UNIVERSITAT DE BARCELONA

Treball final de grau

DOBLE GRAU DE MATEMÀTIQUES I ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

EXPLORING TRANSFORMERS FOR LOCALIZING MOMENTS OF ACTIONS

Autor: Joel Diéguez Vilà

Directores: Dra. Petia Radeva i Dra. Estefanía Talavera
Realitzat a: Departament de Matemàtiques i Informàtica

Barcelona, January 24, 2023

Abstract

The field of machine learning is being applied to all aspects of our daily lives. From chatbots that speak like a human to artificial intelligence able to generate art, there are many neural networks capable of doing a better job than humans.

In 2022, the Ego4D dataset was published, a set of large-scale first-person annotated videos as never seen before, which opened the door to new branches of research on video analysis. The publication of the dataset was accompanied by several challenges, in particular, one of them is the one we will face in this memory: **the Moment Queries**, which consists of the temporal localization of concrete actions in a video. This is a highly complex problem that needs very powerful image analysis techniques!

Transformers are a type of neural network based on the concept of attention that revolutionized the field of Deep Learning. Since their appearance in 2017, they have proven their usefulness in various artificial intelligence applications such as natural text processing or computer vision, being able to outperform all previous results of Neural networks.

In this work, we have made an in-depth study of Transformers and we have tested their performance in the field of image classification. With the knowledge obtained, we have analyzed how they can be applied to the Moment Queries problem. We have developed our proposal - a model that uses a pyramid of attention mechanisms to refine the data and provide the prediction modules with the best possible information. With this implementation, we have managed to locate and catalog 28% of the actions with a tIoU value better than 0.5.

Resum

El camp de l'aprenentatge automàtic s'està aplicant en tots els aspectes de les nostres vides. Des de xats que parlen com un humà fins a intel·ligències artificials capaces de generar art, hi ha moltes xarxes neuronals que aconsegueixen fer tasques fins i tot millors que els humans.

El 2022 es va publicar la base de dades Ego4D, un conjunt de vídeos anotats en primera persona com mai no s'havia vist abans, que obria la porta a noves branques d'investigació sobre anàlisi de vídeos. La publicació del dataset venia acompanyada d'uns desafiaments, en particular un és el que tractarem d'afrontar en aquest treball: el desafiament dels Moment Queries, que consisteix en la localització temporal d'accions concretes en un vídeo. Aquest és un problema molt complex que requereix tècniques d'anàlisi d'imatges molt potents!

Els Transformers són una mena de xarxa neuronal basada en el concepte d'atenció que va revolucionar el camp del Deep Learning. Des de la seva aparició el 2017 han demostrat la seva utilitat en diverses aplicacions d'intel·ligència artificial com el processament de text natural o la visió per computador, éssent capaços de superar tots els resultats anteriors.

En aquest treball hem fet un estudi en profunditat sobre els Transformers i n'hem testejat el rendiment en el camp de classificació d'imatges. Amb el coneixement obtingut, hem analitzat com es poden aplicar pel problema dels Moment Queries i hem desenvolupat la nostra pròpia proposta. El nostre model utilitza una piràmide de mecanismes d'atenció per perfilar les dades i proveir els mòduls de predicció la millor informació possible. Amb

aquesta implementació hem aconseguit localitzar i catalogar un 28% de les accions amb un valor de tIoU millor que 0.5.

Resumen

El campo del aprendzaje automático está siendo aplicado en todos los aspectos de nuestras vidas. Desde chats que hablan como un humano hasta inteligencias artificiales capaces de generar arte, existen muchas redes neuronales que consiguen hacer tareas de forma incluso mejor que los humanos.

En 2022 se publicó la base de datos Ego4D, un conjunto de videos en primera persona anotados como nunca se había visto antes, que abría la puerta a nuevas ramas de investigaión sobre análisis de vídeos. La publicación del dataset venia acomparañada de unos desafíos, en particular uno de ellos es el que trataren de afrontar en este trabajo: el desafío de los Moment Queries, que consiste en la localización temporal de acciones concretas en un video. Este problema es muy complejo y requiere técnicas de análisis de imágenes muy potentes!

Los Transformers son un tipo de red neuronal basado en el concepto de atención que revolucionó el campo del Deep Learning. Desde su aparicióninen 2017 han demostrado su utilidad en diversas aplicaciones de inteligencia artificial como el procesamiento de texto natural o la visión por computador, siendo capaces de superar todos los anteriores resultados.

En este trabajo hemos hecho un estudio en profundidad sobre los Transformers y hemos testeado su rendimiendo en el campo de clasificación de imagenes. Con el conocimiento obtenido, hemos analizado cómo se pueden aplicar para el problema de los Moment Queries y hemos desarrollado nuestra propia propuesta. Nuestro modelo utiliza una piramide de mecanismos de atención para perfilar los datos y proveer a los módulos de predicción la mejor información posible. Con esta implementación hemos conseguido localizar y catalogar un 28% de las acciones con un valor de tIoU mejor que 0.5.

2020 Mathematics Subject Classification. 68T07, 68T10, 68T45

Acknowledgements

During the making of this work, I have received help from many people, both moral and academic support, and I would like to thank them in this section.

First of all to Petia Radeva, my supervisor in Barcelona, for proposing and motivating this work and guiding me through the first steps. Her great knowledge on Transformers has been a great help to learn them myself. She was always available to solve any doubt and give me ideas for new lines of development. I also thank her for introducing me to Estefania and the research team.

Secondly to Estefania Talavera, my supervisor from Groningen, for being there every week guiding and supporting me. For encouraging me when I thought everything was too big for me. For her knowledge and experience in neural network research and the brilliant ideas she brought to the work. In particular, the sentence she always said to me and which will stay with me forever: "Try it and see how it works", which got me out of a lot of loops when I had a thousand ideas and I did not know if they were going to work. She has helped me to understand that often we cannot control the results, because artificial intelligence sees many relationships that we do not, the best thing is to try it and if it works, try to understand why.

To Federico González, for his moral support and empathy when I was overwhelmed by the complexity of the problem and the lack of documentation.

To the University of Groningen for giving me access to their facilities even after finishing the Erasmus. Especially to the person who had to put up with my pleas every month for a little more time with the server.

To my family and friends, for supporting and encouraging me during these months of so much work and stress.

Contents

1 Introduction

The purpose of this chapter is to introduce the context of this project, the reasons that have motivated it, to show how it has been approached and, finally, to give a general overview of the memory.

1.1 Project Motivation

It is curious to see how artificial intelligence increasingly invades our daily lives and helps humans to perform all kinds of tasks. From weather predictions to self-driving cars, the truth is that deep learning has completely revolutionized what we think a machine can and cannot do.

Furthermore, since the appearance of Transformers in the publication of the paper *Attention Is All You Need* [1] in 2017, originally intended for the natural language problem, this incipient technology has demonstrated a revolutionary utility in many other fields such as computer vision, where it has given results even better than those obtained by convolutional neural networks, considered so far the best method for this problem.

Some examples of Transformers applications that are probably known by the reader would be ChatGPT3, an expert chat on a multitude of topics which can hardly be distinguished from a conversation with a human. In addition, it is capable of writing or structuring all kinds of texts, from poetry or essays to final degree project reports if proposed. Another example would be Dall-e, a generative AI that creates pictures of whatever we want in whatever style we want and that makes us question the limits of how far artificial intelligence can go, since it manages to recreate human capabilities that no machine was thought to be able to achieve, such as art generation.

Given the amazing results of these neural networks, it is not unreasonable to consider applying them to other problems to see if they can be improved. This is the idea that will be pursued in this work.

In other matters, in October 2021, Ego4D [2] was published, it is a huge dataset of first-person videos that opened the door to new groundbreaking problems. For example, the problem that we will deal with in this paper: the Moment Queries.

In a world where technology surrounds us completely and we tend to capture more and more information with our cameras, Instagram stories or even with smart glasses like Google Glass or ZShades, we can use all this information to strengthen our weaknesses, such as boosting our memory. Let us put ourselves in the situation that we have lost our keys, but, as we have glasses that have recorded our day, we can ask them where we have left them and they will tell us exactly where to find them. This is the challenge that Moment Queries tries to solve, to be able to ask exactly when a particular action has happened, allowing us to have a practically perfect memory.

Hence, the goal of this work is to address this complex challenge understanding in depth the Transformer models and applying them to this problem, as well as analyzing what advantages and disadvantages they bring.

1.2 Context of the Project

This work was started in May 2022, while the author was on Erasmus in Groningen, the Netherlands. While there, it was possible to make video calls with the tutor in Barcelona, Petia Ivanova Radeva, but it was proposed to collaborate with another tutor there, Estefania Talavera. Both tutors had a lot of experience in the field of computer vision, but the tutor there had special expertise working with first person videos. She knew that the Ego4D dataset had recently been published and it was a unique opportunity to work with it. Therefore, we decided to do this work together, applying knowledge from both fields.

Also, being there, we were given access to the research facilities of the University of Groningen, which has been a great help to us in executing the different parts of the project, as they are terribly large in space and computational cost.

1.3 Objectives of the Project

In this project, we want to analyze in depth the performance of different transformers and compare their capabilities with those of other typical neural networks. Then, using this knowledge, we propose to apply it to the Moment Queries problem and analyze its performance.

Therefore, the objectives of this project are:

- Study the Moment Queries challenge, become familiar with the database on which it is based, understand the baseline proposed by Ego4D and reproduce it.
- To achieve the necessary knowledge on which the Transformers are based.
- To train a Transformer for Image Classification and optimize its performance by understanding and varying the hyperparameters that configure the training and evaluate them. As well as comparing it to different neural networks that constitute the state of the art in this problem.
- Make a proposal of improvement over the Ego4D baseline using a Transformer, implement it and study its performance.

1.4 Project Planning

This project was carried out during the summer, autumn and winter of 2022.

The work had already been agreed before but, as some time was needed to settle in the Erasmus location, the first official meeting was at the end of May.

The month of June was committed to basic training on Transformers and how they compare to other networks. For learning, we mainly used the book *Dive into Deep Learning* [3]. After that, we also started to run the Transformer ViT on the CIFAR10 dataset[4].

During the summer months, the training continued with books, articles and videos and we worked in depth on how to run the Transformer and how it compares to other models. In general, we became familiar with the functioning of these neural networks and their hyperparameters.

In October, we started working on Ego4D. This month was committed to the understanding in depth of the dataset and the baseline and to the making of the proposal.

In November and December, the model was implemented, tested and improved.

Finally, in January, 2023 the last tests were done and the memory was finished.

1.5 Organization of the Memory

This report is organized as follows. First, we present the basic theoretical background and the more specific to this problem. Then, we explain the concrete problem we want to face and the code developed. Finally, we show and discuss the results and extract some conclusions.

More specifically, the chapters of this report include:

1. **Introduction:** a presentation of the thesis motivation and the work organization.
2. **Scientific background:** basic information about what a neural network is, how it works and different models, going into special detail on Transformers.
3. **Moment queries problem:** a detailed explanation of the Moment Queries challenge and information about the implementation of the code done, including different diagrams for better understanding.
4. **Testing:** for both the Image Classification and the Moment Queries problems. For each, first, a specification of the execution environment and how the tests have been done, metrics used, etc. Then, a display of the results obtained, analysis of what worked best and graphs to evaluate them.
5. **Discussions:** reflections on the results obtained, what changes with respect to the original baseline, ethical consequences of this development and possible future work.
6. **Conclusions:** evaluation of the objectives achieved and the work done.
7. **Bibliography:** compilation of the articles and materials used.

2 Scientific background

We will start from the most basic and build up to the challenge at hand.

2.1 Neural Network Basics

The Neural Networks (NNs) or Artificial Neural Networks (ANNs) are a technology inspired by the neural networks that make up the brain. They are based on a set of abstract nodes called artificial neurons that process information and transmit it to each other, just as a human neuron would, like the one in Figure 1.

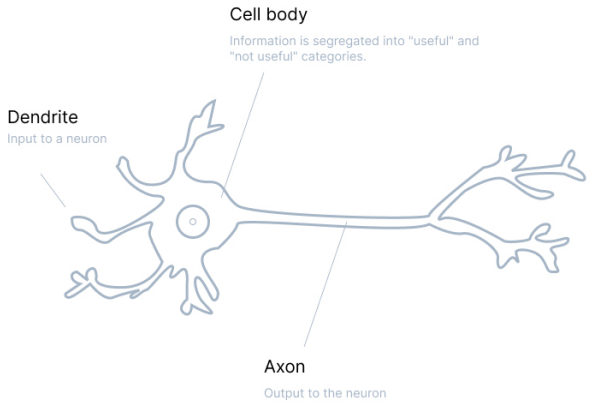

Figure 1: Human neuron.

These neurons are organized in layers that sequentially process information. We must have at least 2 layers: the first one, which receives the input, and the last one, which gives the final shape to the output. The rest of the layers in between are called **hidden layers**. There is an example on Figure 2.

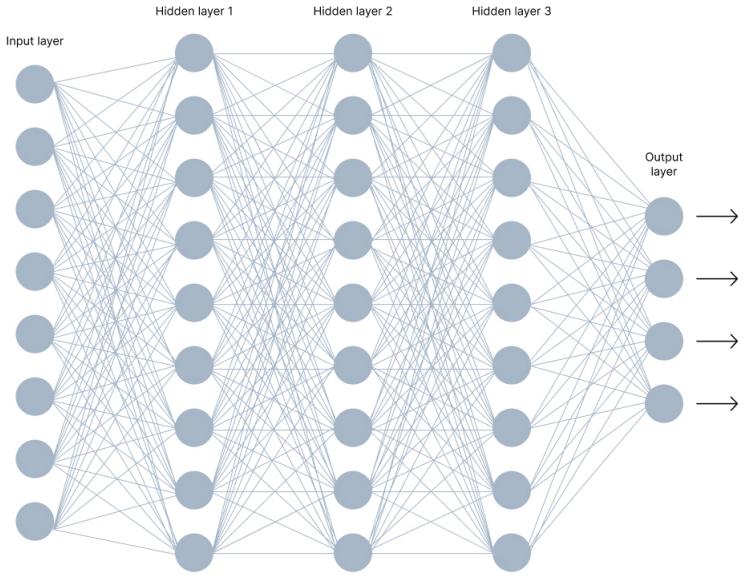

Figure 2: Neural Network appearance.

It is often said that each neuron has a **weight**, but that is not technically correct, the weights are in the edges that join two neurons. To pass from one neuron to another of the next layer, the information has to pass through this edge, which means that it will be multiplied by the weight in it. The data from each neuron goes to all the neurons of the next layer.

If a layer has n neurons and the following one m neurons, to pass from the first to the second layer we will have to calculate $n * m$ multiplications. Once in the neuron, all the weighted inputs will be added plus one extra independent value called **bias**. Normally, the operation can be represented as a matrix in the following way:

$$Z = WX + b \text{ where } X \in \mathbb{R}^n,\ W \in \mathcal{M}_{m\times n}(\mathbb{R}),\ b \in \mathbb{R}^m,\ Z \in \mathbb{R}^m. \tag{2.1}$$

Definition 2.1. *We call **parameter** to all those values of the neural network that can vary and can be trained, like the weights and the bias. They are intended to be updated automatically, not manually, except at the beginning, when they are usually initialized to random values.*

For example, in Figure 3, there are 4 parameters, the 3 weights and the bias b.

