

UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

UNIVERSITY OF MODENA AND REGGIO EMILIA

"Enzo Ferrari" Department of Engineering

Master's Degree in Artificial Intelligence Engineering (LM-32)

CakeStack: a Mixture of Experts mechanism of Aggregated Attention Heads

Supervisor:
Prof. Lorenzo Baraldi
Prof. Marcella Cornia
Co-supervisor:
Davide Bucciarelli
Leonardo Zini

Candidate:
Giovanni Forti
Student ID 207001

ACADEMIC YEAR 2025/2026

Contents

1 Introduction **1**

2 Preliminaries **6**

2.1 Deep Learning 6

2.1.1 The Perceptron and the linear layer 6

2.1.2 Gradient descent 11

2.2 Transformers 17

2.3 Positional encoding 22

2.4 Decoder-only architectures 25

2.5 Llama 3 26

2.6 Mixture of Experts 28

2.7 Mixture of Attention 30

3 Related work **33**

4 Methodology **41**

4.1 Architectural modifications 41

4.2 Gating strategies 43

5 Experiments **47**

5.1 Data preparation 47

5.2 Experiments Settings 48

5.3 Experiments Results 50

6 Conclusion **58**

6.1 Limitations and Future Improvements 59

Ringraziamenti **60**

References **61**

List of Figures

2.1 A plot of the ReLU function. 8
2.2 A schema of the information flow inside multiple linear layer. 9
2.3 A simple example of gradient descent in two dimensions. The arrow represents the values taken by the parameters (two in this case) across the various iterations while minimizing the function. 13
2.4 (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several Attention layers running in parallel [103]. 17
2.5 The complete encoder-decoder transformer architecture [103]. 19
2.6 Overview of Multi-Head Attention, Grouped-Query Attention, Multi-Query Attention [1]. 27
2.7 Sparsely Gated Mixture of Experts Layer. 29
2.8 Mixture of Attention layer [117]. Note that the key and value projections are shared between the heads. 31

3.1 A comparison between a classic Top-1 MoE mechanism and a Pyramidal Residual MoE [79], where orange experts represent those experts which are always kept active. 34
3.2 A comparison between a classic Top-2 MoE mechanism, and a MoE modified with the technologies introduced by DeepSeek-MoE [21]: a fine-grained MoE mechanism and a fine-grained mechanism with shared experts. Please note how, with respect to the classic mechanism, the variants have a different number of experts active, in order to keep the number of active parameters constant across the variants. 36
3.3 A comparison between a classic multi head attention mechanism, and a mixture of heads attention [45] mechanism. 37
3.4 An explaination of the mechanism of DEMix Layers [34]. Particularly relevant for our work is the training method, which involves selecting explicitly a topic for each different expert. 39

3.5 The training process used in Branch-Train-Merge [59]. 40

4.1 A comparison between a classic Multi-Head Attention module and the modified attention module we propose. The dotted line starting from the Top-k function is grey for non active slices. Note that the slices can contain an arbitrary number of heads, not necessarily two, like shown in this image. 45

5.1 A visualization of how a standard attention module can be modified into our architecture for the initialization of a Finetuning. Note that the core can contain an arbitrary number of heads, not necessarily two like shown in figure. 48

5.2 Losses of the models trained with full finetuning. 51

5.3 Losses of the models trained with a subset of the parameters. 51

1. Introduction

In recent years, deep learning has grown at an exponential rate in both scale and relevance, becoming a central technology across a wide range of fields: text generation [8, 33, 113], multimodal language models [62, 9], computer vision [82, 52], image [37, 84, 81, 87], image editing [19, 76], video generation [38, 6, 93, 63] being some of the most prominent examples. Such widespread adoption has been made possible by the convergence of three main factors: the large availability of public data spanning many different modalities, the development of high performance computing hardware of previously unseen power, and the growing acknowledgment, both from the general public and, especially, from the market, of the great opportunities that deep learning offers. Among the events that have marked this growth, the launch of ChatGPT in November 2022 can be considered a turning point, as it revealed, in a very short span of time, the power of deep learning applied to Large Language Models to a public for which Artificial Intelligence had until then been something distant and almost fictional. In the months and years that followed, a period of sustained growth has ensued, during which considerable amounts of money have been invested in training and, more broadly, in the development of increasingly intelligent and versatile models, in the pursuit of Artificial General Intelligence. This race towards ever larger and more capable models is not without a cost, however: as demonstrated by [48], obtaining a model with a significant increase in performance relative to another requires a correspondingly significant increase in the number of parameters, in the dimensions of the training dataset, and in the computing resources employed. Such an increase, in turn, translates into an increasingly large amount of electrical energy required to train and run these models, which results in exponentially increasing costs [96, 73]. These costs are made even more inefficient by the fact that deep learning activations tend to be sparse: a good portion of the electric energy used at training and inference time is spent computing operations involving values that are close to zero, and therefore contribute little or nothing to the final output of the network. In other words, such a portion of energy, and consequently the money spent to produce it, is effectively wasted. This phenomenon is not accidental, but a direct consequence of how these networks organize their internal representations: similarly to what happens in biological neural networks, the virtual neurons inside deep learning networks tend to specialize for specific

tasks or domains, producing strong activation values only when an input relevant to that particular task or domain is presented, and remaining largely inactive otherwise.

One of the solutions developed to cope with this inefficiency is the Mixture of Experts (MoE) paradigm, which makes it possible to increase the number of parameters of a model without a corresponding, significant increase in the energy used at inference time. The idea is not new: its introduction [42] predates the current generation of LLMs by several decades. However, the idea remained largely dormant for many years, attracting little attention from the research community, until the introduction of sparsely activated experts [91] showed that the approach could be made to scale. From that moment on, the paradigm has steadily grown in popularity [10, 69], especially in the development of LLMs, where it now underlies some of the most capable openly available models [44, 61], and, to a lesser extent, in vision and multimodal architectures. The core idea, shared by all the variants that have been experimented with so far, consists in dividing the parameters of a model into sections, called experts, and activating only a small subset of such experts through a gating function, ideally only the ones strictly necessary to complete the task requested by the user. This design yields a twofold advantage. Considering a dense model with the same total number of parameters as an MoE model, and assuming both possess the same amount of knowledge, the MoE model is clearly more efficient, since the dense model must activate the totality of its parameters to execute any given task, while the MoE model only activates a fraction of them. Symmetrically, considering a dense model whose number of parameters equals the number of parameters active in the MoE model at runtime, the MoE model is clearly more effective, since it has access to a much larger pool of knowledge, distributed across all its experts, than the dense model of comparable runtime cost. It is also worth noting the biological analogy that can be drawn between the human brain and an MoE model: as famously stated by the (often misunderstood) locution "We only use 10% of our brian", the different areas of the brain are not simultaneously active at all times, but each area activates only when required by the task at hand, in a way that closely mirrors the behaviour of experts inside an MoE layer. In more recent times, increasingly refined approaches have been proposed to further improve this design: among them, [21] introduced an idea similar to the one at the core of this work, namely a set of shared experts, always kept active, dedicated to storing general, task-agnostic knowledge, while the remaining experts are left free to specialize.

Numerous approaches and variants of MoE have been experimented with in the literature. The

most common choice is the implement the mechanism in the feed forward layer [55, 31, 120], but some works have experimented with implementing it in the attention heads instead, under the assumption that a certain degree of redundancy exists among the heads of a trained model, and that it is therefore possible to prune some of them without a significant loss in performance [104]. The first work to explore this concept is Mixture-of-Attention-Heads [117], which selects a subset of heads, sharing their key and value projections, and merges their results through a weighted summation. A more recent and relevant work in this direction is MoH [45], which refines the concept with an idea similar to the one proposed in this work: complete heads, *i.e.* without shared key and value projections as in the previous work (here every head retains independent projections), are used as experts, and their results are merged through weighted summation; in addition, a set of heads is kept as shared experts, meant to learn general knowledge common to all inputs. Beyond the question of where the MoE mechanism is placed, a wide range of experiments has focused on developing gating functions to select which experts to activate: the majority of these are sparse functions [5, 2], activating only a subset of experts, typically the top-k, *i.e.* the k experts classified as most relevant for the task at hand; a smaller subset of works [29, 27] has instead proposed dense gating functions, which activate all the experts and aggregate their results according to weights that reflect the relevance of each one. The granularity at which these gating decisions are taken also varies considerably across the literature: many gating functions operate at token-level [91], *i.e.* each token receives its own relevance scores for every expert, so that different tokens within the same sequence may activate different experts; other works have instead proposed gating functions that operate at sentence-level [53], assigning a single set of experts to an entire sentence rather than to each individual token. To further complicate the picture, a large variety of auxiliary loss terms has been developed with the purpose of balancing the importance and the computational load between the experts [91, 55], so as to prevent a small number of experts from being consistently over- or under-utilized. Finally, while in the vast majority of the works the gating functions are learnable, trained end-to-end via back-propagation or, in some cases, reinforcement learning, a minority of works have explored non-trainable gating functions [83], relying instead on fixed, hand-designed routing rules.

This work proposes a model employing an attention-level mixture of experts, selected by a sentence-level gating function. Two series of experiments have been carried out: one with a frozen,

pre-trained router used as the gating function, and one with a learnable one. The novel element of this architecture, especially with respect to MoH [45], the most closely related work in the literature, lies in the way the attentive heads are organized with respect to the MoE mechanism: the heads are divided into two sections, referred to as "core" and "slices". The core is a set of shared experts that is not influenced by the gating function and is always kept active, regardless of the gating function's output; the slices, on the other hand, constitute the "experts" of the architecture proper, even though each slice is a group of heads rather than a single head or network. With respect to DeepSeekMoE [21], which also implements a shared-experts mechanism, the main differences lie in the location at which such a mechanism is applied, namely the attention heads rather than the MLP layer, and in the granularity of the gating function, which here operates at sentence-level rather than at token-level. When this architecture is applied to a pretrained model, the slices are obtained by replicating a portion of the existing heads, so as to obtain a model that is functionally identical, in terms of activations and active parameters, to the original pretrained model (or nearly identical, since this characteristic varies with the number of selected slices). The purpose of this design is twofold: on one hand, to localize the knowledge and the logic mechanisms useful for all the topics and tasks handled by the model inside the core, by exposing it to all the data seen during training, since it is not influenced by the gating function; on the other hand, to let the slices, each of which is exposed only to a fraction of the training data thanks to the gating function, learn domain-specific knowledge and mechanisms instead. As an additional benefit, the resulting model is more interpretable than the original one, since its knowledge is explicitly divided into sections, rather than being diffused, in an opaque manner, throughout the entirety of its parameters.

Our model, obtained after an architectural modification and a finetuning on a Wikipedia dump [108], achieves performance comparable to our baseline, a well known Llama3.2-1B [33] model finetuned on the same data and with the same training hyperparameters, while at the same time showing various desirable characteristics. Across various experiments carried out with different variants of our architecture, we observe an increase in accuracy and, more generally, in overall performance, together with better performance on out-of-distribution data and tasks; furthermore, thanks to the larger amount of parameters made available by the MoE mechanism, we observe signs of a less severe catastrophic forgetting [50] occurring during training, and a significant improvement in performance on language modeling benchmarks such as RTE [20] and MNLI [109], both

provided by GLUE [105].

2. Preliminaries

2.1 Deep Learning

The core idea on which deep learning is based is to define functions able to simulate the behaviour of human neurons, under the assumption, which time has proved to be correct, that the combination of such functions would be able to produce machines able to replicate human intelligence.

2.1.1 The Perceptron and the linear layer

The first mathematical model of a neuron is the one proposed by McCulloch and Pitts [68], which however was not trainable, since it relied on hand-fixed weights and no learning rule was proposed. For this reason, it cannot be considered a machine learning algorithm in the modern sense, but rather a demonstration that networks of simple, binary threshold units could in principle compute arbitrary logical functions. The first work that was able to produce a virtual neuron able to be trained on data is the Perceptron [85], which was a probabilistic model that modeled the neuron as a linear combination of weights and inputs, together with a learning rule capable of adjusting those weights directly from labeled examples. Even if the Perceptron was designed to be a precise biological model, rather than an efficient engineering solution, the idea of the linear combination is the one on which the field of deep learning is based, and it is in this sense that the Perceptron can be regarded as the direct conceptual ancestor of every neural network discussed in this thesis. In modern neural networks, inspired by the Perceptron, a single neuron is implemented by a row matrix W of weights and a scalar bias b:

$$y = Wx + b = \begin{pmatrix} w_1 & w_2 & \dots & w_n \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + b \tag{2.1}$$

Where x represents the input of the neuron and y is the result. The terms W and b are parameters which are modified by the training process, typically by the gradient-based methods discussed in

the following section. The result y of the neuron is passed to a non-linear function. The original Perceptron relied on a hard threshold (step) function, which does not require differentiability, since no gradient-based training procedure was involved: the weights were instead updated according to a simple rule that nudged them in the direction that would have corrected a misclassified example. Once neural networks began to be trained end-to-end with backpropagation [86], differentiable non-linearities became necessary, since computing a gradient through a hard threshold function is not possible almost everywhere, and historically the hyperbolic tangent $tanh(x)$ or the sigmoid function $\sigma(x)$ became the most common choices, owing to their smoothness and to the fact that their output range is naturally bounded:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

In modern deep learning, the most common choice for the non-linear function is instead the $ReLU(x)$ function [32] (figure 2.1), whose adoption was largely motivated by its ability to mitigate the vanishing gradient problem that tends to affect saturating functions such as the sigmoid and the hyperbolic tangent, and by its considerably lower computational cost:

$$ReLU(x) = \begin{cases} x & \text{if} \quad x > 0 \\ 0 & \text{else} \end{cases} \tag{2.3}$$

Many variants of the $ReLU(x)$ function exist, generally designed to address some of its own shortcomings, such as the fact that a neuron whose input is always negative never updates its weights again, a phenomenon sometimes referred to as a "dead" neuron. Usually, for compactness, multiple neurons are embedded in the same linear layer, which in this case is expressed by a matrix W where every row represents the weights of a neuron and by a column matrix b where every element represents the bias of a neuron. In this case, the output of the layer is a column vector where every element represents the output of a single neuron. The non-linear function is usually applied on such vector elementwise, meaning that each output value is transformed independently of the others. A relevant example of a non-linear function that does not act elementwise, since each of its outputs depends on every element of its input, is the $Softmax(x)$ function [7]:

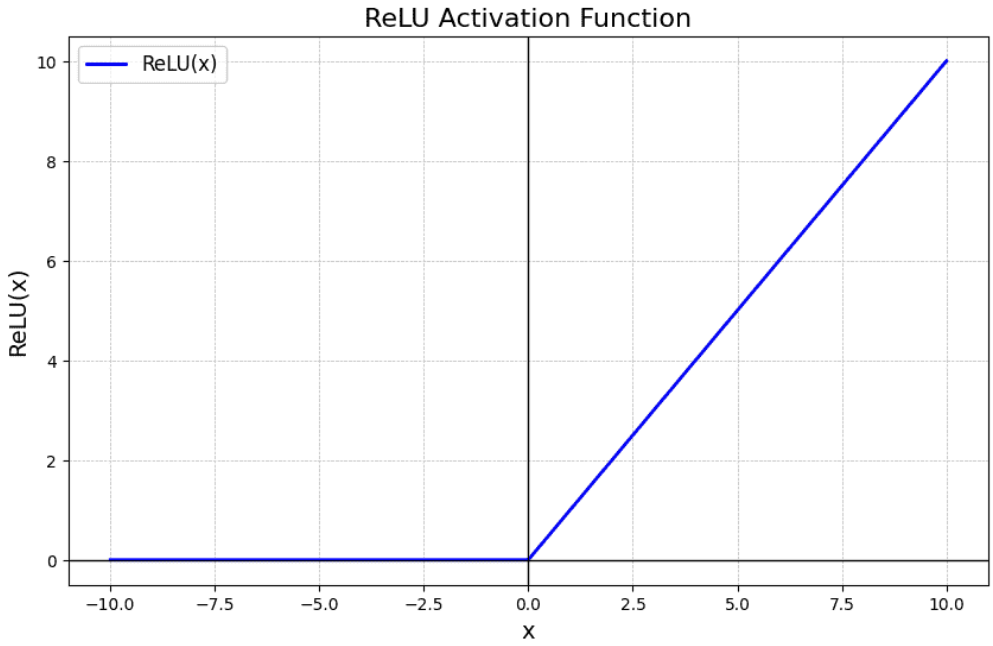


Figure 2.1: A plot of the ReLU function.

$$Softmax : \mathbb{R}^n \to \mathbb{R}^n, \quad Softmax(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{2.4}$$

This function is often used after a layer whose purpose is to output a discrete random distribution, for its property of producing a vector whose elements are all non-negative and sum to unity, which allows its output to be interpreted directly as a probability distribution over a discrete set of classes.

In almost all cases, linear layers (figure 2.2) are not employed singularly, but in sequences, in a feedforward manner: the outputs of one are passed as input to the following one, forming what is commonly referred to as a network's depth. It is important to note that every layer of a Multi Layer Perceptron (MLP) has its own non-linearity. In fact, it is trivial to verify that two or more linear layers without a non-linearity are equivalent to a single linear layer, since the composition of affine transformations is itself an affine transformation. Hence, the effectiveness of such a set of layers is upper-bounded by the effectiveness of the equivalent single layer, which is in general lower than the effectiveness of an MLP with non-linearities: this is formalized by the Universal Approximation Theorem [18, 40], which states that a feedforward network with at least one hidden layer and a non-linear activation function can approximate any continuous function on a compact domain arbitrarily well, given a sufficient number of neurons; a purely linear stack of layers, lacking

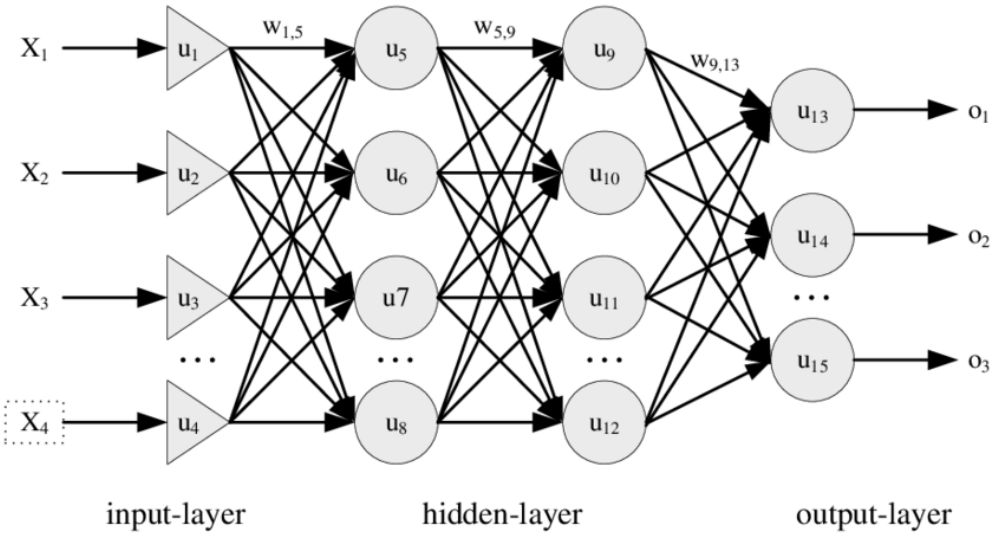


Figure 2.2: A schema of the information flow inside multiple linear layer.

non-linearities, does not enjoy this property, being limited to representing linear functions regardless of its depth. It is precisely this expressive gap, guaranteed by the presence of non-linear activations between layers, that justifies the design of the deep and complex architectures discussed throughout the remainder of this thesis.

