Large Language Models (LLMs)

Simply but faithfully explained

An Introduction

Yves Bernas, M.Sc. (ETH Zürich)*

Version 12

First published: 12 December 2025.
Revised version: March 2026

(Each question answered raises a new one)

Abstract

Large Language Models (LLMs) are fascinating in their use as well as in their mechanism. Their architecture and principles are not limited to language but find application in many other areas like image, video, music generation, robotics, prediction of protein properties as well as many other less known uses or still to be discovered applications.
Any stream of data that has some structure -and there are many in nature- can be modeled by it, enabling prediction or generation. This is what makes it so powerful, and the center of current AI. Explaining it can get complicated and obscure.

The purpose of this paper is to explain LLMs with standard school-level mathematics, without misleading simplifications.

LLMs combine Transformer attention layers with forward neural networks (FNNs). They are trained on a very large corpus of human text from the Internet and other sources.

In **Chapter 1**, we begin with a Bengio-style language model [2] to discover the role of embeddings and FNNs, and to establish the basic principle of prediction.

In **Chapter 2**, we focus on forward neural networks and how they can be used for prediction. We favor a *feature-recognition* interpretation over the purely geometric one (rotation, shearing, folding, warping): the geometric view is a consequence; the feature view is closer to an explanation.

In **Chapter 3**, we focus on Transformers [5] and their attention mechanism—the breakthrough that enabled language models to scale to large context windows (for which the "T" in ChatGPT stands for).

***

*Disclaimer: [This article is a personal publication and is not affiliated with or endorsed by ETH Zürich.]

Contents

1 Language Models

1.1 The Central Mechanism of Language Models

Before we begin, I want to mention the central mechanism of language models.

The central mechanism of language models can be stated in a few precise key ideas:

- During training on a large text corpus, the model learns to classify prefix sequences into token classes. For example, the class "you" consists of all sequences that preceded the token "you" in the corpus ("I love", "I know", etc.—in practice much longer sequences). This happens for all word-tokens in the vocabulary. Each word-token is thus associated with its class of prefixes.
- This classification is implemented through multi-pattern matching by forward neural networks.
- At inference (use of the trained network), each prefix sequence produces a pattern-matching score (logits) for every possible token class.
- These scores are interpreted as likelihood scores via a Boltzmann–Gibbs assumption: stronger matches are seen as indicating more likely outcomes (alike $\approx$ likely).
- They are then converted into probabilities via Softmax.
- Next-token prediction then simply consists of selecting (or sampling from) the token whose class has the highest probability.

1.2 Indexing the vocabulary

Because an LLM is a mathematical and computer-implemented system, we have to convert words to numbers. All words encountered in our language are each assigned first an integer value. Common English contains let's say roughly 70000 words, so each word will be assigned an integer value between, e.g., 1 and 70000.

In fact, it is a little more complicated; not words per se are indexed, but bits of words, or tokens, like prefixes, roots, and suffixes, etc. For example, "redundant", "redundancy" are not indexed as such but probably as "redundan", "t", and "cy". We will sometime talk about words for simplicity but what is really meant is tokens.

1.3 Embeddings

In reality, a word or a token is not represented inside the forward neural network by its integer index, but by a high-dimensional vector called an embedding. Embeddings are learned during training and then reused at inference. These vectors allow semantic (meaning) and syntactic (grammar) mathematical expression/representation of the words/tokens, and are the mathematical objects that the forward neural network actually works with. It is precisely this "splitting" or learned decomposition in more generic features that actually allows the prediction to work so well.

They can be seen as distributed representation of the tokens in the sense expressed in 2.1.2.1 below, which Bengio applied to the token itself. Embeddings, a distributed representation of symbolic data was disclosed by Hinton(1986)[1], but Bengio(2003)[2] was the first one to use it for a language model and it his central to his model and contribution.

We will come back to them later in this article. For the moment you may simply think of each word/token as having a unique vector identity.

