# **TensorFlow Tutorial #20**

# **Natural Language Processing**

by [Magnus Erik Hvass Pedersen](http://www.hvass-labs.org/) / [GitHub](https://github.com/Hvass-Labs/TensorFlow-Tutorials) / [Videos on YouTube](https://www.youtube.com/playlist?list=PL9Hr9sNUjfsmEu1ZniY0XpHSzl5uihcXZ)

## **Introduction**

This tutorial is about a basic form of Natural Language Processing (NLP) called Sentiment Analysis, in which we will try and classify a movie review as either positive or negative.

Consider a simple example: "This movie is not very good." This text ends with the words "very good" which indicates a very positive sentiment, but it is negated because it is preceded by the word "not", so the text should be classified as having a negative sentiment. How can we teach a Neural Network to do this classification?

Another problem is that neural networks cannot work directly on text-data, so we need to convert text into numbers that are compatible with a neural network.

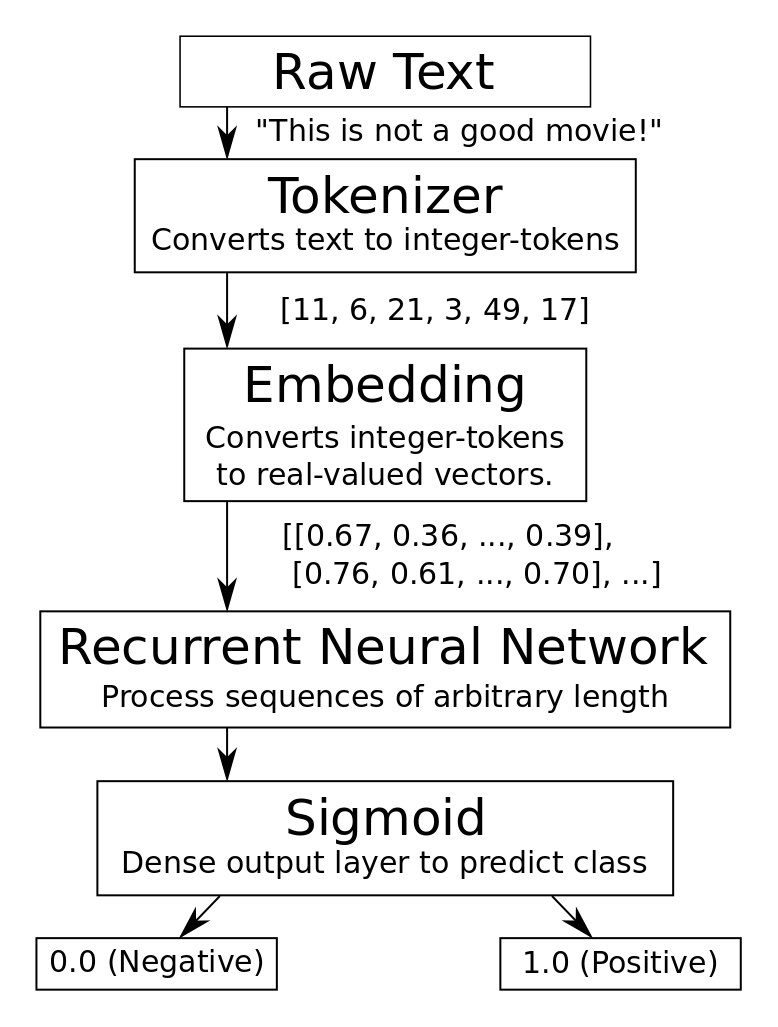
Yet another problem is that a text may be arbitrarily long. The neural networks we have worked with in previous tutorials use fixed data-shapes - except for the first dimension of the data which varies with the batch-size. Now we need a type of neural network that can work on both short and long sequences of text.

You should be familiar with TensorFlow and Keras in general, see Tutorials #01 and #03-C.

## **Flowchart**

To solve this problem we need several processing steps. First we need to convert the raw text-words into so-called tokens which are integer values. These tokens are really just indices into a list of the entire vocabulary. Then we convert these integer-tokens into so-called embeddings which are real-valued vectors, whose mapping will be trained along with the neural network, so as to map words with similar meanings to similar embedding-vectors. Then we input these embedding-vectors to a Recurrent Neural Network which can take sequences of arbitrary length as input and output a kind of summary of what it has seen in the input. This output is then squashed using a Sigmoid-function to give us a value between 0.0 and 1.0, where 0.0 is taken to mean a negative sentiment and 1.0 means a positive sentiment. This whole process allows us to classify input-text as either having a negative or positive sentiment.

The flowchart of the algorithm is roughly:



## **Recurrent Neural Network**

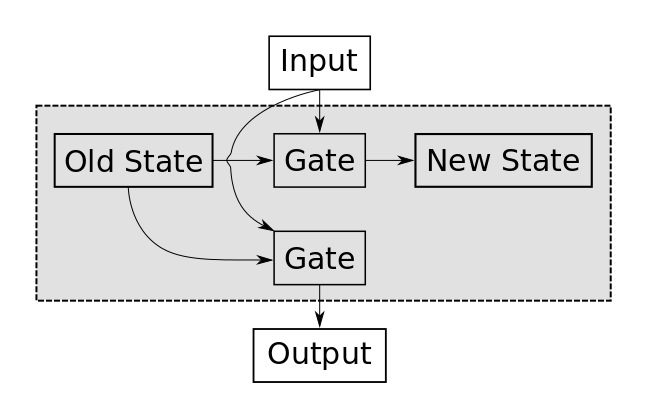
The basic building block in a Recurrent Neural Network (RNN) is a Recurrent Unit (RU). There are many different variants of recurrent units such as the rather clunky LSTM (Long-Short-Term-Memory) and the somewhat simpler GRU (Gated Recurrent Unit) which we will use in this tutorial. Experiments in the literature suggest that the LSTM and GRU have roughly similar performance. Even simpler variants also exist and the literature suggests that they may perform even better than both LSTM and GRU, but they are not implemented in Keras which we will use in this tutorial.

The following figure shows the abstract idea of a recurrent unit, which has an internal state that is being updated every time the unit receives a new input. This internal state serves as a kind of memory. However, it is not a traditional kind of computer memory which stores bits that are either on or off. Instead the recurrent unit stores floating-point values in its memory-state, which are read and written using matrix-operations so the operations are all differentiable. This means the memory-state can store arbitrary floating-point values (although typically limited between -1.0 and 1.0) and the network can be trained like a normal neural network using Gradient Descent.

The new state-value depends on both the old state-value and the current input. For example, if the state-value has memorized that we have recently seen the word "not" and the current input is "good" then we need to store a new state-value that memorizes "not good" which indicates a negative sentiment.

The part of the recurrent unit that is responsible for mapping old state-values and inputs to the new state-value is called a gate, but it is really just a type of matrix-operation. There is another gate for calculating the output-values of the recurrent unit. The implementation of these gates vary for different types of recurrent units. This figure merely shows the abstract idea of a recurrent unit. The LSTM has more gates than the GRU but some of them are apparently redundant so they can be omitted.

In order to train the recurrent unit, we must gradually change the weight-matrices of the gates so the recurrent unit gives the desired output for an input sequence. This is done automatically in TensorFlow.



### **Unrolled Network**

Another way to visualize and understand a Recurrent Neural Network is to "unroll" the recursion. In this figure there is only a single recurrent unit denoted RU, which will receive a text-word from the input sequence in a series of time-steps.

The initial memory-state of the RU is reset to zero internally by Keras / TensorFlow every time a new sequence begins.

