IT00DP82-3007 Artificial Intelligence and Machine Learning

peter.hjort@metropolia.fi

Note: please include string IT00DP82 in the subject field of any email correspondence

Keras functional API

In addition to the sequential API used this far, Keras has an API that allows for more flexible model definition - functional API.

In functional API layers behave like functions; they are called with parameter (the previous layer result), and they return a value (result of the layer):

```
x = layers.Dense(32, activation='relu')(input_tensor)
```

```
x = layers.Dense(32, activation='relu')(x)
```

It makes sense to define inputs and outputs separately:

```
inputs = Input(shape=(64,))
```

```
outputs = layers.Dense(10, activation='softmax')(x)
```

Keras functional API

Layer sharing: if a layer is instantiated once and then used in different branches of the network, the weights in the layer are shared - they are learned “together” in the branches.

```
mylayer = layers.Dense(32, activation='relu')

x = mylayer(x)
```

To create the model, call `Model()` and give the input and output; the network is now fully defined, and can be compiled, fitted etc. in the normal way:

```
model = Model(inputs, outputs, 'mymodel')
```

Multi-input networks

With functional API it is possible to create networks that have more than one input data sets:

```
input1 = Input(shape=(16,))
x1 = layers.Dense(32, activation='relu')(input1)
x1 = layers.Dense(32, activation='relu')(x1)

input2 = Input(shape=(64,))
x2 = layers.Dense(64, activation='relu')(input2)
x2 = layers.Dense(128, activation='relu')(x2)

common = layers.concatenate([x1, x2], axis=-1)
output = layers.Dense(10, activation='softmax')(common)

model2 = Model([input1, input2], output)
model2.compile(optimizer='adam',
               loss='categorical_crossentropy',
               metrics=['acc'])
model2.summary()
```

Multi-output networks

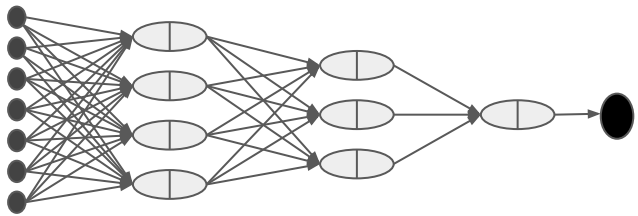
With functional API it is possible to create networks that have more than one output:

```
input1 = Input(shape=(28,28,1,))
x1 = layers.Conv2D(filters=6, kernel_size=5, strides=1,
                   use_bias=False, padding='valid', activation='relu')(input1)
...
x1 = layers.Flatten()(x1)

category = layers.Dense(100, activation='relu')(x1)
cat_output = layers.Dense(10, activation='softmax')(category)

loops = layers.Dense(100, activation='relu')(x1)
loops_output = layers.Dense(1, activation='sigmoid')(loops)

model = Model(input1, [cat_output, loops_output])
model.compile(optimizer='adam',
              loss=['categorical_crossentropy','binary_crossentropy'],
              metrics=['acc'])
```

Note different loss functions for the two branches.

Recurrent neural network (RNN)

So far we have been processing input to the network without any notion of sequentiality (although with CNNs for image tasks 2D data is spatial structure is taken into account):



No concept of ordering for inputs - same applies to intermediate outputs.

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
- We will now concentrate on text data →

Text processing

Chollet 6.1

Text processing: Preprocessing

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) # limit vocabulary to 20 words
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 sequence (sentence) into one-hot encoded version by use of to_categorical():

```
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.]]
```

Lemmatization and other more advanced preprocessing

Good, better, best all come from word “good” → replace with good.

In Finnish language identify cases - pöydällä, pöytään, etc → pöytä

Stopword elimination - strip “a”, “the”, “and” etc

NLTK (Natural Language Toolkit) at https://www.nltk.org/ has tools for many of the tasks (for Finnish language might need to look up specific solutions).

Word embeddings

Word embeddings

One-hot encoding has some very undesirable properties:

- Dimension = size of dictionary (could be 10000 - 60000)
- 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?

Could there be a mapping that is more compact and also captures something about the relationships between words?

Word embeddings - imaginary example

	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 = [-0.94 0.01 0.05 0.01 -0.04]

uncle - aunt = [-0.95 -0.02 0.1 -0.02 -0.04]

So 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://epsilon-it.utu.fi/wv_demo/)

Word embeddings in practice

Usually the embedding vectors are 200-1000 dimensional. Compare this with 10000-50000 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).

Learning embeddings

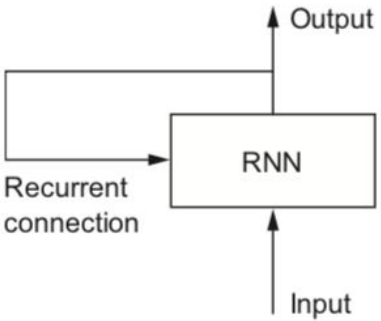
Learning embeddings is self-supervised learning task where the real target is to learn a representation. Self-supervised - no labeling needed; rather use text corpus properties as targets.

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”.

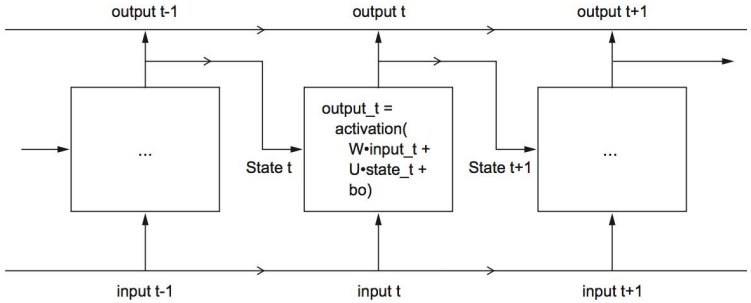
Keras `Embedding` layer can also be initialized with existing set of weights - **transfer learning**. See Chollet chapter 6.1.3.

