**PRE-DATA SEMINAR FOR Ph.D. PROGRAMME**

**TITLE:**

**Grouped-Global Attention Mechanism for Efficient Memory Management in Transformer**

**BY**

**DADA, Ibidapo Dare**

**PG17/0045**

**UNDER THE SUPERVISION OF:**

**PROF A.T. AKINWALE (MAJOR SUPERVISOR)**

**PROF. (MRS) ONASHOGA (CO SUPERVISOR)**

**PROF. I.A. OSINUGA (CO SUPERVISOR)**

**CHAPTER ONE**

**INTRODUCTION**

**1.1 Background to the study**

Machine learning is a research field that strengthens computers to learn and manipulate data appropriately to make new data predictions or classifications. Deep learning is a subset of machine learning built upon artificial neural networks. The idea of artificial neural networks is to simulate the human brain, allowing it to learn from huge amounts of data. Deep learning only become very popular in the recent year while most of its building blocks has been in existence for decades. So, the idea of artificial neural networks is dated back to the 1940s when McCullough-Pitts Neuron (Mcculloch et al., 1943) was first introduced. McCullough-Pitts Neuron is a linear model which linearly aggregates information from input data with two categories to make decisions. Thereafter, traced back to 1958, the perceptron (Rosenblatt, 1958) was developed which was the beginning of parameters learning given some training data samples. One of the big success of artificial neural network was in the 1980s when backward propagation algorithms (Rumelhart, 1986) was used to train deep neural network models. The use of this backward propagation algorithm still dominates the training of deep neural networks to date. Deep learning models are gaining unprecedented attention with the emergence and availability of big data and powerful computational resources. The advantage of fast GPUs/TPUs allows us to train deep learning models on huge sizes of data and an increase in large data (parallel data) helps the models to generalize very well. So, for these two reasons, deep learning models have recorded huge success in numerals research fields which has greatly impacted the real world at large. Deep learning techniques have greatly impacted fields like Natural Language Processing, Computer Vision, Speech Recognition, bioinformatics et cetera.

Deep learning architectures like feedforward networks, convolutional neural networks, recurrent neural networks, long short-term memory networks, gated recurrent unit networks and autoencoder networks has made great advancement in the field of deep learning. Furthermore, in modern deep learning architectures, the transformer network (Vaswani et al., 2017) has shown excellent results in large-scale deep learning applications. The transformer model was first proposed in the context of neural machine translation by (Vaswani et al., 2017) which has been shown to be able to generalize well across a wide variety of natural language processing tasks when pre-trained on a large chunk of text such as text generation, summarization, questioning answering, named entity recognition, text classification and text translation (Brown et al., 2020; Devlin et al., 2018; Radford et al., n.d.). The successful application of transformers to natural language processing applications has facilitated the transformer’s application to other domains such as speech recognition (Luo et al., 2020), music generation (Huang et al., 2018), bioinformatics (Y. Du et al., 2020; Madani et al., 2020; Rives et al., 2019), In biology transformer is used to predict protein structure and function (Elnaggar et al., 2019), vision transformer (Dosovitskiy et al., 2020), video vision transformer (Arnab et al., 2021) to mention a few.

The core component of the transformer architecture is the attention mechanism which helps in identifying complicated dependencies among every set of the input sequence. The defining trait of the transformer model is its attention mechanism. Attention mechanisms have become immensely popular within the Artificial Intelligence community as indispensable components of neural architectures for a great many applications. The intuition for attention mechanism can be illustrated via the human biological system. Attention is an essential cognitive ability that is vital for humans (Niu et al., 2021). Humans tend to not take in entire information at once, but instead, they focus on particular aspects of the information when and where it is necessary, without taking into account other perceivable information at the same time. This is a way for humans to rapidly identify meaningful information from vast quantities of information using limited processing power. Let's consider how humans process their visualization system, Humans focus on some selected part of an image and disregard the rest for improved perception (Xu et al., 2015). Similarly, in language, speech or vision-related problems, certain parts of the input are more significant than others. For example, in tasks like machine translation and text summarization, only a few words from the input sequence are necessary for predicting the next word. The attention mechanism accounts for this notion of relevance by allowing the model to pay attention to specific parts of the input which would aid in effectively performing a particular task.

Unfortunately, the regular transformer is faced with a well-known problem for the attention mechanism which is the quadratic time and memory complexity problems. These two major problems in that the scalability of the transformer model. Therefore, this study proposes a new set of attention-based mechanisms for transformer models to reduce the memory complexity for performing some natural language processing tasks.

**1.2 Motivation**

The attention mechanism has become a popular part of transformer architectures, applied to a variety of tasks like image generation (Lu et al., n.d.; Xu et al., 2015), machine translation (Britz et al., 2017; Su et al., 2020; Sutskever Google et al., n.d.; Vaswani et al., 2017), action recognition (Song et al., 2017; P. Zhang, 2018), speech recognition (Chan et al., 2015; Chorowski et al., 2014; di Gangi et al., 2019) and recommendation systems (H. Ying et al., 2018; Wang et al., 2018). However, this expressive pairwise attention uses a lot of computation resources (time and memory) (Ren et al., n.d.) so having an attention mechanism that is more efficient would be better.

Since the development of the vanilla transformer which was originally used for machine translation, transformer architectures have been applied to other fields speech recognition (Luo et al., 2020), music generation (Huang et al., 2018), bioinformatics (Y. Du et al., 2020; Madani et al., 2020; Rives et al., 2019), vision transformer (Dosovitskiy et al., 2020), video vision transformer (Arnab et al., 2021).

**1.3 Problem Statement**

The regular transformer (Vaswani et al., 2017) is cost and resource-intensive with a quadratic cost in both Time And memory consumption for an input sequence of length . However, recently proposed solutions such as sparse transformers, low rank/kernels transformers, memory down sampling transformers, and fixed/factorized/Random patterns transformers are more efficient alternatives. These approaches do not try to emulate regular attention, but rather offer simpler and more manageable attention mechanisms - except for low-rank/kernel transformers that need more base models to better replicate the exponential kernel. However, these approaches can limit the ability of the model to capture complex dependencies between tokens in the input sequence.

While the majority of natural language processing (NLP) research has traditionally centered on tasks involving single texts or documents, there exists a crucial realm of applications that require handling aggregated information spread across multiple texts. The conventional approach to capturing relationships among documents, particularly in tasks like extracting information from several documents, tends to mirror the methods used for single-document tasks. Unfortunately, this approach may inadvertently overlook potential relationships between distinct documents, which could contain overlapping, complementary, or conflicting information. Interestingly, the trajectory of research on self-attention mechanisms has consistently adhered to a similar pattern. The prevalent practice involves amalgamating multiple documents into a flat sequence, neglecting the nuanced exploration of relationships between these documents. Consequently, a promising avenue for further investigation lies in the development of enhanced mechanisms capable of integrating diverse cross-document relationships.

**1.4 Research Questions**

Attempts were made to provide answers to the following questions

i. How can the computational efficiency of the traditional transformer model be improved while retaining its ability to capture complex dependencies within input sequences?

ii. How can attention mechanisms be enhanced to effectively integrate diverse cross-document relationships in multi-document NLP tasks?

iii. What novel methodologies can be devised to address the inefficiencies of current transformer architectures and facilitate more robust performance in handling aggregated information spread across multiple texts?

v. How do these proposed enhancements impact the scalability, performance, and generalizability of transformer-based models across a range of NLP applications?

* 1. **Research Objectives of the Study**

This study aims to develop an efficient attention mechanism to enhance the computational efficiency and effectiveness of transformer models for NLP tasks, particularly in the context of handling aggregated information across multiple documents. The specific objectives for this research are to:

1. Develop an efficient attention technique to improve the computational efficiency of transformer architectures.
2. To assess the effectiveness of the technique in balancing computational efficiency with the ability to capture information in cross-document relationships effectively.
3. Apply and evaluate the performance of the proposed technique to natural language processing tasks, especially in multi-document summarization tasks.
4. Comparing the proposed technique to baseline models with appropriate metrics.

**1.6 Organization of the Thesis**

**CHAPTER TWO**

**2.0** **LITERATURE REVIEW**

As this research on “efficient transformer models” has a strong foot in deep learning, it is essential to understand some basic deep learning techniques. This chapter, therefore, introduces some fundamental deep learning techniques and carries out an in-depth review of related works and literature to meet the objectives of this research. Particular attention is paid to finding an effective transformer model that can be used in deep learning and its related application domains.

**2.1** **Foundation of deep learning**

Deep learning is a branch of Artificial Intelligence that uses algorithms to enable machines to learn from data. It enables computers to learn without explicit instruction. The basis of deep learning is to create a robust set of algorithms and architectures that enable computers to learn from a vast array of data. This involves applying layers of neurons to represent complex ideas and systems so that the computer can identify patterns and make decisions. Many deep learning algorithms and architectures have been proposed over the past decade which include feedforward neural networks, convolutional neural networks, recurrent neural networks, autoencoders and transformers. This architecture will serve as the foundation for this research work.

**2.2** **Feedforward Networks**

Feedforward Networks, sometimes known as Deep Feedforward Networks or Multi-Layer Perceptrons (MLPs), are essential for many key deep learning models. Feedforward Networks are hugely important for machine learning practitioners and underpin many commercial applications. These networks are a crucial part of the journey leading to Convolutional Networks that are used in object recognition and the powerful building blocks for Recurrent Networks that power many natural language processing applications.

A Feedforward Network aims to approximate a given function using some set of data . For instance, in the case of a classification task, a perfect classifier would map an input to a category . Feedforward Networks should find a mapping and learn the parameters that result in the best approximation of . This process is known as 'feedforward' since the information from input data flows sequentially through various operations in order to define function , before being sent out to the output y as shown in Figure 1. The intermediate computational operations generally take place in the form of networks composed of several functions. Figure 1 shows four functions connected in a chain to create This chain structure is the typical neural network, with being referred to as the first layer, as the second layer, as the third layer, and as the output layer since it is the last layer. The overall length of this chain denotes the depth of the model and it is from this idea that 'deep learning' derives its name.

When training a neural network, the goal is to make the output, , resemble the ideal output, . For example, when the input is given with a label , it is decided that at each point , the output layer should generate a value that resembles . However, what is required of the other layers is not specified by the training as there is no given desired output during this process. Consequently, the learning algorithm must determine how best to use those layers in order to approximate , these layers are called hidden layers.

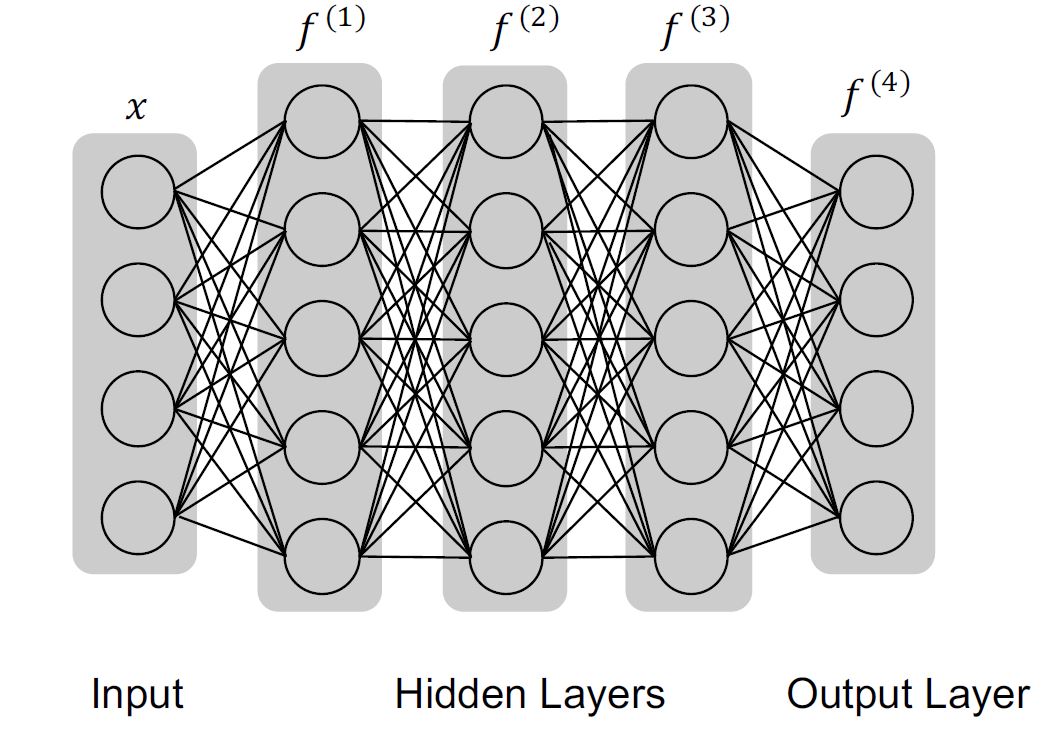


Figure 1: An example to illustrate a feedforward network.

Every layer of a neural network can be seen as a vector-valued function, where both the input and output are in the form of vectors. The components of the layer can be thought of as nodes (or units). Therefore, each layer is essentially a collection of vector-to-scalar functions, with each node representing one particular function.

**2.2.1 The Feed-forward Architecture**

In any fully connected feedforward neural network as shown in Figure 1, one node in a layer is connected to all nodes in the next layer. So the best way to understand in detail the computation in the neural network is to focus on a single note in the first layer. Therefore, operation on a single node is shown in Figure 2.

The input of neural network is represented by a vector , where is used to denote element. Viewing all inputs elements as nodes which are fully connected to a node in the layer next to it as illustrated in Figure 2. The operations in one node can be grouped into two: the first operation is to combine the elements in the input layer linearly with some weights and the second operation is to pass the value obtained in the first operation through an activation function. This operation can be represented mathematically as,

Where is an activation function, is a bias term, is an arbitrary hidden layer, is an arbitrary number of elements in each layer.

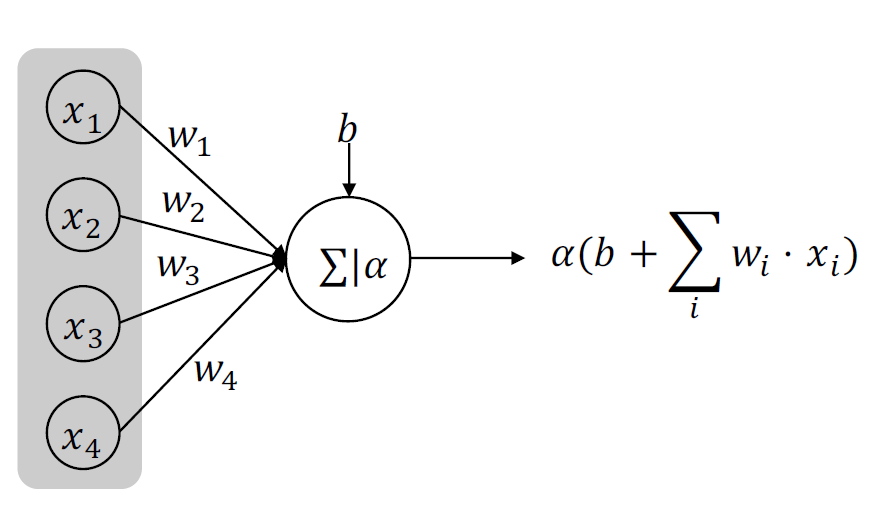


Figure 2: Operations in a node

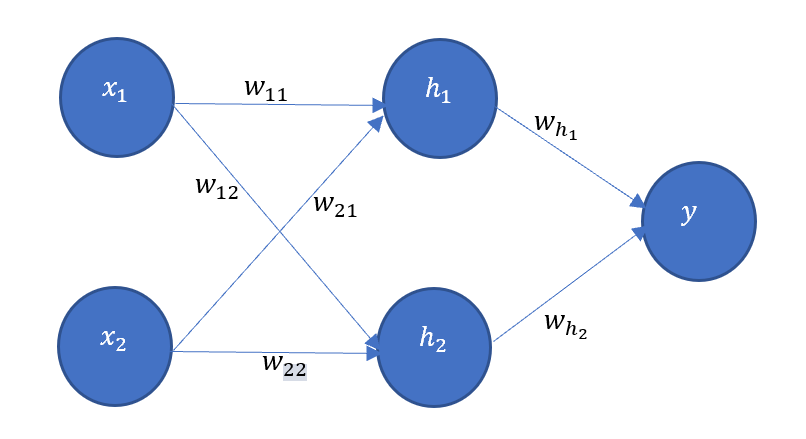
To be more specific, assuming in the layer of the neural network, there are nodes and the output of this layer can be represented as with denoting its element. Computing the in the layer can be done through this mathematical operation:

Where denote the weight corresponding to the connections between layer and layer and is the bias term used to calculate . In order to calculate all the elements in layer, using matrix form will make it easier. The matrix form is given as:

Note that contains all weights, consists of all the bias terms.

**2.2.1.1 Example: Learning XOR**

In order to fully understand the idea behind feedforward network, a very simple task will be performed using a fully functioning forward neural network: Learning the XOR function also known as " the exclusive or function". The XOR perform its operation on two binary values and . The function returns 1, if and only if one of these binary values is 1 otherwise the function returns 0. The target function has been provided by the XOR function while a feedforward model is expected to provide a function so that the learning algorithm should use the parameters to make similar as much as possible to .

Figure 3: illustration of feedforward model.

So in the case of the XOR problem, a very simple forward neural network with one hidden layer consisting of two hidden nodes as shown in Figure 3 is constructed to solve this problem.

From Figure 3, the following matrices can be constructed:

, ,

the network has a vector of hidden unit , which is computed by a function . The values of the hidden units are used as input for the second layer which happens to be the output layer in this case. Therefore, the model has two functions combined together and to give . Where c is biases matrix of the hidden layer and b is a bias of the output layer.

The use of nonlinear function is needed to describe the features. Many neural networks do so by using an affine transformation controlled by the learned parameters, followed by a fixed nonlinear function called and activation function. Therefore, can be redefined as:

Where is weights of a linear transformation, the biases and the activation function. The activation function is typically chosen to be a function that is applied element-wise with . The ReLu function also known as rectified linear unit is the default activation function recommended (Glorot et al., 2011; Nair & Hinton, n.d.) which is defined by . (Activation functions will be discussed in detail later in the chapter). The complete network can be define as:

To solve the XOR. Let

, , and

**Step1:-** multiply the inputs matrix x by weight matrix of layer one.

Step 2: Add the bias vector c,

Step 3: compute the hidden layer value by applying the ReLu activation function.

Step 4: multiply hidden layer value by weight vector w to give the output of the model y.

The neural network gives a very correct results for each sample point. In this example, it is observed that the solution obtained was zero error because of the small number of parameters and training samples but in real life situation, billions of parameters and training samples may be involved, so the solution cannot be guessed. To solve this types of problem many other components will be considered to better enhance the feed-forward architecture.

**2.2.2 The Activation Functions**

Activation functions tells whether or not a neuron or node should be activated or not. This means the activation functions decide to what extent is the inputs to the node (or neuron) important or contributing to the process of prediction using some mathematical operation. In other words, activation functions enables a neural network to utilize important information and disregard irrelevant information. When activation functions act on neural networks it introduced non-linearity to the network which improves its approximation capacity. We will consider some commonly used activation function here.

**2.2.2.1 Step Function**

Step function which is commonly called binary step-function, it deals with weather a neuron (or node) should be activated or not. A threshold value will be given to the neuron; the input to the activation function and the threshold value will be compared. If the input is greater than the threshold value, the neuron is activated otherwise the neuron is deactivated. Figure 5 shows how step function works.

Its mathematical representation goes as:



Figure 4: binary step-function

(Source: https://www.v7labs.com/blog/neural-networks-activation-functions)

One common drawback of this activation function is that it does not provide multi-value outputs also the problem with backward propagation since the gradient of this function will definitely be zero. However, the step function can be used for single output network to solve binary classification problems.

**2.2.2.2 Linear Activation Function**

Neural network without any activation function can be referred to linear activation function which can also be called no activation or identity activation. This function makes the activation proportional to its input that is the function output the same number given to it by the weighted sum of the input. Figure 5 shows the graphical representation of the linear activation function.

It can also be represented mathematically as:

One of the drawback of this activation function is the backward propagation problem since the derivative of the linear function will definitely be a constant and the function shows no representation between the inputs x and derivative of the function. Another problem of this activation function is that it combines all layers of the neural network into one no matter how many layers in the network since the last layer is still using a linear function.

However, the linear activation function is useful in the output layer of a neural network when solving regression problem.

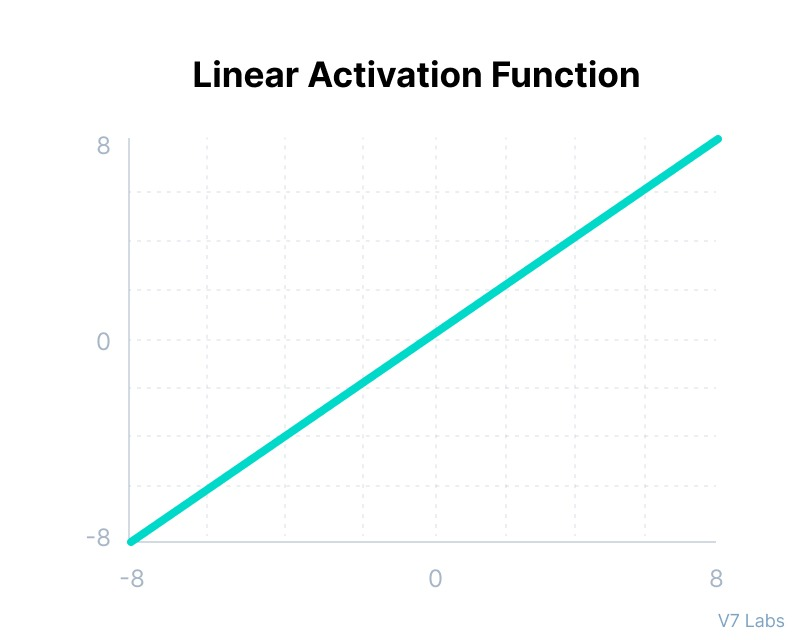


Figure 5: Linear Activation Function

(Source: https://www.v7labs.com/blog/neural-networks-activation-functions)

**2.2.2.3 Sigmoid/Logistic Activation Function**

Logistic sigmoid activation function is one of the commonly adopted activation functions. The sigmoid activation function can be represented mathematically as

The input of this function can be any real value, and its output will always fall within the range of 0 to 1. The output value tends towards 1.0 as the input value becomes larger (more positive), and towards 0.0 as the input value becomes smaller (more negative). This behavior is demonstrated in Figure 6.

The sigmoid or logistic activation function is widely used for several reasons. Firstly, it is often used in models where the output is a probability because its range is between 0 and 1. Therefore, ensuring that the predicted probability falls within this range is a suitable choice.

Secondly, the sigmoid function is differentiable and has a smooth gradient. The S-shaped curve of the function prevents abrupt changes in output values and ensures that small changes in the input result in small changes in the output. This smoothness is useful in optimizing the parameters of the model using gradient descent or other optimization techniques.

Overall, the sigmoid activation function is a popular choice for many machine learning applications because it satisfies the need for a smooth and differentiable function with a range of 0 to 1, making it well-suited for predicting probabilities.

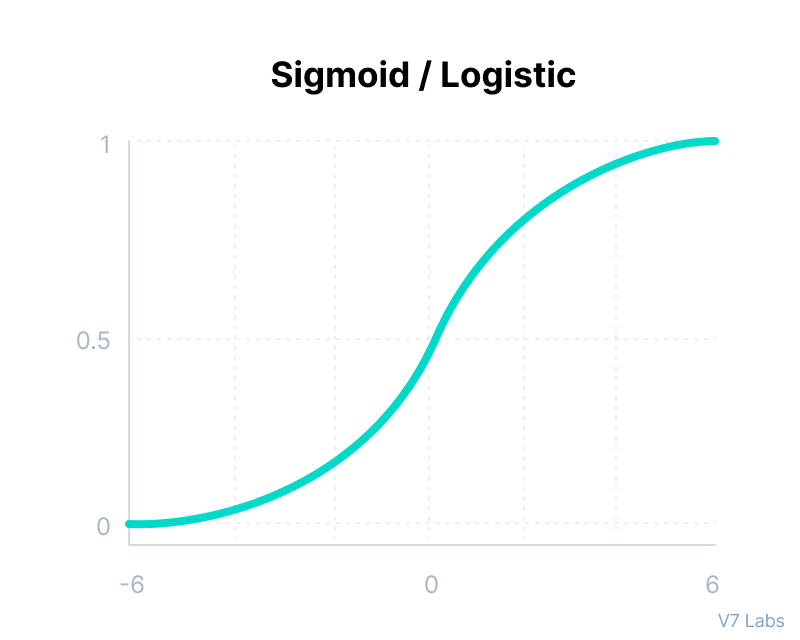


Figure 6: Logistic Sigmoid Activation Function

(Source: https://www.v7labs.com/blog/neural-networks-activation-functions)

One of the limitations of this activation function according to research, the sigmoid function is not ideal because the Y values exhibit a slight response to changes in X values towards either end of the function. This results in a small gradient at this point, which leads to the vanishing gradient problem. This problem occurs on the near-horizontal parts of the activation function’s curve on either side, where the gradient becomes very small and cannot cause significant changes since its value is minimal. As a result, the network may learn very slowly or even refuse to learn further, depending on how it is untilized until the gradient approaches the floating-point value limits.

**2.2.2.4 Hyperbolic Tangent Activation Function**

Hyperbolic tangent also known as tanh function is highly related to the Sigmoid activation function. The output range of tanh function is from -1 to 1 as opposed to the 0 to 1 range of sigmoid. Using tanh function, the output value tends toward 1 as the input value becomes larger (more positive) and tend toward -1 as the input value becomes smaller (more negative). This is demonstrated in the Figure 7.

The mathematical representation is as follow:

The tanh function is often used as an alternative to Sigmoid function in neural networks especially in situations where negative value maybe highly significant. As a result of these properties, the hyperbolic tangent function maps negative inputs to negative outputs and input near zero to output near zero. This means that the network is less likely to become stuck during training and can effectively and do imports of varying magnitudes. Tanh is often preferred over Sigmoid due to the derivatives of the tanh function are significantly larger near zero than the

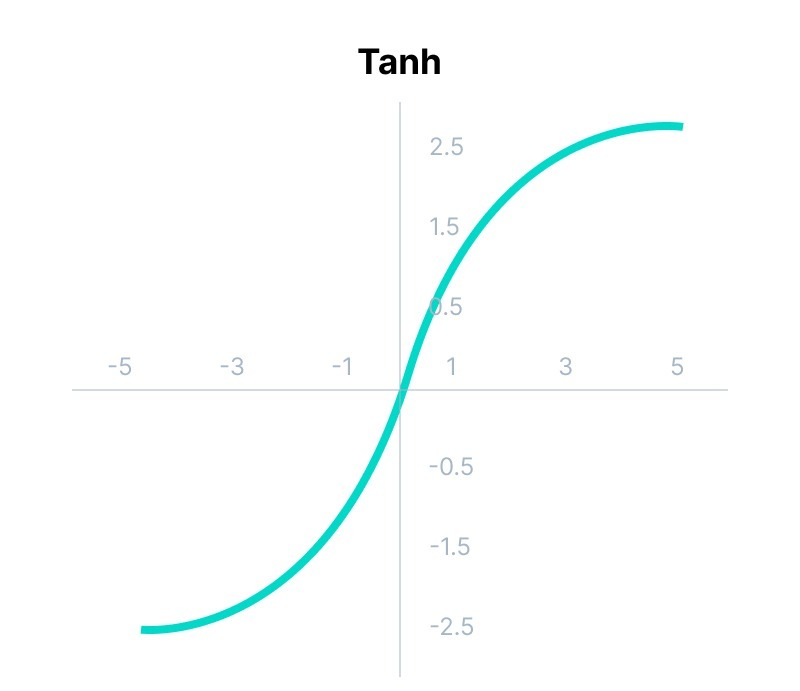


Figure 7: Hyperbolic Tangent Activation Function

(Source: https://www.v7labs.com/blog/neural-networks-activation-functions)

derivatives of Sigmoid, for this reason tanh function minimize the cost function quickly which help in speeding up the optimization process of the network.

Tanh activation function suffers the same vanishing gradient issue faced by the sigmoid function. The gradients of the tanh function is stepper than that of the Sigmoid function. However, the tanh activation function are commonly used in case of binary classification. So using tanh or Sigmoid in the final layer produces a quality that can be scaled from 0 to 1.

**2.2.2.5 ReLu Activation Function**

ReLu activation function means Rectified linear unit. This activation function is one of the most used activation function in neural network. From Figure 8, the rectifier activation function is similar to linear functions, with the only difference being that it output 0 for negative inputs. The main reason for the rectifier activation function to be a popular choice for neural network is that it does not activate all the neurons at the same time, that is it is linear for all positive inputs values and is 0 for all negative values which the neurons will only be activated if the output of the transformation is positive (greater than 0). ReLu activation function is defined mathematically as:

One disadvantage of ReLu activation function is that on the negative half of the domain is gradient is zero which means that during training weights and biases of some neurons with negative value will not be updated, this will lead to dead neurons which will never be activated.

