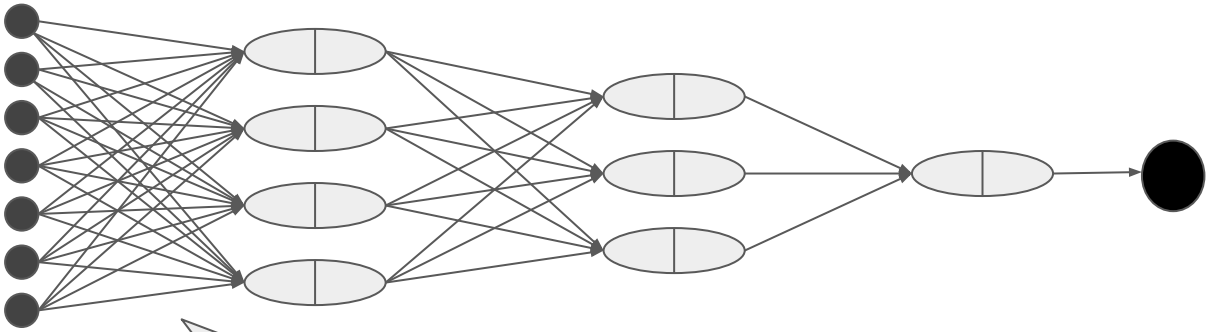
TX00DH43-3001
Introduction to Deep Learning

peter.hjort@metropolia.fi

Recurrent neural network (RNN)

So far we have been processing input to the network without any notion of sequentiality:



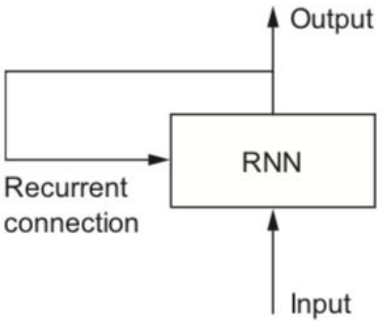
No notion of ordering when layers of network are connected. (Remember we first compute weighted sum of inputs in a neuron).

Recurrent neural network (RNN)

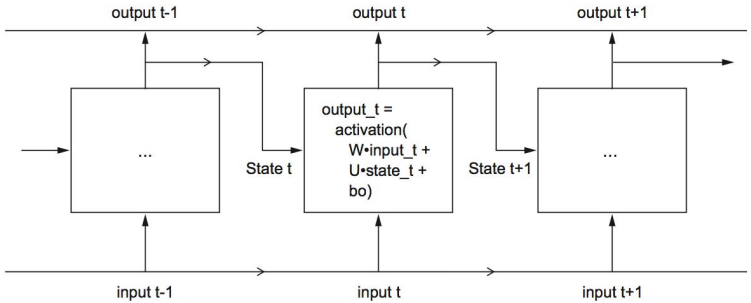
Sequential data is, however, abundant:

- Text data is by nature sequential (interpreted in word or character level)
- Voice data is sequential
- Video can be interpreted as sequence of frames
- Processes in nature, such as temperatures etc exhibit sequentiality
- Human behaviour can often be interpreted as sequences