Definition 2.2. *We call **hyperparameters** the parameters that we manually modify in a neural network. They are very important to achieve good performance, since they shape the neural network and define its training. Some examples are: the number of layers, the number of neurons per layer, the learning rate and the batch size.*

So far we have seen that each neuron in the latter layer will receive n values and process them. After this, it applies an activation function to the result, like in Figure 3.

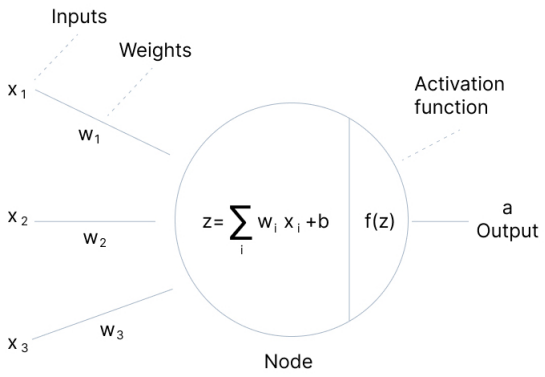

Figure 3: Neural network with three inputs and one output.

Definition 2.3. *An **activation function** is a nonlinear function $f : \mathbb{R} \to \mathbb{R}$ that "decides" whether a neuron should be activated or not, i.e. it evaluates whether the information of the neuron is relevant to the network or not.*

There are several types of activation functions. The first version proposed in what is considered the first ANN, created by Warren S. McCulloch and Walter Pitts [5], was a simple threshold.

$$f(x) = \begin{cases} 0, & \text{if } x < t \\ 1, & \text{if } x \geq t \end{cases} \tag{2.2}$$

Later, functions such as the Sigmoid were considered:

$$Sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

This function "potentiates" the value and projects it between 0 and 1, high values tend asymptotically to 1, but low values tend asymptotically to 0, becoming negligible. However, this function has some problems that do not allow the system to shine completely. Without going into too much detail, it is said that this function saturates and kills the gradient, causing it to converge very slowly or even not at all.

There are other famous activation functions such as $tanh(x)$ but the one that really revolutionized the world of machine learning was the Rectified Linear Unit, also known as ReLU.

$$Relu(x) = \max(0, x) \tag{2.4}$$

The properties of this very simple function allowed the training of much more complex models and led to a great development in the field of neural networks circa 2012.

Recently, the use of the activation function GELU (Gaussian Error Linear Unit) has also been very popular, especially in Transformers. This function can be thought of as a smoother ReLU.

$$GELU(x) = xP(X \leq x) = x\Phi(x) = x \cdot \frac{1}{2} \cdot [1 + \text{erf}(\frac{x}{\sqrt{2}})] \tag{2.5}$$

where $X \sim \mathcal{N}(0, 1)$, $\Phi(x)$ is the standard Gaussian cumulative distribution function and erf is the Gauss error function.

It is usually approximated with:

$$\approx 0.5x \left(1 + \tanh\left[\sqrt{2/\pi}\left(x + 0.044715x^3\right)\right]\right). \tag{2.6}$$

If a neural network has only the input and the output layer and nothing else in between, the result will be linear and extremely simple. In addition to having a monotonous relationship with respect to each parameter, which we do not want.

By simply adding a hidden layer in between we can solve these problems. In fact, it is proven that with a single hidden layer we can approximate any continuous function, regardless of its complexity. This fact is known as the **Universal approximation theorem**.

This model with a single hidden layer was first proposed by Frank Rosenblatt in 1958 [6] and is called a **single-layer perceptron**.

Although a single layer is already enough to approximate any continuous function, it is advisable to add depth to the network with new layers, i.e. stacking. Few layers with many neurons accumulate many parameters and a very large computational cost. On the other hand, distributing these neurons in deeper layers leads to more, but lighter matrices, obtaining the same performance with fewer parameters. Such models are called **multi-layer perceptron** or **MLPs**.

In general, the models we have seen so far are usually called Feed Forward Neural Networks.

Definition 2.4. *A **Feed Forward Neural Network (FFN)** is an artificial neural network in which the connections between nodes do not form a cycle.*

In contrast, we have the Recurrent Neural Network.

Definition 2.5. *A **Recurrent Neural Network (RNN)** is an artificial neural network that contains loops, which allow information to be stored within the network.*

These neural networks are the ones that were used in the problem of understanding and reproducing natural language before Transformers were popularized and completely displaced them. RNNs have the problem that they sometimes get stuck repeating a loop [7, 8].

Another way to classify models is depending on whether all the neurons are connected to all the others in the following layer, in which case they are called **fully connected** or **dense**, like the ones we have seen so far; or if they are not all connected with each other, in which case they are called **partially connected**, like the Convolutional Neural Networks, which are explained later in this work.

To finish this section on neural networks, we would like to point out the difference between neural networks and deep learning. Although they are often used interchangeably in conversation, "deep" in deep learning refers to the depth of the layers of a neural network. Therefore, a neural network is only considered deep learning if it has at least one input layer, one output layer and at least one hidden layer.

Deep learning models learn to extract features from raw data automatically, a process also known as feature learning. They are able to learn hidden features and relationships between attributes and use this information to predict outcomes.

2.2 Training

There are three ways to train a neural network:

- **Supervised**: the simplest of these learning paradigms. Here, we have examples of what the output has to be like, i.e. we have a set of labeled information that will be given to the model to modify its parameters in order to infer general rules that can be applied to reproduce the results on its own. It is usually applied to the regression or classification problems.
- **Unsupervised**: it learns from unlabeled information and tries to make sense of it on its own. Its main goal is to explore the underlying patterns and predict the output. Typically, it deals with associative rule mining and clustering problems.
- **Reinforcement learning**: this type of algorithms learns to react to an environment. Each action they take has a reward or penalty that makes them learn to optimize the outcome from the environment.

In any case, all these types of training need a loss function to quantify how well (or wrongly) the networks are solving the problem.

Definition 2.6. *The **loss function** is a function that measures how well the neural network models the data and generates outputs from it. In supervised learning, it compares the target and output values predicted by the neural network. When we train, our goal is to minimize the loss between the predicted and target outputs.*

An example of a fairly simple loss function would be the **Mean Squared Error (MSE)**.

Let x be the target value and y - the value predicted by the system, both n-element vectors, then the MSE formula is:

$$MSE(x, y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - y_i)^2. \tag{2.7}$$

This function measures the mean squared error, i.e. the average squared difference between the estimated values and the true values. Since it is derived from the square of the Euclidean distance, it is always positive and decreases as the error approaches zero. It is usually applied to regression models.

The most used loss function in classification problems and the one that we will use during practically all the project is the **Cross-Entropy function**, also known as Log Loss:

$$L : [0,1]^n \times \{0,1\}^n \to \mathbb{R}$$

$$L(p, y) = -\sum_{c=1}^{n} y_c \log(p_c), \tag{2.8}$$

where n is the number of classes, log is the the natural logarithm, y_c is the binary indicator (0 or 1) that indicates if class label c is the correct classification for this observation, and p_c is the probability predicted by the system of this observation being of class c.

This function satisfies the following properties:

1. $L(p, y) \geq 0,\ \forall p \in [0,1]^n, y \in \{0,1\}^n$ because of the fact that $log(x) \leq 0\ \forall x \in (0,1]$
2. $\lim_{p \to y} L(p, y) = 0$.

Now that we have a function to evaluate the results, let us go through the process by which we optimize the loss function and improve the model performance.

We call **Forward propagation** the process in which neural networks create predictions from inputs. Initially, the weight values W and the bias b are initialized to random values in each neuron. The training data goes through various layers until it reaches the output layer, where we give it its desired shape. Then, the loss function evaluates these predictions and gives a score. From this score, we will apply the backpropagation algorithm, which will update the parameters. This cycle is repeated as many times as we indicate.

The **Backpropagation** algorithm is very important, because it is in charge of understanding how the parameters affect the result and optimizing the loss function to improve the neural network's predictions. To run this algorithm we need an Optimizer.

Definition 2.7. *An **Optimizer** is an algorithm that modifies the parameters of the neural network so that the loss function is minimized.*

One of the most famous Optimizers is the **Stochastic Gradient Descend (SGD)** method. This iterative algorithm is based on finding the value of each parameter that

minimizes the loss function. To find this value, we use the gradient of the loss function:

$$x^{k+1} = x^k - \eta \cdot \left(\frac{\nabla L\left(x^k\right)}{\|\nabla L\left(x^k\right)\|} \right) \tag{2.9}$$

where k indicates the iteration, ∇ is the gradient of the loss function, η is a parameter that dictates how much we "move" in that direction, it is usually called **learning rate** and can be fixed or variable during iterations, L is the loss function and $x = (x_1, ..., x_n)$ are the parameters.

Definition 2.8. *The **gradient** of a function $f : \mathbb{R}^n \to \mathbb{R}$ is a vector formed by its partial derivatives:*

$$\nabla f(x_1, x_2, \ldots, x_n) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(x_1, x_2, \ldots, x_n) \\ \frac{\partial f}{\partial x_2}(x_1, x_2, \ldots, x_n) \\ \vdots \\ \frac{\partial f}{\partial x_n}(x_1, x_2, \ldots, x_n) \end{bmatrix}. \tag{2.10}$$

Calculating the derivative can be a very difficult or even an impossible task to do computationally, however, it can be easily approximated.

The formula for a analytical derivative of f is

$$\lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \tag{2.11}$$

But we can approximate it fairly accurately with a very small h value. Usually, $h = 1e-6$ is used, lower values are considered noise.

$$\frac{f(x+h) - f(x)}{h}. \tag{2.12}$$

With the SGD optimizer, we can modify the parameters until we reach the combination that minimizes the loss function. However, there is a problem: during training we will probably have millions of parameters and millions of target outputs, comparing them all at once can be extremely expensive computationally.

Hence the "stochastic" part of the name. In each iteration, instead of taking all the data points, we want to learn on, we will sort them randomly and apply the algorithm on one or a few at a time. This way we may not go straight to the minimum, we may go back and forth a bit, but we will end up getting there anyway and with a much lower computational cost.

Each time we evaluate on an example and update parameters, it is called a **step** and each time we have iterated through all the examples we call it an **epoch**.

In this thesis, we will also use another very famous optimizer called Adam. This optimization algorithm is an extension of the stochastic gradient descent that has recently given very good results in the field of deep learning in computer vision and natural language processing.

It was presented by Diederik Kingma and Jimmy Ba in their 2015 ICLR paper entitled *Adam: a method for stochastic optimization* [9], aiming to combine the advantages of the two most popular optimizers at the time: AdaGrad [10] and RMSProp [11].

While stochastic gradient descent maintains a single learning rate for all weight updates and it does not change during training, Adam adapts the parameter learning rates based on the average first moment (the mean) and the average of the second moments of the gradients (the non-centered variance).

2.2.1 Datasets for training

Before finishing this section, we have to define some basic concepts that will be discussed during the work.

Definition 2.9. *The **batch size** is the number of data units that each epoch will work with.*

When working with large amounts of data, as in this case, it is important to divide the information into smaller batches in order not to saturate the computation. It may be the case that the machine does not have enough memory space for so much information at once or that it is not able to process it.

Definition 2.10. ***Overfitting** refers to when a neural network is trained so much on the same training data that it fits them too much, causing it to give good results on this set, but losing generality and giving bad results on any other data.*

To avoid this phenomenon, we have several options:

- Reduce the number of parameters.
- Add Weight decay (aka L2 regularization). Penalize the magnitude of the parameters so that there is no feature with too much weight.
- Stop training, when we already have good results and before it fits the data too closely. This is known as **early stopping**.
- Add variability in the data. For example, in the problem of object detection in images, we can add images with variations in the light level or rotated.
- Add **dropout**, i.e. drop out neurons randomly during the training process. By eliminating these neurons and their connections, we force the network to have more general results.

Definition 2.11. ***Underfitting** is the opposite of overfitting. It happens when the neural network did not learn enough on the training data and, in general, performs very poorly in all predictions.*

Once we have a dataset to train our model on, we have to divide it into two or three disjoint parts:

- **Training set**: this is the data set used for the training of the neural network.

- **Validation set**: data set on which we will evaluate the training results and evaluate how the hyperparameters performed in order to find the best hyperparameters for the network.
- **Test set**: this set is not used during training, it is used for the final performance evaluation once the training is finished and we have found the optimal hyperparameters.

It is important that these sets do not overlap. We want the validation and test sets not to contain data from the training set in order to be able to assess whether overfitting is occurring.

Definition 2.12. *K-**fold cross-validation** is a neural network evaluation technique based on evaluating the results on a statistical analysis and ensuring that they are independent of the partition between training and test data.*

To perform it, we first separate a part of the dataset for testing. The rest of the dataset will be divided into k equal parts numbered from 1 to k. Then, we train the model k times. The model number i will use as validation set the set number i and the rest of the sets will be used for training. Once trained, they will be evaluated on the test set that we separated at the beginning. This way, we make sure that they have all been trained on different data and the results do not depend on the partitioning in training and test sets. If all the k performance results are good, then the model is consistent. Normally $k = 5$ is used.

2.2.2 Training Properties

When we work with neural networks, they have some properties that can help a lot during training, especially in very complex systems like the ones we deal with in this work.

Transfer Learning

Many times, training a big model from scratch with random weights can be a very demanding task, as it can take a lot of time and resources. This is why this property is very interesting.

Theorem 2.1. *It is more efficient to take an already trained neural network model, even if it was trained on a completely different dataset and adapt it to our problem than to train one from scratch.*

There are many steps in the execution of a neural network, such as extracting features from the information or establishing relationships between them. Most of these steps are shared among all models, and, no matter how different the problems are, the structure of the neural network will be similar. Therefore, we can take a network already trained on another problem, change the input and output layer to match our situation, and even change more layers if we want, and we will save a lot of time and resources. When we adapt a neural network like this, we have to do some extra training to make it consolidate to the new problem.

Definition 2.13. *We call **fine-tuning** to finish tuning an already trained neural network to a new problem and environment by further training it a bit more and adjusting the hyperparameters.*

Fine-tuning as such has not been used much in this project, since we wanted to study how the neural network is trained from scratch and how its hyperparameters influence it. However, we have done tests fine-tuning already trained models to reproduce the results that are considered state of the art in the image classification problem. For example, the tests performed with ResNet50 [12].

Regarding the Moment Queries challenge, due to the novelty of the problem and its complexity, we have not found similar models already trained and have decided to create one from scratch. However, we have taken a similar approach, since we took a transformer designed for image classification and a transformer designed for natural language processing and adapted it to a Data Augmentation problem, thinking about the possible similarities in both information processing.

Multi-task Learning

The principle of multi-task learning says that if we want to tackle several problems, it is better to do it in one network all together than using several networks. This is due to the fact that each problem will force the network to recognize new information in the data that may be useful for another problem.

