Neural Machine Translation - https://arxiv.org/pdf/1409.0473.pdf

The usual encoder-decoder structure for language translation has the drawback of not being able to cope with long sentences, because in such a setting every sentence has to be encoded in a fixed-length vector (by the encoder).

The Neural Machine translation mechanism solves this problem by allowing the decoder to **learn** "where" (i.e., in which of the hidden states produced by the input sequence) to put its "attention" in order to compute the probability of some output word.

Encoder: through a **bidirectional RNN**, for an input sequence $\{x_1, \dots, x_T\}$ produces a sequence of hidden states $\{h_1, h_2, \dots, h_{T_x}\}$ where each h_j is given by the concatenation of the hidden states of the "forward" RNN $\vec{h}_j$ and of the "backward" RNN $\overleftarrow{h}_j$.

$$h_j = [\vec{h}_j, \overleftarrow{h}_j]$$

Since the annotations $\vec{h}_j$ contain information about the previous states that are close to the input word x_j and $\overleftarrow{h}_j$ contain information about the following ones, $\boldsymbol{h_j}$ **is a summary of the words around the word** $\boldsymbol{x_j}$.

Decoder:

A classical decoder models the output sentence as the product of the probabilities of each word. Each of these probabilities is a joint distribution over

- The previously predicted words
- A context vector c, which is a nonlinear function of the hidden states produced by the encoder

$$p(\mathbf{y}) = \prod_{t=1}^{T} p(y_t \mid \{y_1, \cdots, y_{t-1}\}, c)$$

$$c = q(\{h_1, \cdots, h_{T_x}\})$$

In such a setting, for each sentence there is only one context vector, and the probability of each new word y_i is a nonlinear function g of the current state s_t of the RNN, the previous words and the context vector

$$p(y_t \mid \{y_1, \cdots, y_{t-1}\}, c) = g(y_{t-1}, s_t, c).$$

In the new model proposed by the neural machine, the probability of a new word directly takes into account a different context vector c_i for each word of input sentence $\boldsymbol{x} = \{x_1, \dots, x_{Tx}\}$

$$p(y_i|y_1, \dots, y_{i-1}, \mathbf{x}) = g(y_{i-1}, s_i, c_i),$$

We compute the hidden state s_i as a function of

- Previous state s_{i-1}
- Previous word y_{i-1}
- Context vector c_i associated to input word x_i

$$s_i = f(s_{i-1}, y_{i-1}, c_i)$$

NB: h_i are the states produced by the bidirectional RNN encoder. States s_i are the states that the decoder uses for translation.

Here comes the concept of **attention mechanism**: each context vector c_i is computed as a weighted sum of the hidden sates, in which the weights α_{ij} are a "compression" (**SoftMax distribution**) of the energies e_{ik} of all words $x_k \in \{x_1, \dots, x_{Tx}\}$ with respect to output word y_i.

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j \qquad \alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})} \qquad e_{ij} = a(s_{i-1}, h_j)$$

Each energy e_{ij} is computed as a function a (usually a **feedforward neural network**) of the the (output) state s_i for output word y_i and the (input) state h_j for word x_j.

So, e_{ij} is a sort of "attention" function between output word y_i and input word x_j, and as it is computed through a FFNN it can be learned! With the SoftMax function we convert such energies into **probabilities** α_{ij}.

This means that this machine is able to learn how to compute a context vector as a (weighted) sum of input hidden states by **learning on which ones to put its attention**, in the form of probabilities predicted by a feedforward neural network, that receives as input an input state and, for each output state i, tells how much it is important in order to it. (**Corretto??????**)

NB: the number of hidden states h_i or s_i is variable, but their lengths are fixed! Otherwise, we couldn't design a FFNN! (**Mia supposizione**)

To wrap up, we have

- An encoder (a bidirectional RNN) that learns how to encode input sequences $\{x_1, \dots, x_{T_X}\}$ into sequences of hidden states $\{h_1, \dots, h_{T_X}\}$ by considering the close words (forward and backward)
- A decoder (an RNN) that learns how to compute the next word combining information on the previous predicted word and the current decoder state s_t, which is computed through an **attention mechanism** trained to recognize whether some input state should be considered or not.

