Introduction to Transformers

D. Gueorguiev 6/24/22

Table of Contents

[Preliminaries 1](#_Toc107550344)

[Language Modeling 1](#_Toc107550345)

[Sequential Modeling 1](#_Toc107550346)

[Notes on RNN (Recurrent Neural Networks) 2](#_Toc107550347)

[Notes on LSTM (Long Short Term Memory) 6](#_Toc107550348)

[Notes on Gated Recurrent Neural Nets 6](#_Toc107550349)

[Latent Semantic Analysis (LSA) 6](#_Toc107550350)

[Encoder-Decoder Architectures 7](#_Toc107550351)

[The Attention Mechanism 7](#_Toc107550352)

[Bibliography 7](#_Toc107550353)

# Preliminaries

*Note 0*: Knowledge of feed-forward neural networks and the back-propagation algorithm is assumed throughout this document.

It is important to understand how the *Transformer architecture* fits historically as a modeling tool. For the purpose first we will look into other architectures and models which evolved earlier compared to the *Attention mechanism* and the *Transformer architecture*.

## Language Modeling

Tools for Language Modeling:

* Sequential Models
* Probabilistic Models e.g. *Latent Semantic Analysis* (LSA)

Besides used in Language Modeling the *Sequential models* are widely used in the modeling and prediction of Time-Series, Images, and Voice. Thus, we will start with a discussion on various architectures for Sequential Models which precede the *Transformer architecture*.

## Sequential Modeling

In general, we build ML models where we have data points which are usually uncorrelated, and no order relation can be imposed on them.

*Sequential models*: In many cases such as language, voice, and time-series data a data point is dependent on a set of other data points which have already been processed. We call such stream of data points a stream of *sequential data*. Machine learning models which accept input or create an output sequence of data points are known as *sequential models*.

Previous tools for sequence modeling:

* Recurrent Neural Networks
* Long Short Term Memory
* Gated Recurrent Neural Nets

### Notes on RNN (Recurrent Neural Networks)

What is RNN?

RNN is a Neural network with at least one cycle. If the network contains cycle the computation is not uniquely defined by the interconnection pattern and the *temporal dimension* must be considered. When the output of one neural node is fed back to the same node we are dealing with recursive computation. We must define what we expect from the network – is the fixed point of the recursive evaluation the desired result or one of the intermediate computations? We can assume that every computation takes a fixed amount of time and can be expressed as a certain number of time units. If the inputs of a neural node have been sent at time , its output will be available at time . A recursive computation can be stopped after predetermined number of steps and the last output will be considered the result of the recursive computation.

Figure 1: Node with a loop

Continuous vs Epochwise mode

Two approaches to operate and train an RNN:

* *Epochwise operation*

The network is run from a given start time until a given stopping time is reached. After reaching the stopping time the network is reset in its *initial state* for the next epoch. It is not essential that all of *the state* at the beginning of each epoch is the same. The important aspect of the *Epochwise operation* is the starting state of a new Epoch is not causally related to the ending state of the previous epoch.

Thus, every epoch serves as a boundary through which learning credit cannot pass. The purpose of the epoch boundaries is to make sure that activity from one epoch is causally related to activity in another later epoch. Note that the notion of epoch is defined in a loose sense indicating only that the boundaries are present between an interval(s) and an interval of past activity is separated by boundary from the activities after the boundary.

This allows us to introduce the notion of *batch training* distinguished from the notion of *incremental training*. The difference between the batch training vs incremental training is in when the network weights are updated. With the batch training approach weights are updated only after presenting a complete set of training examples. With the incremental approach the weights are updated after presenting each training example.

* *Continuous operation*

No manual state resets nor any barriers against the flow of training credit are imposed in the network.

Continuous operation makes sense when online learning is required.

Backpropagation through time (BPTT)

This paragraph follows the notation introduced by [Rojas](https://github.com/dimitarpg13/transformers_intro/blob/main/articles_and_books/neuron_Rojas.pdf) and borrows some of the diagrams introduced in his work.

For simplicity, let us consider a finite number of iterations only. Assume that a network of computing nodes is fully connected and that is the weight associated with the edge from node to node . We can unfold the network at times , , , transforming the original RNN into a feed-forward network with stages of computation. At each discrete time an external input is fed into the network and the outputs of all computing nodes are recorded. We will denote vector of all network outputs at time with . We assume that i.e. all network outputs at the initial moment are zeros. The unfolded network is depicted below:

. . .

