Implementation of the Transformer Architecture for Machine Translation of Image Captions

Steffen Maass

Abstract

In this project, I implemented the network architecture *the transformer* that forgoes recurrence for a (self-) attention mechanism that enables faster training and more accurate machine translation of image captions. The architecture is described in detail and scores for comparison to recurrent neural networks are provided. It is demonstrated that the transformer architecture is competitive with similar RNN implementations with our BLEU score of 33.3 after only half an hour of training.

1 Introduction

Natural Language Processing (NLP) is an important area where machine learning techniques are used. Next to text classification, text summarization, and natural-language generation, machine translation (MT) is a topic of NLP that is already used in many applications such as *Google Translate* or in social networks like *Facebook*.

In this report, we implement a model proposed by [16] that notably discards the sequential nature of the data and instead introduces a concept of (self-) attention to better weigh contextual importance in text.

1.1 Related work

Machine translation attempts began with statistical models based on translation examples in the 1980s [8], but similarly to the field of computer vision, incorporation of neural networks drastically improved scores on test datasets. Neural machine translation (NMT) models will therefore be the focus of this section.

Classical NMT models are based on the problem formulation of conversion from an input sequence to an output sequence, termed seq2seq [8]. The recurrence relation implied by sequences of letters and words supports the use of recurrent neural networks (RNN) for NMT. RNNs are similar to feed-forward networks in structure, but hidden layers have "states" that feed forward in time, as depicted in figure 1. In 2014, Cho et al. proposed the basic encoder-decoder structure [3] which has been widely adopted. The architecture uses an encoder to understand the input language and convert the text to a latent representation, and a decoder generates text from the latent representation to the output language. The encoder-decoder representation mirrors one understanding of human translation as a two-step process: humans extract the underlying meaning of a sentence, then construct a sentence in the target language from the meaning.

To prevent problems in training basic RNNs such as vanishing and exploding gradients, Cho et al. [3] used long short-term memory (LSTM) units to compose their encoder and decoder networks. LSTM units have gates which are able to logically "forget" past inputs, reducing the cumulative impact on gradients backpropagated through sequences. Luong et al. demonstrate a further improvement to the LSTM NMT model by including a term for attention [11]. Conceptually, attention is a mechanism of estimating relative importance of each input in the context of the entire input vector, information which might be lost in the serial structure of a standard RNN or LSTM.

Preprint.

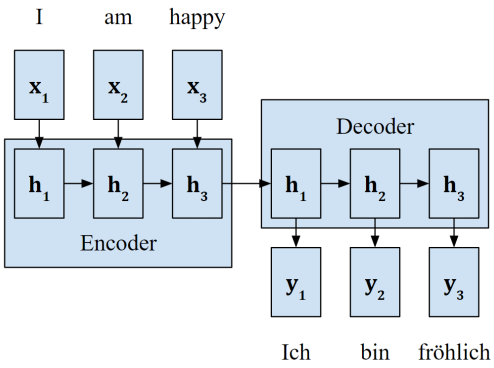

Figure 1: Seq2seq problem formulation with encoder-decoder model.

1.2 Transformer

In 2017, "Attention is all you need" [16] described a new model for NLP termed the "transformer". The transformer model uses a similar encoder-decoder structure that [3] proposed, but the layers only use the concept of (self-) attention to process inputs.

Vaswani et al. extend the concept of attention from Luong's model [11], removing the recurrence in the network entirely. One of the key insights from their implementation of the transformer is that it can effectively substitute positional encoding of words within a sentence as an additional input to the network [16] in place of a recurrent structure. By forgoing the recurrence, the network does not need to process the data in series and can use a parallel structure for the encoder instead. The self-attention structure also enables the network to learn a more complex understanding of the input data beyond a left-to-right sequence. The architecture has been shown [6],[2] to regularly outperform the more classical methods using recurrent neural networks, long-short term memory [9] and gated recurrent [4] on different NLP tasks.

1.3 Dataset

In this report, we implement the attention-based transformer architecture described by Vaswani et al. [16] to Multi30K, a dataset of English-German caption translations [7]. We demonstrate that this architecture is well-suited to this task, achieving high accuracy in testing in comparison to other models on the same dataset, especially models that include a recurrence relationship.

