Module 7

Andrew Estes

5/2/2022

Introduction

Analysis of sequential data is one of the tasks for which recurrent neural networks, and LSTM networks in particular, are touted as being especially applicable. We saw a classification problem involving numeric sequences of sensor data in the lectures. In this homework, you'll look at a classification problem involving text strings.

In particular, those who have had PDAT 613 will remember the problem of classifying headlines as coming either from *The Onion* (a satirical "news" source) or from *The Huffington Post* (a perhaps click-baity, but legitimate news source). Using a tuned support vector machine in that class, we achieved an accuracy of approximately 79%. Can we do better with neural networks?

The Data

Let's load the data and take a look at its raw form.

```
news.raw <- read.csv("headlines.csv") %>% select(headline, is_sarcastic)
head(news.raw)
```

```
##                                                                           headline
## 1       former versace store clerk sues over secret 'black code' for minority shoppers
## 2 the 'roseanne' revival catches up to our thorny political mood, for better and worse
## 3      mom starting to fear son's web series closest thing she will have to grandchild
## 4 boehner just wants wife to listen, not come up with alternative debt-reduction ideas
## 5                        j.k. rowling wishes snape happy birthday in the most magical way
## 6                                                               advancing the world's women
##   is_sarcastic
## 1            0
## 2            0
## 3            1
## 4            1
## 5            0
## 6            0
```

In order to work with this data, we need it to look like a matrix of numbers, where each row represents a word, and the columns identify *which* word:

Step	Man	Bites	Dog	...
1	1	0	0	...
2	0	1	0	...
3	0	0	1	...

This is achieved in several steps:

Step 1: Integer Tokenizer

A maximum number of words (`num.words`) is established, and the most frequently occurring words are encoded with integers up to that maximum number.

```
# Define the number of words to be used.
num.words <- 10000
# Create a tokenizer object that converts words to integers.
tokenizer <- text_tokenizer(num_words=num.words)
```

```
## Loaded Tensorflow version 2.8.0
```

```
# Fit the tokenizer to our lexicon so that it will tokenize our headlines.
tokenizer %>% fit_text_tokenizer(news.raw$headline)
# Convert the original words to integer tokens.
news.seq <- texts_to_sequences(tokenizer, news.raw$headline)
head(news.seq)
```

```
## [[1]]
##  [1]  307  678 3336 2297   47  381 2575    5 2576 8433
##
## [[2]]
##  [1]    3 8434 3337 2745   21    1  165 8435  415 3111    5  257    8 1001
##
## [[3]]
##  [1]  144  837    1  906 1748 2092  581 4718  220  142   38   45    1
##
## [[4]]
##  [1] 1484   35  223  399    1 1831   28  318   21    9 2923 1392 6968  967
##
## [[5]]
##  [1]  766  718 4719  907  622  593    4    3   94 1308   91
##
## [[6]]
## [1]   3 364  72
```

Note that the end result is a list of integer sequences, one for each headline.

Step 2: Pad the sequences to have the same length.

Padding the sequences (headlines) to have the same length is a necessary step in using our keras/tensorflow package. To do this, 0's are inserted at the beginning of each sequence (headline) to give them all the same number of "words." The `pad_sequences` command also converts our object into an array.

In the last step, we creates a *list* of integer sequences because each headline had a different length, and lists in R are the objects that can handle collecting objects of different lengths. Once each sequence has been padded, they can be stored more efficiently in an array.

```
# Pad the sequences.
news.seq <- pad_sequences(news.seq)
head(news.seq)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13] [,14]
## [1,]    0    0    0    0    0    0    0    0    0     0     0     0     0     0
## [2,]    0    0    0    0    0    0    0    0    0     0     0     0     0     0
## [3,]    0    0    0    0    0    0    0    0    0     0     0     0     0     0
## [4,]    0    0    0    0    0    0    0    0    0     0     0     0     0     0
## [5,]    0    0    0    0    0    0    0    0    0     0     0     0     0     0
## [6,]    0    0    0    0    0    0    0    0    0     0     0     0     0     0
##      [,15] [,16] [,17] [,18] [,19] [,20] [,21] [,22] [,23] [,24] [,25] [,26]
## [1,]     0     0     0     0     0     0     0     0     0     0   307   678
## [2,]     0     0     0     0     0     0     3  8434  3337  2745    21     1
## [3,]     0     0     0     0     0     0     0   144   837     1   906  1748
## [4,]     0     0     0     0     0     0  1484    35   223   399     1  1831
## [5,]     0     0     0     0     0     0     0     0     0   766   718  4719
## [6,]     0     0     0     0     0     0     0     0     0     0     0     0
##      [,27] [,28] [,29] [,30] [,31] [,32] [,33] [,34]
## [1,]  3336  2297    47   381  2575     5  2576  8433
## [2,]   165  8435   415  3111     5   257     8  1001
## [3,]  2092   581  4718   220   142    38    45     1
## [4,]    28   318    21     9  2923  1392  6968   967
## [5,]   907   622   593     4     3    94  1308    91
## [6,]     0     0     0     0     0     3   364    72
```

```
# Find out the maximum headline length for later use. It's possible to cut off
# texts that are too long, but here we are just letting the longest headline
# determine this length.
max.len <- dim(news.seq)[2]
```