For example, in this same work, the model used for the Moment Queries challenge has to detect the time instances in which an action starts and ends, classify which action it is and give it a probability. All this is done with the same model, the only part that is separated are the last layers. We can clearly realize that the original model was already thought with this in mind, since it first locates the action, then classifies it and with the information from the classification and finishes specifying the exact location.

2.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a type of feedforward networks designed to exploit the structure of matrix data, essentially images. It is therefore not surprising that they are considered state of the art in many computer vision problems. An example of the structure of a CNN is shown in Figure 4.

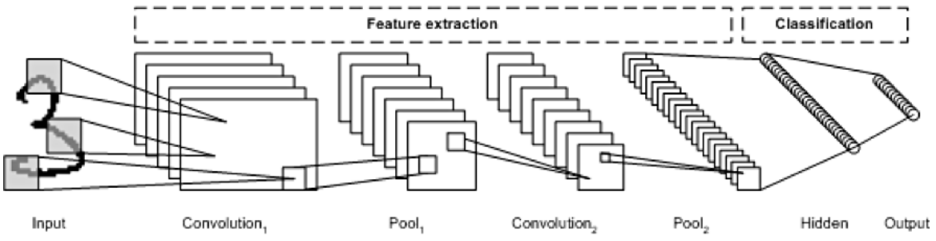

Figure 4: Example of the structure of a CNN.

These neural networks are based on three principles:

- Spatial invariance: the network must behave in the same way to the same patch, regardless of where it appears in the image.
- Locality: it should focus on local regions, uninfluenced from what is farther away.
- As we advance through the layers, we want to see the overall picture and detect feature characteristics of the whole image.

All these principles could actually be achieved with an MLP, but it would require an overwhelming amount of parameters. Convolutional networks make it affordable thanks to convolutions.

Definition 2.14. *A **convolution** is a function over two functions, $f, g : \mathbb{R}^n \to \mathbb{R}$, that measures the overlap between them, when one is flipped and shifted, i.e.*

$$(f * g)(\mathbf{x}) = \int f(\mathbf{z})g(\mathbf{x} - \mathbf{z})d\mathbf{z}. \tag{2.13}$$

And, in the discrete case:

$$(f * g)(i) = \sum_a f(a)g(i - a). \tag{2.14}$$

In 2 dimensions:

$$(f * g)(i, j) = \sum_a \sum_b f(a, b)g(i - a, j - b). \tag{2.15}$$

In fact, a convolution calculates the cross-correlation.

In practice, these convolutions are done with convolutional kernels. The kernel slides through the image and, in each position, multiplies its values by the values in the image, adding the result, Figure 5 shows an example of this calculation.

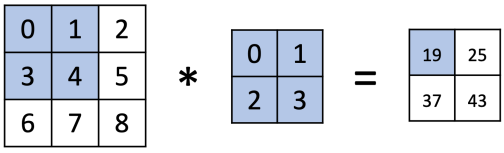

Figure 5: Example of a convolution with a 2x2 kernel. The blue values of the image are multiplied by the kernel and summed to form the first position of the output.

These convolutional filters have weight values that allow them to perform different functions such as detecting lines or edges, blurring, sharpening, etc. However, in CNNs, it is not necessary to specify the function, these weights can be trained and the model itself learns to detect what is needed.

Definition 2.15. *A **convolutional layer** cross-correlates the input and kernel and adds a scalar bias to produce an output.*

In addition, these convolutions can have other parameters:

Padding

Normally, when we apply a convolution, the kernel is smaller than the input, thus the output size is reduced, as happens in Figure 5. To avoid this, we can add an extra imaginary border to the input to avoid reducing the size. Normally, these added rows and columns are filled with zeros. If an input has n_r rows and n_c columns, the kernel k_r rows and k_c columns and we add p_r rows and p_c columns of padding, the output will have dimension:

$$(n_r - k_r + p_r + 1) \times (n_c - k_c + p_c + 1). \tag{2.16}$$

Stride

Usually, when applying a convolution, we slide the kernel through all the positions of the

image, but we can move more than one position at a time, leaving spaces in between. This way we can reduce the computational cost or downsample the data. The stride defines the number of positions we skip at each step and it can be different for the vertical and horizontal axis. In the same case as before, if we apply a vertical stride of s_r and a horizontal stride of s_c, the resulting output will have the following dimension:

$$(n_r - k_r + p_r + 1)/s_r \times (n_c - k_c + p_c + 1)/s_c. \tag{2.17}$$

We can have more than one data input channel, in such case we would have a different kernel for each channel.

We can also have a different number of output channels. For example, it is very typical to apply multiple kernels and keep each output in a different channels, which will represent a different set of features. Each output channel will have its own set of kernels for each input channel.

As we advance through the CNN's layers, we will usually want to get more general information. We need to reduce the image's size without losing significant information. To achieve this, we can use **pooling** operators, the most typical are average pooling and max pooling, which average the values or pick the greater one from a set of pixels and condense it into a single pixel.

In fact, our baseline for the Moment Queries challenge is downsampling the information to obtain more general features at each level. However, we do not use polling operators, but a 1-dimensional convolution with stride 2.

Definition 2.16. ***Deconvolution*** *is the opposite operation to convolution, it is used for signal restoration, to recover data that has been lost.*

In our code, since we need a lot of temporal precision, after reducing the feature space and obtaining the general features, we reconstruct the original information. The reconstructed data is more significant data, because it is using the information we obtained by looking at the bigger picture.

To do this, we use **transposed convolutions**. To calculate a transposed convolution, we apply the whole kernel to a single matrix position i.e. we multiply the whole kernel by the value in the pixel. Then, we add up all the resulting matrices. It will be better understood with Figure 6.

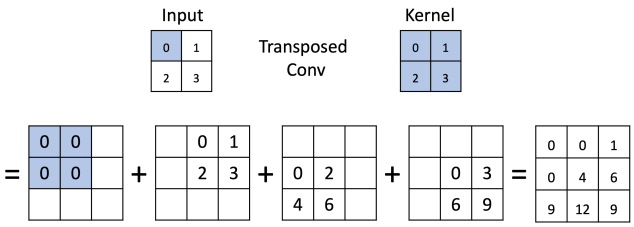

Figure 6: Example of a transposed convolution. The first box of the image is multiplied by the whole kernel, resulting in the blue 0's in the matrix of the bottom row. This is repeated for each position of the initial image and the resulting matrices are added.

The transposed convolution broadcasts input elements via the kernel, thereby producing an output that is larger than the input.

Also, in these convolutions, padding and stride work differently. Padding indicates the number of rows and columns that are removed from the final result's border. A stride of n causes to project each kernel n positions further forward, making the final result even larger.

In the same example as above, a transposed convolution with stride 2 would result in the matrix from Figure 7.

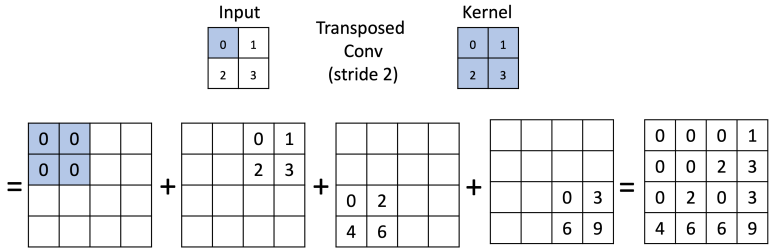

Figure 7: Example of a transposed convolution with stride 2. The first box of the input is multiplied by the whole kernel, resulting in the blue 0's in the matrix of the bottom row. The next position will not be applied right next to it but leaving a blank space in between.

2.3.1 ResNets

Since the moment, the **residual networks** or **ResNet**, developed by Kaiming He et al. [12], won the ILSVC 2015 challenge, they have been considered among the best models of CNNs.

One problem that can occur during training is the vanishing gradient or even the exploding gradient. This happens in networks with many layers when we apply back-propagation based on the gradient of the loss function. In these models, there are so many layers that the influence that each layer has on the final gradient is practically negligible, which causes the weights not to change and the training to stagnate.

The key to solve this problem are the **skip connections**, in which the output of one layer is given as input to another layer a few steps ahead. These connections are "shortcuts" to skip two or three layers. They form the residual blocks and are built using the identity function (so they do not change the information). There is an example in Figure 8.

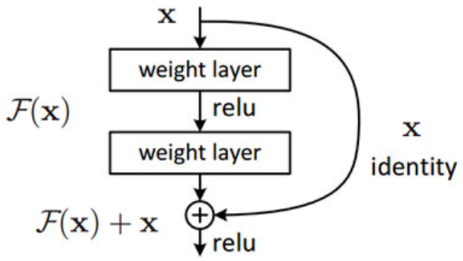

Figure 8: Diagram of a residual block.

When initializing a neural network, we normally initialize all the weights to numbers close to 0, which causes the loss of a lot of information, but this type of connections are

initialized with the identity, thus no information is lost in the early stages and we allow for much faster training. In addition, when backpropagating, the jumps between layers at different depths help to restore the gradient and make it not disappear or explode.

Moreover, adding these residual blocks does not harm the performance of the system since they are very easy to execute, more likely the other way around, they allow to hide layers that are not being useful.

In this work, we have compared the performance of Transformers to ResNet18 and ResNet50 for an image classification task. ResNet18 and ResNet50 are residual networks with 18 and 50 layers.

2.4 Graph Neural Networks

Definition 2.17. *A **graph** is an abstract type of data, consisting of a set of nodes (or vertices) and a set of edges that establish relationships between the nodes.*

A graph is a much more general way of representing information. We may think of text, which has a sequential form, as a line graph or an image as a fixed-size grid graph. But graphs can be much more varied and represent very different relationships.

Graph Neural Networks (GNNs) are designed to work with this type of data. While a CNN is based on the spatial relationships between nodes, a GNN can use other relationships, it is a generalization.

In a graph neural network, nodes collect information from neighbours because they regularly exchange information with them. This way, the graph neural network is able to learn: data is transmitted and incorporated into the features of the corresponding nodes.

The implementation of a GNN is based on three steps:

1. **Locality:** It needs a data structure that represents the data and their relationships. It will use an embedding that projects each data point in a space. The more similar data, the closer they will be projected. This can be achieved by using a computational graph.
2. **Aggregate information:** The information of each node will be aggregated with that of its nearby nodes. The way we approach this step can change, normally a simple neural network is used.
3. **Stacking multiple layers** (computation): The aggregation process will be repeated several times, improving the information of each node.

In practice, we will use the following functions:

Let v be the vector we are looking at. The embedding function X will project its features to the embedding space:

$$h_v^0 = X_v \text{ (feature vector)} , \tag{2.18}$$

where h_v^0 is the initial representation of v in the embedding space. To aggregate features, we will use the following formula:

$$h_v^k = \sigma\left(W_k \sum_{u \in N(v)} \frac{h_u^{k-1}}{\mid N(v) \mid} + B_k h_v^{k-1} \right) \text{ where } k = 1, \ldots, k-1, \tag{2.19}$$

where σ is the activation function, k indicates the layer where we are, W_k are the weights in the layer k, B_k are the bias in the layer k and $N(v)$ indicates the neighbours of v.

There are two parts in this equation:

$$W_k \sum_{u \in N(v)} \frac{h_u^{k-1}}{\mid N(v) \mid},$$

is the average of all the neighbouring nodes of v.

$$B_k h_v^{k-1}$$

is the previous layer embedding of node v multiplied with a bias B_k.

Definition 2.18. ***Graph Convolutional Networks (GCNs)*** *are Graph Neural Networks that organize information in a way that allows to operate with convolutions.*

2.5 Transformers

Next, we will explain how the coveted Transformers work. First, we will explain their main algorithm: the attention; then, other mechanisms necessary for their operation, namely the embedding and positional encoding. Finally, we will review the structure of the Transformers for both text and image and some renowned versions of them.

2.5.1 Attention

The main component of the Transformers is a mechanism called **attention**. Its origin comes from problems where there was a very long sequence of data, which was impossible to store in memory. There was a need for a way to reference this past information without having to evaluate it all at once.

The inspiration of this mechanism comes from the databases. A database consists of a set of pairs (key, value) accessible with a query. The database checks which keys match the query and returns their values.

Let us consider $\mathcal{D}$ is a database, $\mathcal{D} = \{(k_1, v_1), ..., (k_m, v_m)\}$, with m tuples of keys and values. Then, we can define the attention over $\mathcal{D}$ as:

$$\text{Attention}(q, \mathcal{D}) \stackrel{\text{def}}{=} \sum_{i=1}^{m} \alpha\left(q, k_i\right) \mathbf{v}_i, \tag{2.20}$$

where $\alpha\left(q, k_i\right) \in \mathbb{R} (i = 1, \ldots, m)$ are scalar attention weights.

The name attention comes from the fact that we pay more attention to the pairs that have obtained a higher α value.

For this to be meaningful, we want the α weights to define a probability, that is, to satisfy the following conditions:

$$0 \leq \alpha(q, k_i) \leq 1, \ \forall i \text{ and } \sum_i \alpha(q, k_i) = 1.$$

A very easy way to accomplish this is to apply the **Softmax** function.

$$Softmax(\alpha(q, k_i)) = \frac{e^{\alpha(q,k_i)}}{\sum_j e^{\alpha(q,k_j)}} \quad for\ i = 1, 2, \ldots, M. \tag{2.21}$$

There are different ways to define how the weights α are obtained, but the most typical and successful is the one below.

Let us suppose that the information for the query, key and value comes in vectors of the form $x \in \mathbb{R}^{d_x}$, $y \in \mathbb{R}^{d_y}$, $z \in \mathbb{R}^{d_z}$, respectively. They can be grouped into the matrices $X \in \mathbb{R}^{n_x \times d_x}$, $Y \in \mathbb{R}^{n_y \times d_y}$, $Z \in \mathbb{R}^{n_z \times d_z}$.

The computation of the attention could be done vector by vector, but in practice it is always calculated using the whole matrix at once, because it is more efficient.

First, the data is projected into the attention's dimension, to be able to compare keys and queries. It is done with the following matrices: $W^Q \in \mathbb{R}^{d_x \times d_k}$, $W^K \in \mathbb{R}^{d_y \times d_k}$, $W^V \in \mathbb{R}^{d_z \times d_v}$.

These are weight matrices, which can be trained. This matrices also enhance any particular part of the data where the attention should focus.

Next, the Query, Key and Value matrices are obtained with the matrix product between the previous ones such that:

- Query matrix: $Q = X \times W^Q$, $Q \in \mathbb{R}^{d_k}$,
- Key matrix: $K = Y \times W^K$, $K \in \mathbb{R}^{d_k}$,
- Value matrix: $V = Z \times W^V$, $V \in \mathbb{R}^{d_v}$.

From here, to get the attention values between a vector q of Q and a vector k of K, we would use the dot product. In matrix form, it can be expressed as $Q \times K^T$.

Then, it is divided by the square root of the dimension of the key vectors (this leads to having more stable gradients), we apply the softmax function and it is multiplied by V. Thus, the complete attention formula is:

$$Softmax(\frac{Q \times K^T}{\sqrt{d_k}})V. \tag{2.22}$$

It is important to note that attention values provide a lot of information. For example, in the analysis of a sentence using attention between the words, if there is a pronoun replacing a noun, the attention value between the two words will be very high.