RNN

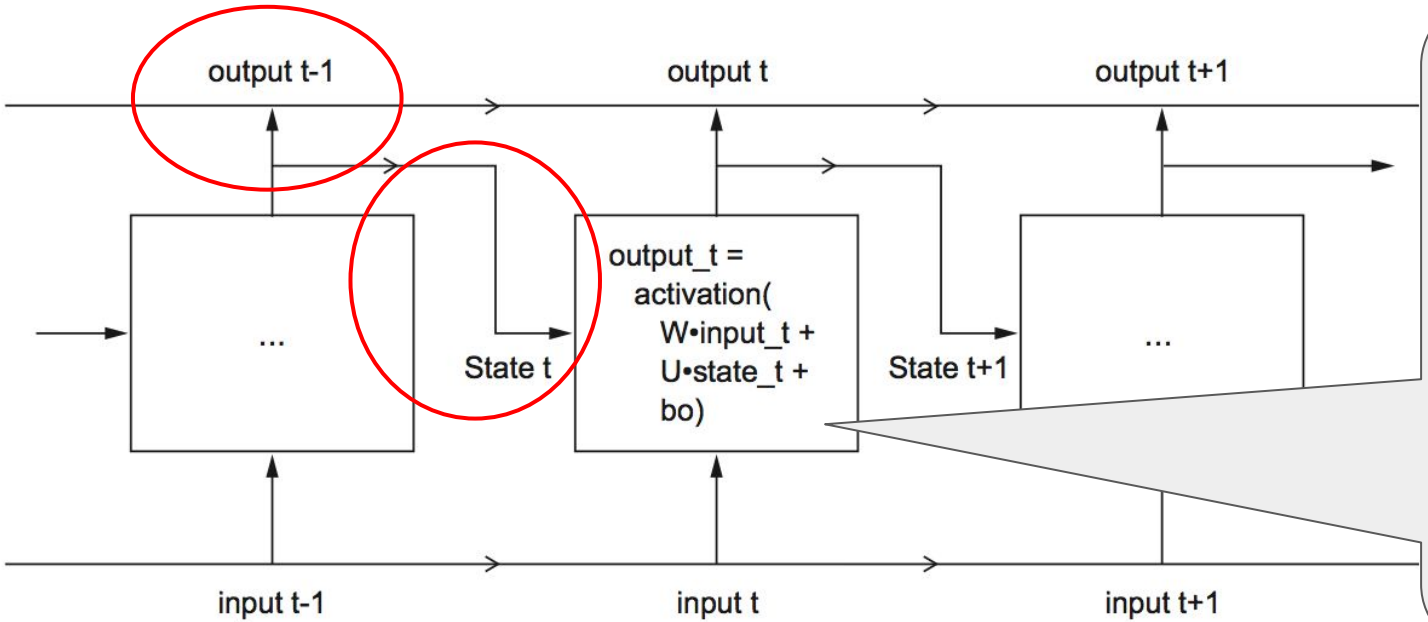


RNN processes its input by iterating over input sequence elements and maintains a state (a memory) that is determined by the data it has processed so far. Note that state is reset between two input sequences.



RNN **unrolled** over timesteps:
computation proceeds from left to right

Recurrent neural network (RNN)



When computing output for position `t`, input `t` and output from previous position are combined. Note that multiplier matrices `W` and `U` as well as bias `bo` are **shared across the layer (= over all timesteps)**.

SimpleRNN in Keras

```
model1 = Sequential()
model1.add(SimpleRNN(20, input_shape=(40,5,)))
```

```
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn_6 (SimpleRNN)     (None, 20)                520
=================================================================
Total params: 520
Trainable params: 520
Non-trainable params: 0
_________________________________________________________________
```