Considerazioni:

- mi sembra di capire che tale FFNN faccia una specie di "classificazione" (senza selezionare un output con l'argmax) in cui ogni classe appartiene ad "indice/posizione" della frase di output, allo scopo di calcolare la probabilità che un determinato stato di input sia rilevante per ogni parola/classe/posizione.
- Come avviene il training?

Attention is all you need

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

- https://www.youtube.com/watch?v=dichIcUZfOw&ab_channel=Hedu-MathofIntelligence (video1)

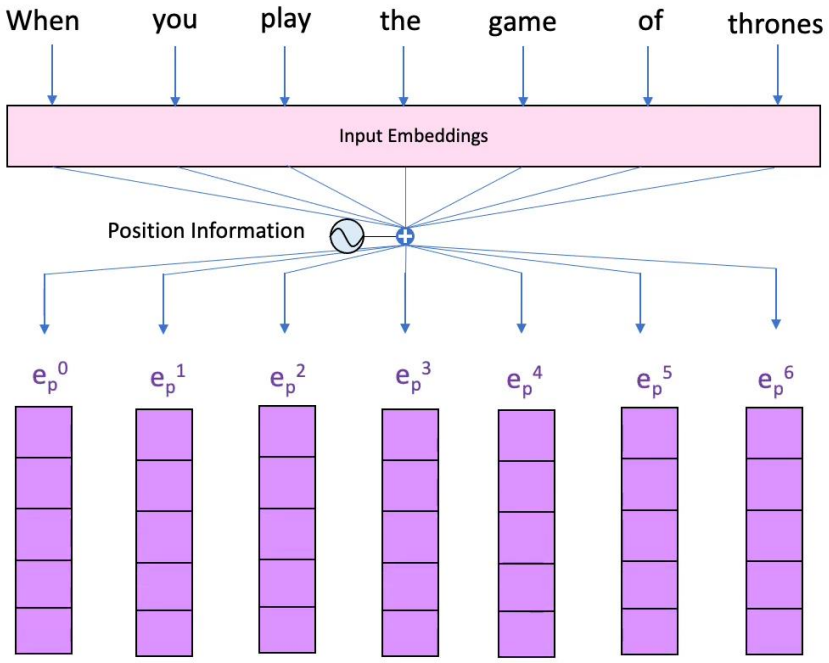
- https://www.youtube.com/watch?v=mMa2PmYJlCo&ab_channel=Hedu-MathofIntelligence (video2)

- https://www.youtube.com/watch?v=gJ9kaJsE78k&ab_channel=Hedu-MathofIntelligence (video3)

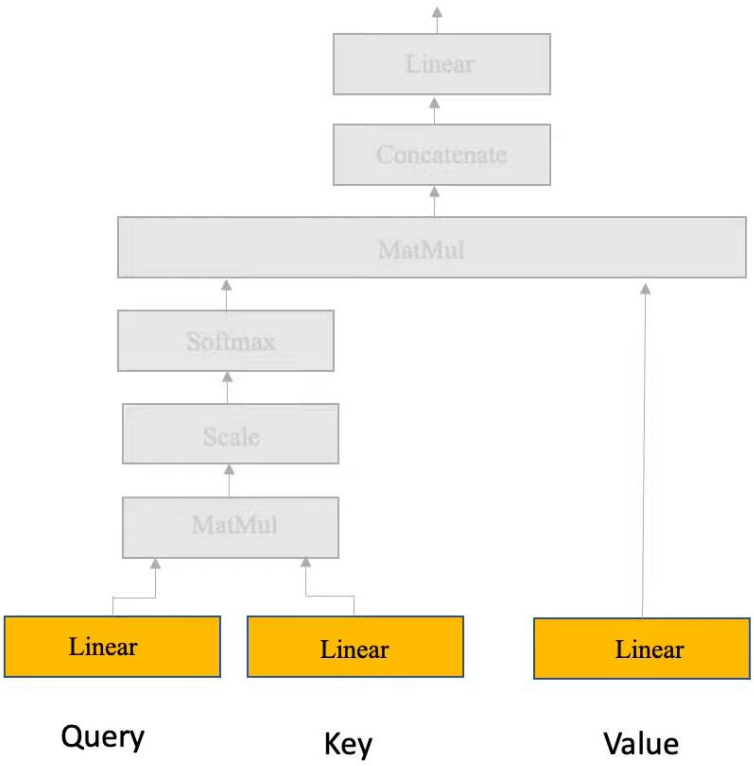
A new architecture based only on the concept of attention is the Transformer. It still has an encoder/decoder structure, but it has not the recurrent component of neural machines. This allows to process all the words in a sentence in parallel!

