420 - val_acc: 0.7810
```



Варто відмітити цікаву поведінку не лише валідаційної втрати, а і власне кажучи точності, яка подекуди просідала. У випадку з валідаційною точністю це свідчить про неякісну модель, але в цілому ми при даному тренуванні маємо в принципі вищі точності. Подивимось на тестових даних.

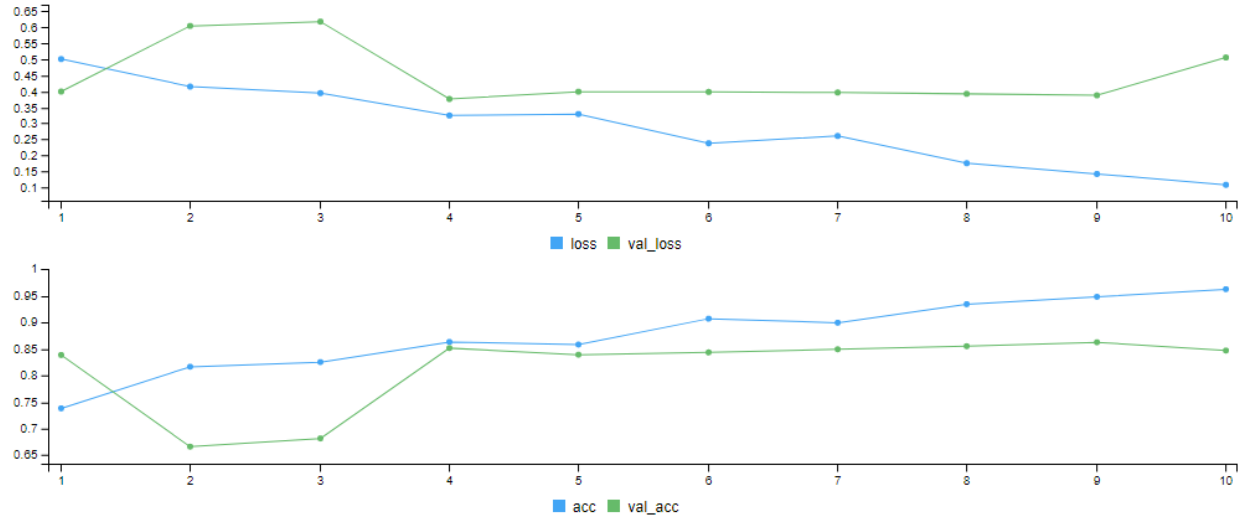
```
> model %>% evaluate(x_test, y_test)
782/782 [==============================] - 38s 49ms/step - loss: 0.5323 - acc
: 0.7827
     loss       acc
0.5323213 0.7827200
```

Точність на тестових даних вийшла рівною 78.27%. Це, звісно, краще, ніж у випадку одного прихованого шару, але це все одно поступається точності при натренованому власному вкладенні.

Як показала минула практика, збільшувати розмірність латентного простору як такого сенсу немає – час обчислень зростає, а точність при цьому підвищується незначно.

Додамо до моделі з простим рекурентним шаром 1 блок GRU.