2 Methods

2.1 Problem statement

The problem of NMT is formatted as generating one word at a time in sequence in the output language. Equation (1) provides the seq2seq model, where at each index n in the output sentence it is desirable to produce a distribution $p(\hat{y}_n|\hat{y}_{i<n}, x)$ over the vocabulary of possible words.

$$p(\hat{y}_n|\hat{y}_{i<n}, x) = g(x, \hat{y}_1, \hat{y}_2, ..., \hat{y}_{n-1}) \tag{1}$$

The maximizing position of the distribution can then be selected as the most likely next word conditioned on the input sentence x and the previously predicted words $\hat{y}_1, \hat{y}_2, ..., \hat{y}_{n-1}$.

Encoder-decoder RNNs such as the architecture proposed by Cho et al. solve this problem by computing a hidden state h_n as they iterate sequentially over the input data. The encoder formulation is described by Eq. (2), adapted from Garg and Agarwal [8].

$$h_n = f(h_{n-1}, x_n) \tag{2}$$

The decoder hidden state equation described by Eq. (3) from Garg and Agarwal [8] includes an optional "summary" term c; in the model proposed by Luong et al., this summary is given by attention

[11].

$$h_n = f(h_{n-1}, \hat{y}_{n-1}, c) \tag{3}$$

Finally, the hidden state, previous predicted word, and summary term can be mapped to a conditional distribution of the output word by Eq. (4), adapted from Garg and Agarwal [8].

$$p(\hat{y}_n|\hat{y}_{i<n}, x) = g(h_n, \hat{y}_{n-1}, c) \tag{4}$$

The transformer model is a modification of this problem statement that removes the need to compute hidden states h_n. The transformer encoder computes the summary term, c as

$$c = f(x) \tag{5}$$

and the decoder produces a target representation $p(\hat{y}_n|\hat{y}_{i<n}, x)$ by

$$p(\hat{y}_n|\hat{y}_{i<n}, x) = g(c, \hat{y}_{n-1}, \hat{y}_{n-2}, ..., \hat{y}_1) \tag{6}$$

In the following sections, we describe the components that comprise the encoder f and decoder g.

2.2 Data Pre-Processing

A pre-processing step is needed to transform the *original text* (in the following also called *input text*) into the input format of the model, a one-dimensional array of integers. One unit of input text is called *input sentence* in the following. Note, however, that in natural languages there are inter-dependencies between sentences in a text. A full paragraph might also be chosen as one unit of input text. Therefore, the choice of the input unit is a design choice of the model that has implications on the computational complexity of the model [1].

Data cleaning and standardization. In a first step each character of the step is converted to lowercase and special characters are removed. In the following all characters are assumed to be from the Latin alphabet (with the addition of the characters `ä, ö, ü` which are frequently used in German words), digits or punctuation characters.

Tokenization. In a second step the text is separated into *tokens*. A token is a lexical unit with a meaning attached to it. This might be a word (e.g. "report"), a subword (e.g. "pre" as in "*pre*trained"), or a punctuation (e.g. "!"). [16] proposed to use the byte-pair encoding, a compression algorithm adapted to word segmentation, that chooses subwords as tokens according to compression principles given a target number of distinct tokens (see [13] for more details). The set of all computed tokens is called the *vocabulary* and its cardinality is the *vocabulary size*. The target vocabulary size was chosen to be $8192 = 2^{13}$ in the experiments. The vocabulary is enumerated so each token is assumed to be represented by an integer n: $1 \leq n <$ vocabulary size, which is also called its ID. The token ID 0 is reserved for *padding* which will be explained below. In the experiments the punctuation characters were first separated from the other characters. Then the TensorFlow-implementation [2] of the byte-pair encoding was used.

Add beginning of sentence (bos) and end of sentence (eos) token. Finally a bos and eos token has to be added at the beginning and end of the input sentence respectively.

After the data pre-processing an input sentence is an one-dimensional array of integers where the integer n at index i represents the i-th token in the input sentence.

2.3 Transformer architecture

An overview of the Transformer architecture is illustrated in Figure 2. It consist of two major components (both in grey boxes): the encoder (left) and the decoder (right). The input of the encoder is the full input sentence represented by an one-dimensional array of integers. The input of the decoder is also an one-dimensional array representing the tokens translated thus far. The decoder input will also be called *translated sub-sentence*. From a high-level perspective, the encoder learns a hidden representation also called *latent* representation of the input sentence using self-attention. The

[1] The number of model parameters increases quadratically with the input length measured in tokens (see below).