Pre-computed (based on huge text corpuses) embeddings are available (and are often used in text-related problems). word2vec and Glove are popular examples, for word2vec trained with Finnish corpus, see https://bionlp.utu.fi/finnish-internet-parsebank.html

Deep network layer type for sequential data - RNN

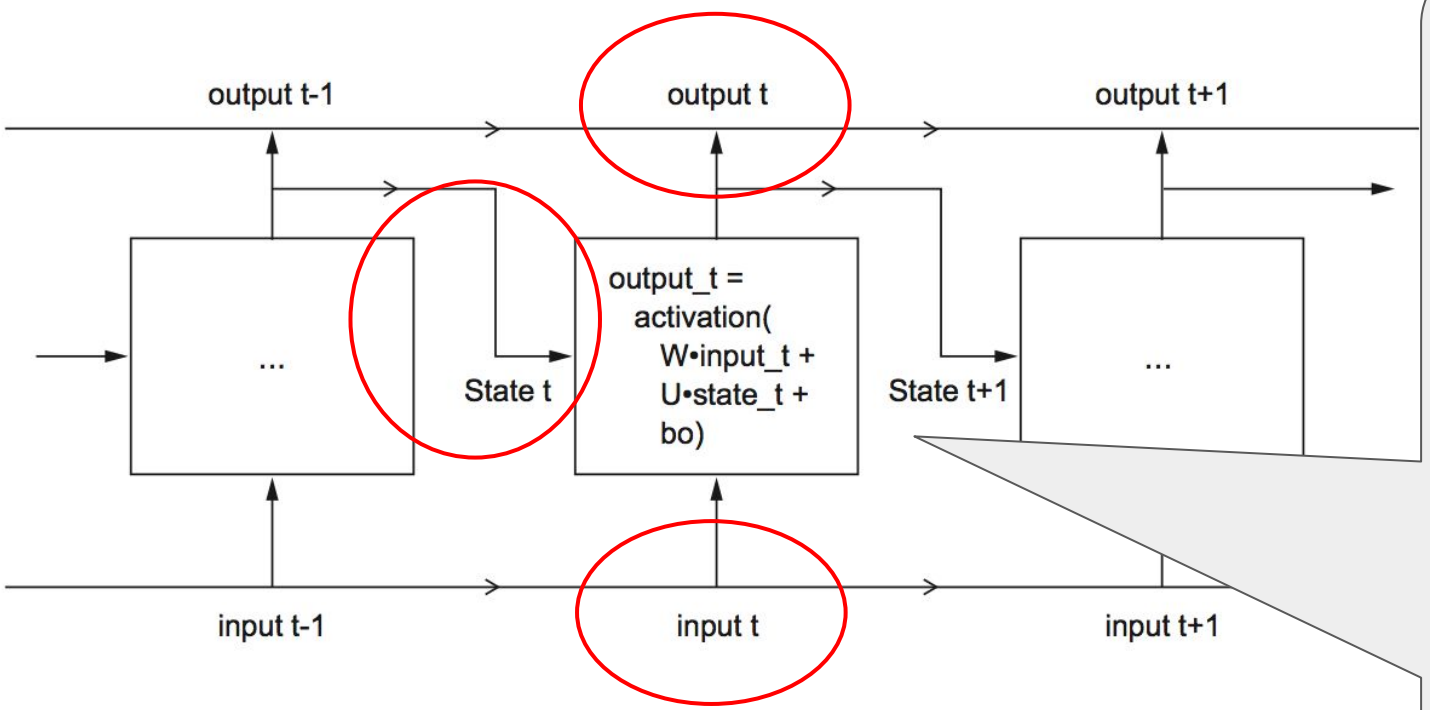


RNN processes its input by iterating over input sequence elements one by one and maintains a state (a memory) that is determined by the data it has processed so far.



RNN **unrolled** over timesteps: computation proceeds from left to right (from the beginning of the sequence to the end). When sequences are long, this means that the network is quite deep.

Recurrent neural network (RNN)



When computing output for position `t`, input `t` and output from previous position are combined. Multiplier matrices `W` and `U` as well as bias `bo` are for the unrolled node, they are **shared across the layer (= over all timesteps)** when unrolling. All signals flowing in the network are vectors.