1.4 Next Word prediction

The core function of an LLM is next-word prediction. The basic task an LLM accomplishes is -given an input sequence, i.e., an incomplete sentence- to predict the most likely next word, as in autocomplete add-ons available in text editors, like the one I am using right now. This input sequence has a maximal length: the context window. In modern LLMs like ChatGPT, the context window is roughly 100,000 words or 128,000 tokens.

1.4.1 Shannon N-gram

The first word predictor was the N-gram, invented by C. Shannon in 1948 and proposed in: "A Mathematical Theory of Communication"[3]. It was not neural; it worked by making statistics through counting how many times words occurred after a sequence of n words in a corpus of text, which is akin to the basic concept of word prediction. For this reason, it can be regarded as the great ancestor of modern LLMs. Its main drawback was that the computing power grew exponentially with n, the size of the input sequences, and that there could be no prediction for sequences that had not been encountered in the corpus.

1.4.2 Next word neural predictor, autocomplete

It is Yoshua Bengio et al. who invented the first word predictor based on forward neural networks back in 2003, published in the article "A Neural Probabilistic Language Model"[2]. It alleviated the drawbacks of the Shannon N-grams: Because of its neural implementation, it could correctly predict the next word for inputs that had not been encountered in the corpus and also required less computing. Because of the similarity of the task with the Shannon N-grams, this model is sometimes called the neural n-gram.

This system encapsulates a fundamental idea of Language Models and can be regarded as conceptually the real ancestor of LLMs. Its only drawback is a limited context window — the concatenation of embeddings makes scaling impractical beyond 5 to 10 words (perhaps 100 with today's hardware). This suffices for autocomplete but not for Large Language Models.

How does this prediction work? In Bengio's model, it is implemented using i.a. a forward neural network trained on a large corpus of text. We will not explain in this first chapter how a forward neural network is trained, i.e, backpropagation, rather we are going to explain how it works when used i.e, inference.

1.4.3 A neural network cannot approximate a non-deterministic relation (multi-value)

One often hears that a Forward Neural Network is a Universal Function Approximator. This was proven by G. Cybenko (1989) for the sigmoid function as activation function and K. Hornik et Al. (1989) who generalized it to FNNs with other activation functions.

However this feature cannot help us directly in modeling language:

Prediction means: using the past, predict the future. In language models it amounts to: input a sequence of words/tokens (past) X, predict the next word Y.
Prediction is a function: $Y = f(X)$, $next\_word = f(input\_sentence\_chunk)$

However, in language, the relation between a word sequence and the following word is not a function in the classical sense, since the same sequence may be followed by several different words in the corpus. For example: "I love" is not always followed by "you", but sometimes by "me", "him", "her", "cats", "dogs", etc...

A neural network, though a Universal Function Approximator, cannot directly approximate such a non-deterministic relation (non-single value) because it is not a function. To assume it is a common misunderstanding.

1.4.4 The trick: estimate probabilities of each possible outcome

But there is a way out of this which was inspired by John S. Bridle[4] in 1990, inventively combining known mathematical tools (Logits/Softmax, Maximum Likelihood Estimation, Neural Networks) for classification:

It is to estimate the probability of each possible outcome and then select the one having the highest probability.

This is one of the key two insights to understand LLMs.

In the world of language it means to estimate the probability for each token/word over the entire vocabulary to be the next word. This vocabulary is huge (128 000 tokens), so it means estimating 128 000 probabilities.

It was one of Bengio's main insights to use this idea for word prediction in his model. This idea was further used in LLMs.