[2] `tensorflow_datasets.deprecated.text.SubwordTextEncoder`

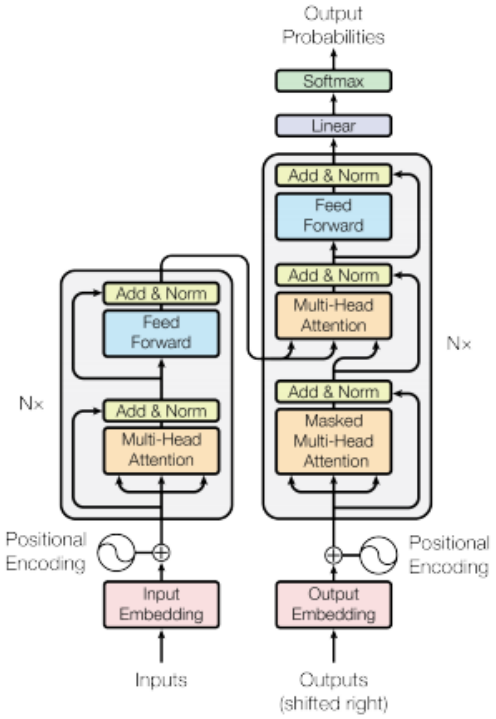

Figure 2: Transformer architecture with an encoder (left) and decoder (right). Image taken from [16]

decoder also learns a latent representation of the translated sub-sentence and combines it together with the latent representation of the input sentence to predict the *next token*.

In the following the important sub-parts of the layer are explained together with additional information on their implementation.

Embedding and Positional Encoding. In a first step both for the encoder and decoder, each token is embedded into a latent space of dimension d_{model} using a linear projection from a space of dimension equal to the vocabulary size where each token is represented by a unit vector, the one-hot encoding of a token. Next to each embedded vector a vector determined by the position of the token in the (sub)-sentence is added. The reason – as mentioned in [16] – is that the encoder and decoder are invariant under permutation of the embedded tokens. In order to take the relative position of tokens into account when finding a good latent representation of the (sub-)sentence, additional information has to be added. [16] proposed to add the positional vector (illustrated in Figure 5), $\mathbf{v_p}$, to the token at position p in the (sub)-sentence:

$$\mathbf{v_p} = \left[\sin\left(\frac{p}{1000^{0/d_{model}}}\right), \cos\left(\frac{p}{1000^{0/d_{model}}}\right), \sin\left(\frac{p}{1000^{2/d_{model}}}\right), \ldots, \cos\left(\frac{p}{1000}\right)\right]^T. \quad (7)$$

Since there exists a linear transformation A_k such that $A_k \mathbf{v_p} = \mathbf{v_{p+k}}$ independent of p, [16] claim that the encoder/decoder can learn the relative position of tokens.

Multi-Head Attention. The latent representations of the tokens is learned using the attention mechanism contained in the Multi-Head Attention layer together with the feed forward network with one hidden layer of dimension d_{ff}. The Multi-Head Attention takes the current latent representation, Z, of the (sub-)sentence and projects it onto the three inputs called the *value* V, *key* K and *query* Q:

$$V = ZA_i^V, \quad K = ZA_i^K, \quad Q = ZA_i^Q \qquad \text{with} \qquad A_i^V, A_i^K, A_i^Q \in \mathbb{R}^{d_k \times d_{model}} \quad (8)$$

where i is the index of the *head*, $i = 1, 2, \ldots, h$ and $d_k = d_{model}/h$. Each head uses attention scores to obtain an update, V_i, of the projected latent representation in parallel. The updates are then concatenated and projected back into the latent space of dimension d_{model}:

$$Z = \left[V_1^T, V_2^T, \ldots, V_h^T\right] A^O \qquad \text{with} \qquad A^O \in \mathbb{R}^{d_{model} \times d_{model}}. \quad (9)$$

Note that the query in the second decoder layer is the projected latent representation of the translated subsentence and the value and key are the project latent representation of the input sentence. The attention weights are calculated using the softmax function and multiplied with the value:

$$A_i^{\text{weights}} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right), \qquad V_i = A_i^{\text{weights}} V. \tag{10}$$

Thus, the projected latent representation of each token (a column in V) gets scaled according to the attention weights.

