Deep Learning, Transformers and Graph Neural Networks: a Linear Algebra Perspective

Abdelkader Baggag[1] and Yousef Saad[2]

[1]Qatar Computing Research Institute, Hamad Bin Khalifa University, HBKU Research Complex, Doha, 34110, Qatar.
[2]Dept. Computer Science and Engineering, University of Minnesota, 200 Union Street S.E., City, 55455, MN, USA.

Contributing authors: abaggag@hbku.edu.qa; saad@umn.edu;

Abstract

In an era where Artificial Intelligence is permeating virtuallly every single field of science and engineering, it is becoming critical for Numerical Linear Algebra specialists to start exploring the main ingredients of deep learning and to find ways to contribute to its advancement. What is fascinating is that Numerical Linear Algebra (NLA) is at the core of Machine Learning (ML) and Artificial Intelligence (AI). All AI methods rely essentially on four ingredients: data, optimization methods, statistical intuition, and linear algebra. The very first step of any neural network model is to convert the problem into one that can be exploited by numerical methods, employing optimization techniques. The task of this first step is to map words or sentences into tokens which are then embedded into Euclidean spaces. From there on, the models refer to vectors and matrices. The goal of this article is to describe the main ingredients of deep learning methods with an emphasis of a linear algebra viewpoint. The article will describe deep neural networks, the idea of multilayer perceptrons, and the notion of 'attention' which is a key ingredient in large language models but also in other machine learning applications. A big part of the discussion will be devoted to methods that exploit graphs in neural networks, e.g., Graph Neural Networks (GNNs). The paper concludes with remarks on the future of numerical linear algebra in the era of AI.

1 Introduction

The progress that is predicted in Artificial Intelligence for the coming decade is truly staggering. The article [1] summarizes the evolution of AI in recent years (from GPT-2 to GPT-4) and concludes that "Artificial General Intelligence (AGI) by 2027 is strikingly plausible". The author elaborates by predicting that *"by 2027, models will be able to do the work of an AI researcher/engineer"*. The narrative laid out in the article is rather compelling and seems to be corroborated by other authors, see, e.g., [2] a book authored by a co-founder of the pioneering AI company DeepMind. The rapid advancement in AI has caught the world off-guard, prompting researchers, corporations, and governments to a fierce competition for super-intelligence. The same article [1] also highlights the challenges on the horizon, such as issues related to available data (the "data wall") and states that one of the key components to progress is new algorithms. *"... current architectures and training algorithms are still very rudimentary, and it seems that much more efficient schemes should be possible."* According to the author, what has been accomplished in algorithms so far has been to "pick the many obvious low-hanging fruits". This provides a unique opportunity for researchers in Numerical Linear Algebra (NLA) to participate in the progress of AI.

We note that the NLA community as a whole has been rather slow in embracing AI research. This is rather surprising when considering that *almost all the important recent contributions to improved AI algorithms are rooted in simple NLA ideas.* For example the `LoRA` work [3] has had a huge impact because it enables practitioners to cut down training (and inference) costs by orders of magnitude without sacrificing accuracy - but the idea exploits a simple low-rank approximation. Similarly, the work on 'Linear Attention' [4] exploits a clever idea based on approximating a nonlinear mapping by a linear one, reducing the cost from $O(n^2)$ to $O(n)$. The key idea is again grounded in NLA. There are a number of similar techniques most of which were contributed by AI specialists. It is quite likely that big gains in performance may be achieved if we were able to bring more elaborate NLA techniques than those currently in use by specialists in the field.

The article will be mostly a survey of known methods in AI with the primary goal of unraveling the linear algebra aspects of deep learning. Our primary goal is not to provide a comprehensive coverage of the field which is very broad, but rather to focus on a few ideas that played a major role and that shaped AI in recent years. Thus, after covering deep neural networks and multilayer perceptrons, we will describe the idea of 'attention' which is a key ingredient in large language models but also in other machine learning applications. In addition, a big part of our discussion will be devoted to methods that exploit graphs in neural networks, e.g., Graph Neural Networks (GNNs). Finally, a secondary goal of the paper is to make the point that the numerical linear algebra community will find numerous opportunities for research in AI and that it should increase its exposure to this emerging field.

2 Historical perspective

The idea of thinking machines began alongside that of modern computing toward the middle of the 20th century. In 1950 Turing [5] designed a test (now known as the

'Turing test') to answer the question as to whether or not machines can think. Since then, the field of Artificial Intelligence (AI), has seen several periods of advancement culminating with the idea of Transformers and Large Language Models and generative AI, such as Chat-GPT, and LLama3, among others. There was a significant surge of excitement surrounding AI research during the 1970s and 1980s but it became evident that the promises made in the field were too ambitious for the time, leading to a subsequent negative reaction and a decline in interest. AI resurfaced with force beginning around 2010. A big factor for this come-back was big improvements in computing power. Then AI began an astoundingly fast ascent around 2016 ending with systems that can solve typical university-level homeworks and disrupting every aspect of science and engineering.

2.1 Major Milestones

The first major idea in AI was of neural networks and the invention of the 'perceptron' by Rosenblatt in 1958 [6]. Getting insight from human brains, Rosenblatt thought of a simple system that could perform classification tasks by building a function that imitated neurons. Thereafter, much of AI research took the viewpoint of symbolic reasoning, and natural language processing. Incidentally the term 'Artificial Intelligence' was coined in 1956 by John McCarthy at the Dartmouth College Artificial Intelligence Conference [7]. Among related developments, one can mention the invention of the LISP language from 1958 which emphasized symbolic processing capabilities. Then came the so-called AI Winter (1970s) - when AI researchers realized that their progress was stalling. Funding and interest in AI research fell significantly. The main reason that is often cited for this lack of progress was the limitations of the AI approaches of that time, such as symbolic reasoning.

Readers in the field of Numerical Analysis will certainly be interested to learn that it is the departure from symbolic reasoning and the adoption of numerical approaches that saved AI. Indeed the next big milestone was the expansion of the Perceptron idea to Multilayer Neural Networks and the exploitation of *Backpropagation* techniques around the mid 1980s. The development of the backpropagation algorithm in [8] gave a huge boost to AI ideas of the time as it allowed multi-layer neural networks to be trained effectively. This revolutionized neural networks and provided the foundation for practical deep learning. In spite of this major breakthrough AI still fell short of fulfilling the promises of its early days, due to the limited computational resources of the time.

The year 1997 saw the first major demonstration of AI systems: IBM's Deep Blue system defeated world chess champion Garry Kasparov in a six-game match. This was achieved by a brute-force approach, evaluating millions of chess positions per second. This event was a landmark moment in AI, showing that intensive search algorithms using pure computational power can defeat a human expert. However, Deep Blue did not exploit standard AI ideas of the time.

One had to wait until the period of the Mid 1990s to the decade of the 2000s to witness the real rise of machine learning and data-driven AI. The main ingredient behind this surge was the dramatic improvement of computational power and availability of large datasets. The field of AI took a definite turn away from rule-based systems (symbolic reasoning AI) and into data-driven approaches like machine learning and statistical modeling, i.e., more numerical approaches. One must realize that in this period machine learning was not limited to Neural Networks. In fact a common term used at the time was 'data mining', which essentially included any method that helps extract information from data. These methods were in the form of *unsupervised, supervised,* or *semi-supervised* learning. Among the powerful supervised learning techniques that were developed were classification methods such as Support Vector Machines (SVM) [9], or the Linear Discriminant Analysis, e.g., [10]. Fascinating methods were also discovered in unsupervised learning. These include the techniques of Eigenmaps [11], and Locally Linear Embedding [12] which are 'embedding' techniques that map an item in high dimensional space into a low dimensional one. Simplified versions of embeddings will later become part Deep Neural Network models. They are also the backbone of graph-based approaches such as Graph Convolutional Networks (GCN) and Graph Attention Networks (GAT), see Section 5.1.

Another breakthrough took place in 2012 where Convolutional Neural Networks (CNN) appeared in force. In 2012 a CNN called AlexNet [13] won a yearly competition of image recognition called ImageNet by a significant margin, demonstrating the potential of deep learning for image recognition tasks. What took researchers by surprise was the margin of superiority compared to anything that was achieved up to that date. In a short time AI systems surpassed humans at the task of classifying images. This development ushered a resurgence in deep learning research and applications, leading to advances in speech recognition, image processing, and even natural language understanding.

Then in 2016, following the steps taken by Deep-Blue almost a decade earlier, came another powerful demonstration of machine versus man when AlphaGo, a program developed by DeepMind, defeated Lee Sedol a champion of the game Go. Go is a game considered to be much more complicated than chess due to its huge number of possible moves. This fascinating episode of AI is narrated in detail in [2] a book co-authored by one of the founders of DeepMind.

AI as we know it today took-off with the advent of generative AI and Large Language models. Many developments led to LLMs. In summary, these involved modifications of neural networks that tried to process natural languages. If you give an unfinished sentence to the system, what are the best words that can be generated to complete the sentence? The answer is obtained by training a large model using a big data base. A major technique that was developed was Recurrent Neural Networks [14] but it was soon discovered that the learning process for RNNs often led to numerical difficulties. Corrective techniques were invented including Long Short Term Memory (LSTM), e.g., [15], or Gate Recurrent Units [16].

Then came another breakthrough when a paper with the title "Attention is all you need" [17] suggested essentially that all of these were not needed. This led to GPT-1 by OpenAI in 2018 followed by GPT-2 (2019) and GPT-3 (2020). GPT stands for Generative Pretrained Transformer and the main idea used is that of transformers exploiting two basic building blocks: the multi-layer perceptron idea of earlier times and attention. When it was released in 2020 GPT-3 was one of the largest language models capable of understanding and generating human-like text. It was seen as a

Model	GPT-1	GPT-2	GPT-3	GPT-4
Year	2018	2019	2020	2023
# Params	177M	1.5B	175B	1.7 T

Table 1 Evolution of the number of parameters in GPT-x models

decisive advance, that marked the dawn of a new era. The training of Chat-GPT3 was a tour-de-force, requiring a massive corpus of text and using 175 billion parameters.

Nowadays experts speak of General Artificial Intelligence (AGI), or the ability of an AI system to perform general tasks done by humans, e.g., the work of an engineer. Currently, computer software companies are already starting to reduce the number of programmers they employ because they can rely on AI instead.

2.2 A comparison with Moore's law

It is interesting to draw a comparison of the remarkable progress in AI with progress made in another field over the past few decades: chip-making. In this context, Moore's Law [18] which was stipulated in 1965 predicted the exponential progress of chip manufacturing for the following decades. The prediction was fairly accurate until recent years when device manufacturing started to confront the limits imposed by physics at the nanoscale. It is rather intuitive that any major discovery should follow a progression that exhibits an exponential behavior in its early stages. Therefore we can ask the question: Is there a sort of new Moore's law for AI? Here we could compare hardware but it is more relevant to compare number of parameters.

Moore's Law stated that the number of transistors that can be placed on a chip will double every two years. In Artificial Intelligence we can look at the number of parameters in large language models (LLMs) to see if there are any such behavior that can be discerned. Using only open-AI's GPT models for consistence, we can put the number of parameters in a table as shown in Table 1 Based on these 4 points, we can estimate the doubling time to 0.425 years (or 5.1 months) and the time to a 10-fold increase to 1.41 years (or 17 months). It is unlikely that this rate will be sustained and the primary reason for this is the 'data-wall', or the foreseen limitation of available data that can be exploited by LLMs. Since the optimal number of parameters used in a model is tied to the amount of data, according to a number of 'neural network scaling laws' [19, 20], it is clear that data is the bottleneck that limits the growth of the number of parameters and we are seeing this today.

While computer hardware manufacturing is highly competitive and protected, AI is characterized by a global openness. This environment is a blessing for research/science but it is also a curse as it makes *containment* challenging. However, there are now indications that the current success of AI will lead it to become less open in the future [1].

3 Deep Neural Networks

Neural Networks were initially developed with the help of an analogy with the human brain: some picture is seen (input), and analyzed to enable a comparison with stored images that have been memorized. In a machine context, we can view the process from the angle of finding a function ϕ through 'training', or optimizing ϕ on known data. Once the function is found, i.e., once the model is trained, then 'inference' becomes a matter of evaluating the function for some new input and deciding on an outcome based on the result. This, roughly speaking, is common to all neural network methods.

Two ideas that originated from numerical analysis played a major role in DNNs: The universal representation theorem proved by George Cybenko (1989) [21], and the idea of backward differentiation, which led to Backpropagation see e.g., Andreas Griewank [22–24]. These will be discussed briefly in the next few sections.

3.1 Multi-Layer Perceptrons

The problem of classification can be viewed from the angle of approximating a function which maps a data item to a certain label. In regression the output of a function can be any value in $\mathbb{R}$ or vector in $\mathbb{R}^k$ and the objective is to find one such function that transforms sample inputs to outputs. For example, we could be interested in finding the best function to predict an asset price (a value in $\mathbb{R}$) given a few commodity prices (a vector in $\mathbb{R}^d$). In classification the functions are allowed to have a finite number of values usually referred to as 'labels'. An example would be to predict if a message is Spam or non-Spam given a few of its key-words. Multilayer Perceptrons (MLPs) express these functions in the form of composition of functions that combine linear and nonlinear mappings. MLPs are key components of Transformers which are at the core of of LLMs.

Suppose we have a set of sample images of digits and that we know the labels of these samples, a number between 0 and 9. The problem is to use the given dataset, say n pictures along with their correct labels, to build a function which will identify the label of an arbitrary new image (not in the sample). This function will take an array of pixels x and produce the label via a function $\phi(x)$, a digit between 0 and 9 or a vector of length 10, giving the probabilities that the label of x is $0, 1, \cdots, 9$.

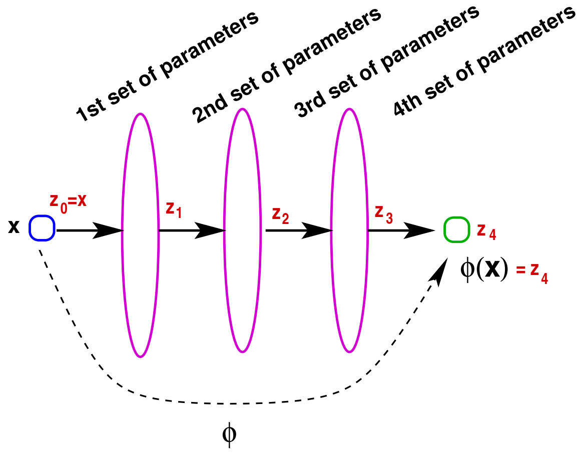

Fig. 1 A simple neural network with three hidden layers

The unknown function ϕ is defined through a number of stages corresponding to layers in the network. In this section, we consider the simplest case of Multi-Layer Perceptron, depicted in Figure 1. The network in Figure 1 has an input layer (leftmost, where x enters), 3 hidden layers (elongated ellipses) and output layer (rightmost). The input on the left side is typically a vector of length d, whose components are often called 'features'. It is common to represent feature vectors as row-vectors, i.e., $x \in \mathbb{R}^{1\times d}$, and they form the rows of a sample matrix that is processed in block, through the layers, as is discussed later. For convenience we will assume for now that x is a regular d-dimensional vector, i.e., $x \in \mathbb{R}^d$.

