UNIVERSITÀ DEGLI STUDI DI GENOVA

DIBRIS

DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLOGY,
BIOENGINEERING, ROBOTICS AND SYSTEM ENGINEERING

RESEARCH TRACK II

The Transformer model
Description of state-of-the-art machine learning model and its applications for Robotics

Author:

Giordano Emanuele

Student ID:

s4479444

Professors:

Carmine Tommaso Recchiuto

7

Contents

0 Abstract

Machine learning is a branch of Artificial Intelligence, whose purpose is to design algorithms or artificial systems capable of improving their performances on pattern recognition over a certain dataset.
ML has developed over the course of the XX century, starting from the first theories of Alan Turing or the first models of Frank Rosenblatt and Bernard Widrow, and eventually acquiring optimization techniques coming from statistics, as well as the mathematics of dynamical systems.
Today, Machine Learning and Deep Learning play a fundamental role in our society, handling data and recognizing patterns from all fields of human knowledge, such as Medicine, Finance & Economy, Physics and Engineering.

The following paper presents the assignment of Research Track II, which explains how the Transformer model [1] works, being one of the most performing models of the last years, in fields like Natural Language Processing or in general handling sequential data, but can be also employed in Robotics applications, aside with Reinforcement Learning techniques.

1 RNNs and their limits

1.1 Recurrent Neural Networks and sequential data

Sequential data is a particular type of data, arranged in sequences where order matters: data points are dependent on past data points, therefore violating the assumption of independently and identically distributed data samples. In a more formal way, sequential data can be described by assigning to each sequence in the dataset, to a matrix:

$$\mathbb{D}_{(n,m,d)} = \{X_1, X_2, ..., X_n\} \quad \text{where } X_i \in \mathbb{R}^{(m,d)}$$

being n is the number of sequences, m is the number of timesteps in each sequence and d is the number of features of each timestep.

In order to handle this kind of data, Recurrent Neural Networks have been introduced, being a family of learning machines with an internal state, or memory.

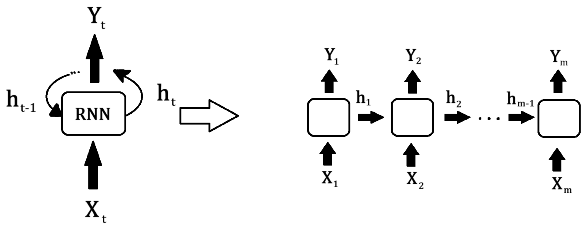


Figure 1: How RNNs work, wrapped version on the left, unwrapped version on the right

At each timestep, the network receives one input sample of d features X_t, as well as the internal (or hidden) state of the network itself h_{t-1}, coming from the previous timestep. Then it produces the corresponding output timestep Y_t, passing the current hidden state h_t to the next iteration until the sequence has terminated.

One of the most successful models regarding RNNs is called Long-Short Term Memory, where the network has two internal states and a system of interconnections and layers, that allow the model to learn causal patterns among timesteps.
In particular, at each timestep, the LSTM unit receives the two internal memory states from the previous itera-

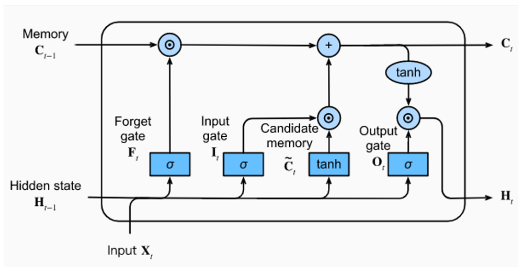


Figure 2: Architecture of a LSTM unit (credits to d2l.ai)

tion, as well as the input; then, through a series of activation functions and concatenations, an output timestep is produced alongside the hidden states for the next timestep.

1.2 The limits of RNNs

Despite their advantages, simple RNNs have been shown to have non-negligible disadvantages, especially for real-world scenarios, such as:

- **slow computation for long sequences**, as RNNs are built from a ”for” loop for each timestep to compute;
- **vanishing/exploding gradients**, resulting in failing to make the model learn;
- **difficulty in accessing distant past informations**, ending up the network to rely more on short-term patterns;

2 ”Attention is all you need”

In 2017, the paper ”Attention is all you need” [1] proposed a new machine learning model capable of handling sequential data without any kind of recurrent units - thus capable of faster computations.
Such model, namely the **Transformer**, is based on a serial architecture of encoder and decoder blocks, in which the so-called **attention mechanism** evaluates the correlations between different timesteps.