The encoder has, in its first part, some "pre-processing" layers that prepare the input

- The input is coded into some bag of words or other representation
- An **input embedding layer** project those words into a multi-dimensional space: this means that each word is represented by a vector, and each position of a vector represents a dimension/feature. Intuitively, if two words have similar value for some feature it means that they are similar according to that feature (e.g., they both are verbs). Usually, the projections in this space are **learned** (i.e., they are improved during training)
- A **position embedding** is added to our input embedding: the position of a word in a sentence is a critical information and processing all the words in parallel we lose it. So, we have to "embed" it! (The video explains how it is done in practice). At the end of this step, we have, for each word, an embedding vector of d dimensions.



After this layers we have a **MULTI-HEAD ATTENTION LAYER**: in this layer we have, initially, 3 independent linear layers (linear layer = fully connected layer without nonlinear activation function: a weighted sum). The one on the right computes the **values**, the one in the middle the other ones respectively the **key** and the **query**. To do an analogy, when we search something on YouTube we put, in the search bar, a **query**. YouTube compares this query to a set of keys, which are (for example) the titles of the videos. If some video has a **key** that has a similarity with the query, its content (**value**) is retrieved.



All these layers receive as input the input embedding.

The point is to compute a similarity between the query and the key (that are two matrices). This is done through a **cosine similarity**: the dot product between two vectors is maximum (equal to 1 for unitary vectors) when they point in the same direction, and minimum (-1 for unitary vectors) when they point in opposite directions. This concept can be formalized with the formula $similarity(A, B) = \frac{A*B}{scaling}$. If we plug in the query matrix Q and the key matrix K we obtain a similarity matrix called Attention Filter

$$similarity(Q, K) = \frac{Q * K^T}{scaling}$$

The attention filter has this kind of structure: for each pair of words in the input sentence, it computes a number. At the beginning of the learning process these can be random numbers, but after the training we can see that they take some meaning.

	When	you	play	the	game	of	thrones
When	89	48	41	36	35	40	19
you	67	91	11	92	17	99	11
play	91	10	11	11	12	41	98
the	11	96	28	12	98	11	00
game	76	11	91	24	12	12	12
of	11	29	77	78	22	93	13
thrones	11	87	12	12	13	98	19

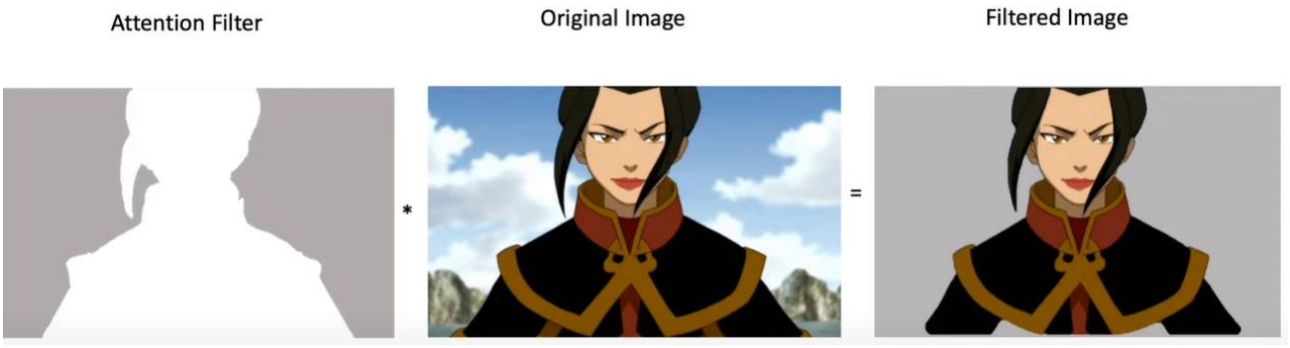
	When	you	play	the	game	of	thrones
When	89	20	41	10	55	78	59
you	90	98	81	22	87	15	32
play	29	81	95	10	90	30	92
the	10	22	67	12	88	40	89
game	22	70	90	56	98	44	80
of	10	15	30	40	44	44	59
thrones	59	72	92	90	13	59	99