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

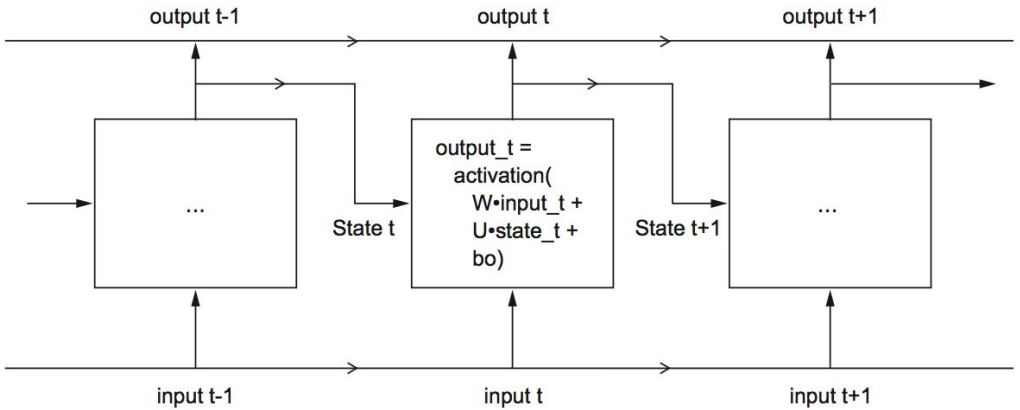
# of input features: 5

Matrix `W` dimensions: 20 x 5

Matrix `U` dimensions: 20 x 20

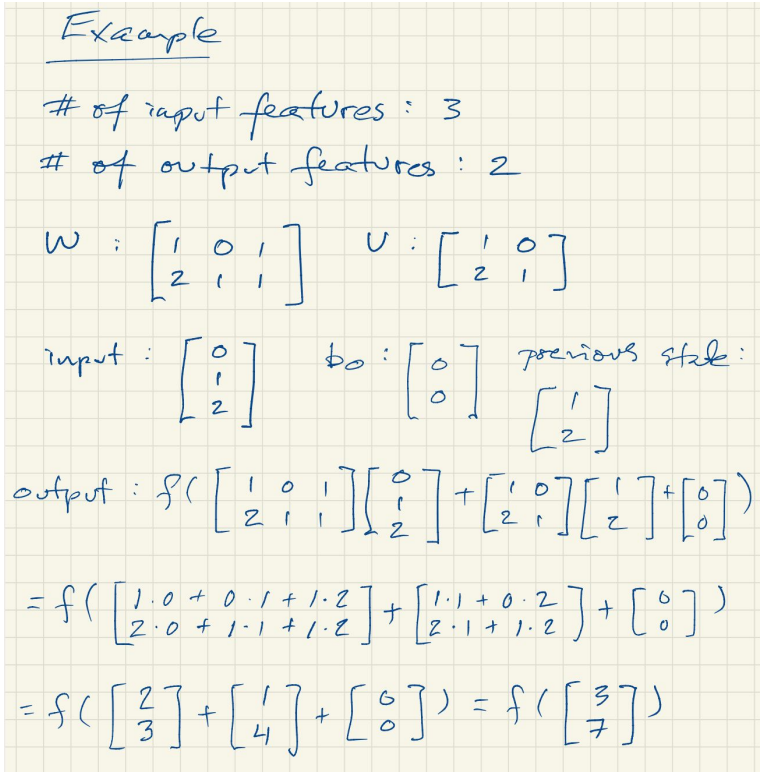
Bias `bo` dimensions: 1 x 20

Total: 100 + 400 + 20



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

SimpleRNN example calculation

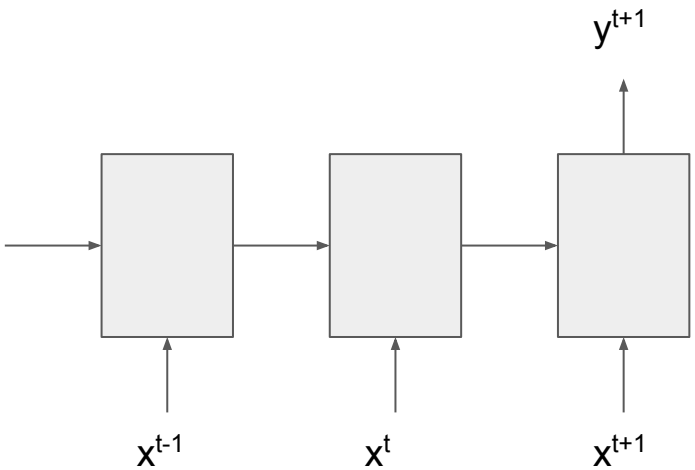


Here we are assuming W, U and bo have some values (learned during previous steps), and previous state (state t in diagram) has a value.

The next output is computed based on those values, an activation function (in the example f()) is applied to that which gives the next output (and state for next step).

RNN architectures

RNN architectures: many-to-one

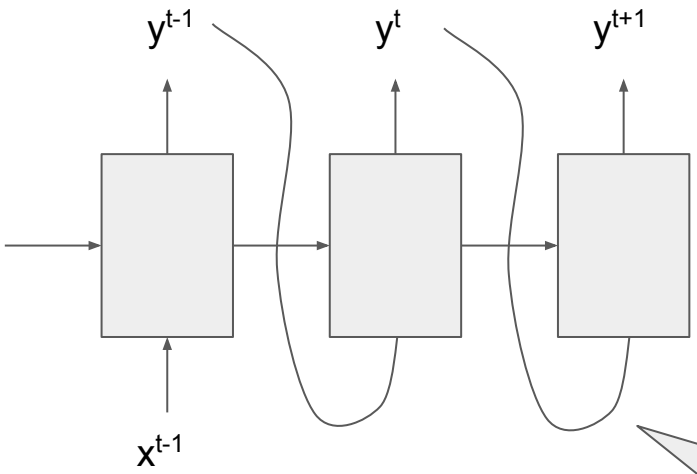


Output at the end of sequence only (`return_sequences=False` in Keras). The output depends on the whole input sequence.

Application examples for many-to-one

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

RNN architectures: one-to-many



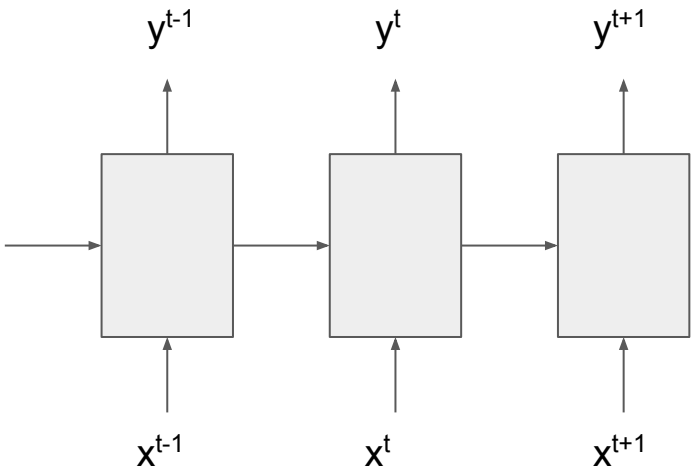
Output a value from each element. Internally assign output from previous step to input of next step.

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

- Generate words or music (sometimes start from a few elements, especially when generating text).