Encoder Layer. The encoder consists of N layers. Each layer has two sub-layers: a multi-head attention layer and a feed forward network. The output of each sub-layer is an updated d_{model}-dimensional latent representation of the input sentence. After each layer the updated latent representation is 'averaged' with the previous latent representation using a residual connection followed by a layer normalization (see [1]). We used the TensorFlow-implementations for the feed forward network[3] and the layer normalization[4] but implemented the multi-head layer by ourselves.

Decoder Layer. The decoder consists of N layers too. Each layer has three sub-layers: two multi-head attention layers and one feed forward network. The output of each sub-layer is an updated d_{model}-dimensional latent representation of the translated subsentence plus the next token. At the beginning, the latent representation of the next token is set to be unknown (e.g. token ID 0). Similarly to the encoder, the latent representation is updated after each layer by 'averaging' with the previous latent representation using a residual connection followed by layer normalization. There are two important differences in the use of the multi-head attention:

1. The first multi-head attention again uses self-attention but to prevent that the next token attends to subsequent tokens a masked is used (see Figure 6).
2. The second multi-head attention uses the queries from the previous decoder layer and the values and keys from the encoder. As [16] note, this allows that the latent representation of the translated subsentence plus the next token can be learned by attending on all tokens in the input sentence. This is in contrast to conventional, recurrent approaches as noted before.

Transpose Embedding and Softmax. The final layer in the Transformer architecture after the decoder is a transpose embedding from the d_{model}-dimensional latent space into the space of dimension equal to the vocabulary size. [16] proposed to tie the parameters to the parameters from the decoder embedding which we also implemented manually. Finally, a softmax layer is used to obtain a probability mass function over all possible tokens in the vocabulary of the target language.

2.4 Training

The training set consists of two observations: (1) sentences $\{x^1, \ldots, x^K\}$ in the source language where each sentence $x = (x_1, \ldots, x_S)$ consists of multiple tokens. Here x_n denotes the token at position n. (2) sentences $\{y^1, \ldots, y^S\}$ in the target language, the ground truth.

Teacher Forcing. As explained in the previous section, the model is *auto-regressive*. That is, it takes the previously translated subsentence $(\hat{y}_{n-1}, \ldots, \hat{y}_1)$ as its input into the decoder to predict $\hat{y}_n$. The training uses a method that is called the *teacher forcing* algorithm. Instead of iteratively calculating the probability mass function

$$p = p(\hat{y}_n | \hat{y}_{i<n}, x) = g_\theta(x, \hat{y}_{i<n}) \tag{11}$$

and optimizing the loss $L(\theta; y_n, p)$ with respect to the model parameters θ, the probability mass functions

$$p_n = p(\hat{y}_n | y_{i<n}, x) = g_\theta(x, y_{i<n}) \qquad n = 1, 2, \ldots, L \tag{12}$$

are calculated simultaneously using the ground truth as the decoder input. Then the averaged loss function

$$\frac{1}{S} \sum_{n=1}^{S} L(\theta; y_n, p_n) \tag{13}$$

[3] `tensorflow.keras.layers.Dense`

[4] `tensorflow.keras.layers.LayerNormalization`

is optimized with respect to θ. In the experiments the **cross-entropy loss** was chosen.

Batching and Padding. The training data was batched into a *batch size* of 64. Since each sentence in a batch has a different length, *padding* had to be used to obtain contiguous batches. That is, to obtain sentences of the same length in a batch, padding tokens with token ID 0 are added at the end of the sentences. The padded tokens had to be masked out so that actual tokens cannot attend on padded tokens (compare to Figure 6). The mask was created during the embeddings and had to be propagated to the multi-head attention layers. Due to broadcasting in tensor operations using TensorFlow, the code scales to batched training sentences.

Optimizer. The loss function is optimized using the Adam optimizer introduced in [10]. We used the TensorFlow-implementation[5] but implemented the custom learning schedule

$$LearningRate(step) = d_{model}^{-1.5} \ \min\left(WarmUpSteps^{-0.5} \cdot step, \ \ step^{-0.5}\right) \tag{14}$$

and a custom training loop.

Regularization. Dropout layers as introduced in [15] were used for regularization. Dropout layers were used after the embeddings and between each sub-layer and layer normalization.

3 Experiment & Results

The model was implemented and trained in Google Colaboratory [6] on a freely available GPU.