These numbers are **attention scores**: for example, we can see that the word “game” has a high attention towards the words “play”, “thrones” and (of course) “game” itself. Such a matrix is then scaled and “squashed” through a SoftMax layer.

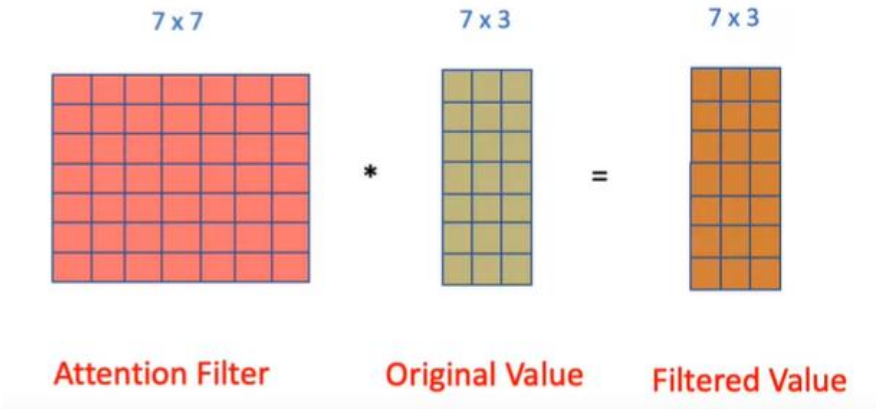
After this, our attention filter looks like this

	When	you	play	the	game	of	thrones
When	1.00	0.00	0.00	0.00	0.00	0.00	0.00
you	0.00	0.97	0.03	0.00	0.00	0.00	0.00
play	0.00	0.00	0.68	0.00	0.10	0.00	0.22
the	0.00	0.00	0.00	0.65	0.14	0.00	0.21
game	0.00	0.00	0.03	0.02	0.72	0.00	0.23
of	0.00	0.00	0.00	0.00	0.00	1.00	0.00
thrones	0.00	0.00	0.05	0.03	0.17	0.00	0.75