In practice, we will normally use the same data as key and value.

Definition 2.19. *When the attention uses the same data for query, key and value, it is called* ***self-attention****.*

Nevertheless, we may want our model to combine knowledge of different behaviors. In order to detect more patterns, it is possible to repeat the process with different projections. The whole process is equal except for the projection matrices.

Each **head** runs a different version of the attention. At the end, all the outputs will be condensed together using a MLP. This method is called **Multi-Head Attention**.

2.5.2 Embedding and Positional Encoding

One problem so far is that the input data of the system can have any format, even be variable between different elements. For example, in natural language, each word has a

different length. In order to be able to process them, they have to be standardized. For that, an embedding is used.

Definition 2.20. *An **embedding** is a function that projects each piece of information to a space of the same dimension. This function is usually trainable.*

In the case of a text sequence, this embedding can be based on the number of words in the vocabulary.

However, the attention mechanism shown until here does not take into account the position of each data with respect to the others and it is very important information. For example, in natural language the order of the words can change the meaning of the whole sentence. In an image, it is also important to know where each patch was. This is why we want to return this information to the data.

Definition 2.21. *The **positional encoding** is a function that assigns a different value to each element depending on its location. This value will be added to the embedding (so they must have the same shape) to inject positional information.*

In sequential data structures such as a sentence or the frames of a video, one could take a naive approach and number the elements. In binary code, the values would be 000, 001, 010, etc. Clearly, the frequency of each digit is less than the previous one. It is possible to use a similar representation using sinus and cosinus and decreasing their frequency. It is also more space efficient, because we are dealing with float data, as is the embedding. The sinus and cosinus are usually used interspersed, either in one or two dimensions.

The most typical implementation is:

$$\begin{cases} p_{i,2j} = sin(\frac{i}{1000^{2j/d}}) \\ p_{i,2j+1} = cos(\frac{i}{1000^{2j/d}}), \end{cases} \tag{2.23}$$

where i indicates the number of the sequence, j the number of the token within the sequence and d is the number of tokens within the sequence. Figure 9 is the heatmap of this function.

Still, it is possible to not use a fixed positional encoding and learn it.

2.5.3 The Original Transformer Architecture

It is time to introduce the whole architecture with which the Transformers were originally proposed in the paper *Attention Is All You Need* [1]. Mind that this architecture was designed for the processing of natural language. Its diagram is presented in Figure 10.

The model is divided in two main parts: Encoder and Decoder. Both of them have the same first step: embedding the input and adding a positional encoding i.e. giving the data some form to be able to work with it. From this point on, they differ.

2.5.3.1 Encoder

The goal of the encoder is to process the input sequence and look for relationships that can help us understand it better.

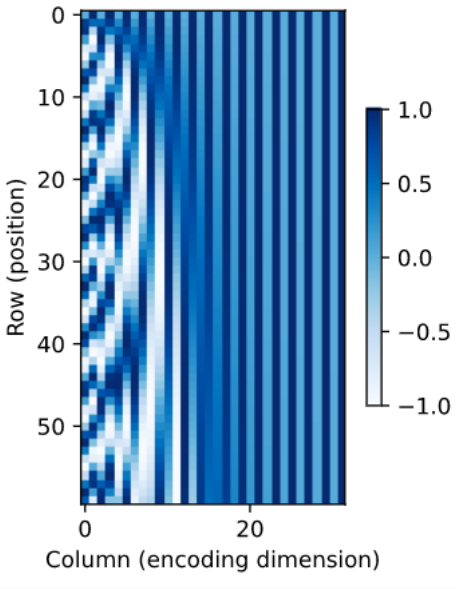

Figure 9: Heatmap of the positional encoding with sinus and cosinus.

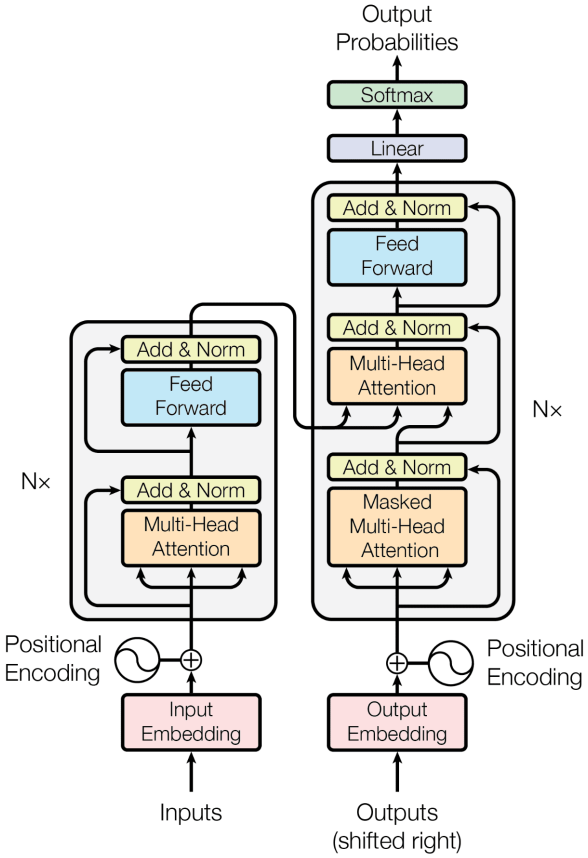

Figure 10: The Transformer Architecture. The Encoder is shown on the left and the Decoder on the right, each with its corresponding layers.

Both, the Encoder and the Decoder are formed by blocks that are repeated a number of times, both the same number of times.

The Encoder Block is made up of four layers. The first is a self-attention layer, where the data is analysed in order to find relationships or behaviors between the tokens of the sequence. Its output does not replace the previous data, but they are added and then normalized. The next step is a Feed-Forward Network in the form of an MLP. Finally, the result is added and normalized again.

Note that the add and norm layers are residual layers like the ones seen in ResNets[12], with the same benefits, namely faster training, limitation of information loss, etc.

Moreover, after each layer the data is normalized, which makes it less dependent on scale and batch size.

Note as well that the format of the Bock Encoder output is the same as the input, it has not been changed at any time. This allows to sum them properly and, at the end, repeat the whole process with the same data structure.

2.5.3.2 Decoder

The goal of the decoder is to add meaningful words to the sentence one by one using the information collected in the Encoder. Each execution of the Decoder will add one more token to the sequence. Therefore, its input is the output of the previous execution, that is the previous sequence with one more token. It can be understood as: each time it is executed, it checks what has been written so far and writes the next word.

The first layer of the Decoder is a self-attention layer, but with the difference that it can only attend to previous positions in the output sequence. It can be done by masking future positions i.e. setting their attention values to zero.

The next attention layer is also called the "Encoder-Decoder Attention" layer. It computes the attention using as queries the results of the previous layer and as keys and values the output of the encoder stack. Finally, we have an FFN block as before.

The output of the whole Decoder algorithm is a list with a score for each word in the vocabulary. We pass this score through softmax to convert it into probabilities and return the most probable word.

Nowadays, one of the best transformers for Natural Language processing is BERT (Bidirectional Encoder Representations from Transformers) [13], shown in Figure 11. This model has more Encoder layers, but no Decoder block, it computes it all at once. It was proposed as inspiration for our model but at the end it was discarded.

2.5.4 Transformers for Vision

The structure of the Transformer can also be used for computer vision problems, but in this case we usually take a slightly different approach. In fact, it was the BERT model just mentioned that inspired the application of the Transformer to this field.

In the following we will present the **Vision Transformer (ViT)**, it is the model presented by Dosovitskiy et al. in the paper *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale* [14] to solve image classification problems. This approach, shown in Figure 12 is the basic structure, the other Transformers for vision add layers to

this one.

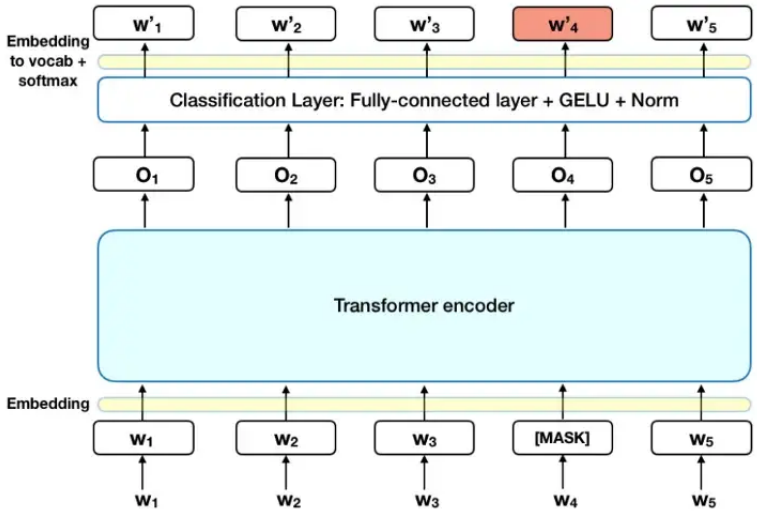

Figure 11: Structure of the BERT Transformer. The words go through an embedding, then an encoder block of a Transformer and a fully connected layer plus embedding determine the resulting words.

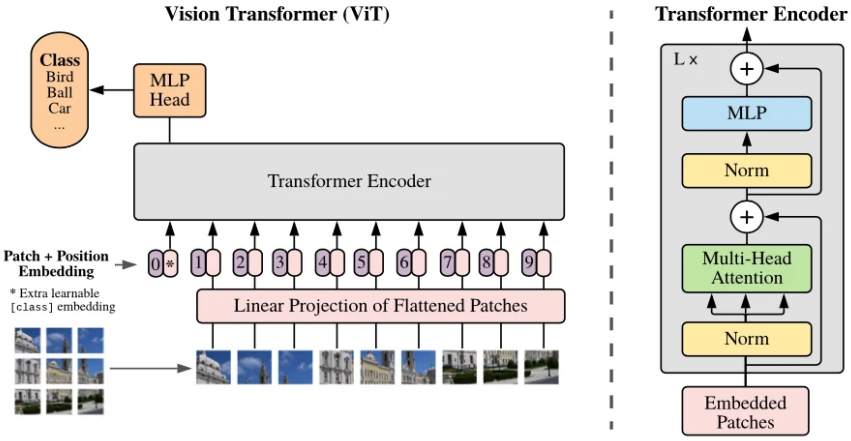

Figure 12: The ViT Architecture.

This model is based on the Encoder block of a Transformer. Due to the fact that it is not needed to rewrite a sentence, the Decoder is no needed, it will only encode the data.

The obvious question now is: an image is a single piece of information, how do we extract the different data tokens to analyze the attention? The answer is **patch embedding**. As shown in the lower part of Figure 12, the image is split into different patches,

i.e. it is divided into equal parts. These patches can be obtained using a convolution with a kernel and stride equal to the patch size. Then, they are flattened to a vector. This way the image patches have the same structure as tokens of a sequence.

Apart from the flattened patches, a new [class] token is added. As the attention mechanisms relate the [class] token to all the patches, this token will store the information needed to extract the label for the image. Therefore, each image is represented by n patches +1. Note that, in Figure 12, there are 10 tokens extracted from the image before entering the encoder and the first one is the [class] token. Then, the positional encoding is applied to them.

From here on, the encoder block is exactly the same, with the only difference that each normalization is done before each layer, not after, to reduce the dependency on the image configuration.

The last step after applying all the encoding blocks is to apply a feed-forward network called head (it can be input and output layer only or more layers), which transforms the information stored in [class] into a probability for each of the possible classes.

In general, Transformers show better scalability than CNNs. They are larger models and need very large training datasets but, in these conditiones, they outperform even the best CNNs.

2.5.4.1 SWIN Transformer

One particular model that is nowadays considered to be the best model to cope with vision tasks is the **SWIN Transformer**, researched by the Microsoft research team in Asia and presented in the paper *Swin Transformer: Hierarchical Vision Transformer using Shifted Windows* [15]. This model greatly reduces the computational cost of the ViT by using a method called Shifted Window, shown if Figure 13.

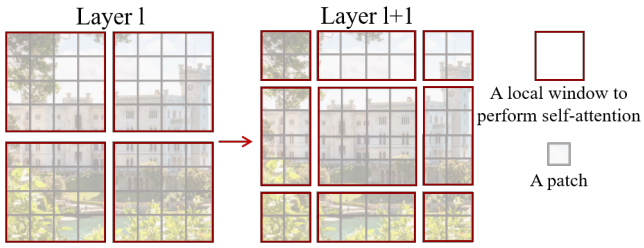

Figure 13: Graphical representation of the Shifted Window method. The first layer applies local attention in the red windows, then, the window is shifted by as many positions as half the size of the window. The second layer will apply local attention in the new windows.

ViT has a problem in that it has quadratic computational complexity cost due to the fact that the self-attention is computed globally. As the size of the images increases, the computation time increases quadratically.

The Swin model faces the patch embedding differently, instead of taking patches all in the same shape and size, as ViT did. It starts with very small patches and applies local self-attention, i.e. it divides the image into zones based on a "window" and, in each zone, compares the patches with each other, but not with the ones outside. Then, it shifts the window, moving it forward by half the size of the window, so that all zones have new

patches, and recalculates the local attention in these conditions. After the calculations, it groups the small patches into larger ones and repeats the process.

Let n be the total number of patches and m the number of patches per window. With ViT, the attention related all patches with each other, thus, the complexity was n^2. With Swin, it is done only in local zones, each patch is only related to as many as there are in its zone, twice. Thus, the complexity is $m * 2 * n$, i.e. linear with respect to the number of patches.

3 The Moment Queries Challenge

The Moment Queries (MQ) challenge is about trying to solve the problem "Find all the times I did X in the video"; that is, given an egocentric video and an action belonging to a list of possible actions, return a list with timestamps indicating each time the action has been done. This task is called activity detection. To summarize:

Input:	**Output:**
- egocentric video: v - category of the requested action: c	All the instances in which the action is done: $\Phi_c = \{\phi_n = (t_{n,s}, t_{n,e}, s_n)\}_{n=1}^{N}$

where Φ_c is a list of tuples with the timestamps of the start: $t_{n,s}$ and end: $t_{n,e}$ of each instance of the action and the score given to this prediction s_n is its probability of veracity.

This challenge was proposed by Ego4D alongside with the publication of the dataset [16]. Until now, it would have been practically unthinkable to propose a challenge like this due to the lack of data to train the models, but Ego4D gave the perfect dataset for it, as well as open source baselines [17].

3.1 Baseline and State of the art

Besides proposing the Moment Queries challenge and providing a dataset specifically dedicated to it, Ego4D also proposes the baseline of a model to solve it [18], shown in Figure 14 and explained in the following. Our first goal will be to understand it and then, if possible, to improve it.

As already mentioned, the code receives as input a video and a query with an action category. When executed, the system will search for all the actions it detects in the video and will return only those classified as the requested action.