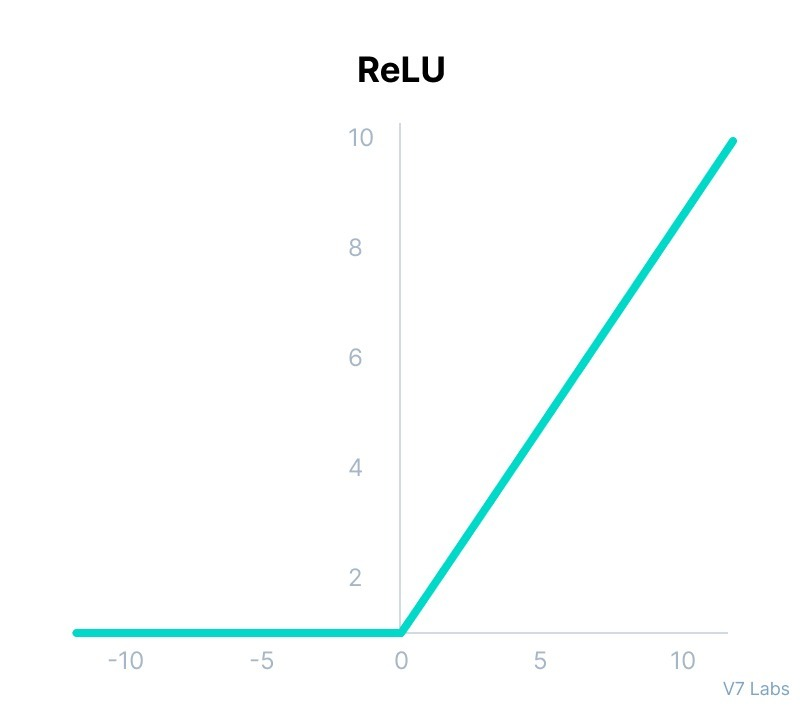


Figure 8: ReLu Activation Function

(Source: https://www.v7labs.com/blog/neural-networks-activation-functions)

**2.2.2.6 Leaky ReLu Activation Function**

Leaky ReLu is an improvement on ReLu activation function to overcome the drawback of ReLu activation function. Leaky ReLu performs a linear transformation with a small positive slope to the negative value, Figure 9 shows that. Mathematical representation of LeakyReLu is given as:

One of the problems faced by this function is that the predictions may varies for negative input value and the model may be very slow in training the model parameters due to the small value produced by the gradient of the negative values.

**2.2.2.7 Exponential Linear Unit Activation Function (ELU)**

ELU is another generalization of the ReLu activation function which modifies the slope of the negative values of the function. As shown in Figure 10, it adopts an exponential transformation for the negative part of the function while still hold on to the identify transform for the positive part. Mathematically, ELU is represented as:

The major problem with this activation function is that it computational time increases because of the exponential operation.

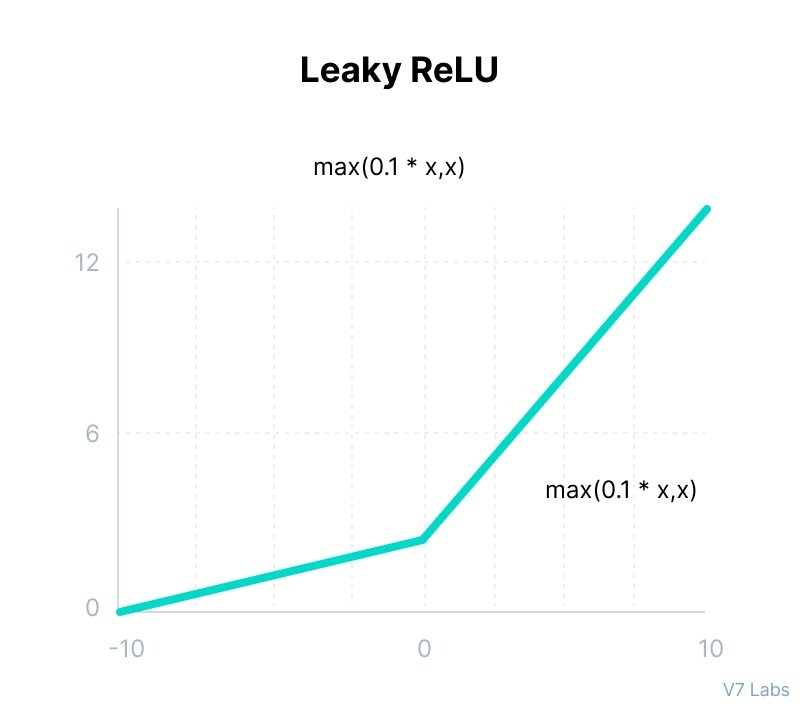


Figure 9: LeakyReLu Activation Function

(Source: https://www.v7labs.com/blog/neural-networks-activation-functions)

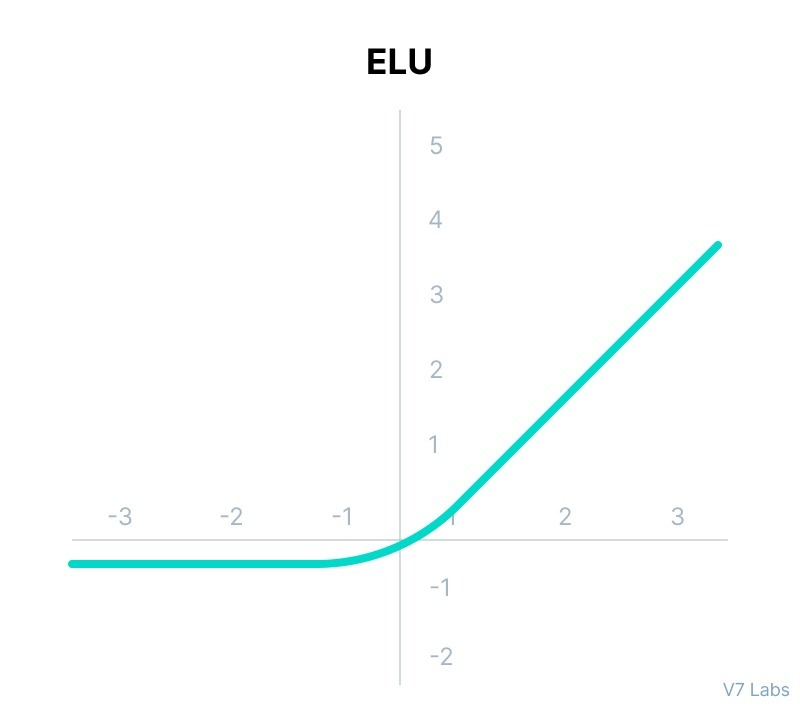


Figure 10: ELU Activation Function

(Source: https://www.v7labs.com/blog/neural-networks-activation-functions)

**2.2.2.8 Softmax Activation Function**

The softmax activation function is a commonly used activation function in machine learning, particularly in the output layer of neural networks. It is used to convert a vector of real numbers into a probability distribution. The softmax function takes as input a vector of real numbers, denoted by , and outputs another vector of the same length, denoted by . The component of the output vector is given by:

Where indicates the element of the vector and indicates the i-th element of the output of the softmax function. The softmax function exponentiates each element of the input vector and normalizes the resulting vector so that the sum of its elements is equal to 1. The output of the softmax function can be interpreted as a probability distribution over N possible outcomes. The softmax function is often used in classification tasks where the goal is to assign an input to one of several categories. In this case, the output of the softmax function can be interpreted as the probability that the input belongs to each of the categories. The category with the highest probability is chosen as the predicted output.

**2.2.3 Output Layer and Loss Function**

The output layer is always the final layer in any neural network which is used for producing the output or prediction for a given input. The output layer may contain one or more neurons. The choice of the number of neurons largely depends on the type of problem the neural network is to solve. The output layer and the Loss function work hand-in-hand to produce a better output or prediction. Also the choice of loss function to the used in any neural network also depends largely on the type of problem to be solved. They loss function is a mathematical function that measure the difference between the predicted output of the neural network and the actual output of the target value. Loss function is used to calculate the error or costs of the neural network prediction. Regression and classification problems common problems solved by neural networks.

**2.2.3.1 Regression Tasks**

Considering regression problems, the neural network needs its output layer to output continuous values. Squared error sometimes called mean squared error (MSE) is a common loss function used to measure the difference between the predicted output and actual values when solving a regression problem. The mean squared error is calculated as the average of the squared difference between the predicted and actual value for each data point as shown in figure 11.

This can be achieved by performing affine transformation without the non-linear activation. This can be formulated mathematically as, assuming a set of n input data (or data from previous layers) and a layer of linear units outputs a vector as:

Where and are learned parameters. The squared loss function can be used to measure the difference between the predicted value and the actual value as:

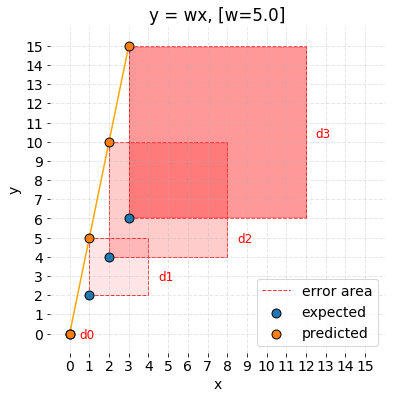


Figure 11: Mean Squared Error

(Source: https://www.medium.com/dev-genius)

**2.2.3.2 Classification Tasks**

In neural networks, classification tasks involves predicting the class or category of an input based on a set of input data or features. The output of a classification model produce a discrete probability distribution over the possible classes which indicates the likelihood of each class for a given input instead of producing a discrete output which indicate the predicted label of a given sample. Neural networks can be used for both binary classification where the output is one of two possible classes and multiple class classification where the output is one of several possible classes. For this two scenario difference output layer and loss function are used.

**A. Binary Target Classification**

Binary classification is a type of machine learning parks where the goal is to classify data into two distinct classes as either 0 or 1. In neural networks, binary classification is often performed using a special type of activation function called the sigmoid activation function. This activation function as discussed earlier, is a mathematical function that takes any input value and maps it to a value between 0 and 1 which interprets the output of the Sigmoid function as the probability of belonging to one of the two classes. To perform this task, A linear layer to transform the input data or data from the previous layers into a single value is needed first thereafter, play Sigmoid activation function is applied. This process can be mathematically modelled as:

Where and are parameters to be learned; denotes the probability of predicting the inputs data that have a label of 1 while denotes the probability of predicting the input data that have a label of 0. To measure the difference between the predicted output y and the actual output y while the binary cross entropy loss function is employed as:

During the inference, if the predicted output probability is greater than 0.5, the input data can be classify as belonging to class 1 otherwise they input data can be classify as belonging to class 0.

**B. Categorical Targets Classification**

Categorical classification is a type of machine learning problem which the goal is to classify data into multiple distinct classes, usually represented as categorical labels. Categorical classification can be called n-class classification tasks. The assumption here is that the actual output y is denoted or represented as integers between 0 and n - 1. One-hot vector which is used to denote the label where is given as 1 (i.e this indicates that the samples is labelled as ). In neural networks, n-class classification prediction is often performed by using a linear layer to transform the input data or data from the previous layer to a n-dimensional vector z as:

Where and are learned parameters. Then a special type of activation function called softmax activation function is applied to normalize the linear transformed input data into a discrete probability distribution over the classes as in equation 13.

To measure the difference between the predicted output y and the actual output, cross entropy loss function can be employed as:

During the inference, new input data is predicted into one of the possible classes label based on its probability distribution output, if is the highest probability of value among all output units.

**2.3 Convolutional Neural Networks**

Convolutional neural networks or CNNs, a widely used type of non-metal that is particularly effective in handling data arranged in grid-like structure, such as images. CNNs shares many similarities with feed-forward neural network which includes the use of trainable weights and biases within each neurons to transform and receive information from the previous layer. However, CNNs, some neurons are designed differently from those in feed-forward networks with the convolution operation used to create specific neuron designs. These neurons are part of the convolutional layers, which typically only connects two small number of neurons in previous layers, resulting in Sparse connections between layers. In addition to Convolutional layers, CNNs also make use of pooling layers which summarize the output of nearby neurons to generate a new output.

**2.3.1 Convolutional Layers**

The convolutional layer forms the foundational building block of a convolutional neural network and it is responsible for the majority of the computation in CNNs. This layer consists of several components, including the input data, a set of filters (kernels) and output feature maps. Let consider a coloured image with N x N pixels, each pixel is associated with three values representing the intensity of red, green and blue respectively. The three colours correspond to the three channels of the input image. Which channel is determined by the element of the vectors at all position of the inputs and the length of each vector at a given position (pixel) is the number of channels. The convolution operation usually involves three dimensions but only slides across two dimension (that is not across the channel dimension). Furthermore, in typical convolutional layer, multiple kernels are applied in parallel to extract features from the input layer. As a result, the output layer is also in multiple channels where the results for each kernel corresponds to each output channel. For example, if the image I has L channels then the convolution operation with P kernels can be written mathematically as:

Where is the kernel with . L parameters which the output give P channels.

To reduce the computational complexity of the convolutional operation, it is often possible that's some positions in the input can be skipped when sliding the kernel. This approach is often referred to as strided convolution, where the convolution operation is only executed at every s positions with s denoting the stride. Figure 12 shows an illustrative example of strided convolutions where the stride is s= 2. The mathematical equation of stride convolution with stride S can be represented as:

When the stride is set to s=1, this means that the strided convolution operation is equivalent to non-strided convolution operation in equation 20. The concept of zero-padding is commonly applied in other to maintain the size of the output. The output size is determined by the padding size, the receptive field (kernel) size and the stride, assuming that the input size remains fixed.

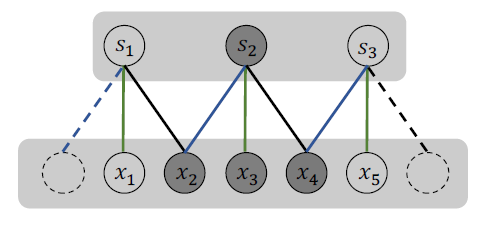


Figure 12: Strided Convolution

(Source: <https://cse.msu.edu/~mayao4/dlg_book/>)

Let consider this, if a 1-D input of size N, with padding size Q, kernel size F and stride size s then the output size can be computed using this formula:

**Work example:** let the input size of the strided convolution be N=5 as shown in Figure 12 with a kernel size of F=3, Zero-padding is Q=1 and stride size s=2, the output can be calculate as:

**2.3.2 Non-Linear Activation Layer**

Same as with feedforward neural networks, each unit in a convolutional neural network is passed to a nonlinear activation function after the convolution operation. The Rectified linear unit (ReLu) is a common activation function used in CNNs. This stage of applying non-linear activation is also known as the detector layer or the detector stage.

**2.3.3 Pooling Layer**

Typically, a convolution layer and a detector layer are followed by a pooling layer, which uses a pooling function to summarize the statistics of a local neighborhood and represent it in the output. The pooling layer reduces the width and height of the data but not the depth (number of channels). Max pooling and average pooling are commonly used pooling operations that takes a 2 x 2 local neighborhood as input and generate a single value as output. The max pooling operation outputs the maximum value in the local neighborhood, while the average pooling operation outputs the average value.

**2.3.4 Overall Convolutional Neural Network**

After introducing the convolution and pooling operations next is to present a comprehensive framework for convolutional neural networks (CNNs) with classification task as the downstream objective. The framework illustrated in Figure 13 can be divided into two main components: the feature extraction component and the classification component. The feature extraction component composed of convolution and pooling layers extracts features from the input data. On the other hand, the classification components are uses fully connected feedforward neural networks. The two components are connected by a flattening operation which transforms the multi-channel feature matrices generated by the feature extraction component into single feature vector that serves as input to the classification component. Figure 13 only illustrated a single convolutional layer and a single pooling layers are typically used. Similarly, the feedforward neural networks in the classification component can have multiple fully connected layers.

**2.3 Recurrent Neural Networks**

Sequence of data, where each data sample is represented as a sequence of values, examples of this type of data are common In tasks like speech recognition, machine translation, sentiment classification and question-and-answer task. In machine translation, for instance a sequence of words is given in one language and aim to translate it to another language. In this task both the input and output data are sequential. Similarly, in questioning and answering task, the aim is to give answer to any given question, in this task as well, go to the input and output are also sequential data. In a tux like sentiment classification, which the task is to predict the sentiment of a document or sentence by processing the input sequence but the output is to predict the sentiment class of the input sequence. In all this case mentioned above trying to use standard

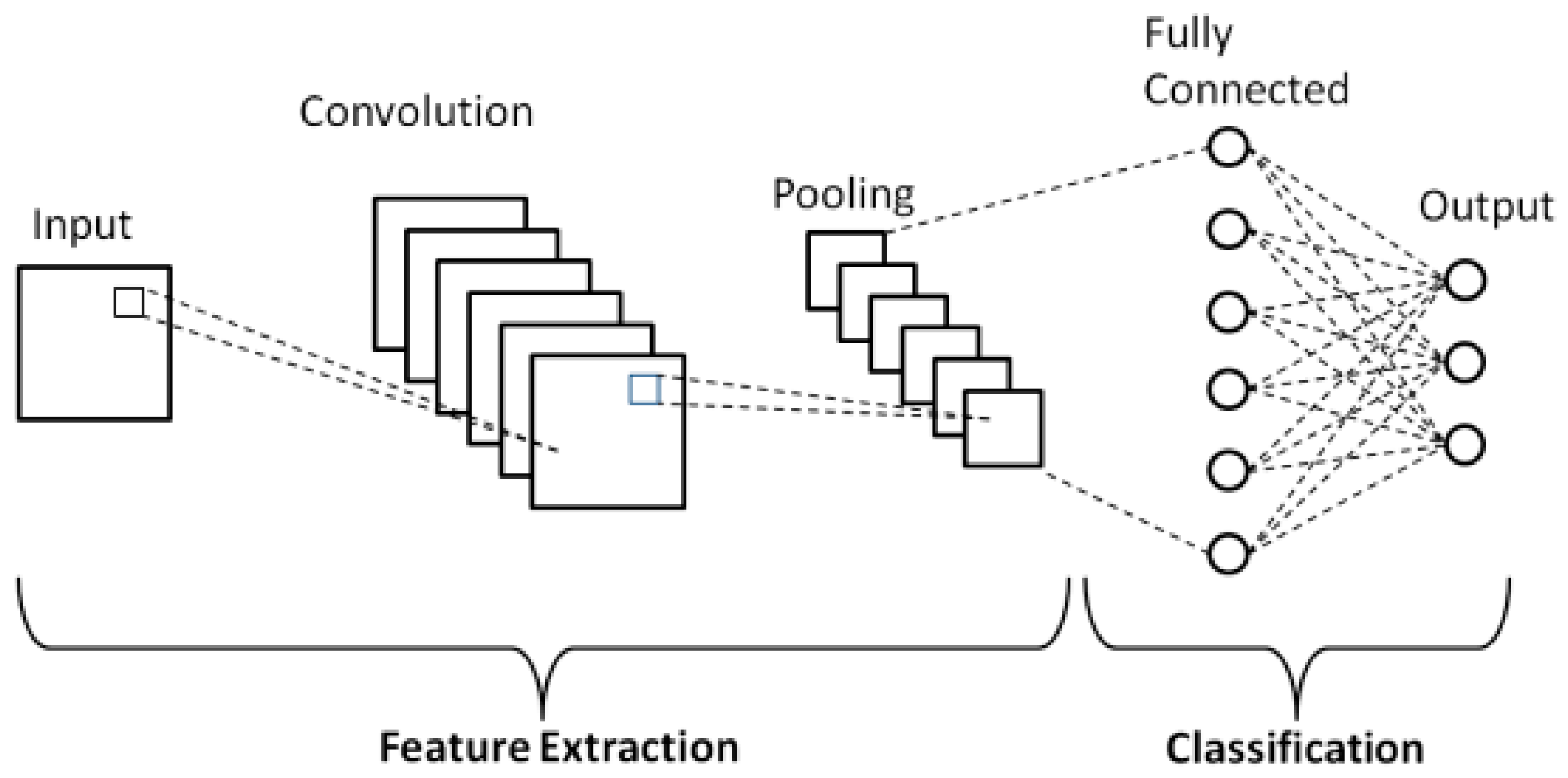


Figure 13: An overall framework of Convolutional neural network

(Source: https://www.v7labs.com/blog/neural-networks-activation-functions)

neural network models to deal with sequential data will fail due to two major limitations: first, the standard neural network models have a fixed input and output size, however sequence can have different length for different data samples which makes it challenging to use the standard neural network models. Another limitation of standard neural network is that it does not share parameters to handle input from various position in the sequence. Considering a language related tasks such as language modelling,

machine translations or sentiment analysis. it is important to capture the contextual information of a sentence. For example given these two sentences:

***Sentence 1:*** "last month, I went to Olumo rock"

***Sentence 2:*** "I went to Olumo rock, last week"

It is expected that the model captures the semantic similarity between these two sentences. One way to achieve this semantic similarity is to use the idea of parameter sharing as seen in the convolutional neural network. The recurrent neural network (RNNs)are powerful tools to handle this to limitations as RNNs allow for parameter sharing and can also handle sequence of various length. When processing is sentence using RNNs, the same set of weights is applied to every element in the sequence one by one which allows the network to learn patterns and dependencies within the sequence. Therefore, the network is capable of capturing the semantic similarity between sentence 1 and sentence 2 even when the ordering of the words are positioned differently. Furthermore, RNNs can process sequence of different lengths by allowing the hidden state of the network to store information from previous time steps. This allows the network to remember information from earlier in the sequence and use it to make prediction at later time steps. As a result, RNNs are widely used in natural language processing tasks where the input length may be of various lengths.

**2.3.1 Traditional Recurrent Neural Network Architecture**

Let be a sequence with the length . As shown in Figure14, the traditional RNN model processes the sequential data one elements at a time using a block of neural networks. The block of neural networks takes not only the current element but also the information passed through the previous block as input, allowing information from earlier positions to flow through the entire sequence. The blocks of neural network in RNNs are typically identical. As shown in figure 14, the RNN model as an output at each position , it is not a requirement for all RNN models to have an output at each position. The block of neural networks in a RNN model has two inputs and two outputs. The outputs produced at each time step is denoted as and the information flowing to the next time step is denoted as . To process the first elements of the sequence; the initial value of is typically sets to 0.

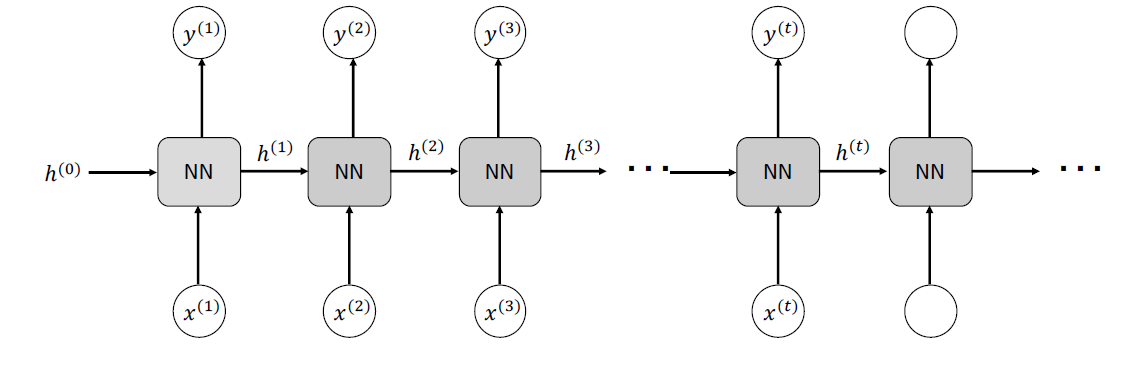


Figure 14: Traditional RNNs architecture

(Source: <https://cse.msu.edu/~mayao4/dlg_book/>)

The procedure of processing the element of the sequence in an RNN can be expressed mathematically as:

Where and are the weight matrices used to perform linear transformations of the input and hidden state, and are the corresponding bias terms, and are the activation functions. Capturing long-term dependencies in sequential data is crucial, as distant elements can often be closely related. However traditional RNN models are not well-suited for this tasks, as gradients things through either vanish or explode over many time steps, making training difficult. The vanishing gradient problem makes it challenging for information in later time steps to impact computations in earlier time steps while the exploding grading can damage the optimization process. To solve these issues, some other RNN variance has been proposed.

**2.3.2 Long Short-term Memory (LSTM)**

The basic structure of LSTM is similar to that of the conventional are RNN model, with a chain structure consisting of neural network blocks that process sequence elements. However, unlike RNN, LSTM deploys the use of gated units to regulate information flow. As depicted in Figure 15a which illustrates how information flows through consecutive positions in a sequence, incorporating the cell state and hidden state . The Hidden state determines how the information is transmitted while the cell state conveys information from the previous state which

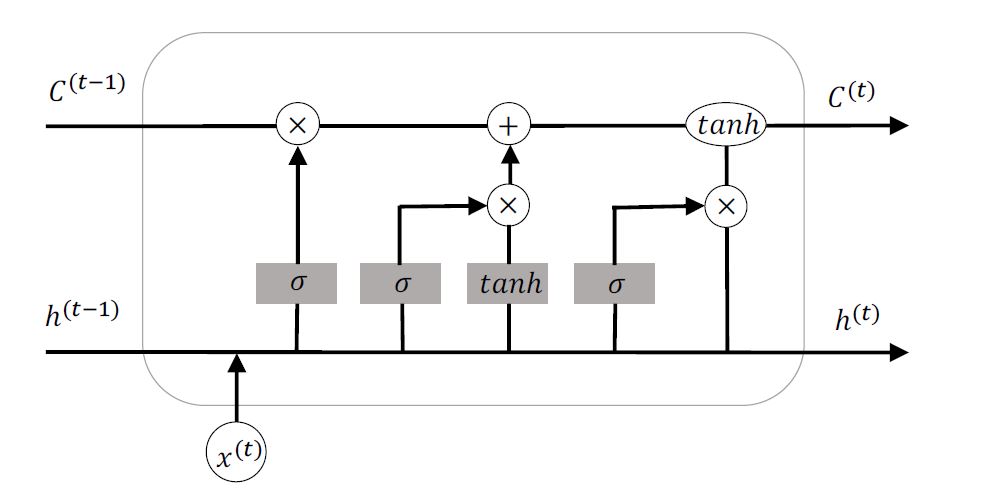


Figure 15a: Block of LSTM architecture

(Source: <https://cse.msu.edu/~mayao4/dlg_book/>)

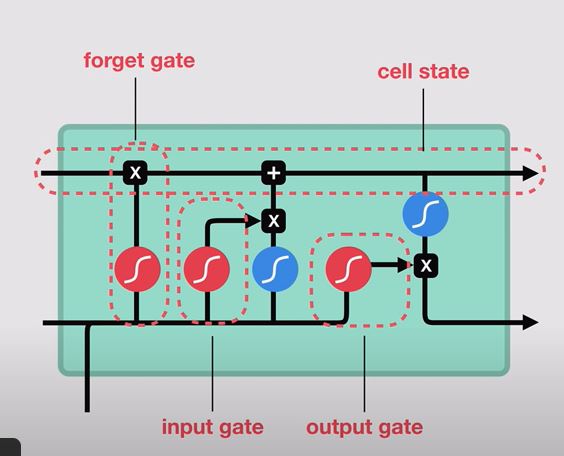


Figure 15a: Gate blocks with LSTM

(Source: https://www.v7labs.com/blog/neural-networks-activation-functions)

are propagated to the next position. The hidden state can also serve as the output of the position as required in sequence-to-sequence applications. In other words, LSTMs incorporate a sequence of gates that controls the flow of information in a sequence data as it flows in and out

of the network. In a typical LSTM, there are four major components: they forget gate, input gate, cell state and output gate as depicted in Figure 15b. Each of these gates acts as a filter and has its own neural network.

**A. Forget Gate**

This gate of the LSTM involves determining which information from the previous cell state should be discarded. The forget gate take into account the previous hidden state and the new input generate a value between 0 to 1 for each element in the cell state . Each value corresponds to an element in the cell state and regulates how the information in the element is discarded. The output can be stored in a vector , which has the same dimensions as the cell state . The forget gate can be expressed mathematically as:

Where and are parameters, is the bias term and is the sigmoid activation function.

**B. Input Gate**

The subsequent stage is to identify which information from the new input should be retained in the new cell state . The input gate functions similarly to the forget gate to make the decision. Input gate can be expressed mathematically as:

**C. Cell State**

To update the cell state, candidate values is required to be generated. Candidate values are formulated as:

Followed by combining the old cell state and the new candidate cell as:

The notation denotes the Hadamard product that is element-wise multiplication.