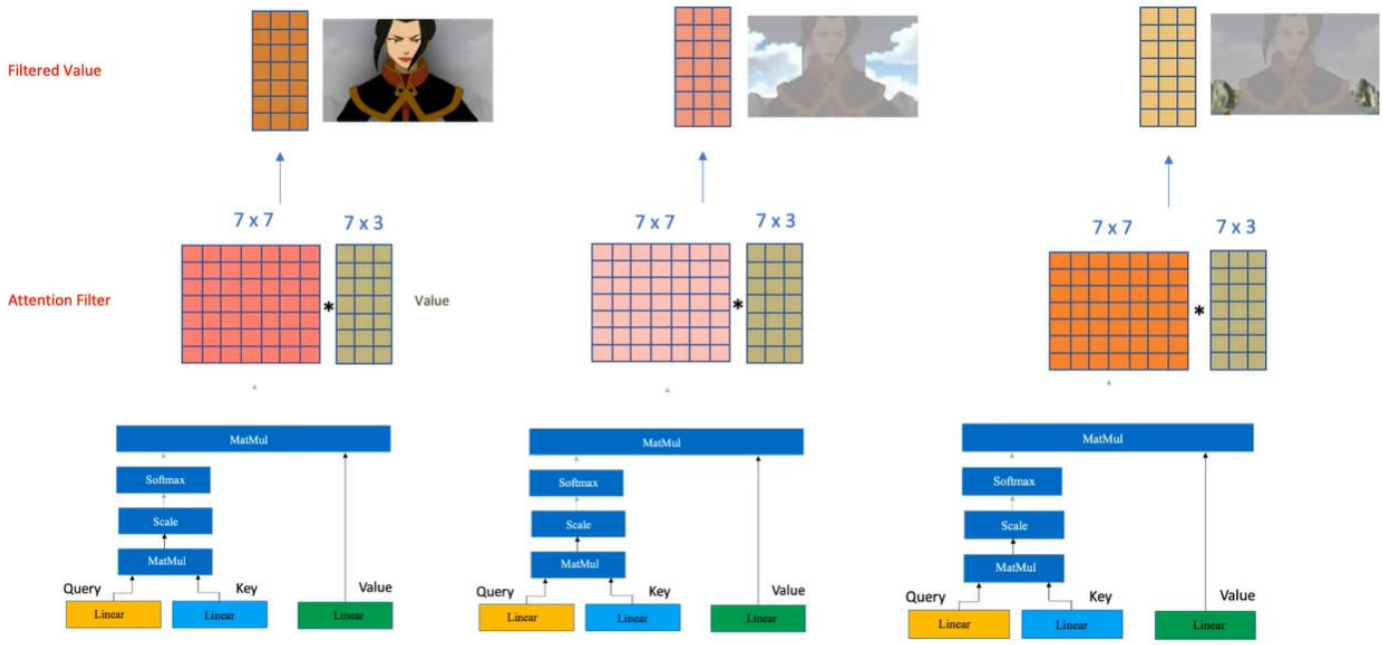
To recap: we have the right part (“value” layer) that just projects the input embedding into a new vector space (usually with less dimensions), while on the left we have a network that produces an attention filter. The MatMul layer then performs a multiplication between the transformed input embedding and the attention filter: what we obtain is a **filtered input**, i.e., the parts of the input on which the algorithm has learned to put its attention. Intuitively it could be seen as something like this



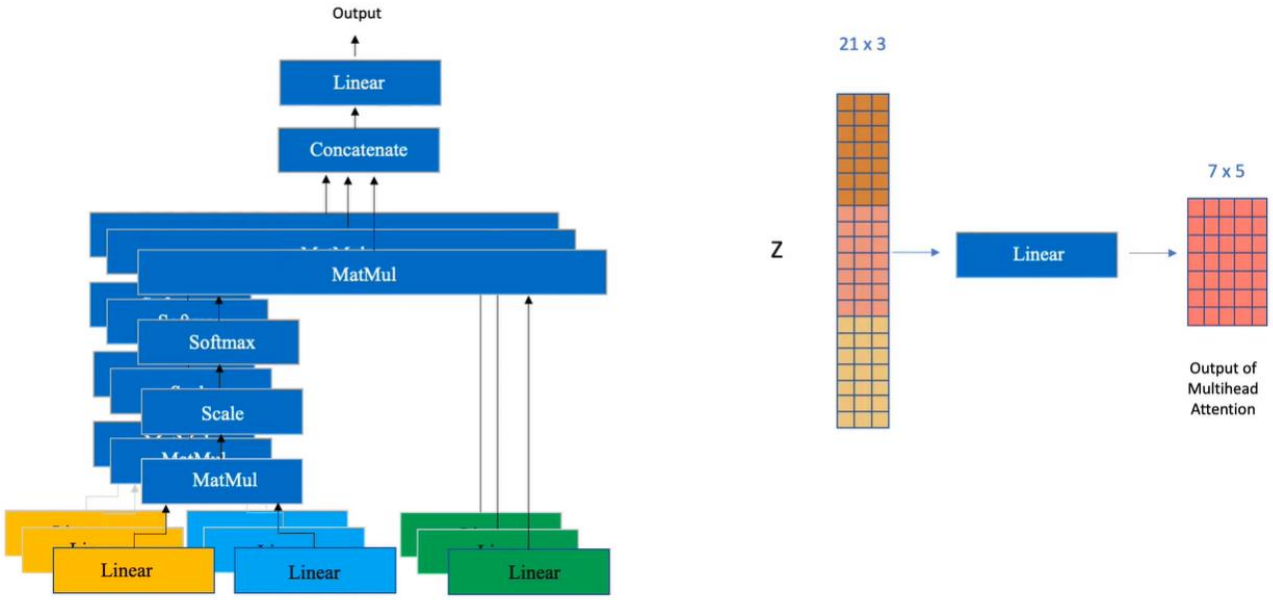
After this MatMul layer, we have obtained a filtered value, i.e., a data structure that contains the “selected” features, that are chosen to be more important for solving the task.