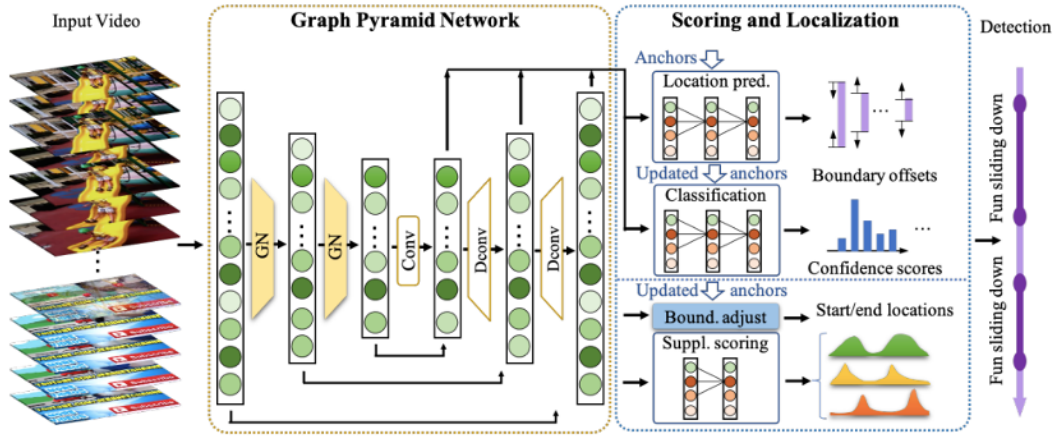

Figure 14: Diagram of the model proposed by Ego4D

The model is based on a method called *Video Self-Stitching Graph Network (VSGN)* [19], shown in Figure 15, designed to detect actions in third-person videos. However,

the two models have some small variations. The differences are due to the fact that the original model was especially focused on improving the detection of short actions. It is a well-known fact that one of the most difficult tasks in temporal action detection is the detection of short actions. These actions last only a few frames and, due to the amount of downsampling, their data is modified and reduced until it is finally lost. Both models will be explained in detail below and the differences will be analyzed.

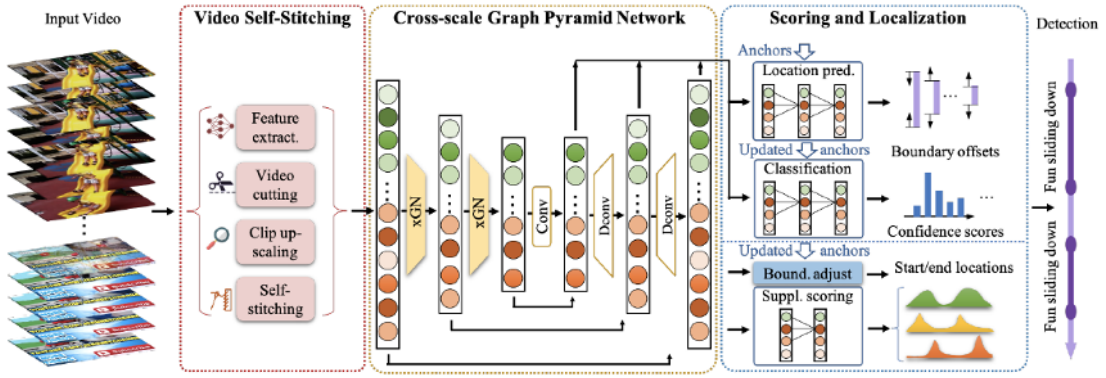

Figure 15: VSGN [19] model in which the Ego4D code is inspired.

3.1.1 Feature extraction

The first step is dividing the videos into small snippets, of at most 8 minutes. Then, the features will be extracted from the snippets. This can be achieved using different models, even using an MViT [20] or BERT [13, 21]. However, Ego4D already offers datasets with the features of the videos already extracted in different ways, and it is advisable to use them. For this analysis, we will use the pre-extracted ones using SlowFast [22].

SlowFast features are made up of 2304 feature vectors and each one has approximately 928 values, one for each frame. But not all this information is used. The first layer of the model is an embedding that reduces the dimension of the features from 2304 to *bb_hidden_dim*, by default this value will be 256.

3.1.2 Video Self-Stitching (VSS)

This part of the original system was removed in the Ego4D baseline. It is shown in the diagram in Figure 16. In this block, the features of each clip are projected to a larger scale using linear interpolation. Thus, having more information to represent the short actions it becomes harder to lose them. However, with this method a lot location information between the frames is lost. On one hand, the original video has the complete information. On the other, the enlarged one makes it easier to detect actions. The conclusion is to use them both together, taking the best of both methods.

Therefore, the original video and the extended version are concatenated leaving a space in the middle filled with black frames, so that the system does not interpret them as a single video.

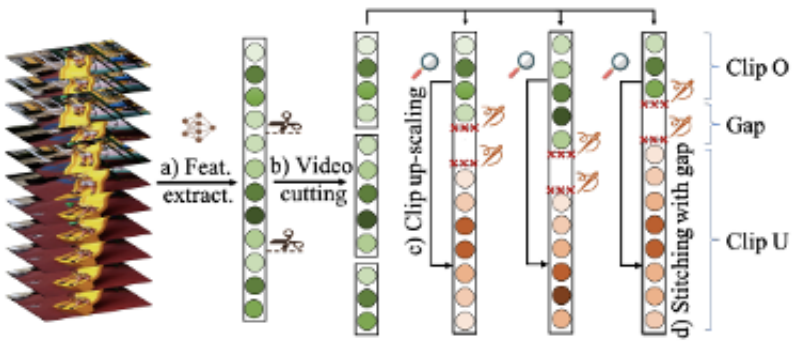

Figure 16: Diagram of the Video Self-Stitching module.

3.1.3 Graph Pyramid Network (GPN)

Once we have the features, we will pass them through a Graph Pyramid Network (GPN) that will progressively modify them. This structure is shown in Figure 17.

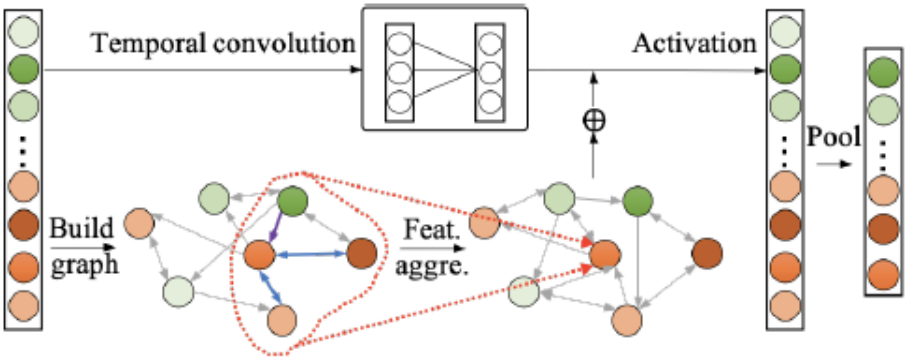

Figure 17: Diagram of the xGPN module with both its branches: the convolutional and the graph one.

A pyramid-shaped network is a network formed by a sequence of models, where between each model, we downsample or upsample the data. In this case, first it is downsampled, then, upsampled.

In the original one, they used an xGPN, i.e. Cross-Scale Graph Pyramid Network [19]. It is composed of two branches: a temporal branch and a graph-based branch. The first one combines the information by making temporal convolutions, therefore combining features close in time.

The second branch is more complex. First, it creates a graph where each node is a feature and the edges are of two types: cross-scale - they relate the features of a video with those of its extended version, or free - they relate similar features. To measure the similarity between two nodes, they use their negative mean square error and a threshold. Based on these relationships, an MLP aggregates the information and defines the resulting values of the features.

However, in the implementation proposed by Ego4D, since there are no video exten-

sions, it does not make sense to use cross-scale relationships, only the free edges are used.

3.1.4 Encoder-Decoder Structure

As shown in Figure 14, the process is composed of different layers forming an encoder and a decoder.

Each layer of the Encoder connects the information iteratively using GNs (or xGNs) and convolutions, as we have just seen. After each layer, the data goes through an one-dimensional convolution with stride 2, downsampling it by half along the time axis, the effect is like reducing the number of frames of the features. After each level, the encoder stores the obtained features in a list. They are also used as input for the next level.

Subsequently, the decoder will take this information to reconstruct the time axis of the features. There are as many layers to encode as to decode and each decoder layer receives the output of the previous layer and the direct output of the corresponding encoder layer. In fact, the decoder's system is very simple. It takes the most simplified feature vectors, enlarges them using a transposed convolution with stride 2; at the same time, it applies a convolutional layer to the features of the previous encoder layer, and adds them together. Then, they go through a convolution layer plus ReLU. Recall that they were downsampled with stride 2 convolutions. So, to undo it, they now pass through transposed convolutions with stride 2. The following decoder block will take the output of this one and the previous output from the encoder.

3.1.5 Evaluation and localization modules

Figure 18 represents the final layers of the model. Here, all the information collected with the encoder and decoder is used to make predictions of the location and class and, subsequently, improve them with a boundary adjustment and calculate a prediction score.

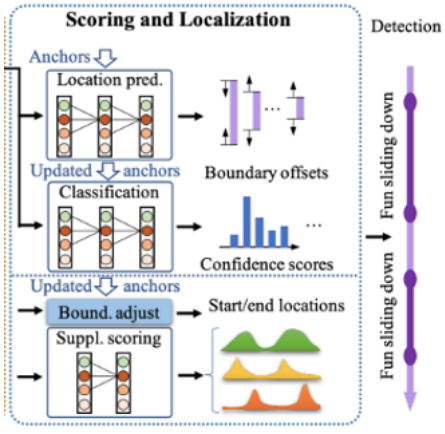

Figure 18: Evaluation and localization modules

The time localization relies on a system of anchors. First, the anchors are distributed uniformly along the frames. Then, the module M_{loc} predicts when each action starts and ends and updates the anchors. From this result, the module M_{cls} determines the

class of the detected actions. Both of them are composed by 4 blocks of convolution, normalization, ReLU and another convolution.

To improve the bounds of the prediction given by M_{loc}, we will use the module M_{adj}. For each anchor, it takes 3 samples around the start and end, concatenates the feature vectors and applies convolution - ReLU - convolution to predict an offset value. The anchor will be updated by adding the offset value to the start and end values. Finally, there is the module M_{scr}, which is also in the form of convolution - ReLU - convolution and gives a score to each one of the above predictions: type of action, start time and end time. The total score will be the multiplication of these 3.

3.2 Our Proposal for Improved Memory Retrueval

We will focus on the Graph Pyramid Network block, which extracts the data and processes it. Since it does not make predictions, we are facing a Data Augmentation problem.

After an in depth study, we saw the similarities the model had with a Transformer. They share the same Encoder - Decoder structure and they treat the information in a similar manner. Recall that a Transformer usually has the structure shown in Figure 10.

The attention mechanism is clearly similar to the way the GPN links different nodes. This block is based on relating information in two manners: in temporal proximity and in similarity. For the first, they use a CNN and for the second - a GNN. This system is very bundled and could be reimplemented in a simple Transformer. The attention mechanisms are capable of analysing the data based on similarities as done in the graph and build upon them. In addition, thanks to the positional encoding, we preserve the temporal information of the features to recreate space proximity. With the same transformer, we can consider both relationships and interpret them to refine the data.

In addition, the Transformer allows a much deeper analysis. We can have one head that focuses on the temporal closeness and another one on the similarities between the features. Moreover, we will have several other heads that can focus on other types of relationships that we had not considered. This is the grace of the Transformer, we give it the tools and the data and it will be able to understand where it should pay attention to analyze what is happening and be able to represent it.

3.3 Initial proposal for improvement

For the first modification, we thought to do the most obvious thing: changing the GN and convolutions for a Transformer. As one of the Transformers that gave best results in the literature to date is the ViT, we decided to base the model on it. However, while adapting the code, we modified so much of its structure that it became quite unrecognizable. That is why we called it ViT+. Its structure is shown in Figure 19.

Recall the structure of the ViT in Figure 12. We have used the block called Transformer Encoder for each level of the pyramid Encoder.

We eliminated the embedding patch, since we do not work with images, and the token [class], the features will be directly updated.

As embedding, it uses the same as the Ego4D baseline: a convolution that projects the features. For positional encoding, it uses a learnable one.

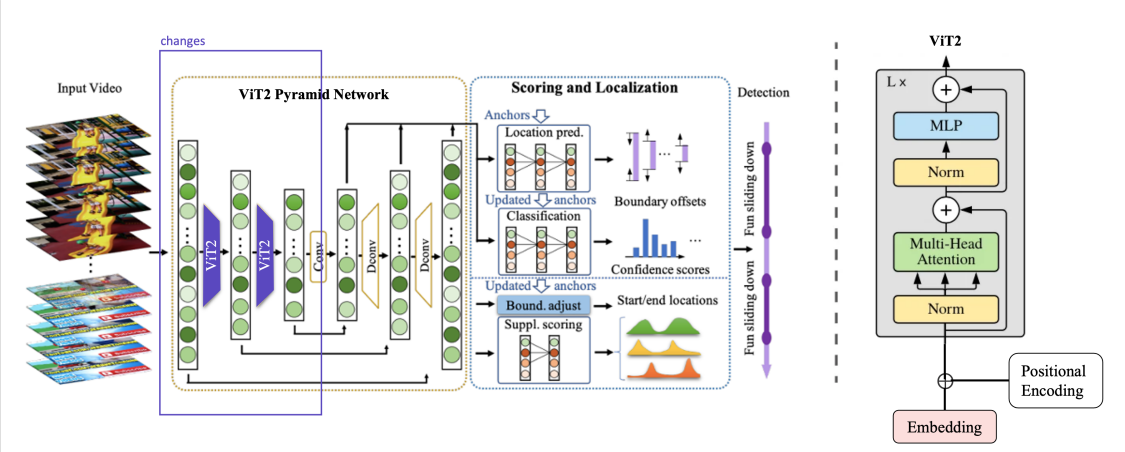

Figure 19: Diagram of the baseline with ViT+. The changes are marked in purple. The ViT+ structure shown at the right.

The ViT encoder block has been kept the same. It is important to point out that before applying attention, we have to transpose the matrix. The original model worked with features in the form of: the first dimension indicates the batch, the second the feature and the third the time axis. When applying the attention, it should compare feature vectors, so the matrix needs to be transposed. Later, the transposition will be reversed so that the features have the expected shape for the subsequent layers.

For downsampling, it uses a convolution with stride 2, as the original baseline used.

All residual connections between blocks remain in place.

The activation functions have been changed to GELU, due to the fact that many studies show that they tend to work better with Transformers [23, 24].

3.4 Our final proposal: ReMoT

After applying the first proposal and learning more about how the system worked, we decided to propose a few more modifications. The first approach had been very conservative, but now we wanted to go all out, gathering all the knowledge we had acquired to try everything that could improve it.

This time, we modified the entire pyramid block. Since the data is sequential, we took inspiration from Transformers for text processing. The model is called ReMoT from Retriveal Moments Transformer. Its structure is shown in Figure 20.

The embedding is the same: a convolutional network. The positional encoding is a typical text encoding using sinus and cosinus, since the information is also sequential. Later we saw that, because of the way we calculate the attention, this encoding does not influence the results very much, so it can be removed.