It should be noted that various differences exist between the biological neurons and the model described in this section:

- A neuron receives inputs from other neurons thanks to cellular appendices called dendrites, and outputs a signal through an appendix called an axon. Each dendrite forms, together with the axon terminal of another neuron, a junction called a synapse: when an action potential travels down the axon and reaches its terminal, it triggers the release of chemical messengers, called neurotransmitters, into the small gap separating the two neurons (the synaptic cleft), which in turn alter the electric potential of the receiving neuron's membrane [47]. We will not delve further into the chemical details of this process here, since they would not be relevant for our discussion. What is relevant is the fact that the potential change induced by the dendrite inside the neuron varies according to the position of the dendrite with respect to the main body of the cell and with the frequency with which the corresponding axon activates. This implies that not all the dendrites induce the same potential variation inside the neuron and that such variation is also influenced by the nature of the input. This justifies the choice

to model the neuron as a linear combination of weights and inputs, where each weight can loosely be thought of as capturing the strength and character of a corresponding synaptic connection. However, biological neurons don't output signals that vary in intensity, like in the model expressed by the equation 2.1: once the membrane potential accumulated from its dendrites crosses a certain threshold, the neuron fires an action potential, a sharp, stereotyped spike whose shape and amplitude do not depend on how far above threshold the triggering input was [39]. This is known as the all-or-none principle: the neuron either fires a spike of fixed amplitude or does not fire at all, unlike the model of equation 2.1, whose output varies continuously with its input, and can in principle take on any real value.

- Another important difference is the fact that, as mentioned before, the dendrites of a neuron induce a potential difference which varies with respect to the frequency with which the axon to which they are connected activates. This means the output of a neuron would be more accurately represented by a time series of the potential difference, encoding information in the rate at which spikes occur, rather than the result of a single function evaluation, like the model expressed by 2.1, which produces a single, static value for each input it is given, with no explicit notion of time or of the history of previous inputs.

- The deep learning neural networks, as previously described, usually work in various layers, that can be expressed by a single, unified function. Biological neurons are organized in more complex structures: it is not uncommon to see neurons organized in layers, but recurrent circuits, in which the output of a neuron can, directly or indirectly, influence its own future inputs, are also common. Also, in biological tissues each neuron is independent of the others and activates asynchronously, following its own, locally-determined timing, while in deep learning all the neurons belonging to the same layer activate synchronously, as a consequence of the fact that the entire layer is typically evaluated as a single, unified matrix operation.

Regardless of these differences and of its extreme simpleness, this model has widely proved its effectiveness in a large set of fields and tasks, and it remains, to this day, the fundamental building block upon which the vast majority of modern deep learning architectures, including the ones discussed in the rest of this thesis, are constructed.

2.1.2 Gradient descent

The fundamental idea that allowed the training of MLPs, as opposed to single Perceptrons, is the backpropagation algorithm [86]. A single Perceptron can be trained directly, since the error committed on a given input can be immediately used to update its own weights. When multiple linear layers are stacked, however, this is no longer sufficient: the weights of the earlier (hidden) layers do not receive the target output directly, so the correction each of their parameters should receive cannot be computed as directly as for a single-layer model. Backpropagation solves this credit-assignment problem: by applying the chain rule of differentiation across the composition of layers, it makes it possible to propagate an error signal backward from the output layer to every earlier layer, and therefore to compute a gradient of the loss function with respect to every parameter of the network, regardless of its depth. This is the property that, in practice, makes it possible to train networks composed of an arbitrary number of stacked linear layers, and it is the foundation on which the rest of this section builds.

In order to train a neural network via gradient-based optimization, one must first define a "loss function", i.e. a function expressing how well the current state of the MLP is able to associate the inputs to the corresponding target outputs. Various loss functions exist, suited to various different tasks. One of the simplest and most important is the so-called Cross-Entropy loss, used for classification tasks where an input x_i has to be associated with a label $l \in \{1, 2, \ldots, C\}$:

$$CE(x, t) = -\sum_{i=1}^{N} \sum_{j=1}^{C} t_{ij} \log(f(x_i)_j) \tag{2.5}$$

Where x represents N inputs of the MLP, $f : \mathbb{R}^d \to \mathbb{R}^C$ is the function computed by the network, which for a given input x_i returns a vector $f(x_i)$ representing, for each label, the probability that the input corresponds to such label (typically obtained by applying a Softmax function, as introduced in Section 2.1.1, to the network's last layer); t represents the labels of the inputs as a series of vectors, each one associated with an input. Usually, the vector associated with an input x_i is encoded as a so-called "one-hot" vector: it is equal to 1 in the element t_{ij} with $j = l$, and 0 everywhere else. More generally, the cross-entropy loss evaluates the difference between two discrete probability distributions, namely the predicted distribution $f(x_i)$ and the target distribution t_i, and returns a lower value the closer the two distributions are to one another; this is also why it pairs so naturally

with a Softmax output layer, which is itself designed to produce a valid probability distribution.

Another common loss function, used for those cases where the MLP has to associate one or more continuous values to a given input x, is the mean squared error:

$$MSE(x, y) = \frac{1}{2N} \sum_{i=1}^{N} ||f(x_i) - y_i||^2 \tag{2.6}$$

Where y_i is the target vector associated with the input x_i, which the MLP, represented by the function $f : \mathbb{R}^n \to \mathbb{R}^m$, has to approximate. Some implementations omit the N term that justifies the denomination of this loss as a "mean" rather then a "sum", or fold it into the learning rate, since it only rescales the gradient uniformly across all parameters, but it is worth keeping explicit here for consistency with the name of the loss. The factor $\frac{1}{2}$, on the other hand, is a common convention adopted purely for convenience: it cancels out the exponent 2 once the loss is differentiated, without affecting the location of its minimum.

It is easy to note, from the definition of the loss functions and from the definition of the MLP given in Section 2.1.1, that the loss functions can be expressed as functions of the inputs, the target values and the parameters of the MLP, *i.e.* the weight matrices and the biases of its layers. Hence, in order to improve the performance of the MLP, one should minimize the loss function with respect to such parameters. In principle, this minimization could exploit second-order information, such as the Hessian matrix of the loss with respect to the parameters, to converge faster; in practice, however, computing, storing and inverting the Hessian is usually computationally intractable, since its size grows quadratically with the number of parameters of the network, and modern MLPs routinely have millions or billions of them. For this reason, a gradient-based, iterative optimization method called gradient descent (figure 2.3) is used instead, which only requires first-order information:

$$w_{i+1}^k = w_i^k - \alpha \frac{\partial L(w_i, \tilde{x}, \tilde{y})}{\partial w_i^k} \tag{2.7}$$

Where w_j^k is the value of the parameter corresponding to the index k at iteration j, w_i is the full vector of parameters at iteration i, L is the loss function, and $\tilde{x}$ is, in general, a subset of the available data x (usually called the "dataset") used for the current iteration i, while $\tilde{y}$ are the associated target values. The choice of $\tilde{x}$ is important for the effectiveness of the training process: with small amounts of data, x and $\tilde{x}$ may be identical (full-batch gradient descent), but in the vast

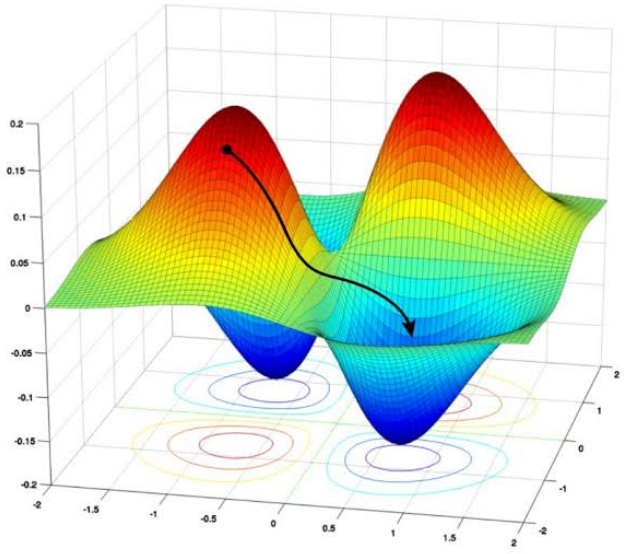


Figure 2.3: A simple example of gradient descent in two dimensions. The arrow represents the values taken by the parameters (two in this case) across the various iterations while minimizing the function.

majority of large-scale applications this approach is computationally intractable, hence a randomly sampled subset of x is used as $\tilde{x}$, resampled at every iteration i. When $\tilde{x}$ consists of a single randomly sampled example, this procedure is called stochastic gradient descent; instead, when $\tilde{x}$ is a small batch of examples, as is almost always the case in modern deep learning, the more precise term would be mini-batch gradient descent, although in practice the two names are used largely interchangeably in the literature.

The gradient descent process iterates until a "stop condition" is met, *i.e.* when the value of the loss function or of the parameters meets a certain condition. Some examples of stop conditions are, but are not limited to:

$$||\nabla L(w)|| \leq \tau, \quad \frac{||w_{i+1} - w_i||}{||w_i||} \leq \tau \tag{2.8}$$

the first one halting the training once the gradient becomes small enough to be considered negligible, the second one halting it once consecutive updates no longer change the parameters significantly. Also, in many large-scale applications, instead of a stop condition, a fixed number of iterations may be chosen. Usually such a fixed number is chosen so that training covers the dataset

an integer number of times, referred to as epochs.

Many variants of the gradient descent method have been proposed. The momentum method [75], not proposed specifically for deep learning but to accelerate gradient-based optimization methods in general, involves using an auxiliary vector to keep track of a velocity term, which accumulates a decaying history of past gradients:

$$v_{i+1} = \mu v_i + \alpha \nabla L(w_i); \quad w_{i+1} = w_i - v_{i+1} \tag{2.9}$$

Here $\mu \in [0, 1)$ is a hyperparameter controlling how much of the previous velocity is retained at each step. Intuitively, the velocity term acts like the momentum of a physical object rolling down the loss landscape: it allows the optimizer to keep moving in directions that have consistently reduced the loss over recent iterations, damping oscillations across steep, narrow valleys of the loss surface, and accelerating progress along shallow, consistent ones. Note that, to remain consistent with the minimization convention introduced above, the velocity must be subtracted from, rather than added to, the parameters at each step, since v_{i+1} is built by accumulating the positively-signed gradient.

A work that can be related to Polyak's is Nesterov's Accelerated Gradient [70], which improves on classic momentum's convergence rate, particularly on convex problems, by evaluating the gradient not at the current position w_i, but at the position the momentum term would already carry the parameters to. As shown by [99], its expression can be written as:

$$v_{i+1} = \mu v_i + \alpha \nabla L(w_i + \mu v_i); \quad w_{i+1} = w_i - v_{i+1} \tag{2.10}$$

The only difference with respect to classic momentum, in this formulation, lies in where the gradient is evaluated: rather than at w_i, it is evaluated at the "look-ahead" point $w_i + \mu v_i$, corresponding to the position the parameters would reach if no further gradient correction were applied. This look-ahead allows the algorithm to partially correct its own trajectory before actually taking the step, which is what yields its improved convergence guarantees. Nesterov's algorithm is the one employed by [3] in its standard stochastic gradient descent implementation.

Among modern algorithms designed specifically for deep learning, a relevant and influential publication is AdaGrad [28], which introduced the idea of an adaptive learning rate, on which various later algorithms are based. It involves scaling every parameter's update by the square root of the cumulative sum of its squared gradients:

$$w_{i+1} = w_i - \frac{\alpha}{\sqrt{G_i + \epsilon}} \nabla L(w_i) \tag{2.11}$$

Where α is a fixed learning rate, G_i is the sum, accumulated across all previous iterations, of the squared gradients of the current parameter, and ϵ is a small value added to prevent division by zero. This means that parameters which have historically received large gradients see their effective learning rate shrink over time, while parameters that have received small or infrequent gradients keep a comparatively larger effective learning rate. This behaviour was originally motivated by online learning settings with sparse features, where rarely-observed but informative parameters should be allowed to move more, proportionally, than frequently-updated ones, but it generalizes naturally to the dense parameter updates typical of deep learning.

A work worth noting, even if it has never been formally published, is RMSProp [100]. It derives from AdaGrad, and uses a nearly identical update rule, except for how the term G_i is accumulated: rather than a cumulative sum over all past iterations, as in AdaGrad, it uses a running exponential average of the squared gradients:

$$G_{i+1} = \gamma G_i + (1 - \gamma)(\nabla L(w_i))^2 \tag{2.12}$$

Since older gradients are discounted geometrically rather than accumulated indefinitely, this modification prevents the effective learning rate from shrinking excessively over the course of a long training run, which is a known failure mode of AdaGrad: once G_i has grown large enough, training can effectively stall, since every subsequent update becomes vanishingly small regardless of the current gradient's magnitude.

A widely used algorithm, derived from AdaGrad and RMSProp, is Adam [49], also implemented by PyTorch [3]:

$$g_i = \nabla L(w_i) + \lambda w_{i-1} \tag{2.13}$$

$$m_i = \beta_1 m_{i-1} + (1 - \beta_1) g_i \tag{2.14}$$

$$v_i = \beta_2 v_{i-1} + (1 - \beta_2) g_i^2 \tag{2.15}$$

$$\hat{m}_i = \frac{m_i}{1 - \beta_1^i} \tag{2.16}$$

$$\hat{v}_i = \frac{v_i}{1 - \beta_2^i} \tag{2.17}$$

$$w_{i+1} = w_i - \alpha \frac{\hat{m}_i}{\sqrt{\hat{v}_i} + \epsilon} \tag{2.18}$$

Adam implements both adaptive learning, via the v_i term, and momentum, via the m_i term; m_i and v_i are commonly referred to as the first and second moment estimates of the gradient, respectively, while $\hat{m}_i$ and $\hat{v}_i$ are their bias-corrected versions, needed because m_i and v_i are initialized at zero and are therefore biased towards zero during the first iterations of training. β_1, β_2, α, λ and ϵ are fixed hyperparameters: β_1 and β_2 modulate the decay rate of the momentum term and of the adaptive learning term, respectively; α is the learning rate; λ modulates weight decay, here applied as a form of L2 regularization added directly to the gradient; and ϵ prevents division by zero.

Finally, in transformers the AdamW algorithm [65], derived from Adam, represents a de facto standard for training. Its main difference with respect to Adam lies in how weight decay is applied: rather than being folded into the gradient term g_i, as in the λw_{i-1} term of the equations above, it is applied directly to the parameter update itself, decoupling it from the adaptive, gradient-based part of the step:

$$g_i = \nabla L(w_i) \tag{2.19}$$

$$w_{i+1} = w_i - \alpha \left(\frac{\hat{m}_i}{\sqrt{\hat{v}_i} + \epsilon} + \lambda w_i \right) \tag{2.20}$$

Where m_i, v_i, $\hat{m}_i$ and $\hat{v}_i$ are computed exactly as in Adam, but now starting from a gradient g_i that no longer includes the weight decay term. Note how the weight decay, previously mixed into the gradient before being accumulated into the momentum and adaptive-learning terms, is here added as an independent term at the very end of the update, applied at the same time as, but computed

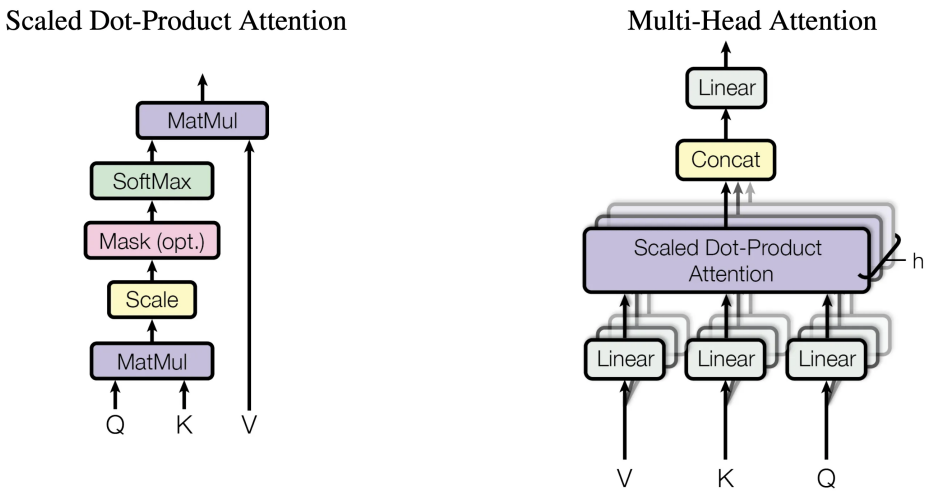


Figure 2.4: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several Attention layers running in parallel [103].

independently of, the adaptive gradient-based step. This distinction matters because folding weight decay into the gradient, as Adam does, causes it to be scaled, and distorted, by the same adaptive per-parameter learning rate as the gradient itself; AdamW's decoupled formulation instead applies a uniform decay regardless of a parameter's gradient history, which the original authors show leads to better generalization in practice.

2.2 Transformers

The vast majority of deep learning architectures used to process and generate sequences are based on the transformer architecture [103], which represents the *de facto* standard for a wide range of tasks, including text generation [8, 33], text encoding [24], image processing [26, 11], multimodal architectures [106], and machine translation [103], the task for which it was originally proposed. Unlike the recurrent architectures that preceded it, the transformer processes all elements of a sequence in parallel rather than one at a time, relying entirely on an attention mechanism, rather than on recurrence, to model the relationships between elements of a sequence, regardless of how far apart they are positioned.

The transformer architecture is built around the Scaled Dot-Product Attention mechanism (depicted in Figure 2.4), a general-purpose building block used to aggregate information across a

sequence of elements. The module receives three sequences, called *query*, *key*, and *value* (where key and value must have the same length, since each key is paired with a corresponding value), represented by matrices $Q \in \mathbb{R}^{l_q \times d}$, $K \in \mathbb{R}^{l_k \times d}$, and $V \in \mathbb{R}^{l_k \times d}$ respectively, and computes the result as follows:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.21}$$

When Q, K and V are all computed from the same input sequence, this mechanism is more specifically referred to as Self-Attention, since every element of the sequence attends to every other element of the same sequence, including itself; when, instead, the query comes from a different sequence than the key and value, as will be discussed later for the decoder block, the mechanism is referred to as Cross-Attention. The first part of the formula creates a matrix of shape $l_q \times l_k$, which represents the (scaled) similarities between the tokens in Q and K. The *softmax* function normalizes the rows of this similarity matrix, so that each row sums to unity, and the subsequent multiplication with the V matrix results in a sequence of the same length as the *query*, where each output token is an aggregation of information from the elements of the *value* sequence, weighted by the corresponding row of the similarity matrix.