**D. Output Gate**

The objective is to produce the hidden state , which can proceed to the next position and concurrently serve as the output for this position, if required. The updated cell state serves as the basis for the hidden state, with an output gate determining which aspect of the cell state to maintain. The output gate is structured similarly to the forget and the input gates and can be expressed mathematically as:

Therefore, the new hidden state can be generated mathematically as:

In summary, the entire process of the LSTM is depicted in Figure 15a.

Equations 31 summaries the LSTM process from equation 24-30.

**2.3.3 Gated Recurrent Unit (GRU)**

The simplified gated recurrent unit (GRU) model can be seen in Figure 16a which is essentially a modified version of LSTM. In the GRU model, the forget gate and the input gate are combined into a single update gate. Also the cell state and hidden state are combined into a single state as shown in Figure 16b. These two changes, results in a more streamlined gated RNN model which can be expressed mathematically as:

**A. Update Gate**

**B. Reset Gate**

**C. Output or Hidden State**

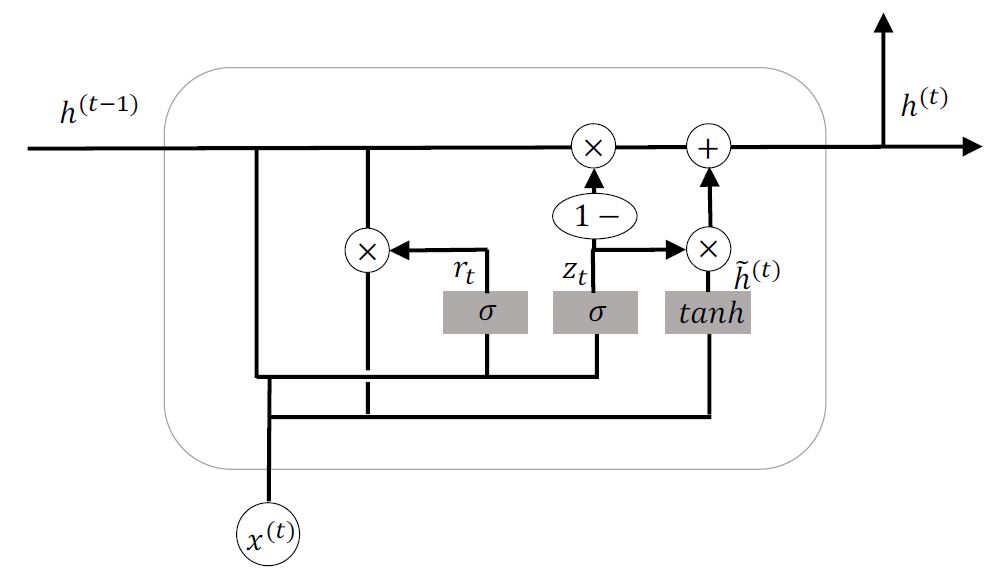


Figure 16a: Block of GRU architecture

(Source: <https://cse.msu.edu/~mayao4/dlg_book/>)

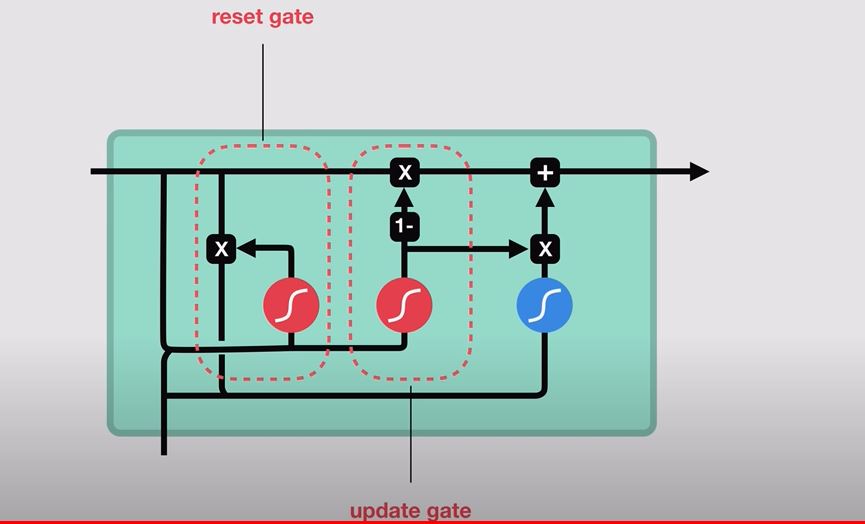


Figure 16b: Gate blocks with LSTM

(Source: <https://www.v7labs.com/blog/neural-networks-activation-functions>)

**2.4 Training Deep Neural Networks**

Training a deep neural network involves iteratively adjusting the parameters of the neural network to minimize the difference between the predicted output and the actual output for a given set of inputs. This process is commonly called optimizing the network. This process is done using two great algorithms which are the backpropagation and optimization algorithms. In this section, the backpropagation algorithm will be discussed briefly while the optimization algorithm will be discussed in respect to gradient descent and its common variants.

**2.4.1 Backpropagation Algorithm**

To perform gradient-based optimization, calculating the gradients with respect to all parameters is a crucial step. The Backpropagation algorithm is an efficient way to compute these gradients, utilizing dynamic programming. It involves two phases: the Forward Phase, where inputs are fed into the deep model and passed through the layers to calculate outputs using the current set of parameters, which are then used to evaluate the loss function; and the Backward Phase, which computes the gradients of the loss function with respect to the parameters. These gradients are dynamically calculated in a backward direction using the chain rule, starting from the output layer.

**2.4.2 Optimization Algorithms**

Research in the field of optimization has a long history and its evolved constantly. Using the gradient information, optimization methods can be divided into ﬁrst-order and high-order optimization methods (Sun et al., 2019). The goal here is to minimize a loss function with respect to the parameters to be learn when training the deep neural networks. Let denotes the loss function where denotes all parameters to be optimized. To minimize the loss function in deep learning, Gradient Descent and its variations are widely used.

**A. Gradient Descent Training**

Gradient Descent is a first-order iterative optimization algorithm that updates the parameters at each iteration. This update involves taking a step in the direction of the negative gradient, which can be expressed as follows:

where is the learning rate which is commonly fixed small constant and represents the gradient of the loss function. As shown in Figure 17, Typically, the loss function is expressed as a summation of many function over a set of training samples. Thus, it can be written in the following form:

where is the loss function for the sample and represents the number of samples. The gradient descent process described here can be called the batch gradient descent which could be very expensive in both space and time complexity to train.

**B. Stochastic Gradient Method**

(Sun et al., 2019) conducted a survey research that highlighted the use of stochastic gradient descent (SGD) as an alternative to the computationally expensive batch gradient descent. The stochastic gradient method was first proposed by Robbins and Monro in 1951(Robbins & Monro, 1951). The main idea behind SGD is to update the gradient using one random sample per

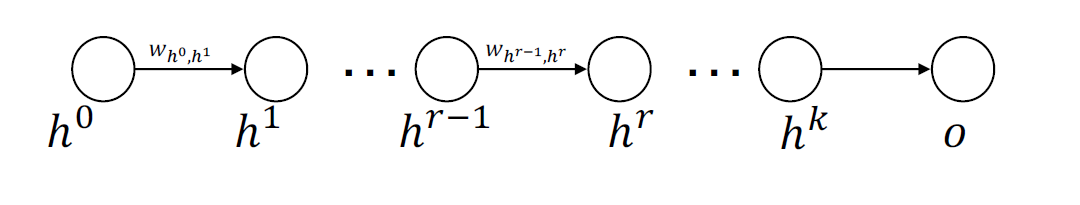


Figure 17: A collection of neurons derived from successive layers

(Source: <https://cse.msu.edu/~mayao4/dlg_book/>)

iteration, which can significantly reduce computational complexity, especially for large-scale data. Several studies have shown that SGD can achieve convergence speed in large numbers of samples and is independent in nature (Johnson & Zhang, 2013; Le Roux et al., 2012).

The algorithm repeatedly updates each parameter using a learning rate .

With the use of only one sample per iteration, the computation complexity for each iteration is O(W), where W represents the number of features. As a result, training the model with only a thousand samples in a dataset with hundreds of thousands of samples is feasible. When N, the number of samples, is large, the update rate for each iteration of stochastic gradient descent (SGD) is faster than that of batch gradient descent. Thus, SGD can significantly decrease the computational complexity and speed up the convergence process. One of the major drawback of this method is that solution to some problems can be trapped at a saddle point.

**C. Mini-batch Gradient Descent Method**

Stochastic Gradient Descent may get stuck at a saddle point. To overcome this issue, (Robbins & Monro, 1951) proposed a technique called mini-batch gradient descent (MSGD). According to (Ruder, 2016), MSGD employs 50 to 256 independently and identically sampled data points to update parameters in each iteration. This approach reduces the gradient variance and provides more stable convergence. Due to these advantages, MSGD is highly likely to attain the global optimal solution for complex problems. Some of the limitations of Mini-batch Gradient Descent is that the learning rate must be chosen carefully. A too small learning rate might lead to very slow convergence and a too large learning rate make it difficult to convergence.

Several other gradient descent variations have been developed for deep neural network training, including Adagrad algorithm (Duchi & Singer, 2011), Adadelta algorithm (Zeiler, 2012), and Adam algorithm(Kingma & Ba, 2014). These techniques usually exhibit superior convergence rates when compared to conventional gradient descent methods.

**2.4.3 Preventing Overfitting**

Overfitting is a common issue in machine learning where a model becomes too complex, resulting in a high variance. It means that the model has learned the training data too well and may fail to generalize well to new, unseen data. Here are some techniques that can be employed to prevent overfitting:

**A. Cross-validation**

Cross-validation involves splitting the data into multiple parts to validate the model's performance on unseen data. This approach can help to identify overfitting by assessing the model's performance on different data subsets.

**B. Regularization**

Regularization is a technique that involves adding a penalty term to the loss function to discourage the model from overfitting. L1 and L2 regularization are two popular methods.

**C. Dropout**

Dropout is a regularization technique that randomly drops out some neurons during training to prevent them from co-adapting and overfitting.

**D. Early stopping**

Early stopping is a technique that stops the training process before the model starts overfitting. The training is stopped when the validation error starts to increase, indicating that the model is starting to overfit.

**E. Data Augmentation**

Data augmentation involves creating more training data by applying transformations such as rotation, scaling, or flipping. It can help to prevent overfitting by providing the model with more diverse and representative data.

By applying these techniques, it is possible to prevent overfitting and create models that generalize well to new data.

**2.5 Sequence-to-Sequence Models**

Sequence-to-sequence models is broader call of models that include all models that maps on a

sequence to another sequence. Seq2seq types of neural network architecture commonly used in natural language processing NLP tasks such as machine translation text summarization speech recognition and so on. The basic idea of seq2seq models is to take a length of input sequence and generate a length of output sequence. A good example to explain it is when thinking of computer programs, computer programs can be thought of as something that takes in a sequence of input bits, then outputs a sequence of output bits. Therefore, computer programs can be seen as sequence-to-sequence model. Although this is not the most intuitive way to express it.

Seq2seq model was an approach proposed by (Sutskever et al., 2014) and (Cho et al., 2014). The architecture of seq2seq models consists of two parts: the encoder and the decoder. The encoder takes in the input sequence and generate a fixed-length vector representation of the input, called the "context vector". The decoder then takes in the context vector and generate the output sequence one token at a time at each step of decoding, the decoder uses the perviously generated token and the context vector to predict the next token in the output sequence.

There are three major approaches adopted for seq2seq modeling in literature which are the recurrent neural network RNN-based seq2seq model, seq2seq model with attention mechanism and the transformer-based model.

**2.5.1 RNN-based seq2seq Architecture**

The first version of the model was applied to NLP domain which consist of an encoder and a decoder where both are based on recurrent neural network (RNN). When applied to NLP tasks, the model is similar to language modeling problem of calculating the conditional probability of the next word in a sequence.

In case of seq2seq model, it assigns probabilities to sequence of words based on some conditioning context vector.

This model was introduction by (Sutskever et al., 2014) to map a fixed length input with a fixed length output where the length of the input and output may differ. For example, translating “I am going to the market” from English to Yoruba “mo lo si Oja”. It can be observed that the English sentence has 6 words while its Yoruba translation has 4 words. This shows clearly that regular RNNs model cannot be used to map tasks like this. As shown in Figure 18, the complete architecture of the RNN-based seq2seq model which can be divided into three parts: Encoder, Encoder vector (context vector) and Decoder. The general idea of this architecture is that the encoder is a RNN based model that encodes the information from a source sentence as a vector of real-valued number called encoder vector (or sometimes called context vector) then the decoder which is also another RNN based model is used to predict or decodes information into a target sentence.

**A. Encoder**

In a recurrent neural network with multiple layers, an input token is received at each time step, which is used to gather relevant information and generate a hidden state. The specific type of RNN employed will determine how this is achieved. For instance, using traditional RNN, LSTM or GRU, the unit combines the present hidden state with the input, resulting in an output that is then discarded, and a new hidden state is generated. The hidden states are computed using the traditional RNNs formula in equation 24 which can also be rewritten as:

Also, The hidden states can be computed using the LSTM or GRU cells formula in equation 30 and 34 respectively.

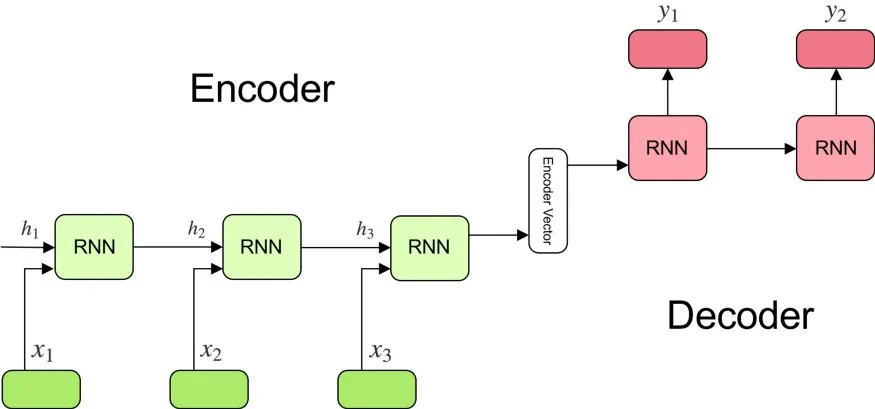


Figure 18: Encoder-decoder sequence to sequence model

(Source: [Understanding Encoder-Decoder Sequence to Sequence Model | by Simeon Kostadinov | Towards Data Science](https://towardsdatascience.com/understanding-encoder-decoder-sequence-to-sequence-model-679e04af4346))

**B. Encoder Vector (Context Vector)**

The encoder vector refers to the final hidden state of the encoder, which aims to encapsulate as much relevant input information as possible to aid the decoder in achieving optimal outcomes. This vector represents the sole input information that the decoder has access to.

**C. Decoder**

A series of recurrent units (traditional RNN, LSTM or GRU), are stacked on top of one another, with each unit generating an output during a specific time step. The initial unit's hidden state serves as the encoder vector, and each subsequent unit receives the previous unit's hidden state. The output is then determined by applying a softmax function, resulting in a probability value for each token in the output vocabulary. The hidden states is computed using the same formula as equation 24 for the traditional RNN, equation 31 for the LSTM and equation 35 for the GRU while the output at time step t is computed as:

The hidden state in conjunction with its corresponding weight , outputs are computed. To generate a probability vector for determining the ultimate output, Softmax activation function is applied.

**2.5.1.1 Generating Output**

After appling the softmax activation function, next step is to chose an output. In chosing the output token from the output vocabulary these three methods are adopted:

**Random Sampling:** Randomly select an output from the probability distribution.

**Greedy 1-best search:** Find the output that maximizes the probability distribution.

**n-best search:** find the n output with the highest probabilities according to the probability distribution.

**2.5.2 Bidirectional RNN Encoder-Decoder**

A Bidirectional RNN Encoder-Decoder is a type of neural network architecture used in natural language processing (NLP) tasks such as machine translation, summarization, and sentiment analysis. the bidirectional RNN encoder-decoder architecture has been shown to be effective in improving the performance of various NLP tasks, especially when dealing with long input sequences. This model was proposed by (Bahdanau et al., 2014) which was used for machine translation task (English-to-French).This model made use of two encoders: one traveling forward and the other on traveling backward over the input sentence which were later combined into the hidden states for the decoder RNN as shown in Figure 19. It can be expressed mathematically as:

Where represent the forward states and represents the backward states of the bidirectional RNN which can be computed as equation 44 and 45 respectively.

Where can be the traditional RNN, LSTM or GRU unit. Both the forward and backward state are combined by simply concatenating the two final vector.

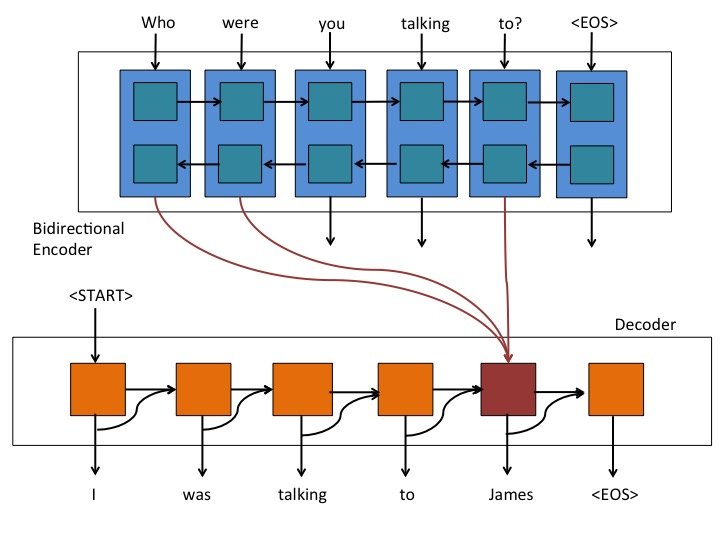


Figure 18: Bidirectional RNN Encoder-Decoder model

(Source: <https://www.researchgate.net/profile/Carlos-Segura-6/publication/332549468/figure/download/fig1/AS:750102306119680@1555849734811/Left-Encoder-decoder-with-RNNs-Right-Encoder-decoder-with-RNNS-and-attention.jpg>)

**2.5.2 Encoder-Decoder with Attention Mechanism**

RNN-based models have an issue when dealing with long sequences since the info from earlier tokens can be lost or weakened as more tokens are processed. The context vector has the difficult task of encoding info from the source sentence into a relatively small vector. This makes it hard for the models to manage long sentences. To solve this problem a solution was proposed in (Bahdanau et al., 2014; Luong et al., 2015a) which introduce a technique called attention to the encoder-decoder model, this model helps improve the quality of machine translation tasks. The attention mechanism enables the model to focus on the applicable parts of the input sequence as necessary, accessing all the prior hidden states of the encoder, not just the last one. During each step of decoding, the decoder can inspect any particular state of the encoder and selectively extract certain elements from that sequence to generate its output. As shown in Figure 19, the model can be divided into three broad parts: Encoder layer, Attention layer, Decoder layer. The encoder uses a bidirectional RNNs layers which as explained later. The most relevant point here is the addition of the attention layer which consist of three components: Alignment vector, Softmax calculation and Context vector calculation.

**A. Alignment vector**

At each timestep of the decoders operation, a vector with the same length as the input sequences is computed; this alignment vector holds scores or probabilities of each word in the source sequence, letting the decoder know which elements to focus on. The idea is the pass the hidden state of both the encoder and decoder into a single hidden layer based on feedforward neural network which can be given mathematically as:

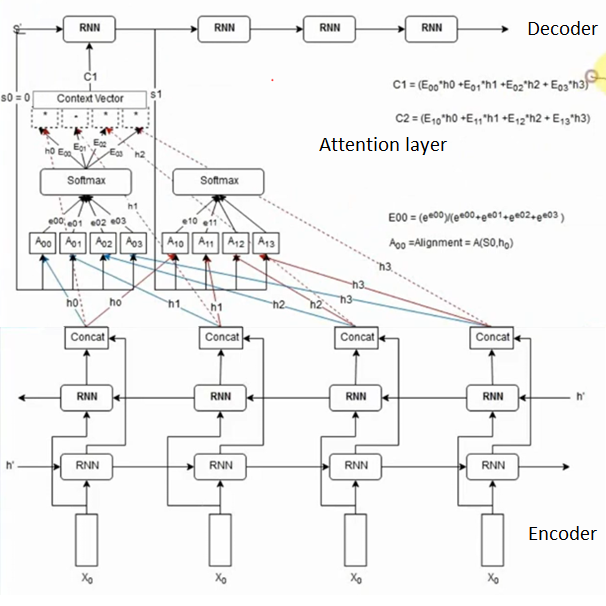


Figure 19: Encoder-Decoder with Attention model

(Source: <https://ai-leader.com/deep-learning/>)

Where represent the hidden state of the encoder and represent the hidden state of the decoder. From figure 19, the alignment score are denoted as . This alignment score can be calculated in different way which some of these ways will be lighted in table 1.

Table 1: Alignment score

|  |  |  |
| --- | --- | --- |
| **Function** | **Equation** | **References** |
| Similarity |  | (Graves et al., 2014) |
| Dot product |  | (Luong et al., 2015) |
| Scaled dot product |  | (Vaswani et al., 2017) |
| General |  | (Luong et al., 2015) |
| Biased general |  | (Sordoni et al., 2016) |
| Activated general |  | (Ma et al., 2017) |
| Generalized kernel |  | (Likhosherstov et al., 2021) |
| Concat |  | (Luong et al., 2015) |
| Additive |  | (Bahdanau et al., 2014) |

Note: denotes length of input, denotes activation function, ( ) denotes trainable parameters and denotes trainable bias term.

**B. Softmax Calculation**

To calculate the softmax for each time step of the decoder. From Figure 19, . So, the softmax can be calculated as:

Same formula can be used for

**C. Context Vector**

The context vector is the weighted average sum of the encoders output and it softmax calculation which can be given as:

is the context vector which will be passed into RNN model of the time step 1. Equation 47 can be used for other context vector.

**2.6 The Transformer Model**

Attention mechanisms have become essential for successful sequence modeling and transduction models for a range of tasks. They allow for the reflection of relationships between input or output sequences without paying attention to their distance. Nevertheless, these attention mechanisms are usually associated with recurrent networks. The transformer model was proposed by (Vaswani et al., 2017) in a paper “Attention is all you need”. The Transformer - a model architecture that deliberately avoid using recurrent neural network and instead relies solely on an attention mechanism to draw global connections between input and output in order to boost the speed with which these models can be trained. This model offers much more potential for parallelization. This model is based on the encoder-decoder architecture which was used widely for machine translation task. The encoder converts an input sequence into a sequence of embedding vectors called context vector and the decoder uses the context vector from the encoder to generate an output sequence one element at a time iteratively. As illustrated in Figure 20, the overall architecture of the transformer model uses stacked self-attention and point-wise, full connected layers for the encoder and the decoder. Some other components are also used to characterize the transformer architecture such as input text, token embeddings, positional embeddings, stack of encoder layers or blocks and stack of decoder layers or blocks as shown in Figure 21.

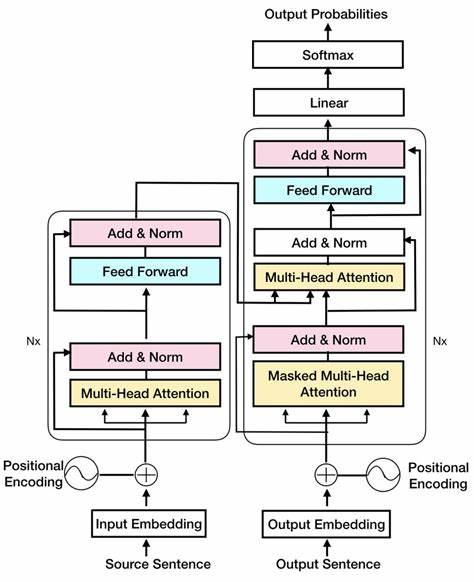


Figure 20: Transformer-model Architecture

(Source: [Transformer Architecture - Bing images](https://www.bing.com/images/search?view=detailV2&ccid=xYEO%2Ftct&id=BB2161B9EEE3660330755A82E660BBF8EE81CB04&thid=OIP.xYEO_tctV2-SlaITOn3XSgHaJG&mediaurl=https%3A%2F%2Fwww.researchgate.net%2Fprofile%2FMd_Arid_Hasan%2Fpublication%2F338223294%2Ffigure%2Fdownload%2Ffig2%2FAS%3A841443144900609%401577627087767%2FTransformer-Encoder-Decoder-architecture-taken-from-Vaswani-et-al-9-for-illustration.jpg&cdnurl=https%3A%2F%2Fth.bing.com%2Fth%2Fid%2FR.c5810efed72d576f9295a2133a7dd74a%3Frik%3DBMuB7vi7YOaCWg%26pid%3DImgRaw%26r%3D0&exph=1045&expw=850&q=Transformer+Architecture&simid=608033349442370647&form=IRPRST&ck=C2B116E77C2E72B89A3329F841DCA1DE&selectedindex=2&ajaxhist=0&ajaxserp=0&vt=0&sim=11))

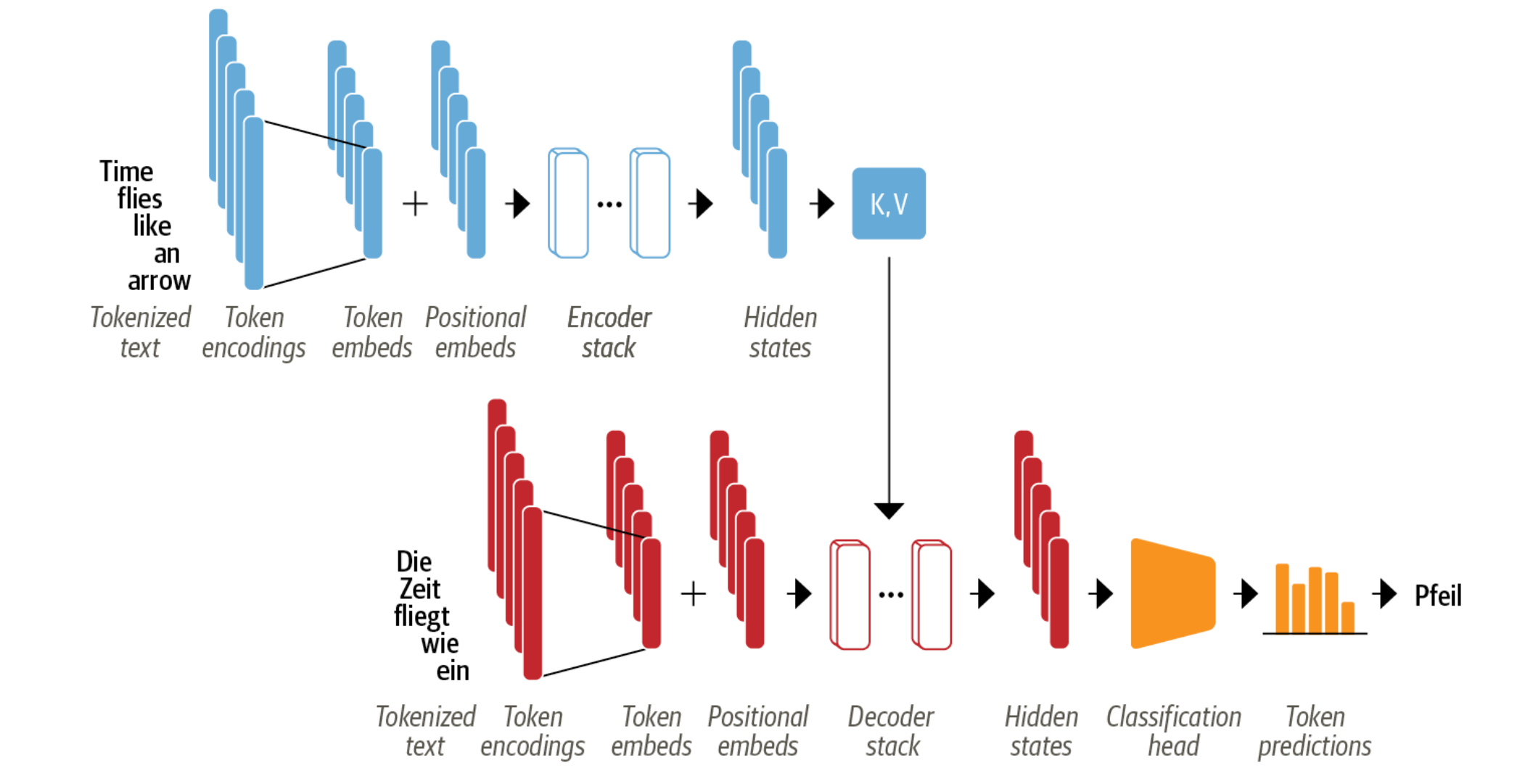


Figure 21: Encoder-Decoder Architecture of Transformer

(Source: [notebooks/03\_transformer-anatomy.ipynb at main · nlp-with-transformers/notebooks · GitHub](https://github.com/nlp-with-transformers/notebooks/blob/main/03_transformer-anatomy.ipynb))

**3.6.1 Encoder for Transformer**

From the paper “Attention is all you need” (Vaswani et al., 2017) as shown in the left halves of Figure 20, the encoder component consists of 6 identical layers stacked on top of each other. Each of these encoder layer can be divided into two sublayers: multi-head self-attention layer and position-wise fully connected feed-forward layer. Each sublayers uses skip connections and layer normalization.