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

RNN architectures: many-to-many

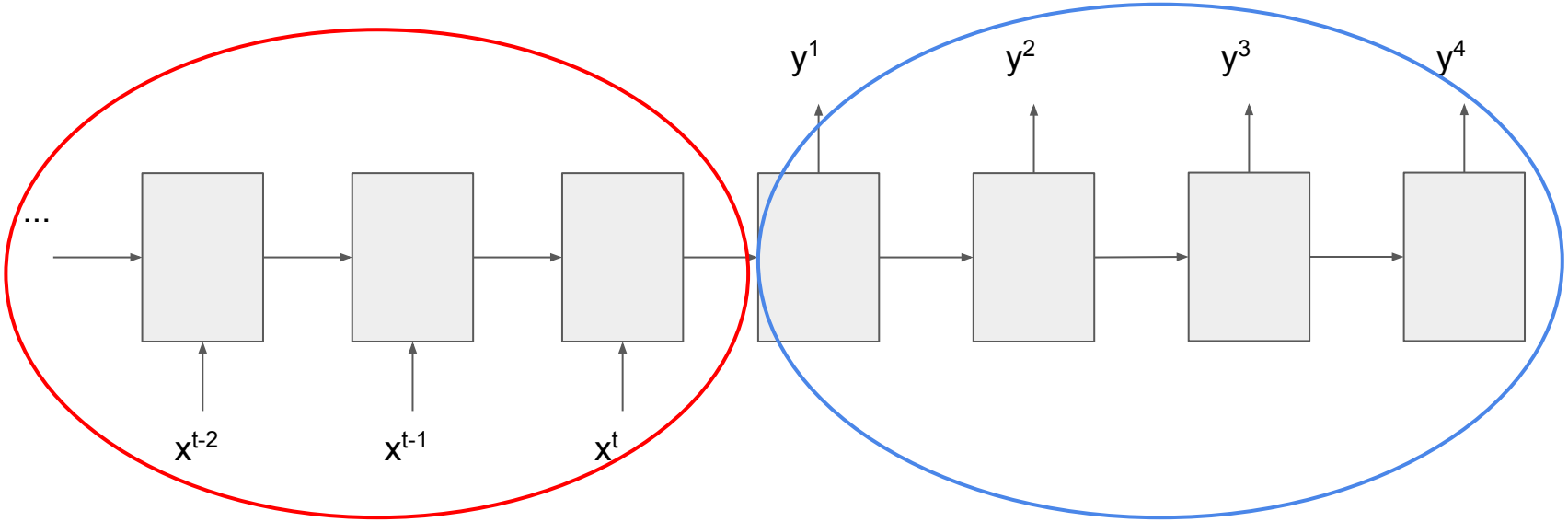


Some typical examples for many-to-many architectures include:

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

For many of these problems input sequence length ≠ output sequence length. How to deal with that? Padding, restricting sequence length, and see next.

RNN architectures: many-to-many (encoder-decoder)



- 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)

LSTM and GRU

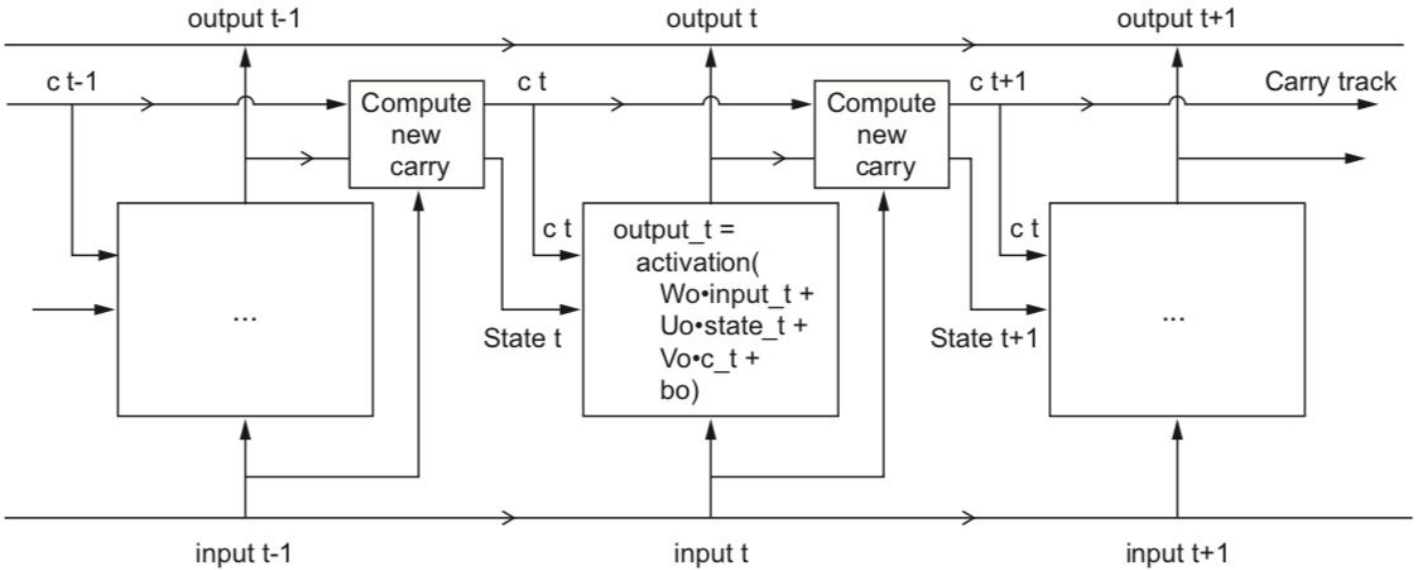
RNN units

SimpleRNN unit is not used in practice:

- Suffers from vanishing and exploding gradients phenomena (much like deep convolutional networks), see for example: https://medium.com/learn-love-ai/the-curious-case-of-the-vanishing-exploding-gradient-bf58ec6822eb
- Does not maintain information over longer sequences well

Units with explicit memory (or carry), a direct data link 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, for a more complete explanation see http://colah.github.io/posts/2015-08-Understanding-LSTMs/

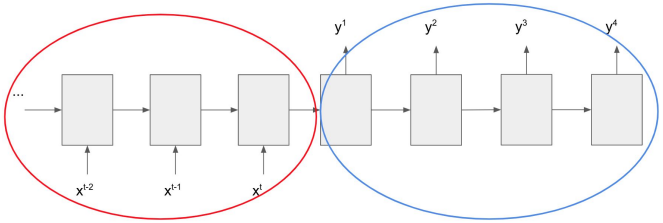
GRU (Gated Recurrent Unit)

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

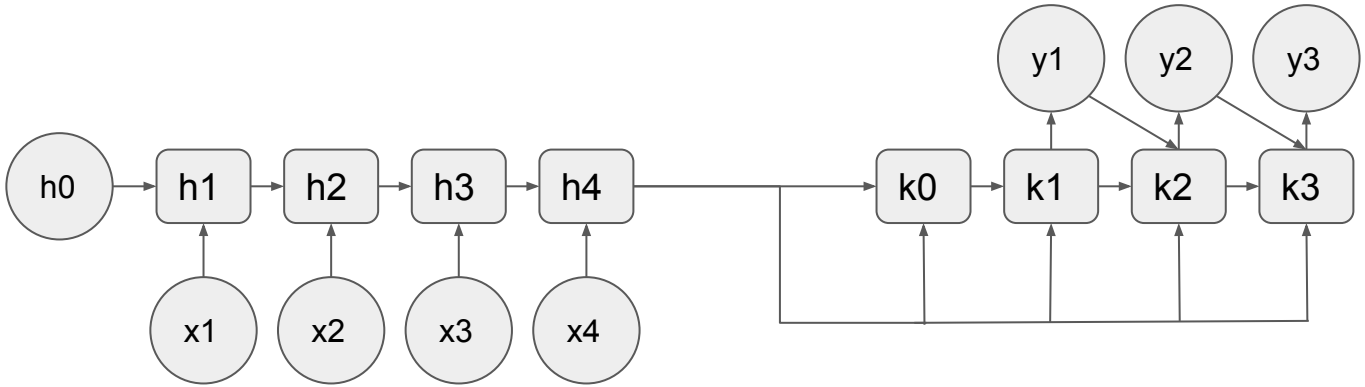
In practice less complex computationally than LSTM.