Dataset. In our experiments we used the database Multi30K [7] that was used during the First Conference on Machine Translation in 2016 (WMT 2016). The dataset consist of 29000 training instances, 1047 test instances and 1000 instances in the validation set. Each training instance consist of a triple: English source sentence, German translation, and an image. In our experiments we disregarded the additional information from the images.

Hyper-parameters. The most important hyper-parameters of the model are the dimension of the latent space, d_{model}, the dimension of the hidden layer, d_{ff}, in the feed forward networks, the number of heads, h, in the multi-head attention layers, and the number of encoder and decoder layers N. Manual hyper-parameter tuning was performed using the validation set. It was found that choosing $d_{model} = 128$ (instead of 512 as used in the paper), $d_{ff} = 512$ (instead of 2024), $h = 2$ (instead of 8), and $N = 2$ (instead of 6) yielded good and fast results on the validation set. Note that in their experiments, [16] trained their model on a dataset contain several million training instances. Therefore, the hyper-parameters had to be chosen orders of magnitude smaller to adapt the model to the small dataset we used our experiments.

Early-Stopping. The model was trained for 28 epochs for about 30 minutes using checkpoints every 4 epochs. The training and validation loss are shown in Figure 3. Since the validation loss slowly increased after the 16 epoch due to possible overfitting, after training the model was also tested using the state after 16 epochs.

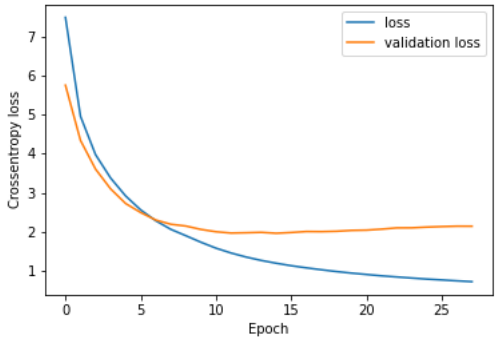

Figure 3: Training cross-entropy loss and validation cross-entropy loss per epoch.

[5] `tensorflow.keras.optimizers.Adam`

[6] `https://colab.research.google.com/`

System	BLEU	TER
Transformer (after 16 epochs)	33.3	49.0
Transformer (after 28 epochs)	33.0	48.9
LIUM_1_MosesNMTRnnLMSent2Vec_C	34.2	48.7
1_en-de-Moses_C	32.5	50.2
DCU_1_min-risk-baseline_C	31.8	52.6

Table 1: BLEU and TER values

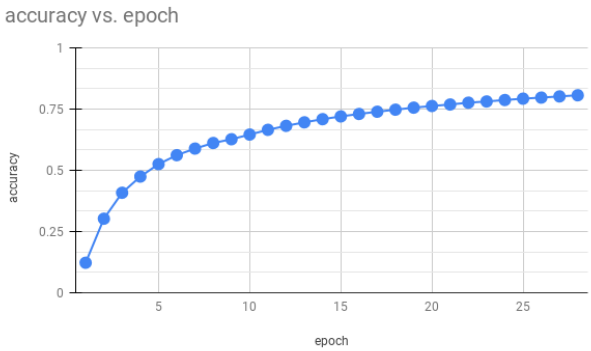

Figure 4: Training accuracy per epoch.

Testing. In the machine translation task of the WMT 2016, the systems were evaluated on the 1047 test instances using the BLEU [12] and TER scores. Both scores are common metrics to evaluate the quality of machine translations. Better translations have a higher BLEU score and lower TER score. The performance of the systems was scored using the `MultVal` scoring tool [5][7] in the WMT 2016 competition. The performance of our systems together using the `MultVal` scoring tool is shown together with some systems that were submitted to the competition in Table 1.

Comparison. The baseline and the best scoring system are from the survey of [14]. Both the BLEU and TER values of our implementation show significant improvement over 1_en-de-Moses_C, our baseline in this context.

4 Conclusion

In this paper we implemented the transformer architecture proposed by [16] and trained it on the Multi30K dataset of English-German captions.

The transformer model trained in only 30 minutes, much faster than recurrent networks on the same dataset. Further, we demonstrated competitive BLEU and TER scores in comparison to recurrent neural networks.

While the basic transformer architecture performed well on this dataset with short training time, more recent state-of-the-art approaches make improvements on the BLEU and TER scores on large-scale datasets. In 2020, OpenAI released a pretrained transformer model termed GPT-3. GPT-3 improves the BLEU score of the basic transformer model on large-scale datasets such as EUROPARL by several points [2].