**2.6.1.1 Self-Attention**

Attention is a mechanism that allows neural networks to assign a different amount of weight or attention to each element in a sequence. For the “self” part of the self-attention refers to the fact that these weights are computed for all hidden states in the same set. The main idea behind the self-attention is that instead of using a fixed embedding for each token, whole sequence can be used to compute a weighted average of each embedding. Mathematically, the self-attention model can be formulated as say: given a sequence of token embeddings , to produces a sequence of new embeddings , wher each is a linear combination of all such as:

is the attention weights which are normalized as .

Consider this sentence “time flies like an arrow”. One may think of “flies” as an annoying insects but when the model is processing the word “flies”, self-attention allows it to look at other positions in the sentence for clues that can help lead to a better encoding for the word.

**A. Self-Attention in Detail**

In Self-attention, three vectors are created from each encoder’s input (word embedding). For each word, a query vector , a key vector and a value vector is created. These three vectors can be created by multiplying the word embedding of each word , ) by three different weight matrices that can be trained .

**B. Scaled Dot-Product Attention**

At this stage, the self-attention is to calculate an attention score of each word in the input sentence against all words in the sentence. By so doing, a particular attention **“Scaled Dot-Product Attention”** (Vaswani et al., 2017) as shown in Figure 22. Mathematically:

is the dimension of the key vector .

**C. Multi-Head Attention**

Multi-head attention is an improvement of the self-attention layer which expand the transformer model ability to focus on different positions and also gives the attention layer multiple representation subspaces. The results of the multi-head attention layer is illustrated in Figure 22b. This means that the softmax of one head tends to focus on mostly one part of similarity, so having several heads allows the model to focus on several aspects at once as seen in computer vision models.

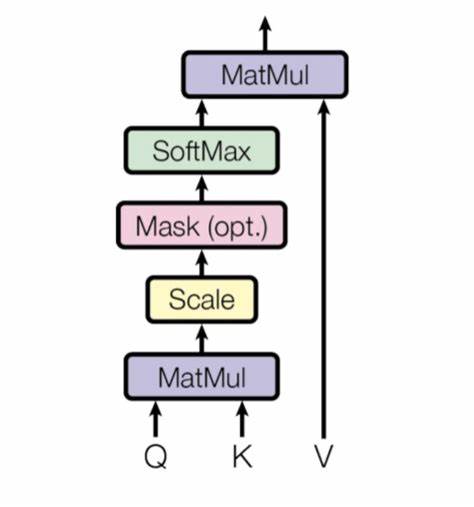


Figure 22a: Scaled Dot-Product Attention

(Source: <https://th.bing.com/th/id/OIP.glkmu3PCc8oISSO43-lEdgHaH7?pid=ImgDet&rs=1>)



Figure 22b: Multi-Head Attention

(Source: <https://th.bing.com/th/id/OIP.glkmu3PCc8oISSO43-lEdgHaH7?pid=ImgDet&rs=1>)

Where , was set to be 8 from the original paper.

**2.6.1.2 Position-wise Feedforward Layer**

The feedforward sublayer in the encoder and decoder is just a simple two-layer fully connected neural network applied to each position separately and identically. This step consists of two linear transformation and GELU activation function is mostly used from literatures.

Note that the first hidden layer size is four times the word embedding size.

**2.6.1.3 Positional Encoding**

Since the transformer model do not contain any recurrence and convolution, it is important to account for the order of the words in the input sequence. In order to address this, the transformer adds some additional information about each input embedding position called positional encoding. This positional encoding is added to each input embedding. Both the input embedding and the positional encoding are of the same dimension. The sine and cosine functions of different frequencies are commonly used which its mathematically equations are:

**2.6.2 Decoder for Transformer**

The main distinction between the decoder and encoder is that the decoder has two attention sublayers as depicted in the right side of Figure 20.

**A. Masked Multi-Head Self-Attention Layer**

A Masked Multi-Head Self-Attention Layer ensures that the tokens produced at each timestep are only based on prior outputs and the current tokens being predicted. Thus, it prevents the decoder from simply copying target translations and making training too easy.

**B. Multi-Head Attention Layer**

Furthermore, a Multi-Head Attention Layer employs multiple heads to attend to the output keys and values of the encoder stack, utilizing intermediate representations of the decoder as queries so as to understand how tokens from two different sequences (such as two languages) are related. As a result, every block allows for access to both encoder keys and values by the decoder.

**2.7 Vanilla Self-Attention with Example**

This section aims to simplify the self-attention mechanism of the transformer architecture with an example step by step.

**Step 1: Input and the input embedded Layer**

**Table 1:** Simple Dataset

|  |  |
| --- | --- |
| S/N | Text |
| 1 | I can switch between feeling very down and feeling extremely determined. |
| 2 | The weather can change from sunny and warm to stormy and cold very quickly. |
| 3 | She can go from being incredibly shy to becoming incredibly outgoing. |

**Step 2: Vocabulary**

At this stage, we find the total number of unique words in the dataset by combining all the sentences into one string.

**combined\_text** = {I can switch between feeling very down and feeling extremely determined. The weather can change from sunny and warm to stormy and cold very quickly. She can go from being incredibly shy to becoming incredibly outgoing.}

Then, split the dataset into words and remove punctuation.

**words** = ['I, 'can', 'switch', 'between', 'feeling', 'very', 'down', 'and', 'feeling', 'extremely', 'determined', 'The', 'weather', 'can', 'change', 'from', 'sunny', 'and', 'warm', 'to', 'stormy', 'and', 'cold', 'very', 'quickly', 'She', 'can', 'go', 'from', 'being', 'incredibly', 'shy', 'to', 'becoming', 'incredibly', 'outgoing']

After, count unique vocabulary words.

Number of unique vocabulary words: 27

**Step 3: Encoding**

This stage is where numbers (integers) are assigned to the unique word in the dataset.

**Table 3: Word encoding**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **I** | **Can** | **Switch** | **Between** | **Feeling** | **Very** | **Down** | **And** | **Extremely** | **determined** |
| **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| **The** | **Weather** | **Change** | **From** | **Sunny** | **Warm** | **To** | **Stormy** | **Cold** | **Quickly** |
| **11** | **12** | **13** | **14** | **15** | **16** | **17** | **18** | **19** | **20** |
| **She** | **Go** | **Being** | **Incredibly** | **shy** | **becoming** | **outgoing** |
| **21** | **22** | **23** | **24** | **25** | **26** | **27** |

**Step 4: Word Embedding**

Pick a sentence from the dataset:

“She can go from being incredibly shy to becoming incredibly outgoing”

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| She | Can | Go | From | Being | Incredibly | Shy | To | Becoming | Incredibly | Outgoing |
| 21 | 2 | 22 | 14 | 23 | 24 | 25 | 17 | 26 | 24 | 27 |

So, the corresponding integer value attached to each word will be associated with an embedding vector using and word embedding algorithm such as Google word2vec, or Word Golve. The original transformer paper uses 512 dimensions of embedding vector but for this example, a small dimension 7 and random values between (0 and 1) will be used.

**Table 4:** Word embedding vector

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| She | Can | Go | From | Being | Incredibly | Shy | To | Becoming | Incredibly | Outgoing |
| 21 | 2 | 22 | 14 | 23 | 24 | 25 | 17 | 26 | 24 | 27 |
|  |  |  |  |  |  |  |  |  |  |  |

**Step 4: Positional Embedding**

Since the transformer model do not contain any recurrence and convolution, it is important to account for the order of the words in the input sequence. In order to address this, the transformer adds some additional information about each input embedding position called positional encoding. This positional encoding is added to each input embedding. Both the input embedding and the positional encoding are of the same dimension. The sine and cosine functions of different frequencies are commonly used which its mathematically equations as in equation 6.

is the index of word in the sequence, represents whether even or odd, and represents the dimensionality of the word embedding.

Following the example, the first word “she” will be considered and it positional encoding vector will be calculated as presented in Table 4 and the table 5 shows the full word embedding + positional encoding on our dataset.

**Table 5:** Positional encoding

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **I** | **Word embedding for “she”** |  |  | **positional encoding** | **Word embedding + positional encoding** |
| **0** |  | **Even** |  | **1** |  |
| **1** |  | **Odd** |  | **0** |  |
| **2** |  | **Even** |  | **1** |  |
| **3** |  | **Odd** |  | **0** |  |
| **4** |  | **Even** |  | **1** |  |
| **5** |  | **Odd** |  | **0** |  |
| **6** |  | **Even** |  | **1** |  |

Pos = 0 and

**Table 6:** Word embedding + Positional encoding

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| She | Can | Go | From | Being | Incredibly | Shy | To | Becoming | Incredibly | Outgoing |
| 21 | 2 | 22 | 14 | 23 | 24 | 25 | 17 | 26 | 24 | 27 |
| **[1.8823, 0.9150, 1.3829, 0.9593, 1.3904, 0.6009, 1.2566]** | **[1.3339, 1.0127, 1.1332, 0.9350, 1.5936, 0.8694, 1.5677]** | **[0.3250, 0.5728, 1.8853, 0.5746, 1.2666, 0.6274, 1.2696]** | **[-0.5486, 0.5111, 1.8316, 0.1064, 1.2695, 0.3588, 1.1994]** | **[-0.1064, 0.2901, 1.9514, 0.0768, 1.8860, 0.5832, 1.3376]** | **[1.0927, 0.9300, 1.9037, 0.5566, 1.3423, 0.6343, 1.3644]** | **[1.6706, 1.3649, 1.7885, 0.2836, 1.7886, 0.5895, 1.7539]** | **[0.9491, 0.4877, 1.3061, 0.1191, 1.9103, 0.6440, 1.7071]** | **[0.5126, 1.0358, 1.8904, 0.1477, 1.5315, 0.1587, 1.6542]** | **[-0.5833, 1.2566, 1.3947, 0.9181, 1.2036, 0.2018, 1.2018]** | **[0.1106, 1.3257, 1.9798, 0.0911, 1.0041, 0.1088, 1.1637]** |

**Step 4: Multi-head**

The Multi-head Attention operation stands out as a critical and intricate part of the transformer model, featuring multiple essential components illustrated in Figure 3b. This operation plays a fundamental role in both the transformation of the input dataset and the extraction of significant and informative representations from it. The multi-head attention is made up of multiple single head attention layer and a single head attention consists of several key components as shown in Figure 3a. single head attention can also be called Scaled Dot-Product Attention.

For this numerical example, only one single attention head will be considered. From figure 3a, it is observed that the single head attention has three inputs, this means that the Word embedding + Positional encoding output will be divided into three inputs in the attention layer which are Query, key and value.

Where , was set to be 8 from the original paper.

Where , and .

. Where = number of attention head, = dimension of the word embedding.

In the original paper ‘attention is all you need’, the word embedding was 512 dimension and 8 attention head was used.

Therefore, , this means that and

From the numerical example, h = 1 and the word embedding dimension = 7 therefore . , and which are randomly generated.

**Table 7:** Input Embedding Matrix (x)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| She | **1.8823** | **0.9150** | **1.3829** | **0.9593** | **1.3904** | **0.6009** | **1.2566** |
| Can | **1.3339** | **1.0127** | **1.1332** | **0.9350** | **1.5936** | **0.8694** | **1.5677** |
| Go | **0.3250** | **0.5728** | **1.8853** | **0.5746,** | **1.2666,** | **0.6274,** | **1.2696** |
| From | **0.5486,** | **0.5111,** | **1.8316,** | **0.1064,** | **1.2695,** | **0.3588,** | **1.1994** |
| Being | **0.1064,** | **0.2901,** | **1.9514,** | **0.0768,** | **1.8860,** | **0.5832,** | **1.3376** |
| Incredibly | **1.0927,** | **0.9300,** | **1.9037,** | **0.5566,** | **1.3423,** | **0.6343,** | **1.3644** |
| Shy | **1.6706,** | **1.3649,** | **1.7885,** | **0.2836,** | **1.7886,** | **0.5895,** | **1.7539** |
| To | **0.9491,** | **0.4877,** | **1.3061,** | **0.1191,** | **1.9103,** | **0.6440,** | **1.7071** |
| Becoming | **0.5126,** | **1.0358,** | **1.8904,** | **0.1477,** | **1.5315,** | **0.1587,** | **1.6542** |
| Incredibly | **0.5833,** | **1.2566,** | **1.3947,** | **0.9181,** | **1.2036,** | **0.2018,** | **1.2018** |
| Outgoing | **0.1106,** | **1.3257,** | **1.9798,** | **0.0911,** | **1.0041,** | **0.1088,** | **1.1637** |

**Table 8:** Matrix

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **0.3250,** | **0.3631,** | **0.1917,** | **0.5239,** | **0.9504,** | **0.2182,** | **0.5119** |
| **0.9755,** | **0.6294,** | **0.2458,** | **0.4890,** | **0.5144,** | **0.3149,** | **0.8956** |
| **0.3735,** | **0.5572,** | **0.0860,** | **0.7188,** | **0.1738,** | **0.5646,** | **0.6297** |
| **0.9650,** | **0.3490,** | **0.2507,** | **0.5625,** | **0.6428,** | **0.3522,** | **0.6517** |
| **0.4603,** | **0.4590,** | **0.3703,** | **0.2146,** | **0.4434,** | **0.8542,** | **0.2187** |
| **0.2765,** | **0.9188,** | **0.8047,** | **0.7083,** | **0.4500,** | **0.6865,** | **0.9504** |
| **0.1044,** | **0.0474,** | **0.8022,** | **0.6153,** | **0.2265,** | **0.4213,** | **0.0967** |

**Table 9:** Matrix

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **0.6412,** | **0.8290,** | **0.2115,** | **0.5220,** | **0.5978,** | **0.6350,** | **0.3189** |
| **0.5358,** | **0.6423,** | **0.9220,** | **0.4815,** | **0.4033,** | **0.9569,** | **0.7168** |
| **0.4144,** | **0.9267,** | **0.5107,** | **0.9084,** | **0.0851,** | **0.9295,** | **0.3275** |
| **0.5173,** | **0.5714,** | **0.9622,** | **0.0082,** | **0.8716,** | **0.2098,** | **0.3361** |
| **0.8173,** | **0.0998,** | **0.0871,** | **0.5707,** | **0.1677,** | **0.5455,** | **0.2982** |
| **0.9548,** | **0.4646,** | **0.6071,** | **0.2324,** | **0.8601,** | **0.2649,** | **0.6276** |
| **0.6178,** | **0.7995,** | **0.4722,** | **0.8291,** | **0.4607,** | **0.7845,** | **0.3684** |

Table 10: Matrix

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **0.8823,** | **0.9150,** | **0.3829,** | **0.9593,** | **0.3904,** | **0.6009,** | **0.2566** |
| **0.7936,** | **0.9408,** | **0.1332,** | **0.9346,** | **0.5936,** | **0.8694,** | **0.5677** |
| **0.7411,** | **0.4294,** | **0.8854,** | **0.5739,** | **0.2666,** | **0.6274,** | **0.2696** |
| **0.4414,** | **0.2969,** | **0.8317,** | **0.1053,** | **0.2695,** | **0.3588,** | **0.1994** |
| **0.5472,** | **0.0062,** | **0.9516,** | **0.0753,** | **0.8860,** | **0.5832,** | **0.3376** |
| **0.8090,** | **0.5779,** | **0.9040,** | **0.5547,** | **0.3423,** | **0.6343,** | **0.3644** |
| **0.7104,** | **0.9464,** | **0.7890,** | **0.2814,** | **0.7886,** | **0.5895,** | **0.7539** |

**Table 11:** self-attention connectivity output

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | She | Can | Go | From | Being | Incredibly | Shy | To | Becoming | Incredibly | Outgoing |
| She | 0.1523 | 0.1585 | 0.0464 | 0.0145 | 0.0299 | 0.1141 | 0.2963 | 0.0634 | 0.0649 | 0.0259 | 0.0338 |
| Can | 0.1518 | 0.1582 | 0.0457 | 0.0140 | 0.0292 | 0.1141 | 0.3012 | 0.0621 | 0.0647 | 0.0256 | 0.0335 |
| Go | 0.1404 | 0.1439 | 0.0573 | 0.0237 | 0.0414 | 0.1155 | 0.2451 | 0.0726 | 0.0769 | 0.0366 | 0.0466 |
| From | 0.1270 | 0.1286 | 0.0685 | 0.0371 | 0.0550 | 0.1127 | 0.1935 | 0.0807 | 0.0862 | 0.0501 | 0.0607 |
| Being | 0.1353 | 0.1373 | 0.0613 | 0.0280 | 0.0459 | 0.1152 | 0.2271 | 0.0750 | 0.0812 | 0.0414 | 0.0523 |
| Incredibly | 0.1477 | 0.1525 | 0.0497 | 0.0170 | 0.0335 | 0.1151 | 0.2817 | 0.0663 | 0.069 | 0.0290 | 0.0380 |
| Shy | 0.1525 | 0.1586 | 0.0427 | 0.0122 | 0.0266 | 0.1131 | 0.3189 | 0.0593 | 0.0623 | 0.0229 | 0.0309 |
| To | 0.1433 | 0.1470 | 0.0556 | 0.0218 | 0.0388 | 0.1159 | 0.2529 | 0.0701 | 0.0749 | 0.0353 | 0.0446 |
| Becoming | 0.1411 | 0.1448 | 0.0567 | 0.0231 | 0.0408 | 0.1155 | 0.2478 | 0.0722 | 0.0762 | 0.0360 | 0.0458 |
| Incredibly | 0.1362 | 0.1400 | 0.0618 | 0.0284 | 0.0470 | 0.1144 | 0.2227 | 0.0771 | 0.0798 | 0.0415 | 0.0510 |
| Outgoing | 0.1358 | 0.1389 | 0.0620 | 0.0286 | 0.0471 | 0.1146 | 0.2226 | 0.0771 | 0.0803 | 0.0415 | 0.0515 |

The core component of the transformer architecture is the attention mechanism which helps in identifying complicated dependencies among every set of the input sequence as shown in Table 10. The defining trait of the transformer model is its attention mechanism. Attention mechanisms have become immensely popular within the Artificial Intelligence community as indispensable components of neural architectures for a great many applications. The intuition for attention mechanism can be illustrated via the human biological system. Attention is an essential cognitive ability that is vital for humans (Niu et al., 2021). The attention mechanism accounts for this notion of relevance by allowing the model to pay attention to specific parts of the input which would aid in effectively performing a particular task. Unfortunately, the regular transformer is faced with a well-known problem for the attention mechanism which is the quadratic time and memory complexity of . This presents a bottleneck, especially when handling long sequences, as the attention module becomes computationally burdensome. Moreover, the imperative to learn the sequence order directly from the training data increases proneness to overfitting, especially when working with limited or moderately sized datasets. The combination of these factors underscores the need for addressing scalability concerns and mitigating the risk of overfitting in the conventional transformer architecture. In attempts to solve these two problems many attention module has been proposed using different approaches.

**2.8 Related Works on Transformer Models**

There have been numerous models suggested thus far, utilizing the vanilla Transformer as a foundation, which can be categorized into three perspectives: pre-training techniques, modifications to the architecture and applications. Pre-training techniques are majorly built on model which can be used in three different ways: Encoder only model, Decoder only model and Encoder-Decoder model. For the application areas, Transformers has been applied in many areas of research such as Natural Language Processing, Computer Vision, Programming Languages Intelligence tasks, Audio Applications and Multimodal Applications. Another aspect of improving the vanilla Transformer is modifications to the architecture which can also be categorized into four groups: Attention mechanism, Position Encoding, LayerNorm and Feed Forward Network. In this section, a review on pre-trained techniques and the application will be done briefly while that of modifications to the architecture will be discussed in detailed especially on the improvements on the Attention mechanism since that is the main focus of this research.

**2.8.1 Pre-train models with Transformer Models**

Transformer-based models have recently become the go-to solution in natural language processing due to their superior performance compared to recurrent neural networks. The improved performance is mainly attributed to their architecture that grows with training data and model size, allowing them to take advantage of large-scale unsupervised pre-training. This technique, known as transfer learning, lets Transformer models gain an understanding of general features and patterns from expansive corpora before they are fine-tuned with domain-specific datasets. This approach has proven successful for numerous downstream tasks such as text classification, named entity recognition, machine reading comprehension, machine translation and text summarization. The following subsections gives a short overview of three popular pre-trained Transformer-based models from literature.

**2.8.1.1 Encoder only Transformers**

The first encoder-only model based on the Transformer architecture was Bidirectional Encoded Representations from Transformers BERT. It was used meanly for natural language understanding (NLU). Enoder-only models still dominate research area on NLU tasks such as text classification, named entity recognition, machine reading comprehension, machine translation and text summarization. Some of the variants of Encoder-only models are:

**A. Bidirectional Encoded Representations from Transformers (BERT)**

BERT is a pre-trained language model developed by Google that uses the Transformer architecture to generate contextual word embeddings for natural language processing tasks (Devlin et al., 2018). The BERT model is trained on a large corpus of text data using an unsupervised approach, which means that it learns to generate word embeddings without explicit labels or annotations. During training, the model is presented with a sentence or paragraph of text and learns to predict the probability of each word in the text given the surrounding context. BERT uses a bidirectional approach, which means that it takes into account both the preceding and following context of each word when generating its embeddings. This enables the model to capture more complex and nuanced relationships between words and their context, leading to more accurate and contextually appropriate embeddings. In summary, BERT was two objectives: one is to predict masked tokens in texts called *masked language modeling* (MLM) and the other is to determine the next text passage after one called *sentence prediction* (NSP).

**B. DistilBERT**

DistilBERT is a type of language model developed by Hugging Face, a natural language processing company (Sanh et al., 2019). It is a smaller and faster version of the popular BERT (Bidirectional Encoder Representations from Transformers) language model. DistilBERT was developed by training a smaller version of the BERT model using a technique called knowledge distillation, which involves transferring the knowledge learned by a larger model to a smaller model. This makes DistilBERT much faster and more memory-efficient than BERT while maintaining a similar level of accuracy on a range of natural language processing tasks. DistilBERT has been used for a variety of natural language processing tasks, including sentiment analysis, text classification, and question answering. Its smaller size and faster speed make it particularly well-suited for use in real-time applications, such as chatbots or virtual assistants, where response time is critical.

**C. Robustly Optimized BERT Pretraining Approach RoBERTa**

RoBERTa is a language model developed by Facebook AI Research (FAIR) in 2019 (Y. Liu et al., 2019). RoBERTa is an extension of BERT (Bidirectional Encoder Representations from Transformers), another popular language model introduced by Google in 2018. RoBERTa uses a similar architecture to BERT, but it is trained using a larger and more diverse corpus of text data and with longer training times. Additionally, RoBERTa uses a dynamic masking strategy during pre-training that helps it to better understand the relationships between different parts of a sentence. RoBERTa has achieved state-of-the-art performance on a wide range of natural language processing (NLP) tasks, including language modeling, sentiment analysis, question answering, and natural language inference. It has been widely used in industry and academia for various NLP applications.

**D. ALBERT**

ALBERT (A Lite BERT) is a type of transformer-based neural network architecture for natural language processing (NLP) that was proposed by Google researchers in 2019 (Lan et al., 2019). The architecture is designed to improve the efficiency and effectiveness of pretraining large language models by reducing the number of parameters required while maintaining or improving the performance on downstream NLP tasks. ALBERT achieves this by using several techniques to reduce the number of parameters required in the model:

**Factorized Embedding Parameterization**: This technique reduces the size of the embedding matrix by decomposing it into two smaller matrices.

**Cross-layer Parameter Sharing:** This technique shares the parameters between the layers of the model, reducing the total number of parameters required.

**Inter-sentence coherence Loss:** This technique encourages the model to learn relationships between sentences, which improves its ability to understand the context of text.

**Sentence Order Prediction:** This technique encourages the model to learn the order of sentences in a document, which helps it to better understand the structure of text.

ALBERT has been shown to achieve state-of-the-art performance on a wide range of NLP tasks, while using significantly fewer parameters than previous models such as BERT. It has also been shown to be effective in low-resource settings, where large models such as BERT may not be feasible due to memory and computational constraints.

**E. DeBERTa**

DeBERTa (Decoding-enhanced BERT with disentangled attention) is a transformer-based language model that was introduced in 2020 by a team of researchers from Microsoft Research Asia and Peking University (He et al., 2020). It builds on the popular BERT (Bidirectional Encoder Representations from Transformers) model and aims to improve its performance by addressing two key limitations: the inability to handle long sequences and the inability to model the relationships between different parts of the input sequence. DeBERTa achieves this by introducing two novel mechanisms: disentangled attention and enhanced mask decoder. Disentangled attention allows the model to selectively attend to different parts of the input sequence without interference, while the enhanced mask decoder allows the model to handle longer sequences by generating masks that indicate which parts of the input sequence to attend to. Overall, DeBERTa has shown significant improvements in several benchmark datasets, including the GLUE and SuperGLUE benchmarks, and has achieved state-of-the-art performance on a wide range of natural language processing tasks, such as question answering, language inference, and sentiment analysis.

**2.8.1.2 Decoder only Transformers**

These models are exceptionally good at predicting the next word in a sequence and are thus mostly used for text generation tasks. Some of the variants of decoder-only models are as follows:

**A. Generative Pre-Training Transformer (GPT)**

The Generative Pre-Training Transformer (GPT) is a language model architecture that uses unsupervised learning to generate text. It was introduced in 2018 by OpenAI and is based on the Transformer architecture (Alec Radford et al., 2018). GPT is pre-trained on a large corpus of text data using a language modeling objective, which involves predicting the next word in a sequence of text. The pre-training process allows the model to learn the statistical patterns and relationships between words in the text, which enables it to generate coherent and fluent sentences. GPT has been used for a variety of natural language processing tasks, including language translation, question answering, and text generation. It has achieved state-of-the-art performance on several benchmark datasets, and its large-scale versions have been used to generate high-quality text in various applications, such as chatbots and language generation for games. One of the notable features of GPT is its ability to generate text that is coherent and grammatically correct, even when prompted with a few words or a short phrase. This makes it a powerful tool for natural language generation tasks, and it has the potential to revolutionize the way we interact with machines and the internet.

**B. Generative Pre-Training Transformer-2 (GPT-2)**

Generative Pre-Training Transformer-2 (GPT-2) is a language model developed by OpenAI, released in 2019 (Radford et al., 2019). It is based on the transformer architecture, similar to its predecessor GPT-1, but with significant improvements in terms of model size, training data, and performance. GPT-2 has 1.5 billion parameters, making it one of the largest language models at the time of its release. It was trained on a diverse range of internet text, including web pages, articles, and books. The model was trained using a unsupervised learning approach, which means it was not explicitly trained on any specific task, but instead learned to generate coherent text by predicting the next word given a sequence of words. The model's performance on various language tasks, such as question answering, text completion, and language translation, was significantly better than its predecessor GPT-1 and other state-of-the-art models at the time. However, due to concerns about the potential misuse of its text-generation capabilities, OpenAI decided not to release the full version of the model and instead released a smaller version with reduced performance capabilities. Despite this limitation, GPT-2 remains a significant milestone in natural language processing and has been used in a variety of applications, such as chatbots, text summarization, and content generation.

**C. Generative Pre-Training Transformer-3 (GPT-3)**

Generative Pre-Training Transformer-3 (GPT-3) is a state-of-the-art language model developed by OpenAI (Kaplan et al., 2020). It is part of the family of transformers models, which are based on a neural network architecture that can learn to process sequences of words or other tokens and generate new sequences. GPT-3 is trained on a massive amount of text data from the internet, including books, articles, and websites. It has a very large number of parameters, with the largest version having 175 billion parameters. This allows it to generate human-like text with impressive accuracy and coherence. The model can perform a wide range of natural language processing tasks, such as language translation, text completion, and question answering. It has also shown impressive capabilities in natural language generation, including generating realistic and coherent text in a variety of styles and genres. GPT-3 has been used in various applications, including chatbots, writing assistants, and content creation tools. Its impressive capabilities have sparked interest and debate about the potential risks and benefits of large language models and their impact on society.