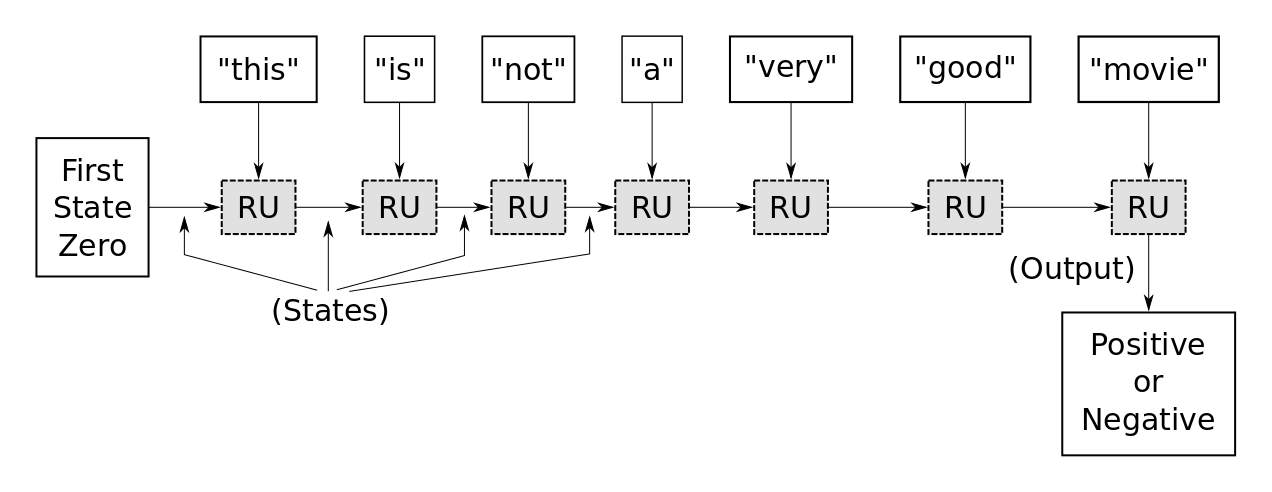
In the first time-step the word "this" is input to the RU which uses its internal state (initialized to zero) and its gate to calculate the new state. The RU also uses its other gate to calculate the output but it is ignored here because it is only needed at the end of the sequence to output a kind of summary.

In the second time-step the word "is" is input to the RU which now uses the internal state that was just updated from seeing the previous word "this".

There is not much meaning in the words "this is" so the RU probably doesn't save anything important in its internal state from seeing these words. But when it sees the third word "not" the RU has learned that it may be important for determining the overall sentiment of the input-text, so it needs to be stored in the memory-state of the RU, which can be used later when the RU sees the word "good" in time-step 6.

Finally when the entire sequence has been processed, the RU outputs a vector of values that summarizes what it has seen in the input sequence. We then use a fully-connected layer with a Sigmoid activation to get a single value between 0.0 and 1.0 which we interpret as the sentiment either being negative (values close to 0.0) or positive (values close to 1.0).

Note that for the sake of clarity, this figure doesn't show the mapping from text-words to integer-tokens and embedding-vectors, as well as the fully-connected Sigmoid layer on the output.

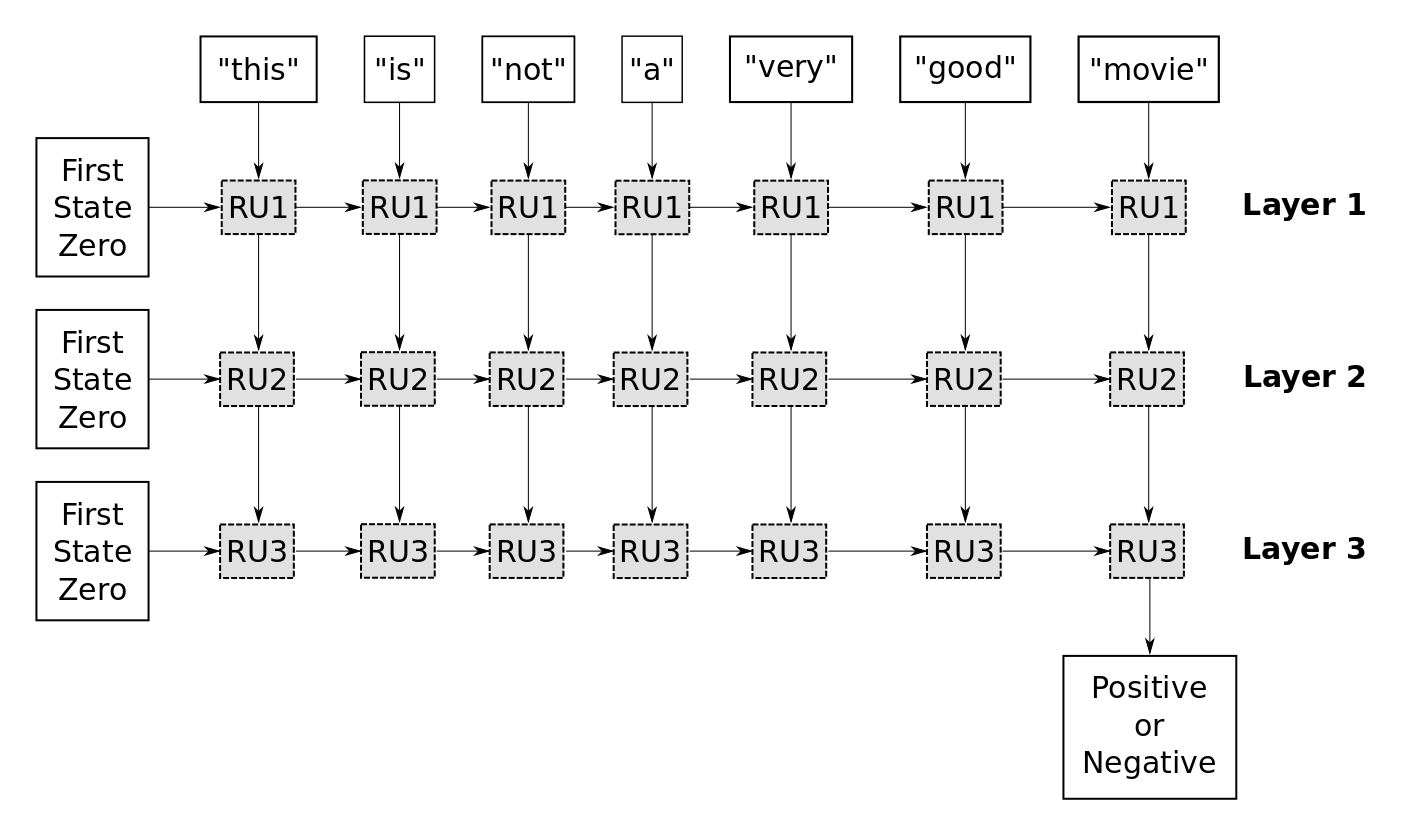


### **3-Layer Unrolled Network**

In this tutorial we will use a Recurrent Neural Network with 3 recurrent units (or layers) denoted RU1, RU2 and RU3 in the "unrolled" figure below.

The first layer is much like the unrolled figure above for a single-layer RNN. First the recurrent unit RU1 has its internal state initialized to zero by Keras / TensorFlow. Then the word "this" is input to RU1 and it updates its internal state. Then it processes the next word "is", and so forth. But instead of outputting a single summary value at the end of the sequence, we use the output of RU1 for every time-step. This creates a new sequence that can then be used as input for the next recurrent unit RU2. The same process is repeated for the second layer and this creates a new output sequence which is then input to the third layer's recurrent unit RU3, whose final output is passed to a fully-connected Sigmoid layer that outputs a value between 0.0 (negative sentiment) and 1.0 (positive sentiment).

Note that for the sake of clarity, the mapping of text-words to integer-tokens and embedding-vectors has been omitted from this figure.



### **Exploding & Vanishing Gradients**

In order to train the weights for the gates inside the recurrent unit, we need to minimize some loss-function which measures the difference between the actual output of the network relative to the desired output.

From the "unrolled" figures above we see that the reccurent units are applied recursively for each word in the input sequence. This means the recurrent gate is applied once for each time-step. The gradient-signals have to flow back from the loss-function all the way to the first time the recurrent gate is used. If the gradient of the recurrent gate is multiplicative, then we essentially have an exponential function.

In this tutorial we will use texts that have more than 500 words. This means the RU's gate for updating its internal memory-state is applied recursively more than 500 times. If a gradient of just 1.01 is multiplied with itself 500 times then it gives a value of about 145. If a gradient of just 0.99 is multiplied with itself 500 times then it gives a value of about 0.007. These are called exploding and vanishing gradients. The only gradients that can survive recurrent multiplication are 0 and 1.

To avoid these so-called exploding and vanishing gradients, care must be made when designing the recurrent unit and its gates. That is why the actual implementation of the GRU is more complicated, because it tries to send the gradient back through the gates without this distortion.

## **Imports**

In [1]:

%matplotlib inline

**import** **matplotlib.pyplot** **as** **plt**

**import** **tensorflow** **as** **tf**

**import** **numpy** **as** **np**

**from** **scipy.spatial.distance** **import** cdist

/home/magnus/anaconda3/envs/tf-gpu/lib/python3.6/site-packages/h5py/\_\_init\_\_.py:36: FutureWarning: Conversion of the second argument of issubdtype from `float` to `np.floating` is deprecated. In future, it will be treated as `np.float64 == np.dtype(float).type`.

from .\_conv import register\_converters as \_register\_converters

We need to import several things from Keras.