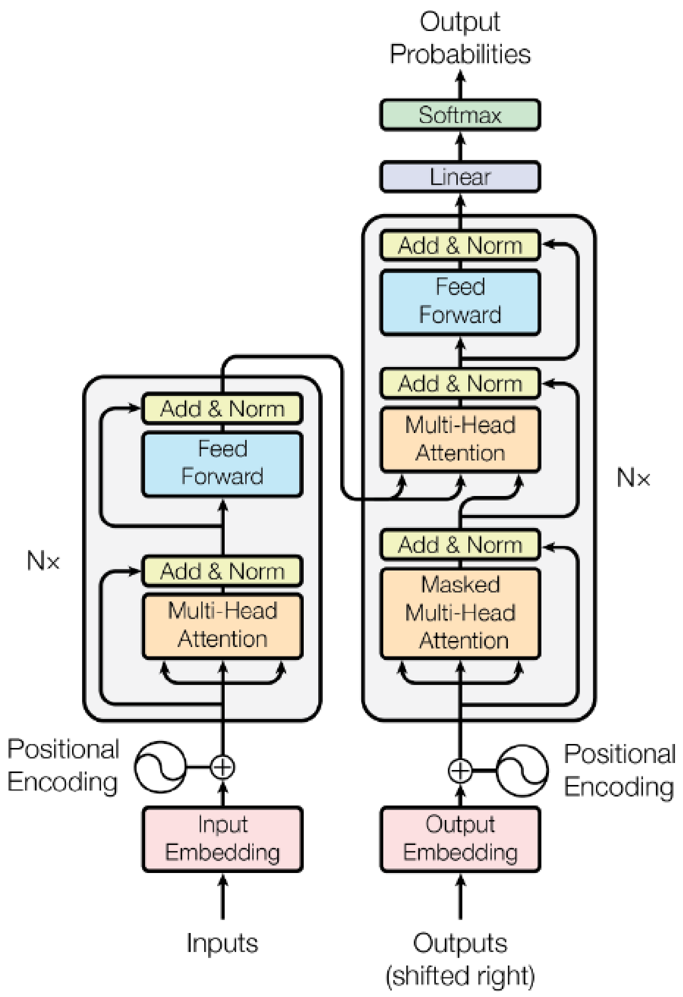


Figure 3: The Transformer model, credits to the authors of the paper [1]

As in Figure 3, the model can be divided into two main blocks: the **encoder block**, on the left, and the **decoder block**, on the right, each of the two parts is actually made up by a series of stacked encoder and decoder layers.

In particular, given a dataset made up by a n-tuple of training and target sequences, each of m timesteps and d features:

$$\mathbb{D}_{(n,m,d)} = \{(X_i, T_i)\}_{i=1,\dots,n} \text{ where } X_i, T_i \in \mathbb{R}^{(m,d)} \quad \forall i$$

for instance, being them a sentence in two different languages, and the task is to translate the input sequence into the target sequence's language; then, the input sequences are given to the encoder block, whereas the target sequences are given to the decoder block.
In doing so, the two sequences must pass through the **Embedding layer**, which tokenize the timesteps of the training sequences into a vocabulary of d_{model} dimensionality and let the model learn its own best embedding parameters.
Then, the embedded sequences are added to a **Positional Encoding** of the same sequences, in order to give the machine a sense of ”distance” between different timesteps, and at the same time giving this informations using a learnable pattern.

Finally, the two embedded sequences are passed into the encoder and decoder block: although they rely on the same **Attention mechanism**, the decoder block has two parts, the first being a masked attention layer (which simulates causality), and the second computing its output, not only using the previous layer's output, but also those of the encoder.
In the end, the output of the decoder is passed through a linear layer, and then a softmax layer, which predicts the probability distribution of the next timestep in the sequence.

2.1 The Attention mechanism

2.1.1 Self-attention

At the heart of the Transformer, lies the Attention mechanism, which is a manipulation of sequential data in such a way that similarity scores, for each possible timestep of the sequence, are computed.
The simplest attention mechanism implementation is Self-attention; in the encoder block, it can be noticed that the embedded input sequences are copied twice, each being a particular matrix in the attention layer.

In particular, given the input sequence $X_i \in \mathbb{R}^{(m,d)}$, the encoder block can be unwrapped as in Figure 4: at first, the sequence gets embedded into a vocabulary of a certain number of items: through the tokenization

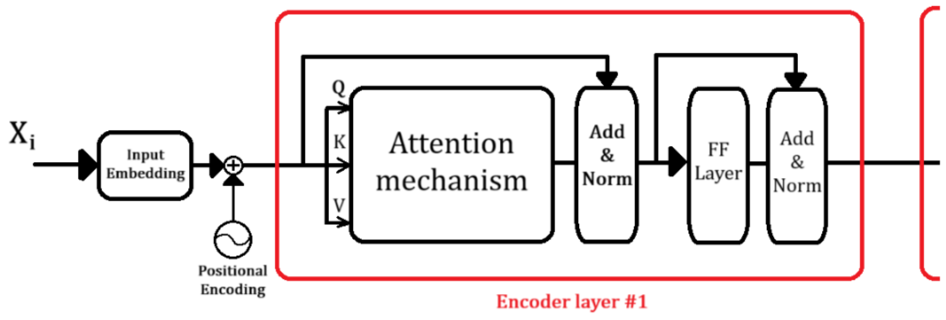


Figure 4: Encoder block unwrapped, focus on the first encoder layer

procedure, each timestep gets converted into a series of integers, corresponding to a position in the vocabulary. The size of the vocabulary is an hyperparameter, that depends on the task of the machine.
During embedding, the input sequences change their dimensionality:

$$X_i \in \mathbb{R}^{(m,d)} \longrightarrow X_{i,emb} \in \mathbb{R}^{(m,d_{model})}$$

where again, d_{model} is an hyperparameter of the model.
Then, after the sum of the Positional Embedding, the embedded input sequence is copied twice for the attention mechanism, and once more for the residual block at the end of it. In this copying procedure, the three inputs for the attention mechanism are called:

$$Q = X_{i,emb} \quad \textbf{queries}$$

$$K = X_{i,emb} \quad \textbf{keys}$$

$$V = X_{i,emb} \quad \textbf{values}$$

Under these premises, the Self-Attention mechanism formula is the following:

$$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_{model}}}\right)V \tag{1}$$

and by comparing the dimensionalities of the matrices, the output is in the same dimensions as the input:

$$(m, d_{model}) \times (d_{model}, m) \times (m, d_{model}) = (m, m) \times (m, d_{model}) = (m, d_{model})$$

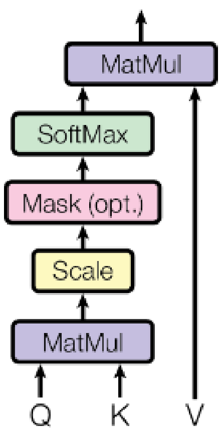


Figure 5: Self-Attention mechanism as a block scheme, credits to the authors of the paper

The process can be divided into three steps:

- multiply the queries with the keys transposed, ending up with a matrix of (m, m) dimensions;
- apply a normalization factor of $1/\sqrt{d_{model}}$ and then the softmax activation function, row-wise:

$$\sigma_i(\mathbf{z}) = \frac{e^{z_i}}{\sum_{j=1}^{m} e^{z_i}} \quad \forall i = 1, ..., m$$

  where **z** is a generic vector $\in \mathbb{R}^{(m,1)}$, in this case the rows of the matrix $QK^T/\sqrt{d_{model}}$. By doing this, the **attention scores** matrix is obtained, which quantifies the relationship between each timestep (rows) with any other timestep (columns), such that their row-wise sum is always one.
  In Figure 6, an example of attention scores visualized can show how the machine associates each timestep to all the other ones: in particular, the part of ”zone économique européenne” gets translated into ”European Economic Area”, but the two languages have completely different grammar rules; the transformer can see this pattern, and, as the image suggests, learns how to correctly translate the senteces.

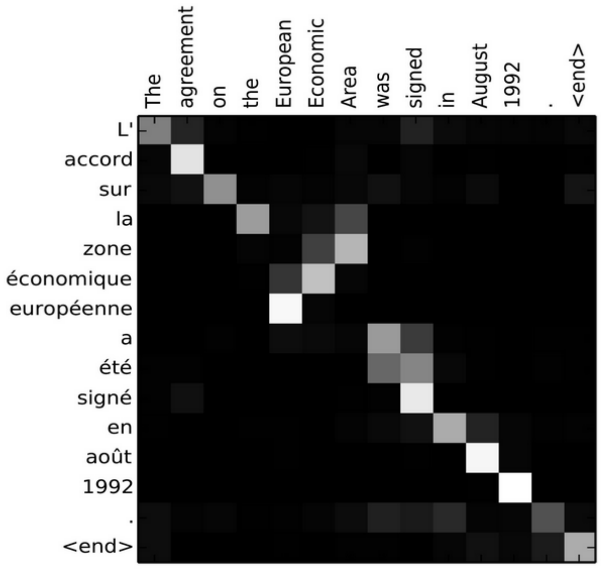


Figure 6: Attention scores visualization example for NLP applications, translation from French to English, credits to slds-lmu.github.io

- at the end, the attention scores are multiplied to the values matrix, obtaining therefore a matrix of dimensions (m, d_{model}), where each row captures, not only the meaning (given by the embedding) or the position in the sentence (positional encoding), but also each word’s interaction with other words.

Self-Attention has some interesting properties:

- **permutation invariance**: by changing the two (or more) rows of the input matrix, the output of the attention mechanism only swaps the corresponding rows, but its values are still the same;
- **requires no parameters**, as the interaction between timesteps has been driven by their embedding and positional encodings (but this will change later);
- the values of the principal diagonal of the attention scores matrix are expected to be the highest, as each timestep is strongly related to itself;
- **masking**: it's possible to mask future steps of each timetep, by multiplying the attention scores with a lower triangular matrix of ones;

2.1.2 Multi-Head Attention

In order to parallelize the process, and also grant the model the capability of ”specialization”, the Multi-Head Attention mechanism was introduced. In a very similar way to the simple Self-Attention, the embedded input sequence is copied twice into queries, keys and values, but now these matrices are first multiplied to three corresponding **weight matrices**:

$$Q = X_{i,emb} \longrightarrow Q' = QW^Q \quad \textbf{weighted queries}$$

$$K = X_{i,emb} \longrightarrow K' = KW^K \quad \textbf{weighted keys}$$

$$V = X_{i,emb} \longrightarrow V' = VW^V \quad \textbf{weighted values}$$

where the weight matrices are $W^Q, W^K, W^V \in \mathbb{R}^{(d_{model}, d_{model})}$.

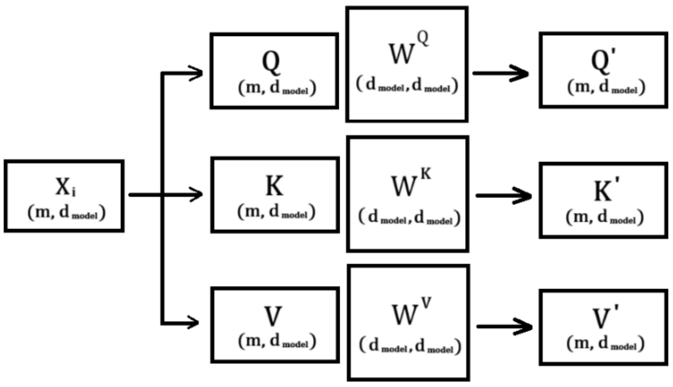


Figure 7: Weighted queries, keys and values

Afterward, the weighted queries, keys and values are column-wise split into h sub-matrices, alongside the d_{model} dimension. In this way, the attention mechanism will be performed onto portions of the sequence, of dimension: $d_k = d_{model}/h$. The Multi-Head Attention mechanism introduces the **attention heads**: for each of the h sub-matrices, the queries, keys and values are used to compute attention:

$$head_j(Q_j, K_j, V_j) = softmax\left(\frac{Q_j K_j^T}{\sqrt{d_k}}\right) V_j \quad \forall j = 1, ..., h \tag{2}$$

and the final result is obtained by concatenating all the heads' results, and post-multiply by a final weight matrix:

$$MultiHead(Q, K, V) = concat(head_1, head_2, ..., head_h)W^0 \tag{3}$$

where $W_0 \in \mathbb{R}^{(d_{model}, d_{model})}$.
A visualization of this process can be seen in Figure 7 and 8.

The result of this process is that each head sees the full sequence, but only a restriction of its embedded features, so that each head can map relationships between different features of each timestep.
This means that, although to-be-learned parameters are introduced, the process can be sped up and specialized, in the sense that each head will learn how to handle only a subset of input features, rather than the whole sequence.