The purpose of the $\frac{1}{\sqrt{d_k}}$ scaling factor is to reduce the magnitude of the similarities before the softmax is applied. Without this scaling, for large values of d_k the dot products QK^T tend to grow large in magnitude, since they sum over d_k terms; this pushes the softmax function into regions where its gradient is extremely small, close to a one-hot output, which would slow down or even stall the learning process. Dividing by $\sqrt{d_k}$ counteracts this effect, keeping the similarities, and consequently the softmax output, in a numerically well-behaved range.

In transformers, each layer contains a structure named the Multi-Head Attention layer (Figure 2.4), composed of several Scaled Dot-Product Attention modules, referred to as heads, running in parallel. The mechanism requires three learned linear transformations for each head i: $W_i^Q \in \mathbb{R}^{d \times d_k}$, associated with the query, $W_i^K \in \mathbb{R}^{d \times d_k}$, associated with the key, and $W_i^V \in \mathbb{R}^{d \times d_v}$, associated with the value, where d is the dimensionality of the model.

The layer takes as input the sequences $Q \in \mathbb{R}^{l_q \times d}$, $K \in \mathbb{R}^{l_k \times d}$, $V \in \mathbb{R}^{l_k \times d}$, and computes the result as follows:

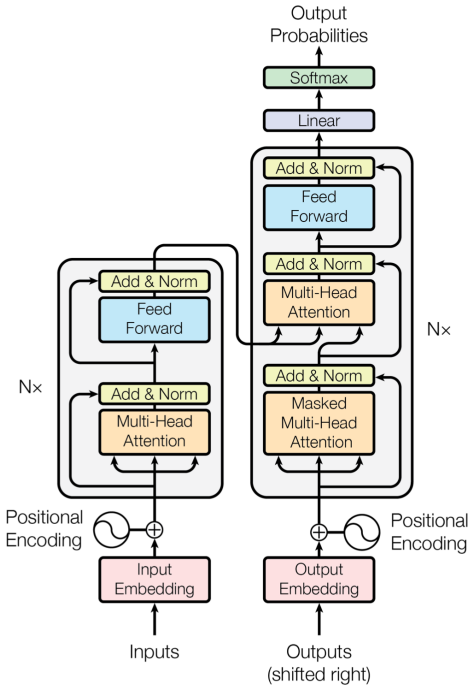


Figure 2.5: The complete encoder-decoder transformer architecture [103].

$$MultiHead(Q, K, V) = concat(head_1, \ldots, head_h)W^O \tag{2.22}$$

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V) \tag{2.23}$$

Where h is the number of heads, and the transformation $W^O \in \mathbb{R}^{hd_v \times d}$ projects the concatenation of the results of all the heads back to the dimensionality d of the model. Splitting the attention mechanism into several heads, each operating on a lower-dimensional projection of the input, allows the model to jointly attend to information from different representation subspaces at different positions, rather than being restricted to a single, shared notion of similarity between tokens.

The complete transformer architecture is made of two blocks: an encoder and a decoder (Figure 2.5). The former receives a sequence of tokens representing the input, while the latter generates a new sequence representing the output. The original work [103] designed it for machine translation, but later proposals demonstrated how it can be trained for a wide range of other tasks, such as object detection [11], image captioning [106], and more general-purpose sequence-to-sequence pretraining objectives [78, 57], applicable to many downstream tasks beyond translation itself.

The architecture takes as input, for the encoder block, a sequence representing the sentence to translate, divided into tokens. Each token is one-hot-encoded and then projected into its embedding representation in $\mathbb{R}^d$ through a learned embedding layer, equivalent to a lookup table indexed by the token's identity. The embeddings are then summed with a positional encoding, agnostic with respect to the content of the sequence, marking every token with its position relative to the others; without this addition, the attention mechanism described above would have no notion of token order at all, since it treats its input as an unordered set of elements. In the original work [103], the positional encodings are formulated as follows:

$$PE_{(pos,2i)} = sin\left(\frac{pos}{10000^{2i/d}}\right) \tag{2.24}$$

$$PE_{(pos,2i+1)} = cos\left(\frac{pos}{10000^{2i/d}}\right) \tag{2.25}$$

Where pos is the position of the token in the sequence, and $2i$, $2i + 1$ index the even and odd dimensions, respectively, of the resulting embedding vector. The tokens given as input to the decoder block are processed by an identical pipeline, possibly with a different embedding layer.

The tokens in the encoder are passed to a Multi-Head Attention layer, normalized [4] after a residual connection [35] is added, followed by a series of feed-forward layers, applied independently to every token, and a second residual connection and normalization. The residual connections allow gradients to flow more directly through the many stacked layers of the network during backpropagation, which is particularly important in deep transformer stacks, while the normalization stabilizes the distribution of the activations passed to the following layer.

The tokens in the decoder, instead, are first passed to a masked Multi-Head Attention layer, where the Self-Attention is defined as follows:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V \tag{2.26}$$

Where M is a masking matrix defined as:

$$M_{ij} = \begin{cases} -\infty & \text{if} \quad j > i \\ 0 & \text{else} \end{cases} \tag{2.27}$$

The masking matrix sets to $-\infty$ every element of the similarity matrix that represents the similarity of a token with a subsequent element of the sequence, resulting in null weights once the softmax is applied, and therefore in no contribution from those elements while aggregating the *values*. This ensures that, in the decoder, information is integrated among tokens in only one direction, from earlier to later positions, unlike in the encoder, where every token can attend to every other token regardless of position. This restriction is what allows the decoder to be trained to generate text autoregressively: at training time, every position can be processed in parallel while still being prevented from "looking ahead" at the tokens it is meant to predict, and at inference time the same restriction simply reflects the fact that later tokens genuinely have not been generated yet.

The second Multi-Head Attention layer of the decoder employs the Cross-Attention case introduced above, in which the *keys* and *values* are computed from the output of the encoder stack, while the *query* is computed from the decoder's own sequence. This allows the tokens of the decoder to integrate information from the encoder, even though the two sequences may have different lengths: in general, an attentive mechanism outputs a sequence of l_q elements, where l_q is the number of elements passed as the *query*, regardless of the length l_k of the *key* and *value* sequences.

After the cross-attention layer, a feed-forward layer is applied to every token. Similarly to the encoder, every sub-layer of the decoder (masked Self-Attention, Cross-Attention, and feed-forward) is followed by a residual connection and a normalization.

At the top of the decoder, a final linear layer is applied independently to the output embedding of every token in the sequence, projecting each one to a vector with the same dimension as the dictionary, *i.e.* the number of possible tokens. Each such vector, after passing through a softmax, represents a probability distribution over the vocabulary, predicting the token that should follow the corresponding position in the sequence; during training, this operation is applied to every position at once, while at inference time, as will be discussed below, only the distribution corresponding to the last, most recently generated, token is actually needed to sample the next one.

The training of this architecture involves a single forward pass for each batch of pairs (s_i, s_o), where s_i is the input sequence and s_o is the desired output sequence to be generated at inference time. The output of the decoder is a sequence $y \in \mathbb{R}^{l_o \times t}$ of vectors, with the same length as s_o, where l_o is the length of s_o, t is the number of tokens in the decoder's dictionary, and y_i represents the predicted probability distribution over the vocabulary for the $(i+1)$-th token of the sequence.

Training follows a self-supervised approach: the parameters of the model are optimized to maximize the probability assigned by y_i to the actual $(i+1)$-th token of s_o, for every position i simultaneously, via a Cross-Entropy loss:

$$L(y, s_o) = \sum_{i=0}^{l_o-1} CE(y_i, s_o^{(i+1)}) \tag{2.28}$$

During training, the decoder is always given the ground-truth tokens of s_o as input, regardless of what it would have predicted itself at the previous step, which allows every position of the sequence to be trained in parallel, in a single forward pass, rather than requiring one forward pass per generated token, as is instead the case at inference time.

At inference time, the pipeline is more complex, since the output sequence is not known in advance and must instead be generated one token at a time: each new token requires a separate forward pass through the decoder, in which the sequence generated by the encoder is used for the Cross-Attention, and the tokens generated so far are given as input to the decoder; this forward pass outputs a single distribution $y \in \mathbb{R}^t$, corresponding to the last position of the sequence, which is used to sample the next token. This process is then iterated, appending the newly sampled token to the input of the next forward pass, until a special token, denoted <EOS>, is sampled, signalling that the generation process is complete.

2.3 Positional encoding

It has been shown [115] that Self-Attention modules are position-agnostic: without any additional information, permuting the tokens of the input sequence would simply permute the output in the same way, with no other effect on the result. Hence, the use of positional embeddings to explicitly mark every token with its position is essential for the effectiveness of the transformer architecture. The positional encoding proposed by [103], introduced earlier alongside the description of the original transformer architecture, belongs to the class of position embeddings called *absolute position embeddings*. Other examples of this class [77, 24, 16] rely on learnable vectors added to the token embeddings, rather than on a fixed, hand-designed formula, as follows:

$$f_t(x_i, i) = W_t(x_i + p_i), \quad t \in \{q, k, v\} \tag{2.29}$$

Where p_i is a learnable vector associated with position i, added to the token embedding x_i before the query, key, and value projections W_t are applied.

Other authors proposed approaches based on *relative position embeddings*, where each token in the key and value, during the computation of Self-Attention, receives an encoding (e.g. a learnable vector) depending on its relative distance to the other tokens, rather than on its absolute position in the sequence. The approach used by [88] is one of the simplest examples of this mechanism:

$$\begin{aligned} f_q(x_m) &= W^q x_m \\ f_k(x_n, n) &= W^k(x_n + p_r^k) \\ f_v(x_n, n) &= W^v(x_n + p_r^v) \end{aligned} \tag{2.30}$$

Where p_r^k and p_r^v are learnable vectors depending on $r = clip(m - n, r_{min}, r_{max})$. The clipping operation bounds the range of relative distances the model needs to learn a distinct encoding for, which both limits the number of learnable vectors required and helps the model generalize to sequences longer than the ones seen during training, since distances beyond r_{max} are all mapped to the same encoding. Other works [22, 78, 36] experimented with more complex formulas based on relative position embeddings between the tokens.

Another relevant work in position embeddings is the Rotary Position Embedding (RoPE [97]), used by Llama3 [33], on which the models produced by our work are based. RoPE embeds position using a rotation matrix, applied to the query and key projections at every computation of Self-Attention:

$$f_{\{q,k\}}(x_n, n) = R^d_{\Theta,n} W^{\{q,k\}} x_n \tag{2.31}$$

Where:

$$R^d_{\Theta,n} = \begin{pmatrix} cos(n\theta_1) & -sin(n\theta_1) & 0 & 0 & \cdots & 0 & 0 \\ sin(n\theta_1) & cos(n\theta_1) & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & cos(n\theta_2) & -sin(n\theta_2) & \cdots & 0 & 0 \\ 0 & 0 & sin(n\theta_2) & cos(n\theta_2) & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & cos(n\theta_{d/2}) & -sin(n\theta_{d/2}) \\ 0 & 0 & 0 & 0 & \cdots & sin(n\theta_{d/2}) & cos(n\theta_{d/2}) \end{pmatrix} \tag{2.32}$$

Where d is the dimension of the model, assumed to be even, and $\Theta = \{\theta_i = 10000^{-2(i-1)/d}, i = 1, 2, ..., d/2\}$. The rotation matrix can be described informally as a series of $d/2$ two-dimensional rotation matrices, each applied separately to a distinct pair of consecutive dimensions of the vector. It is important to note the properties of this method: the flexibility with respect to sequence length, since the rotation can be computed for any position n without requiring a fixed maximum length to be chosen in advance, as would instead be needed for a table of learned absolute embeddings; a decaying dependency between tokens as their relative distance increases, since the similarity between distant tokens reduces progressively as the difference between their rotation angles grows; and, most notably, the fact that it captures *relative*, rather than purely absolute, positional information, despite each token individually being rotated according to its own absolute position. This last property becomes apparent when computing the value in the attention matrix corresponding to two tokens in positions n and m:

$$a_{m,n} = q_m^T k_n = (R^d_{\Theta,m} W^q x_m)^T (R^d_{\Theta,n} W^k x_n) = x_m^T (W^q)^T R^d_{\Theta,n-m} W^k x_n \tag{2.33}$$

Where the identity ${R^d_{\Theta,m}}^T R^d_{\Theta,n} = R^d_{\Theta,n-m}$, which holds since rotation matrices are orthogonal and rotations compose additively in their angles, shows that the absolute positions m and n cancel out of the final expression, leaving a term that depends only on their relative distance $n - m$. This is precisely what allows RoPE to combine the simplicity of an absolute, per-token formulation with the generalization benefits typically associated with relative position embeddings.

2.4 Decoder-only architectures

The transformer architecture is often used in an encoder-only or decoder-only configuration, rather than in its complete encoder-decoder form. Encoder-only architectures [24] are used to produce contextual representations of the input data (commonly text or images), typically one for every element, i.e. token or patch, of the input sequence; a classification or task-specific head is then usually applied on top of these representations. The exact loss used to train such architectures, as well as the downstream head applied to their output, depends on the final task: examples include image classification [26, 64] and named entity recognition [111], alongside the text classification and question-answering tasks BERT itself was originally evaluated on. Decoder-only architectures, instead, are used in settings where the sequence of already-generated tokens is sufficient to generate the following one, without needing a separate input sequence encoded beforehand. The main example is text generation [8, 33, 113], where a model generates each new token conditioned on the ones provided by the user as a prompt, together with the ones it has generated so far.

The main difference between a decoder-only architecture and the decoder of an encoder-decoder architecture is the absence of the Cross-Attention mechanism, which follows directly from the absence of an encoder: with no separate encoded sequence to attend to, masked Self-Attention alone is enough to model the dependencies between the tokens of the sequence.

At inference time, decoders usually employ a cache of key and value vectors, generally referred to as the KV-cache, to avoid unnecessary recomputation. The initial processing of the prompt is usually referred to as the "pre-fill" phase: the tokens given as input by the user are all passed to the model in a single forward pass, and their keys and values, for every layer, are saved separately; this same forward pass also produces the first generated token. During the generation of every subsequent token, these saved values never change: the masked Attention mechanism ensures that new tokens do not influence the embeddings of older ones in any way, since a token can only attend to itself and to the tokens preceding it. This implies that the corresponding activations do not need to be recomputed at every step, and can instead be reused, and simply extended with the keys and values of the newly generated tokens, at every subsequent step of generation, commonly referred to as the "decode" phase. Without this mechanism, generating each new token would require recomputing Self-Attention over the entire sequence generated so far, from scratch, which becomes

increasingly expensive as the sequence grows; the KV-cache instead allows each new token to be generated without repeating this earlier work.

The KV-caching mechanism exists in many variants, such as PagedAttention [54], which, by storing the KV-cache in blocks rather than in contiguous memory, addressed several memory efficiency problems caused by the varying lengths of different requests, their unknown final length at the time memory must be allocated for them, and the redundancy arising from requests that share a common prefix of tokens.

Note that KV-caching mechanisms can also be employed by encoder-decoder architectures: the encoder's keys and values, being computed only once from a sequence that does not change over the course of generation, can be cached and reused across every decoding step for the Cross-Attention computation, while the decoder's own keys and values, growing with every newly generated token, can be cached in exactly the same way as in a decoder-only architecture.

2.5 Llama 3

The models presented in this work have been obtained by modifying the architecture of a pretrained Llama model [33].

SwiGLU: introduced by [90], and based on GLU [23], it has the formula:

$$swish_\beta(x) = x \cdot sigmoid(\beta x) = \frac{x}{1 + e^{-\beta x}} \tag{2.34}$$

$$SwiGLU(x, W, V, b, c, \beta) = swish_\beta(xW + b) \odot (xV + c) \tag{2.35}$$

Where, in principle, β, W, V, b, c are all learnable parameters. In practice, however, Llama, like most modern LLMs following PaLM's example, uses this activation in a simplified form: the bias terms b and c are omitted entirely, and β is fixed to 1 rather than learned, which reduces the $swish_\beta$ term to the standard SiLU activation function.

PreNorm: all Llama models employ a variant of the transformer architecture where the normalization is applied to the input of each sub-layer, rather than to its output, an arrangement generally referred to as Pre-Normalization. This choice improves training stability, since it allows gradients to propagate through the residual connections without being distorted by the normalization at every

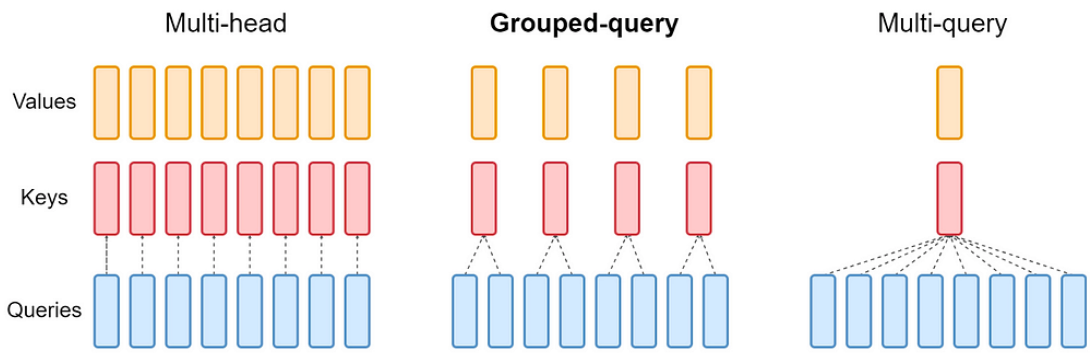


Figure 2.6: Overview of Multi-Head Attention, Grouped-Query Attention, Multi-Query Attention [1].

step, which is particularly beneficial in the very deep stacks typical of modern LLMs. It uses the RMSNorm [116].

Grouped Query Attention: introduced in [1], inspired by Multi-Query Attention [89], where all the query sequences in the Multi-Head layer share the same key and value sequence. It involves dividing the heads into G groups (Figure 2.6, center) and assigning every query of a group to the same key and value sequence. With respect to standard Multi-Head Attention, this is significantly more efficient in terms of memory: since, as discussed in the previous section, the key and value vectors of every token are stored in the KV-cache, reducing the number of distinct key and value sequences from one per head to one per group directly shrinks the size of the cache, and consequently the memory bandwidth required to read it at every decoding step, which is typically the main bottleneck during autoregressive generation. At the same time, Grouped Query Attention retains better performance than Multi-Query Attention, since it does not collapse all heads down to a single shared key and value sequence.

This model employs RoPE for positional embeddings, with the base Θ of the frequency terms, as defined in the previous section, increased from the original 10 000 to 500 000. Counter-intuitively, increasing this base value actually lowers the rotation frequency across the embedding dimensions, resulting in longer wavelengths; this allows the model to better distinguish between token positions over much longer contexts than the ones RoPE was originally designed for, at the cost of some resolution over shorter ranges.