In [2]:

*# from tf.keras.models import Sequential # This does not work!*

**from** **tensorflow.python.keras.models** **import** Sequential

**from** **tensorflow.python.keras.layers** **import** Dense, GRU, Embedding

**from** **tensorflow.python.keras.optimizers** **import** Adam

**from** **tensorflow.python.keras.preprocessing.text** **import** Tokenizer

**from** **tensorflow.python.keras.preprocessing.sequence** **import** pad\_sequences

This was developed using Python 3.6 (Anaconda) and package versions:

In [3]:

tf.\_\_version\_\_

Out[3]:

'1.5.0'

In [4]:

tf.keras.\_\_version\_\_

Out[4]:

'2.1.2-tf'

## **Load Data**

We will use a data-set consisting of 50000 reviews of movies from IMDB. Keras has a built-in function for downloading a similar data-set (but apparently half the size). However, Keras' version has already converted the text in the data-set to integer-tokens, which is a crucial part of working with natural languages that will also be demonstrated in this tutorial, so we download the actual text-data.

NOTE: The data-set is 84 MB and will be downloaded automatically.

In [5]:

**import** **imdb**

Change this if you want the files saved in another directory.

In [6]:

*# imdb.data\_dir = "data/IMDB/"*

Automatically download and extract the files.

In [7]:

imdb.maybe\_download\_and\_extract()

Data has apparently already been downloaded and unpacked.

Load the training- and test-sets.

In [8]:

x\_train\_text, y\_train = imdb.load\_data(train=**True**)

x\_test\_text, y\_test = imdb.load\_data(train=**False**)

In [9]:

print("Train-set size: ", len(x\_train\_text))

print("Test-set size: ", len(x\_test\_text))

Train-set size: 25000

Test-set size: 25000

Combine into one data-set for some uses below.

In [10]:

data\_text = x\_train\_text + x\_test\_text

Print an example from the training-set to see that the data looks correct.

In [11]:

x\_train\_text[1]

Out[11]:

'A simple comment...<br /><br />What can I say... this is a wonderful film that I can watch over and over. It is definitely one of the top ten comedies made. With a great cast, Jack Lemmon and Walter Matthau wording a perfect script by Neil Simon, based on his play.<br /><br />It is real to life situation done perfectly. If you have digital cable, one gets the menu on bottom of screen to give what is on. It usually gives this film \*\*\*% stars but in reality it deserves \*\*\*\* stars. If you really watch this film, one can tell that it will be as funny and fresh a hundred years from now.'

The true "class" is a sentiment of the movie-review. It is a value of 0.0 for a negative sentiment and 1.0 for a positive sentiment. In this case the review is positive.

In [12]:

y\_train[1]

Out[12]:

1.0

## **Tokenizer**

A neural network cannot work directly on text-strings so we must convert it somehow. There are two steps in this conversion, the first step is called the "tokenizer" which converts words to integers and is done on the data-set before it is input to the neural network. The second step is an integrated part of the neural network itself and is called the "embedding"-layer, which is described further below.

We may instruct the tokenizer to only use e.g. the 10000 most popular words from the data-set.

In [13]:

num\_words = 10000

In [14]:

tokenizer = Tokenizer(num\_words=num\_words)

The tokenizer can then be "fitted" to the data-set. This scans through all the text and strips it from unwanted characters such as punctuation, and also converts it to lower-case characters. The tokenizer then builds a vocabulary of all unique words along with various data-structures for accessing the data.

Note that we fit the tokenizer on the entire data-set so it gathers words from both the training- and test-data. This is OK as we are merely building a vocabulary and want it to be as complete as possible. The actual neural network will of course only be trained on the training-set.

In [15]:

%%time

tokenizer.fit\_on\_texts(data\_text)

CPU times: user 10.6 s, sys: 16 ms, total: 10.6 s

Wall time: 10.6 s

If you want to use the entire vocabulary then set num\_words=None above, and then it will automatically be set to the vocabulary-size here. (This is because of Keras' somewhat awkward implementation.)

In [16]:

**if** num\_words **is** **None**:

num\_words = len(tokenizer.word\_index)

We can then inspect the vocabulary that has been gathered by the tokenizer. This is ordered by the number of occurrences of the words in the data-set. These integer-numbers are called word indices or "tokens" because they uniquely identify each word in the vocabulary.

In [17]:

tokenizer.word\_index

Out[17]:

