

#I would like to have a grade

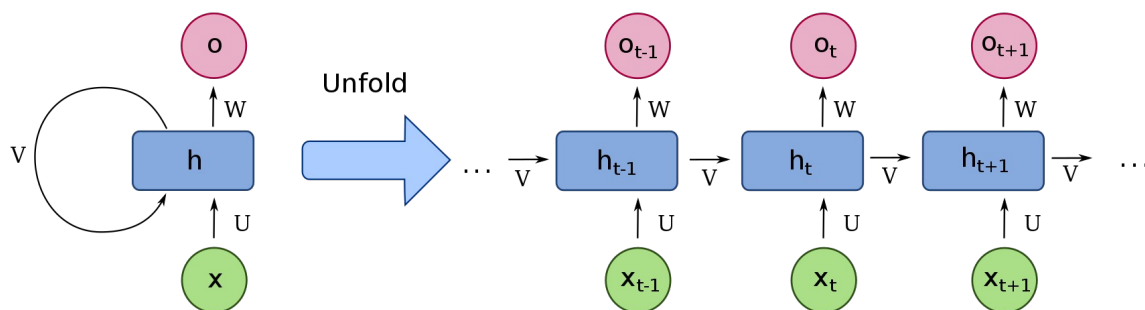
Report on what I wanted to do as a project for the course „Python für Fortgeschrittene – Wissenschaftliche Anwendungen“ and what I learned on the way by failing to produce a working self made solution.

My first Idea was to delve further into the topic of neural networks and machine learning after introducing the first basic concepts in the lecture.

The self set aim was to consider data in an ordered sequence like text music statistics data in time to make predictions for a given startinput about the further development of that sequence based on the training data. So the idea of a NLP (Natural Language Processing) related work was set. As text as well as stocks seemed a common idea and on first sight not that funny, I wanted to try a simple music generation.

### **RNN:**

Therefore I first started to understand what RNN's (recurrent neural networks) are. It is possible the simplest way to keep track on previous seen data in an ongoing sequence. The idea is that every cell in addition to a standard cell has a hidden state containing information on the past inputs. The generated output is then computed not only by the given input but also with the help of the hidden state to take the previous data into account. While the output is processed also a new hidden state has to be computed to keep track of the past input.



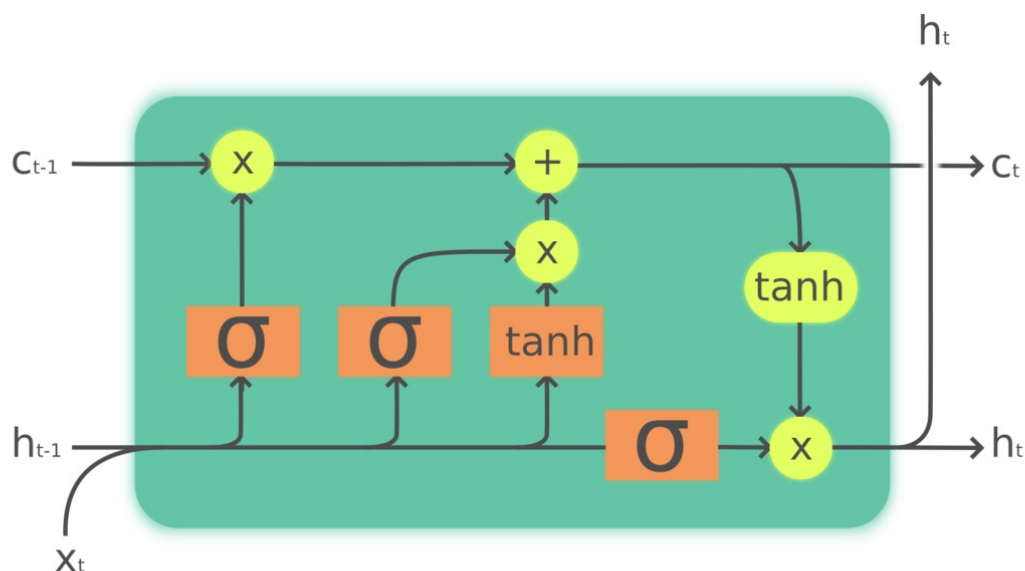
The picture shows a cell of a RNN and its unfolding where  $x$  is the input  $o$  the output and  $V$  the update of the hidden state after each step.

To train such a network normal backpropagation wouldn't work anymore but one has to do a backpropagation through time since also all the hidden states have to be updated. This results in a lot of matrix multiplications which lead to heavy problems with vanishing gradients.