**2.8.1.3 Encoder-Decoder Transformers**

The encoder-decoder variants of Transformers have novel applications across both natural language understanding (NLU) and natural language generation (NLG) domains although it has become very common to build models using encoder or decoder only architecture. Some of the variants of decoder-only models are as follows:

**A. T5**

The T5 (Text-to-Text Transfer Transformer) is a pre-trained neural language model developed by Google's AI research team, Google Brain (Raffel et al., 2020). It is based on the transformer architecture. The T5 model is capable of performing a wide range of natural language processing tasks, such as language translation, text summarization, question answering, and text completion. One of the unique features of the T5 model is its ability to learn from multiple tasks at once using a technique called multitask learning. This approach enables the model to learn a more generalized representation of language that can be applied to a variety of tasks. In addition to multitask learning, the T5 model also uses a technique called "prefix tuning," which involves adding a task-specific prefix to the input text, allowing the model to learn how to solve a specific task.

**B. BART**

BART stands for Bidirectional Encoder Representations from Transformers. It is a state-of-the-art pre-training method for natural language processing (NLP) developed by Facebook AI Research (FAIR) (Lewis et al., 2019). BART is based on the Transformer architecture, which is a neural network architecture used for processing sequences of data, such as text. BART is trained using a combination of denoising autoencoding and masked language modeling objectives. This means that during training, BART learns to reconstruct text that has been corrupted by random noise, as well as predicting missing words in a sentence. By doing so, BART learns to generate high-quality representations of natural language text that can be fine-tuned for a wide range of NLP tasks, such as machine translation, summarization, and question answering. BART has achieved state-of-the-art results on several NLP benchmarks, and it is widely used in industry and academia for various NLP tasks.

**2.8.2 Architecture Modification**

Architecture Modification can be categorized into four groups according to literatures which are Attention mechanism, Position Encoding, LayerNorm and Feed Forward Network. Attention mechanism will be the main focus since it is the goal of this research and that the attention mechanism is the key component of any Transformer. The primary objective of the majority of these models is to enhance the self-attention mechanism memory complexity. This section presents some improvement on attention mechanism which can be divided into several directions such as sparse attention, low rank attention, kernel attention, local + global attention and memory compression. This section provides a broad detail based on fundamental techniques and basic useful area of different transformer models as shown in Table 2.

**2.8.2.1 Sparse Attention**

This line of work introduces sparsity bias into the attention mechanism due to the quadratic complexity nature of the softmax. So only a few positions are strongly attended to.

**A. Sparse Transformer**

In an effort to reduce the quadratic complexity of the standard self-attention mechanism, the Sparse Transformer (Child et al., 2019) proposes a straightforward approach. It involves converting the dense attention matrix into a sparse version by limiting the computation of attention to only a select few , pairs. The Sparse Transformer employs fixed attention patterns defined by local neighborhoods and strides as illustrated in Table 11 and Table 12 respectively. This method was named as Strided attention which can be defined as:

This formulation is convenient if the data naturally has a structure that aligns with the stride, like images or some types of music. For data without a periodic structure, like text, however, (Child et al., 2019) find that the network can fail to properly route information with the strided pattern. For this reason, a fixed attention pattern was proposed which can be defined as:

where represent weight of , and the denote the floor function.

The complexity only affected the memory by reducing its complexity from to .

**B. Star Transformer**

The Star-Transformer, introduced by Guo et al. [41], is a simplified version of the traditional Transformer model, specifically designed to address its size and data dependency issues. Guo et al. achieved this simplification by implementing a fixed sparse pattern, reducing the model's complexity from quadratic to linear. Unlike the conventional Transformer fully-connected structure, Star-Transformer adopts a star-shaped topology, allowing attention only between

**Table 11: strided-attention connectivity output**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | She | Can | Go | From | Being | Incredibly | Shy | To | Becoming | Incredibly | Outgoing |
| She | **0.1523** | **0.1585** | **0.0464** |  |  | **0.1141** | **0.2963** | **0.0634** |  | **0.0259** | **0.0338** |
| Can | **0.1518** | **0.1582** | **0.0457** | **0.0140** |  |  | **0.3012** | **0.0621** | **0.0647** |  | **0.0335** |
| Go | **0.1404** | **0.1439** | **0.0573** | **0.0237** | **0.0414** |  |  | **0.0726** | **0.0769** | **0.0366** |  |
| From |  | **0.1286** | **0.0685** | **0.0371** | **0.0550** | **0.1127** |  |  | **0.0862** | **0.0501** | **0.0607** |
| Being |  |  | **0.0613** | **0.0280** | **0.0459** | **0.1152** | **0.2271** |  |  | **0.0414** | **0.0523** |
| Incredibly | **0.1477** |  |  | **0.0170** | **0.0335** | **0.1151** | **0.2817** | **0.0663** |  |  | **0.0380** |
| Shy | **0.1525** | **0.1586** |  |  | **0.0266** | **0.1131** | **0.3189** | **0.0593** | **0.0623** |  |  |
| To | **0.1433** | **0.1470** | **0.0556** |  | **0.0388** | **0.1159** | **0.2529** | **0.0701** | **0.0749** | **0.0353** | **0.0446** |
| Becoming |  | **0.1448** | **0.0567** | **0.0231** |  | **0.1155** | **0.2478** | **0.0722** | **0.0762** | **0.0360** | **0.0458** |
| Incredibly | **0.1362** |  | **0.0618** | **0.0284** |  |  | **0.2227** | **0.0771** | **0.0798** | **0.0415** | **0.0510** |
| Outgoing | **0.1358** | **0.1389** |  | **0.0286** | **0.0471** |  | **0.2226** | **0.0771** | **0.0803** | **0.0415** | **0.0515** |

**Table 12: fixed-attention connectivity output**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | She | Can | Go | From | Being | Incredibly | Shy | To | Becoming | Incredibly | Outgoing |
| She | **0.1523** | **0.1585** | **0.0464** |  |  | **0.1141** |  |  | **0.0649** |  |  |
| Can | **0.1518** | **0.1582** | **0.0457** |  |  | **0.1141** |  |  | **0.0647** |  |  |
| Go | **0.1404** | **0.1439** | **0.0573** |  |  | **0.1155** |  |  | **0.0769** |  |  |
| From |  |  | **0.0685** |  |  | **0.1127** |  |  | **0.0862** |  |  |
| Being |  |  | **0.0613** | **0.0280** | **0.0459** | **0.1152** |  |  | **0.0812** |  |  |
| Incredibly |  |  | **0.0497** | **0.0170** | **0.0335** | **0.1151** |  |  | **0.069** |  |  |
| Shy |  |  | **0.0427** | **0.0122** | **0.0266** | **0.1131** |  |  | **0.0623** |  |  |
| To |  |  | **0.0556** |  |  | **0.1159** | **0.2529** | **0.0701** | **0.0749** |  |  |
| Becoming |  |  | **0.0567** |  |  | **0.1155** | **0.2478** | **0.0722** | **0.0762** |  |  |
| Incredibly |  |  | **0.0618** |  |  | **0.1144** | **0.2227** | **0.0771** | **0.0798** |  |  |
| Outgoing |  |  | **0.0620** |  |  | **0.1146** |  |  | **0.0803** | **0.0415** | **0.0515** |

adjacent positions. To maintain the Transformer's capability to capture long-term dependencies, the authors introduced a single global token (shared relay node). This global token is assumed to be positioned at index 0, enables every output position to attend to all input positions, while each input position can attend to the global token. This design preserves the model ability to understand both local patterns and long-range dependencies.

Back to our example, let assume that the token as position 0 is the global token which is allowed to attend to every other token in the dataset. For the output position at index 𝑖, it can pay attention to all input positions if 𝑖 = 0. Otherwise, it is permitted to attend to input positions at index 𝑗 for 𝑗 = 0, given that 𝑖 − 1 <= 𝑗 <= 𝑖 + 1. Star transformer is illustrated in Table 13.

**Table 13: star-Transformer connectivity output**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | She | Can | Go | From | Being | Incredibly | Shy | To | Becoming | Incredibly | Outgoing |
| She | **0.1523** | **0.1585** | **0.0464** | **0.0145** | **0.0299** | **0.1141** | **0.2963** | **0.0634** | **0.0649** | **0.0259** | **0.0338** |
| Can | **0.1518** | 0.1582 | 0.0457 |  |  |  |  |  |  |  |  |
| Go | **0.1404** | 0.1439 | 0.0573 | 0.0237 |  |  |  |  |  |  |  |
| From | **0.1270** |  | 0.0685 | 0.0371 | 0.0550 |  |  |  |  |  |  |
| Being | **0.1353** |  |  | 0.0280 | 0.0459 | 0.1152 |  |  |  |  |  |
| Incredibly | **0.1477** |  |  |  | 0.0335 | 0.1151 | 0.2817 |  |  |  |  |
| Shy | **0.1525** |  |  |  |  | 0.1131 | 0.3189 | 0.0593 |  |  |  |
| To | **0.1433** |  |  |  |  |  | 0.2529 | 0.0701 | 0.0749 |  |  |
| Becoming | **0.1411** |  |  |  |  |  |  | 0.0722 | 0.0762 | 0.0360 |  |
| Incredibly | **0.1362** |  |  |  |  |  |  |  | 0.0798 | 0.0415 | 0.0510 |
| Outgoing | **0.1358** |  |  |  |  |  |  |  |  | 0.0415 | 0.0515 |

**C. Cascade Transformer**

In this section, the paper describes three proposed light Transformer architectures. It begins with a background review of the Transformer architecture, focusing on the sequence generation problem and emphasizing the Transformer decoder structure. The full Transformer block includes a masked multi-head attention layer, a position-wise fully connected feed-forward network, residual connections, and layer normalization.

To address the quadratic growth in time and memory requirements associated with the traditional Transformer, the paper introduces three light Transformer architectures. The first is the Dilated Transformer, which employs dilated connections inspired by dilated convolutions to capture long-term dependencies in a sequence. The second is the Dilated Transformer with Memory, which extends the dilated connections by preserving connections from the previous layer, aiming to cache more local contexts. The third is the Cascade Transformer, which explores cascade connections to exponentially incorporate local connections in different depths of the network. Each of these architectures is designed to reduce computation complexity compared to the full Transformer, making them more efficient for various tasks such as sequence modeling for images, audio, and text.

Alternatively, Wang et al. [125] introduced the Cascade Transformer, a model that leverages sliding window attention with an exponentially growing size corresponding to the number of layers. Specifically, at layer 𝑙, the number of cascade connections is, where 𝑏 is the base window size, and 𝑚 is the cardinal number. This design significantly reduces complexity to O(𝑛.𝑏.). Cascade attention is particularly effective for shallow networks. However, its complexity tends to approach that of full attention in deeper networks, as illustrated by the connectivity matrices in Table 14. This approach provides a nuanced exploration of attention mechanisms, offering insights into both shallow and deep network architectures.

**Table 14: Cascade attention connectivity output**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | She | Can | Go | From | Being | Incredibly | Shy | To | Becoming | Incredibly | Outgoing |
| She | **0.1523** | **0.1585** | **0.0464** | **0.0145** |  |  |  |  |  |  |  |
| Can | **0.1518** | **0.1582** | **0.0457** | **0.0140** | **0.0292** |  |  |  |  |  |  |
| Go | **0.1404** | **0.1439** | **0.0573** | **0.0237** | **0.0414** | **0.1155** |  |  |  |  |  |
| From | **0.1270** | **0.1286** | **0.0685** | **0.0371** | **0.0550** | **0.1127** | **0.1935** |  |  |  |  |
| Being |  | **0.1373** | **0.0613** | **0.0280** | **0.0459** | **0.1152** | **0.2271** | **0.0750** |  |  |  |
| Incredibly |  |  | **0.0497** | **0.0170** | **0.0335** | **0.1151** | **0.2817** | **0.0663** | **0.069** |  |  |
| Shy |  |  |  | **0.0122** | **0.0266** | **0.1131** | **0.3189** | **0.0593** | **0.0623** | **0.0229** |  |
| To |  |  |  |  | **0.0388** | **0.1159** | **0.2529** | **0.0701** | **0.0749** | **0.0353** | **0.0446** |
| Becoming |  |  |  |  |  | **0.1155** | **0.2478** | **0.0722** | **0.0762** | **0.0360** | **0.0458** |
| Incredibly |  |  |  |  |  |  | **0.2227** | **0.0771** | **0.0798** | **0.0415** | **0.0510** |
| Outgoing |  |  |  |  |  |  |  | **0.0771** | **0.0803** | **0.0415** | **0.0515** |

For Table 14, the cascade attention assume to take the base widow b = 2 and cardinal number m = 1 while the layer l = 3. Therefore, the window size = 3.

**D. BlockBERT Transformer**

Here, , where is the sequence length and is the hidden dimension. Notably, the inner product between and consumes memory. To mitigate this, the paper suggests sparsifying the attention matrix using a masking matrix defined as:

The operator denotes element-wise multiplication, and is designed as a sparse block matrix, reducing both memory consumption and the number of floating-point operations (FLOPs). Specifically, the sequence is divided into blocks, and the attention matrix is partitioned into blocks. A sparse block matrix is defined by a permutation of {1, 2, ..., n}. This leads to the introduction of Blockwise Attention, reducing both memory and computational complexity.

The Blockwise Multi-Head Attention extends this concept, allowing queries, keys, and values to be projected multiple times and performing blockwise attentions in parallel. Different blockwise attention heads can use distinct masking matrices, and the outputs are concatenated and aggregated with another linear projection. The formal definition involves the number of attention heads and hidden units . The blockwise multi-head attention is expressed as:

Here, each involves the Blockwise Attention mechanism, and is a projection matrix. This architecture reduces both memory consumption and FLOPs, making it efficient for various tasks such as long-document understanding.

To validate the claim that BlockBERT with blocks reduces memory usage by a factor of , the paper conducts memory profiling. Empirical results align with theoretical values, demonstrating substantial memory savings, particularly for longer sequences.

In summary, block-wise attention mechanism was introduced, where the input sequence is divided into non-overlapping blocks. Positions within a block, denoted as block 𝑖, are exclusively allowed to attend to positions in block 𝜋 (𝑖), where 𝜋 represents a permutation. The authors employed a straightforward method of generating permutations by shifting positions. For example, for the set {1, 2, 3}, possible permutations include {1, 2, 3}, {3, 1, 2}, and {2, 3, 1} as an described in Table 15,16 and 17 respectively. Taking {2, 3, 1}, it signifies that the first block attends to the second block, the second block attends to the third block, and the third block attends to the first block. In the multi-head setting, each head is assigned a different permutation. Formally, an output position 𝑖 is permitted to attend to input 𝑗 only if a specific condition is met:

**Table 15: block-wise attention connectivity output for {1,2,3}**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | She | Can | Go | From | Being | Incredibly | Shy | To | Becoming | Incredibly | Outgoing |
| She | **0.1523** | **0.1585** | **0.0464** | **0.0145** |  |  |  |  |  |  |  |
| Can | **0.1518** | **0.1582** | **0.0457** | **0.0140** |  |  |  |  |  |  |  |
| Go | **0.1404** | **0.1439** | **0.0573** | **0.0237** |  |  |  |  |  |  |  |
| From | **0.1270** | **0.1286** | **0.0685** | **0.0371** |  |  |  |  |  |  |  |
| Being |  |  |  |  | **0.0459** | **0.1152** | **0.2271** | **0.0750** |  |  |  |
| Incredibly |  |  |  |  | **0.0335** | **0.1151** | **0.2817** | **0.0663** |  |  |  |
| Shy |  |  |  |  | **0.0266** | **0.1131** | **0.3189** | **0.0593** |  |  |  |
| To |  |  |  |  | **0.0388** | **0.1159** | **0.2529** | **0.0701** |  |  |  |
| Becoming |  |  |  |  |  |  |  |  | **0.0762** | **0.0360** | **0.0458** |
| Incredibly |  |  |  |  |  |  |  |  | **0.0798** | **0.0415** | **0.0510** |
| Outgoing |  |  |  |  |  |  |  |  | **0.0803** | **0.0415** | **0.0515** |

**Table 16: block-wise attention connectivity output for {2,1,3}**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | She | Can | Go | From | Being | Incredibly | Shy | To | Becoming | Incredibly | Outgoing |
| She |  |  |  |  | **0.0299** | **0.1141** | **0.2963** | **0.0634** |  |  |  |
| Can |  |  |  |  | **0.0292** | **0.1141** | **0.3012** | **0.0621** |  |  |  |
| Go |  |  |  |  | **0.0414** | **0.1155** | **0.2451** | **0.0726** |  |  |  |
| From |  |  |  |  | **0.0550** | **0.1127** | **0.1935** | **0.0807** |  |  |  |
| Being |  |  |  |  |  |  |  |  | **0.0812** | **0.0414** | **0.0523** |
| Incredibly |  |  |  |  |  |  |  |  | **0.069** | **0.0290** | **0.0380** |
| Shy |  |  |  |  |  |  |  |  | **0.0623** | **0.0229** | **0.0309** |
| To |  |  |  |  |  |  |  |  | **0.0749** | **0.0353** | **0.0446** |
| Becoming | **0.1411** | **0.1448** | **0.0567** | **0.0231** |  |  |  |  |  |  |  |
| Incredibly | **0.1362** | **0.1400** | **0.0618** | **0.0284** |  |  |  |  |  |  |  |
| Outgoing | **0.1358** | **0.1389** | **0.0620** | **0.0286** |  |  |  |  |  |  |  |

**Table 17: block-wise attention connectivity output for {3,1,2}**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | She | Can | Go | From | Being | Incredibly | Shy | To | Becoming | Incredibly | Outgoing |
| She |  |  |  |  |  |  |  |  | **0.0649** | **0.0259** | **0.0338** |
| Can |  |  |  |  |  |  |  |  | **0.0647** | **0.0256** | **0.0335** |
| Go |  |  |  |  |  |  |  |  | **0.0769** | **0.0366** | **0.0466** |
| From |  |  |  |  |  |  |  |  | **0.0862** | **0.0501** | **0.0607** |
| Being | **0.1353** | **0.1373** | **0.0613** | **0.0280** |  |  |  |  |  |  |  |
| Incredibly | **0.1477** | **0.1525** | **0.0497** | **0.0170** |  |  |  |  |  |  |  |
| Shy | **0.1525** | **0.1586** | **0.0427** | **0.0122** |  |  |  |  |  |  |  |
| To | **0.1433** | **0.1470** | **0.0556** | **0.0218** |  |  |  |  |  |  |  |
| Becoming |  |  |  |  | **0.0408** | **0.1155** | **0.2478** | **0.0722** |  |  |  |
| Incredibly |  |  |  |  | **0.0470** | **0.1144** | **0.2227** | **0.0771** |  |  |  |
| Outgoing |  |  |  |  | **0.0471** | **0.1146** | **0.2226** | **0.0771** |  |  |  |

**E. Longformer Transformer**

The Longformer model, as proposed by Beltagy et al. in 2020, tackles the challenge of processing long sequences inherent in traditional transformer-based models. It introduces an attention pattern that scales linearly with the input sequence, enhancing efficiency for handling longer inputs. The paper outlines three attention patterns: sliding window, dilated sliding window, and global + sliding window.

i. Sliding Window Pattern:

The sliding window pattern utilizes a fixed-size window attention around each token. To achieve a large receptive field, the model employs multiple stacked layers of windowed attention. This enables upper layers to access every input location, constructing representations that encompass information from the entire input sequence. The computational complexity of this pattern is , where is the sequence length, and is the window size.

ii. Dilated Sliding Window:

To further expand the receptive field without a proportional increase in computation, the sliding window can be "dilated," introducing gaps of size within the window. With fixed dilation and window size across all layers, the receptive field is given by . This dilation mechanism allows the receptive field to extend to tens of thousands of tokens, even for small values of .

iii. Global Attention:

The sliding window and dilated sliding window attention, while effective for certain tasks, face limitations in generating task-specific representations, such as those required for classification tasks. To address this, global attention is introduced as an extension to sliding window attention for specific pre-selected input locations. In this approach, all pre-selected input tokens gain the ability to attend to every token across the entire input sequence, as illustrated in Figure 3d. The computational complexity of global attention is given as , making it a more efficient alternative for certain tasks that demand a broader contextual understanding. Longformer attention connectivity output is described in Table 18.

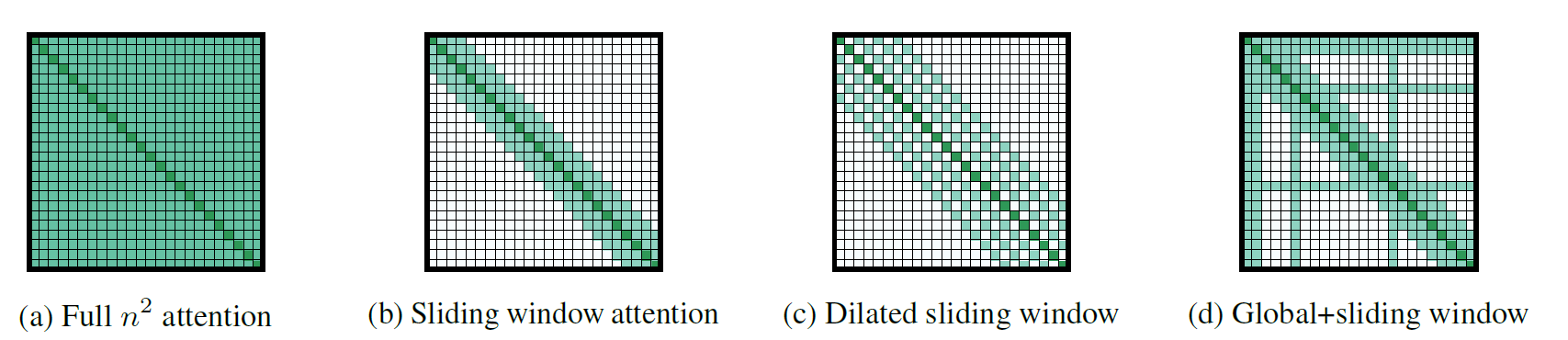


Figure 3: Attention used in longformer

**Table 18: Longformer attention connectivity output**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | She | Can | Go | From | Being | Incredibly | Shy | To | Becoming | Incredibly | Outgoing |
| She | **0.1523** | **0.1585** | **0.0464** | **0.0145** | **0.0299** | **0.1141** | **0.2963** | **0.0634** | **0.0649** | **0.0259** | **0.0338** |
| Can | **0.1518** | **0.1582** | **0.0457** | **0.0140** | **0.0292** | **0.1141** | **0.3012** | **0.0621** | **0.0647** | **0.0256** | **0.0335** |
| Go | **0.1404** | **0.1439** | **0.0573** | **0.0237** |  |  |  |  |  |  |  |
| From | **0.1270** | **0.1286** | **0.0685** | **0.0371** | **0.0550** |  |  |  |  |  |  |
| Being | **0.1353** | **0.1373** |  | **0.0280** | **0.0459** | **0.1152** |  |  |  |  |  |
| Incredibly | **0.1477** | **0.1525** |  |  | **0.0335** | **0.1151** | **0.2817** |  |  |  |  |
| Shy | **0.1525** | **0.1586** |  |  |  | **0.1131** | **0.3189** | **0.0593** |  |  |  |
| To | **0.1433** | **0.1470** | **0.0556** | **0.0218** | **0.0388** | **0.1159** | **0.2529** | **0.0701** | **0.0749** | **0.0353** | **0.0446** |
| Becoming | **0.1411** | **0.1448** |  |  |  |  |  | **0.0722** | **0.0762** | **0.0360** |  |
| Incredibly | **0.1362** | **0.1400** |  |  |  |  |  |  | **0.0798** | **0.0415** | **0.0510** |
| Outgoing | **0.1358** | **0.1389** |  |  |  |  |  |  |  | **0.0415** | **0.0515** |

**F. BigBird Transformer**

The Big Bird model (Zaheer et al., 2020), which is a Transformer designed for handling longer sequences. Its key components include global attention, random attention, and sliding window attention as shown in Figure 4. The concept of using global model memory can be traced back to Longformer. In Big Bird, the global model memory is extended to contain tokens within the sequence, which the authors refer to as the "internal transformer construction (ITC)." This can be seen as a model-memory-based approach. Sliding window attention, which was first proposed in early local-based attention models such as Image Transformer, Compressed Attention, and Sparse Transformer, is also used in Big Bird. Each query attends to tokens to the left and tokens to the right, corresponding to a fixed pattern (FP) approach. Finally, each query attends to r random keys, which is a fixed pattern. The complexity of Big Bird is given as . BigBird attention connectivity output is described in Table 19.

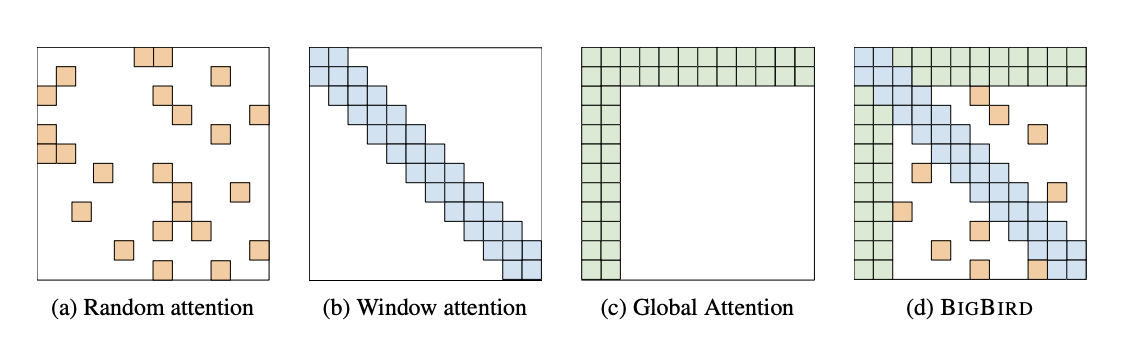


Figure 4: BigBird Attention

(Source: [sliding window in transformer - Bing images](https://www.bing.com/images/search?view=detailV2&ccid=NYVDEi8z&id=58A031C1B846299CF1CBE175BB83C0CB3E91B1F3&thid=OIP.NYVDEi8zkcGbgRnh1RMz4gHaBm&mediaurl=https%3a%2f%2fth.bing.com%2fth%2fid%2fR.358543122f3391c19b8119e1d51333e2%3frik%3d87GRPsvAg7t14Q%26riu%3dhttp%253a%252f%252fpelhans.com%252fimg%252fin-post%252fvarious_attention%252flongformer_arc.png%26ehk%3dAYz9D%252bV%252b62FXVCAuwTaA%252bXuqXl%252bo9l1JYUvuviW81uw%253d%26risl%3d%26pid%3dImgRaw%26r%3d0&exph=370&expw=1707&q=sliding+window+in+transformer&simid=608052578015526791&FORM=IRPRST&ck=91C90FA119C3E0DD8DF2CA84CBC95CF9&selectedIndex=0&ajaxhist=0&ajaxserp=0))

**Table 19: BigBird-attention connectivity output**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | She | Can | Go | From | Being | Incredibly | Shy | To | Becoming | Incredibly | Outgoing |
| She | **0.1523** | **0.1585** | **0.0464** | **0.0145** | **0.0299** | **0.1141** | **0.2963** | **0.0634** | **0.0649** | **0.0259** | **0.0338** |
| Can | **0.1518** | **0.1582** | **0.0457** | **0.0140** | **0.0292** | **0.1141** | **0.3012** | **0.0621** | **0.0647** | **0.0256** | **0.0335** |
| Go | **0.1404** | **0.1439** | **0.0573** | **0.0237** |  | **0.1155** | **0.2451** |  |  |  |  |
| From | **0.1270** | **0.1286** | **0.0685** | **0.0371** | **0.0550** |  | **0.1935** |  |  |  |  |
| Being | **0.1353** | **0.1373** |  | **0.0280** | **0.0459** | **0.1152** |  | **0.0750** | **0.0812** |  |  |
| Incredibly | **0.1477** | **0.1525** |  | **0.0170** | **0.0335** | **0.1151** | **0.2817** |  | **0.069** |  |  |
| Shy | **0.1525** | **0.1586** |  |  | **0.0266** | **0.1131** | **0.3189** | **0.0593** |  |  | **0.0309** |
| To | **0.1433** | **0.1470** | **0.0556** |  |  |  | **0.2529** | **0.0701** | **0.0749** |  | **0.0446** |
| Becoming | **0.1411** | **0.1448** |  | **0.0231** | **0.0408** |  | **0.2478** | **0.0722** | **0.0762** | **0.0360** |  |
| Incredibly | **0.1362** | **0.1400** | **0.0618** | **0.0284** |  |  |  | **0.0771** | **0.0798** | **0.0415** | **0.0510** |
| Outgoing | **0.1358** | **0.1389** |  |  | **0.0471** |  |  |  |  | **0.0415** | **0.0515** |

**G. Sparse Sinkhorn Transformer**

To make block-wise attention more flexible, Tay et al. introduced sparse Sinkhorn attention. This variant is akin to block-wise attention but involves sorting keys. In simple terms, it transforms the input sequence 𝑿 in R 𝑛×𝑑 to 𝑿 ′ in R 𝑛𝑏×𝑑, where 𝑛𝑏 is the number of blocks, and 𝑿 ′ 𝑖 represents the sum of input within that block. A straightforward feed-forward network then learns a mapping 𝑹𝑖 in R 𝑛𝑏 from the 𝑖-th block 𝑿 ′ 𝑖 to all blocks.