{'the': 1,

'and': 2,

'a': 3,

'of': 4,

'to': 5,

'is': 6,

'br': 7,

'in': 8,

'it': 9,

'i': 10,

'this': 11,

'that': 12,

'was': 13,

'as': 14,

'for': 15,

'with': 16,

'movie': 17,

'but': 18,

'film': 19,

'on': 20,

'not': 21,

'you': 22,

'are': 23,

'his': 24,

'have': 25,

'be': 26,

'one': 27,

'he': 28,

'all': 29,

'at': 30,

'by': 31,

'an': 32,

'they': 33,

'so': 34,

'who': 35,

'from': 36,

'like': 37,

'or': 38,

'just': 39,

'her': 40,

'out': 41,

'about': 42,

'if': 43,

"it's": 44,

'has': 45,

'there': 46,

'some': 47,

'what': 48,

'good': 49,

'when': 50,

'more': 51,

'very': 52,

'up': 53,

'no': 54,

'time': 55,

'my': 56,

'even': 57,

'would': 58,

'she': 59,

'which': 60,

'only': 61,

'really': 62,

'see': 63,

'story': 64,

'their': 65,

'had': 66,

'can': 67,

'me': 68,

'well': 69,

'were': 70,

'than': 71,

'much': 72,

'we': 73,

'bad': 74,

'been': 75,

'get': 76,

'do': 77,

'great': 78,

'other': 79,

'will': 80,

'also': 81,

'into': 82,

'people': 83,

'because': 84,

'how': 85,

'first': 86,

'him': 87,

'most': 88,

"don't": 89,

'made': 90,

'then': 91,

'its': 92,

'them': 93,

'make': 94,

'way': 95,

'too': 96,

'movies': 97,

'could': 98,

'any': 99,

'after': 100,

'think': 101,

'characters': 102,

'watch': 103,

'films': 104,

'two': 105,

'many': 106,

'seen': 107,

'character': 108,

'being': 109,

'never': 110,

'plot': 111,

'love': 112,

'acting': 113,

'life': 114,

'did': 115,

'best': 116,

'where': 117,

'know': 118,

'show': 119,

'little': 120,

'over': 121,

'off': 122,

'ever': 123,

'does': 124,

'your': 125,

'better': 126,

'end': 127,

'man': 128,

'scene': 129,

'still': 130,

'say': 131,

'these': 132,

'here': 133,

'why': 134,

'scenes': 135,

'while': 136,

'something': 137,

'such': 138,

'go': 139,

'through': 140,

'back': 141,

'should': 142,

'those': 143,

'real': 144,

"i'm": 145,

'now': 146,

'watching': 147,

'thing': 148,

"doesn't": 149,

'actors': 150,

'though': 151,

'funny': 152,

'years': 153,

"didn't": 154,

'old': 155,

'another': 156,

'10': 157,

'work': 158,

'before': 159,

'actually': 160,

'nothing': 161,

'makes': 162,

'look': 163,

'director': 164,

'find': 165,

'going': 166,

'same': 167,

'new': 168,

'lot': 169,

'every': 170,

'few': 171,

'again': 172,

'part': 173,

'cast': 174,

'down': 175,

'us': 176,

'things': 177,

'want': 178,

'quite': 179,

'pretty': 180,

'world': 181,

'horror': 182,

'around': 183,

'seems': 184,

"can't": 185,

'young': 186,

'take': 187,

'however': 188,

'got': 189,

'thought': 190,

'big': 191,

'fact': 192,

'enough': 193,

'long': 194,

'both': 195,

"that's": 196,

'give': 197,

"i've": 198,

'own': 199,

'may': 200,

'between': 201,

'comedy': 202,

'right': 203,

'series': 204,

'action': 205,

'must': 206,

'music': 207,

'without': 208,

'times': 209,

'saw': 210,

'always': 211,

'original': 212,

"isn't": 213,

'role': 214,

'come': 215,

'almost': 216,

'gets': 217,

'interesting': 218,

'guy': 219,

'point': 220,

'done': 221,

"there's": 222,

'whole': 223,

'least': 224,

'far': 225,

'bit': 226,

'script': 227,

'minutes': 228,

'feel': 229,

'2': 230,

'anything': 231,

'making': 232,

'might': 233,

'since': 234,

'am': 235,

'family': 236,

"he's": 237,

'last': 238,

'probably': 239,

'tv': 240,

'performance': 241,

'kind': 242,

'away': 243,

'yet': 244,

'fun': 245,

'worst': 246,

'sure': 247,

'rather': 248,

'hard': 249,

'girl': 250,

'anyone': 251,

'each': 252,

'played': 253,

'day': 254,

'found': 255,

'looking': 256,

'woman': 257,

'screen': 258,

'although': 259,

'our': 260,

'especially': 261,

'believe': 262,

'having': 263,

'trying': 264,

'course': 265,

'dvd': 266,

'everything': 267,

'set': 268,

'goes': 269,

'comes': 270,

'put': 271,

'ending': 272,

'maybe': 273,

'place': 274,

'book': 275,

'shows': 276,

'three': 277,

'worth': 278,

'different': 279,

'main': 280,

'once': 281,

'sense': 282,

'american': 283,

'reason': 284,

'looks': 285,

'effects': 286,

'watched': 287,

'play': 288,

'true': 289,

'money': 290,

'actor': 291,

"wasn't": 292,

'job': 293,

'together': 294,

'war': 295,

'someone': 296,

'plays': 297,

'instead': 298,

'high': 299,

'during': 300,

'year': 301,

'said': 302,

'half': 303,

'everyone': 304,

'later': 305,

'takes': 306,

'1': 307,

'seem': 308,

'audience': 309,

'special': 310,

'beautiful': 311,

'left': 312,

'himself': 313,

'seeing': 314,

'john': 315,

'night': 316,

'black': 317,

'version': 318,

'shot': 319,

'excellent': 320,

'idea': 321,

'house': 322,

'mind': 323,

'star': 324,

'wife': 325,

'fan': 326,

'death': 327,

'used': 328,

'else': 329,

'simply': 330,

'nice': 331,

'budget': 332,

'poor': 333,

'completely': 334,

'short': 335,

'second': 336,

"you're": 337,

'3': 338,

'read': 339,

'less': 340,

'along': 341,

'top': 342,

'help': 343,

'home': 344,

'men': 345,

'either': 346,

'line': 347,

'boring': 348,

'dead': 349,

'friends': 350,

'kids': 351,

'try': 352,

'production': 353,

'enjoy': 354,

'camera': 355,

'use': 356,

'wrong': 357,

'given': 358,

'low': 359,

'classic': 360,

'father': 361,

'need': 362,

'full': 363,

'stupid': 364,

'until': 365,

'next': 366,

'performances': 367,

'school': 368,

'hollywood': 369,

'rest': 370,

'truly': 371,

'awful': 372,

'video': 373,

'couple': 374,

'start': 375,

'sex': 376,

'recommend': 377,

'women': 378,

'let': 379,

'tell': 380,

'terrible': 381,

'remember': 382,

'mean': 383,

'came': 384,

'understand': 385,

'getting': 386,

'perhaps': 387,

'moments': 388,

'name': 389,

'keep': 390,

'face': 391,

'itself': 392,

'wonderful': 393,

'playing': 394,

'human': 395,

'style': 396,

'small': 397,

'episode': 398,

'perfect': 399,

'others': 400,

'person': 401,

'doing': 402,

'often': 403,

'early': 404,

'stars': 405,

'definitely': 406,

'written': 407,

'head': 408,

'lines': 409,

'dialogue': 410,

'gives': 411,

'piece': 412,

"couldn't": 413,

'went': 414,

'finally': 415,

'mother': 416,

'case': 417,

'title': 418,

'absolutely': 419,

'live': 420,

'boy': 421,

'yes': 422,

'laugh': 423,

'certainly': 424,

'liked': 425,

'become': 426,

'entertaining': 427,

'worse': 428,

'oh': 429,

'sort': 430,

'loved': 431,

'lost': 432,

'hope': 433,

'called': 434,

'picture': 435,

'felt': 436,

'overall': 437,

'entire': 438,

'several': 439,

'mr': 440,

'based': 441,

'supposed': 442,

'cinema': 443,

'friend': 444,

'guys': 445,

'sound': 446,

'5': 447,

'problem': 448,

'drama': 449,

'against': 450,

'waste': 451,

'white': 452,

'beginning': 453,

'4': 454,

'fans': 455,

'totally': 456,

'dark': 457,

'care': 458,

'direction': 459,

'humor': 460,

'wanted': 461,

"she's": 462,

'seemed': 463,

'under': 464,

'game': 465,

'children': 466,

'despite': 467,

'lives': 468,

'lead': 469,

'guess': 470,

'example': 471,

'already': 472,

'final': 473,

'throughout': 474,

"you'll": 475,

'turn': 476,

'evil': 477,

'becomes': 478,

'unfortunately': 479,

'able': 480,

'quality': 481,

"i'd": 482,

'days': 483,

'history': 484,

'fine': 485,

'side': 486,

'wants': 487,

'heart': 488,

'horrible': 489,

'writing': 490,

'amazing': 491,

'b': 492,

'flick': 493,

'killer': 494,

'run': 495,

'son': 496,

'\x96': 497,

'michael': 498,

'works': 499,

'close': 500,

"they're": 501,

'act': 502,

'art': 503,

'matter': 504,

'kill': 505,

'etc': 506,

'tries': 507,

"won't": 508,

'past': 509,

'town': 510,

'turns': 511,

'enjoyed': 512,

'brilliant': 513,

'gave': 514,

'behind': 515,

'parts': 516,

'stuff': 517,

'genre': 518,

'eyes': 519,

'car': 520,

'favorite': 521,

'directed': 522,

'late': 523,

'hand': 524,

'expect': 525,

'soon': 526,

'hour': 527,

'obviously': 528,

'themselves': 529,

'sometimes': 530,

'killed': 531,

'actress': 532,

'thinking': 533,

'child': 534,

'girls': 535,

'viewer': 536,

'starts': 537,

'city': 538,

'myself': 539,

'decent': 540,

'highly': 541,

'stop': 542,

'type': 543,

'self': 544,

'god': 545,

'says': 546,

'group': 547,

'anyway': 548,

'voice': 549,

'took': 550,

'known': 551,

'blood': 552,

'kid': 553,

'heard': 554,

'happens': 555,

'except': 556,

'fight': 557,

'feeling': 558,

'experience': 559,

'coming': 560,

'slow': 561,

'daughter': 562,

'writer': 563,

'stories': 564,

'moment': 565,

'leave': 566,

'told': 567,

'extremely': 568,

'score': 569,