This is one of the major drawback of RNN's and crystallizes to the effect that it has a rather short memory of previous structure and dependencies. (Exploding gradients are a problem to but can relatively easy be solved by normalisation between the computations also called gradient clipping) There are a few concepts for RNN's to attack these problem keeping the style of the RNN structure but I didn't read further into it since the next step in the evolution of sequential neural networks tackle exactly this problem.

### **LSTM**

So next on I studied LSTM neural networks (long short time memory), rely on a lot more computation by introducing methods to decide on the relevance of historical information deletes irrelevant information and also decides about importance of input information.



Legend:

Layer



Pointwise op



Copy



C: cell state; h: hiddenstate; X: input

Going from left to right first unnecessary data of the previous state is forgotten (forget gate) then relevant information of the new input is stored, after that the status is updated and last outputted.

The big advantage in comparison to RNN's is that we have an uninterrupted gradientflow along the cell states which means we mitigate the Problem of vanishing gradient while training the model using gradient descent. (notice that all the operation in updating the cellstate C are pointwise operations)

A very similar approach as LSTM are GRU (Gated Recurrent Units) but I have no deeper knowledge about them

LSTM's were used by company's like google or Facebook for translation tasks a few years ago. (most likely the where replaced by better performing networks)

### Transformer:

One probably the biggest problem all the mentioned networks share is that they have to be computed step by step due to their serial structure. This means that they cant make use of parallelisation as offered by gpu's and multicore cpu's (which nowadays is the most common way to reduce computation time).

This Problem was addressed in a paper of 2017 namely "Attention is all you need" (<https://arxiv.org/abs/1706.03762>), where a new model of neural network is introduced, called the transformer.

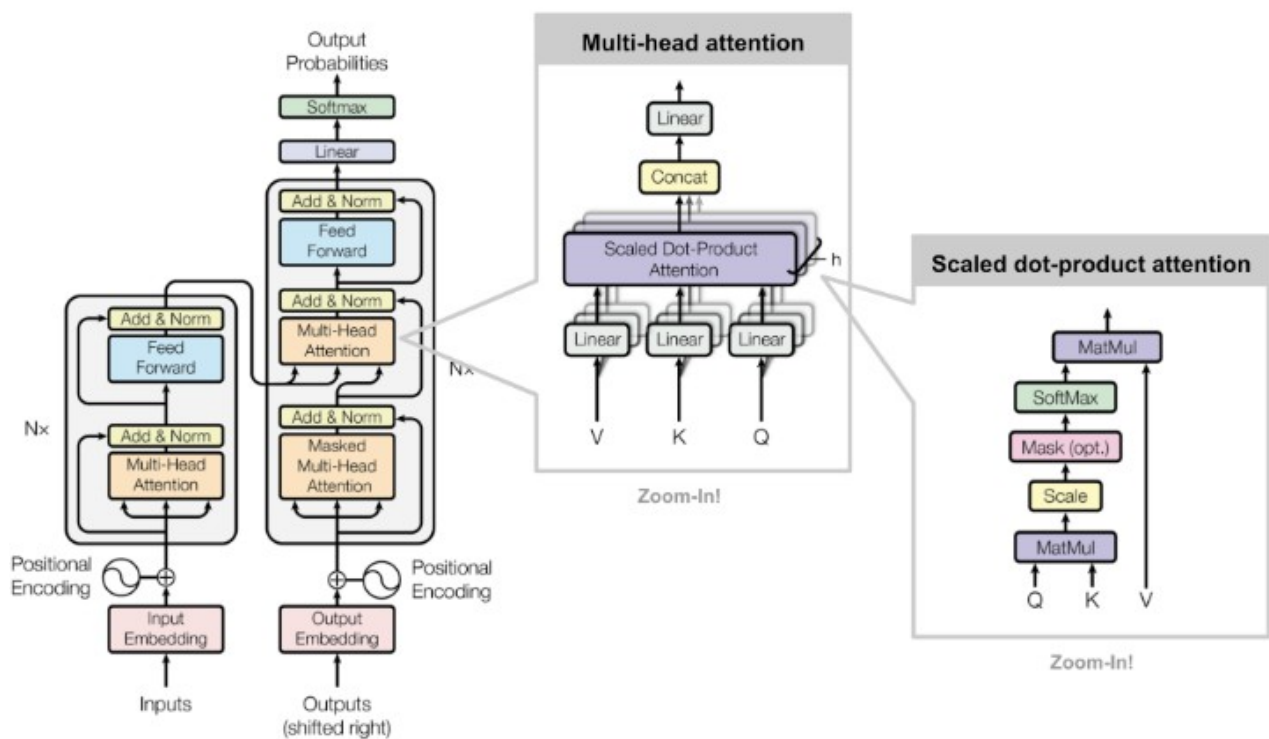
This model uses attention maps to determine the relevance of information via weights. Whats new to the already known attention mechanism, which computes attention between input sequence and

output sequence, is that they proposed self attention which is used to determine the relevance of information between different position of the input sequence.