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

Language translation



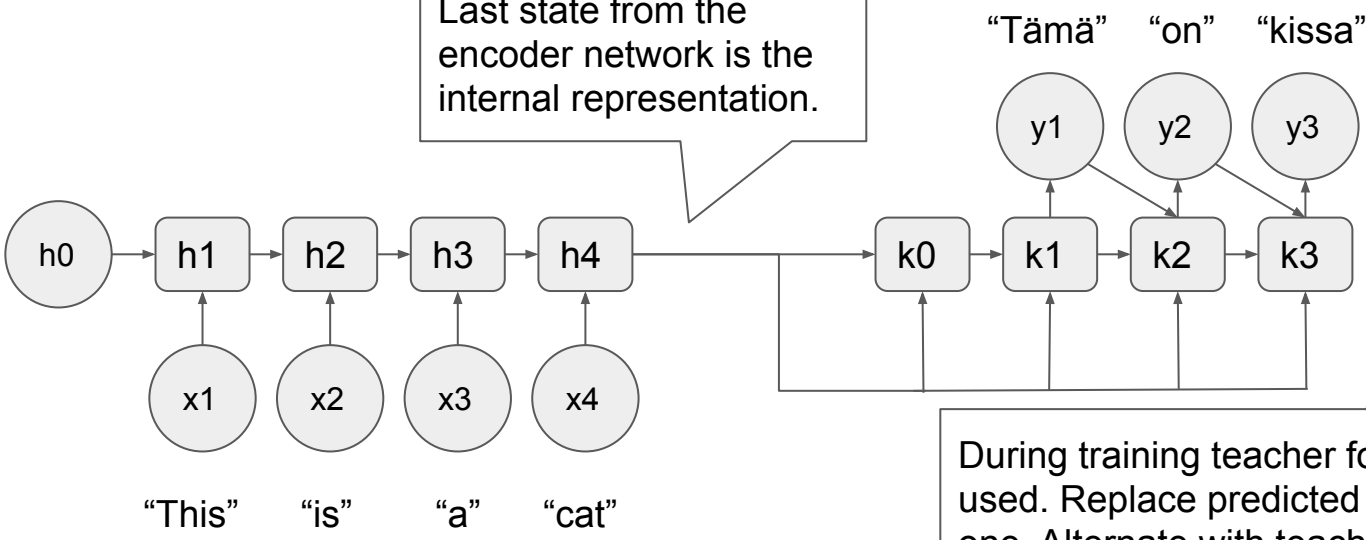
Translation could be implemented with an encoder-decoder network - encoder creates an internal representation of the source sentence, and decoder turns the representation into sentence in target language.



Translation with encoder-decoder

Predict probability distribution of words from each step. But: target is generate the most probable sentence, not just most probable words. *Beam search* algorithm is one possibility.

Last state from the encoder network is the internal representation.

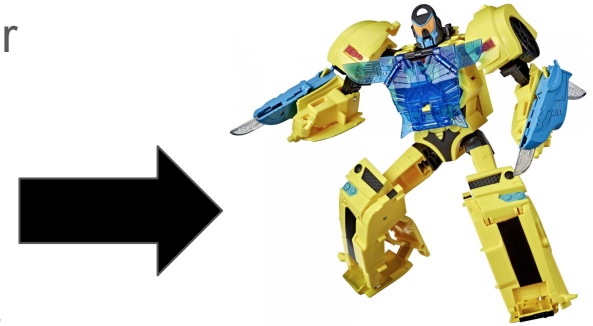


During training teacher forcing is often used. Replace predicted y(n) with correct one. Alternate with teacher forcing on and off during training.

From sequence-to-sequence model to transformer architecture

Sequence-to-sequence translation has shortcomings:

- Maybe very complicated sources (sentencies / paragraphs / documents) can't be held in encoder state vector (to create an effective intermediate representation)
- The length of context in a RNN (even if it is an LSTM) is limited - when processing a large document the context learned in the beginning of the sequence doesn't necessarily progress to the end
- Only a single embedding is learned for each word → a word has only one "meaning"



Transformer architecture

Attention is all you need

When the “prominent features” are selected in a concolutional net (by using maxpool -operations), the convnet learns to ignore non-essential parts of input by concentrating only on features of highest importance scores.

This kind of mechanism can be used for making the word embeddings discussed earlier context-specific: we’d like the word “see” in sentences “See you soon.”, “I’ll see that the bug gets fixed.” and “I see what you mean.” have different embeddings since they appear in different conteces.

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

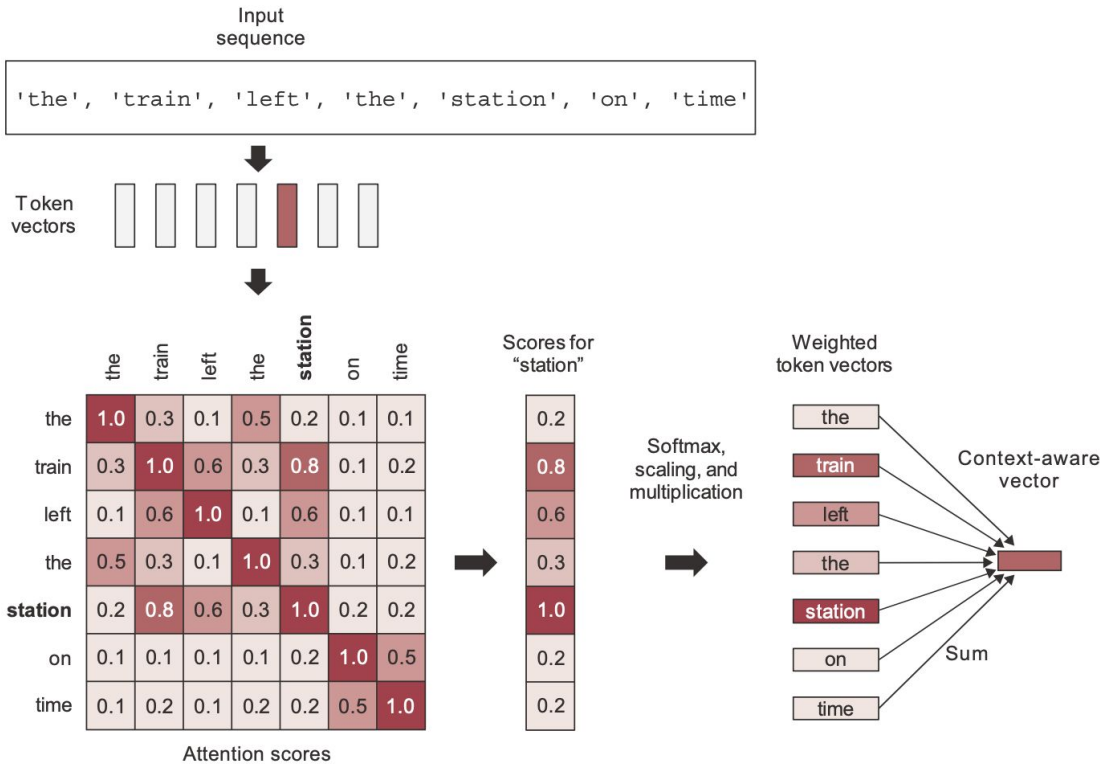
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