SimpleRNN parameter count

of output features: 20

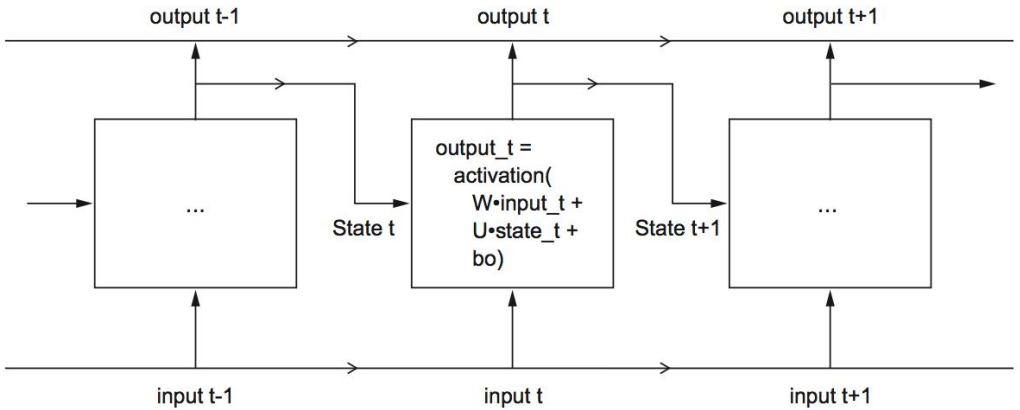
# of input features: 5

Matrix `W` dimensions: 20 x 5

Matrix `U` dimensions: 20 x 20

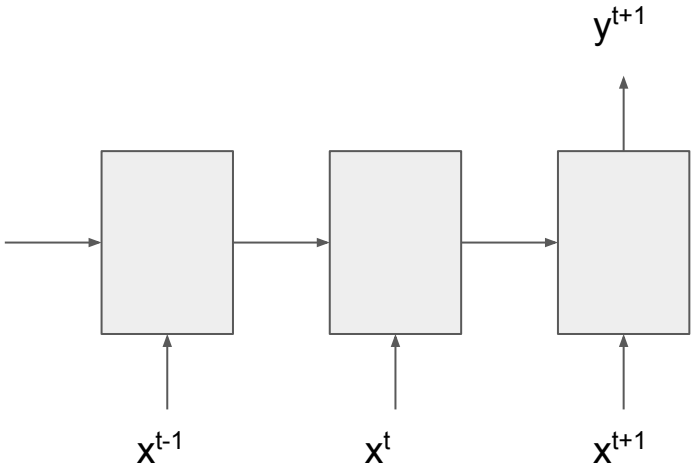
Bias `bo` dimensions: 1 x 20

Total: 100 + 400 + 20



Again, remember the parameters are shared over timesteps = `W`, `U`, `bo` are identical for all time steps.

RNN architectures: many-to-one

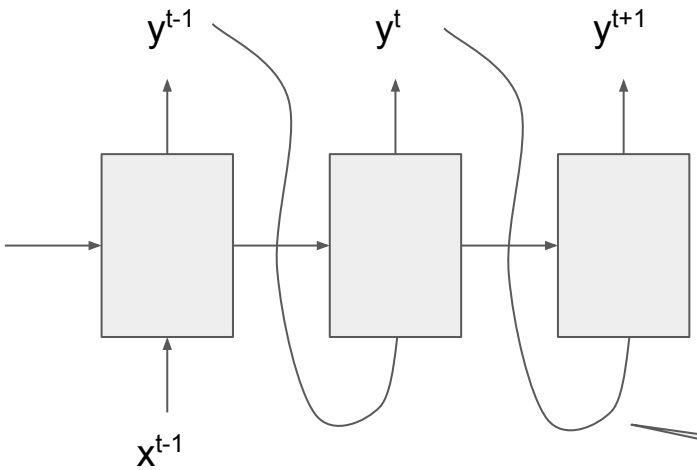


Output at the end of sequence only (`return_sequences=False`). Note that the output depends on the whole input sequence.

Application examples for many-to-one

- Sentiment analysis: from a product review text predict the number of stars, or positive/negative
- Recognize activity from video: running/jumping/etc

RNN architectures: one-to-many



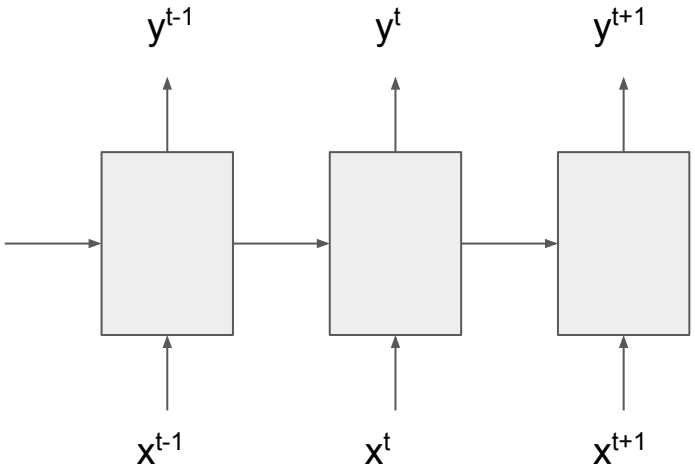
Output a value from each element. Internally assign output from previous step to input of next step.

Starting from an element (or from empty), generate a sequence:

- Generate words or music

Curved lines represent connection from output of preceding node to input of next node