As the title of the paper suggests, transformer utilizes self attention without using any RNN' related structure.

A lot of time (~12h) of the original 24h went into understanding this paper by reading it, watching explanation videos and by experimenting with the tensorflow tutorial implementation example of transformer. ([HTTP://www.tensorflow.org/tutorials/text/transformer](http://www.tensorflow.org/tutorials/text/transformer)) (this is quite nice)

The basic layout is the following:



Source: Attention is all you need

**Scaled dot-product attention:** An attention mechanism where query and key values are multiplied then scaled to minimize problems with small gradients (due to large numbers after huge matrices multiplied). The mask is used to hide information aside which lies in the future (sequence positions still to come). After that a activation function is applied and after that a dot product with the Value matrix.

**Multihead attention:** Values, keys, and query's are split into linear projection then attention is applied and afterwards concatenated instead of applying the attention to the whole matrices, at the end everything is finally projected to obtain the output. Splitting into subspaces has the advantage that every attention head "see's" another type of information and therefore can recognize different patterns.

**Positional encoding:** This is important since we need the information where in the sequence we are, without this the model would just see a bag of words since it has no information about position. This is done by using sin and cos on the even/odd positions and with different frequencies to encode the relative position.

Encoder (left):

Takes a input runs the positional encoding and applies selfattention to the input add the result to the input (residual connection) and applies a layer normalisation. (Think of batch normalisation/ There is a whole paper on layer normalisation (used where batch normalisation wont work)) Afterwards a classical feedforward network is applied and again added and normalized.

Decoder (right): From bottom to top First the output shifted right to get the next step in the sequence is positionally encoded. Then as in the encoder selfattention is applied and the result added and normalized. This result as the query is fed into a multihead (not self attention but “classic” attention) where Values and keys are obtained from the encoder. Again this is added and normalized and finally processed through a feed forward network, added and normalized.

To finish after encoder and decoder the result is projected and runs through an activation function, in the paper a softmax was chosen, to obtain a probability distribution of the next token in the sequence.

The tensorflow tutorial for transformer straight forward implements this picture, and it works quite nice. As one might assume even for small text length it already takes quite a lot of time to train such a model, but the positive effect is that one has to train the model only once and can use it afterwards. Also one can notice that one can checkpoint the training so if no changes were made to the code till training the network the model can still be used or loaded.

After spending so much time understanding the model, I decided to get started with the idea to use the transformer model as proposed in “attention is all you need” and try to build a simple music generator out of it. There would have been further paper on this topic and also a bunch of elaborated examples with this more elaborated models but I recognized that I don't have that much spare time and I wanted to come up with a solution by my own so I dint wanted to read the projects code.

### **About music generation**

Music is like language a sequence of notes but unlike words the are not that easy to tokenize since a single note has more information than a word. A word can be easily tokenized via a dict but a single note has the information about pitch, start time, endtime, velocity if we consider music in the form of a midi file (which we will for simplicity other formats can be converted to midi using some kind of fourier transformation algorithms)

So there are several ways to do this for suggestions see magenta or (<https://towardsdatascience.com/practical-tips-for-training-a-music-model-755c62560ec2>)

This has to be done for every instrument witch adds complexity and computational time therefore I considered only pieces with piano.

Note that we also didn't talk about chords witch can/should be handled separately for beautiful results.

Then one has to convert these tokenized songs into bunched up training data.

It should be noted that every song should be in the training data several times with different pitch offset since for us to melodies in different octaves sound the same melody wise but are not the same for a neural network.

Creating a useful dataset to train the the transfomernetwork to predict next token/note in a melody for a given start sequence is my main problem which a wasn't able to fix yet also I'm already 35+ hours in due to my overintrest of more knowledge of neural networks and overconfidence to get the project done straight from theoretical knowledge.

The aim is to get the project done somewhere in the future but I cant keep infesting all spare time into this. Nevertheless it was an intensive time working on this the last two weeks and in my feeling I now have a better understanding concerning recent neural networks.

Since there is quite a bit of time between the test runs I dug a little bit deeper into state of the art NLP neural networks and found that a lot of recent papers where published to improve the transformer model significantly and to utilize and alter it for specific tasks.

Notable: Transformer XL ; GPT2 by openAi (state of the art translation network)

In the git repository there is also a link of interesting articles I read and found useful.