2.6 Mixture of Experts

The Sparsely-Gated Mixture-of-Experts approach [91] revitalized the older Mixture of Experts paradigm, introduced by [42], by showing that it could be made to scale to modern deep networks, improving the performance of language modeling and machine translation models by increasing the number of parameters while keeping computational costs low. This is achieved by activating only a small fraction of the parameters for every input, thanks to a gating function.

A "Sparsely-Gated Mixture-of-Experts Layer" (Figure 2.7) employs a number n of simple feed-forward networks as experts, with a learnable network, parametrized by a matrix W_g, used as a gating function to select which experts to activate, and a second learnable matrix, W_{noise}, used to inject noise into the gating decision and add variability to the choice of experts:

$$KeepTopK(v,k) = \begin{cases} v_i & \text{if } v_i \text{ is in the top } k \text{ elements of } v \\ -\infty & \text{else} \end{cases} \tag{2.36}$$

$$H(x)_i = (xW_g)_i + StandardNormal() \cdot SoftPlus((xW_{noise})_i) \tag{2.37}$$

$$G(x) = softmax(KeepTopK(H(x),k)) \tag{2.38}$$

$$y = \sum_{i=1}^{n} G_i(x)E_i(x) \tag{2.39}$$

Where k is the number of experts kept active for every input, and $E_i(x)$ is the output of the i-th expert.

The function $KeepTopK$ ensures that, in $G(x)$, only the elements corresponding to the top-k components of the noisy gating scores $H(x)$, rather than of the raw output of W_g alone, are not null. The others, set to $-\infty$, correspond to null elements after the softmax function. This implies that the output y of the layer is a weighted average of the results of the top-k most relevant experts, weighted by the corresponding scores of $G(x)$. It should be noted that, for the experts actually selected on a given input, every part of the layer, including the experts themselves and the gating matrices W_g and W_{noise}, can be reached by the gradient during backpropagation as usual.

It has been observed that during training the gradient tends to favor a few weights of the gating network, resulting in the corresponding experts receiving more inputs than the others. This behaviour is self-reinforcing, since such experts are trained more rapidly on a larger share of the

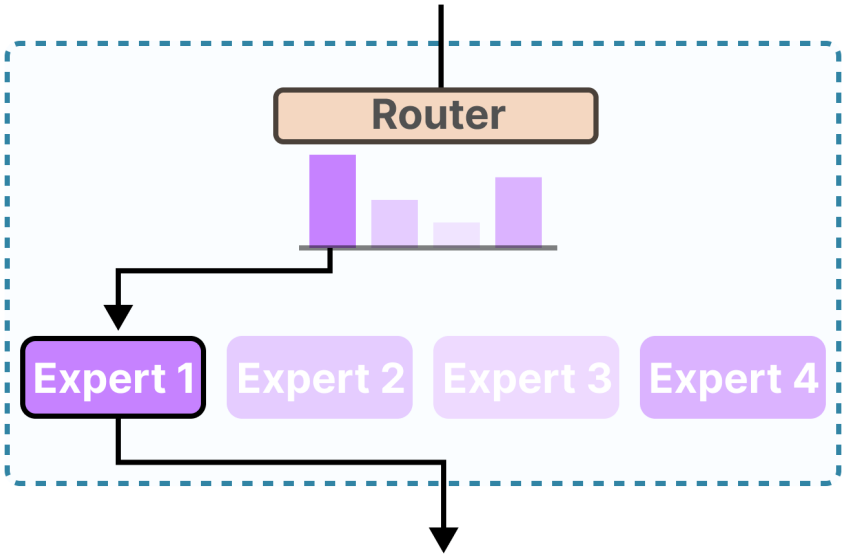


Figure 2.7: Sparsely Gated Mixture of Experts Layer.

data, and are consequently favored even further by the gating network in subsequent iterations. In order to counteract this problem, an auxiliary term is added to the loss:

$$Importance(X) = \sum_{x \in X} G(x) \tag{2.40}$$

$$L_{importance} = w_{importance} \cdot CV(Importance(X))^2 \tag{2.41}$$

Where $w_{importance}$ is a hyperparameter, and CV denotes the coefficient of variation, i.e. the ratio between the standard deviation and the mean of its argument. This auxiliary term encourages a balance between the sums of the gating values received by each expert, computed over a batch X of inputs. Another desirable characteristic would be a balance between the number of inputs actually routed to each expert, rather than just the sum of their gating weights, in order to avoid memory and performance issues on distributed hardware. The previous auxiliary term, in fact, may still produce experts that receive few inputs with disproportionately high weights, alongside experts that receive many inputs with low weights, since what is balanced is only the sum of such weights, not how many inputs contribute to that sum. To address this issue, a second auxiliary term is added:

$$Load(X)_i = \sum_{x \in X} P(x, i) \tag{2.42}$$

$$L_{load}(X) = w_{load} \cdot CV(Load(X))^2 \tag{2.43}$$

Where w_{load} is a hyperparameter, and $P(x, i)$ is defined as the probability for the i-th expert to receive a non-null weight from input x. Since the actual routing decision is discrete, and therefore not differentiable, $P(x, i)$ is instead defined as the probability that expert i would still be selected if the noise term were resampled, which provides a smooth, differentiable proxy for how close a given expert is to being selected, rather than a hard yes/no indicator. The probability is written as follows:

$$P(x, i) = Pr(H(x)_i > kth_{excluding}(H(x), k, i)) \tag{2.44}$$

Where $kth_{excluding}(H(x), k, i)$ is the k-th highest component of $H(x)$, excluding the i-th one. Simplifying:

$$P(x, i) = \Phi\left(\frac{(xW_g)_i - kth_{excluding}(H(x), k, i)}{SoftPlus((xW_{noise})_i)}\right) \tag{2.45}$$

Where Φ is the CDF of the standard normal distribution. Note that, unlike the hard top-k selection itself, this definition of $P(x, i)$ is differentiable with respect to both W_g and W_{noise}, which is what allows the load-balancing loss to provide a useful gradient signal, pushing the gating network towards routing a more even number of inputs to every expert, even for experts that were not among the top-k for a specific input.

2.7 Mixture of Attention

In the transformer architecture [103] it is common to implement MoE in the feed-forward layers [55, 31, 120]. Mixture of Attention Heads [117] experimented with an implementation of mixture of experts in the Multi-Head Attention layer (Figure 2.8). It is based, like all MoE mechanisms, on a gating function G and a set of N experts $\{E_1, ..., E_N\}$, of which only k are activated for every token. It receives three sequences Q, K, V, exactly as in standard Multi-Head Attention, where the t-th element of Q is denoted by q_t. The gating function G is implemented by a linear layer $W_g \in \mathbb{R}^{d \times N}$:

$$p_t = Softmax(q_t W_g), \qquad G_t = TopK(p_t, k) \tag{2.46}$$

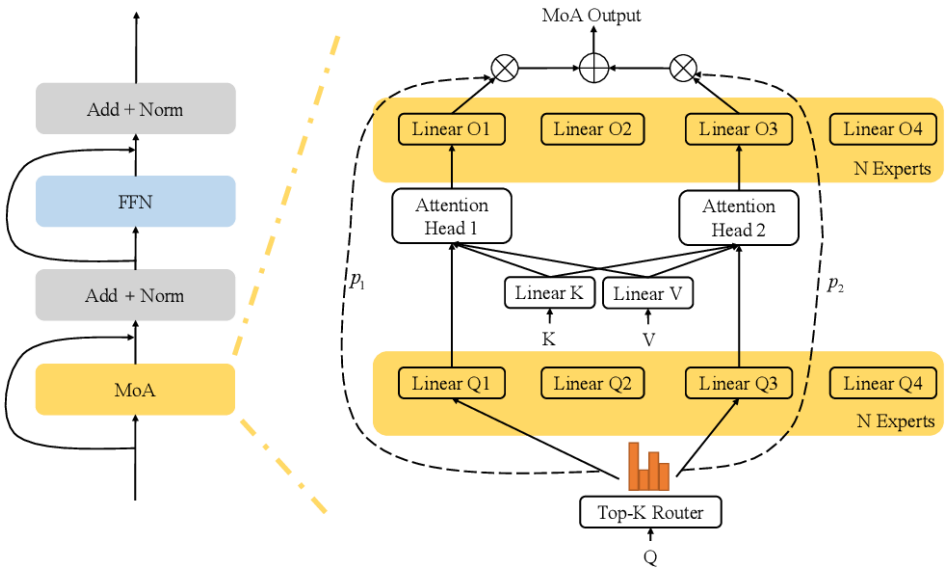


Figure 2.8: Mixture of Attention layer [117]. Note that the key and value projections are shared between the heads.

Where G_t denotes the set of indices of the k experts selected for token t; note that, unlike a single gating decision computed once for the whole sequence, G_t is computed independently for every token, so different tokens of the same sequence may activate different experts. The result of a single Attention expert is given by:

$$W_{i,t}^{att} = softmax\left(\frac{q_t W_i^Q (KW^K)^T}{\sqrt{d_k}}\right) \tag{2.47}$$

$$o_{i,t} = W_{i,t}^{att} V W^V \tag{2.48}$$

$$E_i(q_t, K, V) = o_{i,t} W_i^O \tag{2.49}$$

Note that in this architecture, similarly to Multi-Query Attention [89], the key and value projections W^K, W^V are shared between heads, while the query projection W_i^Q and the output projection W_i^O remain specific to every expert i.

The result of the layer is computed as follows:

$$y_t = \sum_{i \in G_t} w_{i,t} E_i(q_t, K, V) \tag{2.50}$$

With:

$$w_{i,t} = \frac{p_{i,t}}{Detach\left(\sum_{j \in G_t} p_{j,t}\right)} \tag{2.51}$$

Where $p_{i,t}$ denotes the i-th component of p_t, and $Detach$ is a function that prevents the gradient from reaching the denominator, i.e. it treats the sum at the denominator as a constant during backpropagation, even though it is computed from the same p_t used in the numerator. Intuitively, this decouples the gradient received by the routing network from the dynamics of the renormalization itself: without detaching, the gradient with respect to $p_{i,t}$ would be entangled with how the whole denominator changes whenever any of the selected experts' gate values change, whereas detaching it isolates a cleaner, more directly interpretable signal on $p_{i,t}$ alone. This trick has been found to be empirically useful for learning the routing probabilities.

3. Related work

The idea of integrating Mixture-of-Expert with experts that are kept always active has already been introduced, and is not, in itself, a novelty of this thesis. For example, in the Pyramidal Residual MoE (figure 3.1) introduced by [79], FFN layers always activate a fixed MLP and a second MLP selected by a gating function. This design is adopted under the hypothesis that a MoE layer improves performance when its expert capacity is increased, because the extra experts are able to correct the output of the original ones, refining it rather than replacing it outright. Hence, their architecture fixes an MLP as the main function and selects a second one to correct its output, in order to achieve the representation correction phenomenum that they assume a larger expert capacity can provide, without actually increasing the number of experts used by the model, and therefore without paying the corresponding cost in parameters that a larger, standard MoE layer would require. This "Residual-MoE" component, however, is only one half of their overall proposal: the same work also observes, through an ablation they refer to as Phenomenon-I, that deeper transformer layers benefit more from a large number of experts than shallower ones do, since the latter do not appear to learn substantially different representations regardless of how many experts are made available to them. Motivated by this observation, the authors introduce a complementary "Pyramid-MoE" design, in which the number of experts is increased only in the deeper half of the network, while the shallower layers are left dense, or are given comparatively fewer experts, since little would be gained by enlarging them further. Combining Residual-MoE and Pyramid-MoE together yields the full Pyramidal Residual MoE architecture that gives the method its name, which the authors report reduces the total parameter count of a comparable standard MoE model by up to a factor of three, with no corresponding loss in quality. Beyond the architecture itself, the paper also proposes Mixture-of-Students, a further compression of Pyramidal Residual MoE obtained via staged knowledge distillation, and, since the work is as much a systems paper as an architectural one, it is accompanied by an inference system that combines expert, tensor and data parallelism, and that the authors report reduces inference latency and cost by up to 7.3 times with respect to prior MoE serving solutions.

Subsequently, DeepSeek-MoE [21] (figure 4.1) introduced a design motivated by intuitions

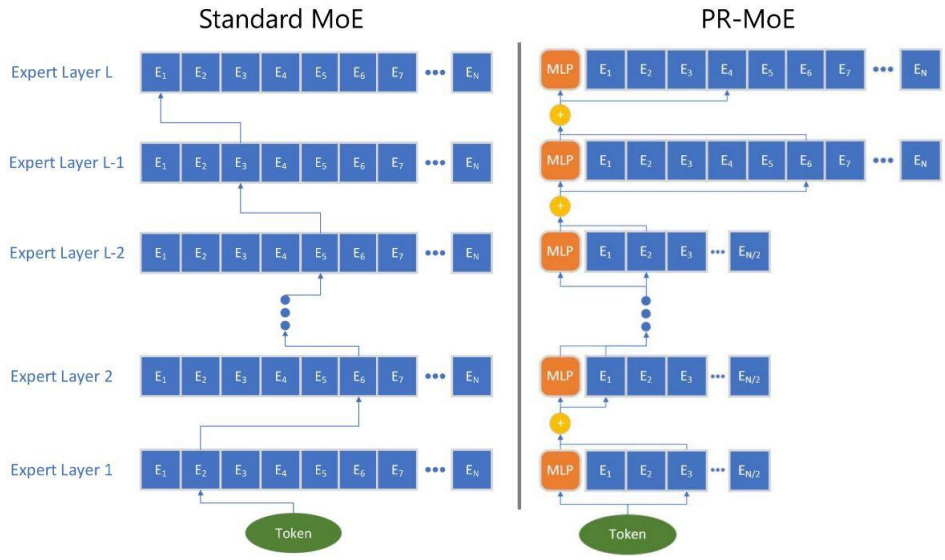


Figure 3.1: A comparison between a classic Top-1 MoE mechanism and a Pyramidal Residual MoE [79], where orange experts represent those experts which are always kept active.

similar to those that motivated our own work: the authors present a MoE model whose original K experts are split into m smaller, more fine-grained experts each, resulting in a total of Km experts, under the assumption that a larger number of narrower experts would allow each individual expert to specialize more precisely than would be possible with fewer, larger ones. In addition, and more importantly for what is directly related to our own work, they also propose grouping K_s of these fine-grained experts into a set that is kept always active, under the assumption that such a set would collect general, broadly applicable knowledge useful to every input, regardless of which of the remaining, routed experts happens to be selected for it. This is done specifically in order to avoid such general-purpose knowledge being redundantly, and wastefully, re-learned and stored inside more than one of the routed experts. The resulting MoE layer computes its outputs as follows:

$$h_t^l = \sum_{i=1}^{K_s} FFN_i(u_t^l) + \sum_{i=K_s+1}^{mK} g_{i,t} FFN_i(u_t^l) + u_t^l \tag{3.1}$$

Where u_t^l is the input of the layer for token t at layer l, the first K_s experts are the always-active shared ones, the remaining routed experts are selected out of the $mK - K_s$ fine-grained experts still available after reserving the shared ones, and $g_{i,t}$ is the gating weight associated with the i-th expert for token t, set to zero for every expert not selected by the top-k routing, exactly as in the standard

formulation of a gating function discussed previously. Please note how the number of experts activated by the gating function is changed with respect to a standard MoE layer, precisely in order to keep the number of active parameters equal to the number of active parameters in that standard MoE, so that the comparison between the two remains fair (also see figure 4.1 for a visual example of this correspondence). To discourage the routing collapse that this particular fine-grained setup would otherwise be especially prone to, given the larger pool of narrower experts among which the gating function must choose, the authors introduce an expert-level balance loss belonging to the same general family already discussed in Chapter 4.2:

$$\mathcal{L}_{ExpBal} = \alpha_1 \sum_{i=K_s+1}^{mK} f_i P_i \tag{3.2}$$

where f_i is the fraction of tokens in a training sequence that are actually routed to expert i, P_i is the average gating probability assigned to expert i over that same sequence, and α_1 is a hyperparameter weighting the overall strength of the balancing term relative to the main training loss. This is, up to a normalization constant that accounts for the larger number of fine-grained experts introduced by the segmentation described above, essentially the same $f_i \cdot P_i$ structure already employed by Switch Transformer and GShard, applied here to a different, more fine-grained pool of experts. For models that are distributed across multiple devices during training, a common requirement at the scale these architectures are typically deployed, the paper additionally introduces a device-level balance loss of an entirely analogous form, computed over groups of experts that are co-located on the same physical device rather than over individual experts considered in isolation, so as to keep computation balanced across the underlying hardware even in cases where individual experts within a given device happen to be unevenly used. It is worth noting, as an aside, that later revisions of the DeepSeek architecture, namely DeepSeek-V3, eventually replace both of these auxiliary losses altogether with a bias-based, auxiliary-loss-free balancing scheme, precisely in order to avoid the interference that such additional loss terms can otherwise introduce into the main training objective; this later development, however, is not the one used in the original DeepSeekMoE work discussed here, and is mentioned only for completeness.