Future work would include an in-depth comparison of performance between our basic transformer implementation and a complex, modern model such as GPT-3 to understand how different components contribute to improved performance.

[7] `https://github.com/jhclark/multeval`

References

[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[2] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language Models are Few-Shot Learners. *arXiv preprint arXiv:2005.14165*, 2020.

[3] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation, 2014.

[4] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[5] Jonathan Clark, Chris Dyer, Alon Lavie, and Noah Smith. Better Hypothesis Testing for Statistical Machine Translation: Controlling for Optimizer Instability. In *Proceedings of the Association for Computational Lingustics*, 2011.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-Training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[7] D. Elliott, S. Frank, K. Sima'an, and L. Specia. Multi30K: Multilingual English-German Image Descriptions. In *Proceedings of the 5th Workshop on Vision and Language*, pages 70–74, 2016.

[8] Ankush Garg and M. Agarwal. Machine translation: A literature review. *ArXiv*, abs/1901.01122, 2019.

[9] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural computation*, 9 (8):1735–1780, 1997.

[10] Diederik P Kingma and Jimmy Lei Ba. Adam: A Method for Stochastic Gradient Descent. In *International Conference on Learning Representations (ICLR)*, 2015.

[11] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation, 2015.

[12] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[13] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, 2016.

[14] Lucia Specia, Stella Frank, Khalil Sima'an, and Desmond Elliott. A shared task on multimodal machine translation and crosslingual image description. In *Proceedings of the First Conference on Machine Translation: Volume 2, Shared Task Papers*, pages 543–553, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/W16-2346. URL `https://www.aclweb.org/anthology/W16-2346`.

[15] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *The Journal of Machine Learning Research (JMLR)*, 15(1):1929–1958, 2014.

[16] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 5998–6008, 2017.

5 Appendix I: Input processing visualization

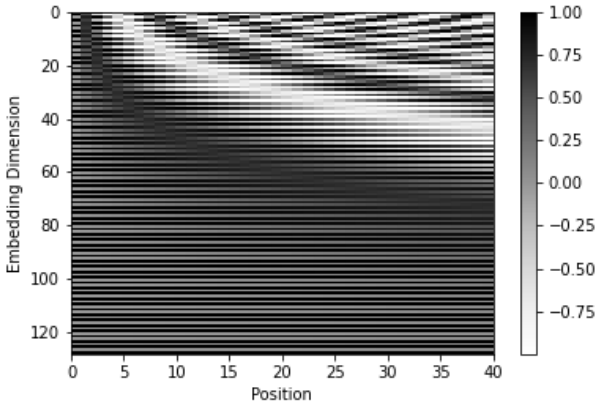

Figure 5: Positional Encoding: Each columns represents a positional vector.

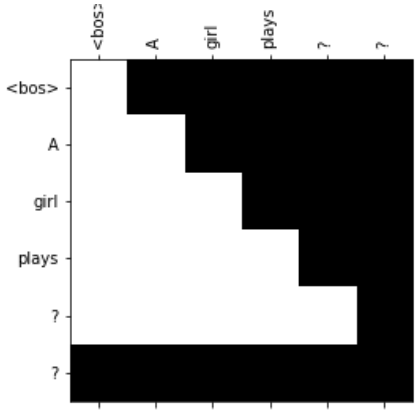

Figure 6: Masking: The token on the y-axis are not allowed to attend on subsequent tokens

6 Appendix II: Training visualization

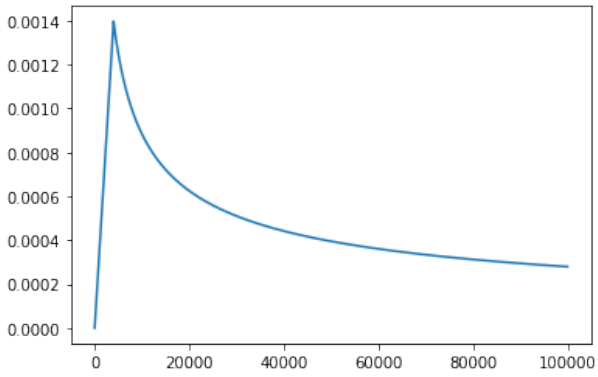

Figure 7: Learning shedule: Learning rate as a function of the step number.