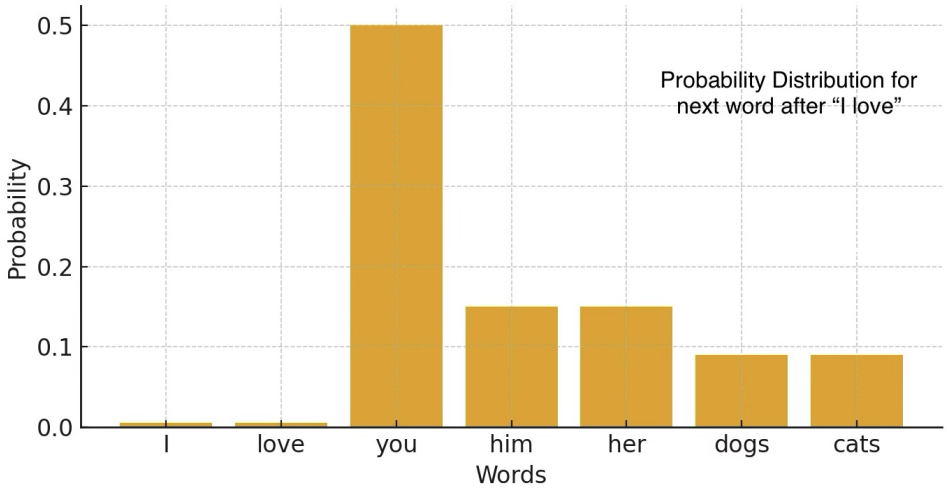
For example, let us say if our vocabulary only comprised :

“I”, “love“, “you”,”him”, “her”, ”dogs” and “cats”,

Let us say the network is trained on a corpus containing only this vocabulary. When we would input the sequence “I love”, it would output the discrete probability distribution over each word of the vocabulary.

$$(0.005, 0.005, 0.5, 0.15, 0.15, 0.09, 0.09)$$

We see here that “you” has the greatest probability. This top candidate is selected by a simple function (Argmax in the jargon), which picks up the word having the maximum value in the vector, here: “you”.



(Argmax / Selection Logic)

Figure 1: The selection of the highest probability candidate.

This is the basic principle of use of FNNs to predict the next word in LLMs. We shall explain the process in Chapter 3.

1.5 Recursion

If you iterate the use of the predictor, that is, after having input “I love” and getting “ you”, you now input “love you” in the model, you are going to get a fourth word, perhaps “so”, and the sentence grows to “I love you so”. If you carry on, inputting “you so”, you might get “much”, so the sentence then grows to “ I love you so much” etc. . .

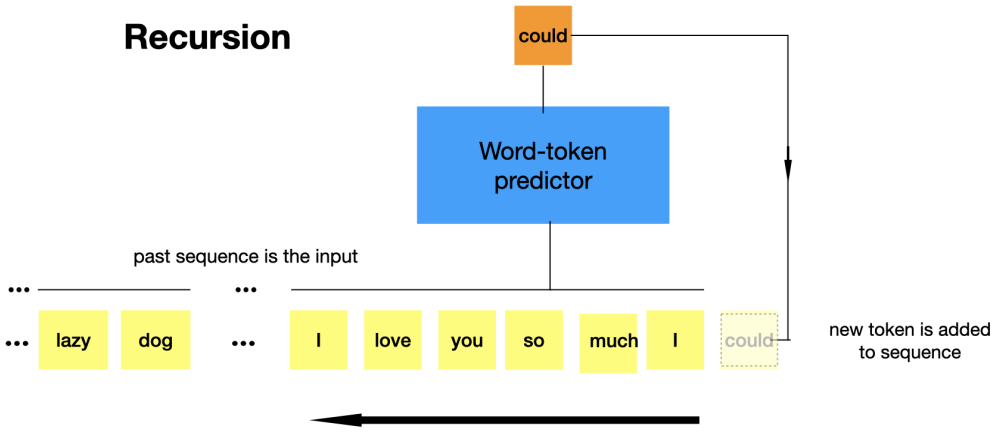


Figure 2: Recursion

A word of caution here. With a context window of 2 , you are not going to get meaningful text as I gave in this example because it is too small, e.g., it is highly improbable that just with an input "you so" the model would spit out "much". But when the context window is larger, i.e.> 5, it is then highly likely, and such models have been implemented as autocomplete programs.

1.6 Extension of the context window. Where the magic happens

Another word of caution, the Bengio model did not allow at that time a context window much larger than 10 (with the hardware of the early 2000s) because the number of parameters demand grows with the width of the window.