```
> model <- keras_model_sequential() %>%
+   layer_embedding(input_dim = num_words, output_dim = 8, input_length =  ma
x_length) %>%
+   layer_simple_rnn(units = 32, return_sequences = TRUE) %>%
+   layer_gru(units = 32) %>%
+   layer_dense(units = 1, activation = 'sigmoid')
> model %>% compile(optimizer = 'Adam', loss = 'binary_crossentropy', metrics
= c('acc'))
> model %>% fit(x_train, y_train, epochs = 10, batch_size = 32, validation_sp
lit = 0.2)
Epoch 1/10
625/625 [==============================] - 210s 336ms/step - loss: 0.5003 - a
cc: 0.7379 - val_loss: 0.3992 - val_acc: 0.8386
Epoch 2/10
625/625 [==============================] - 208s 333ms/step - loss: 0.4146 - a
cc: 0.8164 - val_loss: 0.6034 - val_acc: 0.6656
Epoch 3/10
625/625 [==============================] - 209s 334ms/step - loss: 0.3943 - a
cc: 0.8252 - val_loss: 0.6168 - val_acc: 0.6812
Epoch 4/10
625/625 [==============================] - 207s 330ms/step - loss: 0.3246 - a
cc: 0.8632 - val_loss: 0.3760 - val_acc: 0.8516
Epoch 5/10
625/625 [==============================] - 206s 329ms/step - loss: 0.3291 - a
cc: 0.8583 - val_loss: 0.3987 - val_acc: 0.8392
Epoch 6/10
625/625 [==============================] - 206s 329ms/step - loss: 0.2381 - a
cc: 0.9072 - val_loss: 0.3980 - val_acc: 0.8436
Epoch 7/10
625/625 [==============================] - 207s 331ms/step - loss: 0.2609 - a
cc: 0.8995 - val_loss: 0.3962 - val_acc: 0.8494
Epoch 8/10
625/625 [==============================] - 207s 331ms/step - loss: 0.1760 - a
cc: 0.9345 - val_loss: 0.3920 - val_acc: 0.8552
Epoch 9/10
625/625 [==============================] - 207s 331ms/step - loss: 0.1426 - a
cc: 0.9484 - val_loss: 0.3874 - val_acc: 0.8626
Epoch 10/10
625/625 [==============================] - 207s 332ms/step - loss: 0.1087 - a
cc: 0.9628 - val_loss: 0.5052 - val_acc: 0.8470
```

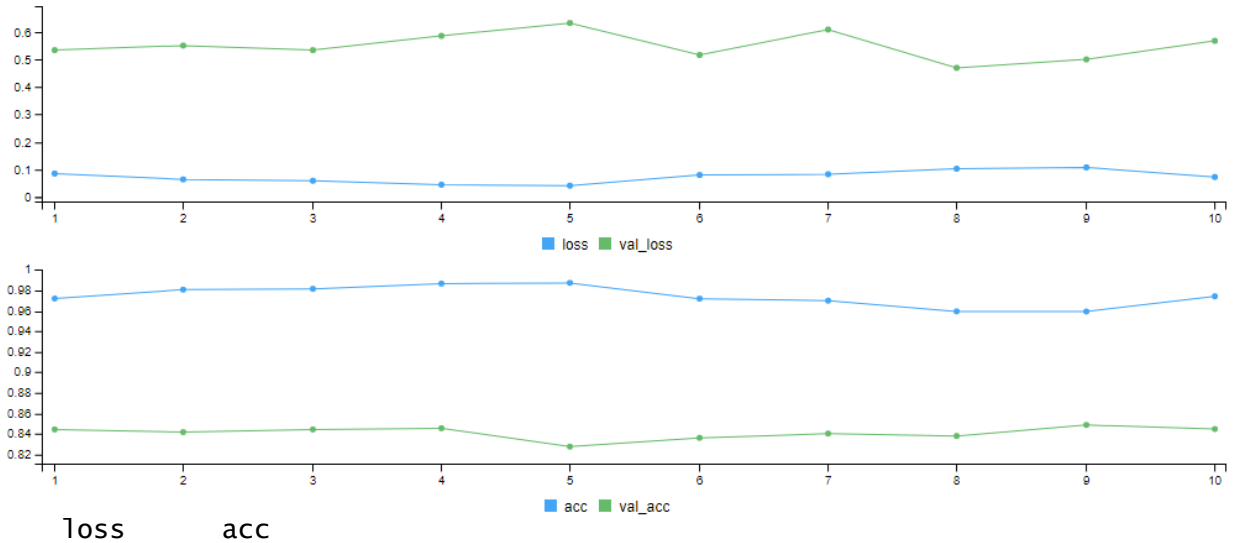



```
> model %>% evaluate(x_test, y_test)
782/782 [==============================] - 54s 69ms/step - loss: 0.5379 - acc
: 0.8353
```

```
   loss     acc
0.53785 0.83528
```

Тут можемо бачити, що таким чином вдалося досягнути точності в 83,5%. Проженемо навчання даної моделі ще раз.