To separate two given sets of input data, we could use the hyperplane delimited by the linear mapping $\phi(x) = w^T x + \beta$. The two sets are defined by the sign of $\phi(x)$: positive sign for one set and nonnegative for the other. In other words:

$$\phi(x) = \sigma(w^T x + \beta) \tag{3.1}$$

where here σ is the sign function, will give the label of each set as $+1$ or -1. Thus, if we had a number of training data points (x_i, y_i) with binary labels (e.g., 'spam'–'non-spam', 'malignant' –'non-malignant',...) where $y_i = \pm 1$, we could use this set to determine an optimal w for which $\phi(x_i) \approx y_i$ for $i = 1, \cdots, n$.

The functions used in neural networks are generalizations of the above function. Instead of a single vector w we will use a $d \times k$ matrix W and σ is replaced by a continuous function known as an *'activation function'.* The result is that $\phi(x)$ is a vector. In addition, the final function ϕ is actually a composition of these elementary functions, associated with the different layers.

Going through the first layer, x is transformed into $W_1^T x + b_1$ where $W_1 \in \mathbb{R}^{d\times d_1}$ is some unknown matrix of weights to be determined and $b_1 \in \mathbb{R}^{d_1}$ is a bias, also to be determined. This first linear transformation is then compounded with the *activation function* usually denoted by σ, so the output of the first layer, denoted by z_1 is

$$z_1 = \sigma(W_1^T x + b_1). \tag{3.2}$$

There are a number of choices for the activation function σ, but the most common is the Rectified Linear Unit, or ReLU:

$$\sigma(t) = \max\{0, t\}. \tag{3.3}$$

Two other common activation functions are the sigmoid $\sigma(t) = (1 + e^{-t})^{-1}$ and the hyperbolic tangent $\sigma(t) = \tanh(t) = (e^t - e^{-t})/(e^t + e^{-t})$. Note that the value of ReLU is nonnegative, while those for the sigmoid and tanh lie in $(0, 1)$ and $(-1, 1)$ respectively. The second layer takes the output z_1 and applies to it a transformation similar to the first one: $z_2 = \sigma(W_2^T z_1 + b_2)$. Generally, going from layer $l - 1$ to layer l we have

$$z_l = \sigma(W_l^T z_{l-1} + b_l), \tag{3.4}$$

where $W_l \in \mathbb{R}^{d_{l-1}\times d_l}, b_l \in \mathbb{R}^{d_l}$, and σ. Assuming there are L hidden layers ($L = 3$ in Figure 1), then counting the input and output layers, we have $L + 2$ layers altogether.

The parameters W_l, b_l involve a mapping from data in layer $l-1$ to data in layer l and we have $L+1$ of these, starting with W_1, b_1 and ending with W_{L+1}, b_{L+1}. The last vector to be computed is z_{L+1} (e.g., z_4 in Figure 1) and $\phi(x)$ is set to this output. Thus, for the example in Figure 1,

$$\phi(x) = \sigma(W_4^T\sigma(W_3^T\sigma(W_2^T\sigma(W_1^Tx + b_1) + b_2) + b_3) + b_4). \tag{3.5}$$

This function is better expressed in algorithmic form, as shown in Algorithm 1. (Note that the algorithm does not solve a problem yet - it just defines the function ϕ, given the parameters).

Algorithm 1 Forward Propagation

```
1: Input: $x \in \mathbb{R}^d$, Output: $y \in \mathbb{R}^C$
2: Set: $z_0 = x$
3: for $l = 1$ : L+1 do
4:     $z_l = \sigma(W_l^T z_{l-1} + b_l)$
5: end for
6: Set: $\phi(x) := z_{L+1}$
```

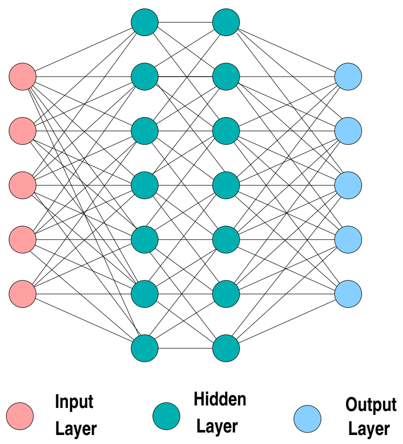

Fig. 2 A simple neural network with 2 hidden layers

The problem is to find a function ϕ thus defined through these parameters in such a way that given some data, i.e., a set of x_i's for which the exact outcome y_i is known (training data), the value of $\phi(x_i)$ is closest to y_i according to some measure.

In the case of digit recognition, we would have a set $x_1, \cdots, x_n$ of images of digits for which the exact digit y_i is known. Each image is an array $m_1 \times m_2$ of pixels which is vectorized into a vector of length $d_0 \equiv m_1 m_2$. Each of these features x_i will have a value of y_i between 0 and 9. It is convenient to recast the digit y_i into a so-called *one-hot* vector which is a vector of 10 entries that are zero except for the one corresponding to the digit y_i which is set to one. Thus, if the digit is 2, the vector y_i will be the canonical vector e_3 in $\mathbb{R}^{10}$, i.e., the 3rd column of the 10×10 identity matrix. Suppose that $m_1 = m_2 = 20$ and that we have 2 hidden layers with $d_1 = 100, d_2 = 100$. Then the input data has size $n \times d_0$ where $d_0 = 400$ and the output will be of size $n \times 10$.

The problem is to find a function ϕ (i.e., matrices W_l) such that $\phi(x) \approx y$ for each if the data pairs (x_i, y_i) in the 'training set'. The question that arises is whether or not it is possible to approximate an unknown function ϕ by a composition of functions of the form (3.1). The 'universal representation theorem' shown by George Cybenko in 1989 [21], answered this answer and provided a major theoretical foundation for neural networks.

3.2 Loss function and training the MLP

Training the model requires a set of data points $x_i, y_i, i = 1 : n$. The input can therefore be set as a matrix X of size $n \times d_0$, in which each row corresponding to a sample. The output can be a vector Y of length n, whose entries are the (known) labels of the samples. In classification, it is also common practice to replace the label by a *one-hot row vector* of length C, where C is the number of classes, as was described earlier. In this situation Y is $n \times C$. Thus, we seek a function ϕ such that $\phi(x_i) \approx y_i$ for $i = 1 : n$ - or, in matrix form $\phi(X) \approx Y$. Using matrix notation again, each of the internal variables z_l defined earlier becomes a matrix Z_l of size $n \times d_l$ and the transformation (3.4) becomes:

$$Z_l = \sigma(Z_{l-1} \times W_l + b_l) \tag{3.6}$$

where W_l $\mathbb{R}^{d_{l-1} \times d_l}, b_l \in \mathbb{R}^{1 \times d_l}$, and σ are the same as before. Note the change of notation where the samples x_i and internal variables z_i seen in Section 3.1 are now row vectors that occupy the rows of the matrix X and Z_l respectively. Here we also need to shed light on a feature of notation that is common in this context. The product $Z_{l-1} \times W_l$ in (3.6) is a matrix of size $n \times d_l$ and when the row vector b_l is added to it, it is meant that it is added to each of its rows[1].

For the given training data $X \in \mathbb{R}^{n \times d_0}$ and corresponding ground truth $Y \in \mathbb{R}^{n \times d_{L+1}}$, we need to express an optimization problem which will consist of finding an optimal set of parameters W, such that $\phi_W(X) \approx Y$. Here by W we mean the set of parameters $W_1, \cdots, W_{L+1}$ along with the biases $b_1, \cdots, b_{L+1}$. The output of the network is the matrix $Z_{L+1} = \phi_W(X) \in \mathbb{R}^{n \times C}$. We added W as a subscript to the function ϕ to emphasize its dependence on these parameters. A better notation might be $\phi(X \mid W)$ which is to be read as "the result of ϕ for data set X, given the parameter set W." When the input data X is fixed as is usually the case for training, then we can just write $\phi(W)$. When W is fixed as is the case when testing, i.e., during the inference phase, we evaluate $\phi_W(x)$, which can be written as $\phi(x)$ without ambiguity, for some data item x.

[1]This is called *broadcasting*, a term borrowed from the Python language. Since Python is heavily used in AI, its syntax and terminology has permeated the notation used in the field. Incidentally, in R, a popular language in computational statistics, the equivalent term to broadcasting is *vector recycling* or *recycling rule*. MATLAB, started supporting broadcasting with version R2016b, and it calls this operation *implicit expansion*.

One possible formulation for an objective function to minimize could be:

$$\min_W \mathcal{L}(W) \equiv \|Y - \phi_W(X)\|_F^2 = \sum_{i=1}^{n} \|y_i - \phi_W(x_i)\|_2^2 \tag{3.7}$$

where $\|\cdot\|_F$ represents the Frobenius norm. Recall that Y and $\phi_W(X)$ are of size $n \times d_{L+1}$ where $d_{L+1} \equiv C$ is the number of classes. This simple formulation works fine but it is seldom employed.

A preferred approach is to exploit a cross-entropy distance - a notion based in maximum likelihood estimation. The output of the model $\phi_W(x_i)$ is often denoted by $\hat{y}_i$. When the output and the y_i's are scalars then the cross-entropy loss is simply defined by

$$\mathcal{L}(W) = -\frac{1}{n}\sum_{i=1}^{n} y_i \log \hat{y}_i \tag{3.8}$$

In classification problems with C classes, the y_i's are often one-hot vectors of length C, and the outputs are processed by the *'softmax'* function, which is defined as follows for a given row vector z:

$$\texttt{softmax}(z) = \frac{\exp(z)}{\mathrm{sum}[\exp(z)]}. \tag{3.9}$$

The exponential is applied componentwise and 'sum$[v]$' is the sum of components of v. With this we define the vector $\hat{y}_i$:

$$\hat{y}_i = \texttt{softmax}(\phi(x_i)), \tag{3.10}$$

and the new output matrix $\hat{Y}$ whose rows are the $\hat{y}_i$'s[2]. The end result is that each row-sum of $\hat{Y}$ is equal to one and each entry is nonnegative. The advantage of this simple transformation is that the entries can now be interpreted as probabilities. Finally, the product $y_i \log \hat{y}_i$ in the scalar case, equation (3.8), is replaced by the inner product $(y_i, \log \hat{y}_i)$. Therefore, in the vector case, we want to minimize the following cross-entropy function:

$$\mathcal{L}(W) = -\frac{1}{n}\sum_{i=1}^{n} (y_i, \log \hat{y}_i) \ . \tag{3.11}$$

The above expression comes from statistical arguments. Since the $\hat{y}_i$ are to be treated as probabilities that depend on W, the categorical distribution given W is the inner product $\mathrm{Cat}(y_i|\hat{y}_i) = (y_i, \hat{y}_i)$ which leads to the likelihood function $\prod_{i=1}^{C}(y_i, \hat{y}_i(W))$. Taking the average negative log yields the *negative log likelihood* function to be minimized:

$$\mathcal{L}(W) = -\frac{1}{n}\sum_{i=1}^{n} \log(y_i, \hat{y}_i) = -\frac{1}{n}\sum_{i=1}^{n} (y_i, \log(\hat{y}_i)), \tag{3.12}$$

[2]A common variation to the softmax function is to replace the variable z in the exponentials of Equation (3.9) by z/T where T is known as the *temperature*.

where the second equality in the above equation comes from the fact that each vector y_i has only one nonzero component, e.g., the c-th component $y_i^{(c)} = 1$, and therefore $\log(y_i, \log(\hat{y}_i)) = y_i^{(c)} \log(\hat{y}_i^{(c)})$ from which the result follows.

3.3 Optimization in DNNs

In spite of its simplicity, the *Stochastic Gradient Descent* (SGD) algorithm is a good representative of iterative optimization algorithms in DL. This is because its use is rather widespread and it shares the same features as those of the more advanced algorithms. The classical (deterministic) gradient descent (GD) method for minimizing a convex and differentiable function $f(w)$ with respect to w, consists of taking the iterates:

$$w_{j+1} = w_j - \eta_j \nabla f(w_j), \tag{3.13}$$

where η_j is a scalar termed the *step-size* or *learning rate* in machine learning. Gradient descent is well understood for functions f that are convex. In this situation, the step-size η_j is usually determined by performing a line search, i.e., by selecting the scalar η so as to minimize the cost function $f(w_j - \eta \nabla f(w_j))$ with respect to η. In data-related applications w is a vector of weights needed to optimize a process. In deep learning, $f(w)$ is often the sum, or the mean, of a large number of other cost functions, i.e., we often have

$$f(w) = \frac{1}{n} \sum_{i=1}^{n} f_i(w) \qquad \to \qquad \nabla f(w) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(w). \tag{3.14}$$

The related optimization problem is known as *the finite sum problem* [25]. The index i here refers to the samples in the training data. In the simplest case where a 'Mean Squared Error' (MSE) cost function is employed, we would have $f(w) = \frac{1}{n} \sum_{i=1}^{n} \|y_i - \phi_w(x_i)\|_2^2$ where y_i is the target value for x_i and $\phi_w(x_i)$ is the prediction of the model for x_i given the weights represented by w.

Since it is usually expensive to compute the 'full' gradient ∇f but inexpensive to compute a component $\nabla f_i(w)$, Stochastic Gradient Descent (SGD) methods replace the gradient $\nabla f(w_j)$ in the gradient descent step (3.13) by $\nabla f_k(w_j)$ where k is an index between 1 and n drawn at random. The result is an iteration of the type:

$$w_{j+1} = w_j - \eta_j \nabla f_k(w_j), \tag{3.15}$$

where f_k is a function among $\{f_1, f_2, \cdots, f_n\}$ drawn at random at each step j. The parameter η_j, called the *learning rate* in this context, is rarely selected by a linesearch but determined adaptively or set to a constant. Convergence results for SGD have been established in the convex case [26–29]. The main point, going back to the pioneering work by Robbins and Monro [29], is that when gradients are inaccurately computed, then if they are selected from a process whose noise has a mean of zero then the process will converge in probability to the root.

The SGD algorithm goes back to 1951 with the seminal article by Robbins and Munro [29] which considered the general problem of finding the root of the equation

$M(w) = \alpha$ where $M(w)$ is not available to the 'experimenter' but can be measured, or sampled, via a random variable $H(w)$ that satisfies $E[H(w)] = M(w)$. In this case the deterministic scheme $w_{n+1} = w_n - a_n M(w_n)$ is not feasible but can be replaced by an iteration of the form $w_{n+1} = w_n - a_n g_n$ in which $g_n = H(w_n)$. The article showed convergence results when the following conditions are satisfied:

$$\sum_{i=1}^{\infty} a_i = \infty, \quad \sum_{i=1}^{\infty} a_i^2 < \infty. \tag{3.16}$$

Further analysis of the Robbins an Munro framework can be found in [27, 28, 30].