To overcome the fixed-size context window -and the fact that enlarging it makes early feed-forward language models grow rapidly in parameters—, several models have been invented, Long Short Term Memory (LSTMs), Recurrent Neural Networks (RNNs), Convolutional Neural Networks(CNNs), Sequence to sequence (Bahdanau attention), but the real breakthrough was Transformers (Vaswani, 2017. "Attention is All You Need")[5], which replaces recurrence with self-attention and enables efficient parallel training.

It also avoided parameter number explosion inherent to FNNs.

We will explain in Chapter 4, below how Transformers work. It is not essential to the understanding of the basic principle of word prediction as implemented for example in Bengio's model.

It is the extension of the context window to tens of thousands of words that made the magic happen. Long texts (input sequences) were completed or expanded in a very syntactically correct and meaningful way. This even partly surprised the inventors:

The model started to render or give back the Human Intelligence and knowledge present in the billions of sentences of the huge text corpus it had been trained on or ingested.

1.7 From Chatterbox to ChatGPT

What do we have now? We have a sophisticated system which can recursively and intelligently, possibly indefinitely, go on and on about and after a text you gave him. But this is not how ChatGPT behaves.

In order to make the system answer questions and be a useful assistant, the model is fine tuned by training with numerous questions followed by answers. So, in a simplified way : when it encounters these questions or equivalent formulations, it answers what it knows. Q: "What is Paris?" ; A : "Paris is the capital of France" (see See Chapter 4: Post-training).

Finally, it is worth mentioning that external measures are taken to stop the chatterbox, like: End of Sequence token (EOS), since the model on its own would continue forever.

So, now we know roughly how an LLM works. We spoiled the plot. If you are eager to get to Transformers, or already familiar with classification and neural network training, you can jump to Chapter 4 and come back to Chapter 3 later. But Chapter 3 is where you will discover why this all works — why geometric scores become probabilities, and why the architecture looks the way it does.

2 Forward Neural Networks

2.1 The FNN as a feature recognition or a likeness score network.

FNNs are a fascinating subject and should be the subject of another article. Here we are only going to uncover the most useful keys to our purpose. There are different perspectives on how to look at them:

There is the geometrical interpretation/explanation where input data is classified into regions through i.a geometrical linear transformations (scale change, rotation, shearing, folding, warping).

There is also the very common Universal Function Approximation Theorem view, which just says the FNN is a parametrized black box which can approximate any function and is inserted in a network trained by a supervised learning loop. Since it is viewed as a black box, it is not necessary and possible to look into it.

There is the Bayesian interpretation, where the output is viewed as an estimation based on the conditional probability of what the model knows and the input.

There is also the Energy Based Model (EBM) interpretation (Yann Lecun) which is very close to the perspective we will explain here, except that it does not make the detour with probabilities:

We choose the feature interpretation because we think it is the most intuitive, the closest to our ability to perceive reality and thereby more an explanation than a representation in other spaces (geometry, probabilities) to which the others lean to.

For example, in the geometric interpretation, the effect of the non-linear elements or RELUs of the neurons (see below) which is to "warp the space" never really resonated with my understanding and seems more a geometric consequence than an explanation.

For our purpose, a FNN is basically a pattern recognition network, a filter. You feed it a sequence of symbols (pixel patches, sound chunks, sentences,. . .) and it tells you: This is a hand, this sounds like "hello", this ("I, love") is a "followed-by-you" sentence bit (a sentence bit usually followed by "you") , etc. . . .

It tells you by giving you a score: a real number standing for the likeness, the similarity between the input and the pattern the filter is tuned to, is trained to recognize.

This FNN is built with a big number of elementary filters each of which also recognizes features.

Before we go slightly deeper, let us explain how these elementary filters work.

For readers unfamiliar with linear algebra—or simply in need of a refresher on vectors, matrices, and their multiplications—we have compiled, in Appendix A, the essentials used in this article.