To create a sorting matrix from 𝑹 in R 𝑛𝑏×𝑛𝑏, consisting of only 0s and 1s with rows and columns summing to one, iterative normalization is applied to rows and columns. This sorting matrix is then used to rearrange the keys, effectively learning which block to attend to (see Figure 16). Although sparse Sinkhorn attention reduces complexity to O (𝑛 2 𝑏 ), the original paper maintains quadratic complexity concerning sequence length due to a constant block size. Additionally, the authors suggested a truncated version of sparse Sinkhorn attention, selecting a few keys after sorting, further reducing complexity to O (𝑛).

**H. Reformer Transformer**

Reformer is another transformer-based model that improve the efficiency of transformer by deploying two techniques. One is replacing the dot-product attention with locality-sensitive hashing to reduce the complexity of attending to over long sequences and also introduces the use of reversible residual layers which further reduces its memory usage.

**Locality-Sensitive Hashing**

The locality-sensitive hashing uses the idea of parameter-sharing between query and keys and further compute a hash function that matches similar vectors together. After the hash function is applied the sequence is rearranged in order to bring element with the same hash together and divide the sorted sequence to small chunks to allow parallel processing. Then, attention is applied between the short chunks and their adjoin neighbors to cover the overflow which this helps in reducing the computational load greatly. To achieve this, a fixed random matrix . The hashing function used is defined as;

62

where [;] denotes the concatenation of two vectors. All position index is maintained for all query and key. Furthermore, the attention can only be computed if the hash of query and key are equal.

**Reversible Residual Layers**

To train a single layer of a network often requires more memory space up to GB usually fits on a single GPU. When training a multi-layer model with gradient decent, activations from each layer need to be saved for use in the back-propagation. A Transformer model has many layers, for this reason memory quickly funs out if used to caches values form each of those layers. To solve this problem, Reformer was used to recompute the input of each layer on-demand during backpropagation, rather than storing it in the memory. The reversible layers was used to accomplish this. Typically, in a residual network, each layer in the stack takes the vectors that pass through the network and adds to them. However, with reversible layers, two sets of activations are used for each layer - one following the usual procedure and becoming increasingly updated as it passes through each layer, and the other capturing only the changes made to it. Consequently, when running the network in reverse, all you have to do is subtract the activations applied at each layer from those from the last one.

The complexity of Reformer is given as

**2.8.2.2 Kernel Attention**

Kernel attention, involves a kernel function denoted as 𝐾(·, ·). This function takes two vectors as inputs and produces the product of their projection by a feature map 𝜙(·):

63

Here, represents the feature map applied to the vectors and . The kernel function essentially measures the similarity or correlation between the two vectors after their projection onto a feature space.

Consider an example where we have two vectors, x = [1, 2, 3] and y = [4, 5, 6], and a simple kernel function defined by their dot product after applying a feature map 𝜙(·) as in equation 15.

For simplicity, choose a linear feature map . The dot product of and is:

Researchers have interpreted Softmax as a kernel and decomposed it as an inner product in a suitable space. This innovative interpretation allows them to rearrange computations in a clever manner, resulting in a reduction of computational complexity. This approach involves transforming the Softmax operation into an inner product operation in a specific space, providing new insights into the computational structure and enabling more efficient calculations. The reinterpretation of Softmax as a kernel, along with the strategic rearrangement of computations, contributes to optimizing the efficiency of the overall process.

**A. Linear Transformer**

The linear Transformer (Katharopoulos et al., 2020) express its own self-attention as a linear dot-product by using a kernel feature based formulation and the associative property of matrix products to improve the complexity of computing which is to . The linear Transformer replace the unnormalized attention matrix with , where is a high-dimensional, feature map which is applied on row-wise manner. For more understanding of the linear Transformer and its class, consider the general form of the self-attention given as:

(.) is a similarity score between input vectors and which is similar to the exponential function of inner product in the vanilla Transformer. A more natural way of computing this (.) is to express the similarity as a kernel function, given as:

Associative property of matrix multiplication was used in equation 69, denotes outer product of vectors. is applied row-wise to both and . This benefits the autoregressive attention more and also enables Transformer decoders runs like RNNs. The authors propose the use of a simple feature map given as:

(.) denotes the exponential linear unit activation function and the result proves that the linear transformer performs on par with the standard Transformer with complexity of for the end-to-end of the model.

**B. Performer Transformer**

The performer Transformer (Choromanski et al., 2020) also falls into the kernel features based family. It uses random feature maps to approximate the dot product attention score of the Transformer. From equation 69, it takes of the following form for functions function and for some distribution :

The performer gain inspiration from the random Fourier feature map that was used to approximate Gaussian kernel. It uses trigonometric functions with . This trigonometric function does not guarantee non-negative attention scores and thus could lead to unstable behaviors. To address this issue, the authors propose another version of Performer (Choromanski et al., 2020) with unbiased positive random feature map estimator that tends to 0 as approximated values tends to 0, which uses .

The complexity of Performer is given as .

**2.8.2.3 Low-Rank Attention**

Another approach to reduce the Transformer’s complexity is to approximate the attention by factorizing it into the product of two matrices with lower dimensions.

**A. Linformer transformer**

The main idea of Linformer transformer (Wang, Li, et al., 2020) is to propose a new self-attention mechanism can be approximated by a low-rank matrix. The model leverage on the fact that the matrix formed by the vanilla transformer self-attention(Vaswani et al., 2017) can be reduced to a low-rank matrix and this can be done by projecting the -dimensional keys and values to a -dimensional projected key and , then the matrix can be computed as -dimensional in-place of -dimensional. Therefore, the new attention head can be given as:

where and are normal linear transformation of into queries, keys and values as done in the vanilla transformer (Vaswani et al., 2017), , are the additional -dimensional projection of the keys and values. The complexity of Linformer is given as .

**B. Synthesizer transformer**

Tay et al. (2022) introduced the Synthesizer family of models, presenting alternatives to traditional attention mechanisms by learning compatibility scores without calculating pairwise dot products between queries and keys. For example, the Dense Synthesizer learns similarity scores using a simple position-wise feed-forward network to project each row of input 𝑿. The Factorized Dense Synthesizer enhances efficiency by employing two feed-forward networks and tiling functions.

From our example in Table 6 input embedding Matrix (x) can be said to be a 11 X 7 matrix, the idea of synthesizer is to pick each row of and pass it through a two feed-forward networks

where and ,

is another parameterized function of the input embedding (x), also where .

Therefore,

The Factorized Random Synthesizer, another baseline, utilizes independent similarity scores and its attention is given by:

which is randomly generated.

While these Synthesizer models eliminate the need for pairwise dot product computations for enhanced speed, the complexity of the attention mechanisms remains quadratic with respect to the sequence length, highlighting their efficiency within the Transformer architecture.

**2.8.2.4 Local + Global Attention**

The approaches discussed above utilize attention approximation to address the challenges associated with training large transformers, mitigating the need for extensive computational and memory resources. However, each of these methods exhibits distinct strengths tailored to specific tasks. Consequently, their performance is concentrated on particular applications, leading to a trade-off where accuracy may be compromised for tasks outside their primary focus [57].

**A. Scatterbrain transformer**

Scatterbrain approximation (Chen et al., 2021) was inspired by the observation that sparse and low-rank approximations are complementary for many attention matrices, outperforming each individually. Theoretical characterizations are provided, delineating regimes where each approximation excels based on attention softmax temperature. Scatterbrain employs a workflow that combines sparse attention via Locality Sensitive Hashing (LSH) and low-rank attention via kernel approximation. This approach minimizes the quadratic complexity of softmax attention, resulting in a more memory-efficient and accurate estimation.

Scatterbrain approximation attention was constructed leveraging the strength of some efficient transformer models by combining some transformers that use the low-rank approximation and sparse approximation. One efficient way to achieve these combinations is by performing addition on the attention matrices. For example, let consider combining a sparse and low-rank approximation attention matrices. let S represent the sparse approximation matrix and for the low-rank approximation. Therefore, the Scatterbrain approximation matrix was formed simply by adding S and and multiplying the results with V as:

This efficiently reduces the quadratic complexity of the vanilla transformers to a linear complexity since S is sparse and .

**B. FMMformer transformer**

The proposed FMMformers (Nguyen et al., 2022) represent a groundbreaking advancement in transformer architectures, drawing inspiration from the efficient fast multipole method (FMM) utilized in particle simulations. FMMformers innovate by decomposing attention into near-field and far-field components, mirroring FMM's approach to particle-particle interaction. One of the standout features of FMMformers lies in their unique attention modeling. A banded matrix elegantly captures near-field attention, while a low-rank matrix briefly represents far-field attention. This decomposition proves to be a game-changer, as it not only enhances computational efficiency but also drastically reduces memory footprint. Unlike standard transformers, which suffer from quadratic complexity, FMMformers exhibit linear complexity concerning both computational time and memory, particularly with respect to sequence length.

**2.8.2.5 Memory Compression**

The transformer complexity may also be reduced by modifying the model’s architecture and preserving the original attention mechanism.

**A. Memory Compressed Transformer**

The Memory Compressed Transformer was introduced by (P. J. Liu et al., 2018) , which the major objective of the study is to modify the transformers for better handling of longer sequences. One simple approach to handle long sequences in Transformers is to restrict the attention span to a local vicinity which was the idea behind (P. J. Liu et al., 2018) work which suggested partitioning the input sequence into blocks of approximately equal length, allowing for self-attention computation within each block independently. This strategy ensures that the attention cost per block remains constant, resulting in a linear scaling of the number of activations with respect to the input length. The work also introduce memory-compressed attention which aims to use the strided convolution method to minimize the number of keys and values while keeping the queries unaltered. This change led to decrease in the size of the attention matrix and the attention computation which is largely determined by the kernel size and convolution strides. The utilization of memory-compressed attention allows the model to globally exchange information throughout the input sequence, in contrast to the limited scope of local attention. The complexity of Memory Compressed Transformer is , where convolutionally-compressed sequence length.

**B. Poolingformer**

Poolingformer (H. Zhang et al., 2021) was inspired by the pooling operation of the convolutional neural network. The model introduces a two-level attention mechanism majorly for modelling long document. The first level of attention is called the sliding window attention same like that of longformer which allows each token to only attend to its neighbor tokens. Given token and its neighbor were set within a window size defined as :

From the sliding window pattern, token output can be computed as:

where and are sub-matrices of K and V with corresponding column indexes of . Due to the limitation of the receptive field size to , the model's performance may suffer when it comes to tasks that require understanding of long documents. To solve this problem a second level of attention was introduced called the pooling attention. The pooling attention was built on the output of the sliding window to produce a new query, key and value matrices given as and . Query vector of token , its corresponding key and value with a very large window size . The pooling attention was applied to and in order to compress the key and the value.

Where represents stride size while represents kernel size. The output of the pooling attention for token is given as:

The complexity of poolingformer is given as .

**2.8.2.6 Other method**

The introduction of Multi-Query Attention (MQA) (Ainslie et al., 2023) has presented a compelling solution for accelerating decoder inference by employing a single key-value head. However, this efficiency gain has been accompanied by concerns about potential quality degradation and the need for a separate model dedicated to faster inference. In response to these challenges, the authors propose a dual-pronged strategy. Firstly, the paper introduces a systematic recipe for uptraining existing multi-head language model checkpoints into models with MQA, achieving this transformation with a remarkably frugal 5% of the original pre-training compute. This uptraining approach not only streamlines the integration of MQA but also serves as an economically viable solution for enhancing existing models. Secondly, the authors present Grouped-Query Attention (GQA) as a novel extension of multi-query attention, incorporating an intermediate number of key-value heads. This approach is positioned as a generalization of MQA, striking a balance between speed and quality without the need for an extensive overhaul of existing models. The empirical results demonstrate the efficacy of the proposed strategies. The uptrained GQA model emerges as a compelling compromise, achieving a level of quality that closely rivals multi-head attention while maintaining a speed comparable to MQA. This finding not only validates the feasibility of uptraining existing models but also showcases GQA as a versatile approach for efficient attention mechanisms. In conclusion, the paper makes significant strides in addressing the trade-off between efficiency and quality in transformer models. The proposed strategies not only contribute practical solutions for MQA integration and uptraining but also open avenues for further exploration of grouped-query attention mechanisms. This work stands as a valuable addition to the ongoing discourse on optimizing transformer architectures for enhanced performance and efficiency.

**Table 20: Summary of Specific Approaches for Attention Mechanism**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Model/Paper** | **Complexity** | **Sparse** | **Low Rank** | **Kernel** | **Memory** |
| Memory Compressed (P. J. Liu et al., 2018) |  |  |  |  |  |
| Set Transformer (Lee et al., 2019) |  |  |  |  |  |
| Transformer-XL (Dai et al., 2019) |  |  |  |  |  |
| Sparse Transformer (Child et al., 2019) |  |  |  |  |  |
| Reformer (Kitaev et al., 2020) |  |  |  |  |  |
| Routing Transformer (Roy et al., 2020) |  |  |  |  |  |
| Axial Transformer (Ho et al., 2019) |  |  |  |  |  |
| Compressive Transformer (Rae et al., 2019) |  |  |  |  |  |
| Sinkhorn Transformer (Tay, Bahri, et al., 2020) |  |  |  |  |  |
| Longformer (Beltagy et al., 2020a) |  |  |  |  |  |
| Synthesizer (Tay et al., 2021) |  |  |  |  |  |
| Performer (Choromanski et al., 2020) |  |  |  |  |  |
| Funnel Transformer (Dai et al., 2020) |  |  |  |  |  |
| Linformer (Wang, Li, et al., 2020) |  |  |  |  |  |
| Linear Transformers (Katharopoulos et al., 2020) |  |  |  |  |  |
| Big Bird (Zaheer et al., 2020) |  |  |  |  |  |
| Long Short Transformers (Zhu et al., 2021) |  |  |  |  |  |
| Poolingformer (H. Zhang et al., 2021) |  |  |  |  |  |
| Nystromformer (Xiong et al., 2021) |  |  |  |  |  |
| Perceiver (Jaegle et al., 2021) |  |  |  |  |  |
| Clusterformer (Wang, Zhou, et al., 2020) |  |  |  |  |  |
| Adaptive Sparse Transformer (Correia et al., 2019) |  |  |  |  |  |
| Product Key Memory (Lample et al., 2019) |  |  |  |  |  |
| Switch Transformer (Fedus et al., 2022) |  |  |  |  |  |
| GShard (Lepikhin et al., 2020) |  |  |  |  |  |
| Scaling Transformer (Jaszczur et al., 2021) |  |  |  |  |  |
| GLaM (N. Du et al., 2020) |  |  |  |  |  |
| Blockwise Transformer (Qiu et al., 2019) |  |  |  |  |  |
| Low-Rank Transformer (Winata et al., 2019) |  |  |  |  |  |
| ST-MoE (Zoph et al., 2022) |  |  |  |  |  |
| Scatterbrain Transformer (Chen et al., 2021) |  |  |  |  |  |
| FMMformer transformer (Nguyen et al., 2022) |  |  |  |  |  |

Memory, Low Rank, Kernel, Sparse (Learnable/Fixed/ Factorized/ Random) Patterns. N represents the sequence length and B represents local window size denotes global model memory length and convolutionally-compressed sequence length.

**2.9 Drawbacks**

**2.9.1 Lack of Theoretical Understanding:**

In the world of machine learning and natural language processing, the self-attention mechanism has become widely popular and has shown success in practice. However, despite its practical achievements, there's a big gap in our theoretical understanding of how it works. Even though it's used everywhere, we still don't have clear answers to basic questions about why self-attention is so effective. While we can see that it's great at capturing complex connections in sequences, the underlying theory that explains why it's so good is still unclear. This lack of a solid theoretical foundation makes it challenging to explain precisely why self-attention performs exceptionally well in certain situations, limiting our understanding of its full potential and the best ways to use it.

It is not entirely sure how crucial the interaction between "queries" (Q) and "keys" (K), known as QK, is in the self-attention mechanism. This uncertainty makes it challenging to understand how self-attention really works. Additionally, the lack of clarity on how self-attention makes decisions creates a hurdle in building more efficient and easily understandable Transformer models. There are some existing works that support (Serrano & Smith, 2019; Wiegreffe & Pinter, 2019) and against (Jain & Wallace, 2019) the claim that self-attention is explainable.

In the sparse methods such as (Beltagy et al., 2020b; Child et al., 2019; Guo et al., 2019; Soldaini & Moschitti, 2020; Tay, Bahri, et al., 2020) used to improve self-attention, the focus is mainly on practical results, lacking a thorough theoretical analysis of the underlying mechanisms in Sparse Transformers. Although the papers provides some intuition on why sparsity in the attention mechanism works, a more rigorous theoretical analysis could enhance our understanding of the principles contributing to Sparse Transformers' performance. One potential avenue for theoretical analysis is exploring the impact of different sparsity patterns on Sparse Transformers' performance. Various sparse factorizations of the attention matrix as shown from Table 11 – 19 has been proposed but falls short of analyzing how these different patterns affect the model's performance. A theoretical analysis would shed light on the trade-offs between different sparsity patterns and their influence on Sparse Transformers.

Another direction for theoretical analysis could delve into the connection between sparsity in the attention mechanism and the model's ability to capture long-term dependencies. These papers shows that Sparse Transformers are adept at handling long sequences, it lacks a detailed analysis of why this is the case. A theoretical examination could unravel how sparsity in the attention mechanism enables the model to capture long-term dependencies.

The lack of theoretical understanding about how self-attention works creates a big challenge in developing and improving Transformers. Closing this knowledge gap is crucial, not just for figuring out the details of self-attention but also for pushing forward the whole field of machine learning, especially as Transformers keep making a big impact. As researchers wrestle with these unanswered questions, the quest to fully unleash the power of self-attention continues. It's pushing the community to dive deeper into understanding and making better use of this influential mechanism.

**2.9.2** **Attention for Capturing Cross-document Relations:**

In capturing relationships among documents, particularly when extracting information from multiple documents, the models examined in this paper, including the conventional self-attention module (Vaswani et al., 2017), often simplify the input by amalgamating documents into a flat sequence. This approach overlooks potential relationships between distinct documents, which may contain overlapping, complementary, or conflicting information. Recognizing these cross-document relations is essential for assisting models in extracting vital information, improving coherence, and minimizing redundancy in the extraction process. Notably, research on self-attention mechanisms has consistently followed a similar pattern of combining documents into a flat sequence, neglecting the exploration of relationships between multiple documents. Consequently, a promising avenue for further investigation involves developing enhanced mechanisms for integrating diverse cross-document relationships, potentially within the attention mechanism of deep learning models. This innovation aims to amplify models' capability to comprehend and leverage cross-document relations.

**2.9.3 Complexity Vs Practical Efficiency:**

The evaluation of model efficiency involves a delicate balance between theoretical metrics and real-world practicality. Theoretical metrics, often expressed through asymptotic complexity notations like *,* provides a partially view into the computational efficiency of models. However, these metrics may fall short of fully encapsulating the practical efficiency of models in real-world scenarios. A noteworthy illustration of this complexity-practicality interplay is observed in models like the Reformer (Kitaev et al., 2020), which, on the surface, boasts lower asymptotic complexity. Despite this theoretical advantage, the Reformer (Kitaev et al., 2020) may exhibit slower performance on smaller sequences as reported in Table 2 of the original paper ‘Long Range Arena: A Benchmark for Efficient Transformers’ (Tay, Dehghani, et al., 2020). This paradoxical scenario arises due to concealed constants embedded in the model's complexity, challenging the direct correlation between theoretical efficiency and real-world execution.

Moreover, the efficiency concern extends specifically to the self-attention mechanism. When confronted with long sequences, the computation and memory complexity of the self-attention module emerge as potential bottlenecks. This challenge becomes pronounced, particularly when dealing with extensive input data. The inefficiency in processing long sequences poses a practical hurdle, especially in the context of handling large-scale datasets where the computational demands can become overwhelming.

A pertinent aspect of this challenge lies in the lack of compatibility between theoretically advantageous operations, such as sparse matrix multiplication, and their alignment with hardware realities, notably on Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs).

Several sparse factorizations of the attention matrix as explained in this paper with examples showing from Table 11 – 19 (Beltagy et al., 2020b; Child et al., 2019; Guo et al., 2019; Soldaini & Moschitti, 2020; Tay, Bahri, et al., 2020), the theoretical benefits of sparse matrix multiplication, aimed at reducing the number of computations required to compute the attention weights. These factorizations include fixed patterns, strided patterns, and learnable patterns, which allow the model to attend to only a subset of the input positions, rather than all positions, as in traditional transformers (Vaswani et al., 2017). By attending to only a subset of the input positions, Sparse Transformers reduce the memory requirements of the model, as the attention weights and activations for the non-attended positions do not need to be stored in memory. However, according to (Buluc & Gilbert, 2008), hardware platforms such as GPUs and TPUs are not well designed for sparse operation. In light of these, there is need for a nuanced understanding of the interplay between theoretical models and real-world efficiency.

The complexity and practical efficiency dilemma underscores the intricacies involved in gauging the true effectiveness of models, particularly in the context of processing long sequences and large-scale datasets. Navigating this challenge is imperative for the development of models that not only showcase theoretical prowess but also exhibit practical efficiency in the demanding landscape of real-world applications.

**2.9.4 Call for Clear Reporting:**

This means going beyond just talking about how well a model performs and getting into the details that help researchers understand why it works the way it does and other factors affecting it performances. One important detail is "computational and memory complexity," which is about how complicated the model as explained in the paper Linear Transformers (Katharopoulos et al., 2020) and FMMformer transformer (Nguyen et al., 2022). It helps the researcher see the thought and planning that went into making the model and lets the researcher know its strengths and weaknesses. Another key point is "floating-point operations (FLOPs)," a fancy way of saying how much math the model has to do. This helps researchers figure out how much computer power the model needs and if it's efficient. Also, sharing "wall-clock time" on specific hardware tells researchers how quickly the model works in real-life situations. This is crucial because sometimes we need models to work fast, and knowing how they perform in different situations helps researchers decide where to use them. Lastly, it stresses the importance of sharing the "memory footprint," or how much memory the model uses. This is vital when we have limits on resources like memory. Knowing the memory footprint helps researchers understand if the model will work well in situations where memory is limited. In simple terms, the call for clear reporting encourages researchers to share not just the good performance of a model but also details like how complex it is, how much math it does, how fast it works, and how much memory it needs. This helps everyone understand the model better and decide where and how to use it.

**2.10 Conclusion**

Transformers have quickly become the go-to model and also achieve state-of-the-art performance across natural language processing tasks. However, this success comes at a high cost, especially on the quadratic complexity of the self-attention module. Due to this challenge, many researchers have explored many techniques to find solutions associated with the quadratic complexity nature of the self-attention module. The inner workings of the self-attention module with worked examples and surveyed some literature on efficient transformer models, especially on the quadratic complexity of the self-attention module was presented. Also, the existing models was divided into different classes based on the techniques used. Finally, general drawbacks of the attention mechanism and possible future research directions on this model are also presented.

**CHAPTER THREE**

**3.0 RESEARCH METHODOLOGY**

**3.1 Introduction**

This chapter presents a formalized proposed methodology that will be used to replace the highly costly attention and replace the add and norm layer mechanism in the Transformer to reduce the computational complexity. The Grouped-Global attention model is proposed to replace the self-attention model in the transformer model. The concept of the dot-product self-attention model in the vanilla Transformer helps show the interaction between all input nodes and outputs. This means if we have 100,000 input each of these nodes can attend to all other nodes which makes the model more complex.

(Caciularu et al., 2021) stated that in handling multiple documents, rather than encoding each document independently, these models, as outlined in section 2.7.2, enable concatenating multiple documents into a long sequence of tokens and encoding them jointly as shown in Figure 25.

Consider this example:

**Doc 1:** “I love programming but it is difficult to learn”

**Doc 2:** “I have a passion for coding”

**Doc 3:** “I find great joy in engaging with programming”

In handling this type of task a simple concatenation of input documents into a flat sequence has been used, Figure 25 depicts that but this method ignores the cross-document relations as pointed out in section 9.2. capturing cross-text relationships can be useful for downstream tasks such as multi-document summarization tasks, cross-document language modeling, multi-document question and answer tasks, and so on. Following this intuition, a new self-attention mechanism is being proposed to capture cross-text relationships at a reduced computational cost.

The idea of Grouped-Global attention is to incorporate both local and global attention mechanisms into the Transformer model to reduce the computational complexity and to improve the model performance on many tasks. This model could help capture both fine-grained details within local contexts and broader dependencies across all input sequences, especially for long sequence tasks in natural language processing. The overview of the model is represented in Figure 25a and Figure 25b. This approach has the potential to provide a more comprehensive understanding of the input sequences by considering both local and global contexts simultaneously.

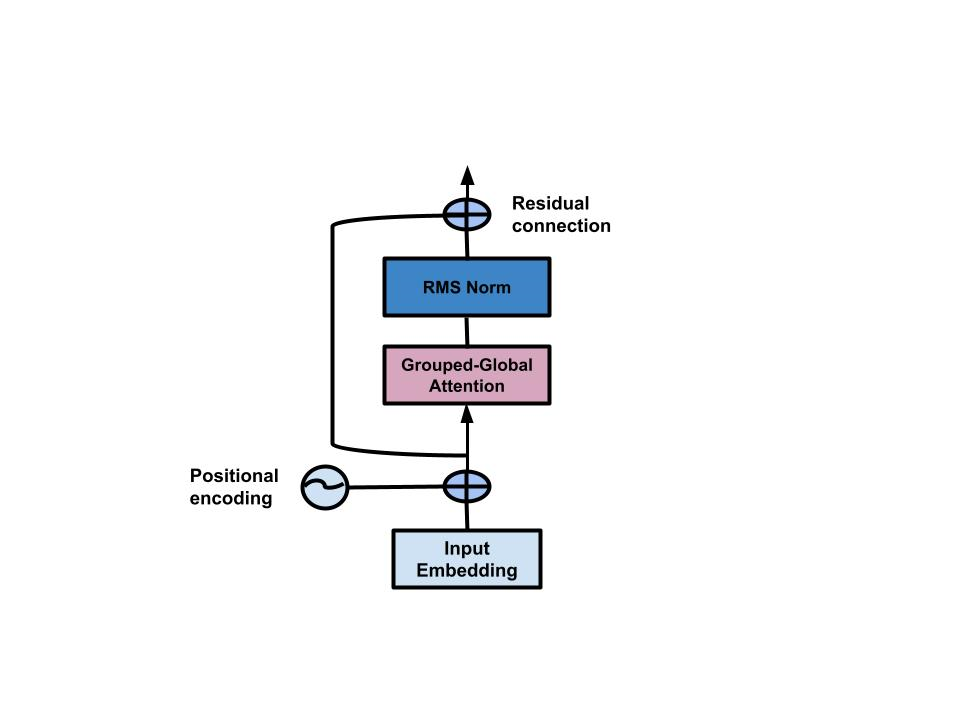


Figure 25a: Grouped-Global Transformer



Figure 25b: Grouped-Global Attention

**3.2 Preliminaries- Transformers and Self-Attention**

The main idea of the Transformer was built upon multi-head self-Attention (MHA), this helps the model to attend jointly to information from different positions. MHA can be defined as:

Let denotes sequence of feature with vectors of dimensions (i.e the input embedding vector). The input sequence is projected by three learned matrices and to correspond to . which is also a learned matrix. and are the hidden layer dimensions of the subspaces projection. is the number of head. can be defined as:

and each can be defined as:

The self-attention in equation (74) is the only part of the Transformer that acts across all sequences which can be refers to as the context mapping matrix . is used to capture the input context of a token , by combining all token in the sequence. So, computing is very expensive because it requires multiplying two matrices resulting into for both time and space complexity which has become the bottleneck for transformers.