The term **MULTI** headed attention comes from the fact that transformers don't learn just one attention filter, but many! And each one focuses on different features of the input



The "concatenate" layer simply concatenates the different filtered values, and then another linear layer transforms it in a different representation to shrink its size.



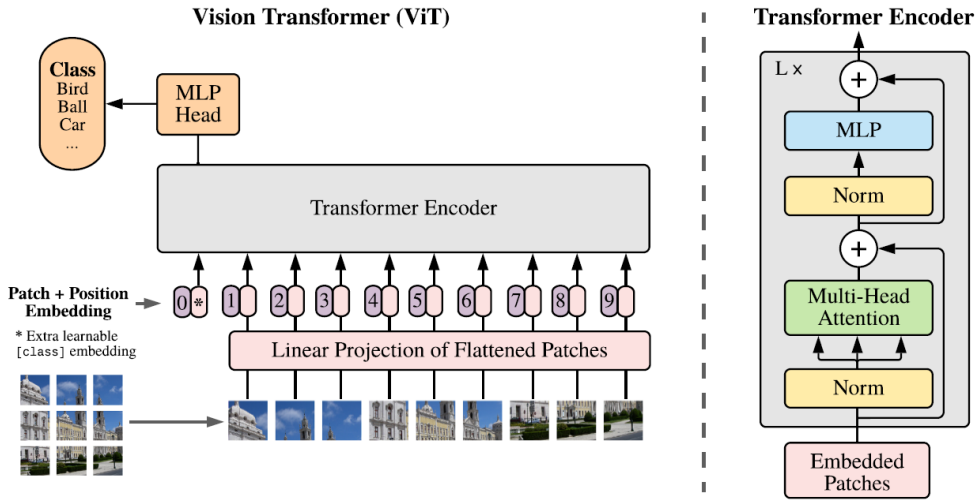
The encoder, apart from these modules, also has **skip connections**: this is done to preserve initial information through the layers! This means that the filtered input (the output of the multi-headed attention) is added to the initial input. After this, it is normalized (to make convergence faster and more stable). Then, it is transformed by another FFNN and normalized again. This is a typical layer of an encoder, and it is the main building block of the ViT, presented in the next paper. The decoder is not really relevant to the aim of my thesis, but it is explained in the videos https://www.youtube.com/watch?v=gJ9kaJsE78k&ab_channel=Hedu-MathofIntelligence and https://www.youtube.com/watch?v=iDulhoQ2pro.

An Image is worth 16x16 words - https://arxiv.org/pdf/2010.11929.pdf

This approach tries to perform image classification without using CNNs, but with only the encoder of a Transformer. It seems that while on mid-size datasets transformers' accuracies are below the ones of CNN, on very big datasets Transformers work really well. It is recommended then to **pre-train** the ViT on some large dataset and then fine-tune on a task specific datatest.

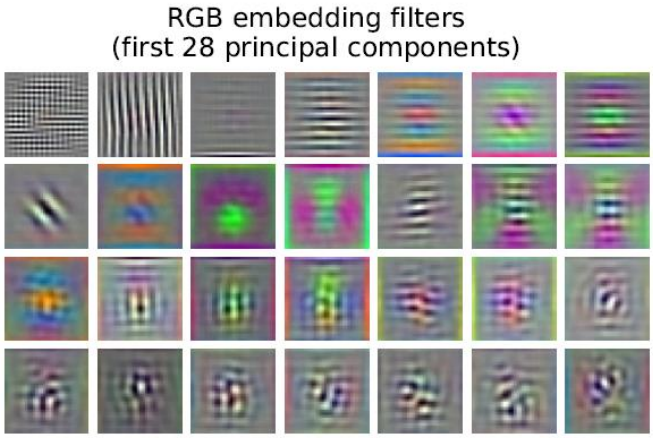
The key idea is that since the transformer is able to learn self-attention on a 1-dimentional input (a sentence), if we reshape an image $H \times W \times C$ into N 2D patches $P \times P$ and flatten them, we obtain a data structure $N \times (P^2 * C)$, that can be interpreted as a sentence of N words where the length of each word is $(P^2 * C)$.