A straightforward SGD approach that uses a single function f_k at a time is seldom used in practice because this typically results in slow convergence. A common middle-ground alternative is to resort to *mini-batching*, where the idea consists of replacing the single function f_k, selected at random from $f_1, f_2, \cdots, f_n$ by an average of a few such functions - again drawn at random from the full set. Thus, the set $\{1, 2, \cdots, n\}$ is partitioned into n_B mutually disjoint 'mini-batches' $\mathcal{B}_j, j = 1, \cdots, n_B$ whose union is the set $\{1, 2, \cdots, n\}$. Here, each $\mathcal{B}_j$ is a small set of indices. Then instead of considering a single function f_i we will consider

$$f_{\mathcal{B}_j}(w) \equiv \frac{1}{|\mathcal{B}_j|} \sum_{k \in \mathcal{B}_j} f_k(w). \tag{3.17}$$

We will cycle through all mini-batches $\mathcal{B}_j$ at each time performing a group-gradient step of the form:

$$w_{j+1} = w_j - \eta_j \nabla f_{\mathcal{B}_j}(w_j) \quad j = 1, 2, \cdots, n_B. \tag{3.18}$$

If each set $\mathcal{B}_j$ is small enough, computing the gradient will be manageable and computationally efficient. One sweep through the whole set of functions as in (3.18) is termed an *'epoch'*. The number of iterations of SGD and other optimization algorithms in DL is often measured in terms of epochs.

Mini-batch processing in the random fashion described above is advantageous from a computational point of view since it typically leads to fewer sweeps through each function to achieve convergence. It is also mandatory if we wish to avoid reaching shallow local minima and overfitting. SGD approaches of this type are at the heart of optimization techniques in DL.

SGD is the simplest algorithm employed in Deep Learning among a class of well-established 'optimizers'. The most common optimization technique invoked to train a neural network is know as the Adaptive Moment Estimation (Adam) algorithm. Adam and other optimizers exploit two ideas: variance reduction and momentum. Variance reduction is borrowed from a technique known as AdaGrad [31] which scales the variables adaptively, adjusting the learning rate for each parameter individually in a model by considering the history of past gradients. The goal is to perform larger updates for 'infrequent' parameters and smaller updates for frequent ones. The second idea is that of momentum. Referring to the basic GD iteration (3.13), the principle of momentum is to add a multiple of the previous increment $w_j - w_{j-1}$ to the iterate:

$$w_{j+1} = w_j - \eta_j \nabla f(w_j) + \nu(w_j - w_{j-1}), \tag{3.19}$$

in essence forcing the iterate not to move too far away from its current trajectory.

Adam has two momentum terms one for the gradient and the other for variance. Both of these are attenuated exponentially. The algorithm is described below where g_t is the gradient at step t,and β_1 and β_2, are decay rates.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \ , \qquad \hat{m}_t = m_t/(1-\beta_1^t) \tag{3.20}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(g_t)^2 \ , \qquad \hat{v}_t = v_t/(1-\beta_2^t) \tag{3.21}$$

$$w_t = w_{t-1} - \frac{\eta \hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}. \tag{3.22}$$

Note that the shown vector divisions in the algorithm are performed component-wise. Similarly the term $(g_t)^2$ in (3.21) represents the vector of the squares of the components of g_t. Then, in (3.22) the components of the vector $\hat{m}_t$ are divided by those of the component-wise square root of $\hat{v}_t + \epsilon$. Here ϵ is a small scalar used to prevent divisions by zero or small numbers. The recommended parameters are $\beta_1 = 0.9,\ \beta_2 = 0.999,\ \epsilon = 10^{-8}$.

3.4 The challenges of DNNs

One of the major issues in optimizing DNNs is the lack of convexity of the objective functions invoked in neural networks. Therefore, all methods that feature a second-order character, such as a Quasi-Newton approach, will face both theoretical and practical difficulties.

The lack of convexity and the fact that the problem is heavily over-parameterized mean that there are many solutions to which the algorithms can converge. To which of these will the optimizer converge and which optimizer is likely to perform better? If we consider the objective function as the sole criterion, one may think that the answer is clear: the lower the better. However, practitioners in this field are more interested in 'generalization' or the property to obtain good predictions on data that is not among the training data set.

In the past few years, quite a few published articles were devoted to understanding the nature of deep learning and, in particular, the problem of generalization has been the object of numerous recent studies, see, e.g., [32–35] among many others. A puzzling character of neural networks is that they tend to do quite well at classifying items that do not belong to the training set. However, the paper [34] shows by means of experiments that looking at DL from the angle of minimizing the loss function fails to explain these nice generalization properties. The authors show that they can achieve a perfect loss of zero in training models on well-known datasets (MNIST, CIFAR10) that have been modified by randomly changing all labels. In other words one can obtain parameters whose loss function is minimum but with the worst possible generalization since the resulting classification would be akin to assigning a random label to each item. A number of other papers explore this issue further [32, 33, 35, 36] by attempting to explain generalization with the help of the 'loss landscape', the geometry of the loss function in high dimensional space. What can be understood from these works is that the problem is far more complex than just minimizing a function and that the random nature of the optimization plays a central role.

3.5 Computational graphs and back-propagation

Back-propagation is another example of an important contribution of scientific computing to machine learning. In fact it may be argued that the impact of back-propagation is just as important as that of high-performance hardware in the revival of deep learning. Given a certain function ϕ the problem faced by software developers is to compute the partial derivatives of ϕ in order to deploy common optimization algorithms. The techniques of 'automatic differentiation' developed for this purpose was popularized by the work of Andreas Griewank [22–24] among others. Back-propagation is the term used for this method in deep learning.

Computational graphs are directed graphs, where vertices represent tasks that must be executed in the order dictated by the directed edges of the graph: An evaluation of a node will depend on other (incoming) nodes. For example we may have a node that evaluates

$$f(x, y, z) = \phi(a(x, y, z), b(x, y, z), c(x, y, x)). \tag{3.23}$$

In this case, node (f) will be evaluated once nodes (a), (b) and (c) have been calculated. Each of these in turn may depend on other nodes. Figure 3 shows a local view of such a graph for the case of MLP seen earlier and captured by Equation 3.4. The arrows indicate the direction of 'forward propagation'. If the graph terminates at a function f at the root, we often have to evaluate (i) the nodes, ending with the node (f); and (ii) the derivatives of the root function f with respect to the primary variables x, y, z, for some set values of x, y, z. Part (i) is performed by a 'forward propagation' algorithm a simple case of which is shown in Algorithm 1. Part (ii) can be evaluated by a standard forward equivalent algorithm using chain rules. However, it is far more convenient to use 'back propagation' where we assume that a forward pass has already been performed so the nodes have been evaluated.

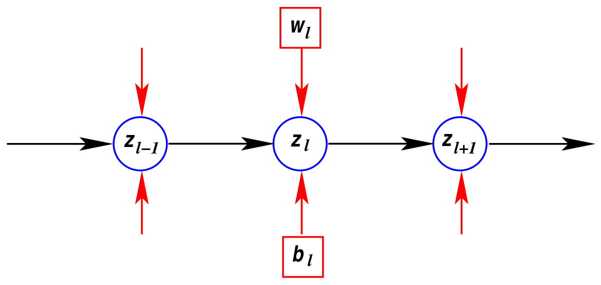

Fig. 3 Graph representation of the forward phase of the calculation in Equation 3.4

Consider now a generic situation like the one shown in Figure 4 illustrating a back-propagation computation where we assume that the partial derivatives of the target function f with respect to the incident nodes a_j, a_l, a_m have already been computed. We now need to compute $\partial f/\partial a_k$ and this can be done with the chain rule:

$$\frac{\partial f}{\partial a_k} = \frac{\partial f}{\partial a_j}\frac{\partial a_j}{\partial a_k} + \frac{\partial f}{\partial a_l}\frac{\partial a_l}{\partial a_k} + \frac{\partial f}{\partial a_m}\frac{\partial a_m}{\partial a_k} \tag{3.24}$$

Here, the nodes $a_i, i = 1 : n$ are tasks in a computational graph with the last node $a_n \equiv f$ being the target function. The leaf nodes $a_i, i = 1, \cdots, e$ are the primary variables (e.g., x, y, z in (3.23)) and we wish to compute $\partial f/\partial a_1, \partial f/\partial a_2, \cdots, \partial f/\partial a_e$.

The notation often used is to let $\delta_k = \frac{\partial f}{\partial a_k}$ (called 'errors'). Then back-propagation amounts to successively evaluating the δ_k's, following the graph backward from the root (function f):

$$\delta_k = \delta_j \frac{\partial a_j}{\partial a_k} + \delta_l \frac{\partial a_l}{\partial a_k} + \delta_m \frac{\partial a_m}{\partial a_k}. \tag{3.25}$$

Here, the nodes $\delta_j, \delta_l, \delta_m$ have been evaluated in earlier steps of back-propagation and the terms $\partial a_i/\partial a_k$ are readily computable. Note that the initial 'error' corresponding to f is $\delta_n = \partial f/\partial f \equiv 1$. The leaves in the back-propagation graph correspond to the desired partial derivatives.

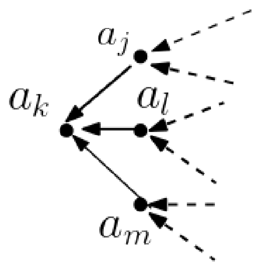

Fig. 4 Zooming in on a node (a_k) in back-propagation

It is clear that in a general computational graph, there is an order in which to proceed in the back-propagation since a task cannot be started before its parent nodes in the graph have been processed. This is a textbook example of *topological sorting*, an ordering of the nodes in a directed acyclic graph (DAG) to obey the precedence relations. Finally, we note that the computation involved in (3.25) amounts to a matrix-vector product.

As an example consider the simple MLP model as represented by equation (3.4). The computational graph localized at node l is shown in Figure 3. Let us call f the original objective function which is obtained from the last (output) layer, e.g., recalling notation from Section 3.2, $f(x) = \|y - \phi_W(x)\|_2^2$ where we essentially take one data item at a time, e.g., x_i, y_i replaced here by x, y as in Section 3.1. This is a function of the parameters throughout all layers and we would like to obtain the gradient of f with respect to these parameters. In the figure, the nodes in the graph are represented by the circles (the z_k's) as well as the squares (the parameters, W_k, b_k). Assume a forward propagation step is taken in which case all nodes have been evaluated. In a back-propagation step, the arrows in Figure 3 are reversed. At layer l we evaluate:

$$\frac{\partial f}{\partial z_l} = \frac{\partial f}{\partial z_{l+1}} \times \frac{\partial z_{l+1}}{\partial z_l} \tag{3.26}$$

Note that $\frac{\partial f}{\partial z_{l+1}}$ is assumed to have been evaluated at a prior step in traversing the graph and that $\frac{\partial z_{l+1}}{\partial z_l}$ is readily computable from by using (3.4) with l replaced by $l+1$. Following the (reversed) arrows, we next compute

$$\frac{\partial f}{\partial W_l} = \frac{\partial f}{\partial z_l} \times \frac{\partial z_l}{\partial W_l} \quad \text{and} \quad \frac{\partial f}{\partial b_l} = \frac{\partial f}{\partial z_l} \times \frac{\partial z_l}{\partial b_l}. \tag{3.27}$$

The calculations in the above equations take place in the 'leaves' of the back-propagation graph. The corresponding nodes are terminal in the sense that the gradients calculated will no longer be modified and will not be used in other calculations. These are the desired gradients of f with respect to parameters W_l, b_l. We have taken the example (3.4) for simplicity - but in reality the calculations just discussed are performed on the matrices Z_l in the relation (3.6). Conceptually, all this means is that we can treat rows of Z_l one at a time in the manner just discussed. For additional details see [25].

4 The idea of Attention – Transformer Architecture

An idea that has played a decisive role in Large Language Models is that of "Attention", see, the breakthrough article titled *"Attention is all you need"* [17]. It is worthwhile to illustrate the notion of attention with a simple example to show the difference in thinking between Numerical Analysis and Machine Learning.

4.1 NA vs AI thinking - an example

Suppose we are given very noisy 'training points' x_i, y_i which are x-coordinates and y-coordinates of some unknown function f, at specific points. The goal is to 'recover' f in some form, see left side of Figure 5. The *Numerical Analysis* approach is straightforward: use the data points to interpolate the function in the Least-Squares sense. We need to first select the type of interpolating function we will use, for example a cubic polynomial.

The *Machine Learning* solution to the problem is very different and relies entirely on the given data points. The approach based on attention uses an advanced form of averaging and it has its roots in databases. We are given known *key, value* pairs $\{k_i, v_i\}$ and would like to guess a value for a certain *query* q which is of the same type as the keys. In the example given above, the key-value pairs are the pairs $(x_i^{train}, y_i^{train}\})$ of the given data. In the same example, q is an arbitray x-coordinate where we want to the approximation to f. A naive first solution would be to take the closest key k_l to q and declare the value of f at q to be v_l. A better solution is to take some weighted

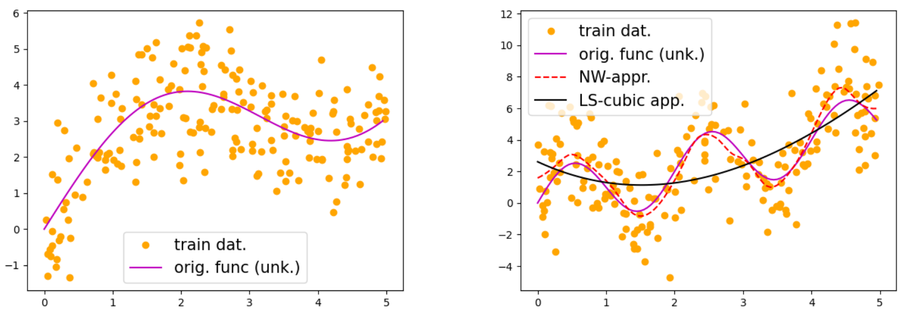

Fig. 5 (Left) The problem: given a cloud of (very) noisy data points, find the original function. (Right) Illustration of the Nadarawa Watson kernel regression on periodic data.

average of all values v_i, with weights defined so as to give more importance (attention) to more relevant training points. These weights are defined through a *kernel* $a(q, k)$.

There are a number of options for the kernel a, one of which is to use Gaussian kernels such as the one expressed below for the more general case where the k_i's, and q are in $\mathbb{R}^d$:

$$a(q, k_i) = \frac{\exp(-\frac{1}{2}\|q - k_i\|^2/\sigma^2)}{\sum_{l=1}^n \exp(-\frac{1}{2}\|q - k_l\|^2/\sigma^2)}. \tag{4.1}$$

Observe that the values of $a(q, k_i)$ are positive and scaled so that they add-up to unity and so they play the role of probabilities. The inferred value v for q is then given by:

$$\sum_{i=1}^n a(q, k_i)v_i.$$

This process is a form of Kernel Regression [25, 37] known as Nadaraya-Watson attention. It is illustrated in Figure 6.

An example is given on the right side of Figure 5 for a function that is periodic. If we did not know the nature of the function we might think of approximating it with a cubic. As can be seen this results in a poor approximation in this case. Here, the attention-based approximation does a much better job. The main point of this illustration is that with a lot of data, we can succeed in obtaining a good approximation to an unknown function by using attention.