. . .

. . .

. . .

. . .

. . .

. . .

. . .

. . .

. . .

. . .

Figure 2: Backpropagation through time

This unfolding strategy which converts the RNN into a feed-forward network in order to apply the classic back-propagation algorithm is called *Back-Propagation Through Time* (BPTT).

Let us denote by the matrix with the network weights . Let us denote by the matrix of interconnections between input sites and units. The feed-forward step is computed in the usual manner with a feed-forward network. At time we feed the transformed network with the dimensional external input . At each discrete time there are given the network state (-dimensional row vector) and the vector of derivatives of the activation function at each node:

(1)

In (1) we have:

(2)

where is the input to the activation function of the -th node at time .

Recall, we need the derivative of the activation function at each node in order to compute the back-propagated error when we are doing the back-propagation step. Refer to the figures below:

*- activation function*

*+*

backpropagation

Figure 3: Result of the backpropagation step

. . .

. . .

hidden node

input site

Backpropagated error to the -th hidden node

backpropagated

error

backpropagation to input site

Figure 4: Backpropagation from extended output layer through a hidden node to input site

The error of the network depicted on Figure 2 can be measured after each discrete time moment if a sequence of values is to be produced or just after the final moment of time if only the final output is of importance. The error vector between the -dimensional target and the output of the network is given with:

(3)

which is -dimensional column vector.

As with feed forward networks we define the network error function to be:

(4)

. . .

. . .

. . .

. . .

output units

-th hidden

unit

Thing to consider: each weight of the network is present at each stage of the unfolded network (Figure 2).

**Theorem**: Any network with repeated weights can be transformed into a network with unique weights.

Let us consider an unfolded feed-forward network with structure as the one shown in Figure 5.

Weight exists in multiple different stages of the unfolded feed-forward network and it receives different input in each stage. In the depicted on Figure 5 stages the inputs are and accordingly.

\*

\*

backprop step

forward step

Figure 5: A duplicated weight in a network Figure 6: Transformation guaranteeing unique weights

backpropagation

Figure 7: Multiplication as integration function

The network on Figure 5 can be transformed as shown on Figure 6. The transformed network is indistinguishable from the original network with duplicated weight from the viewpoint of the result it computes. Note that the two edges associated with weight previously now have weight 1 and a multiplication is performed by two additional units in the middle of the edges. In the transformed network w appears only once and we can perform backpropagation as usual. There are two groups of paths – one coming from the first multiplier (depicted with \*) to and the ones coming from the second. This means we can perform backpropagation as usual in the original network. Let us find the partial derivative of the error function with the respect to the weight . At the first edge we obtain , at the second and since is the same variable as the desired partial derivative is:

(5)

The partial derivative can be expressed in terms of the backpropagated error and the output feed into the current node from the previous stage ([Rojas, Chapter 7](https://github.com/dimitarpg13/transformers_intro/blob/main/articles_and_books/neuron_Rojas.pdf)). Thus, we have:

and (6)

Here and denote the backpropagation errors for layer and accordingly.

Note that the relation between the backprop error of a node in layer and the backprop error of the same node in layer can be inferred from the general recurrence relation for the backprop errors in adjacent layers ([Rojas, Chapter 7](https://github.com/dimitarpg13/transformers_intro/blob/main/articles_and_books/neuron_Rojas.pdf)):

(7)

The partial derivates for the weight coefficients corresponding to layer are given with:

(8)

In case the layer is the output layer we have:

(9)

The partial derivatives for the weights on the output layer are given with:

(10)

Thus, extending (5) for the general case we can write the relation for the partial derivative of the network error function with respect to the network weights :

(11)

In (9) the quantity denotes the -th target value.

The correction of the weights in case of duplicate weight shared between the same node which appears in adjacent layers will be performed as:

(12) where

(13)

Here represents the learning rate.

Let us formally describe the BPTT using matrix notation

The backpropagated error at time is given with :

(14)

Here is a diagonal matrix in which the main diagonal is occupied by the elements of given with (1). In general, for any moment , we can write:

The error vector at moment is given with (3).