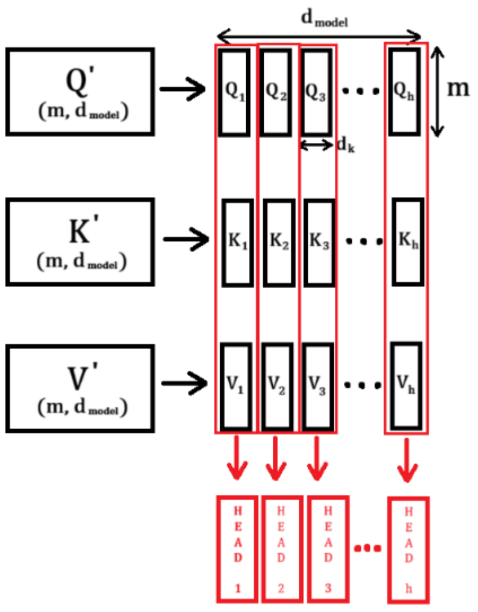


Figure 8: Multi-Head attention mechanism unwrapped, each head only sees a feature-wise portion of the sequence

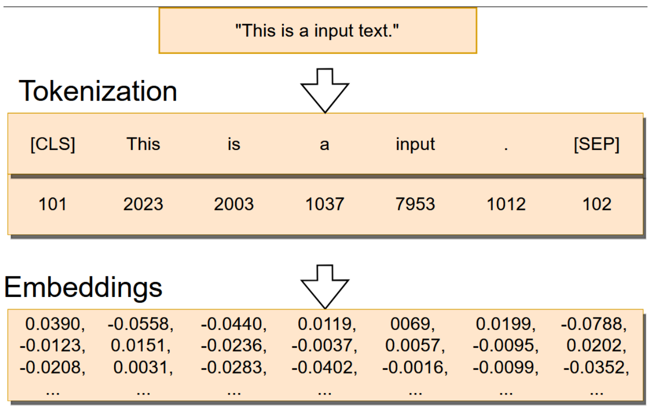
2.2 The Transformer Model

The entire model is now taken into consideration.
Given an input sequence $X \in \mathbb{R}^{(m,d)}$ and a target sequence $T \in \mathbb{R}^{(m,d)}$ of m timesteps and d features, then the algorithm for the Transformer model is composed of several steps.

2.2.1 Tokenization & Embeddings layer

The first step of the algorithm is the **tokenization** of the input and target sequences: each sequence is mapped from their original formulation (for instance, in Natural Language Processing, dealing with sequence of words), into a new representation, where the sentence is broken down into smaller symbols - sentences, words or even letters, and each symbol is associated to unique position into a vocabulary.

The dimension of the vocabulary is an hyperparameter, and the choice of its value depends on the specific task: for instance, in case of sequence translation, from a language to another, all input and target sequences are mapped to two different vocabularies, whose lengths depend on the specific languages of the problem.



[CLS]	This	is	a	input	.	[SEP]
101	2023	2003	1037	7953	1012	102
0.0390, -0.0123, -0.0208, ...	-0.0558, 0.0151, 0.0031, ...	-0.0440, -0.0236, -0.0283, ...	0.0119, -0.0037, -0.0402, ...	0069, 0.0057, -0.0016, ...	0.0199, -0.0095, -0.0099, ...	-0.0788, 0.0202, -0.0352, ...

Figure 9: Example of tokenization and embeddings of an English sentence

Moreover, each tokenized input sentence is concatenated with a Start-of-Sentence (SOS) token an the beginning, an End-of-Sentence (EOS) token at the end; SOS and EOS tokens are unique tokens that will have the sole purpose of indicating the beginning and end of the sentence to the machine.

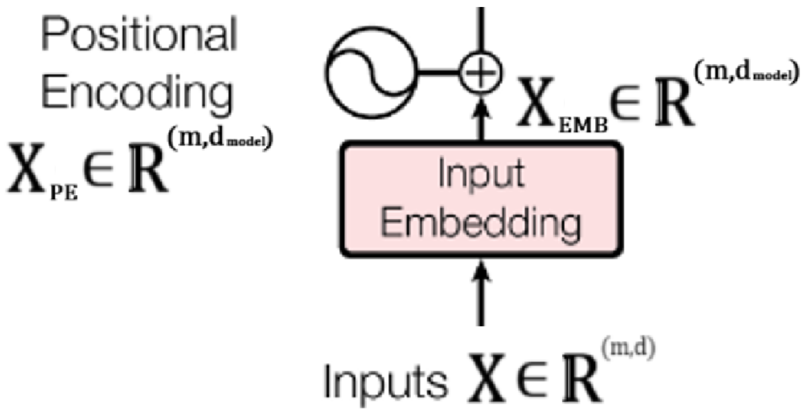

Positional Encoding
$X_{PE} \in R^{(m,d_{model})}$
$X_{EMB} \in R^{(m,d_{model})}$
Input Embedding
Inputs $X \in R^{(m,d)}$

Figure 10: Embedding and Positional Encoding of input sequences, the same for target sequences

Then, the next step is the **Embedding Layer**: each tokenized sequence gets mapped to a new feature dimension, in particular, from the original (m, d), to a new (m, d_{model}) dimensionality, where d_{model} is again an hyperparameter (whose value must be chosen by having a trade-off between accuracy and complexity).
The actual values of each embedded sequence are learned by the model during training, in order to find their optimal values depending on the considered task.