2.1.1 The neuron

In the world of machine learning and neural network, everything is represented by a number, more precisely a series of numbers, a vector. In the medical world for example , a human being might be represented by the following vector: (age, height, weight, blood pressure at rest, average income, etc. . .). A basic operation in the field of neural networks is recognition or identification, it is required for classification, prediction and generation which are the basic applications of networks comprising FNNs . Since FNNs live in a world of vectors, we have to find a mechanism to implement this recognition. This mechanism is simply based on likeness: How similar is this vector to this other vector? How is this likeness assessed?

2.1.1.1 The dot product The similarity is expressed as an angle between those two vectors We are not interested in their length (norm), because a small rabbit is still a rabbit.

This angle may be expressed as the cosine of the angle. We might recall that this cosine of the angle between two vectors Y and W can be computed by their dot product divided by their norms: $cosine(alpha) = X.W/norm(X).norm(W)$.

If we fix W, and submit various input vectors X to this operation we will obtain each time a real number, a scalar which will tell us how aligned the input X is with the vector W. If the output scalar is small or

zero, they are very orthogonal, that is not aligned. If the output scalar is close to 1 (the max value), the input X is aligned with W, that is, it points in the same direction as W. Hence, we have a way to assess the likeness of an input vector to a specific vector.

The basic element of a FNN, the neuron, fundamentally implements this operation with a slight difference, it does not compute the norms of the vectors, the normalization is done before the inputs come into each layers so that the neurons only have to compute the dot product: $X.W = (x_1.w_1 + x_2.w_2 + \dots x_n.w_n)$, with $X = X(x_1, x_2, \dots, x_n)$ and $W = (w_1, w_2, \dots w_n)$. W should be normalized as well but it is common practice in modern LLMs not to enforce this constraint so as to give the network during training more freedom to adjust the parameters which has proven beneficial.

Not normalizing W seems at first ambiguous but here is the thing: The non-normalization of X would be very ambiguous because a long badly aligned input could score more than a small well aligned input. This is why the inputs are normalized.

But for the weight vector W this is different. The effect is only that the correctly measured likeness is amplified by the length of W. This is fundamentally different because it scales all the outputs of this neuron equally, thus preserving their relative magnitude which is what really matters. Choosing not to normalize W just gives the FNN during training more degrees of freedom. This has proven beneficial and is the common way.

So to give an example, in a FNN trained for animal recognition and for a neuron at the output layer and only there: If the output scalar of the neuron is high, that is if the input X and the filter vector W (representing e.g. a rabbit) are aligned, then X resembles strongly a rabbit!. If it is low or zero it means : it is not a rabbit!

2.1.1.2 The bias A bias b is added to this operation. The bias (often negative) sets the sensitivity threshold: how aligned must X be with W before the filter should tell I have recognized my pattern? A negative bias means “I need strong evidence”, while a positive bias means a “very small evidence will suffice”.

2.1.1.3 The rectifier To these elementary filters, another filter is added: a rectifier (RELU): if the value is negative, then say it is zero ! ($RELU(x) = max(0, x)$). A feature score can only be positive or zero. Either the feature was detected or it wasn't. We can't detect how much absent it was. It doesn't anti-exist. It can't be negative. A non-rabbit does not exist! Evidence against a hypothesis is expressed later by giving that feature score a negative weight value in the next layer.

In modern LLMs, the RELUS have been replaced by leaky RELUS or GELUS where small negative inputs are still allowed for stability.