4.2 Transformer architecture

The transformer architecture has become the dominant architecture in many applications such as neural machine translation, text generation, and vision, see e.g. [38, 39]. Transformer-based large language models like GPT-1 [40], BERT [41], GPT-2 [42], GPT-3 [43], ChatGPT, Llama [44, 45], etc. have demonstrated impressive capabilities in natural language processing, e.g., they are able to generate complex language and programming code, answer questions and summarize text.

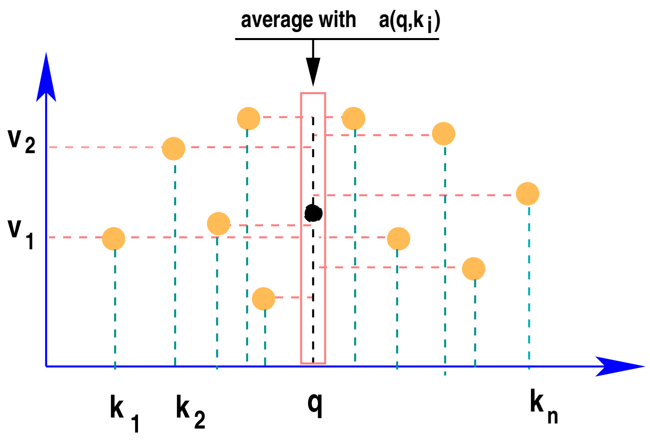

Fig. 6 Illustration of the Nadarawa Watson kernel regression.

A transformer is a sequence-to-sequence model (Seq2Seq). The original (vanilla) transformer architecture, introduced in [17] and used for machine translation, consists of an *encoder* and a *decoder*[3]. The encoder processes the input sequence and produces a vector representation for it, while the decoder generates the output sequence one token at a time, taking into account both the encoder's outputs and previously generated tokens. The encoder consists of multiple stacked encoder blocks. Similarly, the decoder consists of multiple stacked decoder blocks.

Each transformer block ℓ, encoder or decoder, may be considered as a parametrized function $\mathcal{T}_\ell(X_{\ell-1};\Theta_\ell)$ which transforms an input data matrix $X_{\ell-1} \in \mathbb{R}^{n\times d}$ into an output data matrix $X_\ell \in \mathbb{R}^{n\times d}$, i.e.,

$$X_\ell = \mathcal{T}_\ell(X_{\ell-1};\Theta_\ell), \tag{4.2}$$

where Θ_ℓ represents the parameters in transformer block ℓ, sometimes termed layer ℓ.

Hence, the output of a Transformer with L transformer blocks is the composition of L functions $\mathcal{T}_\ell$ corresponding to the transformers blocks $\ell = 1,\cdots,L$, each with their own parameters, i.e.,

$$X_L = \mathcal{T}(X_0) = (\mathcal{T}_L \circ \mathcal{T}_{L-1} \circ \cdots \circ \mathcal{T}_1)(X_0). \tag{4.3}$$

By composing these transformer blocks, the Transformer progressively adjusts these token embeddings so that they don't merely encode individual words but will

[3]Terminology: In deep learning, an encoder is a neural network (a simple RNN, LSTM, GRU, convolutional network, or a transformer network) defined as a parametric function $\mathcal{E}_\varphi$ which maps the original data $X \in \mathbb{R}^{n\times d}$ to a latent representation $Z \in \mathbb{R}^{n\times d'}$, i.e., $Z = \mathcal{E}_\varphi(X)$, where the dimensions d and d' need not be the same, and usually $d' \le d$. A decoder performs the opposite operation, i.e., it is a parametrized function $\mathcal{D}_\theta$ which takes as input the latent bottleneck Z to produce an output, i.e., $\mathcal{D}_\theta(\ \mathcal{E}_\varphi(X)\ ) \in \mathbb{R}^{n\times d}$. An "encoder-decoder architecture" like the ones we are considering here and based on [17] employs both an encoder and a decoder component to process input and generate an output sequence, where $d' = d$ end-to-end. This makes it suitable for tasks like machine translation. An "encoder-only transformer" employs just an encoder, focusing on understanding the input context without generating new text making it ideal for tasks like text classification or sentiment analysis for example.

instead lead to learning some contextual meaning. Each transformer block consists of two distinct stages or sublayers, with layer normalization in each stage and residual connections, as shown in Figure 7 for a pre- and post-`LayerNorm` transformer block, see the ground-breaking work of Vaswani et al. in [17]. The first sublayer is the *multi-headed self-attention* (`MultiHead` or `ATT` for short), followed by the *Multilayer Perceptron* (`MLP`) sublayer, which includes a nonlinear activation function to make the transformer block more expressive. Hence, a Transformer is purely based on attention and dense layers, i.e., $\texttt{Transformer\_block} = \texttt{ATT} + \texttt{MLP}$.

Mathematically, the transformation function class $\mathcal{T}_\ell$ of a pre-`LayerNorm` transformer block ℓ is represented by the equations

$$X_\ell = \mathcal{T}_\ell(X_{\ell-1}) \quad \triangleq \quad \begin{cases} Y_\ell = X_{\ell-1} + \texttt{ATT}(\ \texttt{LN}(X_{\ell-1})\ ), & (4.4) \\ X_\ell = Y_\ell + \texttt{MLP}(\ \texttt{LN}(Y_\ell)\ ), & (4.5) \end{cases}$$

where the functions `ATT, LN, MLP` are implicitly indexed by layer ℓ. And in the final representation, an additional layer normalization is applied, i.e.,

$$X_L \leftarrow \texttt{LN}(\ X_L\ ), \tag{4.6}$$

before this final output is passed into a bias-free linear layer to obtain logits. Table 2 presents the equations for both scenarios of Figure 7.

	Post-`LayerNorm`	Pre-`LayerNorm`
① `ATT`	$Y_\ell = \texttt{LayerNorm}(\ X_{\ell-1} + \texttt{ATT}(X_{\ell-1})\ )$	$Y_\ell = X_{\ell-1} + \texttt{ATT}(\ \texttt{layerNorm}(X_{\ell-1})\ )$
② `MLP`	$X_\ell = \texttt{LayerNorm}(\ Y_\ell + \texttt{MLP}(Y_\ell)\ )$	$X_\ell = Y_\ell + \texttt{MLP}(\ \texttt{LayerNorm}(Y_\ell)\ )$

Table 2 Transformer: Post-`Norm` and Pre-`Norm` with Skip Connections.

4.2.1 Attention mechanism (`ATT`)

The core foundation that supports the capabilities of LLMs is the attention block `ATT`. The central concept of attention is to learn representations that emphasize the most relevant parts of the input prompt. Specifically, the attention mechanism compares the query vectors $q_i^{(\ell)} = W_Q^T x_i^{(\ell-1)}$ (the decoder's output tokens, in the case of an encoder-decoder transformer), with the key vectors $k_i^{(\ell)} = W_K^T x_i^{(\ell-1)}$ (the encoder's input tokens). The attention weights are then determined based on the similarity of this comparison, indicating the relative importance of each input token, i.e.,

$$\texttt{Attention}(Q_\ell,\ K_\ell) = \texttt{softmax}\Big(\frac{Q_\ell\, K_\ell^T}{\sqrt{d_k}}\Big) \ \in \mathbb{R}^{n\times n}, \tag{4.7}$$

where $Q_\ell, K_\ell \in \mathbb{R}^{n\times d_k}$ represent the matrices for queries and keys, respectively. The softmax operation in the above equation was defined defined earlier in (3.9). It is

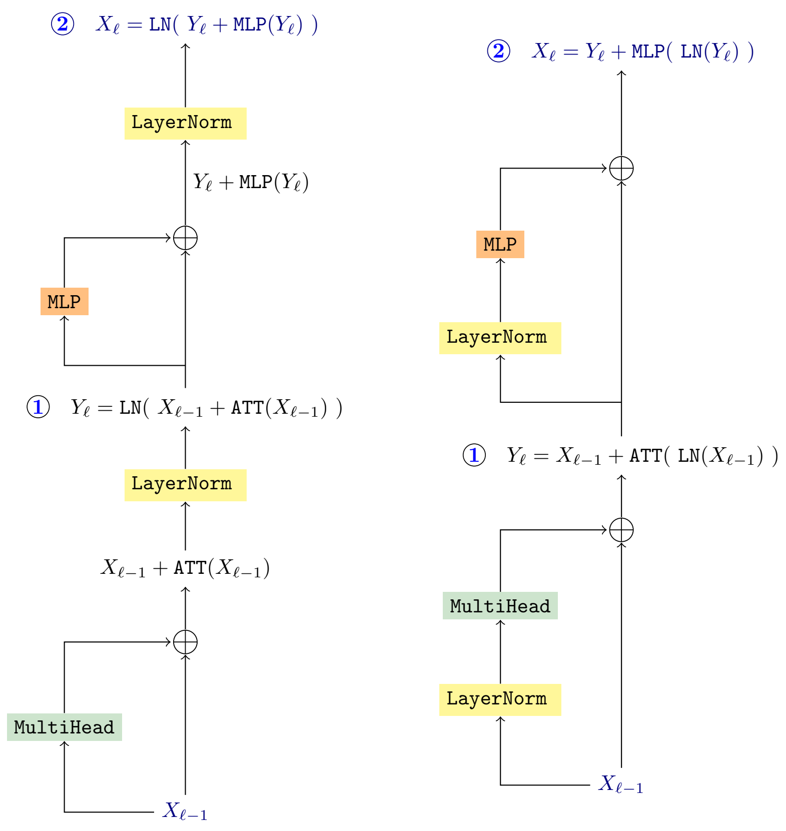
② $X_\ell = \texttt{LN}(\ Y_\ell + \texttt{MLP}(Y_\ell)\ )$
LayerNorm
$Y_\ell + \texttt{MLP}(Y_\ell)$
MLP
① $Y_\ell = \texttt{LN}(\ X_{\ell-1} + \texttt{ATT}(X_{\ell-1})\ )$
LayerNorm
$X_{\ell-1} + \texttt{ATT}(X_{\ell-1})$
MultiHead
$X_{\ell-1}$
② $X_\ell = Y_\ell + \texttt{MLP}(\ \texttt{LN}(Y_\ell)\ )$
MLP
LayerNorm
① $Y_\ell = X_{\ell-1} + \texttt{ATT}(\ \texttt{LN}(X_{\ell-1})\ )$
MultiHead
LayerNorm
$X_{\ell-1}$

Fig. 7 Post-LayerNorm and Pre-LayerNorm of the Transformer block ℓ.

applied row-wise: for each row i ($i = 1, \cdots, n$) the vector z in (3.9) is the row-vector consisting of the i-th row of the matrix $Q_\ell K_\ell^T/\sqrt{d}$ and the resulting row will be the i-th row of the attention matrix on the left-hand side of (4.7). Note that without the scaling factor $\sqrt{d}$, this row vector z consists of all inner products of row i of Q_ℓ with all the rows of K_ℓ thus capturing some form of similarity between item i in Q_ℓ with all other items in K_ℓ. It is said that 'each word attends to each other word in a sentence'. The matrix for (encoder's) values is $V_\ell \in \mathbb{R}^{n \times d_v}$, with $d_v = d_k$ usually, i.e.,

$$Q_\ell,\ K_\ell,\ V_\ell = X_{\ell-1}W_Q,\ X_{\ell-1}W_K,\ X_{\ell-1}W_V. \tag{4.8}$$

The attention weights, Equation (4.7), are used to compute weighted averages of the value vectors $v_i^{(\ell)} = W_V^T x_i^{(\ell-1)}$, resulting in the output context representation of the attention sublayer, i.e., $\texttt{ATT}(X_{\ell-1})$, see Algorithm 2,

Thus, applying the attention matrix (4.7) to the value matrix $(X_{\ell-1}W_V)$ we obtain:

$$\texttt{ATT}(X_{\ell-1}) = \texttt{softmax}\left(\frac{(X_{\ell-1}W_Q)(X_{\ell-1}W_K)^T}{\sqrt{d_k}}\right)(X_{\ell-1}W_V).$$

Algorithm 2 `SingleHead` Self-Attention

```
Input: $X \in \mathbb{R}^{n \times d_{\texttt{model}}}$        ▷ representation of a given sequence of $n$ tokens
Parameters: $W_Q, W_K, W_V$        ▷ learned weight matrices
Output: $A \in \mathbb{R}^{n \times d_{\texttt{model}}}$        ▷ updated representation of tokens in $X$
1: function $A = \textsc{SingleHead}(X \mid W_Q, W_K, W_V)$        ▷ Computes a single self-attention head
2:     $Q,\ K,\ V = XW_Q,\ XW_K,\ XW_V$        ▷ query, key, value matrices
3:     $d_k = \texttt{size}(Q, 2)$
4:     $S = Q\,K^T/\sqrt{d_k}$        ▷ scaled dot-product, $S$ is a matrix of size $n \times n$
5:     $\texttt{probs} = \texttt{softmax}(S)$        ▷ attention matrix of probabilities return $A = \texttt{probs} \times V \quad (\equiv \texttt{ATT}(X))$        ▷ output context representation
6: end function
```

4.2.2 Multihead self-attention

The steps of `SingleHead` self-attention are outlined in Algorithm 2. This requires an input sequence $X \in \mathbb{R}^{n \times d_{\texttt{model}}}$ and each layer ℓ has three parameter weight matrices W_Q, W_K, W_V which are randomly initialized. The output is a context representation $A \in \mathbb{R}^{n \times d_{\texttt{model}}}$. This can be generalized to a `Multihead` self-attention by running independently multiple `SingleHead` attentions. These $H = n_{\texttt{heads}}$ `SingleHead` attentions have the same inputs but they do not share the parameters. The total number of parameter weight matrices is $3 \times n_{\texttt{heads}}$, i.e., $W_{Q,h}, W_{K,h}, W_{V,h}$ for $h = 1, \cdots, n_{\texttt{heads}}$. The transformer's multi-head scaled dot-product attention is given by

$$\texttt{ATT}(X_{\ell-1}) = \texttt{MultiHead}(Q_\ell, K_\ell, V_\ell) = \mathop{\Big\|}_{h=1}^{H} (\ \texttt{HA}_h\ )W_O, \tag{4.9}$$

with the h-th head attention $\texttt{HA}_h$ defined as,

$$\texttt{HA}_h = \texttt{softmax}\Big(\frac{(X_{\ell-1}W_{Q,h})(X_{\ell-1}W_{K,h})^T}{\sqrt{d_{\texttt{head}}}}\Big)(X_{\ell-1}W_{V,h}), \tag{4.10}$$

where $W_{Q,h}, W_{K,h} \in \mathbb{R}^{d_{\texttt{model}} \times d_{\texttt{head}}}$ and $W_{V,h} \in \mathbb{R}^{d_{\texttt{model}} \times d_{\texttt{head}}}$ are the matrices that project the queries, keys, and values into the hth subspace, respectively, i.e., $\texttt{HA}_h \in \mathbb{R}^{n \times d_{\texttt{head}}}$, and where $W_O \in \mathbb{R}^{d_v \times d_{\texttt{model}}}$ is the matrix that computes a linear transformation of the heads. Typically, $d_v = d_k$ and $d_{\texttt{head}} = d_k/n_{\texttt{heads}}$ where $n_{\texttt{heads}}$ is the total number of heads, and $\|$ concatenates the heads together along the channel dimension.