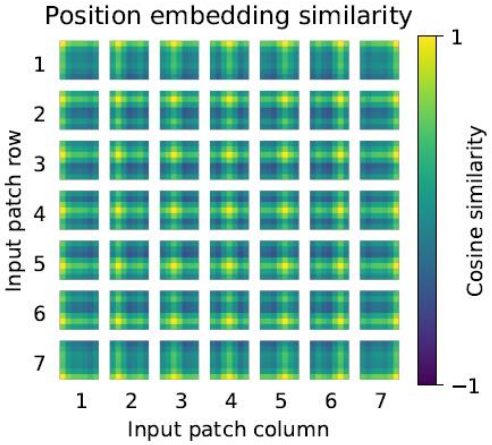
These patches has to be considered like words: so, they pass through the layers of word embedding and positional embedding. Such an encoder is then used as feature extractor: on top of it we put a fully connected feed forward neural network and, like the head of a CNN, perform the last steps of classification (SoftMax).



The advantages of such an approach are

- since a Transformer is able to learn the self-attention between any of the words in the sentence, no matter how long the sentence is, then a ViT can learn to integrate information across the entire image! In a CNN, this happened only in the deepest layers
- the Transformer is cheaper to train in terms of time and computational effort





In the right image we can see the similarity with respect to position embeddings: in general, each patch puts its attention on other patches on the same row or column.

In the left image instead, we can see some low level features learned by the first layers of a transformer.

How to train your ViT – Data augmentation and regularization - https://arxiv.org/pdf/2106.10270.pdf

The transformer has a weaker inductive bias w.r.t. CNNs, so it needs much more data to obtain good results. When we have small datasets, this implies that we have to do data augmentation and regularization.

This paper demonstrate that finding a good compromise between regularization and data-augmentation has the same effects than increasing 10 times the size of the dataset.

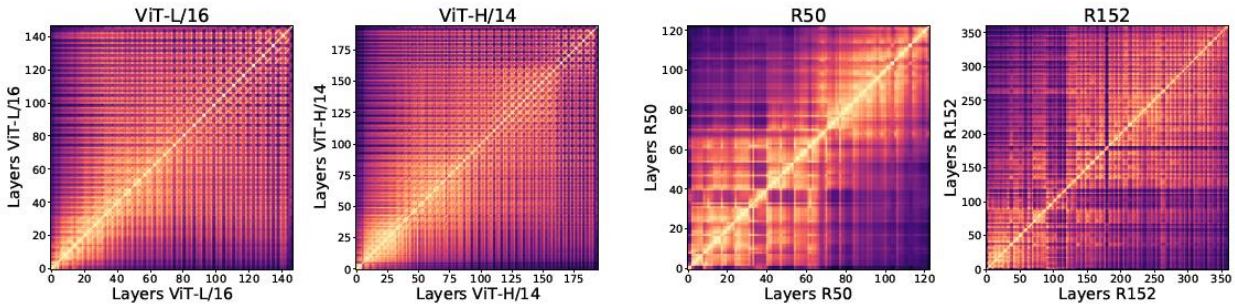
The first point is that Transfer Learning is crucial to obtain good performances, especially when we have a small dataset.

For what concerns Augmentation and Regularization, it always helps when we have a mid-sized dataset (e.g. Image-net-1K), while when we have bigger datasets (image-net-21K) it helps only in largest models, hurting the performances in all the other ones. Increasing the number of epochs, the number of models that take advantage on regularization/augmentation with big datasets increases, but the smaller models' performances continue to be hurt.

Do vision transformers see like CNN? - https://arxiv.org/pdf/2108.08810.pdf

This paper aims to study how the information is represented across the layers of a transformer and compares it to the way it happens in a CNN.

Using a similarity index called CKA (Centred Kernel Alignment), the researchers have found that while in CNNs (ResNet50 and ResNet152) we have a similarity (brighter colours in the diagram) that is more "local" (similar information can be seen in close layers), in the Transformers' layers the similarity is more or less equally distributed.