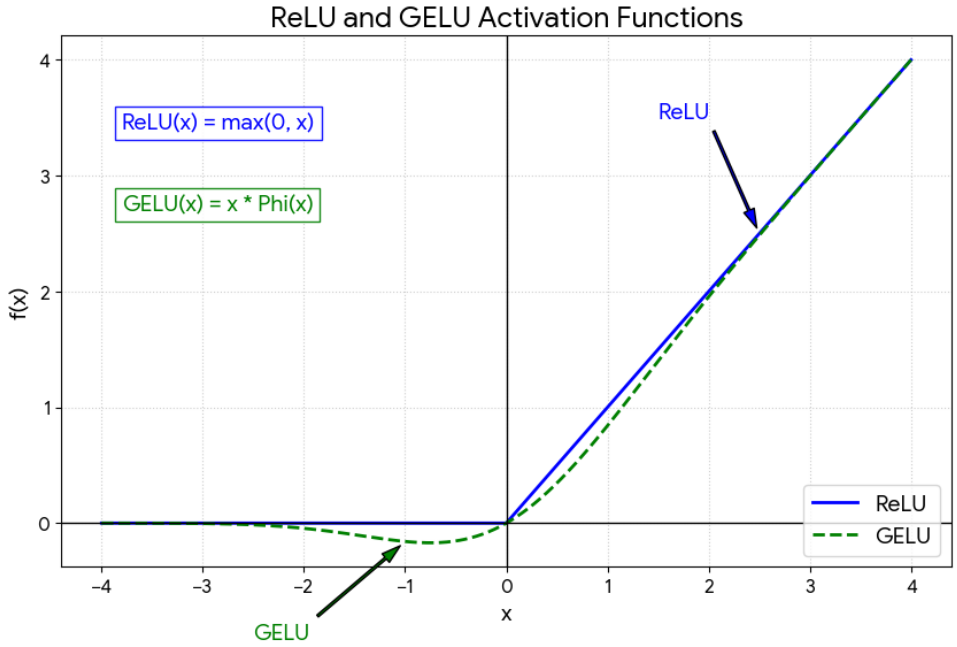


Figure 3: RELU and GELU

2.1.1.4 The final neuron In summary the combination: a dot product maker, a bias and a rectifier is called a neuron. Its input is a vector X, its output a scalar $y = \text{ReLU}(W \cdot X + b)$.

Don't confuse the role of the bias and the rectifier. Although the rectifier RELU as well as other alternatives like the sigmoid are often named the activation function, it is the bias which determines if the neuron fires or not (is active or not), not the activation function (rectifier). The rectifier just prohibits negative signals (negative existence is sort of banned).

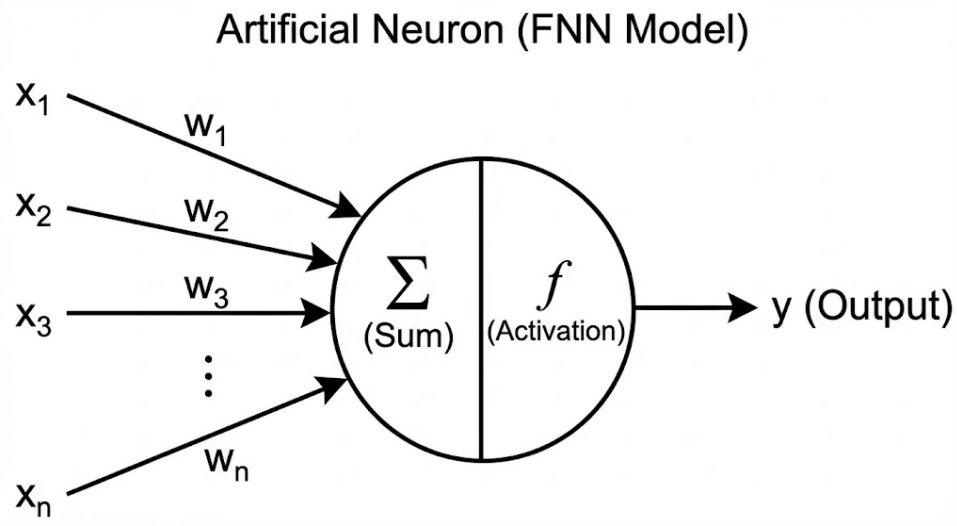


Figure 4: The structure of an artificial neuron.

2.1.1.5 The grandmother's cell What to do with our neuron? A natural idea which comes to mind is to use a single neuron and match its weight vector exactly to the pattern (your grandmother standing) we want to recognize. It will recognize this pattern, perhaps other very similar pictures of your grandmothers but it won't recognize your grandmother sitting or walking or with a hat on. It will also fire occasionally when presented with some other grandmothers. It can basically only recognize this very picture of your grandmother standing. The grandmother's cell is a common example cited in this field.