Let us decompose the output projection matrix W_O into $n_{\texttt{heads}}$ block matrices $W_{O,h} \in \mathbb{R}^{d_{\texttt{head}} \times d_{\texttt{model}}}$ for $h = 1, 2, \cdots, n_{\texttt{heads}}$. It is easy to see that:

$$\begin{aligned}\texttt{ATT}(X_{\ell-1}) &= \sum_{h=1}^{n_{\texttt{heads}}} (\texttt{HA}_h)W_{O,h}, && (4.11)\\ &= \sum_{h=1}^{n_{\texttt{heads}}} \texttt{softmax}\Big(\frac{(X_{\ell-1}W_{Q,h})(X_{\ell-1}W_{K,h})^T}{\sqrt{d_{\texttt{head}}}}\Big)X_{\ell-1}W_{V,h}\,W_{O,h} && (4.12)\end{aligned}$$

which can be interpreted as computing self-attention heads $\texttt{HA}_h$ independently, multiplying each by its own output matrix $W_{O,h}$ and adding them, see [46]. In so doing, each $\texttt{HA}_h$ could find a different kind of relation between tokens.

Algorithm 3 `MultiHead` Self-Attention

```
Input: $X \in \mathbb{R}^{n\times d_{\texttt{model}}}$                                    ▷ sequence representation
Parameters: $W_{Q,[1\ldots n_{\texttt{heads}}]}, W_{K,[1\ldots n_{\texttt{heads}}]}, W_{V,[1\ldots n_{\texttt{heads}}]}, W_O$      ▷ learned matrices
Output: $A \in \mathbb{R}^{n\times d_{\texttt{model}}}$                    ▷ updated representation of tokens in $X$
1: function $A$ = MultiHead($X \mid W_{Q,[1\ldots n_{\texttt{heads}}]}, W_{K,[1\ldots n_{\texttt{heads}}]}, W_{V,[1\ldots n_{\texttt{heads}}]}, W_O$)
2:     for $h = 1, \cdots, n_{\texttt{heads}}$ do
3:         $\texttt{HA}_h = \texttt{SingleHead}\ (X \mid W_{Q,h}, W_{K,h}, W_{V,h})$
4:     end forreturn $A = \overset{H}{\underset{h=1}{\|}}\ (\ \texttt{HA}_h\ )W_O \quad (\equiv \texttt{ATT}(X))$     ▷ output context
   representation
5: end function
```

4.2.3 Multilayer perceptron (MLP)

The second stage of a transformer block is a two-layer MLP. It is a standard feed-forward network consisting of the composition of two affine mappins and a nonlinear activation function. The output of the attention stage, i.e., $Y_\ell = X_{\ell-1} + \texttt{ATT}(\texttt{LN}(X_{\ell-1}))$, becomes the input to the first layer which is parametrized by $(W^\ell_{\texttt{up}}, b^\ell_{\texttt{up}})$ with $W^\ell_{\texttt{up}} \in \mathbb{R}^{d_{\texttt{model}}\times d_{\texttt{mlp}}}$ such that $d_{\texttt{mlp}} \gg d_{\texttt{model}}$. It is essentially a linear projection to a higher-dimensional space, i.e., $Y_\ell W^\ell_{\texttt{up}} + b^\ell_{\texttt{up}}$. Then the second layer, parametrized by $(W^\ell_{\texttt{down}}, b^\ell_{\texttt{down}})$, projects it back to $d_{\texttt{model}}$, after passing through a nonlinearity activation, preparing the token for further processing, that is

$$\texttt{MLP}(Y_\ell) = \overbrace{\texttt{ReLU}(\underbrace{Y_\ell W^\ell_{\texttt{up}} + b^\ell_{\texttt{up}}}_{1^{\text{st}}\ \texttt{Layer}})W^\ell_{\texttt{down}} + b^\ell_{\texttt{down}}}^{2^{\text{nd}}\ \texttt{Layer}} \tag{4.13}$$

where the learned parameter matrix $W^\ell_{\texttt{down}}$ is of size $(d_{\texttt{mlp}}, d_{\texttt{model}})$.

4.2.4 Stack of residual connections

It is well known that deep networks are difficult to optimize due to the gradient vanishing/exploding problem, e.g. see [47, 48]. Residual connections and layer normalization are adopted for a solution, see [49]. A residual unit, with identity mapping, can be expressed in a general form as:

$$\begin{cases} \widetilde{X}_\ell = X_{\ell-1} + \mathcal{F}(X_{\ell-1}; \Theta_\ell), & (4.14) \\ X_\ell = f(\widetilde{X}_\ell), & (4.15) \end{cases}$$

where $X_{\ell-1}$ and X_ℓ are the input and output of the ℓ-th sub-layer, respectively, and $\mathcal{F}$ is a residual function with parameters Θ_ℓ; and $\widetilde{X}_\ell$ is the intermediate output followed by the post-processing function $f(\cdot)$.

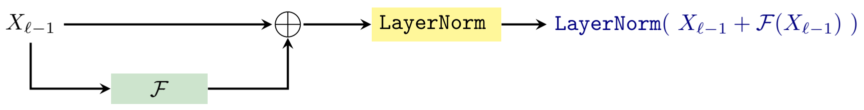
$X_{\ell-1}$
LayerNorm
$\mathcal{F}$
$\texttt{LayerNorm}(\ X_{\ell-1} + \mathcal{F}(X_{\ell-1})\ )$

Fig. 8 Post-`LayerNorm` residual.

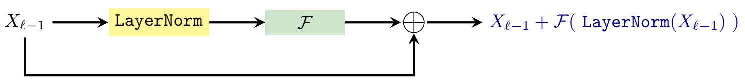
$X_{\ell-1}$
LayerNorm
$\mathcal{F}$
$X_{\ell-1} + \mathcal{F}(\ \texttt{LayerNorm}(X_{\ell-1})\ )$

Fig. 9 Pre-`LayerNorm` residual.

In the case of Transformers, the additive residual function $\mathcal{F}$ is a multihead self-attention function, `ATT`, in the first stage of the Transformer block ℓ, and a multilayer perceptron, `MLP`, in the second stage of the Transformer block ℓ.

In the early versions of the Transformer, layer normalization is placed after the element-wise residual addition, where the function block $\mathcal{F}$ is either `ATT` or `MLP`, see Figure 7. In this case, the layer normalization function, can be seen as a post-processing step of the output, $\widetilde{X}_\ell = X_{\ell-1} + \mathcal{F}(X_{\ell-1})$, i.e., $f(\widetilde{X}_\ell) = \texttt{LayerNorm}(\widetilde{X}_\ell)$. Whereas in recent implementations, layer normalization is applied to the input of every sub-layer, i.e., before the function operation $\mathcal{F}$. Pre-layer normalization is applied before every stage, and residual connections after every stage, i.e., $f(\widetilde{X}_\ell) = \widetilde{X}_\ell$, see Figure 9.

4.2.5 Layer Normalization

Layer normalization is a pivotal step in the transformer architecture. It is applied to the input and output of each sub-layer in the encoder and decoder stacks, in the original Transformer [17]. Layer normalization normalizes the values of the hidden units across the feature dimension, which helps in reducing the internal covariate shift and improves the training process. It computes the mean and variance of the hidden units and then applies a linear transformation to normalize the values. Layer normalization is beneficial as it allows the model to handle inputs with varying lengths and reduces the dependence on the scale of the input. It also helps in stabilizing the training process and improving the overall performance of the Transformer model.

Let $x \in \mathbb{R}^d$ represent an intermediate hidden representation in a transformer-based model, on which the `LayerNorm` operation is applied, i.e., any i^{th} row-vector $X_{i,:}^T$ of the data matrix $X \in \mathbb{R}^{n \times d_{\text{model}}}$. The corresponding normalized vector y, scaled and shifted, is

$$y = \texttt{LayerNorm}(x) = \frac{x - \mu(x)}{\sqrt{\sigma^2 + \varepsilon}} \odot \gamma + \beta, \tag{4.16}$$

where
$gamma, \beta$ are scaling and shifting vector parameters learned during training, and ε is used for numerical stability in case the denominator becomes close to zero by chance, and usually $\varepsilon = 1.0e^{-5}$, e.g., see [50–52].

5 Graph Attention Networks

Graphs serve as powerful tools for representing relationships and interactions in various domains such as social networks (e.g. identify fake news, predict future friends, learn multi-faceted interactions among users) [53, 54], chemistry (generate new drugs and materials, predict chemical properties) [55, 56], recommender systems (e.g., leverage consumer-product choices) [57, 58], knowledge graphs (reasoning with entity relationships) [59, 60], natural language processing (e.g. large language models) [17], physics (e.g. learn from interactions of particles in systems, detect particles, accelerate physics research) [61, 62], neuroscience (e.g., learn functions of brain regions through connectivity, understand brain mechanisms and neuro-degenerative diseases) [63], transportation (e.g., learn traffic behavior across road networks, predict time estimates across multilayered networks) and more.

Graph learning has gained prominence with the advent of graph neural networks, which generalize neural networks to graph-structured data, e.g. see [55, 57, 64–68]. There exists two main classes of graph neural network architectures: *(i) message-passing GNNs* which may be considered as a generalization of convolution, i.e., a direct extension of ConvNets from images to graphs, e.g. see [69], and *(ii) Weisfeiler-Lehman* GNNs which are recently proposed and go beyond message-passing GNNs. They exhibit powerful expressivity and are permutation-invariant universal approximators, see [70–74]. Recent works present a hybrid variant, i.e., *WL-* and *MP-GNNs*, see [75, 76]. This section will explore key developments that blend ideas from neural networks and graphs, with a particular focus on *message-passing* GNNs.

5.1 Graph Neural Networks (GNNs)

A GNN is not a specific model but rather a framework of a general class of neural networks that can be effective when working with graph-structured data. The goal is to generate representations of the nodes in the graph, or of the graph itself, that can then be used for various tasks. GNNs will serve as templates for the specific models to be seen shortly.

All GNNs aim at providing solutions for certain tasks related to graphs. These tasks include: node classification (determining the label of a vertex in a graph), graph classification (given a set of labeled graphs, determine the label of a new graph), and link prediction (predict whether or not there will be an edge between two vertices in a graph.

The goal of a GNN is to produce an embedding for a graph (graph classification) or the nodes of a graph (node classification). It works by iteratively updating node features based on features from their neighbors, allowing the model to learn representations, i.e., embeddings, by exploiting both the content of the nodes (features) and the

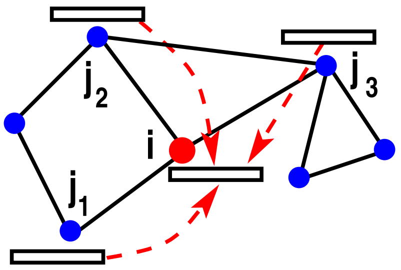

Fig. 10 Illustration of message passing: The new feature at v_i is combined with features of its nearest neighbors $v_{j_1}, v_{j_2}, v_{j_3}$.

adjacency matrix, which captures links between nodes. Each node in the graph *aggregates* information from its neighbors in each layer of the network, and the aggregated information is used to update the node's feature representation, see [54].

The fundamental operation of GNNs is *the message-passing mechanism*, where neural networks are used to exchange and update vectors messages. The feature vector x_i of a node v_i in the graph is updated by aggregating features from its neighbors. The message-passing update rule can be written as:

$$x_i^{(\ell+1)} = \texttt{UPDATE}\Big(x_i^{(\ell)},\ \texttt{AGGREGATE}(\ \{x_j^{(\ell)} : j \in \mathcal{N}(v_i)\}\ )\Big), \tag{5.1}$$

where $x_i^{(\ell)}$ represents the feature vector of node i at the ℓ-th layer, $\mathcal{N}(v_i)$ denotes the set of neighbors of node v_i, `AGGREGATE` is an aggregation function (e.g., summation, mean, or max pooling) that generates a message with the features of neighboring nodes $\mathcal{N}(v_i)$, whereas the parametric `UPDATE` function combines the generated message with the previous embedding to update the node's feature vector, typically involving a neural network layer, with $x_i^{(0)} = x_i \in \mathbb{R}^{d_0}, \forall v_i \in \mathcal{V}$. We can define different GNNs by simply varying the function the `UPDATE` and the `AGGREGATE` operations. A few examples follow.

5.2 Graph Convolutional Networks (GCNs)

GCNs generalize the concept of convolutions from grid-like data (such as images) to arbitrary graph structures, enabling the model to capture spatial relationships and node dependencies efficiently. GCNs have gained popularity due to their ability to process graph data in a hierarchical manner while maintaining the flexibility to handle varying graph sizes and structures.

The idea of GCN followed decades long research on 'graph embeddings' mentioned in the introduction. Following the success of convolutional networks, the authors [77] defined a convolution operator that is sparse and defined directly from the adjacency matrix A. Specifically, we shift the matrix by the identity to obtain $\tilde{A} = A + I$, i.e., self-loops are added to the input graph, and we let $\tilde{D}$ be a diagonal matrix with diagonal entries $\tilde{d}_{ii} = \sum_j \tilde{a}_{ij}$. We then define the normalized adjacency matrix with self-loops:

$$\hat{A} := \tilde{D}^{-1/2} \tilde{A} \tilde{D}^{-1/2}, \tag{5.2}$$

where this normalization step is applied to ensure that the feature aggregation is not biased by nodes with a high degree. Balanced aggregation improves training stability, and adding self-loops ensures that isolated nodes are included in learning. The idea is that convolution is replaced by a product with $\hat{A}$: a given new feature at node i at the next level results from some average of the features of its nearest neighbors using the matrix $\hat{A}$ followed by an application of the activation function σ. This corresponds to the `AGGREGATE` process seen in (5.1), and we omit the `UPDATE` step, i.e.,

$$x_i^{(\ell+1)} = \texttt{AGGREGATE}(\ \{x_j^{(\ell)} : j \in \mathcal{N}(v_i) \cup \{v_i\}\}\ ), \tag{5.3}$$

where the `AGGREGATE` function is taken over the node's neighbors and the node itself. Therefore, with the notation just introduced, and defining $X^{(0)} \equiv X$ the node feature matrix, a basic GCN layer can be described simply as follows:

$$X^{(\ell+1)} = \sigma\Big(\hat{A} X^{(\ell)} W^{(\ell)}\Big), \tag{5.4}$$

where $X^{(\ell)} \in \mathbb{R}^{n \times d_\ell}$ is the matrix of node features at layer ℓ, n is the number of nodes, and d_ℓ is the dimensionality of the feature space at layer ℓ; $W^{(\ell)} \in \mathbb{R}^{d_\ell \times d_{\ell+1}}$ is a learnable weight matrix at layer ℓ; and σ is a non-linear activation function (e.g., `ReLU`). The key operation is the multiplication of the normalized adjacency matrix $\hat{A}$ with the feature matrix, followed by a linear transformation and a non-linear activation.