More directly related to the specific focus of this work is Mixture-of-Head Attention (MoH) [45] (figure 3.3), which applies this same "shared and routed" principle not to the feed-forward layer, as in the works discussed above, but directly to the attention heads of a Multi-Head Attention

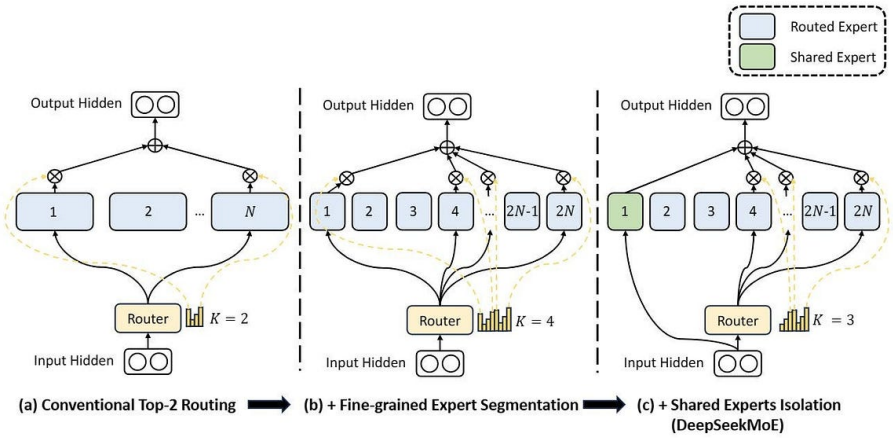


Figure 3.2: A comparison between a classic Top-2 MoE mechanism, and a MoE modified with the technologies introduced by DeepSeek-MoE [21]: a fine-grained MoE mechanism and a fine-grained mechanism with shared experts. Please note how, with respect to the classic mechanism, the variants have a different number of experts active, in order to keep the number of active parameters constant across the variants.

layer instead: individual heads are treated as experts in their own right, a subset of them is kept always active as shared heads, playing a role directly analogous to the shared experts discussed above, and the remaining heads are selected by a per-token gating function, with the outputs of all activated heads merged through a weighted summation, rather than through the plain, unweighted concatenation used in a standard Multi-Head Attention layer. Formally, denoting the h heads of the layer as $\{H_1, ..., H_h\}$, the output of MoH for a given token can be written as a weighted sum $\sum_i g_i H_i$ taken over the selected heads, where the coefficient g_i is non-zero only for the heads that are actually activated for that particular token, either because they belong to the always-active shared group by construction, or because they were specifically selected by the router among the pool of routed heads. In order to properly balance the relative contributions of the shared and routed groups, rather than simply treating every activated head identically regardless of which group it belongs to, the routing weights are computed through a two-stage process: one stage separately scores each individual head, and a second scores the two groups themselves, before the two resulting scores are combined into the final weight g_i. The authors report, through ablation studies, that both the mere presence of shared heads and this specific two-stage weighting scheme are independently

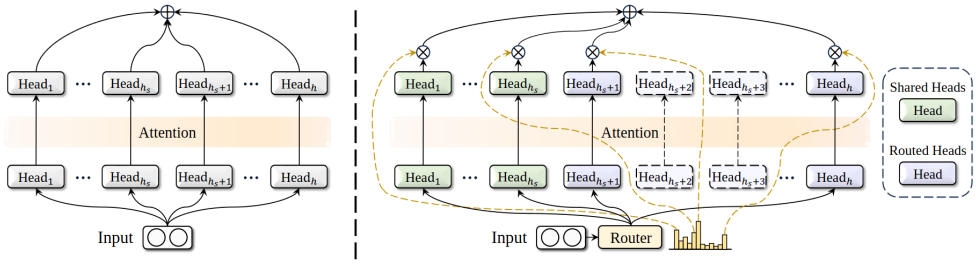


Figure 3.3: A comparison between a classic multi head attention mechanism, and a mixture of heads attention [45] mechanism.

important to the model's final performance, with the overall architecture remaining fairly robust to the exact proportion of heads reserved as shared, which they vary from roughly 14% to as much as 74% of the total in their experiments without a substantial change in outcome. This is, to the best of our knowledge, the closest existing work to the architecture proposed in this thesis, since it is the one that applies the shared-expert idea directly at the level of individual attention heads, rather than at the level of the feed-forward layer as in the works discussed previously. It is also, similarly to our own approach, demonstrated not only when training the architecture from scratch, but also when continue-tuning an already pretrained model, LLaMA3-8B specifically in their experiments, directly into the MoH architecture, which is methodologically close to how the models presented in this thesis are themselves obtained. The main differences with respect to our own approach, discussed in further detail in the Methodology chapter, concern primarily the granularity of the gating function, token-level in the case of MoH as opposed to sentence-level in our own work, and the way in which individual heads are grouped into experts to begin with, namely individual heads taken one at a time in MoH, as opposed to groups of heads, which we refer to as "slices", in our work.

The shared-expert design introduced independently by [79] and [21] has, in the time since these two works were published, been adopted well beyond the scope of these two original papers: the Qwen2 and Qwen2.5 model families [112], for instance, explicitly combine fine-grained expert segmentation with a mixture of shared and routed experts within their own MoE layers, citing both of the aforementioned works directly as motivation for this specific design choice, and reporting improved downstream performance over otherwise comparable MoE variants that do not make use of shared experts at all. Their base routing mechanism, prior to the addition of the shared-expert

term, follows the standard top-k MoE formulation already introduced earlier in this thesis, namely $\mathbf{p} = softmax(G(\mathbf{x}))$ followed by $\mathbf{y} = \sum_{i \in top_k(\mathbf{p})} \mathbf{p}_i E_i(\mathbf{x})$, onto which the shared-expert contribution is then added in essentially the same way already shown in Equation 3.1. Taken together, this broader adoption further supports the assumption that is central to this thesis: that separating always-active, general-purpose parameters from routed, specialized ones is a genuinely useful inductive bias in its own right, rather than being merely a peculiarity specific to a single architecture or model family.

A different, but complementary, line of work has instead explored conditioning expert selection directly on the domain of the input itself, rather than learning the association between inputs and experts in a purely end-to-end fashion as the works discussed so far all do. DEMix Layers [34] (figure 3.4) replace the feed-forward layer of a transformer with a set of domain-specific experts, and route each input to whichever expert is associated with its known domain label at training time, rather than through a learned gating function of the kind described elsewhere in this thesis; at inference time, however, when the domain of a given input is generally not known in advance, the authors show that a simple, entirely parameter-free weighted ensemble of the available experts can still generalize well, even to domains that are unseen during training, or that mix several known domains together. Notably, this ensemble is not weighted uniformly across experts: the authors instead estimate a posterior probability over the possible domains for the specific sequence currently being processed, maintained throughout as an exponential moving average that is updated continuously as the sequence is read token by token, and this posterior is then used directly as the set of mixing weights for the corresponding experts. As a consequence, the model's belief about which domain it is currently processing becomes progressively more confident, and the corresponding expert's contribution correspondingly more dominant in the mixture, the more of the sequence has already been observed. The same underlying architecture is also shown by the authors to support adding entirely new experts to cover domains that were not present during the original training, as well as removing existing experts altogether, in order to restrict the model's access to a particular domain, and crucially, both of these operations are possible without needing to retrain the rest of the network from scratch.

An approach closely related to DEMix Layers, Branch-Train-Merge [59] (figure 3.5), instead trains an entire set of independent expert language models, each one specialized on a different domain, in parallel with one another, and subsequently combines them at inference time through

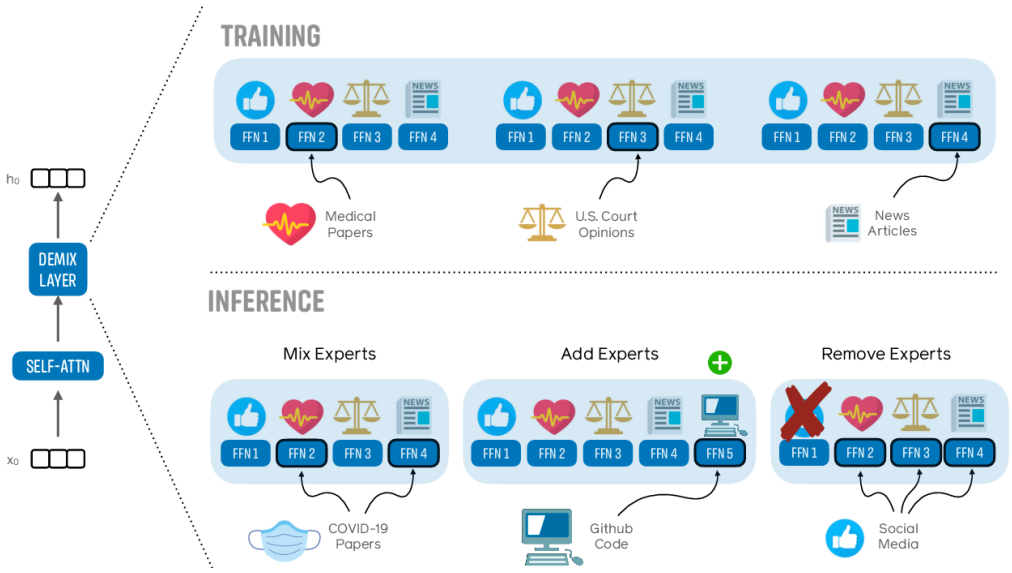


Figure 3.4: An explaination of the mechanism of DEMix Layers [34]. Particularly relevant for our work is the training method, which involves selecting explicitly a topic for each different expert.

either ensembling or parameter averaging, rather than relying on a per-layer gating mechanism of the kind employed by the works discussed above. Rather than training each of these domain experts entirely from scratch, which would be considerably more expensive, new experts are typically instead "branched" from an already existing expert, or from a weighted combination of several existing experts already present in the current set, and are then further trained on data drawn specifically from the new target domain, before finally being "merged" back into the overall pool of experts available for future use. This branching step is precisely what gives the method its name, and, according to the authors, it is also what makes training additional domain experts considerably cheaper in practice than training each one entirely independently, starting from a shared, generic initialization every time. At inference time, the authors further compare two distinct ways of combining the resulting set of experts: ensembling, which averages the output probabilities produced by each expert individually, against parameter averaging, which instead directly interpolates the underlying weights of the experts into a single, combined model. The former of these two approaches is reported to generalize noticeably better to new or mixed domains not directly represented by any single expert, but only at the cost of requiring a separate forward pass through every expert being ensembled; the latter, by contrast, collapses the entire set of experts back down into a single model requiring only a single forward pass, though generally at some cost

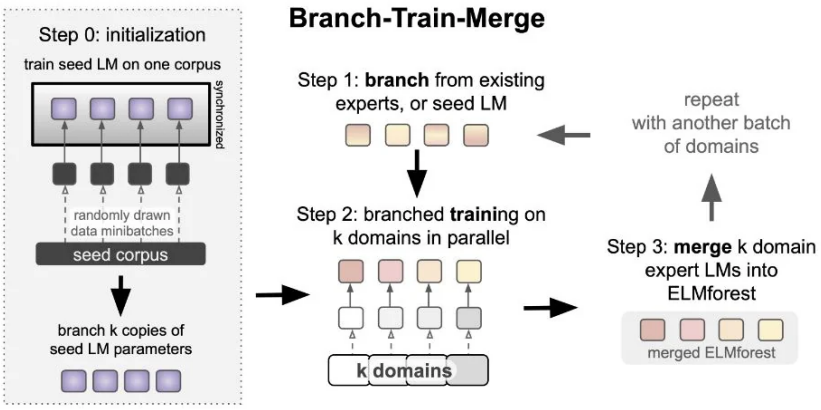


Figure 3.5: The training process used in Branch-Train-Merge [59].

to overall performance relative to ensembling. Both this work and DEMix Layers are conceptually close to the "Frozen router" gating strategy described in Section 4.2 of this thesis, since they both similarly rely on an external, predetermined notion of domain, rather than on an end-to-end learned association of the kind used elsewhere, to decide which parameters of the model should be engaged for a given input. The main difference, however, is that in our own work this domain-conditioned routing is applied at a considerably finer granularity than in either of these two works, selecting groups of attention heads within a single, unified model, rather than routing to entirely separate feed-forward experts, as in DEMix Layers, or to independently trained expert models altogether, as in Branch-Train-Merge.

4. Methodology

4.1 Architectural modifications

Referring back to the concepts presented in the previous section, we now describe in detail the architectural modifications that have been introduced in our model, starting from the standard attention mechanism employed by Llama and describing the Mixture of Experts mechanism that constitutes the core contribution of this thesis.

First, we describe in detail the algorithm (Algorithm 1, referring to the implementation found at [110]) used by Llama models [102, 101, 33] to compute Multi-Head Attention, since our own architecture is built directly as a modification of it. Being decoder-only models, the three sequences Q, K, V are here one and the same sequence, namely the hidden state $h \in \mathbb{R}^{B \times L \times d_m}$ resulting from the previous layer of the network. For the sake of parallelization, rather than computing each head's projection separately, the implementation instead involves three unified linear layers, $W^q \in \mathbb{R}^{d_m \times H d_h}$, $W^k \in \mathbb{R}^{d_m \times H_k d_h}$, and $W^v \in \mathbb{R}^{d_m \times H_k d_h}$, whose projections jointly contain the weights for all the heads at once, where d_m is the dimension of the model, B is the batch size, L is the length of the sequence, d_h is the dimension of the embeddings inside the attention mechanism, H is the number of query heads in Multi-Head Attention, and H_k is the number of key and value heads in Multi-Head Attention, which differs from H only when Grouped Query Attention is employed. The algorithm also involves the output projection layer $W^o \in \mathbb{R}^{H d_h \times d_m}$, used to aggregate the results of the individual heads back into a single representation.

Note that, with respect to the definition originally provided by [103], in Algorithm 1 the function $MultiHeadAttention(Q, K, V)$ is deliberately defined without the final application of W^o, which is instead carried out separately, a few lines later in the same algorithm; this separation will become useful shortly, once the algorithm is modified to accommodate our own Mixture of Experts mechanism.

Our model employs a Mixture of Experts mechanism at head level, applying it directly to the heads of the Multi-Head Attention layer described above, rather than to the feed-forward layer as is more commonly done in the literature. Similarly to the approach experimented with by [21], we

Algorithm 1 The standard Llama implementation of Self-Attention.

Require: h, W^q, W^k, W^v, W^o

$Q \in \mathbb{R}^{B\times L\times Hd_h} \leftarrow hW^q$

$K \in \mathbb{R}^{B\times L\times H_k d_h} \leftarrow hW^k$

$V \in \mathbb{R}^{B\times L\times H_k d_h} \leftarrow hW^v$

$Q, K \leftarrow RoPE(Q, K)$

if Using KV-caching **then**:

    Retrieve cache, update K, V

    Update cache with new K, V

end if

$O \in \mathbb{R}^{B\times L\times H\times d_h} = MultiHeadAttention(Q, K, V)$

Reshape O as $\mathbb{R}^{B\times L\times Hd_h}$ ▷ Concatenation

return OW^o

reserve part of the heads, named the "core", to be always active regardless of the input, and part of the heads, named "slices", to be selectively activated by a gating function depending on the input. This design is motivated by the assumption that the core, being exposed during training to data from every domain the model is trained on, will learn general knowledge useful to all inputs, while the individual slices, each exposed only to a fraction of the training data, will instead specialize to manage specific tasks and domains.

The model has a total number of heads $H = H_c + SH_s$, where H_c is the number of heads reserved for the core, S is the number of slices, and H_s is the number of heads contained in each individual slice. For each head $i \in \{0, 1, ..., H-1\}$, the result is computed as follows:

$$head_i(Q, K, V) = \begin{cases} Att(Q, K, V) & \text{if } i < H_c \vee (H_c + jH_s \leq i < H_c + (j+1)H_s \wedge G_j \in \text{Top-k}(G)) \\ \mathbf{0} & \text{else} \end{cases} \tag{4.1}$$

Where $G \in \mathbb{R}^S$ is the result of the gating function, K is the number of experts used for each input, and $j \in \{0, ..., S-1\}$ indexes through the slices. Note that, moving away from the main trend [31, 55, 91], in our work experts are not selected token-by-token: similarly to [53], a single set of experts is used for the whole input, under the assumption that the domain of a given input does

not change across its tokens, and that routing decisions can therefore be made once per sequence rather than independently for every token. The outputs of the individual heads are then aggregated exactly as in the standard case:

$$MultiHead(Q, K, V) = concat(head_1, head_2, ..., head_H)W^o. \tag{4.2}$$

Note that inactive heads do not contribute at all to the final output of the layer, since their individual output has already been set to the null vector by the piecewise definition above.

This method is implemented by a modified version of Algorithm 1, presented as Algorithm 2, which introduces a separate set of projection matrices for the core and for every individual slice. Here, H_{ck} and H_{sk} denote, respectively, the number of key and value heads for the core and for a single slice, which differ from H_c and H_s, only when Grouped Query Attention is used, exactly as H_k differed from H earlier. Using this notation, $W_c^q \in \mathbb{R}^{d_m \times H_c d_h}$ is the unified matrix for the Q projections of the core heads, $W_c^{\{k,v\}} \in \mathbb{R}^{d_m \times H_{ck} d_h}$ are the unified matrices for the K and V projections of the core heads, $W_j^q \in \mathbb{R}^{d_m \times H_s d_h}$ is the unified matrix for the Q projections of slice j, and $W_j^{k,v} \in \mathbb{R}^{d_m \times H_{sk} d_h}$ are the unified matrices for the K and V projections of slice j, for every slice $j \in \{0, ..., S-1\}$.

Algorithm 2 has been deliberately designed so that it can still make use of the efficient, pre-existing implementations of RoPE, KV-caching, and Multi-Head Attention already provided by Llama in the repository of [110], rather than requiring these components to be reimplemented from scratch. In other words, it has been designed specifically to produce Q, K, and V tensors of the same shape as, and playing the same role as, those produced by the matrices W^q, W^k, and W^v in Algorithm 1, so that everything downstream of this point in the original, optimized implementation can be reused without modification.

4.2 Gating strategies

Two main gating strategies have been experimented with for the gating function $G(x)$. In both cases, the function receives an entire sequence as input, and outputs a single set of experts to be used across all of its tokens, consistently with the sentence-level, rather than token-level, granularity discussed above. Both functions are, in practice, implemented by a `LlamaForSequenceClassification`

Algorithm 2 Our modified Attention mechanism.

Require: $h, W_c^q, W_c^k, W_c^v, \{W_0^q, ..., W_{S-1}^q\}, \{W_0^k, ..., W_{S-1}^k\}, \{W_0^v, ..., W_{S-1}^v\}, W^o$

$Q \in \mathbb{R}^{B \times L \times H d_h} \leftarrow \mathbf{0}$

$K \in \mathbb{R}^{B \times L \times H_k d_h} \leftarrow \mathbf{0}$

$V \in \mathbb{R}^{B \times L \times H_k d_h} \leftarrow \mathbf{0}$

$Q[..., 0 : H_c d_h] \leftarrow hW_c^q$

$K[..., 0 : H_{ck} d_h] \leftarrow hW_c^k$

$V[..., 0 : H_{ck} d_h] \leftarrow hW_c^v$

for $j \in \{0, ..., S-1\}$ **do** ▷ loop across the slices

  $B_j \leftarrow \{i | G_j(h[i]) \in \text{Top-k}(G_j(h[i])), i \in \{1, ..., L\}\}$ ▷ Indexing: the inputs in the batch that activate the slice i

  $Q[B_j, ..., (H_c + jH_s) : (H_c + (j+1)H_s)] \leftarrow h[B_j]W_j^q$

  $K[B_j, ..., (H_{ck} + jH_{sk}) : (H_{ck} + (j+1)H_{sk})] \leftarrow h[B_j]W_j^k$

  $V[B_j, ..., (H_{ck} + jH_{sk}) : (H_{ck} + (j+1)H_{sk})] \leftarrow h[B_j]W_j^v$

end for

$Q, K \leftarrow RoPE(Q, K)$

if Using KV-caching **then**:

  Retrieve cache, update K, V

  Update cache with new K, V

end if

$O \in \mathbb{R}^{B \times L \times H \times d_h} = MultiHeadAttention(Q, K, V)$

Reshape O as $\mathbb{R}^{B \times L \times H d_h}$ ▷ Concatenation

return OW^o

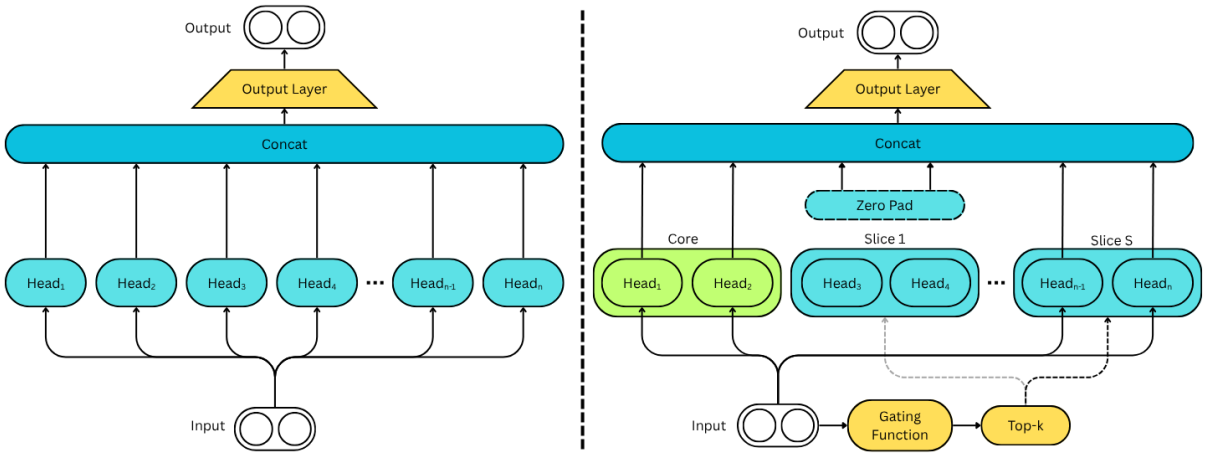


Figure 4.1: A comparison between a classic Multi-Head Attention module and the modified attention module we propose. The dotted line starting from the Top-k function is grey for non active slices. Note that the slices can contain an arbitrary number of heads, not necessarily two, like shown in this image.

model, provided by [110], adapted so that its classification head produces the gate values required by our architecture instead of standard class logits.