```
> model %>% fit(x_train, y_train, epochs = 10, batch_size = 32, validation_sp
lit = 0.2)
Epoch 1/10
625/625 [==============================] - 224s 359ms/step - loss: 0.0846 - a
cc: 0.9721 - val_loss: 0.5350 - val_acc: 0.8440
Epoch 2/10
625/625 [==============================] - 217s 348ms/step - loss: 0.0634 - a
cc: 0.9808 - val_loss: 0.5516 - val_acc: 0.8414
Epoch 3/10
625/625 [==============================] - 218s 349ms/step - loss: 0.0585 - a
cc: 0.9816 - val_loss: 0.5349 - val_acc: 0.8440
Epoch 4/10
625/625 [==============================] - 209s 334ms/step - loss: 0.0440 - a
cc: 0.9867 - val_loss: 0.5870 - val_acc: 0.8452
Epoch 5/10
625/625 [==============================] - 215s 343ms/step - loss: 0.0405 - a
cc: 0.9874 - val_loss: 0.6339 - val_acc: 0.8272
Epoch 6/10
625/625 [==============================] - 215s 345ms/step - loss: 0.0795 - a
cc: 0.9720 - val_loss: 0.5173 - val_acc: 0.8358
Epoch 7/10
625/625 [==============================] - 210s 336ms/step - loss: 0.0822 - a
cc: 0.9700 - val_loss: 0.6102 - val_acc: 0.8400
Epoch 8/10
625/625 [==============================] - 210s 336ms/step - loss: 0.1024 - a
cc: 0.9595 - val_loss: 0.4704 - val_acc: 0.8376
Epoch 9/10
625/625 [==============================] - 211s 338ms/step - loss: 0.1072 - a
cc: 0.9594 - val_loss: 0.5011 - val_acc: 0.8484
Epoch 10/10
625/625 [==============================] - 214s 342ms/step - loss: 0.0721 - a
cc: 0.9743 - val_loss: 0.5689 - val_acc: 0.8444
> model %>% evaluate(x_test, y_test)
782/782 [==============================] - 55s 70ms/step - loss: 0.6103 - acc
: 0.8347
```

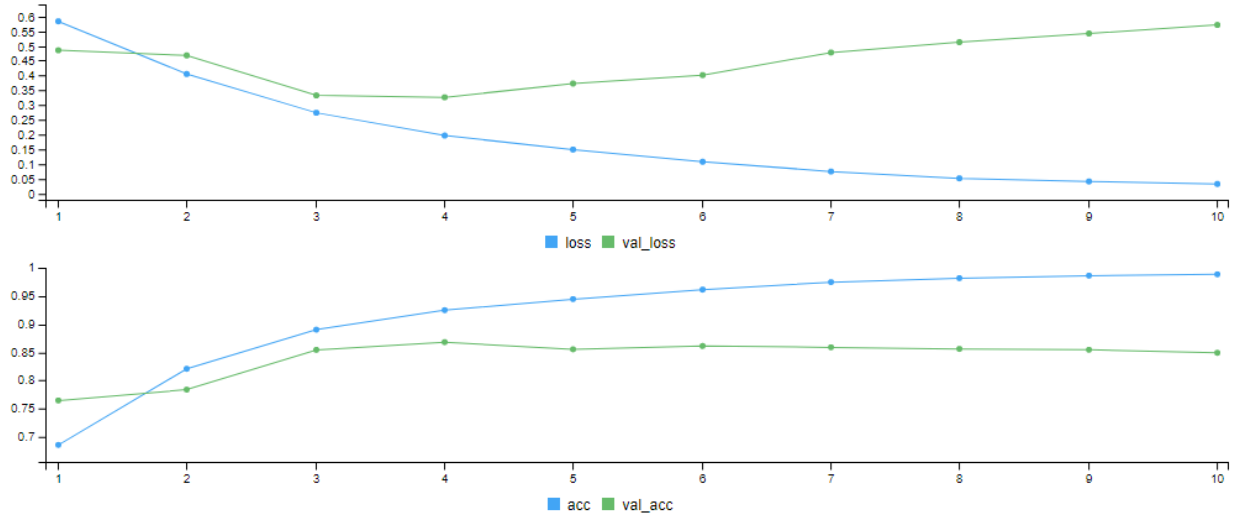



```
    loss      acc
0.610315 0.834720
```

Ні, це явно невдала спроба. Мало того, що понизили точність на тестових даних, так вона навіть на тренувальних в певний момент почала падати. Хоча на тренувальних вона досягала більше 95%. Перенавчання.

Спробуємо змінити деякі параметри моделі. Саме моделі з одним рекурентним шаром і одним блоком GRU. Нехай тепер розмірність латентного простору становить 16, і в кожному прихованому шарі буде 64 нейрони.