https://arxiv.org/pdf/1706.03762.pdf

Context-aware attention example



Example from Deep Learning With Python 2nd Edition by Chollet

Find context-aware vector for "station": attention score computed by taking the inner product, or dot product, of each pair of vectors representing words in the sentence is calculated.

For "station" attention score of "train" is highest (ignoring "station") → it gets highest weight in the context-aware vector .

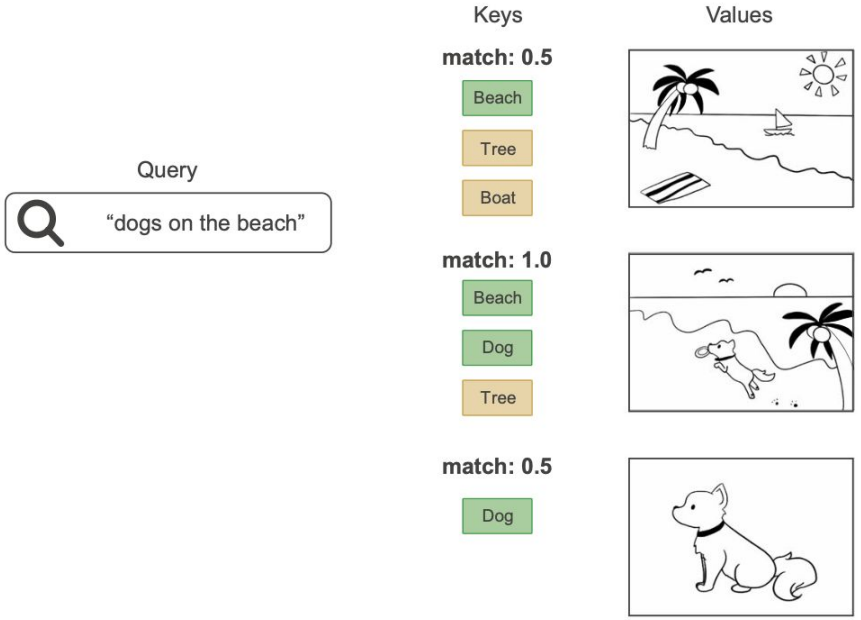
The resulting context-aware vector is the one that is used as the word vector in this context ie. it replaces the original static embedding.

Query - Keys - Values

The way context-aware embedding was computed in previous slide can be generalised into Query-Keys-Values schema.

- Query: sequence that describes what is being looked for
- Value: the information that is looked for
- Key: description of value in a format similar to query

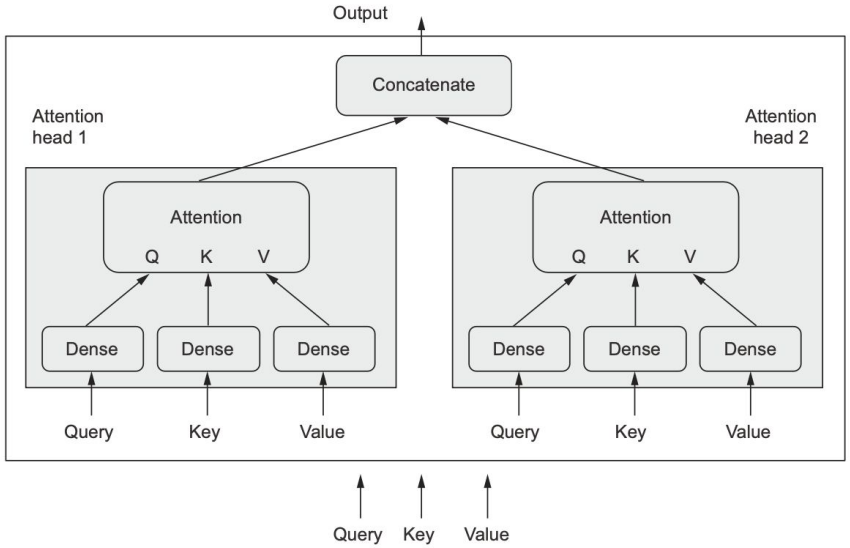
In translation, for example, keys and values are the same sequence.



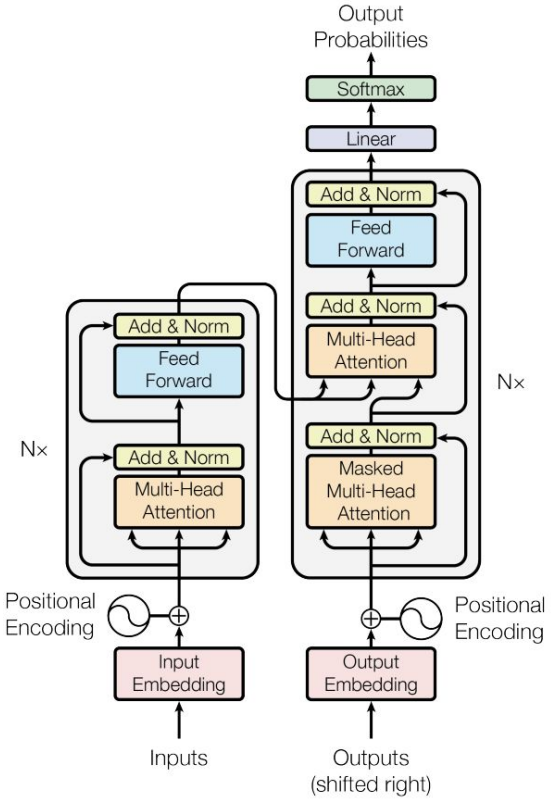
Multi-head attention

Instead of learning only one attention vector, why not learn multiple ones (to reflect different conteces). This is not entirely different from using multiple filters in a convolutional layer.