RNN architectures: many-to-many

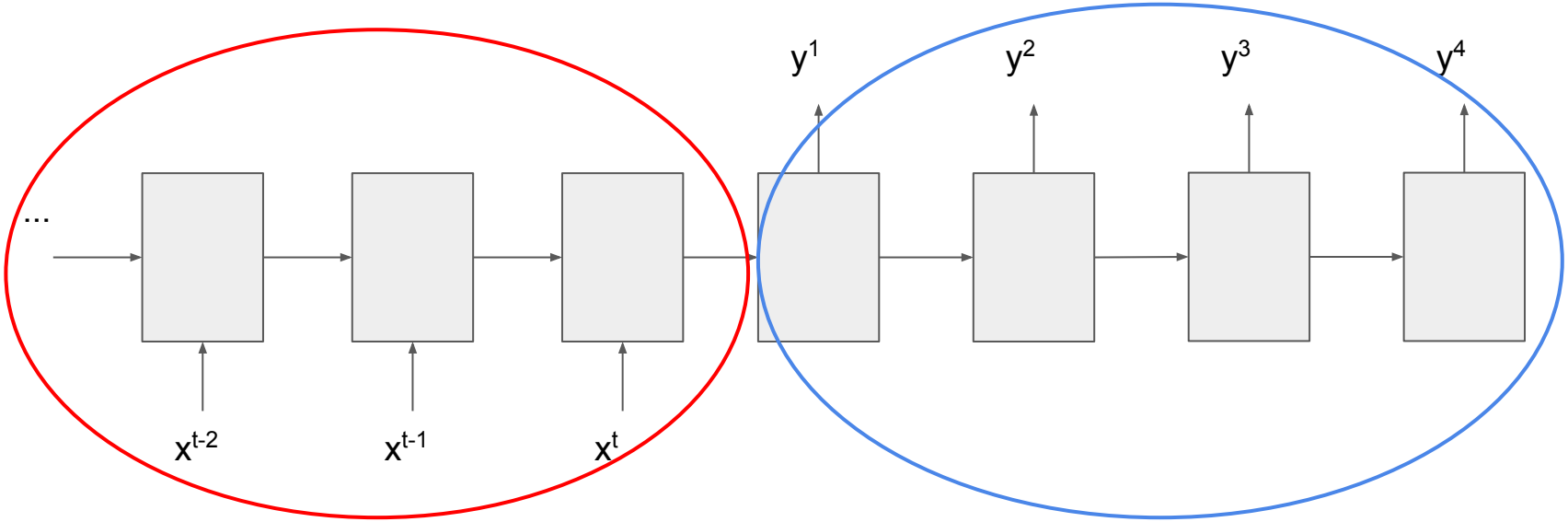


Some typical examples for many-to-many architectures include:

- Speech-to-text
- Machine translation
- Image captioning - based on an image, produce text to caption it

Note that for many of these problems input sequence length ≠ output sequence length. How to deal with that?

RNN architectures: many-to-many



- Input sequence length ≠ output sequence length
- Learn **representation** of input and generate output sequence from that
- Two parts: encoder (reads input, encodes it into internal representation) and decoder (reads internal representation to produce output)

Stacking RNN units

```
model.add(LSTM(seqlen, input_shape=(seqlen, 1,), return_sequences=True))
model.add(LSTM(seqlen))
model.add(Dense(1, activation='sigmoid'))
```

When stacking RNN layers one needs to return a sequence from preceding layer for the next layer to work (`return_sequences=True`). Effect of stacking is similar to stacking any layers - more parameters; more expressive (and complex) model.