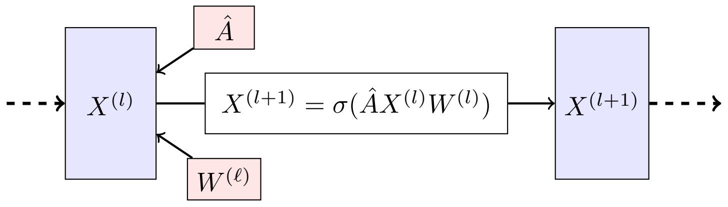

Fig. 11 The l-th layer of GCN.

Among the problems that have been successfully tackled by GCN are: node classification and graph classification. Node classification starts with an input graph, and the main goal of GCN is to find embeddings of the nodes of this graph. In the case of graph classification we are given a set of graphs (e.g., graphs of molecules) and GCN will provide an embedding for each graph. These embeddings will then help with the tasks mentioned above, whether graph or node classification.

In what follows we take a close look at node classification. The input is the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of nodes v_i, $i = 1, \cdots, n$ and $\mathcal{E}$ is a set of edges $e_{ij} = \{v_i, v_j\}$. This graph is conveniently represented by its adjacency matrix A, a sparse matrix, with values a_{ij} equal to 1 or 0, which indicates the presence or absence of an edge between node v_i and node v_j. With each node v_i of the graph we associate a feature vector that contains information deemed characteristic of this node. Assuming the dimension of the feature vector is d, we can represent all features by an $n \times d$ matrix X – where row i holds the features of node i (transposed). For example, X can just be the so-called term-document matrix where row i (node i) is the i-th item (a document) and the entries are either 1 (the term is present in document) or 0 (term not present in document). Our goal is to go through L layers of a neural network and end up with an embedding $\hat{Y} \in \mathbb{R}^{n\times\mu}$. For example if the problem is to classify a node into one of μ classes, the ideal embedding for a given node v_i would be a one-hot vector $\hat{y}_i$ of length μ, where a value of one in location k means that v_i is in class k.

The target used for training is typically a one-hot matrix Y of size $n \times \mu$. Then, the loss function that is often used is the common one used for classification, namely the cross-entropy loss seen in Section 3.2, see, (3.11).

In graph classification[4], the input data is now a *set of graphs* $\{\mathcal{G}_1, \cdots, \mathcal{G}_m\}$. Like in node classification each of these graphs is represented by an adjacency matrix and it is complemented by a set of node features. So we have m adjacency graphs $A_1, \cdots, A_m$ and m feature matrices of $X_1, X_2, \cdots, X_m$. If the number of vertices in $\mathcal{G}_k$ is n_k, A_k is of size $n_k \times n_k$ and X_k is $n_k \times d_k$. Assume at first that there is no mini-batching performed. Then for all layers except the last one, GCN works *exactly as in the node classification case with a single graph formed from the union of all graphs.* In other words, the matrix $\hat{A}$ in (5.4) can be viewed as a block diagonal matrix whose diagonal blocks are the $\hat{A}_k$'s obtained from each of the m graphs. Thus, the feature matrix $X^{(l)}$ at layer $l < L+1$ is of size $n \times d_l$ where $n = n_1 + n_2 + \cdots + n_m$ and d_l is the selected feature dimension for level l. The node features are simply concatenated in the node dimension.

$$\hat{A} = \begin{pmatrix} \boxed{\hat{A}_1} & & & \\ & \boxed{\hat{A}_2} & & \\ & & \boxed{\ddots} & \\ & & & \boxed{\hat{A}_m} \end{pmatrix} \in \mathbb{R}^{n\times n}, \qquad X^{(\ell)} = \begin{pmatrix} \boxed{X_1} \\ \boxed{X_2} \\ \boxed{\vdots} \\ \boxed{X_m} \end{pmatrix} \in \mathbb{R}^{n\times d_\ell}. \quad (5.5)$$

When $l = L$ in (5.4), the matrix $X^{(L+1)}$ obtained at the last layer will be $n \times d_{L+1}$. To classify graphs we need this matrix to be of size $m \times C$ where C is the number of classes and m the number of graphs. Indeed, we will need to embed each graph $\mathcal{G}_k$ into a (row) vector $z_{\mathcal{G}_k}$ of length C where the c-th component holds a probability that $\mathcal{G}_k$ is in class c. The dimension d_{L+1} should therefore be selected to be equal to C. In addition, at the last layer GCN applies a *'global pooling'* operation whereby all the node features for the same graph $\mathcal{G}_k$ are combined, e.g., averaged, to yield a single representation (a row of length C) for this graph, e.g., by applying a small `MLP` to create a C-dimensional graph encoding, which is a vector representation of the graph computed as

$$z_{\mathcal{G}_k} = \texttt{MLP}\Big(\underset{i\in\mathcal{V}_k}{\texttt{mean}}\; x_i^{(L+1)}\Big) \in \mathbb{R}^C, \qquad (5.6)$$

[4]`PyTorch Geometric`

where the `mean` operation is simply defined as

$$\underset{i \in \mathcal{V}_k}{\texttt{mean}}\, x_i^{(L+1)} = \frac{1}{|\mathcal{V}_k|} \sum_{i \in \mathcal{V}_k} x_i^{(L+1)}. \tag{5.7}$$

However, using the full set of all graphs is not practical. Instead, it is common to resort to mini-batching whereby a set of ν graphs is selected at random at each step.

A well-known dataset used to illustrate GNNs is the ENZYME dataset [78] which contains 600 molecules represented by graphs that translate their protein structure. An illustration of 4 such graphs is shown [5] in Figure 12. The dataset is divided into six classes corresponding to six main categories of enzymes that catalyze various reactions. These are Oxidoreductases, Transferases, Hydrolases, Lyases, Isomerases, and Ligases. Therefore, the problem is, given a molecule with a given structure (graph) and the nature of the compounds of the nodes in the graph (features) to determine what type of enzyme it is. It turns out that each node has a feature vector of length 3. The sizes of the 4 graphs are $n_1 = 5, n_2 = 14, n_3 = 12$, and $n_4 = 15$. If this set were a certain batch then for this batch $X^{(0)} \in \mathbb{R}^{46\times 3}$, $X^{(l)} \in \mathbb{R}^{46\times d_l}$, and $X^{(L+1)} \in \mathbb{R}^{46\times d_{L+1}}$, by setting $d_{L+1} = 6$ classes, and running (5.7) to get $Z \in \mathbb{R}^{4\times 6}$, i.e., 4 graphs and 6 classes.

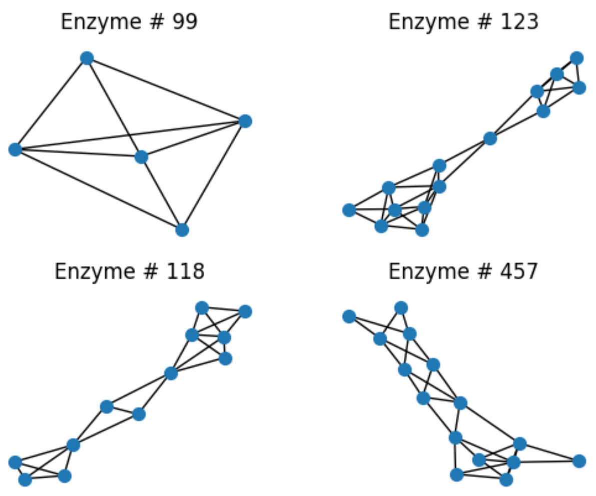

Fig. 12 Four sample graphs from the ENZYME data set.

Some of the limitations of GCNs are *(i) over-smoothing*, i.e., deep layers cause node features to converge, losing differentiation, which means that the representations for

[5]To show the graph, an embedding to 2-dimensional space is needed. The `NetworX` package in Pytorch was used for this.

all nodes in the graph can become very similar to one another; and *(ii) fixed aggregation*, i.e, it treats all neighbors equally, missing heterogeneous relationships. Therefore, building deeper GCN models can actually hurt performance, and information about local neighborhood structures is lost as more layers are added. On the positive side, we note that GCNs leverage graph sparsity, and since the node update equation is local, GCNs are naturally *parallelizable* with sparse matrix multiplications. This is efficiently implemented in libraries such as `DGL` [79], `PyG` [80] and `Spektra` [81]. Also, GCNs benefit from batch normalization and residual connections, e.g., see [82].

5.3 GraphSAGE (Graph Sample and Aggregation)

GraphSAGE is a scalable and efficient graph neural network architecture designed to handle large-scale graphs see [83]. Unlike traditional GNNs, which require the entire graph to be available during training, `GraphSAGE` uses a sampling-based approach to aggregate information from a fixed-size neighborhood around each node, making it suitable for inductive learning on large graphs. The main idea behind `GraphSAGE` is to aggregate features from a node's local neighborhood to update the node's feature representation. The key distinction is that `GraphSAGE` *samples a fixed-size subset of neighbors*, rather than using all neighbors, which makes it computationally efficient for large graphs. This process enables inductive learning, meaning that `GraphSAGE` can generalize to unseen parts of the graph, such as newly added nodes or edges.

In `GraphSAGE`, the feature representation $x_i^{(\ell)}$ of each node v_i is updated by aggregating information from its ℓ-hop neighbors. The update rule in the ℓ-th layer can be described as:

$$x_i^{(\ell+1)} = \mathtt{ReLU}\Big(W^{(\ell),T} \times \mathtt{AGGREGATE}\left(\{x_j^{(\ell)} : j \in \mathcal{N}(v_i)\} \cup \{x_i^{(\ell)}\}\right)\Big), \qquad (5.8)$$

where the `AGGREGATE` function combines the features of neighboring nodes. Common aggregation functions include mean, LSTM [15], and pooling.

To efficiently aggregate node features, `GraphSAGE` uses a sampling technique to limit the size of the neighborhood. For each node, a fixed number of neighbors are randomly sampled to aggregate their features, i.e., $\mathcal{N}(v_i) = \mathtt{sample}(v_i)$. This reduces the computational cost compared to using all the neighbors, especially in large graphs. It is also possible to use *importance sampling* to reduce variance, as in `FastGCN` [84]. The simplest mean-aggregation function, which computes the element-wise mean of the neighboring node features, is

$$\mathtt{AGGREGATE}\left(\{x_j : j \in \mathcal{N}(v_i)\}\right) = \frac{1}{|\mathcal{N}(v_i)|} \sum_{j \in \mathcal{N}(v_i)} x_j. \qquad (5.9)$$

As mentioned before, over-smoothing is an issue in GNNs with deeper layers, where the node-specific information becomes less important relative to the aggregated information from the neighbors during message passing. The updated node representation $x_i^{(\ell+1)}$ will depend more strongly on the messages aggregated from the neighbors,

$\{x_j^{(\ell)} : j \in \mathcal{N}(v_i)\}$, and less on the representation of the node itself from the previous layer, $x_i^{(\ell)}$. Therefore, a simple strategy to alleviate this over-smoothing issue is to use vector concatenations, or skip connections, whereby the information from the previous iteration of message passing is preserved during the update step. It turns out that this strategy helps also in improving the numerical stability of optimization. It has been adopted in `GraphSAGE`, see Algorithm 4.

Algorithm 4 `GraphSAGE` Embedding

```
Input: Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, input features $\{x_v, \forall v \in \mathcal{V}\}$, neighborhood function $\mathcal{N}$
Parameters: depth $K$, weight matrices $W^{(\ell)}, \forall \ell \in \{1, \cdots, K\}$
Output: $z_v, \forall v \in \mathcal{V}$        ▷ vector representations after message passing completion
1: for $\ell = 1, \cdots, K$ do
2:     for $v \in \mathcal{V}$ do
3:         $x_{\mathcal{N}(v)}^{(\ell)} \leftarrow$ AGGREGATE( $\{x_u^{(\ell-1)}, \forall u \in \mathcal{N}(v)\}$ )
4:         $x_v^{(\ell)} \leftarrow \mathtt{ReLU}\Big(W^{(\ell),T}\Big[x_v^{(\ell-1)} \parallel x_{\mathcal{N}(v)}^{(\ell)}\Big]\Big)$        ▷ $\parallel$ denotes concatenation
5:     end for
6:     $x_v^{(\ell)} \leftarrow x_v^{(\ell)}/\|x_v^{(\ell)}\|_2, \quad \forall v \in \mathcal{V}$        ▷ normalization
7: end forreturn $z_v \leftarrow x_v^{(K)}, \quad \forall v \in \mathcal{V}$
```

`GraphSAGE` is particularly effective for inductive learning tasks, where the model needs to generalize to unseen data (e.g., new nodes in the graph). By learning to aggregate information from a node's neighbors during training, `GraphSAGE` can apply the learned aggregation functions to unseen nodes during testing, even if those nodes were not present in the training graph. Some of the advantages of `GraphSAGE` are

1. *Scalability.* By using a sampling approach, `GraphSAGE` can handle large graphs that would be computationally expensive to process using traditional GNNs.
2. *Inductive Learning.* `GraphSAGE` can generalize to unseen nodes, making it effective for applications where the graph structure evolves over time.
3. *Flexibility.* The ability to choose different aggregation functions allows `GraphSAGE` to be tailored to specific applications and data types.

5.4 Graph Attention Networks (GAT)

GATs [85] are extensions of Graph Convolutional Networks (GCNs) that use attention mechanisms to weigh the importance of neighboring nodes during the aggregation process. In GCNs, `GraphSAGE` and other popular GNN architectures [55, 70], all neighbors are treated equally, i.e., the feature update of a node is typically the average of the features of its neighbors. However, GATs assign different attention scores to each neighbor, allowing the model to focus more on relevant neighbors. This enables a node to decide which neighboring nodes are more important when aggregating messages, i.e., it adaptively adjusts the influence of each neighboring node depending on the task.

In a Graph Attention Network, each node aggregates information from its neighbors by applying an attention mechanism. The attention mechanism computes a weight for each neighbor, which determines the degree to which the node's features are incorporated into its updated representation. Every node updates its representation by attending to its neighbors (input keys) using its own representation as the query. This generalizes the averaging or max-pooling operation of neighbors. GAT is inspired by the self-attention mechanism of the Transformer [17], and the attention mechanism in [86]. Given a graph with an adjacency matrix A and a node feature matrix X, the attention-based message-passing mechanism, which computes a weighted average of the transformed features of the neighboring nodes as the new representation of i, followed by an activation function σ, can be expressed as

$$X^{(\ell+1)} = \sigma\Big(A_\alpha X^{(\ell)} W^{(\ell)}\Big), \tag{5.10}$$

where the entries α_{ij} of A_α are the attention weights between nodes i and j, and are computed as:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k\,\in\,\mathcal{N}(i)} \exp(e_{ik})}, \tag{5.11}$$

in which the attention score e_{ij}, which indicates the importance of the features of the neighbor j to the node i, is calculated between each node i and its neighbors $j \in \mathcal{N}(i)$ through an attention function that uses a neural network with one layer, that is,

$$e_{ij} = \texttt{LeakyReLU}\Big(a^T \cdot \Big[W^{(\ell),T} x_i \parallel W^{(\ell),T} x_j\Big]\Big), \tag{5.12}$$

where $\parallel$ denotes vector concatenation, $a \in \mathbb{R}^{2d}$ is a learnable attention vector. The function `LeakyReLU` is a 'leaky' version of the `ReLU` activation function defined as $\texttt{LeakyReLU}(t) = \max\{\eta t, t\}$ where η is a small positive parameter. It ensures better gradient flow compared to `ReLU` by allowing small negative values. One can recognize the `softmax` function in (5.11) and so the coefficients α_{ij} form a probability distribution. It is is beneficial to generalize GAT to multihead attention, i.e., use separate attention heads and concatenate their outputs, see [85]:

$$x_i^{(\ell+1)} = \mathop{\Big\|}_{h=1}^{H} \texttt{LeakyReLU}\Big(\sum_{j\,\in\,\mathcal{N}(v_i)} \alpha_{ij}^{(h)} W_h^{(\ell),T} x_j^{(\ell)}\Big). \tag{5.13}$$

This multiheaded learning flexibility provides a mechanism that allows to attend to separate semantic characteristics of the problem. The results of these attention mechanisms are combined by concatenating the vectors together.