We have also decided to change the attention mechanism, inspired by the SWIN windows [15]. As explained in the scientific background, SWIN computes local self-attention by windows and then merges the data into larger windows. We have done something similar: we compute attention per window using a mask that does not change size. We do not merge the patches manually, but the downsampling process already does it. At the beginning of the execution, the mask is small enough to calculate the attention of nearby

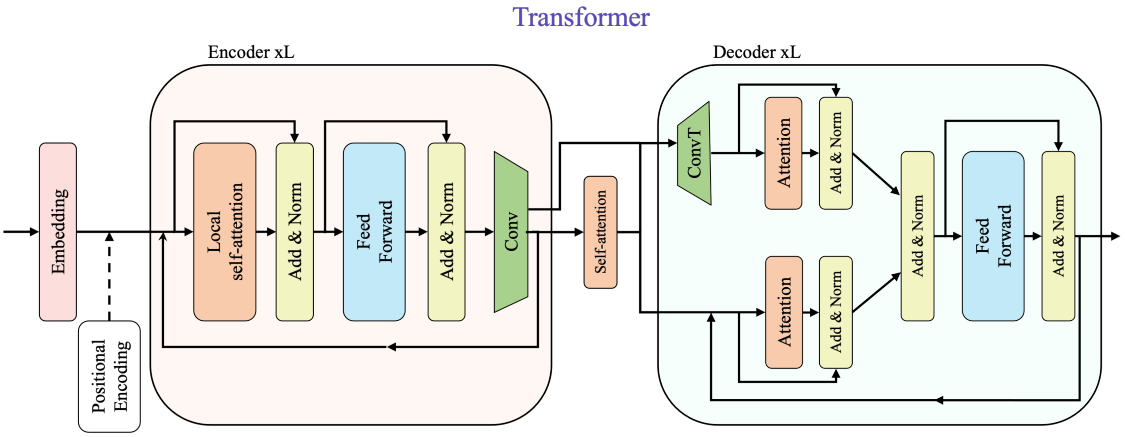

Figure 20: Diagram of our second proposal where we replace the entire GPN with the ReMoT Transformer.

features, making a very local study of the frames. As we advance through the levels and the information is more reduced, the local self-attention uses the same mask, but now it studies more and more general features. With this change, the complexity of the system is reduced to linear depending on the length of the video.

The encoder is formed by several encoder blocks, each one of them applies the local self-attention we just mentioned and a FFN and downsamples the data with a stride 2 convolution. The data still needs to be transposed for subsequent predictive blocks, it would be nice to change them to avoid this task as future lines, but in this thesis we decided to focus only on the GPN part.

Between the last level of the encoder and the first level of the decoder, we apply self-attention on the final output of the encoder and add it.

The decoder has been completely reconstructed. We have taken inspiration from the decoder for text, but in this case it is not rewriting a sentence, it is reconstructing the feature vectors on the time axis, so the functioning is very different. In each block of the decoder, it first upsamples the features received from the previous level with a stride 2 transposed convolution. Then, it calculates the attention taking as queries the upsampled features and as key and value those of the same level of the encoder and the other way around. Finally, we sum and normalize results in residual blocks.

Between each operation, there is a sum plus normalization that acts as a residual block.

The attention functions used are GELUs, we have checked that they work slightly better than ReLU.

When everything has been executed, the features obtained by the encoder and the decoder are passed to the next blocks to make predictions.

The model of the final proposal has been divided into 2 versions: Small and Large. The reason for this is that both runs have given very good results. The Large gets better metrics, but has many more parameters and is more difficult to train. Therefore, we have decided to show both. The difference between the two models is the dimension in which they represent the data, we will go into more detail in the implementation section.

4 Validation

In the following section, first we discuss the experimental setup for testings of transformers for image classification and Moment Queries, followed, in the next section, by discussing the validation of our proposals. The first part details all the experiments that were done experimenting with Transformers in order to have the knowledge to face the Moment Queries problem. Then, the results are detailed.

4.1 Image Classification Experimental Setup

The testing on Transformers applied to image classification has been focused on implementing and training a ViT from scratch. We have experimented as much as possible with it to get the best performance. Apart from this, we have also trained SWIN, ResNet18 and ResNet50 models and we have fine-tuned some of these models using already trained ones, to compare the results.

4.1.1 Dataset

The image classification experiments were performed on the CIFAR10 dataset. CIFAR10 is a dataset typically used for computer vision testing. It consists of 60,000 images divided into 50,000 for training and 10,000 for testing. The images measure 32x32x3 in RGB space. There are 10 classes and each class has 6,000 images, no image belongs to two classes at the same time. The possible classes are: airplane, automobile, bird, cat, deer, dog, frog, horse, ship or truck.

In addition, we have applied some data augmentation on the dataset. This step improves the results greatly. On the train set, we enlarge the images to 40 pixels and crop them randomly to 32. They are also randomly flipped horizontally and normalized. The test set is only normalized.

4.1.2 Validation metrics

We have used the following measures as evaluation metrics:

Accuracy: It is the percentage of predictions that are correct i.e. the ratio between the number of correct predictions and the total number of predictions.

Let tp be the number of true positive predictions, fp false positive, tn true negative, fn false negative and T the total number of predictions, then the formula for the accuracy is:

$$acc = \frac{tp}{T}. \tag{4.1}$$

Recall: It is the ability of a classifier to find all positive instances. It is defined as the ratio of true positives to the sum of true positives and false negatives. Its formula is:

$$rec = \frac{tp}{tp + fn}. \tag{4.2}$$

Precision: It is the percentage of predictions that are correct i.e. the ratio of true positives to the sum of all positives.

$$prec = \frac{tp}{tp + fp}. \tag{4.3}$$

F1-score: It is a weighted harmonic mean of precision and recall such that the best score is 1.0 and the worst is 0.0. It is calculated using the following formula:

$$\text{F1-score} = \frac{2 \cdot prec \cdot rec}{prec + rec}. \tag{4.4}$$

The python function we created to measure these metrics counts each of the tp, fp, tn and fn values for each of the classes and calculates the above metrics for each class. This function also calculates the macro average (unweighted) and a weighted average, based on the number of occurrences found from each class, of all the above metrics. In practice, as CIFAR10 is a very balanced dataset, the values were all too similar and did not provide relevant information. Therefore, we have ended up evaluating only the averaged values of the metrics among all the classes.

The run time and number of epochs were taken into account, but were not used to evaluate the models as some were pre-trained and others were not.

4.1.3 Implementation details

ViT has been implemented from scratch using pytorch and based on the theoretical basis of our reference book [3]. Some modifications have been made, but in general it is still the same structure as the one explained before. We tested ViT small, large and huge from [14], but this dataset is relatively simple and there is no much difference. For its performance study, we have used k-fold with k=5 and the wandb library [25] to visualize the progression of the training.

The SWIN model has also been implemented manually, but we have not done any modifications to the initial design. For ResNet18 and ResNet50 and the pre-trained versions of ViT and SWIN, we tested the ones from the pytorch [26] library or from Hugging Face [27], both trained on ImageNet [28].

4.1.4 Peregrine Server Specifications

All tests have been carried out in the same environment on the Peregrine server [29], provided by the University of Groningen. For processing, we used one Nvidia V100 GPU with 12 nodes and requested for 10.000 MB of memory, which is more than enough.

Due to how the server works, one sends an instruction and it will be executed when the nodes are available. The output of the function is saved in an .out file, but it contains only the *prints*, it cannot show images or graphics. For that, we used the external tool wandb.

4.2 Image Classification Results with Transformers

The best results obtained from our tests are presented in Table 1.

Architecture	accuracy	precision	recall	f1
ViT Base	74.29	74.61	74.28	73.81
ViT Small	76.09	76.37	76.09	75.95
ViT Large	77.07	77.20	77.06	76.76
ViT Huge	76.53	76.57	76.52	76.18
SWIN	**91.47**	**91.48**	**91.47**	**91.48**

Table 1: Best results obtained with different architectures of ViT and SWIN over CIFAR10.

The results are quite good for having trained from scratch. The ViTs have quite good and similar results, close to 80% accuracy, probably due to the fact that image classification on CIFAR10 is relatively easy, but in more complex problems the difference is more noticeable. The SWIN results are much more impressive, with more than 90% being trained from scratch, demonstrating its greater potential.

4.2.1 Hyperparameters

For all ViT models, we used the SGD optimizer, as there was no noticeable difference with the others, with a learning rate of 0.1 and for 100 epochs. For each of the sizes, we have used different hyperparameters as shown in Table 2, originally defined in the original paper [14]. When a hyperparameter increases, the others must increase in turn to give the maximum potential.

Ar.	Optim.	lr	Epochs	#hiddens	#mlp_h	#heads	#blks
ViT base	SGD	0.1	100	512	2048	8	4
ViT Small	SGD	0.1	100	768	3072	12	5
ViT Large	SGD	0.1	100	1024	4096	16	5
ViT Huge	SGD	0.1	100	1280	5120	16	5

Table 2: Hyperparameters used for every ViT size. #hiddens is the embedding dimension, #mlp_h is the number of neurons in the hidden layer of the MLPs.

For the SWIN, we used AdamW with learning rate 0.0001 and 100 epochs. The patches initially have a size of 2 and the windows of 4. The number of heads is variable and the hidden dimension is 4 times the dimension of the flattened patch.

4.2.2 Comparison to the state of the art

The image classification problem is considered to be solved since we have achieved models with an error of less than human error. The best result obtained on CIFAR10 is 99.5% accuracy [1, 30] with a ViT Huge 14 pre-trained on the JFT-300M dataset [31]. The best results with a ResNet50 are 98.3% accuracy [32].

4.3 Moment Queries Experimental Setup

Here, we will focus on our final proposals, validating their performance by different tests.

4.3.1 Dataset

For this testing, we are using the Ego4D dataset [2]. As the founders of the dataset themselves say: *Ego4D is a massive-scale Egocentric dataset of unprecedented diversity.*

Ego4D was released on the 17 February 2022 and is the first dataset ever to contain over 3670 hours of egocentric (i.e. filmed in a first-person point of view) videos of day-to-day activities. It has been very revolutionary as it drastically expands public video content for research. The videos are made with different types of cameras (perfect to avoid overfitting to a particular type), but all of them record the field of view that a human would see. The target of the dataset are daily life scenarios, so they deal with topics related to activities at home like cooking, cleaning, hobbies like playing or going to a party, transportation, etc.

Moreover, to allow doing research with the videos, they are densely narrated, with an average of 13.2 sentences per minute, and there are temporal markers pointing out the occurrence of different actions.

The action labels for the Moment Queries problem were taken semi-automatically using a pre-trained BERT [13] from the annotations and assigned manually to each category. Consequently, the MQ dataset has 110 categories, spans a total 326.4 hours of videos, 2,488 video clips and 22.2k action instances, split in a 6:2:2 ratio into the training, validation and test sets.

The distribution of the duration of actions is shown in Figure 21, the shortest actions (and therefore, the most difficult to detect) are the most common, with the average duration being 45.2 seconds.

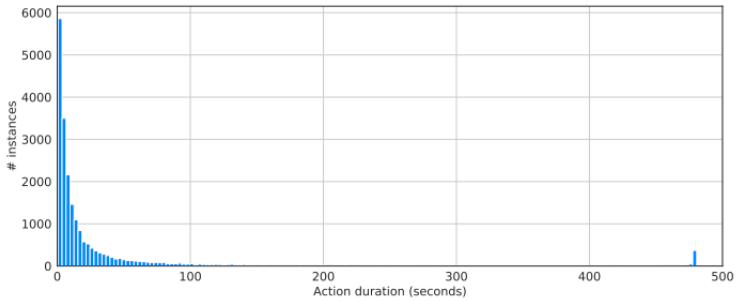

Figure 21: Distribution of moment duration.

The distribution of the number of actions in each category is shown in Figure 22. It follows a long-tail distribution, meaning that some actions are very common, but most of them have fewer examples.

For example, let us look at the video with ID: b8ea1d7f-1928-4699-8b6f-16fdbf5482f1 in Figure 23. This video is about someone cooking and doing housework. On the right of the interface, there is different information such as the narration of what happens in the video or the timestamps of the actions that occur.

In total, the dataset occupies more than 30 TB, which makes it seeming crazy to have a machine capable of coping with this information.

Fortunately, the videos are cataloged according to the benchmark for which they are prepared. We will only work with the ones that are tagged for the Moment Queries challenge. The videos in this folder are all annotated with the actions that occur in them.

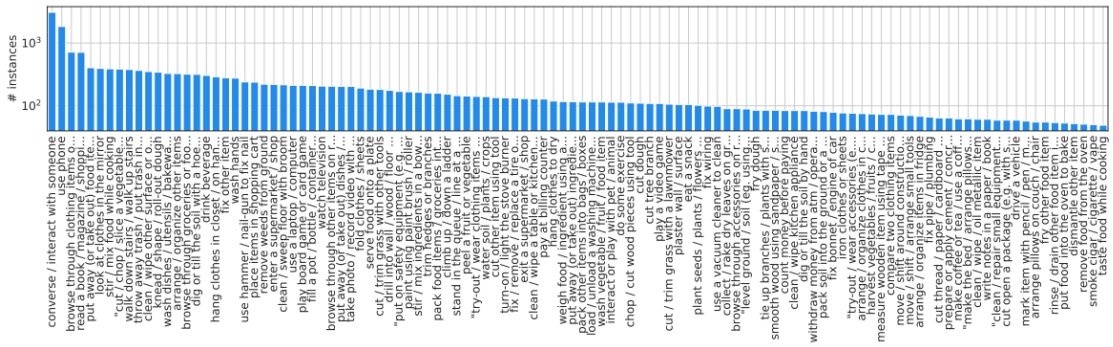

Figure 22: Distribution over scenarios for visual queries.

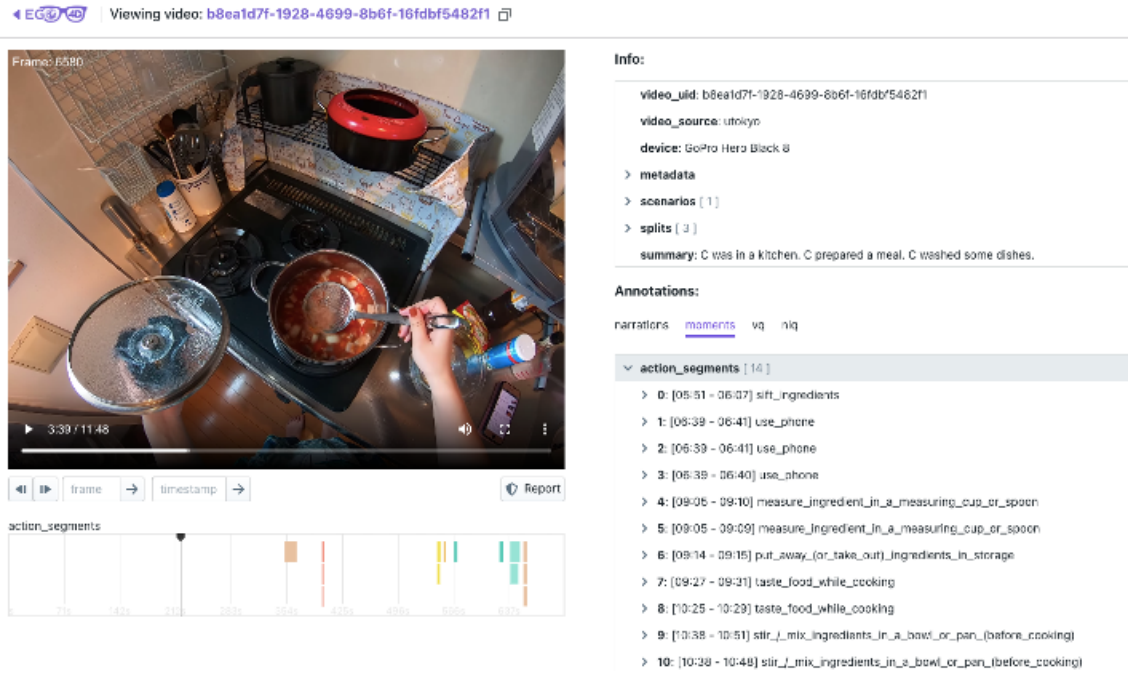

Figure 23: Capture of the interface for the video b8ea1d7f-1928-4699-8b6f-16fdbf5482f1 from the dataset.