Along Input Embedding, **Positional Encodings** of the sequences are added: since the model has no internal notion on how to treat different words and their order in the sentences, it must be given this information with an off-line computation, following the formulas:

$$X_{PE}\,[j, 2i] = sin\left(\frac{j}{10000^{\frac{2i}{d_{model}}}}\right) \tag{4}$$

$$X_{PE}\,[j, 2i+1] = cos\left(\frac{j}{10000^{\frac{2i}{d_{model}}}}\right) \tag{5}$$

Each column represents a word, while each row represents the features of d_{model}.
Positional encoding is fixed with respect to the context of the input data, so it can be computed once and re-used for every subsequent step. In order to add this information in a meaningful and pattern-behaving way, sine and cosine functions have been employed, and the resulting encoding can be visualized in Figure 11.

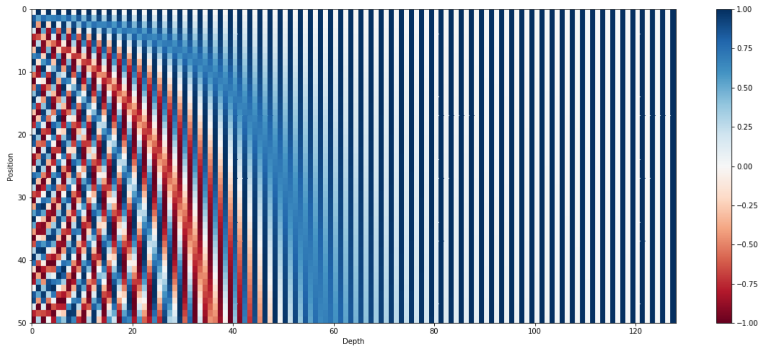


Figure 11: Positional Encoding, credits to kazemnejad.com

2.2.2 Encoding layer

The key feature of the Transformer model is the Multi-Head Attention mechanism, already explained in chapter 2.1.2. The output of the attention block then is passed through an **Add&Normalization Layer**, which is a composite block made by:

- a **residual block**, which takes the output of the attention block and sums it to its input, in order to highlight the patterns in data;

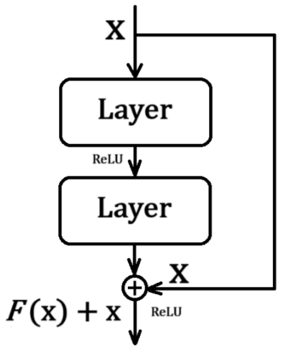


Figure 12: Example of a bi-layer residual block, with ReLU activation function

- a **"LayerNorm" normalization**: given a batch of N items, LayerNorm finds the optimal representation of the normalized data, as:

$$\hat{x}_j = \frac{x_j - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

  where μ_j and σ_j^2 are the mean and variance of the new representation, and will be learned during training in order to optimally internalize concepts. The two parameters introduce some fluctuations in the data, and the network will learn how to tune then to introduce fluctuations when necessary.
  While BatchNorm applies this transformation feature by feature, comparing data from the N different items, LayerNorm acts separately on each sample, therefore combining the features. The smoothing

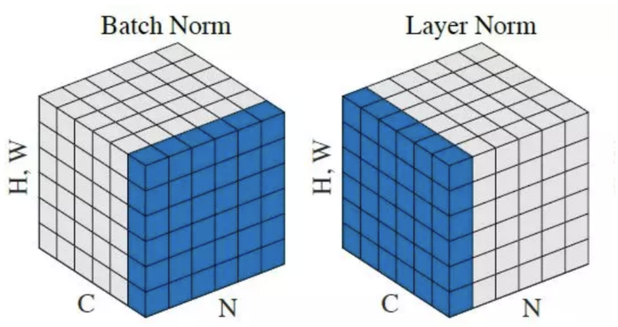


Figure 13: BatchNorm and LayerNorm compared, credits to PyTorch.com

  coefficient ϵ is an hyperparameter, whose value is usually very small, in order to avoid divisions by zero.

Finally, the output of the Add&Norm block is passed through a **Feed-Forward Layer** of d_{ff} (hyperparameter) densely connected units, and yet another Add&Norm block.
Since the Encoder block is composed by N (hyperparameter) encoder layers, the output of the first encoder layer is sequentially passed to the second encoder layer, and so on; moreover, the output of each encoder layer will be the keys and values matrices for the Crossed-Attention layers in the decoder.

2.2.3 Decoding layer

The first part of the decoder block is the same as the encoder: the target sequence gets tokenized, embedded into the latent dimension d_{model}, and added to the Positional Embedding.

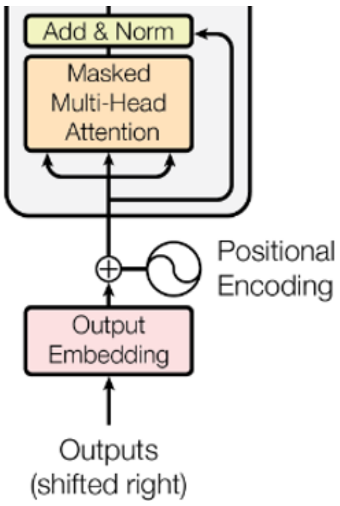


Figure 14: Target encodings and first part of decoder layers, credits to the authors of the paper

Then, the same copying of the sequence as in the encoder takes place, as the embedded sequences are copied three times into queries, keys and values:

$$Q = T_{i,emb} \longrightarrow Q' = QW^Q \quad \textbf{weighted queries}$$

$$K = T_{i,emb} \longrightarrow K' = KW^K \quad \textbf{weighted keys}$$

$$V = T_{i,emb} \longrightarrow V' = VW^V \quad \textbf{weighted values}$$

and then a **Causal Multi-Head Attention** mechanism computes the first evaluation of the target sequence: while performing the above-mentioned Multi-Head Attention, the timesteps of the target sequences are masked in such a way that only the subsequent steps can see the previous ones, and not vice versa. In this way, the model can have a sense of causality, meaning that each timestep only sees its past.
This can be achieved by element-wise multiplying the attention scores obtained with the regular Multi-Head Attention, with a lower triangular matrix of 1s; by doing so, all the scores of subsequent timesteps to any step are set to zero.