Now the first index (rows) of the `news.seq` array indexes the headlines.

Step 3: Convert integer sequences into "one hot" encoding.

Steps 3 and 4 are both done in the first layer of the neural network, so we'll see the code in a moment. But in this step, we want to convert from a two-dimensional array where each headline is single row and each word is an integer:

Man	Bites	Dog	...
27	10	105	...

to a three-dimensional array where the first index still indexes which headline we're talking about, but the next two dimensions encode the one hot encoding of the headline:

Man	Bites	Dog	...
1	0	0	...
0	1	0	...
0	0	1	...

Step 4: Dimension Reduction

Since the array above can get very large (10000 words gives 10000 columns for each headline, for example), dimensional reduction techniques are employed to create a matrix with fewer columns that still contains most of the information of the matrix from Step 3. (That's the `output_dim` parameter in the code below.)

Question 1: Tuning a LSTM Networks

As previously mentioned, we got about 79% accuracy with a tuned SVM in PDAT 613 when we considered this data set. Your assignment is to do better with a LSTM neural network. The code below should start you out, although you'll need to finish it.

There are several obvious tuning parameters you can adjust:

- `num.words`, which defines how many words will be included in the "vocabulary" of the tokenizer,
- `out.dim`, which defines how many "columns" each headline will be reduced to by dimension reduction,
- the number of epochs you run the model (it could get worse over time, so don't run too long),
- the number of units in the LSTM layer, and
- any decay rate you decide to add to a decay layer.

I'd suggest keeping the number of layers small and the number epochs small. You might be able to do much better than I did, but I certainly was able to beat 79% accuracy without many layers.

Also, I think you can get by without making a train/test split, and simply using the `validation_split=0.2` option in the `fit` command in order to monitor your network's performance.

Specifically, your submission should

- Define the neural network with appropriate code.
- Run the fit command to fit the neural network.
- Output a graph showing the training and validation accuracy of the final model.
- Include a brief comment on how well the tuning did and any patterns you saw in the graph that were worth noting.