This is why FNNs appeared.

A FNN is made of a massive number of such interconnected neurons each of which recognizes specific features.

2.1.2 The FNN

We don't want and need to completely explain Forward Neural Networks here, but a few insights should be mentioned. In FNNs there are two distinct modes: inference and training. Inference is the machine learning jargon word meaning using the trained neural network. Training is tweaking the parameters of the FNN (all weights and biases of all neurons) to make it able to perform the task. This is done by using a huge set of data and telling it in each case what the datum is, labeling it. This is called supervised learning. For example, the data is a huge set of digitized pictures of animals and the labels are the name human beings gave to these animals. The hope and aim is that once trained, the FNN will be able to recognize animals from other images than the ones in the training data set.

As an example we are going to take a neural network which has been trained to recognize a pattern Xs e.g. a rabbit, "hello word sound", or a sentence chunk always followed by "you".

2.1.2.1 Distributed sparse representation We are going to use the most basic FNN able to recognize a pattern. It comprises:
—One input layer consisting of a several neurons.
—One hidden layer also consisting of a multitude of neurons.
—One output neuron.

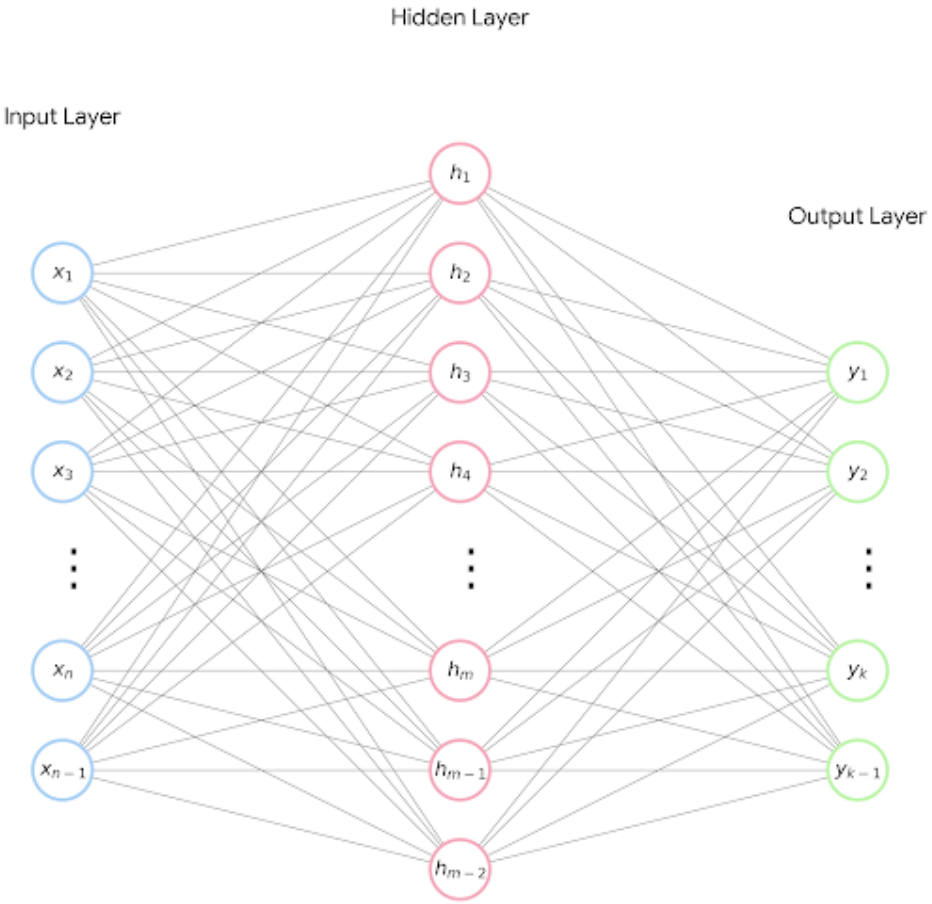


Figure 5: A one hidden layer FNN.