**3.2.1 The self-attention Algorithm**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm 1: Self-Attention with Multi-Head Attention** **Algorithm**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***// Algorithm to perform multi-head self-attention***

**Algorithm** **multihead\_self\_attention(input\_sequence, num\_heads, d\_model):**

*// Linear Projections for Queries, Keys, and Values*

1. Q = linear\_projection(input\_sequence, W\_Q) # Linear projection for Queries

2. K = linear\_projection(input\_sequence, W\_K) # Linear projection for Keys

3. V = linear\_projection(input\_sequence, W\_V) # Linear projection for Values

*// Step 2: Splitting into Multiple Heads*

4. Q\_heads = split\_into\_heads(Q, num\_heads)

5. K\_heads = split\_into\_heads(K, num\_heads)

6. V\_heads = split\_into\_heads(V, num\_heads)

*// Step 3: Apply Scaled Dot-Product Attention for each head*

7. attention\_outputs = []

8. for i in range(num\_heads):

9. attention\_output\_i = scaled\_dot\_product\_attention(Q\_heads[i], K\_heads[i], V\_heads[i])

10. attention\_outputs.append(attention\_output\_i)

*// Step 4: Concatenate Attention Outputs*

11. concatenated\_output = concatenate\_along\_axis(attention\_outputs, axis=-1)

*// Step 5: Linear Projection for Final Output*

12. output\_sequence = linear\_projection(concatenated\_output, W\_O)

13. **return output\_sequence**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm 2: Linear projection function** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm** **linear\_projection(input\_sequence, weight\_matrix):**

**return input\_sequence @ weight\_matrix**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm 3: Splitting function to divide into multiple heads**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm split\_into\_heads(matrix, num\_heads):**

*// Assuming matrix is 2D*

1. head\_size = matrix.shape[-1] // num\_heads

2. heads = []

3. for i in range(num\_heads):

4. start\_idx = i \* head\_size

5. end\_idx = (i + 1) \* head\_size

6. head\_i = matrix[:, start\_idx:end\_idx]

7. heads.append(head\_i)

8. **return heads**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm 4: Scaled Dot-Product Attention function**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm** **scaled\_dot\_product\_attention(Q, K, V):**

*//Assuming Q, K, V are 2D arrays or tensors*

1. d\_k = Q.shape[-1]

2. scores = softmax(Q @ K.T / sqrt(d\_k))

3. output = scores @ V

4. **return output**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm 5: Concatenation function along the last axis**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm** **concatenate\_along\_axis(tensors, axis=-1):**

*//Assuming tensors is a list of 2D arrays or tensors*

**return concatenate(tensors, axis=axis)**

**3.2.2 Time and Space Complexity for Multi-head Self-Attention**

**A. Time Complexity:**

**1. Linear Projections (Queries, Keys, and Values):**

For each linear projection, it involves a matrix multiplication, which has a time complexity of approximately ) where:

is the number of attention heads, is the length of the input sequence, is the size of each attention head, is the dimension of the word embedding.

2. **Scaled Dot-Product Attention for Each Head:**

The time complexity of the scaled dot-product attention is dominated by the softmax operation and the matrix multiplication, which has a time complexity of approximately ).

3. **Concatenation of Attention Outputs:**

Concatenating the attention outputs involves combining the outputs of all attention heads, which has a time complexity of ).

4. **Linear Projection for Final Output:**

The final linear projection has a time complexity similar to the initial linear projections, i.e., )

Therefore, the overall time complexity is approximately ).

**B. Space Complexity:**

1. **Linear Projections (Queries, Keys, and Values):**

The space complexity for each linear projection is approximately ).

2. **Scaled Dot-Product Attention for Each Head:**

The space complexity for each attention head is approximately ).

3. **Concatenation of Attention Outputs:**

The concatenated attention output has a space complexity of approximately ).

4. **Linear Projection for Final Output:**

The space complexity for the final linear projection is similar to the initial linear projections, i.e., ).

Therefore, the overall space complexity is approximately ).

**3.3 Methods and Algorithms**

This section presents many variants of grouped-global attention mechanisms and their algorithms used for this research work. Here are the names of the proposed methods and algorithms:

1. Grouped-Average-Global-Pooling Attention (GAGP Attention)
2. Grouped-Average-Global-Pooling + MultiHead Attention (GAGP+MultiHead Attention)

Each of these methods will beapplied on encoder-decoder transformer Architectures to solve a wide range of natural language processing tasks.

**3.3.1 Grouped-Average-Global-Pooling Attention (GAGP Attention)**

In GAGP attention, the idea is that the input nodes are divided into several groups using the content-based grouping method which is then used to calculate the local attention within the groups, and summarized nodes are generated from each group. Each group summarized nodes are concatenated to compute another self-attention to reflect the information interaction among groups.

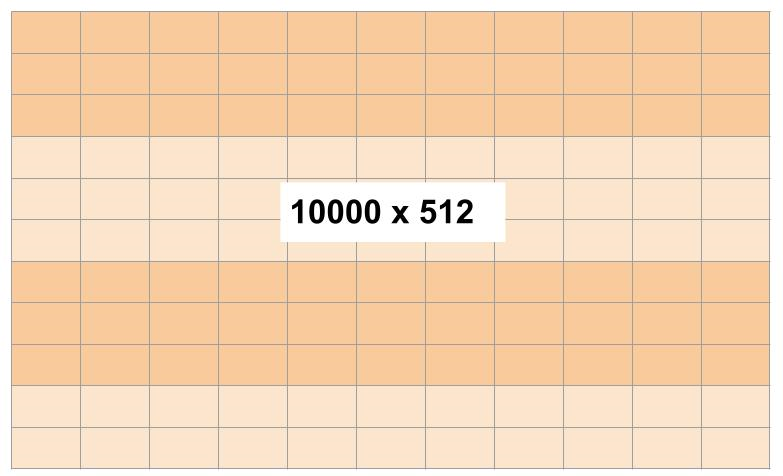
**Derivation Process**

**Step 1: Input Sequence**

Given a set of input sequence , where for each ). represent the length of the input node and represent the dimension of the word embedding.

**Practical Example:**

For clearer explanation, let take with 512 word embedding vector.



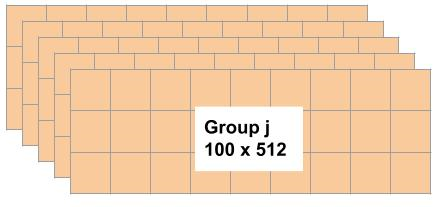
**Step 2: Grouping**

The input nodes are divided into groups using local sensitive hashing to capture the context meaning of the tokens. Let represent the group set which is defined as:

Where represent each group, reprensents numbers of groups,is the length of input nodes per group. Also .

**Example:**

From the example in step 1, 10000 input tokens can be divided into 100 groups and each group has 100 tokens of 512 word embeddings.



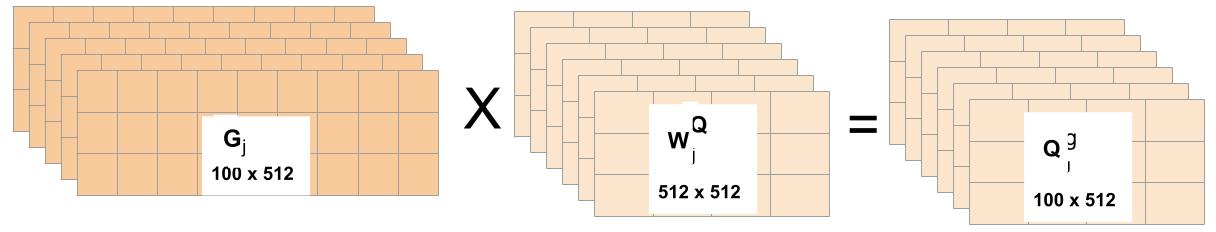
**Step 3: Local Attention**

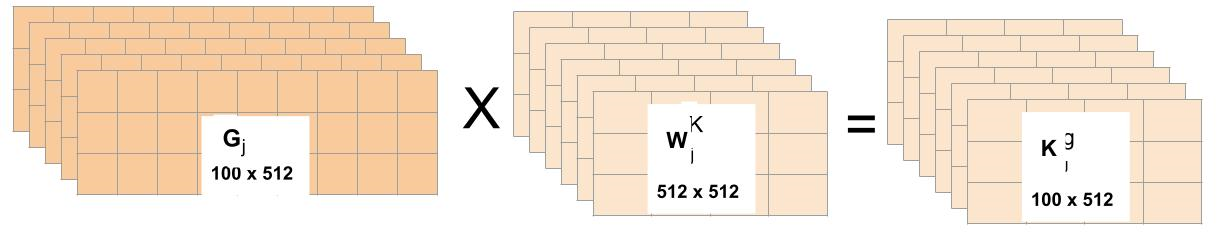
Local attention is been computed to capture the locality information using single-attention approach. This is done by the means of calculating the Queries , Keys and Values using linear projection for each group first which can be defined as:

Where , .

The local attention for each group can be calculated. Let be the output of the local attention of each group , which can be computed as:

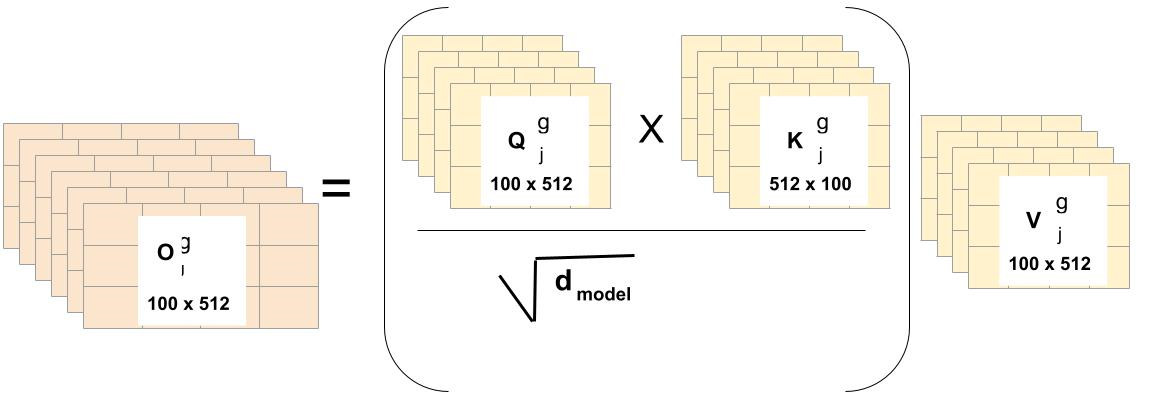
Example:







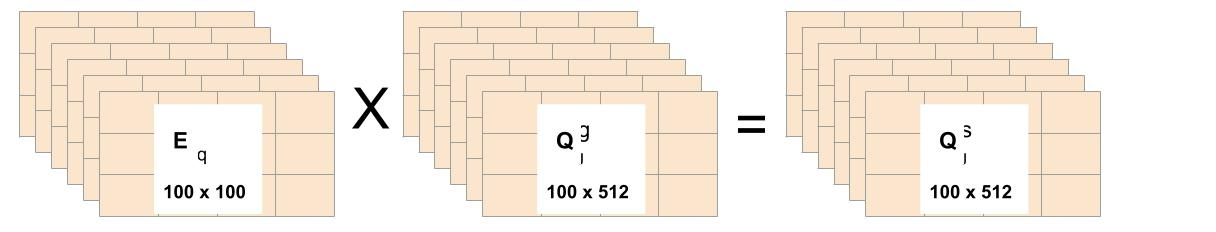
Now, calculate the local attention



**Step 4: Summarized Group Information**

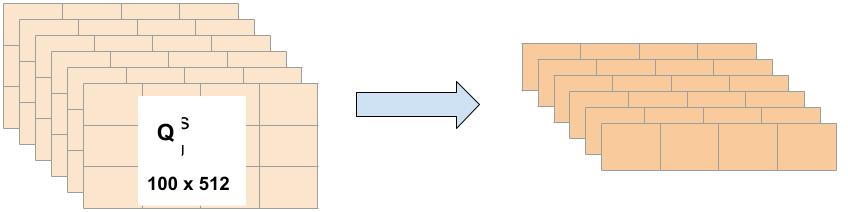
The summarized group nodes are computed. Let be the summarized group nodes which is define as:

is the length of group. To achieve this, a linear projection matrices (i.e canonical self-attention method) are used to calculate the queries, keys and values for the summmarized node for group . This can be computed as:



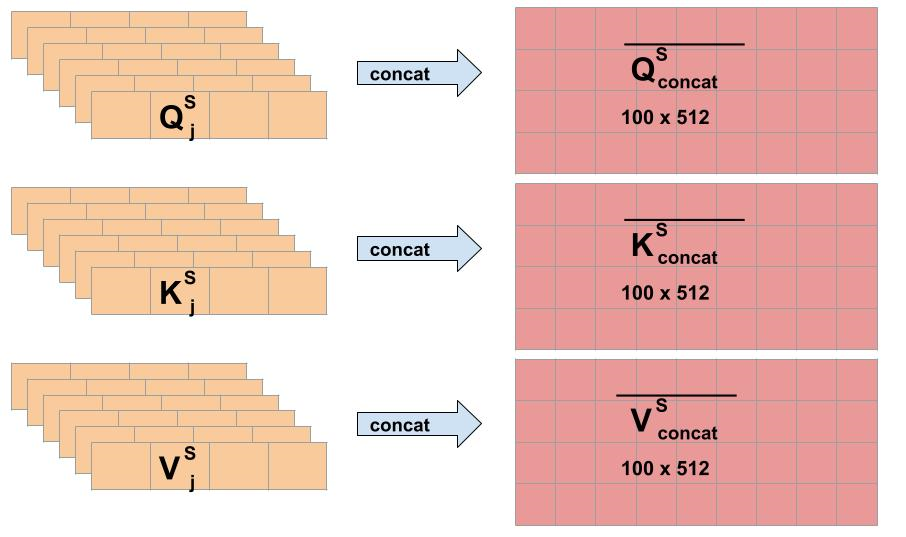
**Step 5: Max pooling**

Max-pooling is then applied on , and to form , and



**Step 6: Global Attention**

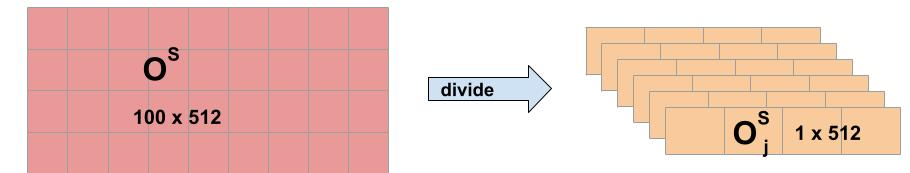
Here, single self attention is also used to calculate the global attention and this is done by concatenating every , and into the , and . Recall that is the length of .



Then , and are used to calculate the summarized node sellf-attention to reflect the global information. Let  be the output of the global attention, which can be computed as:

**Step 7: Divide**

is now divided into m segments  this represent the output reflects global features with local information in group .



**Step 8: Combining Local and Global Attention**

The output from global attention is then added using element-wise addition to the output of the local attention which makes each group have a combined representation that captures both the local and global context. This can be represented as:

Where are learnable parameters for each group that shows how much ration of the global output is reflecting in the local attention

**Step 9: Final Output**

All are concatenated to form a final output of the GAGP Attention where .

**3.3.1.1 GAGP Attention Algorithm**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm 6: GAGP Attention Algorithm**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm GAGP\_Attention(input\_sequence, grouping\_method, local\_attention\_method, global\_attention\_method):**

**Input:**

n = input\_sequence

l = length(input\_sequence)

d\_model = dimension\_of\_word\_embedding

**Output:**

Final Output

**Algorithm Steps:**

1. **groups = grouping\_method(n, l, d\_model**) // *Grouping*

m = number\_of\_groups

lg = length\_of\_input\_nodes\_per\_group

1. **local\_attention\_output = local\_attention\_method(groups, d\_model)** *// Local Attention*
2. **summarized\_group\_nodes = summarize\_group\_nodes(local\_attention\_output)** *// Summarized Group Information*
3. **max\_pooled\_values = max\_pool(summarized\_group\_nodes)**  *// Max Pooling*
4. **global\_attention\_output = global\_attention\_method(max\_pooled\_values)** *// Global Attention*

6. **divided\_segments = divide\_global\_attention(global\_attention\_output, m)** *// Divide*

7.  **combined\_output = combine\_local\_and\_global(local\_attention\_output, divided\_segments)**  *// Combining Local and Global Attention*

8. **final\_output = concatenate\_outputs(combined\_output)**  *// Final Output*

**9. return** **final\_output**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm 7: Local attention method**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm local\_attention\_method(groups, d\_model):**

**Algorithm Steps:**

*// Initialize an empty list to store local attention outputs for each group*

1. local\_attention\_outputs = []

*// Linear Projections*

2. for each group in groups:

3. Q\_j^g = linear\_projection(group, W\_Q) # Linear projection for Queries

4. K\_j^g = linear\_projection(group, W\_K) # Linear projection for Keys

5. V\_j^g = linear\_projection(group, W\_V) # Linear projection for Values

*//Step 2: Compute Local Attention Scores*

1. local\_attention\_scores = compute\_local\_attention(Q\_j^g, K\_j^g)

*//Step 3: Weighted Sum of Values*

1. local\_attention\_output = weighted\_sum\_of\_values(local\_attention\_scores, V\_j^g)

*//Append the local attention output to the list*

8. local\_attention\_outputs.append(local\_attention\_output)

9. **return local\_attention\_outputs**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm 8: summarize group nodes**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm** **summarize\_group\_nodes(local\_attention\_output):**

*// Linear Projections for Summarized Nodes*

1. Q\_j^s = linear\_projection(local\_attention\_output, E\_q) *// Linear projection for Queries*

2. K\_j^s = linear\_projection(local\_attention\_output, E\_k) *// Linear projection for Keys*

3. V\_j^s = linear\_projection(local\_attention\_output, E\_v) *// Linear projection for Values*

*// Max Pooling*

4. Q\_j^s\_bar = max\_pooling(Q\_j^s) *// Max pooling for Queries*

5. K\_j^s\_bar = max\_pooling(K\_j^s) *// Max pooling for Keys*

6. V\_j^s\_bar = max\_pooling(V\_j^s) *// Max pooling for Values*

*// Step 3: Global Attention*

7. O^S = global\_attention(Q\_j^s\_bar, K\_j^s\_bar, V\_j^s\_bar) *//Global attention computation*

*//Step 4: Divide*

8. O\_j^S = divide\_global\_attention(O^S) *// Divide the global attention into segments*

9. **return O\_j^S**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm 9: max\_pooling**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm** **max\_pooling(matrix):**

*//Assuming matrix is a 2D array or tensor*

1. rows, cols = dimensions(matrix)

*//Initialize an array to store the maximum values along each column*

1. max\_values = array of size cols, initialized with negative infinity

*//Iterate over each row*

1. for row in range(rows):

*//Iterate over each column*

1. for col in range(cols):

*//Update the maximum value for each column*

1. max\_values[col] = max(max\_values[col], matrix[row][col])

1. **return max\_values**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm 10: global\_attention**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm** **global\_attention(Q\_j^s\_bar, K\_j^s\_bar, V\_j^s\_bar):**

*//Assuming Q\_j^s\_bar, K\_j^s\_bar, V\_j^s\_bar are 2D arrays or tensors*

1. rows, cols = dimensions(Q\_j^s\_bar)

*//Compute attention scores using scaled dot-product attention*

2. attention\_scores = softmax(Q\_j^s\_bar @ K\_j^s\_bar^T / sqrt(cols)) //

*//Weighted sum of values using attention scores*

3. weighted\_sum = attention\_scores @ V\_j^s\_bar

4.  **return weighted\_sum**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm 11: divide global attention**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm** **divide\_global\_attention(O^S, m):**

*//Assuming O^S is a 2D array or tensor with shape (m, d\_model)*

1. lg, d\_model = dimensions(O^S)

*// Initialize an empty list to store the divided attention segments*

2. divided\_segments = []

*// Divide O^S into m segments*

3. for j in range(m):

4. start\_idx = j \* lg

5. end\_idx = (j + 1) \* lg

6. segment = O^S[start\_idx : end\_idx, :]

7. divided\_segments.append(segment)

8.  **return divided\_segments**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm 12: divide global attention**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm** **combine\_local\_and\_global(local\_attention\_output, divided\_segments, alpha, beta):**

*// Assuming local\_attention\_output is a 2D array or tensor with shape (m \* lg, d\_model)*

1. m, lg, d\_model = dimensions(local\_attention\_output)

*// Initialize an empty list to store the combined attention segments*

2. combined\_segments = []

*// Combine local and global attention for each group*

3. for j in range(m):

4. local\_segment = local\_attention\_output[j \* lg : (j + 1) \* lg, :]

5. global\_segment = divided\_segments[j]

*// Apply element-wise addition with weights alpha and beta*

6. combined\_segment = alpha \* local\_segment + beta \* global\_segment

7. combined\_segments.append(combined\_segment)

8. **return combined\_segments**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm 13: concatenate\_outputs**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm:** **concatenate\_outputs(combined\_outputs):**

*// Assuming combined\_outputs is a list of 2D arrays or tensors*

*//Each element in combined\_outputs should have shape (lg, d\_model)*

*// Concatenate the combined outputs along the first axis*

1. concatenated\_output = concatenate\_along\_axis(combined\_outputs, axis=0)

2. **return concatenated\_output**

**3.3.1.2 Time and Space Complexity for GAGP Attention**

A. **Time Complexity Breakdown:**

1. **Grouping Method:**

Time Complexity: )

2. **Local Attention Method:**

Time Complexity: )

3. **Summarize Group Nodes:**

Linear Projections: )

Max Pooling: )

4. **Global Attention Method:**

Time Complexity: )

5. **Divide Global Attention:**

Time Complexity: )

6. **Combine Local and Global:**

Time Complexity: )

7. **Concatenate Outputs:**

Time Complexity: )

**Overall Time Complexity: )**

**B. Space Complexity Breakdown:**

1. **Grouping Method:**

Space Complexity: )

2. **Local Attention Method:**

Space Complexity: )

3. **Summarize Group Nodes:**

Space Complexity: )

4. **Global Attention Method:**

Space Complexity: )

5. **Divide Global Attention:**

Space Complexity: )

6. **Combine Local and Global:**

Space Complexity: )

7. **Concatenate Outputs:**

Space Complexity: )

**Overall Space Complexity: )**

**3.3.2 Grouped-Average-Global-Pooling + MultiHead Attention (GAGP+MultiHead Attention)**

The idea of GAGP+MultiHead is similar to GAGP attention where the input nodes are divided into several groups, local attention is then calculated within the groups and summary nodes are generated for each group by concatenating the summarized nodes to compute the global attention. But one major difference of this approach is that instead of performing a single attention function on both the local and global attention, a multihead attention function is used with the main of further reducing memory usage. The overview idea is that the word embedding dimension is divided times with different learned linear projection to a lower dimension .

**Derivation Process**

**Step 1: Input Sequence**

Given a set of input sequence , where for each ). represent the length of the input node and represent the dimension of the word embedding.

**Step 2: Grouping**

The input nodes are divided into groups using local sensitive hashing to capture the context meaning of the tokens. Let represent the group set which is defined as:

Where represent each group, reprensents numbers of groups,is the length of input nodes per group. Also .

**Step 3: Local Attention**

The local attention captures the locality information using multihead attention approach for each group. Here is the process:

Firstly, queries, keys and values are generated by the means of linear projection for each group and attention head for group also (i.e the idea is to have multiple head for each group). This can be computed as:

Given that , and are randmonly generated. Therefore, the queries, keys and values for head can be computed as :

Where , is the number of attention head per group, is the number of group and represent number of head per time. So, the local attention output of each head in a group , can b e computed as:

Then the output of each groups can be computed by concatenating all heads per group and then it project linearly. This is shown as:

Where , . Therefore, .

**Step 4: Max pooling**

Max-pooling is then applied on to form . Then all are concatenated to form the summary group output .

**Step 5: Global Attention with Multihead**

Global attention is been computed using multihead-attention approach. This is done by the means of calculating the Queries , Keys and Values using linear projection for summary group output which can be defined as:

Given that , and are randmonly generated. Therefore, the queries, keys and values for head can be computed as:

Where , is the number of attention head per group and represent number of head per time.

With this, global attention output can be computed:

Where;

, , and . Therefore, .

**Step 6: Divide**

is now divided into m segments this represent the output reflects global features with local information in group .

**Step 7: Combining Local and Global Attention**

The output from global attention is then added using element-wise addition to the output of the local attention which makes each group have a combined representation that captures both the local and global context. This can be represented as:

Where are learnable parameters for each group that shows how much ration of the global output is reflecting in the local attention

**Step 8: Final Output**

All are concatenated to form a final output of the GG-LSTM attention where .

**3.3.2.1 GAGP+MultiHead Attention Algorithm**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm 14: GAGP+MultiHead Attention Algorithm**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm GAGP\_Attention(input\_sequence, grouping\_method, local\_attention\_method, global\_attention\_method):**

**Input:**

n = input\_sequence

l = length(input\_sequence)

d\_model = dimension\_of\_word\_embedding

**Output:**

Final Output

**Algorithm Steps:**

1. **groups = grouping\_method(n, l, d\_model**) // *Grouping*

m = number\_of\_groups

lg = length\_of\_input\_nodes\_per\_group

1. ***// Step 3: Local Attention with MultiHead***

local\_attention\_output = []

for group in groups:

Q\_ji^g, K\_ji^g, V\_ji^g = generate\_queries\_keys\_values(group, W\_Q, W\_K, W\_V)

heads\_output = multihead\_attention(Q\_ji^g, K\_ji^g, V\_ji^g, h)

local\_attention\_output.append(concatenate\_heads\_and\_project(heads\_output, W\_O))

1. **summarized\_group\_nodes = max\_pool(local\_attention\_output)**  *// Max Pooling*
2. ***//Global Attention with MultiHead***

Q\_i^S, K\_i^S, V\_i^S = generate\_queries\_keys\_values(summarized\_group\_nodes, W\_SQ, W\_SK, W\_SV)

global\_attention\_output = multihead\_attention(Q\_i^S, K\_i^S, V\_i^S, h)

5. **divided\_segments = divide\_global\_attention(global\_attention\_output, m)** *// Divide*

6.  **combined\_output = combine\_local\_and\_global(local\_attention\_output, divided\_segments, alpha, beta)** *// Combining Local and Global Attention*

7. **final\_output = concatenate\_outputs(combined\_output)**  *// Final Output*

**8. return** **final\_output**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

// **Algorithm 15: Generate Queries, Keys, and Values for Multihead Attention**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm generate\_queries\_keys\_values(group, W\_Q, W\_K, W\_V):**

//Initialize lists to store queries, keys, and values for each head

1. queries\_list = []

2. keys\_list = []

3. values\_list = []

*//Linear Projections for Queries, Keys, and Values*

4. for i in range(number\_of\_attention\_heads):

*//Linear Projection for Queries*

5. Q\_ji^g = linear\_projection(group, W\_Q[i]) *// Assuming W\_Q[i] is the i-th set of parameters for linear projection*

*//Linear Projection for Keys*

6. K\_ji^g = linear\_projection(group, W\_K[i]) *//Assuming W\_K[i] is the i-th set of parameters for linear projection*

*// Linear Projection for Values*

7. V\_ji^g = linear\_projection(group, W\_V[i]) *// Assuming W\_V[i] is the i-th set of parameters for linear projection*

*//Append queries, keys, and values to the lists*

8. queries\_list.append(Q\_ji^g)

9. keys\_list.append(K\_ji^g)

10. values\_list.append(V\_ji^g)

11. **return queries\_list, keys\_list, values\_list**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**//Algorithm16: Multihead Attention**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm** **multihead\_attention(Q, K, V, h):**

*//Initialize lists to store heads' output*

1. heads\_output = []

*// Perform attention for each head*

2. for i in range(h):

*//Compute attention scores using scaled dot-product attention*

3. attention\_scores = softmax(Q @ K^T / sqrt(d\_k)) //

*//Weighted sum of values using attention scores*

4. weighted\_sum = attention\_scores @ V

//Append the output of the current head to the list

5. heads\_output.append(weighted\_sum)

6. **return heads\_output**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**//Algorithm:17: Concatenate Heads' Output and Project**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm** **concatenate\_heads\_and\_project(heads\_output, W\_O):**

// Concatenate the outputs of all heads along the last axis

1. concatenated\_output = concatenate\_along\_axis(heads\_output, axis=-1)

//Linear projection using the parameter matrix W\_O

2. projected\_output = linear\_projection(concatenated\_output, W\_O)

3. **return projected\_output**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**//Algorithm:18: Max Pooling**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm** **max\_pool(input\_matrix):**

//input\_matrix is a 2D array or tensor

1. max\_values = max\_pooling(input\_matrix)

2. **return max\_values**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**//Algorithm 19: Divide Global Attention Output**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm** **divide\_global\_attention(global\_attention\_output, m):**

*// Assuming global\_attention\_output is a 2D array or tensor with shape (m, d\_model)*

1. divided\_segments = divide\_global\_attention(global\_attention\_output, m)

2. **return divided\_segments**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***//Algorithm 20: Combine Local and Global Attention***

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm** **combine\_local\_and\_global(local\_attention\_output, divided\_segments, alpha, beta):**

*// Assuming local\_attention\_output and divided\_segments are lists of 2D arrays or tensors*

1. combined\_segments = combine\_local\_and\_global(local\_attention\_output, divided\_segments, alpha, beta)

2. **return combined\_segments**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**//Algorithm 21: Concatenate Outputs** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm** **concatenate\_outputs(combined\_output):**

*// Assuming combined\_output is a list of 2D arrays or tensors*

1. concatenated\_output = concatenate\_along\_axis(combined\_output, axis=0)

2. return concatenated\_output

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**//Algorithm 22: Combine Local and Global Attention** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithm** **combine\_local\_and\_global(local\_attention\_output, divided\_segments, alpha, beta):**

*// Initialize a list to store the combined attention segments*

1. combined\_segments = []

*//Combine local and global attention for each group*

2. for j in range(number\_of\_groups):

3. local\_segment = local\_attention\_output[j] # Assuming local\_attention\_output[j] is the local attention output for group j

4. global\_segment = divided\_segments[j] # Assuming divided\_segments[j] is the global attention output segment for group j

*// Apply element-wise addition with weights alpha and beta*

5. combined\_segment = alpha \* local\_segment + beta \* global\_segment

//Append the combined segment to the list

6. combined\_segments.append(combined\_segment)

7. **return combined\_segments**

**3.3.1.2 Time and Space Complexity for GAGP+MultiHead Attention**

**Time and Space Complexity Analysis for GAGP Attention Algorithm**

**Time Complexity Breakdown:**

1. **Grouping Method:**

Time Complexity: )

2. **Local Attention Method (MultiHead):**

- Time Complexity: )

**Generate Queries, Keys, and Values:**

Time Complexity: )

**MultiHead Attention:**

Time Complexity: )

**Concatenate Heads and Project:**

Time Complexity: )

**Overall Time Complexity: )**

3. **Summarize Group Nodes (Max Pooling):**

Time Complexity:

4. **Global Attention Method (MultiHead):**

Time Complexity: )

5. **Divide Global Attention:**

Time Complexity:

6. **Combine Local and Global:**

Time Complexity:

7. **Concatenate Outputs:**

Time Complexity:

Overall Time Complexity is dominated by the Local Attention Method (MultiHead), approximately **).**

**B. Space Complexity Breakdown:**

1. **Grouping Method:**

Space Complexity:

2. **Local Attention Method (MultiHead):**

Space Complexity:

**Generate Queries, Keys, and Values:**

Space Complexity:

**MultiHead Attention:**

Space Complexity:

**Concatenate Heads and Project:**

Space Complexity:

3. **Summarize Group Nodes (Max Pooling):**

Space Complexity:

4. **Global Attention Method (MultiHead):**

Space Complexity:

5. **Divide Global Attention:**

Space Complexity:

6. **Combine Local and Global:**

Space Complexity:

7. **Concatenate Outputs:**

Space Complexity:

Overall Space Complexity is dominated by the Local Attention Method (MultiHead), approximately .