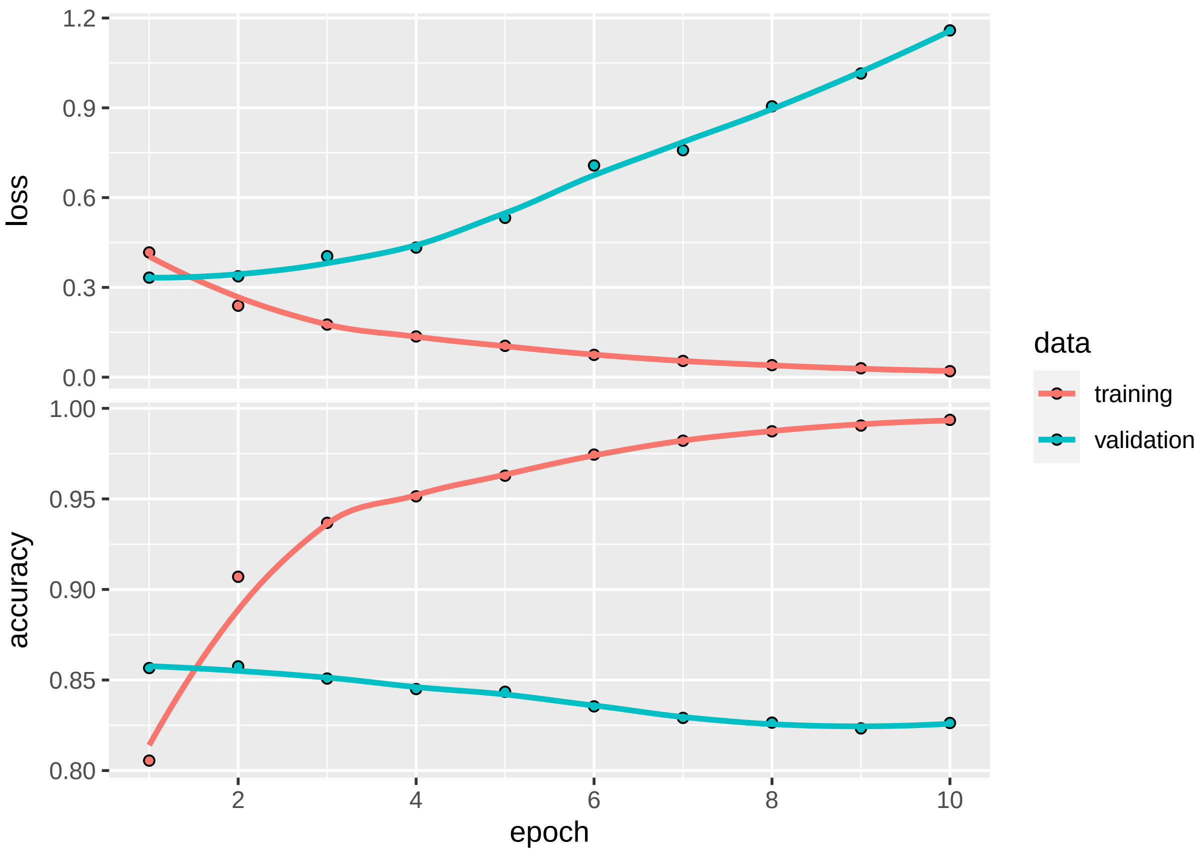
Here's some partial code to get you started:

```
out.dim <- 10

model.lstm <- keras_model_sequential() %>%
  layer_embedding(input_dim=num.words, output_dim=out.dim, input_length=max.len,
                  mask_zero=TRUE) %>%
  layer_lstm(units=32) %>%
```

```
  layer_dense(units=16, activation="relu") %>%
  layer_dropout(rate=0.2) %>%
  layer_dense(units=1, activation="sigmoid") %>%
  compile(
    optimizer="adam",
    loss="binary_crossentropy",
    metrics="accuracy"
  )
```

```
history <- model.lstm %>% fit(
  x= news.seq,
  y= news.raw$is_sarcastic,
  validation_split = .2,
  epochs=10,
  batch_size=25
)


plot(history)
```

```
## `geom_smooth()` using formula 'y ~ x'
```



The tuning did a good job. The LSTM model had a better accuracy (83%) than the SVM model (79%).

Question 2: Making Predictions

We might want to use our neural network to predict the source of new headlines. In order to do that, the new headline will also have to be tokenized:

```
headline.text <- c("It was the best of times, it was the worst of times.")
new.headline <- texts_to_sequences(tokenizer, headline.text)
new.headline <- pad_sequences(new.headline, maxlen=max.len)
model.lstm %>% predict(new.headline)
```

```
##              [,1]
## [1,] 2.152155e-08
```

Since a "1" in the is_sarcastic column indicated a satirical headline, our model thinks it's highly unlikely that this headline came from the Onion.

With that example, now collect a few examples of headlines from both *The Onion* and *The Huffington Post*, put them into a character vector, and see what the model predicts. It was trained on headlines from a few years ago–does it still have what it takes to make new predictions (at least for your small sample)?

[Note: If you're morally opposed to visiting either site, feel free to try on other headlines from other sources.]

```
headline.text2 <- c("Best Ways To Make Friends As An Adult",
                   "Ohio Law Mandates Rape Victims Send Thank You Notes For Gift Of Parenthood",
                   "Biden Tries To Boost Approval Ratings By Showing A Little Ankle",
                   "JAN. 6 PANEL SEEKS MORE GOP BIG LIE BOOSTERS",
                   "Ex-Defense Secretary: Trump Told Pentagon To Shoot George Floyd Protesters",
                   "Gas Giants Have Been Ghostwriting Letters Of Support From Elected Officials",
                   "This is real",
                   "Royals win"
)

new.headline2 <- texts_to_sequences(tokenizer, headline.text2)
new.headline2 <- pad_sequences(new.headline2, maxlen=max.len)
model.lstm %>% predict(new.headline2)
```

```
##              [,1]
## [1,] 1.443489e-05
## [2,] 1.924878e-05
## [3,] 9.798148e-01
## [4,] 1.210474e-07
## [5,] 9.405603e-01
## [6,] 8.871042e-01
## [7,] 3.201365e-04
## [8,] 1.416609e-02
```