'violence': 570,

'involved': 571,

'police': 572,

'strong': 573,

'chance': 574,

'lack': 575,

'cannot': 576,

'hit': 577,

'roles': 578,

'hilarious': 579,

's': 580,

'wonder': 581,

'happen': 582,

'particularly': 583,

'ok': 584,

'including': 585,

'living': 586,

'save': 587,

'looked': 588,

"wouldn't": 589,

'crap': 590,

'simple': 591,

'please': 592,

'murder': 593,

'cool': 594,

'obvious': 595,

'happened': 596,

'complete': 597,

'cut': 598,

'serious': 599,

'age': 600,

'gore': 601,

'attempt': 602,

'hell': 603,

'ago': 604,

'song': 605,

'shown': 606,

'taken': 607,

'english': 608,

'james': 609,

'robert': 610,

'david': 611,

'seriously': 612,

'released': 613,

'reality': 614,

'opening': 615,

'interest': 616,

'jokes': 617,

'across': 618,

'none': 619,

'hero': 620,

'possible': 621,

'today': 622,

'exactly': 623,

'alone': 624,

'sad': 625,

'brother': 626,

'number': 627,

'saying': 628,

'career': 629,

"film's": 630,

'usually': 631,

'hours': 632,

'cinematography': 633,

'talent': 634,

'view': 635,

'annoying': 636,

'running': 637,

'yourself': 638,

'relationship': 639,

'documentary': 640,

'wish': 641,

'huge': 642,

'order': 643,

'whose': 644,

'shots': 645,

'ridiculous': 646,

'taking': 647,

'important': 648,

'light': 649,

'body': 650,

'middle': 651,

'level': 652,

'ends': 653,

'started': 654,

'call': 655,

'female': 656,

"i'll": 657,

'husband': 658,

'four': 659,

'power': 660,

'word': 661,

'turned': 662,

'major': 663,

'opinion': 664,

'change': 665,

'mostly': 666,

'usual': 667,

'silly': 668,

'scary': 669,

'rating': 670,

'beyond': 671,

'somewhat': 672,

'happy': 673,

'ones': 674,

'words': 675,

'room': 676,

'knows': 677,

'knew': 678,

'country': 679,

'disappointed': 680,

'talking': 681,

'novel': 682,

'apparently': 683,

'non': 684,

'strange': 685,

'upon': 686,

'attention': 687,

'finds': 688,

'basically': 689,

'single': 690,

'cheap': 691,

'modern': 692,

'due': 693,

'jack': 694,

'musical': 695,

'television': 696,

'problems': 697,

'miss': 698,

'episodes': 699,

'clearly': 700,

'local': 701,

'7': 702,

'british': 703,

'thriller': 704,

'talk': 705,

'events': 706,

'five': 707,

'sequence': 708,

"aren't": 709,

'class': 710,

'french': 711,

'moving': 712,

'ten': 713,

'fast': 714,

'review': 715,

'earth': 716,

'tells': 717,

'predictable': 718,

'songs': 719,

'team': 720,

'comic': 721,

'straight': 722,

'whether': 723,

'8': 724,

'die': 725,

'add': 726,

'dialog': 727,

'entertainment': 728,

'above': 729,

'sets': 730,

'future': 731,

'enjoyable': 732,

'appears': 733,

'near': 734,

'space': 735,

'easily': 736,

'hate': 737,

'soundtrack': 738,

'bring': 739,

'giving': 740,

'lots': 741,

'similar': 742,

'romantic': 743,

'george': 744,

'supporting': 745,

'release': 746,

'mention': 747,

'filmed': 748,

'within': 749,

'message': 750,

'sequel': 751,

'clear': 752,

'falls': 753,

'needs': 754,

"haven't": 755,

'dull': 756,

'suspense': 757,

'eye': 758,

'bunch': 759,

'surprised': 760,

'showing': 761,

'sorry': 762,

'tried': 763,

'certain': 764,

'easy': 765,

'working': 766,

'ways': 767,

'theme': 768,

'theater': 769,

'named': 770,

'among': 771,

"what's": 772,

'storyline': 773,

'monster': 774,

'king': 775,

'stay': 776,

'effort': 777,

'stand': 778,

'fall': 779,

'minute': 780,

'gone': 781,

'rock': 782,

'using': 783,

'9': 784,

'feature': 785,

'comments': 786,

'buy': 787,

"'": 788,

'typical': 789,

't': 790,

'sister': 791,

'editing': 792,

'tale': 793,

'avoid': 794,

'deal': 795,

'mystery': 796,

'dr': 797,

'doubt': 798,

'fantastic': 799,

'kept': 800,

'nearly': 801,

'subject': 802,

'okay': 803,

'feels': 804,

'viewing': 805,

'elements': 806,

'oscar': 807,

'check': 808,

'points': 809,

'realistic': 810,

'greatest': 811,

'means': 812,

'herself': 813,

'parents': 814,

'famous': 815,

'imagine': 816,

'rent': 817,

'viewers': 818,

'crime': 819,

'richard': 820,

'form': 821,

'peter': 822,

'actual': 823,

'lady': 824,

'general': 825,

'dog': 826,

'follow': 827,

'believable': 828,

'period': 829,

'red': 830,

'brought': 831,

'move': 832,

'material': 833,

'forget': 834,

'somehow': 835,

'begins': 836,

're': 837,

'reviews': 838,

'animation': 839,

'paul': 840,

"you've": 841,

'leads': 842,

'weak': 843,

'figure': 844,

'surprise': 845,

'sit': 846,

'hear': 847,

'average': 848,

'open': 849,

'sequences': 850,

'killing': 851,

'atmosphere': 852,

'eventually': 853,

'tom': 854,

'learn': 855,

'premise': 856,

'20': 857,

'wait': 858,

'sci': 859,

'deep': 860,

'fi': 861,

'expected': 862,

'whatever': 863,

'indeed': 864,

'particular': 865,

'note': 866,

'poorly': 867,

'lame': 868,

'dance': 869,

'imdb': 870,

'situation': 871,

'shame': 872,

'third': 873,

'york': 874,

'box': 875,

'truth': 876,

'decided': 877,

'free': 878,

'hot': 879,

"who's": 880,

'difficult': 881,

'needed': 882,

'season': 883,

'acted': 884,

'leaves': 885,

'unless': 886,

'emotional': 887,

'possibly': 888,

'romance': 889,

'sexual': 890,

'gay': 891,

'boys': 892,

'footage': 893,

'write': 894,

'western': 895,

'forced': 896,

'credits': 897,

'memorable': 898,

'doctor': 899,

'became': 900,

'reading': 901,

'otherwise': 902,

'begin': 903,

'air': 904,

'crew': 905,

'de': 906,

'question': 907,

'meet': 908,

'society': 909,

'male': 910,

'meets': 911,

"let's": 912,

'plus': 913,

'cheesy': 914,

'hands': 915,

'superb': 916,

'screenplay': 917,

'beauty': 918,

'interested': 919,

'street': 920,

'features': 921,

'perfectly': 922,

'masterpiece': 923,

'whom': 924,

'laughs': 925,

'stage': 926,

'nature': 927,

'effect': 928,

'comment': 929,

'forward': 930,

'nor': 931,

'badly': 932,

'sounds': 933,

'previous': 934,

'e': 935,

'japanese': 936,

'weird': 937,

'island': 938,

'inside': 939,

'personal': 940,

'quickly': 941,

'total': 942,

'keeps': 943,

'towards': 944,

'result': 945,

'america': 946,

'battle': 947,

'crazy': 948,

'worked': 949,

'setting': 950,

'incredibly': 951,

'earlier': 952,

'background': 953,

'mess': 954,

'cop': 955,

'writers': 956,

'fire': 957,

'copy': 958,

'unique': 959,

'dumb': 960,

'realize': 961,

'powerful': 962,

'mark': 963,

'lee': 964,

'business': 965,

'rate': 966,

'dramatic': 967,

'older': 968,

'pay': 969,

'following': 970,

'directors': 971,

'girlfriend': 972,

'joke': 973,

'plenty': 974,

'directing': 975,

'various': 976,

'creepy': 977,

'baby': 978,

'development': 979,

'appear': 980,

'brings': 981,

'front': 982,

'ask': 983,

'dream': 984,

'water': 985,

'admit': 986,

'bill': 987,

'rich': 988,

'apart': 989,

'joe': 990,

'political': 991,

'fairly': 992,

'reasons': 993,

'leading': 994,

'portrayed': 995,

'spent': 996,

'telling': 997,

'cover': 998,

'outside': 999,

'wasted': 1000,

...}

We can then use the tokenizer to convert all texts in the training-set to lists of these tokens.

In [18]:

x\_train\_tokens = tokenizer.texts\_to\_sequences(x\_train\_text)

For example, here is a text from the training-set:

In [19]:

x\_train\_text[1]

Out[19]:

'A simple comment...<br /><br />What can I say... this is a wonderful film that I can watch over and over. It is definitely one of the top ten comedies made. With a great cast, Jack Lemmon and Walter Matthau wording a perfect script by Neil Simon, based on his play.<br /><br />It is real to life situation done perfectly. If you have digital cable, one gets the menu on bottom of screen to give what is on. It usually gives this film \*\*\*% stars but in reality it deserves \*\*\*\* stars. If you really watch this film, one can tell that it will be as funny and fresh a hundred years from now.'

This text corresponds to the following list of tokens:

In [20]:

np.array(x\_train\_tokens[1])

Out[20]:

array([ 3, 591, 929, 7, 7, 48, 67, 10, 131, 11, 6,

3, 393, 19, 12, 10, 67, 103, 121, 2, 121, 9,

6, 406, 27, 4, 1, 342, 713, 1317, 90, 16, 3,

78, 174, 694, 4910, 2, 2556, 3599, 3, 399, 227, 31,

4033, 2628, 441, 20, 24, 288, 7, 7, 9, 6, 144,

5, 114, 871, 221, 922, 43, 22, 25, 3639, 1897, 27,

217, 1, 9206, 20, 1306, 4, 258, 5, 197, 48, 6,

20, 9, 631, 411, 11, 19, 405, 18, 8, 614, 9,

1003, 405, 43, 22, 62, 103, 11, 19, 27, 67, 380,

12, 9, 80, 26, 14, 152, 2, 1451, 3, 2997, 153,

36, 146])

We also need to convert the texts in the test-set to tokens.