Frozen router: the first gating function is a classifier, pretrained on the data presented in Section 5.1, to classify the domain of a given sequence. Its weights are kept entirely frozen throughout the subsequent training of the rest of the model, which is what gives this strategy its name. The number of domains it can classify is set to match the number of slices present in every layer of the model. By associating every class predicted by the classifier with a corresponding slice index, this association is intended to ensure that such a slice, that is, every slice sharing that same index across every layer of the model, will receive only data relevant to the domain it has been associated with. This is intended to ensure that every slice becomes specialized for its associated domain, leading to a more interpretable model overall, in which the role played by each slice can be inspected and understood directly, rather than having to be inferred indirectly from its behaviour.

Learnable router: the second gating function instead produces, as its output, a matrix $G \in \mathbb{R}^{T \times S}$ for every sequence in the batch, where T is the number of layers in the model; the i-th row of this

matrix is then used as the input to the gating function for the i-th layer. In order to allow gradients to flow back through the weights of the `LlamaForSequenceClassification` model during training, Algorithm 2 has been slightly modified for this specific series of experiments, at the line updating the matrix Q:

$$Q[B_i, ..., (H_c + (i-1)H_s) : (H_c + iH_s)] \leftarrow h[B_i]W_i^q \cdot G[B_i, l, i] \tag{4.3}$$

Where l denotes the index of the current layer. In order to encourage balancing between the slices, and prevent the gating network from collapsing onto only a few of them, an auxiliary term has been added to the training loss, following the same general family of load-balancing losses already discussed earlier in this thesis:

$$L_{aux} = \frac{S}{T} \sum_{i=1}^{S} \sum_{j=1}^{T} f_i^{(j)} P_i^{(j)} \tag{4.4}$$

Where $f_i^{(j)}$ is the empirically calculated probability that slice i is the one with the highest logit for layer j, computed as the fraction of sequences in a batch for which this is the case, and $P_i^{(j)}$ is the mean of the softmaxed gate logits associated with slice i for layer j, averaged over the same batch.

5. Experiments

5.1 Data preparation

The experiments have been carried out using a dump of the English articles of Wikipedia [108]. The articles have been segmented into paragraphs with a naive approach, splitting the strings at every occurrence of the `\n\n` string. The resulting lines have been embedded using BGE-M3 [14] and clustered using K-means with 50 centroids. For every centroid, some associated embeddings have been sampled and their corresponding strings analyzed qualitatively in order to identify the domain they belonged to. In this operation, it was found that various centroids had fallen near clusters of paragraph titles, lists of references, empty paragraphs, links, and other elements similarly poor in content. This portion, labeled as noise and subsequently discarded, included 15 centroids, equivalent to approximately 24.7M lines of the dataset. As can be read in Table 5.1, this is a significant portion of the data with respect to the whole dataset, approximately 36% of it, but this should not come as a surprise, since Wikipedia articles often contain large sections of reference lists that carry little semantic content of their own.

For each of the remaining centroids, the qualitative analysis made it possible to define a common topic for the associated paragraphs. They have been organized by domain into 8 different clusters of comparable size (Table 5.1). The resulting dataset has a total of 42.79M lines, with an average length of 89 tokens and a standard deviation of 76 tokens.

Cluster	1	2	3	4	5	6	7	8	Total
Domain	History	Biographies	Geography	Politics	Sports	Art	Science	Pop Culture	
Lines	7.98M	5.24M	4.64M	4.95M	5.70M	5.63M	4.97M	3.65M	42.79M

Table 5.1: The clusters of the dataset.

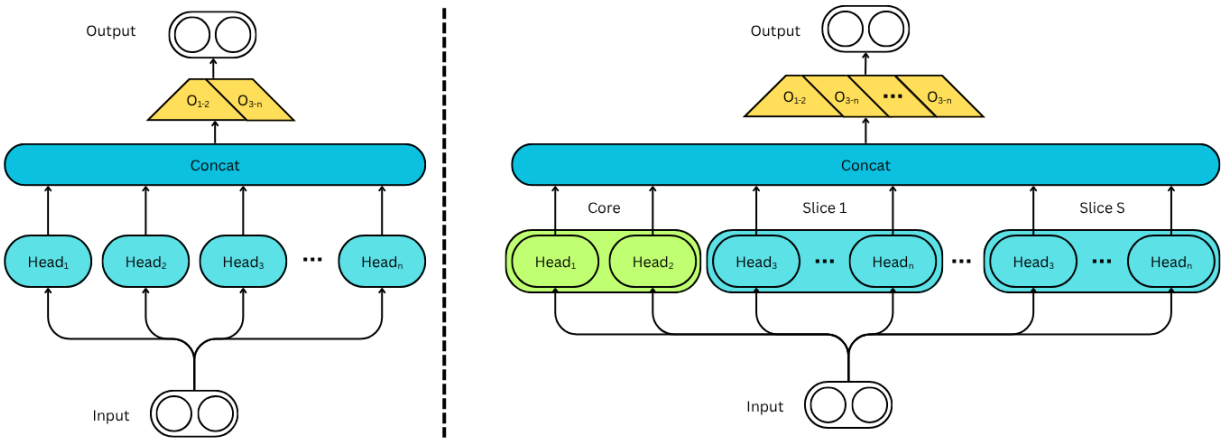


Figure 5.1: A visualization of how a standard attention module can be modified into our architecture for the initialization of a Finetuning. Note that the core can contain an arbitrary number of heads, not necessarily two like shown in figure.

5.2 Experiments Settings

We performed experiments with our architecture by finetuning a Llama3.2-1B [33] model. The modified architecture has been obtained by selecting a set of heads to be used as the core, and a set of heads to be used as a single slice; the heads corresponding to the slice have then been replicated S times, and the output projection layer W^o has consequently been expanded to match. Similarly to what was proposed by [15] in their "Net2WiderNet" algorithm, and also by [13] in their "bert2BERT" algorithm, this construction ensures that, with a Top-1 gating strategy and a frozen router, our model is functionally identical to the original Llama model, and very similar to it when employing a Top-2 gating strategy instead. Hence, the training process is able to start modifying the weights of the slices with knowledge from the dataset almost immediately, rather than having to first spend a portion of training simply adapting to the architectural modifications themselves. Experiments will show that, similarly to what was demonstrated by [67], even with a learnable gating strategy this initialization method allows the modified model to exploit the knowledge already present in the weights of the original model, reaching a performance comparable to the baseline before the end of training, and doing so in fewer steps than a model trained entirely from scratch would require to reach the same loss.

The training used a standard Huggingface `Trainer` [110], except for the experiments with a learnable router, where the class was modified to support the auxiliary loss term. All experiments have been performed with a 3×10^{-5} learning rate, scaling linearly across the training (see the documentation of Huggingface for further details about the standard learning rate scheduler), a batch size of 16 with 32 accumulation steps, and 1900 training steps for a single epoch. The training has been performed on 2 GPUs, NVIDIA L40S, for an average duration of 40 GPUh.

In all experiments, the number S of slices is equal to 8, matching the number of clusters of the dataset. The frozen router presented in Section 4.2 is trained to associate every line of the dataset with its cluster, representing its domain. The model used is a Llama model designed for sequence classification, provided by [110], configured with 2 layers, 4 attention heads, a hidden size equal to 128, and the same tokenizer used by the main models, resulting in a model with 75M parameters. In the version not updated during training, the head of the model outputs a number of classes equal to the number of clusters of the dataset; on the other hand, in the learnable version, updated during training, the model head instead outputs $T \times S$ logits, where T is the number of layers.

All models have been evaluated on the following metrics, on a 100K-line subset of the dataset described in Section 5.1, not overlapping with the training subset:

- **Accuracy (Acc)**

- **Perplexity (PPL)**, introduced by [43]. This metric is expected to decrease as the quality of the language model increases, and is defined, for a sequence of tokens $X = \{x_0, x_1, ..., x_t\}$, as:

$$PPL(X) = -\frac{1}{t}\sum_{i=0}^{t} \log p_\theta(x_i|x_{<i})$$

  Note how this metric increases when the model is uncertain about the prediction for a given token, or associates the theoretically correct token with a low probability, and, conversely, how it decreases when the correct token is instead associated with a high probability.

- **ROUGE** (Recall Oriented Understudy for Gisting Evaluation), introduced by [60], a set of metrics evaluating the differences between a human-written sentence (in our case, the original text from Wikipedia [108]) and the text generated by the model. In particular, it evaluates overlapping 1-grams (**ROUGE-1, R-1**), 2-grams (**ROUGE-2, R-2**), and the longest common subsequence (**ROUGE-L, R-L**).

- **MAUVE**, introduced by [74], measures the differences between the model distribution Q and the target distribution P provided by humans, by approximating the area under the curve defined by:

  $$C(P,Q) = \{(exp(-cKL(P|R_\lambda)), exp(-cKL(Q|R_\lambda))) : R_\lambda = \lambda P + (1-\lambda)Q, \lambda \in (0,1)\}$$

  Where KL is the Kullback-Leibler divergence, and the resulting MAUVE score is precisely this area under the curve. Note that this metric's value lies in $(0, 1]$, and is equal to 1 if and only if $P = Q$.

A set of benchmarks has been employed to evaluate the capabilities of the models on tasks lying outside the distribution of the dataset:

- **GLUE**, introduced by [105], provides a wide range of natural language processing benchmarks with which to evaluate the capabilities of a model on various tasks: linguistic acceptability (CoLA [107]), sentiment analysis (SST-2 [95]), semantic equivalence between two sequences (MRPC [25], QQP), entailment of a premise sentence with respect to a hypothesis (MNLI [109], RTE), association of a question with its answer (QNLI [80]), and association of a pronoun with its referent (WNLI [56]).

- **GSM8k**, introduced by [17], is a set of grade school math problems, designed to evaluate a model's capability of performing multi-step mathematical reasoning.

- **GroundCocoa**, introduced by [51], is a benchmark evaluating the compositional and conditional reasoning of a model by exposing it to the problem of flight booking.

- **LAMBADA**, introduced by [72], is a benchmark that uses word prediction to evaluate the ability of a model to keep track of information across a long discourse, in various languages.

- **The Global MMLU**, introduced by [94], is a translation benchmark spanning 42 languages, with a particular focus on cultural biases and their impact on translation tasks.

5.3 Experiments Results

Two series of experiments have been performed: the first with full finetuning, modifying all the attention layers of the model (Table 5.3), and the second modifying and finetuning only the top half

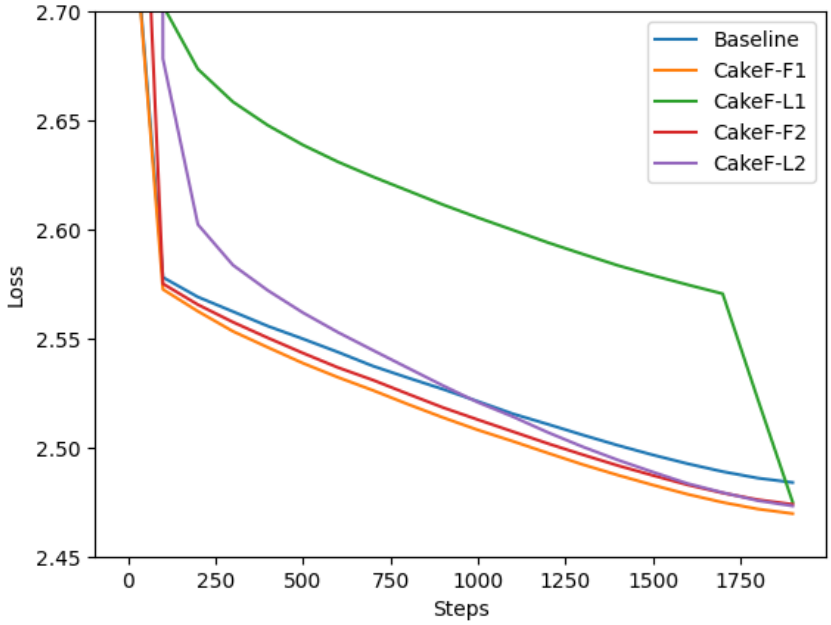


Figure 5.2: Losses of the models trained with full finetuning.

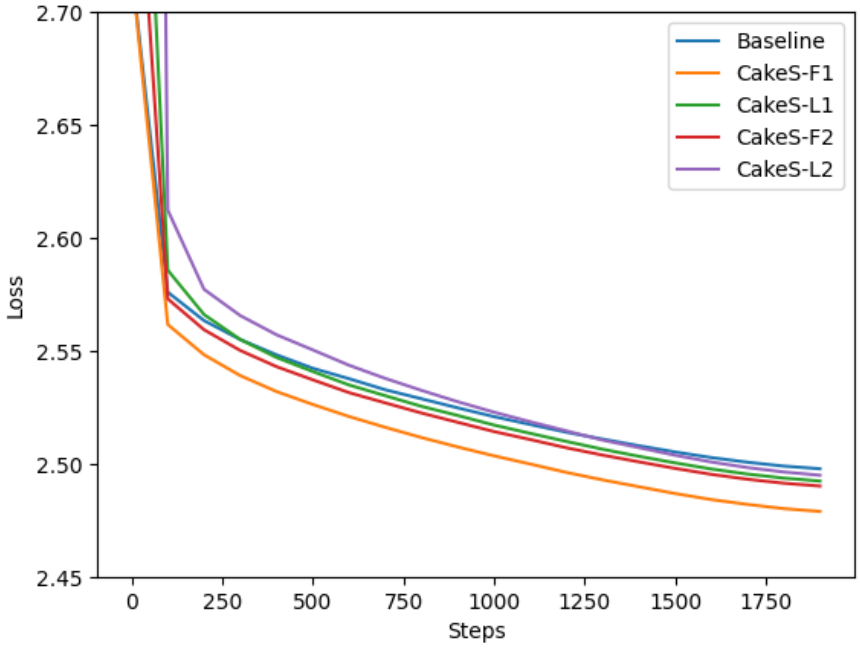


Figure 5.3: Losses of the models trained with a subset of the parameters.

of the layers (Table 5.4). As can be seen in the plots (Figures 5.2 and 5.3), the losses of the custom models reach lower values than those of the baseline in both cases. This is confirmed by the metrics shown in the tables, where the custom models reach comparable performance with respect to the baseline.

Note how the losses of the experiments with the frozen router reach lower values than the baseline in the early steps of training. This is true especially for the models with a frozen router and a single active section, which are functionally identical to the baseline right after initialization, and can therefore start learning new knowledge from the very first steps, while also exploiting a larger amount of parameters, which explains the lower loss. On the other hand, the other versions of the model also have to change their weights in order to adapt to the architectural changes: they presumably have to tune the weights of their attention output layers to support the simultaneous activation of 2 slices, and/or tune the attention projections to support the learnable gating, which is multiplied with the query values in order to allow the gradient to reach the weights of the router. This explains why such variants require more steps to surpass the baseline. Naturally, the variants for which this phenomenon is most relevant are the ones combining 2 active slices with the learnable router.

Regardless of the number of steps a given variant needs before surpassing the loss of the baseline, these experiments clearly show that the initialization method we chose is able to efficiently transfer the parameters of the original model (Llama3.2-1B) into our architecture, without requiring a separate pretraining phase, even when the model is initialized into a configuration that is not functionally identical to the baseline.

The performance shown in Table 5.3 shows how models with a learnable router tend to achieve better accuracy, perplexity, and MAUVE with respect to the baseline, and with respect to the versions using a frozen router. This suggests that the gradient is able to tune the weights of the router, leading, after training, to a level of performance in distributing the inputs that the frozen router, which relies instead on a human-designed, domain-based routing scheme, is not able to match. It is also interesting to notice how the single-section variants, both the ones trained with full finetuning and those trained with subset finetuning, outperform the baseline on MAUVE, while being outperformed by it on the ROUGE metrics. This leads to the hypothesis that such models are able to produce outputs with a more fluent and realistic distribution, even if the final result

deviates from the exact wording of the original line, used as reference in the dataset, more often than the baseline does. Since these metrics are calculated on the evaluation dataset, which, like the training dataset, is drawn from the main dataset presented in Section 5.1, the two, even though non-overlapping, may share a similar distribution, being sampled randomly from the same source [108], and therefore having similar styles and structure. This consideration, taken together with the results on the MAUVE metric, suggests that the baseline achieves better ROUGE scores due to overfitting, rather than due to a genuinely superior effectiveness and human acceptability of the generated text [30], since the ROUGE metrics calculate exact matching between tokens, agnostically with respect to the complete probability distribution the model has actually learned. A similar argument can be made for the perplexity metric, which, like MAUVE, takes into account the value of the logits rather than exact matching: the full-finetuned models with a learnable router outperform the baseline on this metric as well.

In all cases, comparing two models that share the same router and training strategy but differ in the number of active sections shows that the models using a single active section and those using two active sections achieve almost identical accuracy, with an absolute percentage difference lower than or equal to 0.1, while the single-section version performs better on both MAUVE and perplexity, with the sole exception of the models with a learnable router trained with subset finetuning, for perplexity. This suggests that the increase in performance granted by the larger amount of parameters used at inference time by the two-section models is not enough to compensate for the fact that the weights of the model must first adjust, during the early steps of training, to support the architectural modifications introduced by the second active section. In other words, the single-section variants may be able to learn new knowledge from the data earlier in training than the two-section variants, and are therefore able to reach a better performance in fewer steps.