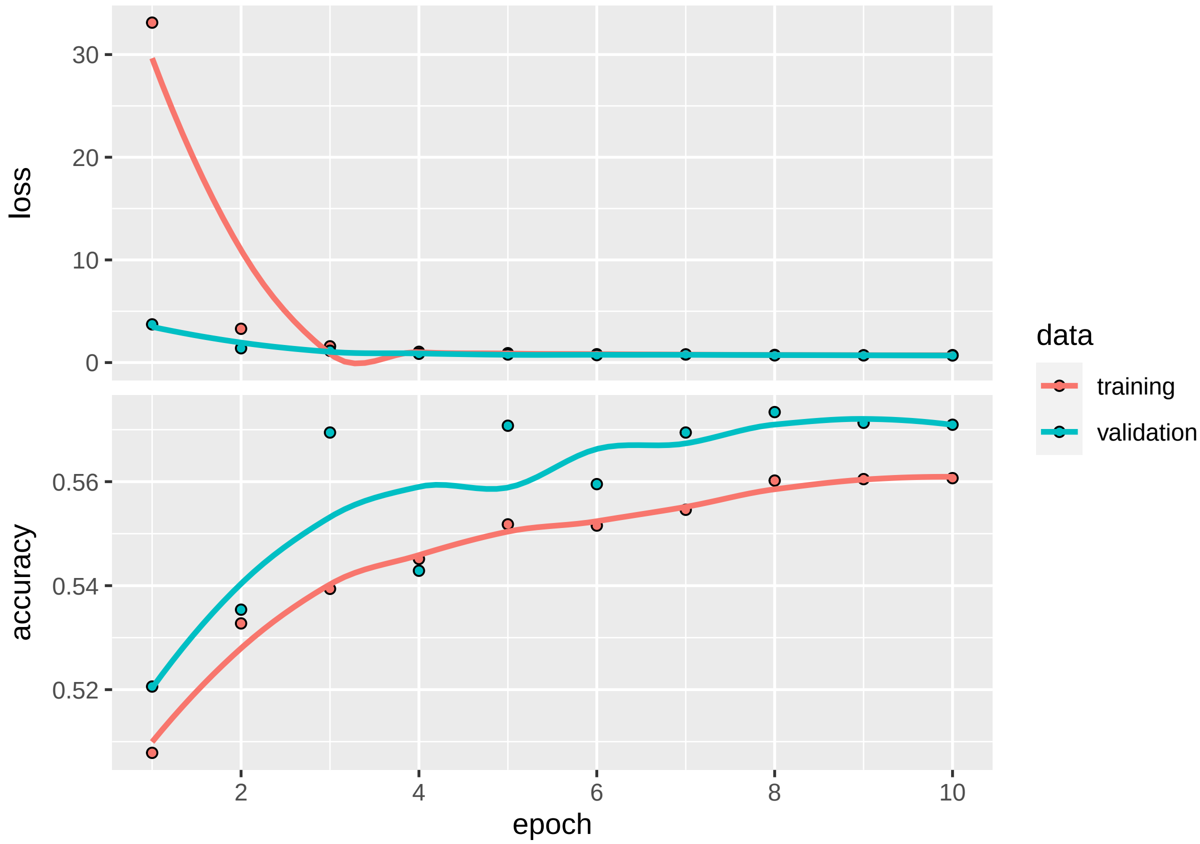
Question 3: Tuning a Regular Neural Network

Ideal homework for the section of recurrent neural networks would give you a data set and problem that illustrates how LSTM networks perform better than alternatives. Is that true in this case? Create a "plain" neural network using `layer_dense`, rather than `layer_lstm`. You'll need a `layer_flatten` before your first layer_dense. Do everything you did for the last part. Then comment on the performance of the plain neural network for this example data.

```
k_clear_session()

model.nn <- keras_model_sequential() %>%
  layer_flatten() %>%
  layer_dense(units=16, activation="relu") %>%
  layer_dropout(rate=0.2) %>%
  layer_dense(units=16, activation="relu") %>%
  layer_dense(units=1, activation="sigmoid") %>%
  compile(
    optimizer="adam",
    loss="binary_crossentropy",
    metrics="accuracy"
  )
```

```
history.nn <- model.nn %>% fit(
  x= news.seq,
  y= news.raw$is_sarcastic,
  validation_split = .2,
  epochs=10,
  batch_size=25
)

plot(history.nn)
```

```
## `geom_smooth()` using formula 'y ~ x'
```



The accuracy was 56% on the regular neural network. The accuracy was 83% on the LSTM model.