In [21]:

x\_test\_tokens = tokenizer.texts\_to\_sequences(x\_test\_text)

## **Padding and Truncating Data**

The Recurrent Neural Network can take sequences of arbitrary length as input, but in order to use a whole batch of data, the sequences need to have the same length. There are two ways of achieving this: (A) Either we ensure that all sequences in the entire data-set have the same length, or (B) we write a custom data-generator that ensures the sequences have the same length within each batch.

Solution (A) is simpler but if we use the length of the longest sequence in the data-set, then we are wasting a lot of memory. This is particularly important for larger data-sets.

So in order to make a compromise, we will use a sequence-length that covers most sequences in the data-set, and we will then truncate longer sequences and pad shorter sequences.

First we count the number of tokens in all the sequences in the data-set.

In [22]:

num\_tokens = [len(tokens) **for** tokens **in** x\_train\_tokens + x\_test\_tokens]

num\_tokens = np.array(num\_tokens)

The average number of tokens in a sequence is:

In [23]:

np.mean(num\_tokens)

Out[23]:

221.27716

The maximum number of tokens in a sequence is:

In [24]:

np.max(num\_tokens)

Out[24]:

2209

The max number of tokens we will allow is set to the average plus 2 standard deviations.

In [25]:

max\_tokens = np.mean(num\_tokens) + 2 \* np.std(num\_tokens)

max\_tokens = int(max\_tokens)

max\_tokens

Out[25]:

544

This covers about 95% of the data-set.

In [26]:

np.sum(num\_tokens < max\_tokens) / len(num\_tokens)

Out[26]:

0.94534

When padding or truncating the sequences that have a different length, we need to determine if we want to do this padding or truncating 'pre' or 'post'. If a sequence is truncated, it means that a part of the sequence is simply thrown away. If a sequence is padded, it means that zeros are added to the sequence.

So the choice of 'pre' or 'post' can be important because it determines whether we throw away the first or last part of a sequence when truncating, and it determines whether we add zeros to the beginning or end of the sequence when padding. This may confuse the Recurrent Neural Network.

In [27]:

pad = 'pre'

In [28]:

x\_train\_pad = pad\_sequences(x\_train\_tokens, maxlen=max\_tokens,

padding=pad, truncating=pad)

In [29]:

x\_test\_pad = pad\_sequences(x\_test\_tokens, maxlen=max\_tokens,

padding=pad, truncating=pad)

We have now transformed the training-set into one big matrix of integers (tokens) with this shape:

In [30]:

x\_train\_pad.shape

Out[30]:

(25000, 544)

The matrix for the test-set has the same shape:

In [31]:

x\_test\_pad.shape

Out[31]:

(25000, 544)

For example, we had the following sequence of tokens above:

In [32]:

np.array(x\_train\_tokens[1])

Out[32]:

array([ 3, 591, 929, 7, 7, 48, 67, 10, 131, 11, 6,

3, 393, 19, 12, 10, 67, 103, 121, 2, 121, 9,

6, 406, 27, 4, 1, 342, 713, 1317, 90, 16, 3,

78, 174, 694, 4910, 2, 2556, 3599, 3, 399, 227, 31,

4033, 2628, 441, 20, 24, 288, 7, 7, 9, 6, 144,

5, 114, 871, 221, 922, 43, 22, 25, 3639, 1897, 27,

217, 1, 9206, 20, 1306, 4, 258, 5, 197, 48, 6,

20, 9, 631, 411, 11, 19, 405, 18, 8, 614, 9,

1003, 405, 43, 22, 62, 103, 11, 19, 27, 67, 380,

12, 9, 80, 26, 14, 152, 2, 1451, 3, 2997, 153,

36, 146])

This has simply been padded to create the following sequence. Note that when this is input to the Recurrent Neural Network, then it first inputs a lot of zeros. If we had padded 'post' then it would input the integer-tokens first and then a lot of zeros. This may confuse the Recurrent Neural Network.

In [33]:

x\_train\_pad[1]

Out[33]:

array([ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 3, 591, 929, 7, 7, 48, 67, 10,

131, 11, 6, 3, 393, 19, 12, 10, 67, 103, 121,

2, 121, 9, 6, 406, 27, 4, 1, 342, 713, 1317,

90, 16, 3, 78, 174, 694, 4910, 2, 2556, 3599, 3,

399, 227, 31, 4033, 2628, 441, 20, 24, 288, 7, 7,

9, 6, 144, 5, 114, 871, 221, 922, 43, 22, 25,

3639, 1897, 27, 217, 1, 9206, 20, 1306, 4, 258, 5,

197, 48, 6, 20, 9, 631, 411, 11, 19, 405, 18,

8, 614, 9, 1003, 405, 43, 22, 62, 103, 11, 19,

27, 67, 380, 12, 9, 80, 26, 14, 152, 2, 1451,

3, 2997, 153, 36, 146], dtype=int32)

## **Tokenizer Inverse Map**

For some strange reason, the Keras implementation of a tokenizer does not seem to have the inverse mapping from integer-tokens back to words, which is needed to reconstruct text-strings from lists of tokens. So we make that mapping here.

In [34]:

idx = tokenizer.word\_index

inverse\_map = dict(zip(idx.values(), idx.keys()))

Helper-function for converting a list of tokens back to a string of words.

In [35]:

**def** tokens\_to\_string(tokens):

*# Map from tokens back to words.*

words = [inverse\_map[token] **for** token **in** tokens **if** token != 0]

*# Concatenate all words.*

text = " ".join(words)

**return** text

For example, this is the original text from the data-set:

In [36]:

x\_train\_text[1]

Out[36]:

'A simple comment...<br /><br />What can I say... this is a wonderful film that I can watch over and over. It is definitely one of the top ten comedies made. With a great cast, Jack Lemmon and Walter Matthau wording a perfect script by Neil Simon, based on his play.<br /><br />It is real to life situation done perfectly. If you have digital cable, one gets the menu on bottom of screen to give what is on. It usually gives this film \*\*\*% stars but in reality it deserves \*\*\*\* stars. If you really watch this film, one can tell that it will be as funny and fresh a hundred years from now.'

We can recreate this text except for punctuation and other symbols, by converting the list of tokens back to words:

In [37]:

tokens\_to\_string(x\_train\_tokens[1])

Out[37]:

'a simple comment br br what can i say this is a wonderful film that i can watch over and over it is definitely one of the top ten comedies made with a great cast jack lemmon and walter matthau a perfect script by neil simon based on his play br br it is real to life situation done perfectly if you have digital cable one gets the menu on bottom of screen to give what is on it usually gives this film stars but in reality it deserves stars if you really watch this film one can tell that it will be as funny and fresh a hundred years from now'

## **Create the Recurrent Neural Network**

We are now ready to create the Recurrent Neural Network (RNN). We will use the Keras API for this because of its simplicity. See Tutorial #03-C for a tutorial on Keras.

In [38]:

model = Sequential()

The first layer in the RNN is a so-called Embedding-layer which converts each integer-token into a vector of values. This is necessary because the integer-tokens may take on values between 0 and 10000 for a vocabulary of 10000 words. The RNN cannot work on values in such a wide range. The embedding-layer is trained as a part of the RNN and will learn to map words with similar semantic meanings to similar embedding-vectors, as will be shown further below.

First we define the size of the embedding-vector for each integer-token. In this case we have set it to 8, so that each integer-token will be converted to a vector of length 8. The values of the embedding-vector will generally fall roughly between -1.0 and 1.0, although they may exceed these values somewhat.

The size of the embedding-vector is typically selected between 100-300, but it seems to work reasonably well with small values for Sentiment Analysis.

In [39]:

embedding\_size = 8

The embedding-layer also needs to know the number of words in the vocabulary (num\_words) and the length of the padded token-sequences (max\_tokens). We also give this layer a name because we need to retrieve its weights further below.

In [40]:

model.add(Embedding(input\_dim=num\_words,

output\_dim=embedding\_size,

input\_length=max\_tokens,

name='layer\_embedding'))

We can now add the first Gated Recurrent Unit (GRU) to the network. This will have 16 outputs. Because we will add a second GRU after this one, we need to return sequences of data because the next GRU expects sequences as its input.

In [41]:

model.add(GRU(units=16, return\_sequences=**True**))

WARNING:tensorflow:From /home/magnus/anaconda3/envs/tf-gpu/lib/python3.6/site-packages/tensorflow/python/keras/\_impl/keras/backend.py:1456: calling reduce\_sum (from tensorflow.python.ops.math\_ops) with keep\_dims is deprecated and will be removed in a future version.

Instructions for updating:

keep\_dims is deprecated, use keepdims instead

This adds the second GRU with 8 output units. This will be followed by another GRU so it must also return sequences.