```
> model <- keras_model_sequential() %>%
+   layer_embedding(input_dim = num_words, output_dim = 16, input_length =  m
ax_length) %>%
+   layer_simple_rnn(units = 64, return_sequences = TRUE) %>%
+   layer_gru(units = 64) %>%
+   layer_dense(units = 1, activation = 'sigmoid')
> model %>% compile(optimizer = 'Adam', loss = 'binary_crossentropy', metrics
= c('acc'))
> model %>% fit(x_train, y_train, epochs = 10, batch_size = 32, validation_sp
lit = 0.2)
Epoch 1/10
625/625 [==============================] - 276s 442ms/step - loss: 0.5832 - a
cc: 0.6852 - val_loss: 0.4855 - val_acc: 0.7644
Epoch 2/10
625/625 [==============================] - 279s 447ms/step - loss: 0.4054 - a
cc: 0.8209 - val_loss: 0.4681 - val_acc: 0.7838
Epoch 3/10
625/625 [==============================] - 281s 449ms/step - loss: 0.2746 - a
cc: 0.8907 - val_loss: 0.3332 - val_acc: 0.8546
Epoch 4/10
625/625 [==============================] - 283s 452ms/step - loss: 0.1977 - a
cc: 0.9254 - val_loss: 0.3267 - val_acc: 0.8684
Epoch 5/10
625/625 [==============================] - 274s 438ms/step - loss: 0.1503 - a
cc: 0.9451 - val_loss: 0.3732 - val_acc: 0.8556
Epoch 6/10
625/625 [==============================] - 279s 446ms/step - loss: 0.1092 - a
cc: 0.9621 - val_loss: 0.4017 - val_acc: 0.8616
Epoch 7/10
625/625 [==============================] - 288s 461ms/step - loss: 0.0763 - a
cc: 0.9754 - val_loss: 0.4779 - val_acc: 0.8592
Epoch 8/10
625/625 [==============================] - 288s 461ms/step - loss: 0.0533 - a
cc: 0.9824 - val_loss: 0.5129 - val_acc: 0.8564
Epoch 9/10
625/625 [==============================] - 294s 471ms/step - loss: 0.0428 - a
cc: 0.9869 - val_loss: 0.5421 - val_acc: 0.8548
Epoch 10/10
625/625 [==============================] - 305s 488ms/step - loss: 0.0342 - a
cc: 0.9893 - val_loss: 0.5710 - val_acc: 0.8494
```



Бачимо, що на тренувальних даних точність досягнулася на рівні аж 98.93%, в той час як валідаційна функція втрат замість спадання вже на 4 кроці навпаки лише зростала. Да і валідаційна точність в 84.94% не викликає великої довіри.

Подивимось на точність на тестових даних.

```
> model %>% evaluate(x_test, y_test)
782/782 [==============================] - 83s 107ms/step - loss: 0.5827 - ac
c: 0.8455
     loss       acc
0.5827005 0.8455200
```

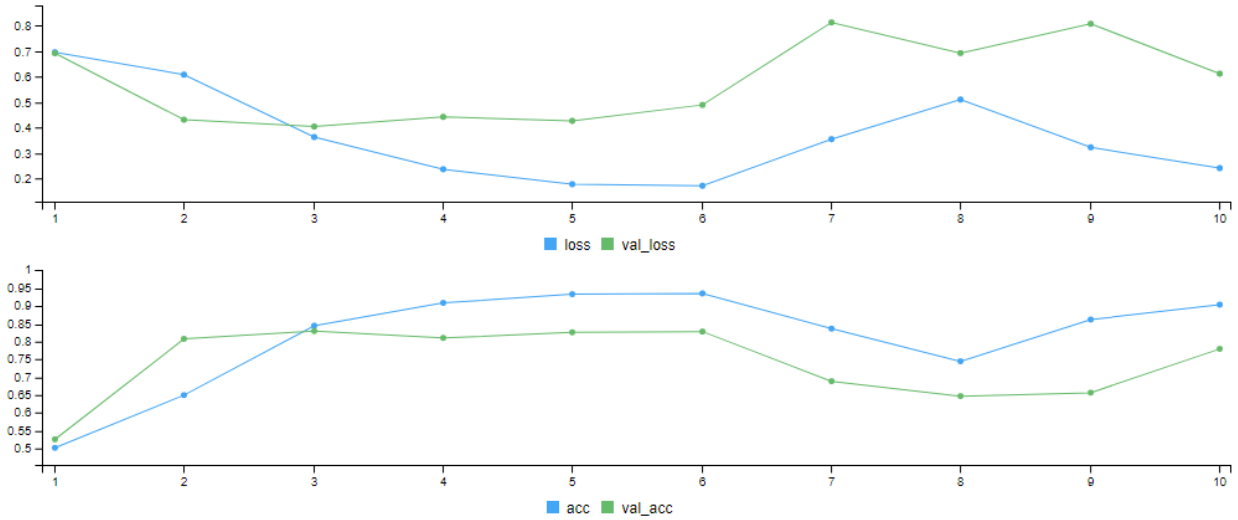
Цього разу вдалося досягнути точності роботи мережі в 84,55%, що, беззаперечно краще на цілий відсоток, але і по часу відбувається неабиякий програш.

Також від себе додам, що весь цей час обчислення виконувались на процесорі, оскільки, навіть маючи гарну відеокарту, я не можу її використовувати, оскільки вона від AMD, а не Nvidia. На жаль.

Можна спробувати збільшити кількість епох, але є велика підозра що це ніяк результат не поліпшить, про що, до речі, свідчить графік зверху.

Спробуємо модифікувати нашу модель: зробимо три рекурентні шари (але цього разу по 32 нейрони, і розмірність латентного простору повернемо 8), і додамо до них один LSTM блок.