In addition, we will not download the videos either, we will directly download the pre-computed features using SlowFast [22]. These features correspond to a vector of 2304 values for each frame of the video. In order to standardize the length of the videos so that none are too long in comparison, the videos are divided into clips of maximum 8 minutes, which would be around 897 frames. The maximum number of features per video is set to 928 to cover even the longest clip.

To run the algorithm, we also need the annotations of the videos and the dataset, which indicates the actions in the videos and the train/validation/test split. In total, the data we are using occupies 57 GB.

We would also like to add that, during the time that the tests have been performed, the competition for the best result on this challenge was still open. Therefore, the annotations of the videos on the test split were not published. The training was done on the train split and the evaluation on the validation split.

4.3.2 Validation metrics

To evaluate the results, we use the following metrics::

- **Average Precision (AP)**. This metric is widely used in temporal detection problems. It measures how close the temporal prediction is to the ground-truth for each action category. To measure the time distance, we use the time intersection of the union (tIoU) of the different time measures and compare it to a threshold.

$$IoU = \frac{\text{area of overlap}}{\text{area of union}} \tag{4.5}$$

  The thresholds, we will use are 0.1, 0.2, 0.3, 0.4 and 0.5. To compare the results, we use the average of the results with each one.

- **Recall@kx, tIoU=m**. This metric measures the percentage of queries that have at least one good prediction (with tIoU greater than the threshold) among the best k predicted results. However, since each action query may appear in more than one instance during the video, suppose it appears x times, then we will have to look at the best kx predictions. It is important to note that this metric only evaluates recall, it does not penalize for false positives.

  We evaluate for $k = 1, 2, 3, 4, 5$ and with thresholds of 0.3, 0.5 and 0.7. The reference measure we will take to compare them is $k = 1$ and threshold of 0.5, i.e. that the correct action is the most probable one and the value of tIoU with the real location is better than 0.5.

- **Time** of execution until the best results to evaluate its time efficiency.

- **Number of epochs** until the best results to evaluate how fast it learns.

- **Number of parameters** of the model to understand how big it is.

4.3.3 Implementation details

All coding has been done on python, using the pytorch and nn libraries. The model has been built from the original Ego4D baseline [18]. Many parts of the code were taken from the ViT implementation used for image classification.

The execution requires a lot of resources that we did not have on our computers, so everything has been written locally, debugged in Google Colab and executed in Peregrine.

The technical specifications were as follows:

- We have used the Adam optimizer with a learning rate of 0.0001 and for a maximum of 30 epochs.

- We have 6 levels in the Transformer. Each level is an Encoder block that also downsamples. The time frame lengths are of 928, 464, 232, 116, 58 and 29, respectively.

- The **Small** version has feature embedding of length 256, its attention is calculated in a space of dimension 256 with 8 heads and the hidden layers of the MLPs have 2048 neurons.

- The **Large** version has feature embedding and attention dimension of length 512 with 8 heads and the MLP's hidden layers have 4196 neurons.

4.3.4 Peregrine Server Specifications

As before, tests have been carried out with one Nvidia V100 GPUs with 12 nodes in the Peregrine Server[29] from the University of Groningen. In this case, we needed more memory space, the possible solutions are requesting for more in the server (which can extend queuing time) or reducing the batch size. The reported tests have been done with a batch size of 11, which is the maximum that can be run with 10.000MB but we also tested with more memory and larger batch sizes. The number of parallel processes has been limited to 12, which is what the server recommends.

4.4 Moment Queries Results

Table 3 shows an average of the results obtained with each of the models.

Arquitecture	**best_epoch**	**best_time**	**Params**	**av. mAP**	**recall**
vanila	15	766s	8.0M	5.88	24.70
xGPN	17	1353s	8.7M	**7.26**	**25.01**
ViT+	18	1122s	17.4M	4.80	25.41
ReMoT Small	15	1090s	22.7M	7.44	27.14
ReMoT Large	16	2301s	86.8M	**8.50**	**27.71**

Table 3: Results of the proposed architectures vs. the evaluated baseline models.

The vanila [18] model only uses convolutions. It is the fastest but also has the worst results. The xGPN model is the Ego4D [18] baseline model for the benchmark. We can see that it takes longer to converge than other models and the time is even worse, this may be due to the fact that it has to build a whole graph and neural network on top of it every iteration.

ViT+ gives rather mediocre results, improving a little in recall, but getting worse in accuracy. It is also the one that is quite difficult to converge, probably due to the fact that the encoder and decoder use completely different combinations.

ReMoT Small gives very good results and is very fast to train, even though it has 22M parameters. It is the best model if we want good results fast. ReMoT Large is the model that gives the best results, but that comes at the cost of the larger number of parameters and the maximum run time. Surprisingly, it does not usually take longer than models like xGPN or ViT+ to reach its best performance. Overall, the best models are the ReMoT ones.

Figure 24 shows the evolution of the loss as we train these models, with both versions of ReMoT reaching a much lower minimum. From the epochs 15-16 forward, the loss tends to increase, probably a sign of overfitting, especially in the ReMoT Large, since it has many more parameters.

The following graphs show the results of the metrics broken down into their different values. In Figure 25, we can see that ReMoT Small does not improve the precision much in any of the thresholds with respect to xGPN. ReMoT Large outperforms all the others by a wide margin.

As for recall, in Figure 26 we see that the two versions of ReMoT are consistently better. Both are fairly even except at the last threshold of 0.7, where the Large model excels a bit more, this is probably due to the fact that it has better precision.

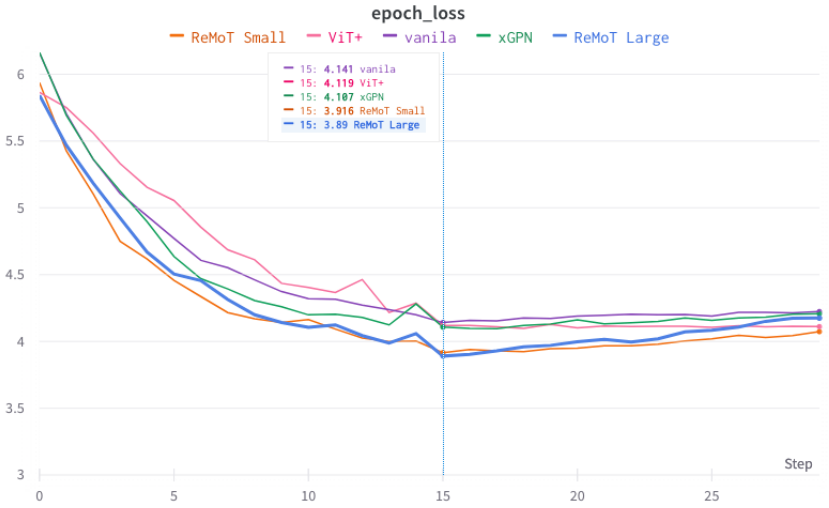

Figure 24: Graph on how the Loss evolves as the epochs advance. In the 15th epoch, the loss values of each model are shown.

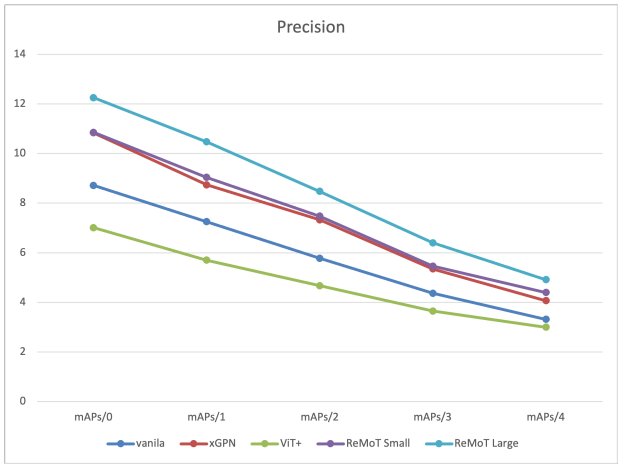

Figure 25: Mean Average Precisions for each threshold.

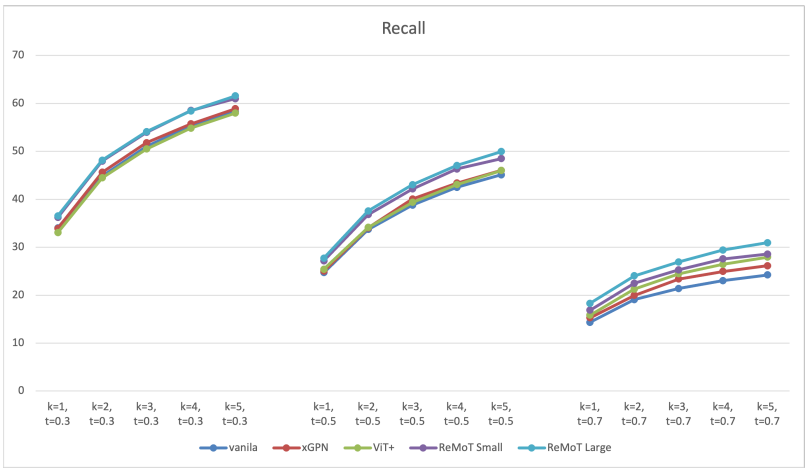

Figure 26: Recall values for each k and threshold t.

4.4.1 Hyperparameters

In order to measure the evaluation of the different models, many tests have been carried out. Below, there is a schematic summary of the results and why it has been decided to use each hyperparameter/structure.

Each test has been done several times and there can always be errors / random variables that can vary the results a bit, thus, the results presented here are the average of different tests or representative examples.

The study on the appropriate hyperparameters has been done by default on the ReMoT Small. Some tests have also been done with the Large version, but not all of them have been repeated, as they are very similar. We will take the Small one as a reference on the results unless otherwise stated.

The values of recall and average_mAP will be represented using two significant figures; for the number of parameters, the values will be given with M of millions.

At several points in the tables, references are made to hidden_dim. This variable represents the value of the embedding dimension and the attention dimension together, since, as will be seen later, we obtain better results if they have the same value.

4.4.1.1 Number of levels

Regarding the number of levels, we tested values from 2 to 6, decreasing the data size from one level to the next by half. It can be seen in Table 4 that the best results are obtained with 6 levels. The data shows a tendency to improve as we add levels. Despite this, given the fact that the dimension of the data is reduced by a factor of 0.5 at each level, at the last level, it is a vector of dimension 29. This vector cannot be reduced any further because it is odd and the decoder would reconstruct a vector of 30 values, which does not fit. We have tried changing the decoder, but each change we have tried has worsened the overall performance. Therefore, it is better to use 6 levels rather than 7.

architecture	num_levels	hidden_dim	average_mAP	recall	Params
ReMoT	2	256	4.84	19.25	10.5M
ReMoT	3	256	5.65	23.50	13.5M
ReMoT	4	256	6.14	24.47	16.6M
ReMoT	5	256	6.36	25.66	19.7M
ReMoT	6	256	**7.62**	**27.59**	22.7M
ReMoT-decoder	6	256	**7.56**	25.71	21.4M
ReMoT-decoder	7	256	7.30	**26.25**	24.4M

Table 4: Differences depending on the number of levels and the decoder used. ReMoT-decoder refers to the ReMoT architecture - the final proposal - with the mentioned modification to the decoder.

4.4.1.2 Different decoders

We have tested several decoder models: the original one, which uses convolutions, the one in our final proposal, which calculates attention with respect to the previous values of the decoder and those of the encoder, and a last one that did not pass the testing phase. The

latter was composed of only one attention relation using the encoder information as key and value and the decoder information as query, and then an MLP. The intuition behind this model was to modify the data in the larger dimension, from the encoder, using the information obtained in the previous levels. However, this model did not work so well, one can see a comparison of the results in Table 4 and 5.

4.4.1.3 Mask size

Different sizes of local attention mask have been tested as well; the results are in Table 5. Values between 10 and 50 have been considered, due to the fact that the last level has 58 time values, thus, larger masks would not make sense.

In the tests with the final Decoder, the metrics give good results. As long as the values are between 10-40, there is not much difference. However, when we used the previous Decoder, we could see a big difference, having a better performance at values around 32, so we decided to use this value.

architecture	**num_levels**	**hidden_dim**	**mask**	**average_mAP**	**recall**
ReMoT	6	256	10	7.63	27.21
ReMoT	6	256	20	7.58	**27.68**
ReMoT	6	256	30	**8.01**	27.43
ReMoT	6	256	40	6.87	26.46
ReMoT-decoder	6	256	10	6.66	23.93
ReMoT-decoder	6	256	20	7.74	25.41
ReMoT-decoder	6	256	30	7.39	25.29
ReMoT-decoder	6	256	40	**7.75**	**25.52**
ReMoT-decoder	6	256	50	7.45	24.47

Table 5: Differences depending on the mask size and the decoder used.

4.4.1.4 Overall size

Regarding the size, too small is about 128, which is not enough and 1024 is too much, it saturates, the best results are between 256 and 512. Changing the dim_att, but not the bb_hidden_dim or the mlp_num_hiddens can be done, but it does not make sense. The results vary very little and in no case improve. It works much better when the three values are on a par. The best results depending on the size are those in Table 6.

The results vary surprisingly little from the number of heads, as long as it is a reasonable value. Within the little change, the best results have always been found in every model with 8 heads.

4.4.1.5 Learning rate and number of epochs

The learning rate is very clear from the results in Table 7, the best learning rate is 0.0001, unlike in the original Ego4d baseline, where it was 0.0005.

The maximum number of epochs in each run is 30, which is already what the initial model does. The best epoch is usually around 15 or 16, sometimes 18, by 30 we are sure to already have obtained the best results.

Arq.	heads	dim_att	hid_dim	mlp_h	mask	av. mAP	recall	Params.
ReMoT	8	128	128	1024	16	4.45	23.76	6.1M
ReMoT	8	256	256	2048	32	7.62	27.59	21.3M
ReMoT	16	256	256	2048	32	7.06	26.42	22.7M
ReMoT	8	512	512	4096	32	**8.09**	**28.20**	86.8M
ReMoT	8	512	512	4096	64	7.63	27.73	86.8M
ReMoT	8	1024	1024	4096	128	6.87	26.02	247.0M

Table 6: Results obtained using different sizes; dim_att is the dimension of the space, where the attention is calculated, hid_dim (short for bb_hidden_dim) is the dimension of the feature vectors, determined by the embedding, mlp_h is the number of neurons in the MLP.