In [42]:

model.add(GRU(units=8, return\_sequences=**True**))

This adds the third and final GRU with 4 output units. This will be followed by a dense-layer, so it should only give the final output of the GRU and not a whole sequence of outputs.

In [43]:

model.add(GRU(units=4))

Add a fully-connected / dense layer which computes a value between 0.0 and 1.0 that will be used as the classification output.

In [44]:

model.add(Dense(1, activation='sigmoid'))

Use the Adam optimizer with the given learning-rate.

In [45]:

optimizer = Adam(lr=1e-3)

Compile the Keras model so it is ready for training.

In [46]:

model.compile(loss='binary\_crossentropy',

optimizer=optimizer,

metrics=['accuracy'])

WARNING:tensorflow:From /home/magnus/anaconda3/envs/tf-gpu/lib/python3.6/site-packages/tensorflow/python/keras/\_impl/keras/backend.py:1557: calling reduce\_mean (from tensorflow.python.ops.math\_ops) with keep\_dims is deprecated and will be removed in a future version.

Instructions for updating:

keep\_dims is deprecated, use keepdims instead

In [47]:

model.summary()

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

layer\_embedding (Embedding) (None, 544, 8) 80000

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

gru\_1 (GRU) (None, None, 16) 1200

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

gru\_2 (GRU) (None, None, 8) 600

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

gru\_3 (GRU) (None, 4) 156

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 1) 5

=================================================================

Total params: 81,961

Trainable params: 81,961

Non-trainable params: 0

## **Train the Recurrent Neural Network**

We can now train the model. Note that we are using the data-set with the padded sequences. We use 5% of the training-set as a small validation-set, so we have a rough idea whether the model is generalizing well or if it is perhaps over-fitting to the training-set.

In [48]:

%%time

model.fit(x\_train\_pad, y\_train,

validation\_split=0.05, epochs=3, batch\_size=64)

Train on 23750 samples, validate on 1250 samples

Epoch 1/3

23750/23750 [==============================]23750/23750 [==============================] - 464s 20ms/step - loss: 0.6517 - acc: 0.6002 - val\_loss: 0.6218 - val\_acc: 0.6752

Epoch 2/3

23750/23750 [==============================]23750/23750 [==============================] - 447s 19ms/step - loss: 0.4292 - acc: 0.8102 - val\_loss: 0.6701 - val\_acc: 0.6512

Epoch 3/3

23750/23750 [==============================]23750/23750 [==============================] - 445s 19ms/step - loss: 0.3092 - acc: 0.8765 - val\_loss: 0.3182 - val\_acc: 0.8752

CPU times: user 35min 19s, sys: 2min 41s, total: 38min

Wall time: 22min 37s

Out[48]:

<tensorflow.python.keras.\_impl.keras.callbacks.History at 0x7ff79f0d6cf8>

## **Performance on Test-Set**

Now that the model has been trained we can calculate its classification accuracy on the test-set.

In [49]:

%%time

result = model.evaluate(x\_test\_pad, y\_test)

25000/25000 [==============================]25000/25000 [==============================] - 175s 7ms/step

CPU times: user 2min 59s, sys: 340 ms, total: 2min 59s

Wall time: 2min 55s

In [50]:

print("Accuracy: **{0:.2%}**".format(result[1]))

Accuracy: 86.71%

## **Example of Mis-Classified Text**

In order to show an example of mis-classified text, we first calculate the predicted sentiment for the first 1000 texts in the test-set.

In [51]:

%%time

y\_pred = model.predict(x=x\_test\_pad[0:1000])

y\_pred = y\_pred.T[0]

CPU times: user 7.01 s, sys: 0 ns, total: 7.01 s

Wall time: 6.88 s

These predicted numbers fall between 0.0 and 1.0. We use a cutoff / threshold and say that all values above 0.5 are taken to be 1.0 and all values below 0.5 are taken to be 0.0. This gives us a predicted "class" of either 0.0 or 1.0.

In [52]:

cls\_pred = np.array([1.0 **if** p>0.5 **else** 0.0 **for** p **in** y\_pred])

The true "class" for the first 1000 texts in the test-set are needed for comparison.

In [53]:

cls\_true = np.array(y\_test[0:1000])

We can then get indices for all the texts that were incorrectly classified by comparing all the "classes" of these two arrays.

In [54]:

incorrect = np.where(cls\_pred != cls\_true)

incorrect = incorrect[0]

Of the 1000 texts used, how many were mis-classified?

In [55]:

len(incorrect)

Out[55]:

121

Let us look at the first mis-classified text. We will use its index several times.

In [56]:

idx = incorrect[0]

idx

Out[56]:

13

The mis-classified text is:

In [57]:

text = x\_test\_text[idx]

text

Out[57]:

'I would like to start by saying I can only hope that the makers of this movie and it\'s sister film The Intruder (directed by the great unheralded stylist auteur that is Jopi Burnama) know in their hearts just how much pleasure they have brought to me and my friends in the sleepy north eastern town of Jarrow.<br /><br />From the opening pre credit sequence which manages to drag ever so slightly despite containing a man crashing through a window on a motorbike, the pitiless destruction of a silence lab, the introduction of one of the most simultaneously annoying and anaemic bad guys in movie history and costume design that Jean Paul Gautier would find ott and garish. Make no mistake; this is a truly unique experience. Early highlight - an explosion (get used to it, plenty more where that came from!) followed by a close up of our chubby heroine and the most hilarious line reading of the word "dad" in living memory. And then... the theme song...<br /><br />Yeah, this deserves its own paragraph. Sung by AJ, written by people who really should wish to remain anonymous, it makes the songs written for the Rocky films sound like Schubert. This is crap 80\'s hero motivation narcissism at an all time high, with choice lyrics such as "its only me and you, its come down to the wire" and much talk of having to "cross the line" (it\'ll make sense in time - our hero cares little for the boundaries of bona fida police work) abounding. Not to mention the Indonesian Supremes cooing the film\'s title seductively. At this point anyone wishing to switch off officially has no pulse.<br /><br />Our hero is Semitic cop Peter Goldson (essayed brilliantly by Intruder star Peter O\'Brien), the "stabilizer" of the title. The man\'s bull in a china shop approach to crime fighting and particularly his less than inconspicuous undercover work truly leaves much to be desired, but he is without question an entertaining guide through the mean streets of downtown Jakarta, with local sleaze ball connection Captain Johnny in tow, as well as Peter\'s own waste of space partner in fashion crime Sylvia Nash, who does little. So many highlights, so little time - the "slide please" arrogance of Peter\'s not all too convincingly argued case against chief baddie Greg Rainmaker (Intruder fans will know hirsute slimy bastard Craig Gavin as the monstrous John White - helluva name eh? No! Oh well...), the x marks the spot location map stupidity, our hero taking horrible advantage of heroine Tina Probost during a moment of weakness on her behalf, the latter turning up at a sting operation dressed like a member of a particularly flamboyant dancing troop. And believe me that barely covers it.<br /><br />There wasn\'t even time to go into the plot revolving around the hunt for a drug detection system and a kidnapped professor with an alarming but commendable amount of national pride. Or our hero turning up at a funeral dressed as if an extra on Boogie Nights. Or the absolutely hysterical craic between Captain Johnny and Goldson - two guys have never made more heavy weather of buddy buddy shtick than these clowns. The trowel was possibly too subtle me thinks.<br /><br />Ah it tails off people, and you never thought scenes of wanton destruction and general mayhem could be so unbelievably boring, but the character interaction is stupendous, the dialogue truly priceless and the incompetence on show somehow endearing. Oh and the shoes people - watch out for the shoes!'

These are the predicted and true classes for the text:

In [58]:

y\_pred[idx]

Out[58]:

0.08332923

In [59]:

cls\_true[idx]

Out[59]:

1.0

## **New Data**

Let us try and classify new texts that we make up. Some of these are obvious, while others use negation and sarcasm to try and confuse the model into mis-classifying the text.

In [60]:

text1 = "This movie is fantastic! I really like it because it is so good!"

text2 = "Good movie!"

text3 = "Maybe I like this movie."

text4 = "Meh ..."

text5 = "If I were a drunk teenager then this movie might be good."

text6 = "Bad movie!"

text7 = "Not a good movie!"

text8 = "This movie really sucks! Can I get my money back please?"

texts = [text1, text2, text3, text4, text5, text6, text7, text8]

We first convert these texts to arrays of integer-tokens because that is needed by the model.

In [61]:

tokens = tokenizer.texts\_to\_sequences(texts)

To input texts with different lengths into the model, we also need to pad and truncate them.