```
> model <- keras_model_sequential() %>%
+   layer_embedding(input_dim = num_words, output_dim = 8, input_length =  ma
x_length) %>%
+   layer_simple_rnn(units = 32, return_sequences = TRUE) %>%
+   layer_simple_rnn(units = 32, return_sequences = TRUE) %>%
+   layer_lstm(units = 32, return_sequences = TRUE) %>%
+   layer_simple_rnn(units = 32) %>%
+   layer_dense(units = 1, activation = 'sigmoid')
> model %>% compile(optimizer = 'Adam', loss = 'binary_crossentropy', metrics
= c('acc'))
> model %>% fit(x_train, y_train, epochs = 10, batch_size = 32, validation_sp
lit = 0.2)
Epoch 1/10
625/625 [==============================] - 396s 633ms/step - loss: 0.6956 - a
cc: 0.5009 - val_loss: 0.6921 - val_acc: 0.5248
Epoch 2/10
625/625 [==============================] - 398s 637ms/step - loss: 0.6082 - a
cc: 0.6493 - val_loss: 0.4321 - val_acc: 0.8076
Epoch 3/10
625/625 [==============================] - 400s 640ms/step - loss: 0.3642 - a
cc: 0.8446 - val_loss: 0.4053 - val_acc: 0.8294
Epoch 4/10
625/625 [==============================] - 395s 632ms/step - loss: 0.2383 - a
cc: 0.9088 - val_loss: 0.4432 - val_acc: 0.8102
Epoch 5/10
625/625 [==============================] - 390s 625ms/step - loss: 0.1799 - a
cc: 0.9333 - val_loss: 0.4275 - val_acc: 0.8260
Epoch 6/10
625/625 [==============================] - 416s 666ms/step - loss: 0.1738 - a
cc: 0.9354 - val_loss: 0.4898 - val_acc: 0.8282
Epoch 7/10
625/625 [==============================] - 418s 669ms/step - loss: 0.3560 - a
cc: 0.8363 - val_loss: 0.8125 - val_acc: 0.6884
Epoch 8/10
625/625 [==============================] - 411s 657ms/step - loss: 0.5109 - a
cc: 0.7445 - val_loss: 0.6925 - val_acc: 0.6464
Epoch 9/10
625/625 [==============================] - 428s 685ms/step - loss: 0.3241 - a
cc: 0.8619 - val_loss: 0.8071 - val_acc: 0.6562
Epoch 10/10
625/625 [==============================] - 416s 665ms/step - loss: 0.2428 - a
cc: 0.9039 - val_loss: 0.6120 - val_acc: 0.7794
```



Можемо спостерігати доволі дивну поведінку, як точності на тренувальних даних, так і валідаційної точності. Те ж саме і у випалку функції втрат, причому обох її варіантів. Це точно не свідчить про якість даної мережі.

Подивимось, все-таки, на результат на тестових даних.

```
> model %>% evaluate(x_test, y_test)
782/782 [==============================] - 95s 122ms/step - loss: 0.6186 - ac
c: 0.7737
     loss       acc
0.6186387 0.7736800
```

Ні, це явно не те, чого ми очікували: точність в 77,3% є дуже для нас несуттєвою. Можливо, це спричинене тим, що ми в кожному прихованому шарі знову зменшили кількість нейронів, а також, можливо, й тим, що ми наші шари не зовсім, як сказати, вдало розташовували.

На мою думку, досягнути більшої точності можливо: для цього треба правильно скомбінувати приховані шари, і підібрати гарні параметри. Але це зробити на практиці не так просто, в силу дуже і дуже сильних витрат по часу. На жаль, хоча в мене відносно і потужна відеокарта, але вона не від Nvidia, тому запустити обчислення на ній виявилось неможливим.