The Equation (5.12) computes the attention scores e_{ij} between all possible pairs of nodes, as if the graph is assumed to be fully connected. We need however to select only those which represent existing edges between nodes. To address this, after the `LeakyReLU` activation is applied to the attention scores, the attention scores are *masked* based on existing edges in the graph, which means that we only keep the scores that correspond to existing edges. This is achieved by using the adjacency matrix of the graph. It is worth mentioning that any attention model from deep learning can be used in computing the attention scores in (5.12), even though it has become the *de facto* practice, and is now implemented in most GAT libraries, e.g. see [79, 80, 87, 88]. The idea of adding attention to graphs, as in GATs, helps in increasing the representational capacity of the GNN model, in particular when there is some prior knowledge which indicates that some neighbors might be more informative than others. GATs have several unique properties, namely

1. *Self-Attention.* GATs use self-attention mechanisms, where each node's attention weights are learned based on its own features and the features of its neighbors.
2. *Adaptive Weights.* Unlike GCNs, where each neighbor contributes equally, GATs assign adaptive weights to each neighbor, depending on their relevance.
3. *Scalability.* GATs allow for parallel computation of attention coefficients, making them more scalable than other attention-based models.
4. *Inductive Learning.* Like GCNs, GATs can be applied inductively to unseen nodes during inference.

Some of the limitations of GATs are *(i) computational overhead*, i.e., increased complexity due to attention computations; and *(ii) sparse data*, i.e., GATs struggle with graphs having few connections.

5.5 Graph Transformer

Graph Transformers [89] aim to extend the success of Transformer models in natural language processing and computer vision to graph-structured data. In these models, the self-attention mechanism is adapted to account for the graph's structure. Each node in the graph attends to its neighbors, as well as potentially distant nodes, using the attention mechanism, thus learning more flexible and expressive node representations.

The Graph Transformer `GT` follows the same transformer architecture of Vaswani et al. [17], which is explained in Section 4 for large language models, see (4.4)–(4.5) and Table 2. It generalizes and extends the success of Transformer models to arbitrary graph-structured data, see [89, 90]. It also generalizes the word `cos/sin` positional encoding `PE` from sequences to graphs, i.e., node ordering with *Laplacian eigenvectors*. The `GT` model is constructed by stacking $\mathtt{GT}_\ell$ layers, as in (4.3), i.e.,

$$X_L = \mathtt{GT}(X_0) = (\mathtt{GT}_L \circ \cdots \circ \cdots \circ \mathtt{GT}_1)(X_0), \tag{5.14}$$

where $\mathtt{GT}_\ell$ is the Graph Transformer block ℓ.

The transformation function class (post-`Norm`) $\mathtt{GT}_\ell$ of each graph transformer block ℓ, for an arbitrary graph with an adjacency matrix A, is represented by the equations[6]

$$X_\ell = \mathtt{GT}_\ell(X_{\ell-1}) \quad \triangleq \quad \begin{cases} Y_\ell = \mathtt{Norm}(\ X_{\ell-1} + \mathtt{gATT}(X_{\ell-1})\ ), & (5.15) \\ X_\ell = \mathtt{Norm}(\ Y_\ell + \mathtt{MLP}(Y_\ell)\ ), & (5.16) \end{cases}$$

[6]For implementation details, see e.g., the Deep Graph Library.

where the graph multihead attention function gATT is defined as,

$$\mathtt{gATT}(X_{\ell-1}) = \overset{H}{\underset{h=1}{\|}} \; (\; \mathtt{gHA}_h \;) W_O, \tag{5.17}$$

in which the h-th graph head attention $\mathtt{gHA}_h$ is (see [90]):

$$\mathtt{gHA}_h = \mathtt{softmax}\Big(A_g \odot \frac{(X_{\ell-1}W_{Q,h}^{(\ell)})(X_{\ell-1}W_{K,h}^{(\ell)})^T}{\sqrt{d_{\mathtt{head}}}} \Big) (X_{\ell-1}W_{V,h}^{(\ell)}). \tag{5.18}$$

The Norm function in (5.15,5.16) can either be LayerNorm [50–52] or BatchNorm [91], and the modified adjacency matrix A_g in (5.18) is such that, see [90],

$$A_g(i,j) = \begin{cases} A(i,j) & \text{if nodes } i,j \text{ are connected,} \\ 1 & \text{for } i=j \text{ to enforce self-loops,} \\ -\infty & \text{if nodes } i,j \text{ are not connected,} \end{cases}$$

where only the attention scores between connected nodes are computed via the point-wise multiplication with A_g. There are different graph attention functions for (5.18), e.g., GraphiT in [92] uses a kernel graph attention, i.e., A_g is some kernel on the graph; whereas the Hadamard graph attention in [93] has a different formulation than (5.18),

$$\mathtt{gHA}_h = A_g \odot \mathtt{softmax}\Big(\frac{(X_{\ell-1}W_{Q,h}^{(\ell)})(X_{\ell-1}W_{K,h}^{(\ell)})^T}{\sqrt{d_{\mathtt{head}}}} \Big) (X_{\ell-1}W_{V,h}^{(\ell)}), \tag{5.19}$$

with A_g being the original adjacency matrix. There are alternative formulations, e.g. see [94].

	Post-Norm	Pre-Norm
① gATT	$Y_\ell = \mathtt{Norm}(\; X_{\ell-1} + \mathtt{gATT}(X_{\ell-1}) \;)$	$Y_\ell = X_{\ell-1} + \mathtt{gATT}(\; \mathtt{Norm}(X_{\ell-1}) \;)$
② MLP	$X_\ell = \mathtt{Norm}(\; Y_\ell + \mathtt{MLP}(Y_\ell) \;)$	$X_\ell = Y_\ell + \mathtt{MLP}(\; \mathtt{Norm}(Y_\ell) \;)$

Table 3 Post-Norm and Pre-Norm with Skip Connections.

Positional Encoding. A naive self-attention model sees only a bag-of-nodes, i.e., a set of feature vectors in X, which is invariant to ordering. Therefore, it is important to incorporate spatial or structural information into graph-based models, such as Graph Transformers. Positional encoding helps the model understand the relative positions of nodes within a graph, and since graphs don't have a natural ordering like sequences, encoding the structural information and relationships between the nodes is important. *Laplacian positional encoding* uses the Laplacian matrix of the graph and its spectral properties to inject structural (positional) information into the model, e,g., see [95–97].

5.6 Graph-Transformer for WSI Classification – an example

In this section, we introduce an example where graph transformers are used in the biomedical field. Computational pathology is the study and analysis of pathology data using machine learning techniques, and it involves analyzing medical images, such as tissue slides, to assist in diagnosing diseases like cancer, and predicting patient outcomes. A Whole Slide Image (WSI) is a digital representation of an entire tissue slide, and the sheer size of a single WSI can exceed a gigabyte, so traditional image analysis techniques may not be able to efficiently process all this data.

Modern methods rely on systematic breakdown of WSIs into a large number of smaller non-overlapping regions like tiles (e.g., 512×512 image patches). A graph-based approach treats these regions as nodes in a graph, where the edges between nodes can represent spatial relationships or interactions between neighboring tissue regions; and the nodes v_i usually contain feature information extracted from the image patches, or deep learning-based embeddings. For a classification task, we follow these steps:

1) Graph construction and embeddings. Given that WSIs contain thousands of patches, it is computationally infeasible to process raw image pixels directly. So, usually a *ResNet*[7] model is trained to extract meaningful feature embeddings from WSI patches, which serve as inputs for the graph-based representation. Each node v_i is associated with a feature vector $x_i \in \mathbb{R}^d$, where d is the embedding dimension extracted from *ResNet*. The resulting graph representation can be expressed as a node feature matrix $X \in \mathbb{R}^{n\times d}$. The adjacency matrix A is used in graph convolutional operations to propagate information across connected nodes, thus allowing the model to aggregate context from nearby patches. This enables the model to learn meaningful relationships between tissue structures rather than treating each patch independently. The node features are passed through multiple GCN layers, as in Equation (5.4), refining their representations by incorporating information from adjacent patches.

2) Graph Transformer. The graph transformer uses *self-attention* to learn dependencies between different nodes (i.e., patches) in the graph, which allows the model to focus on relevant regions of the WSI, thus allowing the model to recognize both local and global patterns in the tissue. The attention output for node i is a weighted sum of the value vectors v_j from its neighbors $\mathcal{N}(i)$, as in Equations (5.15)-(5.16).

3) Classification. After processing through multiple layers of graph transformers, each node's feature vector x_i^L will encode rich contextual information about the patches. The final prediction (such as tumor classification or segmentation) can be derived from these learned representations. If the goal is to classify the entire WSI, we can aggregate the node features (patch features) into a global representation by graph pooling, as in Equation (5.6).

[7]Resnet, or residual neural network, which was originally designed for computer vision, is a deep neural network characterized by additional connections that skip multiple layers in the network, see Section 4.2.4.

4) Min-cut Pooling. For scalability sake and to make this process computationally efficient, usually a *min-cut pooling*[8] layer is introduced between the graph and transformer layers. This pooling operation reduces the number of nodes while preserving the most informative ones, ensuring that the model remains scalable to large WSIs.

To conclude, this step-by-step approach allows efficient and robust analysis of whole slide images, enhancing tasks like disease detection and classification in computational pathology, e.g. see [98–100].

6 Conclusion: The next chapter in NLA

We cannot overstate the important role that Numerical Linear Algebra is playing in the development and deployment of AI. NLA provides software packages used internally and is contributing key ideas and algorithms to reduce computational time. The matrix and tensor formalisms allow to easily express the various transformations employed in deep learning. One of the key tools in optimizing models is Back-Propagation, which can be efficiently carried out because it amounts to a sequence of matrix-matrix products. In fact, one might argue that tools from NLA contribute the most to the improvement of algorithms for training LLMs and for the subsequent inference. Low-rank approximations are heavily exploited for example as are tensor computations in low-precision arithmetic.

While these successes of NLA are noteworthy, the megatrend of AI poses an important question to the Numerical Linear Algebra and Numerical Analysis communities, namely: how should its members react to it? AI is proving to be unusually disruptive in academia. For example, given the excitment generated by AI and the availability of high-paying jobs in AI-related fields most students in computer science and elsewhere, are interested in working in AI, at the exclusion of other topics. On the educational side, students are now able to use LLMs to solve most questions given in homeworks and exams and this makes testing rather challenging. Meanwhile, AI research is promoted by favorable funding at the expense of traditional fields that have taken decades to mature. If researchers do not adapt, their work may become irrelevant or at least generate little interest.

A reasonable goal in this environment is not to abandon intellectual innovations in classical Linear Algebra but to also participate actively in Deep Learning research. However, this is not an easy task due to a number of factors. AI as a field has many differences with NLA, in terms of culture, notation, emphasis, etc. The community is huge and the field diverse, embracing algorithms for Neural Networks on the one hand, and information theory, or graph methods on the other. The publication culture is fast-paced and completely different from what we see in NLA. The main point that we want to make is that *in spite of all these challenges, the NLA community cannot afford to ignore the AI wave. It should fully participate in its advancement.*

[8] Min-Cut Pooling is a method used in deep learning to down-sample graph-structured data. The pooling operation is defined as $X_{\text{pool}} = S^T X$ where $S \in \mathbb{R}^{n\times K}$ is the assignment matrix and K is the number of clusters formed during the pooling process, with $K \ll n$. To adjust the graph structure after pooling, the adjacency matrix A is updated based on the new assignment matrix S as $A_{\text{pool}} = S^T AS$. The key idea of min-cut pooling is to reduce the size of the graph while preserving the important structure of the data.

References

[1] Aschenbrenner, L.: Situational awareness: The decade ahead. Series: Situational Awareness (2024)

[2] Suleyman, M., Bhaskar, M.: The Coming Wave: Technology, Power, and the Twenty-first Century's Greatest Dilemma, p. 352. Crown, New-York (2023)

[3] Hu, E.J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W.: LoRa: Low-rank adaptation of large language models. arXiv preprint arXiv:2106.09685 (2021)

[4] Katharopoulos, A., Vyas, A., Pappas, N., Fleuret, F.: Transformers are rnns: Fast autoregressive transformers with linear attention. arXiv preprint arXiv:2006.16236 (2020)

[5] Turing, A.: Computing machinery and intelligence. Mind **LIX**, 433–460 (October 1950)

[6] Rosenblatt, F.: The perceptron: a probabilistic model for information storage and organization in the brain. Psychol. Rev. **65**, 386–408 (1958) https://doi.org/10.1037/h0042519.PMID:13602029

[7] Moor, J.: The dartmouth college artificial intelligence conference: The next fifty years. Ai Magazine **27**(4), 87–87 (2006)

[8] Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back-propagating errors. nature **323**(6088), 533–536 (1986)

[9] Cortes, C., Vapnik, V.: Support-vector networks. Machine Learning **20**(3), 273–297 (1995)

[10] Bishop, C.M.: Pattern Recognition and Machine Learning. Information Science and Statistics. Springer, New-York (2006)

[11] Belkin, M., Niyogi, P.: Laplacian eigenmaps and spectral techniques for embedding and clustering (2002). citeseer.ist.psu.edu/belkin01laplacian.html

[12] Roweis, S., Saul, L.: Nonlinear dimensionality reduction by locally linear embedding. Science **290**, 2323–2326 (2000)

[13] Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. Advances in neural information processing systems **25** (2012)

[14] Medsker, L.R., Jain, L., *et al.*: Recurrent neural networks. Design and Applications **5**(64-67), 2 (2001)

[15] Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput **9**(8), 1735–1780 (1997)

[16] Dey, R., Salem, F.M.: Gate-variants of gated recurrent unit (GRU) neural networks. In: 2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS), pp. 1597–1600 (2017). IEEE

[17] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. Advances in neural information processing systems **30** (2017)

[18] Moore, G.E.: Cramming more components onto integrated circuits. Electronics **38**(8), 114–117 (1965)

[19] Kaplan, J., McCandlish, S., Henighan, T., Brown, T.B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., Amodei, D.: Scaling laws for neural language models. arXiv preprint arXiv:2001.08361 (2020)

[20] Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., Casas, D.d.L., Hendricks, L.A., Welbl, J., Clark, A., et al.: Training compute-optimal large language models. arXiv preprint arXiv:2203.15556 (2022)