In [62]:

tokens\_pad = pad\_sequences(tokens, maxlen=max\_tokens,

padding=pad, truncating=pad)

tokens\_pad.shape

Out[62]:

(8, 544)

We can now use the trained model to predict the sentiment for these texts.

In [63]:

model.predict(tokens\_pad)

Out[63]:

array([[0.868934 ],

[0.72526425],

[0.33099633],

[0.49190348],

[0.3054021 ],

[0.14959489],

[0.5235635 ],

[0.21565402]], dtype=float32)

A value close to 0.0 means a negative sentiment and a value close to 1.0 means a positive sentiment. These numbers will vary every time you train the model.

## **Embeddings**

The model cannot work on integer-tokens directly, because they are integer values that may range between 0 and the number of words in our vocabulary, e.g. 10000. So we need to convert the integer-tokens into vectors of values that are roughly between -1.0 and 1.0 which can be used as input to a neural network.

This mapping from integer-tokens to real-valued vectors is also called an "embedding". It is essentially just a matrix where each row contains the vector-mapping of a single token. This means we can quickly lookup the mapping of each integer-token by simply using the token as an index into the matrix. The embeddings are learned along with the rest of the model during training.

Ideally the embedding would learn a mapping where words that are similar in meaning also have similar embedding-values. Let us investigate if that has happened here.

First we need to get the embedding-layer from the model:

In [64]:

layer\_embedding = model.get\_layer('layer\_embedding')

We can then get the weights used for the mapping done by the embedding-layer.

In [65]:

weights\_embedding = layer\_embedding.get\_weights()[0]

Note that the weights are actually just a matrix with the number of words in the vocabulary times the vector length for each embedding. That's because it is basically just a lookup-matrix.

In [66]:

weights\_embedding.shape

Out[66]:

(10000, 8)

Let us get the integer-token for the word 'good', which is just an index into the vocabulary.

In [67]:

token\_good = tokenizer.word\_index['good']

token\_good

Out[67]:

49

Let us also get the integer-token for the word 'great'.

In [68]:

token\_great = tokenizer.word\_index['great']

token\_great

Out[68]:

78

These integertokens may be far apart and will depend on the frequency of those words in the data-set.

Now let us compare the vector-embeddings for the words 'good' and 'great'. Several of these values are similar, although some values are quite different. Note that these values will change every time you train the model.

In [69]:

weights\_embedding[token\_good]

Out[69]:

array([0.86528164, 0.6867993 , 0.4362397 , 0.66128314, 0.11546915,

0.94507647, 0.32628497, 0.535881 ], dtype=float32)

In [70]:

weights\_embedding[token\_great]

Out[70]:

array([ 1.0691622 , 1.124244 , -0.04477464, -0.05861434, 0.16965319,

1.2626944 , 0.76136374, -0.00998422], dtype=float32)

Similarly, we can compare the embeddings for the words 'bad' and 'horrible'.

In [71]:

token\_bad = tokenizer.word\_index['bad']

token\_horrible = tokenizer.word\_index['horrible']

In [72]:

weights\_embedding[token\_bad]

Out[72]:

array([ 0.31903917, 0.53934103, 1.3727672 , 1.4083829 , 0.8475107 ,

-0.22946651, 0.0251075 , 0.77032244], dtype=float32)

In [73]:

weights\_embedding[token\_horrible]

Out[73]:

array([ 0.47915924, 0.12226178, 0.90192014, 0.742338 , 0.58730644,

0.32736972, -0.17633988, 1.3744307 ], dtype=float32)

### **Sorted Words**

We can also sort all the words in the vocabulary according to their "similarity" in the embedding-space. We want to see if words that have similar embedding-vectors also have similar meanings.

Similarity of embedding-vectors can be measured by different metrics, e.g. Euclidean distance or cosine distance.

We have a helper-function for calculating these distances and printing the words in sorted order.

In [74]:

**def** print\_sorted\_words(word, metric='cosine'):

*"""*

*Print the words in the vocabulary sorted according to their*

*embedding-distance to the given word.*

*Different metrics can be used, e.g. 'cosine' or 'euclidean'.*

*"""*

*# Get the token (i.e. integer ID) for the given word.*

token = tokenizer.word\_index[word]

*# Get the embedding for the given word. Note that the*

*# embedding-weight-matrix is indexed by the word-tokens*

*# which are integer IDs.*

embedding = weights\_embedding[token]

*# Calculate the distance between the embeddings for*

*# this word and all other words in the vocabulary.*

distances = cdist(weights\_embedding, [embedding],

metric=metric).T[0]

*# Get an index sorted according to the embedding-distances.*

*# These are the tokens (integer IDs) for words in the vocabulary.*

sorted\_index = np.argsort(distances)

*# Sort the embedding-distances.*

sorted\_distances = distances[sorted\_index]

*# Sort all the words in the vocabulary according to their*

*# embedding-distance. This is a bit excessive because we*

*# will only print the top and bottom words.*

sorted\_words = [inverse\_map[token] **for** token **in** sorted\_index

**if** token != 0]

*# Helper-function for printing words and embedding-distances.*

**def** \_print\_words(words, distances):

**for** word, distance **in** zip(words, distances):

print("**{0:.3f}** - **{1}**".format(distance, word))

*# Number of words to print from the top and bottom of the list.*

k = 10

print("Distance from '**{0}**':".format(word))

*# Print the words with smallest embedding-distance.*

\_print\_words(sorted\_words[0:k], sorted\_distances[0:k])

print("...")

*# Print the words with highest embedding-distance.*

\_print\_words(sorted\_words[-k:], sorted\_distances[-k:])

We can then print the words that are near and far from the word 'great' in terms of their vector-embeddings. Note that these may change each time you train the model.

In [75]:

print\_sorted\_words('great', metric='cosine')

Distance from 'great':

0.000 - great

0.016 - touching

0.017 - arguments

0.025 - nevertheless

0.031 - elmer

0.032 - 8

0.036 - ritter

0.037 - juliet

0.041 - randy

0.045 - afterward

...

1.057 - rubbish

1.060 - dull

1.064 - disappointing

1.069 - unlikeable

1.078 - uninspired

1.083 - lacks

1.188 - worst

1.225 - waste

1.247 - awful

1.282 - terrible

Similarly, we can print the words that are near and far from the word 'worst' in terms of their vector-embeddings.

In [76]:

print\_sorted\_words('worst', metric='cosine')

Distance from 'worst':

0.000 - worst

0.047 - embarrassingly

0.053 - terrible

0.094 - retarded

0.095 - poor

0.095 - stereotyping

0.096 - uninspired

0.099 - awful

0.100 - severed

0.108 - lacks

...

1.167 - restraint

1.168 - available

1.176 - foremost

1.188 - great

1.193 - mesmerizing

1.222 - highly

1.229 - exploration

1.239 - delightful

1.268 - wonderfully

1.323 – 7

## **Conclusion**

This tutorial showed the basic methods for doing Natural Language Processing (NLP) using a Recurrent Neural Network with integer-tokens and an embedding layer. This was used to do sentiment analysis of movie reviews from IMDB. It works reasonably well if the hyper-parameters are chosen properly. But it is important to understand that this is not human-like comprehension of text. The system does not have any real understanding of the text. It is just a clever way of doing pattern-recognition.

## **Exercises**

These are a few suggestions for exercises that may help improve your skills with TensorFlow. It is important to get hands-on experience with TensorFlow in order to learn how to use it properly.

You may want to backup this Notebook before making any changes.

* Run more training-epochs. Does it improve performance?
* If your model overfits the training-data, try using dropout-layers and dropout inside the GRU.
* Increase or decrease the number of words in the vocabulary. This is done when the Tokenizer is initialized. Does it affect performance?
* Increase the size of the embedding-vectors to e.g. 200. Does it affect performance?
* Try varying all the different hyper-parameters for the Recurrent Neural Network.
* Use Bayesian Optimization from Tutorial #19 to find the best choice of hyper-parameters.
* Use 'post' for padding and truncating in pad\_sequences(). Does it affect the performance?
* Use individual characters instead of tokenized words as the vocabulary. You can then use one-hot encoded vectors for each character instead of using the embedding-layer.
* Use model.fit\_generator() instead of model.fit() and make your own data-generator, which creates a batch of data using a random subset of x\_train\_tokens. The sequences must be padded so they all match the length of the longest sequence.
* Explain to a friend how the program works.

## **License (MIT)**

Copyright (c) 2018 by [Magnus Erik Hvass Pedersen](http://www.hvass-labs.org/)

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.