To do this, introduce multiple separately learned attentions (attention heads), and combine them in the output.

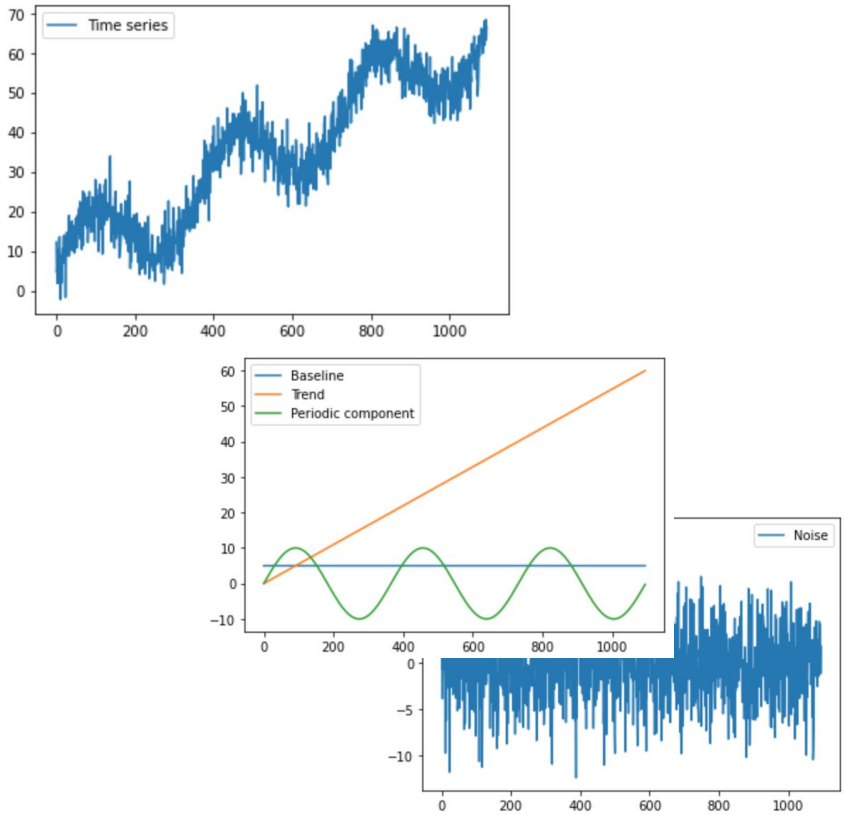


Transformer architecture



- Key parts of the architecture are the multi-head attention modules
- RNN (GRU/LSTM) is not used at all - it is replaced with dense networks only (Feed Forward).
- Residual connections (Add & Norm) are used for the same reason as in resnet architecture - rectify vanishing gradients problem.
- But but but! In dense networks there is no ordering. Add to context-aware embedding vector positional encoding.
- In decoder part multi-head attention is masked - attention can only depend on previous positions, not ones after the current position, in training

Time series analysis

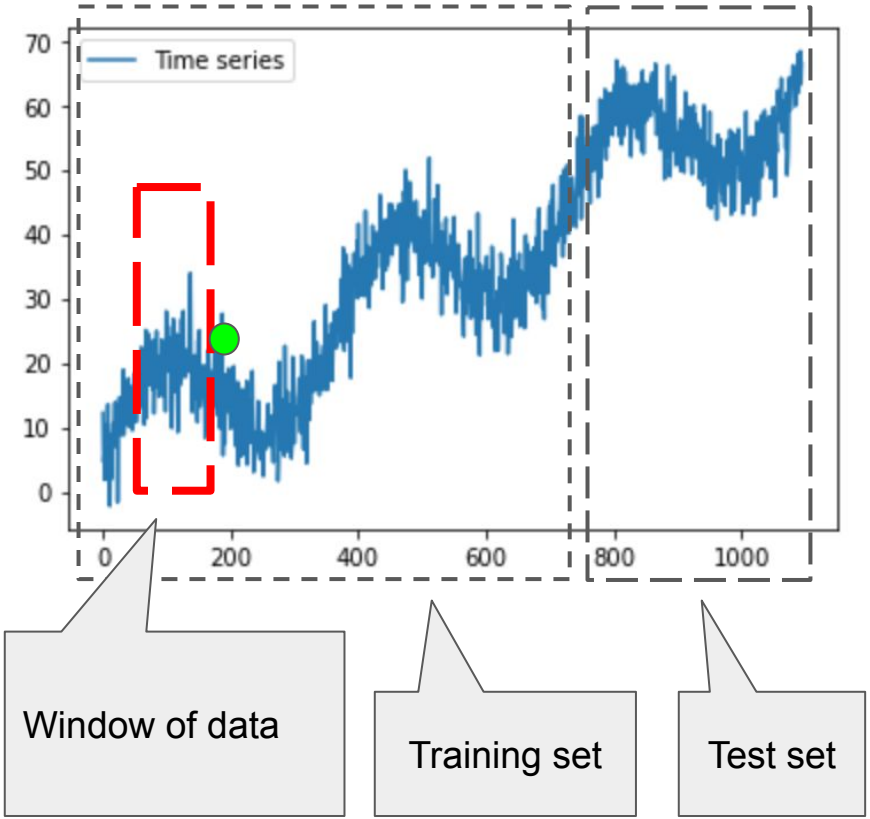


ANNs can be applied to making predictions based on existing time series data. By its nature time series data is strongly correlated - next value typically depends on previous values over a time slice.

Time series can often be divided into

- Baseline
- Trend component
- Periodic/seasonal component
- Noise

Training and test sets and windows



Using training samples, learn to predict next value (green dot) based on values in red box.

Sample for the model:

([data in window], value after the window)

Predict from window values

One can do averages, moving averages, etc. - ANNs can also be used for the prediction task:

- Dense network - use the values in window as input and predict next value,
- Recurrent network - since time series data is by nature sequential, use recurrent layer, for example LSTM, for making predictions,
- 1D convolution - convolution in a single dimension, and
- Combination of all of these!