[21] Cybenko, G.: Approximation by superpositions of a sigmoidal function. Mathematics of control, signals and systems **2**(4), 303–314 (1989) https://doi.org/10.1007/BF02551274

[22] Griewank, A., Walther, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. SIAM, Philadelphia (2008)

[23] Averick, B.A., More, J.J., Bischof, C.H., Carle, A., Griewank, A.: Computing large sparse Jacobian matrices using automatic differentiation. SIAM Journal on Scientific Computing **15**, 285–294 (1994)

[24] Griewank, A.: On automatic differentiation. In: Iri, M., Tanabe, K. (eds.) Mathematical Programming: Recent Developments and Applications, pp. 83–108. Kluwer acad. publ., Borwell, MA (1989)

[25] Murphy, K.P.: Probabilistic Machine Learning: an Introduction. MIT press, Cambridge, MA (2022)

[26] Bubeck, S.: Convex optimization: Algorithms and complexity. Found. Trends Mach. Learn. **8**(3–4), 231–357 (2015) https://doi.org/10.1561/2200000050

[27] Gower, R.M., Loizou, N., Qian, X., Sailanbayev, A., Shulgin, E., Richtárik, P.: Sgd: General analysis and improved rates. In: International Conference on Machine Learning, pp. 5200–5209 (2019). PMLR

[28] Hardt, M., Recht, B., Singer, Y.: Train faster, generalize better: Stability of

stochastic gradient descent. In: International Conference on Machine Learning, pp. 1225–1234 (2016). PMLR

[29] Robbins, H., Monro, S.: A stochastic approximation method. The annals of mathematical statistics, 400–407 (1951)

[30] Ljung, L.: Analysis of recursive stochastic algorithms. IEEE transactions on automatic control **22**(4), 551–575 (1977)

[31] Duchi, J., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. Journal of machine learning research **12**(7) (2011)

[32] Li, H., Xu, Z., Taylor, G., Studer, C., Goldstein, T.: Visualizing the loss landscape of neural nets. Advances in neural information processing systems **31** (2018)

[33] Wu, L., Zhu, Z., E, W.: Towards understanding generalization of deep learning: Perspective of loss landscapes. arXiv preprint arXiv:1706.10239 (2017)

[34] Zhang, C., Bengio, S., Hardt, M., Recht, B., Vinyals, O.: Understanding deep learning (still) requires rethinking generalization. Communications of the ACM **64**(3), 107–115 (2021)

[35] Zhou, P., Feng, J., Ma, C., Xiong, C., Hoi, S.C.H., E, W.: Towards theoretically understanding why sgd generalizes better than adam in deep learning. Advances in Neural Information Processing Systems **33**, 21285–21296 (2020)

[36] Neyshabur, B., Tomioka, R., Srebro, N.: In search of the real inductive bias: On the role of implicit regularization in deep learning. arXiv preprint arXiv:1412.6614 (2014)

[37] Zhang, A., Lipton, Z.C., Li, M., Smola, A.J.: Dive Into Deep Learning. Cambridge University Press, Cambridge, UK (2023). https://D2L.ai

[38] Gao, P., Geng, S., Qiao, Y., Wang, X., Dai, J., Li, H.: Scalable Transformers for Neural Machine Translation (2021). https://arxiv.org/abs/2106.02242

[39] Dehghani, M., Djolonga, J., Mustafa, B., Padlewski, P., Heek, J., Gilmer, J., Steiner, A.P., Caron, M., Geirhos, R., Alabdulmohsin, I., *et al.*: Scaling vision transformers to 22 billion parameters. In: International Conference on Machine Learning, pp. 7480–7512 (2023). PMLR

[40] Radford, A.: Improving language understanding by generative pre-training. Online Preprint (2018)

[41] Kenton, J.D.M.-W.C., Toutanova, L.K.: Bert: Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of naacL-HLT, vol. 1,

p. 2 (2019). Minneapolis, Minnesota

[42] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., *et al.*: Language models are unsupervised multitask learners. OpenAI blog **1**(8), 9 (2019)

[43] Brown, T.B.: Language models are few-shot learners. arXiv preprint arXiv:2005.14165 (2020)

[44] Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al.: Llama: Open and efficient foundation language models. corr, abs/2302.13971, 2023. doi: 10.48550. arXiv preprint arXiv.2302.13971 (2023)

[45] GenAI, M.: Llama 2: Open foundation and fine-tuned chat models. arXiv preprint arXiv:2307.09288 (2023)

[46] Elhage, N., Nanda, N., Olsson, C., Henighan, T., Joseph, N., Mann, B., Askell, A., Bai, Y., Chen, A., Conerly, T., DasSarma, N., Drain, D., Ganguli, D., Hatfield-Dodds, Z., Hernandez, D., Jones, A., Kernion, J., Lovitt, L., Ndousse, K., Amodei, D., Brown, T., Clark, J., Kaplan, J., McCandlish, S., Olah, C.: A mathematical framework for transformer circuits. Transformer Circuits Thread (2021). https://transformer-circuits.pub/2021/framework/index.html

[47] Bapna, A., Chen, M.X., Firat, O., Cao, Y., Wu, Y.: Training deeper neural machine translation models with transparent attention. arXiv preprint arXiv:1808.07561 (2018)

[48] Pascanu, R., Mikolov, T., Bengio, Y.: On the difficulty of training recurrent neural networks. In: International Conference on Machine Learning, pp. 1310–1318 (2013). Pmlr

[49] He, K., Zhang, X., Ren, S., Sun, J.: Identity mappings in deep residual networks. In: Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV 14, pp. 630–645 (2016). Springer

[50] Lei Ba, J., Kiros, J.R., Hinton, G.E.: Layer normalization. ArXiv e-prints, 1607 (2016)

[51] Xu, J., Sun, X., Zhang, Z., Zhao, G., Lin, J.: Understanding and improving layer normalization. Advances in neural information processing systems **32** (2019)

[52] Xiong, R., Yang, Y., He, D., Zheng, K., Zheng, S., Xing, C., Zhang, H., Lan, Y., Wang, L., Liu, T.: On layer normalization in the transformer architecture. In: International Conference on Machine Learning, pp. 10524–10533 (2020). PMLR

[53] Monti, F., Frasca, F., Eynard, D., Mannion, D., Bronstein, M.M.: Fake news detection on social media using geometric deep learning. arXiv preprint arXiv:1902.06673 (2019)

[54] Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907 (2016)

[55] Duvenaud, D.K., Maclaurin, D., Iparraguirre, J., Bombarell, R., Hirzel, T., Aspuru-Guzik, A., Adams, R.P.: Convolutional networks on graphs for learning molecular fingerprints. Advances in neural information processing systems **28** (2015)

[56] Gilmer, J., Schoenholz, S.S., Riley, P.F., Vinyals, O., Dahl, G.E.: Neural message passing for quantum chemistry. In: International Conference on Machine Learning, pp. 1263–1272 (2017). PMLR

[57] Monti, F., Boscaini, D., Masci, J., Rodola, E., Svoboda, J., Bronstein, M.M.: Geometric deep learning on graphs and manifolds using mixture model cnns. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 5115–5124 (2017)

[58] Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W.L., Leskovec, J.: Graph convolutional neural networks for web-scale recommender systems. In: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 974–983 (2018)

[59] Schlichtkrull, M., Kipf, T.N., Bloem, P., Van Den Berg, R., Titov, I., Welling, M.: Modeling relational data with graph convolutional networks. In: The Semantic Web: 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, Proceedings 15, pp. 593–607 (2018). Springer

[60] Chami, I., Wolf, A., Juan, D.-C., Sala, F., Ravi, S., Ré, C.: Low-dimensional hyperbolic knowledge graph embeddings. arXiv preprint arXiv:2005.00545 (2020)

[61] Cranmer, M.D., Xu, R., Battaglia, P., Ho, S.: Learning symbolic physics with graph networks. arXiv preprint arXiv:1909.05862 (2019)

[62] Sanchez-Gonzalez, A., Godwin, J., Pfaff, T., Ying, R., Leskovec, J., Battaglia, P.: Learning to simulate complex physics with graph networks. In: International Conference on Machine Learning, pp. 8459–8468 (2020). PMLR

[63] Parisot, S., Ktena, S.I., Ferrante, E., Lee, M., Guerrero, R., Glocker, B., Rueckert, D.: Disease prediction using graph convolutional networks: application to autism spectrum disorder and alzheimer's disease. Medical image analysis **48**, 117–130 (2018)

[64] Gori, M., Monfardini, G., Scarselli, F.: A new model for learning in graph domains. In: Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005., vol. 2, pp. 729–734 (2005). IEEE

[65] Scarselli, F., Gori, M., Tsoi, A.C., Hagenbuchner, M., Monfardini, G.: The graph neural network model. IEEE transactions on neural networks **20**(1), 61–80 (2008)

[66] Atwood, J., Towsley, D.: Diffusion-convolutional neural networks. Advances in neural information processing systems **29** (2016)

[67] Bronstein, M.M., Bruna, J., LeCun, Y., Szlam, A., Vandergheynst, P.: Geometric deep learning: going beyond euclidean data. IEEE Signal Processing Magazine **34**(4), 18–42 (2017)

[68] Hamilton, W.L.: Graph Representation Learning. Synthesis Lectures on Artificial Intelligence and Machine Learning, vol. 14, pp. 1–159. Morgan & Claypool Publishers, McGill University (2020)

[69] LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86**(11), 2278–2324 (1998)

[70] Xu, K., Hu, W., Leskovec, J., Jegelka, S.: How powerful are graph neural networks? arXiv preprint arXiv:1810.00826 (2018)

[71] Morris, C., Ritzert, M., Fey, M., Hamilton, W.L., Lenssen, J.E., Rattan, G., Grohe, M.: Weisfeiler and leman go neural: Higher-order graph neural networks. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 33, pp. 4602–4609 (2019)

[72] Maron, H., Ben-Hamu, H., Shamir, N., Lipman, Y.: Invariant and equivariant graph networks. arXiv preprint arXiv:1812.09902 (2018)

[73] Maron, H., Ben-Hamu, H., Serviansky, H., Lipman, Y.: Provably powerful graph networks. Advances in neural information processing systems **32** (2019)

[74] Chen, Z., Villar, S., Chen, L., Bruna, J.: On the equivalence between graph isomorphism testing and function approximation with gnns. Advances in neural information processing systems **32** (2019)

[75] Bouritsas, G., Frasca, F., Zafeiriou, S., Bronstein, M.M.: Improving graph neural network expressivity via subgraph isomorphism counting. IEEE Transactions on Pattern Analysis and Machine Intelligence **45**(1), 657–668 (2022)

[76] Bodnar, C., Frasca, F., Otter, N., Wang, Y., Lio, P., Montufar, G.F., Bronstein, M.: Weisfeiler and lehman go cellular: Cw networks. Advances in neural information processing systems **34**, 2625–2640 (2021)

[77] Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. ArXiv e-prints (2017) arXiv:1609.02907 [cs.LG]

[78] Borgwardt, K.M., Ong, C.S., Schönauer, S., Vishwanathan, S.V.N., Smola, A.J., Kriegel, H.-P.: Protein Function Prediction via Graph Kernels. Data retrieved from https://ls11-www.cs.tu-dortmund.de/staff/morris/graphkerneldatasets (2005)

[79] Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., et al.: Deep graph library: A graph-centric, highly-performant package for graph neural networks. arxiv. Learning (2019)

[80] Fey, M., Lenssen, J.E.: Fast graph representation learning with pytorch geometric. arXiv preprint arXiv:1903.02428 (2019)

[81] Grattarola, D., Alippi, C.: Graph neural networks in tensorflow and keras with spektral [application notes]. IEEE Computational Intelligence Magazine **16**(1), 99–106 (2021)

[82] Bresson, X., Laurent, T.: Residual gated graph convnets. arXiv preprint arXiv:1711.07553 (2017)

[83] Hamilton, W., Ying, Z., Leskovec, J.: Inductive representation learning on large graphs. Advances in neural information processing systems **30** (2017)

[84] Chen, J., Ma, T., Xiao, C.: Fastgcn: Fast learning with graph convolutional networks via importance sampling. ICLR (2018)

[85] Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y.: Graph attention networks. arXiv preprint arXiv:1710.10903 (2017)

[86] Bahdanau, D.: Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473 (2014)

[87] Dwivedi, V.P., Joshi, C.K., Luu, A.T., Laurent, T., Bengio, Y., Bresson, X.: Benchmarking graph neural networks. Journal of Machine Learning Research **24**(43), 1–48 (2023)

[88] Gordic, A.: pytorch-gat (2020). https://github.com/gordicaleksa/pytorch-GAT

[89] Cai, D., Lam, W.: Graph transformer for graph-to-sequence learning. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 34, pp. 7464–7471 (2020)

[90] Dwivedi, V.P., Bresson, X.: A generalization of transformer networks to graphs, 2020. arXiv preprint arXiv:2012.09699 (2021)

[91] Ioffe, S.: Batch normalization: Accelerating deep network training by reducing

internal covariate shift. arXiv preprint arXiv:1502.03167 (2015)

[92] Mialon, G., Chen, D., Selosse, M., Mairal, J.: Graphit: Encoding graph structure in transformers. arXiv preprint arXiv:2106.05667 (2021)

[93] He, X., Hooi, B., Laurent, T., Perold, A., LeCun, Y., Bresson, X.: A generalization of vit/mlp-mixer to graphs. In: International Conference on Machine Learning, pp. 12724–12745 (2023). PMLR

[94] Ying, C., Cai, T., Luo, S., Zheng, S., Ke, G., He, D., Shen, Y., Liu, T.-Y.: Do transformers really perform badly for graph representation? Advances in neural information processing systems **34**, 28877–28888 (2021)

[95] Su, J., Ahmed, M., Lu, Y., Pan, S., Bo, W., Liu, Y.: Roformer: Enhanced transformer with rotary position embedding. Neurocomputing **568**, 127063 (2024) https://doi.org/10.1016/j.neucom.2023.127063

[96] Park, W., Chang, W., Lee, D., Kim, J., Hwang, S.-w.: Grpe: Relative positional encoding for graph transformer. arXiv preprint arXiv:2201.12787 (2022)

[97] Dwivedi, V.P., Luu, A.T., Laurent, T., Bengio, Y., Bresson, X.: Graph neural networks with learnable structural and positional representations. arXiv preprint arXiv:2110.07875 (2021)

[98] Zheng, Y., Gindra, R.H., Green, E.J., Burks, E.J., Betke, M., Beane, J.E., Kolachalama, V.B.: A graph-transformer for whole slide image classification. IEEE transactions on medical imaging **41**(11), 3003–3015 (2022)

[99] Hemker, K., Simidjievski, N., Jamnik, M.: Healnet–hybrid multi-modal fusion for heterogeneous biomedical data. arXiv preprint arXiv:2311.09115 (2023)

[100] Shi, Z., Zhang, J., Kong, J., Wang, F.: Integrative graph-transformer framework for histopathology whole slide image representation and classification. In: International Conference on Medical Image Computing and Computer-Assisted Intervention, pp. 341–350 (2024). Springer