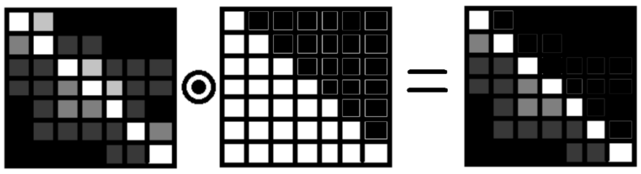


Figure 15: Example of causal attention masking, whiter entries means higher correlation, while blacker ones means lesser correlation

In mathematical terms, Causal Multi-Head Attention can be written as:

$$CausalMultiHead(Q, K, V) = concat(head_1, head_2, ..., head_h)W^0 \tag{6}$$

$$head_j(Q_j, K_j, V_j) = softmax\left(\frac{Q_j K_j^T}{\sqrt{d_k}}\right) L^1 V_j \quad \forall j = 1, ..., h \tag{7}$$

where:

$$L^1 = \begin{bmatrix} 1, & 0, & 0, & ..., & 0, & 0 \\ 1, & 1, & 0, & ..., & 0, & 0 \\ & & & ... & & \\ 1, & 1, & 1, & ..., & 1, & 0 \\ 1, & 1, & 1, & ..., & 1, & 1 \end{bmatrix} \in \mathbb{R}^{(m,m)}$$

The output of the Masked Multi-Head Attention block is then passed through the Add&Norm block, which again is a residual block with LayerNorm normalization.
Then, the second part of the decoder layer is another Attention mechanism, but with the queries coming from the Masked Multi-Head Attention block (after normalization), and the keys and values being the output of the network Encoder. This particular mechanism is called **Crossed Multi-Head Attention**, as it compares its own re-elaboration of the input sequences, with those of the target sequences. The computations of the Crossed

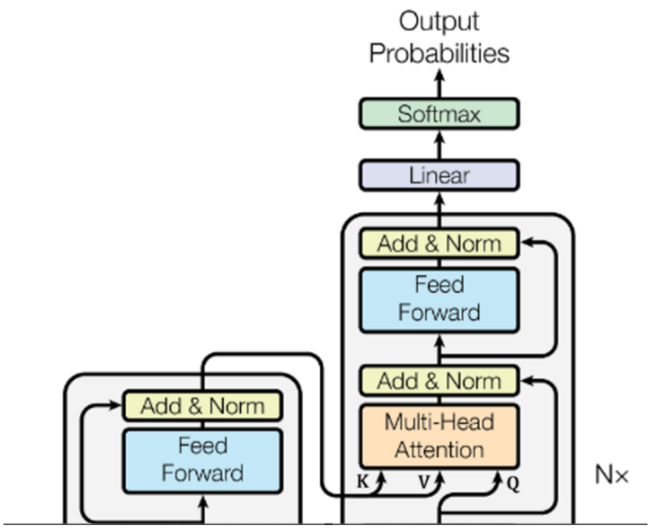


Figure 16: Second part of the decoder, with Crossed Multi-Head Attention, credits to the authors of the paper

Multi-Head Attention mechanism are the same as in Equation (2) and (3); its output is then passed through a Feed-Forward Layer and another Add&Norm Layer.
At the end of the Decoder block, one last Linear layer is applied, to transform the output sequence into a matrix of (m, d_{dict}) dimensions, where d_{dict} is the dimension of the embedding dictionary of the target sequences. Then, a softmax layer is applied, in order to obtain the probabilities, for each timestep, of each ”word” of the embedding dictionary.

Since the last layer is a softmax, Cross-Entropy Loss has been the most likely candidate for loss function (documentations available on pytorch.org), comparing the output probabilities with the actual tokenization labels of each sequence’s timestep.

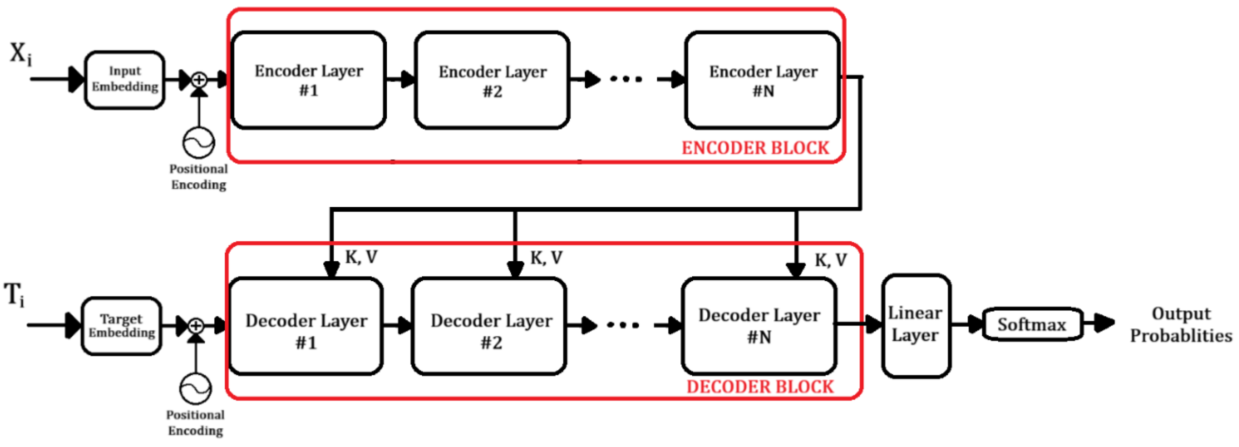


Figure 17: Full model of the Transformer, unwrapped

2.3 Training and Inference

2.3.1 Usage of the model

The Transformer model has versatile applications in Natural Language Processing, two of some of the most employed are Sequence Translation and Autoregression.