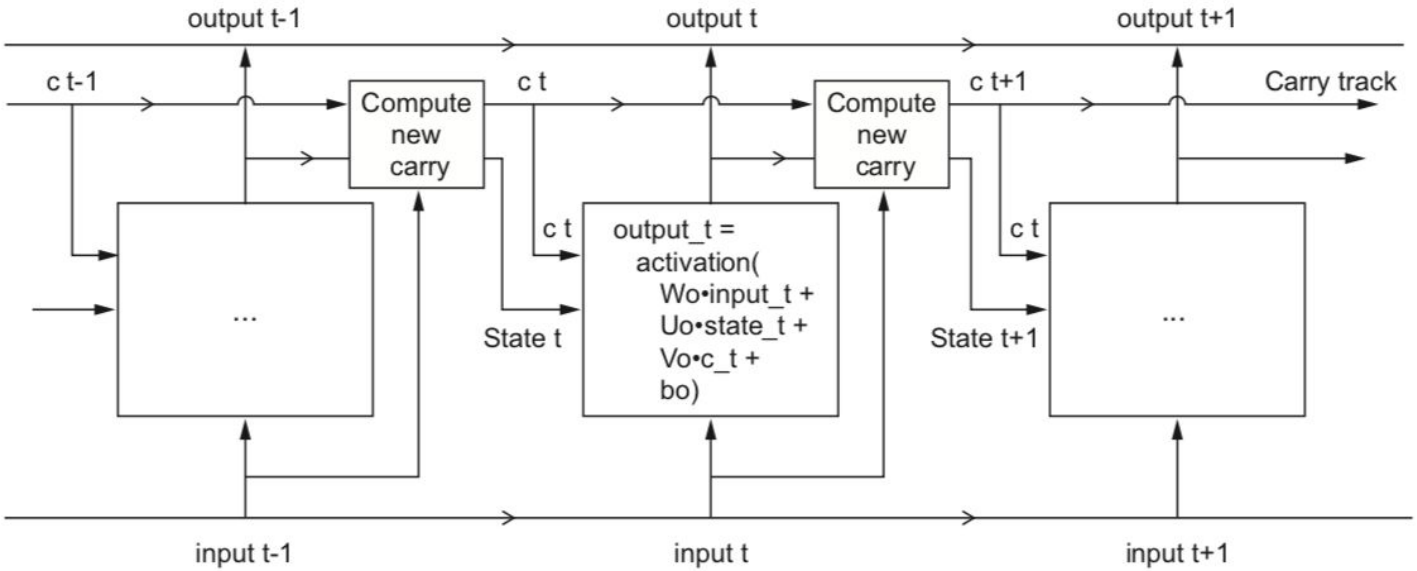
RNN units

SimpleRNN unit is not used in practice:

- Suffers from vanishing gradients phenomenon
- Does not maintain information over longer sequences well

Units with explicit memory (or carry), a direct link with memory data to next steps in the sequence, and more flexible ways to control maintaining the memory: **LSTM** (Long Short-Term Memory) and **GRU** (Gated Recurrent Unit).

LSTM (Long Short-Term Memory)



Chollet, page 204

GRU (Gated Recurrent Unit)

A simplification of LSTM: carry and state lines are not separately controlled.

In practice less complex computationally than LSTM.

Note, however, that computation with both LSTM and GRU elements does not speed up very much when using a GPU - the inherent looping inside RNN elements does not parallelize very well.

Preprocessing text

Use `Tokenizer` class (https://keras.io/preprocessing/text/#tokenizer) in `keras.preprocessing.text`:

- Start with `fit_on_texts()` to remove punctuation and build a word - index mapping. Also word frequencies are available after `fit_on_texts()`.

```
texts = [ "Where is the cat.", "The cat sat on the moon.", "The moon is made of cheese."]

tokenizer = Tokenizer(num_words=20)
tokenizer.fit_on_texts(texts)
```

```
tokenizer.word_index:

1:the, 2:is, 3:cat, 4:moon, 5:where, 6:sat, 7:on, 8:made, 9:of, 10:cheese
```

Preprocessing text

Get sequences of ints from texts with `texts_to_sequences()`

```
sequences = tokenizer.texts_to_sequences(texts)
```

```
sequences:

[[5, 2, 1, 3], [1, 3, 6, 7, 1, 4], [1, 4, 2, 8, 9, 10]]
```

Turn sequences into one-hot encoded sequences:

```
one_hot_encoding = tokenizer.texts_to_matrix(texts, mode='binary')
```

```
one_hot_encoding:

[[ 0.  1.  1.  1.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. 0.]
 [ 0.  1.  0.  1.  1.  0.  1.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0. 0.]
 [ 0.  1.  1.  0.  1.  0.  0.  0.  1.  1.  1.  0.  0.  0.  0.  0.  0. 0.]]
```

Preprocessing text

If one-hot encoded words are needed, use to_categorical on sequences:

```
keras.utils.to_categorical(s, len(tokenizer.word_index)+1) # s is sequence for 1st sentence
```

```
[[ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.]]
```

Word embeddings

One-hot encoding has some undesirable properties:

- Dimension = size of dictionary
- Each word has the same distance to all other words - distance information is quite useless

How about mapping each word to a vector of dimension < size of dictionary?

Word embeddings

	female	juicy	large	edible	yellowish
king	0.01	0.05	0.6	0.04	0.6
queen	0.95	0.04	0.55	0.03	0.64
uncle	0.03	0.01	0.7	0.02	0.62
aunt	0.98	0.03	0.6	0.04	0.66
apple	0.1	0.6	0.1	0.95	0.2
orange	0.1	0.8	0.07	0.99	0.8

Word embeddings - properties

king - queen + aunt = [0.04 0.04 0.65 0.05 0.62]

uncle = [0.03 0.01 0.7 0.02 0.62]

So king - queen + aunt ≈ uncle and king - queen ≈ uncle - aunt

“King is to queen like uncle is to aunt”

(Try out your favorite samples for near words, analogies etc with http://bionlp-www.utu.fi/wv_demo/)

Word embeddings in practice

Usually the embedding vectors are 200-1000 dimensional. Compare this with 10000-30000 word dictionaries leading to very high-dimensional one-hot vectors

Embeddings are created by training a network (starting with random vectors/weights), or, more directly, by computing a correlation matrix.