A comparison between the full-finetuned models and their corresponding subset-finetuned variants shows that the former achieve better accuracy. This may be explained by the smaller number of parameters the training could act upon in the latter. On the other hand, CakeS-F1, with a single active section and a frozen router, is able to match the performance of CakeF-L1, its full-finetuned counterpart, which has a single active section and a learnable router, on MAUVE and perplexity. This may be explained by considering that, even though the finetuning acts on fewer parameters, it acts specifically on the top layers, the ones where one would expect to find more

Model	Gating	Active Slices	Finetuning
CakeF-F1	Frozen	1	Full
CakeF-L1	Learnable	1	Full
CakeF-F2	Frozen	2	Full
CakeF-L2	Learnable	2	Full
CakeS-F1	Frozen	1	Subset
CakeS-L1	Learnable	1	Subset
CakeS-F2	Frozen	2	Subset
CakeS-L2	Learnable	2	Subset

Table 5.2: Reference for design choices of the variants.

Model	Gating	Active Slices	Acc(↑)	PPL(↓)	R-1(↑)	R-2(↑)	R-L(↑)	MAUVE(↑)
Baseline	N/A	N/A	48.97%	6.668	**0.2232**	**0.0595**	**0.1758**	0.3436
CakeF-F1	Frozen	1	49.15%	6.532	0.2217	0.0590	0.1750	0.3536
CakeF-L1	Learnable	1	49.42%	**6.359**	0.2194	0.0574	0.1732	**0.3636**
CakeF-F2	Frozen	2	49.05%	6.628	0.2208	0.0581	0.1740	0.3424
CakeF-L2	Learnable	2	**49.48%**	6.587	0.2078	0.0523	0.1649	0.3126

Table 5.3: The performance of the full finetuning experiments on the metrics.

domain-specific structures [114, 41]. This implies that such subset-finetuning, with respect to full finetuning, has less influence on the weights whose modification might positively affect performance on the training set, but negatively affect general performance on out-of-distribution data. This may be the leading cause behind its outperforming its full-finetuned counterpart.

As shown in Table 5.5, it is clear that the variants perform similarly with respect to the baselines on most of the GLUE benchmarks. It is worth noting how CakeF-L1 outperforms both the baseline and the other full-finetuned variants on MNLI and RTE, which suggests this variant has better reasoning capabilities, since these tasks consist of the entailment of a premise with respect to a hypothesis. The fact that the same model also outperforms the other variants on QNLI and WNLI, which consist of the association of a question with its answer and of a pronoun with its referent,

Model	Gating	Active Slices	Acc(↑)	PPL(↓)	R-1(↑)	R-2(↑)	R-L(↑)	MAUVE(↑)
Baseline	N/A	N/A	48.78%	6.496	**0.2202**	**0.0582**	**0.1740**	0.3510
CakeS-F1	Frozen	1	48.94%	**6.410**	0.2193	0.0574	0.1721	**0.3746**
CakeS-L1	Learnable	1	**49.17%**	6.614	0.2172	0.0565	0.1718	0.3590
CakeS-F2	Frozen	2	48.87%	6.518	0.2175	0.0565	0.1720	0.3469
CakeS-L2	Learnable	2	49.12%	6.571	0.2093	0.0535	0.1665	0.3376

Table 5.4: The performance of the subset finetuning experiments on the metrics.

Model	CoLA	MNLI	MNLI-mis	MRPC-acc	MRPC-f1	QNLI	QQP-acc	QQP-f1	RTE	SST-2	WNLI
Baseline (F)	**0.014**	34.9%	35.2%	67.2%	0.798	**53.2%**	53.0%	0.354	55.2%	**80.4%**	47.9%
CakeF-F1	-0.020	35.2%	36.2%	66.7%	0.798	51.2%	52.9%	0.348	60.7%	72.5%	47.9%
CakeF-L1	-0.031	**36.4%**	**37.0%**	63.2%	0.760	51.3%	**57.4%**	0.209	**62.1%**	76.5%	**53.5%**
CakeF-F2	-0.015	34.8%	35.3%	64.2%	0.765	51.0%	56.3%	0.231	56.7%	78.4%	43.7%
CakeF-L2	0.011	34.7%	35.3%	**67.4%**	**0.804**	51.0%	49.0%	**0.485**	59.9%	76.5%	46.5%
Baseline (S)	**-0.015**	35.6%	35.6%	67.7%	0.805	**51.9%**	49.9%	0.362	58.5%	73.7%	52.1%
CakeS-F1	-0.063	**35.8%**	35.6%	68.1%	0.809	50.7%	51.7%	0.336	59.2%	77.7%	52.1%
CakeS-L1	-0.030	35.7%	**35.7%**	**68.4%**	**0.810**	50.9%	52.4%	0.305	60.3%	**79.5%**	50.7%
CakeS-F2	-0.066	35.7%	35.5%	64.7%	0.779	50.1%	**52.9%**	0.309	58.8%	79.0%	**54.9%**
CakeS-L2	-0.122	35.5%	35.6%	67.9%	0.807	50.6%	52.6%	**0.422**	**60.3%**	75.7%	47.9%

Table 5.5: Results on GLUE.

respectively, points in the same direction.

An interesting fact about the subset-finetuned models is that they outperform their baseline on SST-2, a sentiment analysis task. On the same task, the full-finetuned baseline instead outperforms our variants. This leads to the hypothesis that some of the weights important for this task lie in the top layers of the model [114, 41]: in the full-finetuned version, since more weights were available for the gradient to act upon, the relevant weights were evidently perturbed less, whereas in the subset-finetuned baseline such weights were modified more heavily in order to improve performance on the training set, which, we highlight, did not include any sentiment-analysis-related lines. This, in turn, leads to the further hypothesis that the subset-finetuned variants, having more parameters available to update within the portion of the network being trained, were able to improve their knowledge of

the dataset with a less severe disruption of the relevant weights [46].

Model	GSM8k		GroundCocoa	Global-MMLU			
	exact	flexible	acc	Spanish	French	Italian	German
Baseline (F)	2.88%	3.87%	20.6%	29.5%	30.8%	27.5%	29.0%
CakeF-F1	**3.34%**	4.02%	20.4%	31.6%	**32.8%**	29.3%	27.3%
CakeF-L1	2.35%	2.95%	**21.0%**	29.0%	28.0%	28.5%	25.5%
CakeF-F2	3.34%	4.09%	20.1%	**33.0%**	31.8%	27.0%	28.8%
CakeF-L2	3.34%	**4.47%**	20.4%	32.3%	31.5%	**31.5%**	**31.5%**
Baseline (S)	4.17%	4.85%	19.7%	29.3%	31.5%	33.0%	31.0%
CakeS-F1	**4.47%**	4.85%	19.1%	**32.3%**	**32.5%**	**34.8%**	**32.0%**
CakeS-L1	4.32%	5.00%	19.0%	31.0%	31.3%	33.0%	32.0%
CakeS-F2	3.26%	**5.46%**	19.1%	31.0%	32.5%	34.0%	31.3%
CakeS-L2	3.49%	5.31%	**19.7%**	30.5%	31.5%	28.5%	29.8%

Table 5.6: Results of the models on various benchmarks.

Model	German		English		Spanish		French		Italian		Dutch	
	PPL	acc	PPL	acc	PPL	acc	PPL	acc	PPL	acc	PPL	acc
Baseline (F)	**17.29**	**46.7%**	6.14	61.2%	124.81	38.8%	33.17	42.07%	29.54	41.22%	**113.71**	**36.68%**
CakeF-F1	18.00	45.6%	**5.94**	62.1%	125.87	39.1%	33.62	42.67%	29.09	41.99%	122.00	36.54%
CakeF-L1	18.43	44.7%	5.98	61.4%	**115.94**	**39.9%**	33.00	43.06%	29.56	41.35%	120.55	35.94%
CakeF-F2	19.11	44.7%	6.00	**62.3%**	127.70	39.0%	34.31	42.27%	30.91	41.26%	129.04	35.77%
CakeF-L2	19.19	44.5%	5.96	62.0%	116.92	39.9%	**32.42**	**43.37%**	**28.73**	**42.23%**	121.77	36.13%
Baseline (S)	14.68	49.0%	5.66	63.1%	109.78	41.0%	27.68	44.54%	23.10	44.44%	101.70	38.81%
CakeS-F1	15.21	48.5%	5.52	63.9%	109.89	41.4%	28.45	44.94%	23.33	44.56%	103.52	38.79%
CakeS-L1	**14.02**	50.0%	5.45	64.1%	99.59	42.0%	**26.96**	**45.74%**	22.81	44.79 %	**95.65**	39.38%
CakeS-F2	14.26	**50.1%**	5.48	64.2%	101.63	42.0%	27.45	45.62 %	22.39	45.37 %	97.25	**39.39%**
CakeS-L2	14.43	49.5%	**5.43**	**64.2%**	**97.12**	**42.6%**	28.02	45.18%	**21.99**	**45.70%**	96.17	39.24%

Table 5.7: Results on LAMBADA.

Tables 5.6 and 5.7 show how the custom models tend to outperform the baselines on out-of-

distribution data such as GSM8k (arithmetic) and Global-MMLU (multi-lingual understanding), while achieving comparable results on tasks that involve more general language processing, such as GroundCocoa, LAMBADA, and GLUE. This leads to the hypothesis that our models are less sensitive to catastrophic forgetting [92, 66] with respect to our baseline, possibly because the presence of shared heads in the core reduces the specialization of such heads, and because the heads in the slices, each receiving only a fraction of the data, are able to sparsify and distribute learning across the model [58, 46], leading to less severe overfitting with respect to the baseline.

After analyzing the performance of the variants of our architecture, we choose to propose the *L2* variants as our model (*CakeF-L2* and *CakeS-L2*), for their good perfomance on accuracy and their overall good performance on out-of-distribution data.

6. Conclusion

This thesis set out from the observation that the growing computational and energy cost of modern deep learning models [48, 96, 73] is largely driven by the fact that dense architectures activate all of their parameters for every input, regardless of how relevant each of those parameters actually is to the task at hand. Mixture of Experts [42, 91] addresses this inefficiency by activating only a sparse subset of a model's parameters per input, and, in its more recent variants [21], by additionally reserving a subset of always-active parameters intended to capture general, domain-agnostic knowledge, leaving the routed parameters free to specialize. While this idea has been explored extensively at the level of the feed-forward layer, and, to a lesser extent, at the level of individual attention heads [117, 45], its combination with an always-active core of shared heads had not, to the best of our knowledge, been previously investigated at the attention level.

Building on this gap, we proposed an architecture that applies the Mixture of Experts paradigm directly to the attention heads of a transformer, dividing them into a "core", kept always active, and a set of "slices", selectively activated by a gating function on a per-sequence, rather than per-token, basis. Two gating strategies were designed and compared: a frozen router, pretrained to classify the domain of an input and thereby assign it directly to a specific slice, and a learnable router, trained jointly with the rest of the model through an auxiliary load-balancing loss. Both strategies were implemented as a modification of a pretrained Llama3.2-1B model [33], using a function-preserving initialization scheme that allows the modified architecture to start from a state functionally close to the original model, rather than requiring training from scratch.

We evaluated this architecture by finetuning it on a domain-clustered subset of Wikipedia [108], comparing eight variants of our model, obtained by combining the two gating strategies, one or two active slices, and two different finetuning regimes, against equivalent Llama baselines. The experiments show that the presented architecture achieves slightly superior performance with respect to a baseline with a similar number of parameters active at runtime, while showing signs, of being less vulnerable to catastrophic forgetting on out-of-distribution data. This may be explained by the fact that the slices, each receiving a smaller and more specific portion of the training data than the corresponding parameters in the baseline, are able to preserve more of their original,

pretrained content, limiting the performance loss typically observed on tasks lying outside the training distribution.

In the remainder of this chapter, we discuss the limitations of the current work and propose directions along which it could be improved.

6.1 Limitations and Future Improvements

It exists evidence that in a transformer architecture the feed-forward layers tend to have more sparse activations and contain domain-specific activations [10, 98, 71], which suggests that our "core" and "slices" mechanism could be implemented in the feed forward layer, similarly to how it has been implemented in the attention mechanism, leading to a further improvement with respect to the baseline.

A second, related limitation concerns our choice to apply the architecture to an already pretrained model rather than training it from scratch: since a pretrained transformer already possesses an emergent modular structure of its own [118], it is reasonable to expect that structure to be difficult for gradient descent to substantially reorganize, as the shortest path towards a lower loss is unlikely to require heavily disrupting structures the model has already learned. This suggests that our architecture, if trained from scratch, could plausibly have outperformed a same-sized baseline by a wider margin than observed in our finetuning experiments.

Finally, a more careful selection of which heads to assign to the core may also improve results, particularly if the architecture is applied to a pretrained model rather than trained from scratch. In this work, the split between core and slice heads was determined randomly, by simply fixing a boundary between them. Instead, analyzing the activation patterns of individual heads and assigning the most frequently and strongly activated ones to the core, under the assumption that heads activating often are more likely to encode general-purpose knowledge, could allow the remaining, less frequently used heads to specialize more effectively for the specific domains they are exposed to within their slice.

Acknowledgements

Vorrei ringraziare tutti quelli che sono stati vicini in questo percorso e hanno speso un po' del loro tempo per ascoltare e consigliare.

Non posso non cominciare dai colleghi, con i quali abbiamo condiviso tempo, gag, preoucupazioni, successi e fallimenti, con alcuni per due anni, con altri molto di più: Thomas, Federico, Nicola e Davide che si sono poi uniti al laboratorio e hanno aiutato spesso con pareri e consigli, non solo sulla tesi, Beatrice, Fabio, Alessia, Alessandro, Mattia, i due Francesco, Gibbo (oggettivamente nessuno lo chiama Filippo, neanche i professori) il quale ci manca molto perché è stato prima in Svizzera e ora è in Giappone, Lorenzo, Marianna, Giada, i due Matteo, Teresa, Davide, Alex e da lontano anche Gianmarco.

Anche e soprattutto la mia famiglia va ringraziata, perché tra tutti mi ha sopportato per più tempo: i miei genitori e i miei fratelli, i nonni, in particolare la nonna Luisa, gli zii.

Un ringraziamento deve andare anche agli amici Pietro, Francesco, Davide, Matteo, Vittorio, per la loro grande saggezza. Poi Gabriele, Stefano e Matteo, per gli stessi motivi.

Poi un non si possono non ringraziare i colleghi del laboratorio per il loro aiuto: Marco, che sempre si prodiga per mantenere un'atmosfera fresca e calma, Sara, Alberto, William, Silvia, Evelyn, Davide, Tobia, Leonardo, Federico.

Un ringraziamento speciale va ai bros della "Comu" che mi hanno sentito spesso ripetere ansie e preoucupazioni su questa tesi: Antonio, Emmanuele, le tre Benedetta, Sergio e Rita, Immacolata, Sara, Valeria e Leopoldo, Teresa, Esa e Luca, Giovanni e Chiara, Guglielmo (il quale, non facendo niente per aiutarmi mi ha aiutato molto), le due Anna, Annalisa, Sebastiano, Leonardo, Mattia, Giorgia, Stefano, Davide, Simona, Vittoria. Con loro, anche gli amici Francesco, Pietro, Davide, Ester, Chiara, e l'illustrissimo Emanuele.

Bibliography

[1] Joshua Ainslie et al. *GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints*. 2023. arXiv: 2305.13245 [cs.CL]. URL: https://arxiv.org/abs/2305.13245.

[2] Amjad Almahairi et al. *Dynamic Capacity Networks*. 2016. arXiv: 1511.07838 [cs.LG]. URL: https://arxiv.org/abs/1511.07838.

[3] Jason Ansel et al. "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation". In: *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, Apr. 2024. DOI: 10.1145/3620665.3640366. URL: https://docs.pytorch.org/assets/pytorch2-2.pdf.

[4] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. *Layer Normalization*. 2016. arXiv: 1607.06450 [stat.ML]. URL: https://arxiv.org/abs/1607.06450.

[5] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. *Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation*. 2013. arXiv: 1308.3432 [cs.LG]. URL: https://arxiv.org/abs/1308.3432.

[6] Andreas Blattmann et al. "Align your latents: High-resolution video synthesis with latent diffusion models". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2023, pp. 22563–22575.

[7] John S. Bridle. "Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition". In: *Neurocomputing: Algorithms, Architectures and Applications*. Ed. by Françoise Fogelman Soulié and Jeanny Hérault. Vol. 68. NATO ASI Series F. Berlin, Heidelberg: Springer, 1990, pp. 227–236. DOI: 10.1007/978-3-642-76153-9_28.

[8] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL]. URL: https://arxiv.org/abs/2005.14165.

[9] Davide Caffagni et al. “Mind the Heads: Topological Representation Alignment for Multimodal LLMs”. In: *arXiv preprint arXiv:2606.23885* (2026).

[10] Weilin Cai et al. “A Survey on Mixture of Experts in Large Language Models”. In: *IEEE Transactions on Knowledge and Data Engineering* (2025), pp. 1–20. ISSN: 2326-3865. DOI: 10.1109/tkde.2025.3554028. URL: http://dx.doi.org/10.1109/TKDE.2025.3554028.

[11] Nicolas Carion et al. *End-to-End Object Detection with Transformers*. 2020. arXiv: 2005.12872 [cs.CV]. URL: https://arxiv.org/abs/2005.12872.

[12] Daniel Cer et al. “SemEval-2017 Task 1: Semantic Textual Similarity Multilingual and Crosslingual Focused Evaluation”. In: *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*. Association for Computational Linguistics, 2017, pp. 1–14. DOI: 10.18653/v1/s17-2001. URL: http://dx.doi.org/10.18653/v1/S17-2001.

[13] Cheng Chen et al. “bert2bert: Towards reusable pretrained language models”. In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2022, pp. 2134–2148.

[14] Jianlv Chen et al. *BGE M3-Embedding: Multi-Lingual, Multi-Functionality, Multi-Granularity Text Embeddings Through Self-Knowledge Distillation*. 2024. arXiv: 2402.03216 [cs.CL].

[15] Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. “Net2net: Accelerating learning via knowledge transfer”. In: *arXiv preprint arXiv:1511.05641* (2015).

[16] Kevin Clark et al. *ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators*. 2020. arXiv: 2003.10555 [cs.CL]. URL: https://arxiv.org/abs/2003.10555.

[17] Karl Cobbe et al. *Training Verifiers to Solve Math Word Problems*. 2021. arXiv: 2110.14168 [cs.LG]. URL: https://arxiv.org/abs/2110.14168.

[18] George Cybenko. “Approximation by Superpositions of a Sigmoidal Function”. In: *Mathematics of Control, Signals and Systems* 2.4 (1989), pp. 303–314. DOI: 10.1007/BF02551274.

[19] Fabio D’Oronzio et al. “GramSR: Visual Feature Conditioning for Diffusion-Based Super-Resolution”. In: *International Conference on Pattern Recognition*. Springer. 2026, pp. 326–341.

[20] Ido Dagan, Oren Glickman, and Bernardo Magnini. “The PASCAL Recognising Textual Entailment Challenge”. In: *Machine Learning Challenges. Evaluating Predictive Uncertainty, Visual Object Classification, and Recognising Tectual Entailment*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 177–190. ISBN: 978-3-540-33428-6.