**3.4 Computational Complexity Analysis**

This approach increases the number of dot-product calculations in the self-attention model, which is given as .

Where is the number of groups and is the number of nodes in each groups. The length of input sequence is then . is the length of summarized node for each group which qualifies that , the algorithm can be said to have a complexity of order and is a fixed length representing the number od nodes included in a single group.

Therefore, the final complexity order becomes linear with respect to sequence length .

**CHAPTER FOUR**

**4.0 IMPLEMENTATION PLANNING**

**4.1 System Specification**

The system specification that will be used for this work can be grouped in terms of hardware and software requirements. The details of these requirements are presented as follows:

* + 1. **Hardware Requirements**

All the training experiments will be carried out on Google Colaboratory (Colab) which is a cloud based collaborative platform for conducting research in Artificial Intelligence. Due to the high cost of setting up a local workstation with Tensor Processing Units (TPUs) which is needed to speed up the processing time of the transformer models. The compute engine specifications that will be used are TPU runtime consists of:

i. an Intel Xeon CPU @2.30 GHz,

ii. 13 GB GDDR5 VRAM,

iii. a cloud TPU with 180 teraflops of computational power.

**4.1.2 Software Requirement**

Python programming language will be the main programming language for implementation due to the availability of vast amount of open-source python-based libraries and packages commonly used for Natural Language Processing tasks. The major libraries and packages that will be used include:

1. Tokenizer for text tokenization
2. Numpy for array processing
3. Pytorch for developing deep learning models
4. Wandb for experiment tracking

**4.2 Datasets**

GG-Transformer (GGT) will be evaluated based on its runtime time, memory usage, BERTScore, Human Evaluation and compared with four baselines-Transformers which are Full Attention Transfromer, T5, BART, Longformer on commonly used dataset for multiple document summarization tasks

1. **Multiple Document Text Summarization Task:**
2. OPOSUM Dataset
3. WikiSum Dataset
4. Scisumm Dataset
5. Multi-News Dataset
6. Multi-XScience Dataset

**CHAPTER FIVE**

**5.0 CONCLUSION AND RECOMMENDATIONS**

**5.1 Expected Contribution to Knowledge**

The major expected contribution of this work tends towards proposing new attention models from existing approaches, a class of efficient transformers with linear memory complexity. Also, the model can the used in the field of deep learning for natural language processing tasks such as multiple document summarization tasks. Here are some of the main expected contributions to knowledge from this proposed study:

1. **Efficient Attention Mechanism:**

The grouping strategy in the proposed methods should allow for more efficient attention computation, especially in the case where the input sequences are large. This could potentially reduce the time and space complexity of the attention mechanism.

1. **Adaptive Attention:**

The combination of local and global attention with the inclusion of multi-head attention could offer a more adaptive approach to handling different patterns and structures in data.

1. **Hierarchical Processing:**

The concept of the proposed method of grouping and summarizing in GAGP uses a hierarchical processing approach which may be beneficial in cases where understanding local context and global context simultaneously is important such as in natural language understanding tasks

1. **Improved Memory Efficiency:**

GAGP+MultiHead uses a multi-head attention mechanism, which has shown its effectiveness in capturing diverse patterns. So this could lead to improved memory efficiency in capturing various features and relationships in the data.

1. **Novel integration of techniques:**

The combination of local attention, global attention, and multi-head attention in GAGP+MultiHead demonstrates the integration of different attention mechanisms. This could lead to new insights into how different mechanisms can complement each other.

1. **Contributions to Attention Mechanism Understanding:**

These methods may contribute to the ongoing exploration and understanding of attention mechanisms in neural networks. The ability to combine and customize attention strategies may shed light on the importance of different attention components in learning complex patterns.

**References**

Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebrón, F., & Sanghai, S. (2023). *GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints*. http://arxiv.org/abs/2305.13245

Alec Radford, Karthik Narasimhan, Tim Salimans, & lya Sutskever. (2018). *Improving Language Understanding by Generative Pre-Training*. https://gluebenchmark.com/leaderboard

Arnab, A., Dehghani, M., Heigold, G., Sun, C., Luči´c, M. L., & Schmid, C. (2021). *ViViT: A Video Vision Transformer*.

Bahdanau, D., Cho, K., & Bengio, Y. (2014). *Neural Machine Translation by Jointly Learning to Align and Translate*. http://arxiv.org/abs/1409.0473

Beltagy, I., Peters, M. E., & Cohan, A. (2020a). *Longformer: The Long-Document Transformer*. http://arxiv.org/abs/2004.05150

Beltagy, I., Peters, M. E., & Cohan, A. (2020b). *Longformer: The Long-Document Transformer*. http://arxiv.org/abs/2004.05150

Britz, D., Goldie, A., Luong, M.-T., & Le, Q. (2017). *Massive Exploration of Neural Machine Translation Architectures*. http://arxiv.org/abs/1703.03906

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., … Amodei, D. (2020). *Language Models are Few-Shot Learners*. http://arxiv.org/abs/2005.14165

Buluc, A., & Gilbert, J. R. (2008). *Challenges and Advances in Parallel Sparse Matrix-Matrix Multiplication*.

Caciularu, A., Cohan, A., Beltagy, I., Peters, M. E., Cattan, A., & Dagan, I. (2021). *CDLM: Cross-Document Language Modeling*. http://arxiv.org/abs/2101.00406

Chan, W., Jaitly, N., Le, Q. v., & Vinyals, O. (2015). *Listen, Attend and Spell*. http://arxiv.org/abs/1508.01211

Chen, B., Dao, T., Winsor, E., Song, Z., Rudra, A., & Ré, C. (2021). *Scatterbrain: Unifying Sparse and Low-rank Attention Approximation*. https://github.com/HazyResearch/scatterbrain

Child, R., Gray, S., Radford, A., & Sutskever, I. (2019). *Generating Long Sequences with Sparse Transformers*. http://arxiv.org/abs/1904.10509

Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*. http://arxiv.org/abs/1406.1078

Choromanski, K., Likhosherstov, V., Dohan, D., Song, X., Gane, A., Sarlos, T., Hawkins, P., Davis, J., Mohiuddin, A., Kaiser, L., Belanger, D., Colwell, L., & Weller, A. (2020). *Rethinking Attention with Performers*. http://arxiv.org/abs/2009.14794

Chorowski, J., Bahdanau, D., Cho, K., & Bengio, Y. (2014). *End-to-end Continuous Speech Recognition using Attention-based Recurrent NN: First Results*. http://arxiv.org/abs/1412.1602

Correia, G. M., Niculae, V., & Martins, A. F. T. (2019). *Adaptively Sparse Transformers*. http://arxiv.org/abs/1909.00015

Dai, Z., Lai, G., Yang, Y., & Le, Q. V. (2020). *Funnel-Transformer: Filtering out Sequential Redundancy for Efficient Language Processing*.

Dai, Z., Yang, Z., Yang, Y., Carbonell, J., Le, Q. V., & Salakhutdinov, R. (2019). *Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context*. http://arxiv.org/abs/1901.02860

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018). *Pre-training of deep bidirectional transformers for language understanding.* http://arxiv.org/abs/1810.04805

di Gangi, M. A., Negri, M., & Turchi, M. (2019). Adapting transformer to end-to-end spoken language translation. *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*, *2019-September*, 1133–1137. https://doi.org/10.21437/Interspeech.2019-3045

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., & Houlsby, N. (2020). *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. http://arxiv.org/abs/2010.11929

Du, N., Huang, Y., Dai, A. M., Tong, S., Lepikhin, D., Xu, Y., Krikun, M., Zhou, Y., Yu, A. W., Firat, O., Zoph, B., Fedus, L., Bosma, M., Zhou, Z., Wang, T., Wang, Y. E., Webster, K., Pellat, M., Robinson, K., … Cui, C. (2020). *GLaM: Efficient Scaling of Language Models with Mixture-of-Experts*.

Du, Y., Meier, J., Ma, J., Fergus, R., & Rives, A. (2020). *Energy-based models for atomic-resolution protein conformations*. http://arxiv.org/abs/2004.13167

Duchi JDUCHI, J., & Singer, Y. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization \* Elad Hazan. In *Journal of Machine Learning Research* (Vol. 12).

Elnaggar, A., Heinzinger, M., Dallago, C., & Rost, B. (2019). End-to-end multitask learning, from protein 1 language to protein features without alignments 2. *BioRxiv*. https://doi.org/10.1101/864405

Fedus, W., Zoph, B., & Shazeer, N. (2022). Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. In *Journal of Machine Learning Research* (Vol. 23). http://jmlr.org/papers/v23/21-0998.html.

Glorot, X., Bordes, A., & Bengio, Y. (2011). *Deep Sparse Rectifier Neural Networks*.

Graves, A., Wayne, G., & Danihelka, I. (2014). *Neural Turing Machines*. http://arxiv.org/abs/1410.5401

Guo, Q., Qiu, X., Liu, P., Shao, Y., Xue, X., & Zhang, Z. (2019). *Star-Transformer*. http://arxiv.org/abs/1902.09113

H. Ying, F. Zhuang, F. Zhang, Y. Liu, G. Xu, X. Xie, H. Xiong, & J. Wu. (2018). *Sequential Recommender System based on Hierarchical Attention Network*.

He, P., Liu, X., Gao, J., & Chen, W. (2020). *DeBERTa: Decoding-enhanced BERT with Disentangled Attention*. http://arxiv.org/abs/2006.03654

Ho, J., Kalchbrenner, N., Weissenborn, D., & Salimans, T. (2019). *Axial Attention in Multidimensional Transformers*. http://arxiv.org/abs/1912.12180

Huang, C.-Z. A., Vaswani, A., Uszkoreit, J., Shazeer, N., Simon, I., Hawthorne, C., Dai, A. M., Hoffman, M. D., Dinculescu, M., & Eck, D. (2018). *Music Transformer*. http://arxiv.org/abs/1809.04281

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., & Brockschmidt, M. (2019). *CodeSearchNet Challenge: Evaluating the State of Semantic Code Search*. http://arxiv.org/abs/1909.09436

Jaegle, A., Borgeaud, S., Alayrac, J.-B., Doersch, C., Ionescu, C., Ding, D., Koppula, S., Zoran, D., Brock, A., Shelhamer, E., Hénaff, O., Botvinick, M. M., Zisserman, A., Vinyals, O., & Carreira, J. (2021). *Perceiver IO: A General Architecture for Structured Inputs & Outputs*. http://arxiv.org/abs/2107.14795

Jain, S., & Wallace, B. C. (2019). *Attention is not Explanation*. http://arxiv.org/abs/1902.10186

Jaszczur, S., Chowdhery, A., Research, G., Mohiuddin, A., Kaiser, Ł., Gajewski, W., Michalewski, H., & Kanerva, J. (2021). *Sparse is Enough in Scaling Transformers*. https://cutt.ly/skkFJ7a

Johnson, R., & Zhang, T. (2013). *Accelerating Stochastic Gradient Descent using Predictive Variance Reduction*. http://leon.bottou.org/projects/sgd

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., & Amodei, D. (2020). *Scaling Laws for Neural Language Models*. http://arxiv.org/abs/2001.08361

Katharopoulos, A., Vyas, A., Pappas, N., & Fleuret, F. (2020). *Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention*. http://arxiv.org/abs/2006.16236

Kingma, D. P., & Ba, J. (2014). *Adam: A Method for Stochastic Optimization*. http://arxiv.org/abs/1412.6980

Kitaev, N., Kaiser, Ł., & Levskaya, A. (2020). *Reformer: The Efficient Transformer*. http://arxiv.org/abs/2001.04451

Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., & Soricut, R. (2019). *ALBERT: A Lite BERT for Self-supervised Learning of Language Representations*. http://arxiv.org/abs/1909.11942

Le Roux, N., Schmidt, M., & Bach, F. (2012). *A Stochastic Gradient Method with an Exponential Convergence Rate for Finite Training Sets*.

Lee, J., Lee, Y., Kim, J., Kosiorek, A. R., Choi, S., & Teh, Y. W. (2019). *Set Transformer: A Framework for Attention-based Permutation-Invariant Neural Networks*.

Lepikhin, D., Lee, H., Xu, Y., Chen, D., Firat, O., Huang, Y., Krikun, M., Shazeer, N., & Chen, Z. (2020). *GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding*. http://arxiv.org/abs/2006.16668

Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., & Zettlemoyer, L. (2019). *BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension*. http://arxiv.org/abs/1910.13461

Likhosherstov, V., Choromanski, K., & Weller, A. (2021). *On the Expressive Power of Self-Attention Matrices*. http://arxiv.org/abs/2106.03764

Liu, P. J., Saleh, M., Pot, E., Goodrich, B., Sepassi, R., Kaiser, L., & Shazeer, N. (2018). *Generating Wikipedia by Summarizing Long Sequences*. http://arxiv.org/abs/1801.10198

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., & Stoyanov, V. (2019). *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. http://arxiv.org/abs/1907.11692

Lu, J., Xiong, C., Parikh, D., & Socher, R. (n.d.). *Knowing When to Look: Adaptive Attention via A Visual Sentinel for Image Captioning*.

Luo, H., Zhang, S., Lei, M., & Xie, L. (2020). *Simplified Self-Attention for Transformer-based End-to-End Speech Recognition*. http://arxiv.org/abs/2005.10463

Luong, M.-T., Pham, H., & Manning, C. D. (2015a). *Effective Approaches to Attention-based Neural Machine Translation*. http://arxiv.org/abs/1508.04025

Luong, M.-T., Pham, H., & Manning, C. D. (2015b). *Effective Approaches to Attention-based Neural Machine Translation*. http://arxiv.org/abs/1508.04025

Ma, D., Li, S., Zhang, X., & Wang, H. (2017). *Interactive Attention Networks for Aspect-Level Sentiment Classification*. http://arxiv.org/abs/1709.00893

Madani, A., McCann, B., Naik, N., Keskar, N. S., Anand, N., Eguchi, R. R., Huang, P.-S., & Socher, R. (2020). *ProGen: Language Modeling for Protein Generation*. http://arxiv.org/abs/2004.03497

Mcculloch, W. S., Lrerr, W., & Pitts, H. (1943). *A LOGICAL CALCULUS OF THE IDEAS IMMANENT IN NERVOUS ACTIVITY*.

Nair, V., & Hinton, G. E. (n.d.). *Rectified Linear Units Improve Restricted Boltzmann Machines*.

Nguyen, T. M., Suliafu, V., Osher, S. J., Chen, L., & Wang, B. (2022). *FMMformer: Efficient and Flexible Transformer via Decomposed Near-field and Far-field Attention*.

Niu, Z., Zhong, G., & Yu, H. (2021). A review on the attention mechanism of deep learning. *Neurocomputing*, *452*, 48–62. https://doi.org/10.1016/j.neucom.2021.03.091

Qiu, J., Ma, H., Levy, O., Yih, S. W., Wang, S., & Tang, J. (2019). *Blockwise Self-Attention for Long Document Understanding*. http://arxiv.org/abs/1911.02972

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (n.d.). *Language Models are Unsupervised Multitask Learners*. https://github.com/codelucas/newspaper

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). *Language Models are Unsupervised Multitask Learners*. https://github.com/codelucas/newspaper

Rae, J. W., Potapenko, A., Jayakumar, S. M., & Lillicrap, T. P. (2019). *Compressive Transformers for Long-Range Sequence Modelling*. http://arxiv.org/abs/1911.05507

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2020). Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. In *Journal of Machine Learning Research* (Vol. 21). http://jmlr.org/papers/v21/20-074.html.

Ren, H., Dai, H., Dai, Z., Yang, M., Leskovec, J., Schuurmans, D., & Dai, B. (n.d.). *Combiner: Full Attention Transformer with Sparse Computation Cost*. https://github.com/google-research/google-

Rives, A., Meier, J., Sercu, T., Goyal, S., Lin, Z., Liu, J., Guo, D., Ott, M., Lawrence Zitnick, C., Ma, J., & Fergus, R. (2019). Biological structure and function emerge from scaling unsupervised learning to 250 million protein sequences. *BioArxiv*. https://doi.org/10.1073/pnas.2016239118/-/DCSupplemental

Robbins, H., & Monro, S. (1951). A Stochastic Approximation Method. In *Source: The Annals of Mathematical Statistics* (Vol. 22, Issue 3).

Rosenblatt, Frank. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review, 65(6), 386.*, *65*(6), 386.

Roy, A., Saffar, M., Vaswani, A., & Grangier, D. (2020). Efficient Content-Based Sparse Attention with Routing Transformers. *Proceedings of TACL*. https://doi.org/10.1162/tacl

Ruder, S. (2016). *An overview of gradient descent optimization algorithms*. http://arxiv.org/abs/1609.04747

Rumelhart, D. E. H. G. E. and W. R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536.

Sanh, V., Debut, L., Chaumond, J., & Wolf, T. (2019). *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*. http://arxiv.org/abs/1910.01108

Serrano, S., & Smith, N. A. (2019). *Is Attention Interpretable?* http://arxiv.org/abs/1906.03731

Soldaini, L., & Moschitti, A. (2020). *The Cascade Transformer: an Application for Efficient Answer Sentence Selection*. http://arxiv.org/abs/2005.02534

Song, S., Lan, C., Xing, J., Zeng, W., & Liu, J. (2017). *An End-to-End Spatio-Temporal Attention Model for Human Action Recognition from Skeleton Data*. www.aaai.org

Sordoni, A., Bachman, P., Trischler, A., & Bengio, Y. (2016). *Iterative Alternating Neural Attention for Machine Reading*. http://arxiv.org/abs/1606.02245

Su, C., Huang, H., Shi, S., Jian, P., & Shi, X. (2020). Neural machine translation with Gumbel Tree-LSTM based encoder. *Journal of Visual Communication and Image Representation*, *71*. https://doi.org/10.1016/j.jvcir.2020.102811

Sun, S., Cao, Z., Zhu, H., & Zhao, J. (2019). *A Survey of Optimization Methods from a Machine Learning Perspective*. http://arxiv.org/abs/1906.06821

Sutskever Google, I., Vinyals Google, O., & le Google, Q. v. (n.d.). *Sequence to Sequence Learning with Neural Networks*.

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). *Sequence to Sequence Learning with Neural Networks*. http://arxiv.org/abs/1409.3215

Tay, Y., Bahri, D., Metzler, D., Juan, D.-C., Zhao, Z., & Zheng, C. (2021). *Synthesizer: Rethinking Self-Attention for Transformer Models*. https://github.com/tensorflow/mesh.

Tay, Y., Bahri, D., Yang, L., Metzler, D., & Juan, D.-C. (2020). *Sparse Sinkhorn Attention*.

Tay, Y., Dehghani, M., Abnar, S., Shen, Y., Bahri, D., Pham, P., Rao, J., Yang, L., Ruder, S., & Metzler, D. (2020). *Long Range Arena: A Benchmark for Efficient Transformers*. http://arxiv.org/abs/2011.04006

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). *Attention Is All You Need*. http://arxiv.org/abs/1706.03762

Wang, S., Hu, L., Cao, L., Huang, X., Lian, D., & Liu, W. (2018). *Attention-Based Transactional Context Embedding for Next-Item Recommendation*. www.aaai.org

Wang, S., Li, B. Z., Khabsa, M., Fang, H., & Ma, H. (2020). *Linformer: Self-Attention with Linear Complexity*. http://arxiv.org/abs/2006.04768

Wang, S., Zhou, L., Gan, Z., Chen, Y.-C., Fang, Y., Sun, S., Cheng, Y., & Liu, J. (2020). *Cluster-Former: Clustering-based Sparse Transformer for Long-Range Dependency Encoding*. http://arxiv.org/abs/2009.06097

Wiegreffe, S., & Pinter, Y. (2019). *Attention is not not Explanation*. http://arxiv.org/abs/1908.04626

Winata, G. I., Cahyawijaya, S., Lin, Z., Liu, Z., & Fung, P. (2019). *Lightweight and Efficient End-to-End Speech Recognition Using Low-Rank Transformer*. http://arxiv.org/abs/1910.13923

Xiong, Y., Zeng, Z., Chakraborty, R., Tan, M., Fung, G., Li, Y., & Singh, V. (2021). *Nyströmformer: A Nyström-based Algorithm for Approximating Self-Attention*. https://github.com/mlpen/Nystromformer.

Xu, K., Lei Ba, J., Kiros, R., Cho, K., Courville, A., Salakhutdinov, R., Zemel, R. S., & Bengio, Y. (2015). *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention*.

Zaheer, M., Guruganesh, G., Dubey, A., Ainslie, J., Alberti, C., Ontanon, S., Pham, P., Ravula, A., Wang, Q., Yang, L., & Research, A. G. (2020). *Big Bird: Transformers for Longer Sequences*.

Zeiler, M. D. (2012). *ADADELTA: An Adaptive Learning Rate Method*. http://arxiv.org/abs/1212.5701

Zhang, H., Gong, Y., Shen, Y., Li, W., Lv, J., Duan, N., & Chen, W. (2021). *Poolingformer: Long Document Modeling with Pooling Attention*. https://ai.google.com/research/

Zhang, P. (2018). *Adding Attentiveness to the Neurons in Recurrent Neural Networks*.

Zhu, C., Ping, W., Xiao, C., Shoeybi, M., Goldstein, T., Anandkumar, A., & Catanzaro, B. (2021). *Long-Short Transformer: Efficient Transformers for Language and Vision*. https://github.com/NVIDIA/transformer-ls.

Zoph, B., Bello, I., Kumar, S., Du, N., Huang, Y., Dean, J., Shazeer, N., & Fedus, W. (2022). *ST-MoE: Designing Stable and Transferable Sparse Expert Models*. http://arxiv.org/abs/2202.08906

# Step 1: Input Sequence

# Given input sequence n with dimensions (l, d\_model)

# l: length of the input sequence

# d\_model: dimension of word embedding

# Step 2: Grouping

# Divide input nodes into groups using local sensitive hashing

# Define group set G, each group g\_j has dimensions (lg, d\_model)

# m: number of groups

# lg: length of input nodes per group

# l = m \* lg

# Step 3: Local Attention

# For each group g\_j:

# - Generate queries, keys, and values using linear projection

# - Split into h attention heads per group

# - Perform multihead attention to capture locality information

# - Output O\_j^g for each group by concatenating and linear projection

# Step 4: Max pooling

# Apply max-pooling on O\_j^g to obtain summarized group output (O\_j^S) ̅

# Step 5: Global Attention with Multihead

# Generate queries, keys, and values for global attention using linear projection

# Perform multihead attention on summarized group output

# Output O^S representing global features with local information in group g\_j

# Step 6: Divide

# Divide O^S into m segments O\_j^S representing the output for each group

# Step 7: Combining Local and Global Attention

# Add local attention output O\_j^g with global attention output O\_j^S using element-wise addition

# α and β are learnable parameters representing the ratio of global output reflected in local attention

# Step 8: Final Output

# Concatenate all combined outputs (O\_j^g) ̃ to form the final output O of the GAGP+MultiHead Attention

# O dimensions: (mlg, d\_model)

import torch

import torch.nn as nn

import torch.nn.functional as F

class GroupedAverageGlobalPoolingMultiHeadAttention(nn.Module):

def \_\_init\_\_(self, d\_model, num\_heads, num\_groups, group\_length):

super(GroupedAverageGlobalPoolingMultiHeadAttention, self).\_\_init\_\_()

self.d\_model = d\_model

self.num\_heads = num\_heads

self.num\_groups = num\_groups

self.group\_length = group\_length

self.d\_head = self.d\_model // self.num\_heads

# Linear projections for queries, keys, and values

self.wq = nn.Linear(self.d\_model, self.d\_model)

self.wk = nn.Linear(self.d\_model, self.d\_model)

self.wv = nn.Linear(self.d\_model, self.d\_model)

# Linear projection for final output

self.dense = nn.Linear(self.d\_model, self.d\_model)

def split\_heads(self, x):

batch\_size = x.size(0)

split\_inputs = x.view(batch\_size, -1, self.num\_heads, self.d\_head)

return split\_inputs.permute(0, 2, 1, 3)

def merge\_heads(self, x):

batch\_size = x.size(0)

merged\_inputs = x.permute(0, 2, 1, 3)

return merged\_inputs.view(batch\_size, -1, self.d\_model)

def forward(self, input\_sequence):

# Step 2: Grouping

group\_inputs = input\_sequence.view(-1, self.num\_groups, self.group\_length, self.d\_model)

# Step 3: Local Attention

qs = self.wq(group\_inputs)

ks = self.wk(group\_inputs)

vs = self.wv(group\_inputs)

qs = self.split\_heads(qs)

ks = self.split\_heads(ks)

vs = self.split\_heads(vs)

local\_attention\_output = scaled\_dot\_product\_attention(qs, ks, vs, mask=None)

local\_attention\_output = self.merge\_heads(local\_attention\_output)

# Step 4: Max pooling

summarized\_group\_output, \_ = torch.max(local\_attention\_output, dim=1)

# Step 5: Global Attention with Multihead

qs\_global = self.wq(summarized\_group\_output)

ks\_global = self.wk(summarized\_group\_output)

vs\_global = self.wv(summarized\_group\_output)

qs\_global = self.split\_heads(qs\_global)

ks\_global = self.split\_heads(ks\_global)

vs\_global = self.split\_heads(vs\_global)

global\_attention\_output = scaled\_dot\_product\_attention(qs\_global, ks\_global, vs\_global, mask=None)

global\_attention\_output = self.merge\_heads(global\_attention\_output)

# Step 6: Divide

global\_attention\_divided = global\_attention\_output.view(-1, self.num\_groups, 1, self.d\_model)

# Step 7: Combining Local and Global Attention

combined\_output = local\_attention\_output + global\_attention\_divided

# Step 8: Final Output

final\_output = self.dense(combined\_output)

return final\_output

# Scaled Dot Product Attention function (you need to define this)

def scaled\_dot\_product\_attention(query, key, value, mask=None):

pass # Implement scaled dot product attention here

# Example usage:

# model = GroupedAverageGlobalPoolingMultiHeadAttention(d\_model=512, num\_heads=8, num\_groups=4, group\_length=10)

# input\_sequence = torch.randn(32, 40, 512) # Example input with batch size 32, sequence length 40, and embedding dimension 512

# output = model(input\_sequence)