In **Sequence Translation**, the machine is asked to translate input sentences of a chosen language, into target sentences of another language; under this setup, the training set will be composed of:

$$\mathbb{D}_{(n,m,d)} = \{(X_1, T_1), (X_2, T_2), ..., (X_n, T_n)\} \quad \text{where } X_i, T_i \in \mathbb{R}^{(m,d)}$$

Zero padding might be probably need to be involved, as different languages have different sentence lengths - even expressing the same concept with the same words, so a common choice of m is selecting a value that exceeds the longest training sequence (in order to have some margin over testing), and then zero-pad all sequences to that specific length.

During Sequence Translation, it's common place to append SOS and EOS tokens, in order to make the model learn when to start and end sentences:

$$\textbf{Encoder Input} = [SOS], \text{Tokenized Input }, [EOS], [PAD], ..., [PAD]$$
$$\textbf{Decoder Input} = [SOS], \text{Tokenized Output }, [PAD], ..., [PAD]$$
$$\textbf{Decoder Label} = \text{Tokenized Output }, [EOS], [PAD], ..., [PAD]$$

In **Autoregression**, instead, the job of the model is to predict the future timestep(s), given a history of a certain number of timesteps:

$$\mathbb{D}_{(n,m,d)} = \{(X_1, t_1), (X_2, t_2), ..., (X_n, t_n)\} \quad \text{where } X_i \in \mathbb{R}^{(m,d)} \text{ and } t_i \in \mathbb{R}^{(s,d)}$$

where s is the number of future steps to be predicted for each sequence, and again the same construction with [SOS], [EOS] and [PAD] tokens takes place, in order to make the target sequences of the same length m.

In this assignment, the model has been employed for Sequence Translation.

2.3.2 Training

The input sequences are tokenized and fed into the Encoder, which will produce the keys and values for Cross Multi-Head Attention in the Decoder; at the same time, the tokenized target sequences are presented to the decoder, timestep after timestep.
At each timestep, the input of the decoder changes, each time being the output of the previous iteration; in the end, the result will be a matrix of dimensions (m, d_{dict}), each row being a timestep and each column being the probability distribution of each word in the vocabulary.
Cross-Entropy Loss is then used to compute the error for back-propagation, comparing the output probabilities with the actual labels of the tokenized output.
This process happens in **only one timestep**, with no "for" loops to make up the process, granting faster training sessions than regular RNNs.

2.3.3 Inference

At inference phase, the decoder labels are not present, thus the model will construct the output probabilities by re-using the previous timesteps:

- at first, the tokenized input sentences are presented to the Encoder, obtaining in the end the keys and values for the decoder;
- meanwhile, the decoder is presented with the only [SOS] token, producing the first actual timestep of the sequence as output;
- this output timestep is then used as decoder's input in the next timestep, producing target tokens until the [EOS] token is selected by the model;

The two methods proposed by the paper "Attention is all you need" for the "next-timestep" heuristics are **Greedy Search**, which consists in column-wise applying the max function to the output probabilities, thus selecting the most probable token as the next; and **Beam Search**, which keeps track of a given number of optimal solutions, and constructs various possible answers from the most probable tokens (so, not only considering the one with maximum probability, but also sub-optimal choices).

3 Transformer model & Robotics applications

The Transformer model opened up new possibilities for real-world Robotics applications: from Computer Vision to Kinematic Control Layer, many innovative ideas have risen in the years following the "Attention is all you need" paper [1].

3.1 Vision Transformers and VLMs

As an actual implementation aside from Natural Language Process, the **Visual Transformer** [2], or ViT, is an image classification model, that works by splitting the image into fixed-sized patches, which will then be passed through a linear layer as the tokenization step. This projected patches, alongside with their position

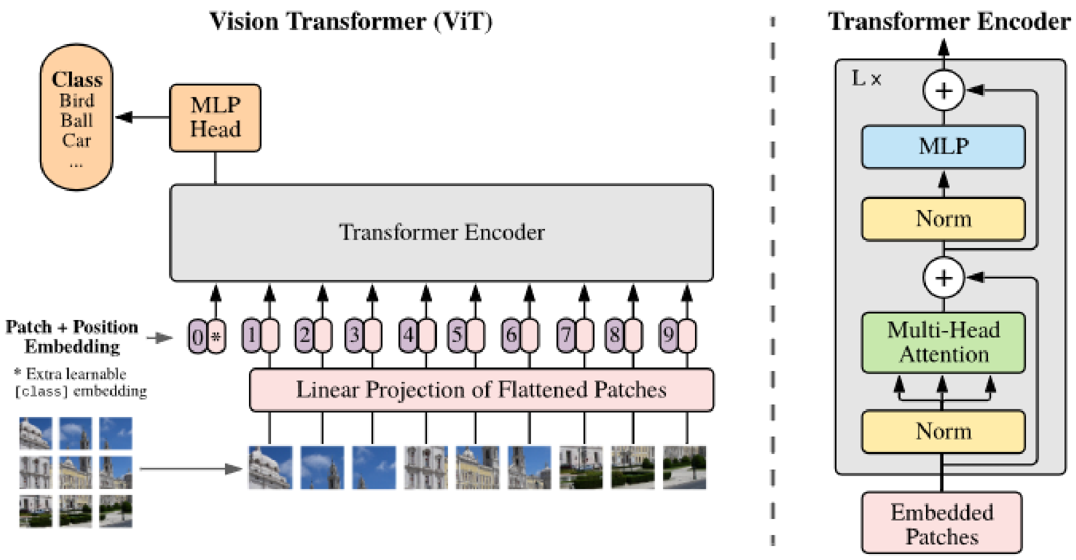


Figure 18: Model overview of the ViT, credits to the paper's authors [2]

embeddings, are then fed into a regular transformer encoder, which, in the end, will produce the predicted class for the input data.
The model is trained with Supervised Learning, meaning that the dataset has to be made of labeled images; through the minimization of the Cross Entropy Loss function, the model is optimized with Gradient Descent techniques, such as the common choice of the ADAM optimizer.
By combining the two aforementioned applications of Transformer models, namely NLP and Computer Vision, Vision-Language models (VLMs) are born. In particular, in the 2021 paper "Learning Transferable Visual Models From Natural Language Supervision" [3], the authors proposed the **Contrastive Language-Image Pre-Training** model (CLIP), capable of joining together image informations and text labels.