[21] Damai Dai et al. *DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models*. 2024. arXiv: 2401.06066 [cs.CL]. URL: https://arxiv.org/abs/2401.06066.

[22] Zihang Dai et al. *Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context*. 2019. arXiv: 1901.02860 [cs.LG]. URL: https://arxiv.org/abs/1901.02860.

[23] Yann N. Dauphin et al. *Language Modeling with Gated Convolutional Networks*. 2017. arXiv: 1612.08083 [cs.CL]. URL: https://arxiv.org/abs/1612.08083.

[24] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL]. URL: https://arxiv.org/abs/1810.04805.

[25] William B. Dolan and Chris Brockett. “Automatically Constructing a Corpus of Sentential Paraphrases”. In: *Proceedings of the Third International Workshop on Paraphrasing (IWP2005)*. 2005. URL: https://aclanthology.org/I05-5002/.

[26] Alexey Dosovitskiy et al. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. 2021. arXiv: 2010.11929 [cs.CV]. URL: https://arxiv.org/abs/2010.11929.

[27] Shihan Dou et al. *LoRAMoE: Alleviate World Knowledge Forgetting in Large Language Models via MoE-Style Plugin*. 2024. arXiv: 2312.09979 [cs.CL]. URL: https://arxiv.org/abs/2312.09979.

[28] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *J. Mach. Learn. Res.* 12.null (July 2011), pp. 2121–2159. ISSN: 1532-4435.

[29] David Eigen, Marc'Aurelio Ranzato, and Ilya Sutskever. *Learning Factored Representations in a Deep Mixture of Experts*. 2014. arXiv: 1312.4314 [cs.LG]. URL: https://arxiv.org/abs/1312.4314.

[30] Alexander R Fabbri et al. "Summeval: Re-evaluating summarization evaluation". In: *Transactions of the Association for Computational Linguistics* 9 (2021), pp. 391–409.

[31] William Fedus, Barret Zoph, and Noam Shazeer. *Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity*. 2022. arXiv: 2101.03961 [cs.LG]. URL: https://arxiv.org/abs/2101.03961.

[32] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep Sparse Rectifier Neural Networks". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS)*. Vol. 15. Proceedings of Machine Learning Research. PMLR, 2011, pp. 315–323.

[33] Aaron Grattafiori et al. *The Llama 3 Herd of Models*. 2024. arXiv: 2407.21783 [cs.AI]. URL: https://arxiv.org/abs/2407.21783.

[34] Suchin Gururangan et al. "DEMix Layers: Disentangling Domains for Modular Language Modeling". In: *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL)*. arXiv:2108.05036. 2022.

[35] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV]. URL: https://arxiv.org/abs/1512.03385.

[36] Pengcheng He et al. *DeBERTa: Decoding-enhanced BERT with Disentangled Attention*. 2021. arXiv: 2006.03654 [cs.CL]. URL: https://arxiv.org/abs/2006.03654.

[37] Jonathan Ho, Ajay Jain, and Pieter Abbeel. “Denoising diffusion probabilistic models”. In: *Advances in neural information processing systems* 33 (2020), pp. 6840–6851.

[38] Jonathan Ho et al. “Video diffusion models”. In: *ICLR workshop on deep generative models for highly structured data*. 2022.

[39] A. L. Hodgkin and A. F. Huxley. “A Quantitative Description of Membrane Current and Its Application to Conduction and Excitation in Nerve”. In: *The Journal of Physiology* 117.4 (1952), pp. 500–544. DOI: 10.1113/jphysiol.1952.sp004764.

[40] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. “Multilayer Feedforward Networks are Universal Approximators”. In: *Neural Networks* 2.5 (1989), pp. 359–366. DOI: 10.1016/0893-6080(89)90020-8.

[41] Jeremy Howard and Sebastian Ruder. “Universal language model fine-tuning for text classification”. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2018, pp. 328–339.

[42] Robert A. Jacobs et al. “Adaptive Mixtures of Local Experts”. In: *Neural Computation* 3.1 (1991), pp. 79–87. DOI: 10.1162/neco.1991.3.1.79.

[43] F. Jelinek et al. “Perplexity—a measure of the difficulty of speech recognition tasks”. In: *The Journal of the Acoustical Society of America* 62.S1 (Aug. 2005), S63–S63. ISSN: 0001-4966. DOI: 10.1121/1.2016299. eprint: https://pubs.aip.org/asa/jasa/article-pdf/62/S1/S63/11558910/s63_5_online.pdf. URL: https://doi.org/10.1121/1.2016299.

[44] Albert Q Jiang et al. “Mixtral of experts”. In: *arXiv preprint arXiv:2401.04088* (2024).

[45] Peng Jin et al. “MoH: Multi-Head Attention as Mixture-of-Head Attention”. In: *arXiv preprint arXiv:2410.11842* (2024).

[46] Min Jae Jung and JooHee Kim. “Pmoe: Progressive mixture of experts with asymmetric transformer for continual learning”. In: *arXiv preprint arXiv:2407.21571* (2024).

[47] Eric R. Kandel, James H. Schwartz, and Thomas M. Jessell. *Principles of Neural Science*. 4th. New York: McGraw-Hill, 2000.

[48] Jared Kaplan et al. “Scaling laws for neural language models”. In: *arXiv preprint arXiv:2001.08361* (2020).

[49] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[50] James Kirkpatrick et al. “Overcoming catastrophic forgetting in neural networks”. In: *Proceedings of the national academy of sciences* 114.13 (2017), pp. 3521–3526.

[51] Harsh Kohli, Sachin Kumar, and Huan Sun. *GroundCocoa: A Benchmark for Evaluating Compositional Conditional Reasoning in Language Models*. 2025. arXiv: 2404.04237 [cs.CL]. URL: https://arxiv.org/abs/2404.04237.

[52] Manikanta Kotthapalli, Deepika Ravipati, and Reshma Bhatia. “Yolov1 to yolov11: A comprehensive survey of real-time object detection innovations and challenges”. In: *arXiv preprint arXiv:2508.02067* (2025).

[53] Sneha Kudugunta et al. *Beyond Distillation: Task-level Mixture-of-Experts for Efficient Inference*. 2021. arXiv: 2110.03742 [cs.CL]. URL: https://arxiv.org/abs/2110.03742.

[54] Woosuk Kwon et al. *Efficient Memory Management for Large Language Model Serving with PagedAttention*. 2023. arXiv: 2309.06180 [cs.LG]. URL: https://arxiv.org/abs/2309.06180.

[55] Dmitry Lepikhin et al. *GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding*. 2020. arXiv: 2006.16668 [cs.CL]. URL: https://arxiv.org/abs/2006.16668.

[56] Hector J. Levesque, Ernest Davis, and Leora Morgenstern. “The Winograd schema challenge”. In: *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning*. KR’12. Rome, Italy: AAAI Press, 2012, pp. 552–561. ISBN: 9781577355601.

[57] Mike Lewis et al. *BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension*. 2019. arXiv: 1910.13461 [cs.CL]. URL: https://arxiv.org/abs/1910.13461.

[58] Hongbo Li et al. “Theory on mixture-of-experts in continual learning”. In: *International Conference on Learning Representations*. Vol. 2025. 2025, pp. 8169–8206.

[59] Margaret Li et al. “Branch-Train-Merge: Embarrassingly Parallel Training of Expert Language Models”. In: *arXiv preprint arXiv:2208.03306* (2022).

[60] Chin-Yew Lin. “ROUGE: A Package for Automatic Evaluation of Summaries”. In: *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, July 2004, pp. 74–81. URL: https://aclanthology.org/W04-1013/.

[61] Aixin Liu et al. “Deepseek-v3 technical report”. In: *arXiv preprint arXiv:2412.19437* (2024).

[62] Haotian Liu et al. “Visual instruction tuning”. In: *Advances in neural information processing systems* 36 (2023), pp. 34892–34916.

[63] Yixin Liu et al. “Sora: A review on background, technology, limitations, and opportunities of large vision models”. In: *arXiv preprint arXiv:2402.17177* (2024).

[64] Ze Liu et al. *Swin Transformer: Hierarchical Vision Transformer using Shifted Windows*. 2021. arXiv: 2103.14030 [cs.CV]. URL: https://arxiv.org/abs/2103.14030.

[65] Ilya Loshchilov and Frank Hutter. “Decoupled weight decay regularization”. In: *arXiv preprint arXiv:1711.05101* (2017).

[66] Yun Luo et al. *An Empirical Study of Catastrophic Forgetting in Large Language Models During Continual Fine-tuning*. 2025. arXiv: 2308.08747 [cs.CL]. URL: https://arxiv.org/abs/2308.08747.

[67] Neeratyoy Mallik et al. “When is Warmstarting Effective for Scaling Language Models?” In: *arXiv preprint arXiv:2605.13405* (2026).

[68] Warren S. McCulloch and Walter Pitts. “A Logical Calculus of the Ideas Immanent in Nervous Activity”. In: *Bulletin of Mathematical Biophysics* 5.4 (1943), pp. 115–133. DOI: 10.1007/BF02478259.

[69] Siyuan Mu and Sen Lin. *A Comprehensive Survey of Mixture-of-Experts: Algorithms, Theory, and Applications*. 2026. arXiv: 2503.07137 [cs.LG]. URL: https://arxiv.org/abs/2503.07137.

[70] Yurii Nesterov. “A method for solving the convex programming problem with convergence rate O(1/k^2)”. In: *Proceedings of the USSR Academy of Sciences* 269 (1983), pp. 543–547. URL: https://api.semanticscholar.org/CorpusID:145918791.

[71] Bowen Pan et al. *Dense Training, Sparse Inference: Rethinking Training of Mixture-of-Experts Language Models*. 2024. arXiv: 2404.05567 [cs.LG]. URL: https://arxiv.org/abs/2404.05567.

[72] Denis Paperno et al. *The LAMBADA dataset: Word prediction requiring a broad discourse context*. 2016. arXiv: 1606.06031 [cs.CL]. URL: https://arxiv.org/abs/1606.06031.

[73] David Patterson et al. “Carbon emissions and large neural network training”. In: *arXiv preprint arXiv:2104.10350* (2021).

[74] Krishna Pillutla et al. *MAUVE: Measuring the Gap Between Neural Text and Human Text using Divergence Frontiers*. 2021. arXiv: 2102.01454 [cs.CL]. URL: https://arxiv.org/abs/2102.01454.

[75] B.T. Polyak. “Some methods of speeding up the convergence of iteration methods”. In: *USSR Computational Mathematics and Mathematical Physics* 4.5 (1964), pp. 1–17. ISSN: 0041-5553. DOI: https://doi.org/10.1016/0041-5553(64)90137-5. URL: https://www.sciencedirect.com/science/article/pii/0041555364901375.

[76] Federico Putamorsi et al. “SPARK: Input-Conditioned Sparse Activation Modulation for Frozen DiT-based Super-Resolution”. In: *arXiv preprint arXiv:2609.03813* (2026).

[77] Alec Radford et al. “Language Models are Unsupervised Multitask Learners”. In: *OpenAI* (2019). Accessed: 2024-11-15. URL: https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

[78] Colin Raffel et al. *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*. 2023. arXiv: 1910.10683 [cs.LG]. URL: https://arxiv.org/abs/1910.10683.

[79] Samyam Rajbhandari et al. *DeepSpeed-MoE: Advancing Mixture-of-Experts Inference and Training to Power Next-Generation AI Scale*. 2022. arXiv: 2201.05596 [cs.LG]. URL: https://arxiv.org/abs/2201.05596.

[80] Pranav Rajpurkar et al. *SQuAD: 100,000+ Questions for Machine Comprehension of Text*. 2016. arXiv: 1606.05250 [cs.CL]. URL: https://arxiv.org/abs/1606.05250.

[81] Aditya Ramesh et al. "Hierarchical text-conditional image generation with clip latents". In: *arXiv preprint arXiv:2204.06125* 1.2 (2022), p. 3.

[82] Joseph Redmon et al. "You only look once: Unified, real-time object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788.

[83] Stephen Roller et al. *Hash Layers For Large Sparse Models*. 2021. arXiv: 2106.04426 [cs.LG]. URL: https://arxiv.org/abs/2106.04426.

[84] Robin Rombach et al. "High-resolution image synthesis with latent diffusion models". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2022, pp. 10684–10695.

[85] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65 6 (1958), pp. 386–408. URL: https://api.semanticscholar.org/CorpusID:12781225.

[86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning Representations by Back-Propagating Errors". In: *Nature* 323.6088 (1986), pp. 533–536. DOI: 10.1038/323533a0.

[87] Chitwan Saharia et al. "Photorealistic text-to-image diffusion models with deep language understanding". In: *Advances in neural information processing systems* 35 (2022), pp. 36479–36494.

[88] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. *Self-Attention with Relative Position Representations*. 2018. arXiv: 1803.02155 [cs.CL]. URL: https://arxiv.org/abs/1803.02155.

[89] Noam Shazeer. *Fast Transformer Decoding: One Write-Head is All You Need*. 2019. arXiv: 1911.02150 [cs.NE]. URL: https://arxiv.org/abs/1911.02150.

[90] Noam Shazeer. *GLU Variants Improve Transformer*. 2020. arXiv: 2002.05202 [cs.LG]. URL: https://arxiv.org/abs/2002.05202.

[91] Noam Shazeer et al. *Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer*. 2017. arXiv: 1701.06538 [cs.LG]. URL: https://arxiv.org/abs/1701.06538.

[92] Haizhou Shi et al. "Continual Learning of Large Language Models: A Comprehensive Survey". In: *ACM Comput. Surv.* 58.5 (Nov. 2025). ISSN: 0360-0300. DOI: 10.1145/3735633. URL: https://doi.org/10.1145/3735633.

[93] Uriel Singer et al. "Make-a-video: Text-to-video generation without text-video data". In: *arXiv preprint arXiv:2209.14792* (2022).

[94] Shivalika Singh et al. *Global MMLU: Understanding and Addressing Cultural and Linguistic Biases in Multilingual Evaluation*. 2025. arXiv: 2412.03304 [cs.CL]. URL: https://arxiv.org/abs/2412.03304.

[95] Richard Socher et al. "Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank". In: *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. Ed. by David Yarowsky et al. Seattle, Washington, USA: Association for Computational Linguistics, Oct. 2013, pp. 1631–1642. URL: https://aclanthology.org/D13-1170/.

[96] Emma Strubell, Ananya Ganesh, and Andrew McCallum. "Energy and policy considerations for modern deep learning research". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 34. 09. 2020, pp. 13693–13696.

[97] Jianlin Su et al. *RoFormer: Enhanced Transformer with Rotary Position Embedding*. 2023. arXiv: 2104.09864 [cs.CL]. URL: https://arxiv.org/abs/2104.09864.

[98] Sainbayar Sukhbaatar et al. *Branch-Train-MiX: Mixing Expert LLMs into a Mixture-of-Experts LLM*. 2024. arXiv: 2403.07816 [cs.CL]. URL: https://arxiv.org/abs/2403.07816.

[99] Ilya Sutskever et al. “On the importance of initialization and momentum in deep learning”. In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, July 2013, pp. 1139–1147. URL: https://proceedings.mlr.press/v28/sutskever13.html.

[100] T. Tieleman and G. Hinton. “Lecture 6.5-rmsprop: Divide the Gradient by a Running Average of Its Recent Magnitude”. In: *Coursera: Neural Networks for Machine Learning* 4 (2012), pp. 26–31.

[101] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. 2023. arXiv: 2307.09288 [cs.CL]. URL: https://arxiv.org/abs/2307.09288.

[102] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. 2023. arXiv: 2302.13971 [cs.CL]. URL: https://arxiv.org/abs/2302.13971.

[103] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[104] Elena Voita et al. “Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned”. In: *Proceedings of the 57th annual meeting of the association for computational linguistics*. 2019, pp. 5797–5808.

[105] Alex Wang et al. “GLUE: A multi-task benchmark and analysis platform for natural language understanding”. In: *arXiv preprint arXiv:1804.07461* (2018).

[106] Zirui Wang et al. *SimVLM: Simple Visual Language Model Pretraining with Weak Supervision*. 2022. arXiv: 2108.10904 [cs.CV]. URL: https://arxiv.org/abs/2108.10904.

[107] Alex Warstadt, Amanpreet Singh, and Samuel R. Bowman. *Neural Network Acceptability Judgments*. 2019. arXiv: 1805.12471 [cs.CL]. URL: https://arxiv.org/abs/1805.12471.

[108] Foundation Wikimedia. *Wikimedia Downloads*. 2023. URL: https://dumps.wikimedia.org.

[109] Adina Williams, Nikita Nangia, and Samuel R. Bowman. “A Broad-Coverage Challenge Corpus for Sentence Understanding through Inference”. In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. Ed. by Marilyn Walker, Heng Ji, and Amanda Stent. New Orleans, Louisiana: Association for Computational Linguistics, June 2018, pp. 1112–1122. DOI: 10.18653/v1/N18-1101. URL: https://aclanthology.org/N18-1101/.

[110] Thomas Wolf et al. “Transformers: State-of-the-Art Natural Language Processing”. In: Association for Computational Linguistics, Oct. 2020, pp. 38–45. URL: https://aclanthology.org/2020.emnlp-demos.6/.

[111] Hang Yan et al. *TENER: Adapting Transformer Encoder for Named Entity Recognition*. 2019. arXiv: 1911.04474 [cs.CL]. URL: https://arxiv.org/abs/1911.04474.

[112] An Yang et al. *Qwen2 Technical Report*. 2024. arXiv: 2407.10671 [cs.CL]. URL: https://arxiv.org/abs/2407.10671.

[113] An Yang et al. “Qwen3 technical report”. In: *arXiv preprint arXiv:2505.09388* (2025).

[114] Jason Yosinski et al. “How transferable are features in deep neural networks?” In: *Advances in neural information processing systems* 27 (2014).

[115] Chulhee Yun et al. *Are Transformers universal approximators of sequence-to-sequence functions?* 2020. arXiv: 1912.10077 [cs.LG]. URL: https://arxiv.org/abs/1912.10077.

[116] Biao Zhang and Rico Sennrich. *Root Mean Square Layer Normalization*. 2019. arXiv: 1910.07467 [cs.LG]. URL: https://arxiv.org/abs/1910.07467.

[117] Xiaofeng Zhang et al. *Mixture of Attention Heads: Selecting Attention Heads Per Token*. 2022. arXiv: 2210.05144 [cs.CL]. URL: https://arxiv.org/abs/2210.05144.

[118] Zhengyan Zhang et al. *Emergent Modularity in Pre-trained Transformers*. 2023. arXiv: 2305.18390 [cs.CL]. URL: https://arxiv.org/abs/2305.18390.

[119] Zhengyan Zhang et al. *MoEfication: Transformer Feed-forward Layers are Mixtures of Experts*. 2022. arXiv: 2110.01786 [cs.CL]. URL: https://arxiv.org/abs/2110.01786.

[120] Barret Zoph et al. *ST-MoE: Designing Stable and Transferable Sparse Expert Models*. 2022. arXiv: 2202.08906 [cs.CL]. URL: https://arxiv.org/abs/2202.08906.