Optimizer	lr	Epochs	average_mAP	recall
Adam	0.00001	30	3.65	20.94
Adam	0.00005	30	7.21	25.08
Adam	0.0001	30	**7.62**	**27.59**
Adam	0.0005	30	4.27	23.43

Table 7: Example of results obtained by varying the learning rate.

In the Figure 27, there is a sample of how the loss develops as the model learns. Note that in the middle of the first graph, we reach the lowest zone and that the long training of the right graph does not improve at all.

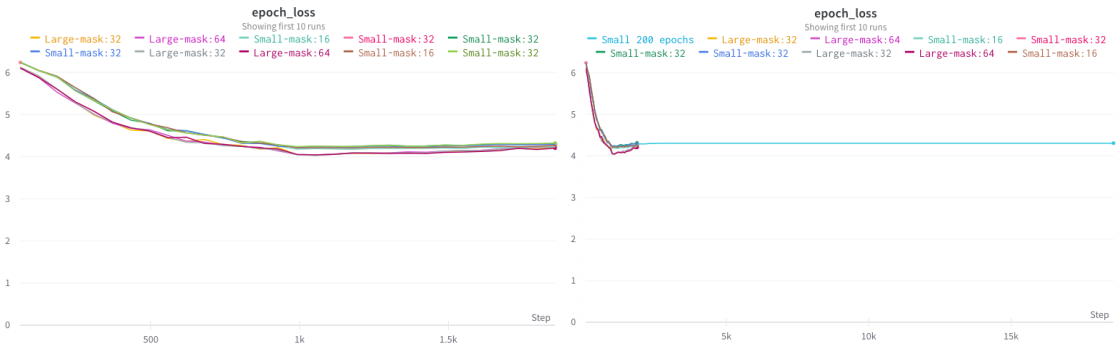

Figure 27: Graph of the loss as the steps progress. On the left, a graph with various runs of the ReMoT Small and Large. On the right, the same graph, but with another run lasting 200 epochs.

4.4.2 Ablation study

We have also performed a small ablation study on our model. We already mentioned in the previous section the variations by removing the decoder. Let us see the other changes.

4.4.2.1 Positional encoding

In the final proposal, it is quite clear that it is best not to use positional encoding, both learned and fixed encoding worsen the performance, as shown in table 8.

POS ENC	average_mAP	recall
None	**7.62**	**27.59**
Learned	6.41	26.53
Fixed	6.99	26.60

Table 8: Performance of the different positional encodings.

4.4.2.2 Wrong attention

This test could be verified thanks to an error in the code: for lack of a transposition, the attention was comparing time axes of the features instead of the features themselves. Table 9 shows the ridiculously bad results obtained. It shows what happens if there is no form of data augmentation or if it compares the time axes.

num_epoch	num_heads	hidden_dim	average_mAP	recall
30	2	768	0.29	11.81
30	4	768	0.38	11.15
30	8	64	0.29	12.61
30	8	256	2.36	**13.43**
30	8	768	1.33	11.55
30	8	1200	2.36	**13.43**
30	16	768	**2.43**	12.18
30	16	1600	1.86	12.96
100	8	768	1.83	12.42

Table 9: Sample of the results obtained with an erroneous attention calculation.

Finally, in Table 10, we wanted to mention some other changes that were proposed: changing the convolutional embedding to one made with a fully connected network, changing the convolutional downsampling method to max pooling and lastly using ReLU [33] instead of GELU [23].

Other changes	average_mAP	recall
Fully connected embedding	5.10	25.05
Downsampling with maxpool	7.46	25.50
ReLU instead of GELU	6.99	26.60
Reference	**7.62**	**27.59**

Table 10: Comparison of model performance with some changes.

5 Discussions

Once the results have been obtained, we proceed with the analysis and conclusions drawn from them.

5.1 Answer to the research objectives

Let us review the objectives of this work and whether they have been met.

Definitely, we have achieved a lot of knowledge about transformers and it has been put into practice with many experiments. Furthermore, we have studied different models and modifications of Transformers for both text and image in order to develop the best proposal.

We have studied the performance of ViT [14], reaching almost 80% accuracy, and SWIN [15], with more than 90%, on CIFAR10 [4], and we have compared it with other state-of-the-art models.

Although it has been an arduous task due to the novelty of the problem and lack of documentation, at the end, we obtained a lot of knowledge about action detection algorithms in egocentric videos and we have been able to combine it with our Transformer's experience in order to develop a proposal to improve the model.

We elaborated not only one, but two proposals to improve the memory retrieval. We have been able to implement it, analyze what went wrong and improve it. Although we have not obtained mind-blowing results, they consistently improve the performance of the original baseline. The proposed models work correctly and create a base for new lines of research in this topic.

As we have seen before in Table 3, our model has improved from 24.25% recall and 5.68% mean average accuracy to as much as 28.20% recall and 8.09% mean average accuracy in our best runs.

However, we know that in the Ego4D competition [34] there are other models that have reached 41.13% recall and 23.59% mean average precision, although there is no explanation on the algorithm used.

In particular, there is a model called ActionFormer [35], released on 2022, that uses the same intuition as ours, but is not anchor based and obtains results of 42.54% recall and 21.76% mean average precision. This model outperforms any other model with an impressive margin in the THUMOS14 [36] dataset, a dataset similar to Ego4D, but smaller and not egocentric.

Apart from the metric results, we believe that our model can bring a lot of value to the field of video action detection. The idea of this model is much simpler than the original baseline. While that model had to create graphs, calculate distances and apply convolutions, ours simply calculates attention, a mechanism that has been shown to be able to substitute all those methods. Moreover, we can detect new relationships between information and visualise them with the attention values. Apart from the fact that these weights can improve the explainability of the results. Furthermore, the attention values could be used for the following modules, for example to help to initially place the anchors, instead of doing it uniformly.

5.2 Future work

From now on, it would be interesting to change the anchor system to generate predictions. During the work we have not touched them because they were out of the study limits we had set, but now it would be time to get hands in these modules.

We could also study the idea of using a pre-trained Transformer, to take advantage of the transfer learning property of neural networks, ideally one dedicated to data augmentation like the ones in [37] or one trained on THUMOS14 [36].

On another note, we could also extend the problem so that, instead of just locating actions from a list, we could ask questions in natural language and the model interprets and answers the question. It is a much more complicated problem, but possible given the results of Transformers in the field of natural language.

5.3 Ethics

Finally, we would like to bring attention to some ethical issues involved in solving the problem that brings us to this paper. Recall that we are making a model to recognise what a person is doing at all times.

If the use of smart glasses becomes popular, this system can detect a lot of personal information that could seriously infringe the personal privacy.

One example is the case of a company giving these glasses/cameras to a worker to constantly analyse what they do during working hours. Yes, it would improve their performance, but in return they would lose all their independence. Nobody should have so much control and power over a person.

We suppose this same idea could be applied to someone who is on provisional release, so that what they do could be monitored to be sure they do not commit anything illegal. But once again, we need to question whether this is too invasive. We know that no matter how much data is deleted or anonymised, there is still the possibility of information leaks.

Even when used for oneself in a domestic environment, there is still the danger that the providers of this service can access this data. Especially knowing the track record that some multinationals have about collecting information and violating the users' privacy, we can only imagine what could happen with all this detailed information.

At the end of the day, this algorithm is a powerful tool to help people remember things and improve their life, but we cannot ignore the danger of invasion of privacy.

6 Conclusions

In this work we have seen how Transformers work and how they can be applied to different fields. In addition, we have faced a very challenging problem of the current artificial intelligence research landscape where we have done our bit.

Transformers are very powerful neural networks that require a lot of data to give their best results. It is on problems like this, with such a specialised and huge dataset that we give it the opportunity to shine. And shine it has done, as it has out-performed the model proposed by Ego4D. The results show that we are on the right track and that there is still a lot to explore.

To sum up, this work has been an invaluable frontline research experience with an international team where we have worked with novel technologies and problems that may define tomorrow's world.

For me, it has been a very interesting challenge and I have learned a lot. Moreover, I have achieved some technical knowledge that can open up many research and professional opportunities. I have always been fascinated by the world of artificial intelligence and this project has allowed me to explore it in much more depth and see the current state of machine learning, an exciting field to which I want to continue to dedicate myself and innovate.

7 Bibliography

[1] Ashish Vaswani et al. "Attention Is All You Need". In: (2017). DOI: 10.48550/ARXIV.1706.03762. URL: https://arxiv.org/abs/1706.03762.

[2] Kristen Grauman et al. "Ego4D: Around the World in 3,000 Hours of Egocentric Video". In: (Oct. 2021). DOI: 10.48550/ARXIV.2110.07058. URL: https://arxiv.org/abs/2110.07058.

[3] Aston Zhang et al. "Dive into Deep Learning". In: *arXiv preprint arXiv:2106.11342* (2021).

[4] Tien Ho-Phuoc. *CIFAR10 to Compare Visual Recognition Performance between Deep Neural Networks and Humans*. 2018. DOI: 10.48550/ARXIV.1811.07270. URL: https://arxiv.org/abs/1811.07270.

[5] Warren S. McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133. DOI: 10.1007/BF02478259. URL: https://doi.org/10.1007/BF02478259.

[6] Frank Rosenblatt. "Perceptron Simulation Experiments". In: *Proceedings of the IRE* 48.3 (1960), pp. 301–309. DOI: 10.1109/JRPROC.1960.287598.

[7] Alex Sherstinsky. "Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network". In: *Physica D: Nonlinear Phenomena* 404 (Mar. 2020), p. 132306. DOI: 10.1016/j.physd.2019.132306. URL: https://doi.org/10.1016%5C%2Fj.physd.2019.132306.

[8] Andrei Shcherbakov µ Saliha Murado and glu Ekaterina Vylomova µ. *Exploring Looping Effects in RNN-based Architectures*.

[9] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: (2014). DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

[10] Rachel Ward, Xiaoxia Wu, and Leon Bottou. "AdaGrad stepsizes: Sharp convergence over nonconvex landscapes". In: (2018). DOI: 10.48550/ARXIV.1806.01811. URL: https://arxiv.org/abs/1806.01811.

[11] Thomas Kurbiel and Shahrzad Khaleghian. "Training of Deep Neural Networks based on Distance Measures using RMSProp". In: (2017). DOI: 10.48550/ARXIV.1708.01911. URL: https://arxiv.org/abs/1708.01911.

[12] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. DOI: 10.48550/ARXIV.1512.03385. URL: https://arxiv.org/abs/1512.03385.

[13] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2018. DOI: 10.48550/ARXIV.1810.04805. URL: https://arxiv.org/abs/1810.04805.

[14] Alexey Dosovitskiy et al. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: (2020). DOI: 10.48550/ARXIV.2010.11929. URL: https://arxiv.org/abs/2010.11929.

[15] Ze Liu et al. "Swin Transformer: Hierarchical Vision Transformer using Shifted Windows". In: (2021). DOI: 10.48550/ARXIV.2103.14030. URL: https://arxiv.org/abs/2103.14030.

[16] Kristen Grauman et al. *Moment Queries challenge proposal*. 2022. URL: https://ego4d-data.org/docs/benchmarks/episodic-memory/.

[17] Kristen Grauman et al. *Ego4D web*. 2022. URL: https://ego4d-data.org/.

[18] Kristen Grauman et al. *GitHub of the Ego4D's baseline*. 2022. URL: https://github.com/EGO4D/episodic-memory/tree/main/MQ.

[19] Chen Zhao, Ali Thabet, and Bernard Ghanem. "Video Self-Stitching Graph Network for Temporal Action Localization". In: 2021. DOI: 10.1109/ICCV48922.2021.01340.

[20] Haoqi Fan et al. "Multiscale Vision Transformers". In: (2021). DOI: 10.48550/ARXIV.2104.11227. URL: https://arxiv.org/abs/2104.11227.

[21] Tianqi Liu and Qizhan Shao. "BERT for Large-scale Video Segment Classification with Test-time Augmentation". In: (2019). DOI: 10.48550/ARXIV.1912.01127. URL: https://arxiv.org/abs/1912.01127.

[22] Christoph Feichtenhofer et al. *SlowFast Networks for Video Recognition*. 2018. DOI: 10.48550/ARXIV.1812.03982. URL: https://arxiv.org/abs/1812.03982.

[23] Dan Hendrycks and Kevin Gimpel. "Gaussian Error Linear Units (GELUs)". In: (2016). DOI: 10.48550/ARXIV.1606.08415. URL: https://arxiv.org/abs/1606.08415.

[24] *Papers with code GELU*. URL: https://paperswithcode.com/method/gelu.

[25] Lukas Biewald. *Experiment Tracking with Weights and Biases*. Software available from wandb.com. 2020. URL: https://www.wandb.com/.

[26] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[27] Clément Delangue. *Hugging Face*. 2016. URL: https://huggingface.co/.

[28] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: (2014). DOI: 10.48550/ARXIV.1409.0575. URL: https://arxiv.org/abs/1409.0575.

[29] *Peregrine Server Documentation*. URL: https://wiki.hpc.rug.nl/peregrine/start.

[30] *Classification Ranking over CIFAR10*. URL: https://paperswithcode.com/sota/image-classification-on-cifar-10.

[31] Chen Sun et al. "Revisiting Unreasonable Effectiveness of Data in Deep Learning Era". In: (2017). DOI: 10.48550/ARXIV.1707.02968. URL: https://arxiv.org/abs/1707.02968.

[32] Ross Wightman, Hugo Touvron, and Hervé Jégou. *ResNet strikes back: An improved training procedure in timm*. 2021. DOI: 10.48550/ARXIV.2110.00476. URL: https://arxiv.org/abs/2110.00476.

[33] Abien Fred Agarap. "Deep Learning using Rectified Linear Units (ReLU)". In: (2018). DOI: 10.48550/ARXIV.1803.08375. URL: https://arxiv.org/abs/1803.08375.

[34] *Moment queries competition leaderboard*. URL: https://eval.ai/web/challenges/challenge-page/1626/leaderboard/3913.

[35] Chenlin Zhang, Jianxin Wu, and Yin Li. *ActionFormer: Localizing Moments of Actions with Transformers*. 2022. DOI: 10.48550/ARXIV.2202.07925. URL: https://arxiv.org/abs/2202.07925.

[36] Haroon Idrees et al. "The THUMOS challenge on action recognition for videos "in the wild"". In: *Computer Vision and Image Understanding* 155 (Feb. 2017), pp. 1–23. DOI: 10.1016/j.cviu.2016.10.018. URL: https://doi.org/10.1016%5C%2Fj.cviu.2016.10.018.

[37] Varun Kumar, Ashutosh Choudhary, and Eunah Cho. *Data Augmentation using Pre-trained Transformer Models*. 2020. DOI: 10.48550/ARXIV.2003.02245. URL: https://arxiv.org/abs/2003.02245.