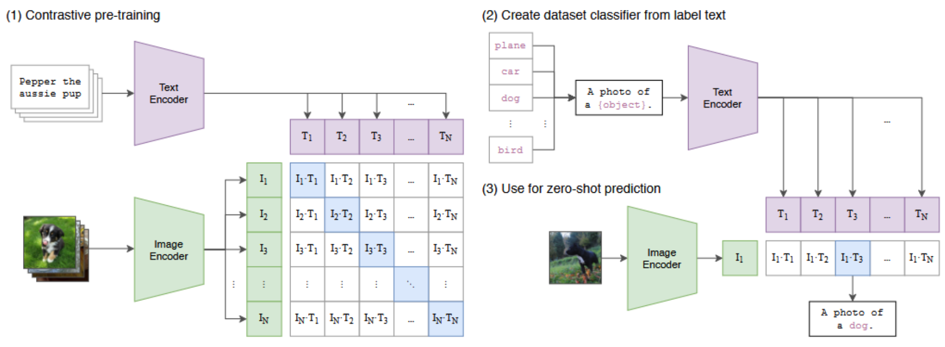

(1) Contrastive pre-training
Pepper the aussie pup
Text Encoder
Image Encoder
T_1 T_2 T_3 ... T_N
I_1 I_2 I_3 ... I_N
$I_1 \cdot T_1$ $I_1 \cdot T_2$ $I_1 \cdot T_3$ ... $I_1 \cdot T_N$
$I_N \cdot T_1$ $I_N \cdot T_2$ $I_N \cdot T_3$ ... $I_N \cdot T_N$
(2) Create dataset classifier from label text
plane
car
dog
bird
A photo of a {object}.
Text Encoder
(3) Use for zero-shot prediction
Image Encoder
A photo of a dog.

Figure 19: CLIP Model overview, credits to the paper's authors [3]

This **multimodality** is able to enhance the performances of the model, which can be combined with Diffusion techniques [4][5] to provide better autoregression capabilities.

3.2 Visual-Language-Action Models (VLAs)

By combining the VLMs with Robotics applications, in 2024, the paper ”OpenVLA: An Open-Source Vision-Language-Action Model” [6], proposed **OpenVLA**, a state-of-the-art AI-driven robotic control for manipulators, as shown in the following figure.

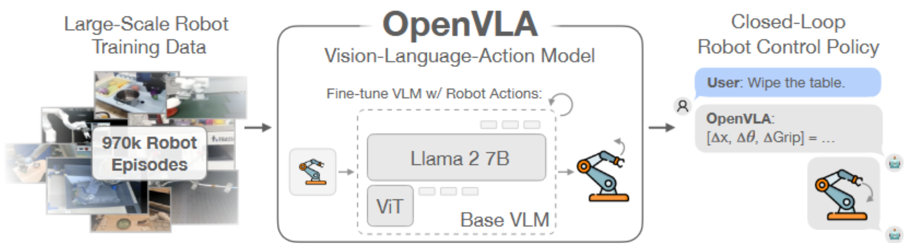

Large-Scale Robot Training Data
970k Robot Episodes
OpenVLA
Vision-Language-Action Model
Fine-tune VLM w/ Robot Actions:
Llama 2 7B
ViT
Base VLM
Closed-Loop Robot Control Policy
User: Wipe the table.
OpenVLA: [Δx, Δθ, ΔGrip] = ...

Figure 20: Model overview of the OpenVLA, credits to the paper’s authors [6]

Given an instruction provided by the user, and an image observation through a camera, the model is trained to predict the end-effector attitude and gripper widening, through Supervised and Reinforcement Learning techniques.

4 Conclusions

As shown in this paper, the Transformer model can be employed in many areas of Computer Science, from its original purpose of Natural Language Processing, to Computer Vision and Robot Control applications.
Its versatility and pattern representation have played a crucial role in modern Deep Machine Learning, being at the heart of popular AI commercial innovations - mainly chatbots and LLMs, and promising many undiscovered applications that might shape the next decade of research of the AI and Robotics fields.

5 Bibliography

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser and Illia Polosukhin. Attention Is All You Need, *31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA* https://arxiv.org/pdf/1706.03762, 2017

[2] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit and Neil Houlsby. An Image is Worth 16x16 Words: Transformer for Image Recognition at Scale, *ICLR 2021*, https://arxiv.org/pdf/2010.11929, 2021

[3] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger and Ilya Sutskever. Learning Transferable Visual Models From Natural Language Supervision, https://arxiv.org/pdf/2103.00020, 2021

[4] Jonathan Ho, Ajay Jain and Pieter Abbeel. Denoising Diffusion Probabilistic Models, 34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada. https://arxiv.org/pdf/2006.11239, 2020

[5] William Peebles and Saining Xie. Scalable Diffusion Models with Transformers, https://arxiv.org/pdf/2212.09748, 2023

[6] Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan Foster, Grace Lam, Pannag Sanketi, Quan Vuong, Thomas Kollar, Benjamin Burchfiel, Russ Tedrake, Dorsa Sadigh, Sergey Levine, Percy Liang and Chelsea Finn. OpenVLA: An Open-Source Vision-Language-Action Model, https://arxiv.org/pdf/2406.09246, 2024