In particular, they have seen that even in the lower self-attention layers, the distance between the (multi) heads and their attended patches are a mix of short and long distances. This means that even at the lower layers the transformer has learned "global" information! This, in CNN, happened just in the higher layers, while in lower layers only local information was considered. Indeed, computing the respective receptive fields, in the Transformer's lower layers we can see that they are bigger with respect with the ones of CNN.

Segmenter: Transformer for semantic segmentation - https://arxiv.org/pdf/2105.05633.pdf

We have seen that the transformer has the important property of modelling global information even in the lower layers. This allows to perform segmentation knowing the context of the patch already in the earlier layers. This is in contrast with the classical encoder-decoder approach used before, because the convolutional filters limit the access to the global information.

It has a ViT encoder that takes input patches embeddings and produces a set of tokens, that are then up-sampled by a decoder that produces per-pixel class scores. The model is trained with a **per-pixel cross-entropy loss**.

Encoder:

- An image $\boldsymbol{x} \in \mathbb{R}^{H \times W \times C}$ is split, as in the ViT, into a sequence of N patches $\boldsymbol{x} = [x_1, x_2, \dots, x_N] \in \mathbb{R}^{N \times P^2 \times C}$ where the patch has size (P, P).
- Each patch is flattened into a 1D vector: we have N vectors of size $P^2 * C$
- We apply patch embedding and positional embedding, obtaining a sequence of patch embeddings $\boldsymbol{z_0} \in \mathbb{R}^{N \times D}$, where D is the dimension of the embedding space.
- These embeddings are fed to L encoder layers (typical ViT encoder structure), each one composed by
    - Multi-headed self-attention layer: for each head, learn how to compute value, query and key to obtain a "filtered" version of the input, on which the head has decided to put its attention, with the addition of residual connection
    - Multi-layer-perceptron / Fully connected layer that takes as input the concatenated filtered values and maps them into another vectorial space that will be taken as input to the subsequent layer (or the decoder)
- After the last layer, we have that the encoder has mapped the input sequence $\boldsymbol{z_0} = [z_{01}, z_{02}, \dots, z_{0N}]$ of N embedded patches into a contextualized sequence $\boldsymbol{z_L} = [z_{L1}, z_{L2}, \dots, z_{LN}] \in \mathbb{R}^{N \times D}$ containing semantic information that will be used by the decoder.

Decoder: it maps the encoder output $\boldsymbol{z_L} = [z_{L1}, z_{L2}, \dots, z_{LN}] \in \mathbb{R}^{N \times D}$ into a segmentation map $\boldsymbol{s} \in \mathbb{R}^{H \times W \times K}$ where K is the number of classes. In particular this is done by

- Learning a mapping between patch encodings $\boldsymbol{z_L}$ and patch-level class scores
- Up-sampling patch-level class scores to pixel-level class scores by bilinear interpolation

The decoder uses a set of K (= number of classes) learnable class embeddings (vectors of D positions) that are initially assigned randomly.

$$\boldsymbol{cls} = [cls_1, \dots, cls_k] \in \mathbb{R}^{K \times D}$$

The **Mask Transformer** will process these class embeddings jointly with the patch encodings $\boldsymbol{z_L}$ and will compute the **class masks** by making a scalar product between the L2-normalized patch embeddings $\boldsymbol{z'_M} \in \mathbb{R}^{N \times D}$ and the class embeddings $\boldsymbol{cls} \in \mathbb{R}^{K \times D}$

$$\boldsymbol{Masks} \in \mathbb{R}^{N \times K} = \boldsymbol{z'_M} * \boldsymbol{cls^T}$$

Each of these K masks is a vector of length N that contains a sequence of patches, that after being reshaped in a 2-dimensional space to form $\boldsymbol{s_{mask}} \in \mathbb{R}^{\left(\frac{H}{P}\right) \times \left(\frac{W}{P}\right) \times K}$ are bilinearly up-sampled to the feature map

$$\boldsymbol{s} \in \mathbb{R}^{H \times W \times K}$$

Domanda: come si traina un Segmenter?