Embedding dimensions are not interpretable in general (no “easy” dimensions like in the example).

The relationships of vectors, however, do hold. (king - queen ≈ uncle - aunt).

Pre-computed (based on huge text corpuses) embeddings are available (and are usually used in text-related problems).

Word embeddings - cosine similarity

With word embeddings (and in other application areas like document classification) similarity of the embedding vectors is often computed using cosine similarity:

cdist(a,b) = cos(angle between a and b) =

np.dot(a,b) / (sqrt(np.dot(a,a)) * sqrt(np.dot(b,b)))

This measurement of similarity ignores vector length, only the angle between vectors contributes to cos similarity.

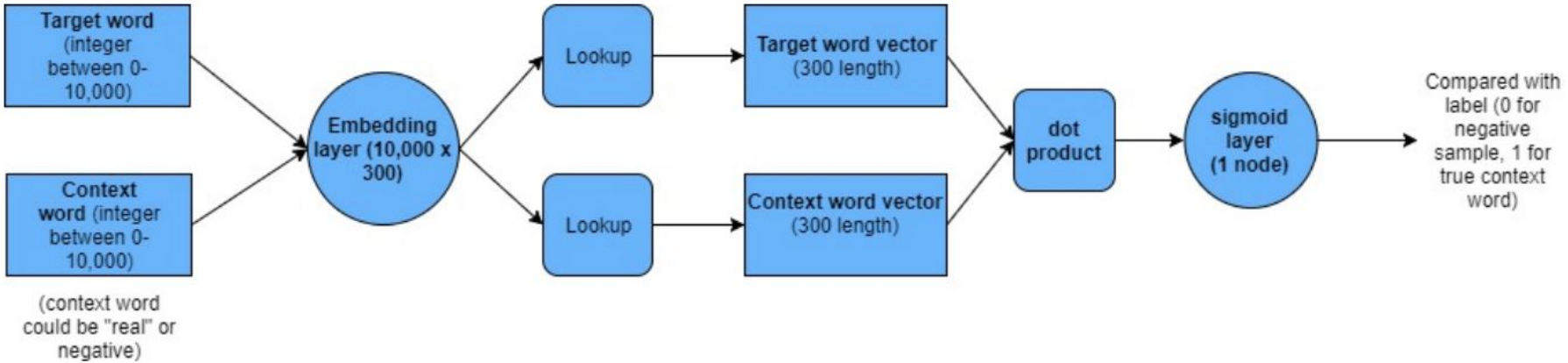
Learning embeddings

Self-supervised learning task where the real target is to learn a representation.
Self-supervised - no labeling needed; rather use text corpus properties as targets.

Positive/negative context sampling (idea behind word2vec):

- create pairs (target_word, context_word) with true (label) value 1 if context_word is within a given distance (window_size) from the target_word, 0 label value otherwise.

Learning embeddings



From http://adventuresinmachinelearning.com/word2vec-keras-tutorial/

Word embeddings in Keras

Keras has `Embedding` layer which can be used for computing embeddings from input text samples. See Chollet chapter 6.1.2 section “learning word embeddings with the embedding layer”.

Keras `Embedding` layer can also be initialized with existing set of weights - **transfer learning**. See Chollet chapter 6.1.3.

Additional reading

https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html

http://adventuresinmachinelearning.com/word2vec-keras-tutorial/ (how to train word2vec style embeddings)

Reading list for exam 5.3

session06.pdf

Chollet: 6.1, 6.2, 6.3

Exercise: reuters newswires classification

Take a look at Chollet chapter 3.5 for an example on classifying reuters newswires (note: the example is available as notebook from github, see https://github.com/fchollet/deep-learning-with-python-notebooks). Modify the example to use learned word embeddings by including an Embedding layer in the model. Take a look at example in Chollet chapter 6.1 (also available as a notebook) for how to do this. Modify the model further to use RNN layer (use LSTM or GRU). What can you say about accuracy compared to the original example?