Let us denote with the matrix of weights , where is the weight associated with the edge from node to node . With we denote the matrix of interconnections between input sites and units. With we denote the -dimensional external input vector.

//TODO

### Notes on LSTM (Long Short Term Memory)

In the BPTT step we calculate the partial derivates at each weight of the network. The RNNs are deep networks where the partial derivatives are formed as series of products as we already know. These series of products can bring the overall value of a partial derivative in an early moment of time to negligibly small value which will be useless to correct error in the weights. This leads us to the *Vanishing Gradient Problem* with RNNs.

#### Vanishing Gradient Problem with RNNs

The usual choice in RNN for activation function is the sigmoid activation function:

The derivative of the sigmoid function is given with:

Obviously is used in the product series of the backpropagated error in the BPTT step. Refer to (7) and (9) for details. Succinct analysis on the problem is presented in (Arbel, 2018) (link [here](https://medium.datadriveninvestor.com/how-do-lstm-networks-solve-the-problem-of-vanishing-gradients-a6784971a577)). Indeed it all boils down to how we calculate the partial derivative of the network error with respect to the network weights as a sum of the partial derivatives with respect the network weights for each moment of time (refer to (11)). Because each term in the right-hand side of (11) is computed with the recurrence relation (7) we end up of series of products where each term looks like:

Those terms end up in the expression for the weight updates given with (13).

//TODO

### Notes on Gated Recurrent Neural Nets

//TODO

## Latent Semantic Analysis (LSA)

LSA belongs to the family of *Probabilistic Methods* for analyzing and modeling the structure of the language.

//TODO

# Encoder-Decoder Architectures

Became popular after 2014 , main article on encoder-decoder architectures is from [Ilya Sutskever et al](https://github.com/dimitarpg13/transformers_intro/blob/main/articles_and_books/SequencetoSequenceLearningwithNeuralNetworksSutsekver2014.pdf). Encoder-Decoder architectures were historically used as machine translation models in Natural Language Processing with encoder and decoder for each language or involve language specific encoder applied to each sentence whose outputs are then compared. An encoder neural network reads and encodes a source sentence in a fixed length vector. A decoder then outputs a translation of the encoded vector. The whole encoder-decoder system which consists of the encoder and the decoder for a language pair is jointly trained to maximize the probability of correct translation given a source sentence.

The Motivation for the Encoder-Decoder Architectures: A drawback of the classical Deep Learning architectures is that DNNs can be applied to problems whose inputs and targets can be sensibly encoded with vectors of fixed dimensionality. It is significant limitation as many important problems are best expressed with sequences whose lengths are not known a-priori. For example, speech recognition and machine translation are sequential problems.

//TODO

# The Attention Mechanism

This paragraph follows the discussion on the Attention Mechanism in [Galassi et al](https://github.com/dimitarpg13/transformers_intro/blob/main/articles_and_books/AttentionInNaturalLanguageProcessing.pdf).

In many NLP problems the components of the text source have different relevance for the task which is being performed.

*The Motivation for the Attention Mechanism*: A potential issue with the traditional neural network-based encoder-decoder approach is that a neural network needs to be able to compress all the necessary information of a source sentence into a fixed length vector.

Examples:

*Aspect Sentiment Analysis*:

Words such as “*good*” or “*bad*” could be more relevant to some aspects under consideration or less relevant to other aspects.

*Machine Translation*:

Some words in the source text could be irrelevant in the translation of the next word.

*Visual Question Answering Task*:

Background pixels could be irrelevant in answering a question regarding an object in the foreground but relevant to questions regarding the scene (Arbel, 2018)ry.

Effective solutions of such …

//TODO

# Bibliography

Arbel, N. (2018, December 21). *How LSTM networks solve the problem of vanishing gradients*. Retrieved from https://medium.datadriveninvestor.com/: https://medium.datadriveninvestor.com/how-do-lstm-networks-solve-the-problem-of-vanishing-gradients-a6784971a577

Galassi, A., Lipp, M., & Torroni, P. (2020). Attention in Natural Language Processing. *IEEE Transactions on Neural Networks and Learning Systems*.

Rojas, R. (1996). *Neural Networks Systematic Introduction.* Berlin: Springer Verlag.

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to Sequence Learning with Neural Networks. *arXiv:1409.3215 [cs.CL]*, 9.