1D convolution

Like 2D convolution but in one dimension only:

1D convolution could be used for processing voice signal, but it can be useful with any sequential data.

Example

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Conv1D(filters=32, kernel_size=5,
                         strides=1, padding="causal",
                         activation="relu",
                         input_shape=[None, 1]),
  tf.keras.layers.LSTM(32, return_sequences=True),
  tf.keras.layers.LSTM(32, return_sequences=True),
  tf.keras.layers.Dense(1),
  tf.keras.layers.Lambda(lambda x: x * 100)
])
```

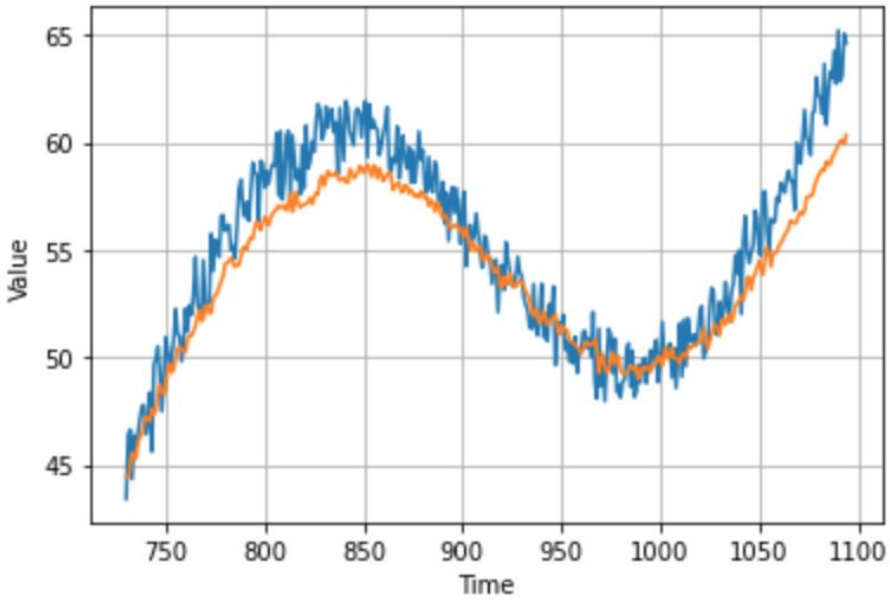
Here the model is specified in a bit different way compared to standard Sequential we used before. No change in functionality.

Lambda layer: an arbitrary function can be applied to data (here to the output of the last dense layer).

Layer (type)	Output Shape	Param #
conv1d_6 (Conv1D)	(None, None, 32)	192
lstm_12 (LSTM)	(None, None, 32)	8320
lstm_13 (LSTM)	(None, None, 32)	8320
dense_1617 (Dense)	(None, None, 1)	33
lambda_5 (Lambda)	(None, None, 1)	0

Total params: 16,865
Trainable params: 16,865
Non-trainable params: 0

Example



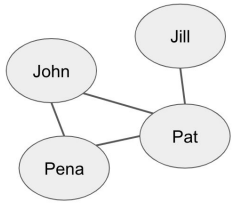
Window size is 20 timesteps: all predictions (orange line) are made based on previous 20 values.

For more on time series prediction, see https://www.tensorflow.org/tutorials/structured_data/time_series

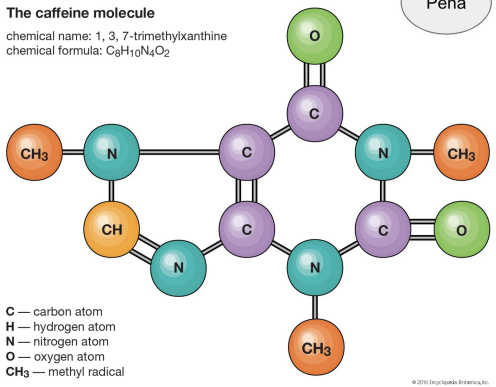
More ANN types: Graph neural nets

Data on which predictions are made has graph structure

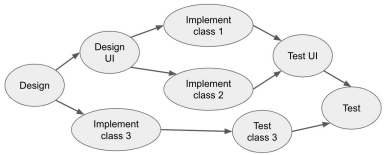
- Social network



- Molecule



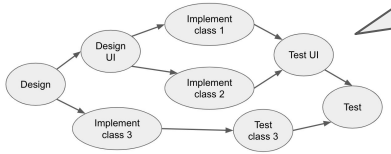
- Project task graph



From Encyclopedia Britannica

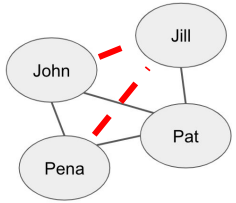
Some graph machine learning tasks

- Node classification
  - Predict a property of a node based on graph structure

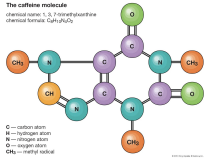


Duration of task taking into account the predicted duration of other relevant nodes in the graph

- Link prediction
  - Predict if there are links missing in the graph

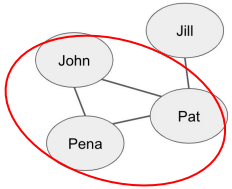


- Graph classification
  - Molecule property prediction

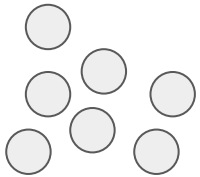


poisonous

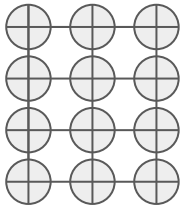
- Clustering
  - Detect social circle / clique



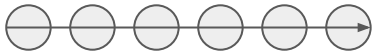
Different type of data → different ANNs



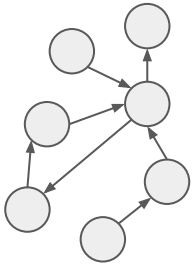
- Dense net
- Unstructured data



- Convnet
- 2D data



- RNN
- 1D sequential data



- Graph neural nets
- Graph-structured data
- Semi-supervised learning - some nodes are labeled, some not → predict labels for unlabeled, based on graph structure

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)

For text-related tasks, see NLTK (Natural Language Toolkit) at https://www.nltk.org/ - stemming, lemmatisation, stopwords elimination and much much more.