The mechanism it implements is distributed sparse representation.

The hypothesis is that there exists a learned feature basis or feature bank $\{W_i\}$ such that an input x can be usefully described by the collection of scores $s_i = W_i^\top x$. A layer realizes this through its neurons, each one acting as a feature detector: its weight vector W_i defines the feature, and its activation s_i measures how much that feature is present. High activations therefore indicate which learned features best explain the input.

This is exactly what the so-called hidden layer does. It is a bank of such filters implemented by neurons: If the hypothesis is true and we also have a sufficiently wide bank of neurons, the weights (a vector W_i) of which have been set during training to make the neuron fire for one of these features of X, some of these neurons will react/fire more than other ones because they correspond to the characterizing features we have suspected to exist. The other neurons will output a much lower score because they do not represent/resonate with the features of this input, but probably with other inputs.

In order to select these suspected characterizing features which have high scores, we need a mechanism. The most obvious mechanism is to shunt/ignore the score s_i (make it zero) if it is too small, that is if it is below a threshold. A weak score means that the feature for which this low-score neuron is trained to detect is not a feature of this input.

This mechanism is precisely what the bias b (in combination with the RELU) does in a neuron:
$y = 0$ if $W.x < -b$; $y = Wx$ if $W.x > b$ or $y = \max(0, W.x + b)$.

Through this mechanism, we have, let us say only 1/10th of all the neurons of the first layer which fired. This is called sparsity. This is why the distribution is called sparse.

The weight vectors of these neurons are the vectors which characterize the input vector. The hypothesis has been verified in practice.

Each of these neurons outputted an evidence that the input X is the input Xs the FNN looked for. We then add these evidences/indices (with weighting parameters) to a sum which is then a greater evidence/clue that the input is Xs (eg. a rabbit, etc. . .)

If the hidden layer is large enough, this is usually a reliable and final evidence. This final evidence is computed by the single output neuron.

A word of caution here: when two different inputs belong to the same class (two different rabbits) It is not necessarily the same neurons that are going to fire, in other words the characterizing features of both inputs are not necessarily the same but many are the same with different weights precisely because they belong to the same class, they are both rabbits. It is so because the images differ.

2.1.2.2 Super pseudo-neuron We have explained how a one hidden layer FNN, the most basic FNN is in a position to recognize a pattern. We can view it this way: we increased the pattern recognition capacity of a neuron by replacing it by a basic FNN comprising: 1 input neuron, 1 hidden layer, 1 output neuron. Let us call it the super pseudo-neuron.

In it, the signal is distributed and then re-concentrated, achieving a much more robust pattern recognition through shared representation.

This concept is useful to understand deeper layers.

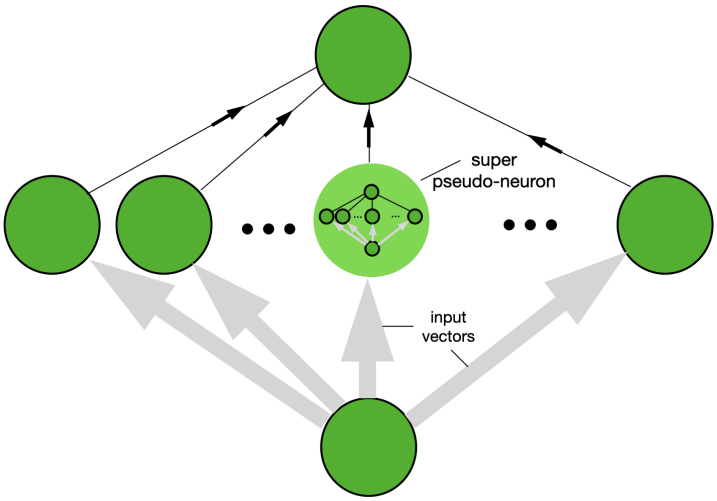


Figure 6: The pseudo super-neuron inserted in a FNN

2.1.3 Deep layers

As seen above, the input of a layer (input layer, hidden layer, output layer) is a vector and its output can also be a vector.

Thus in general form a FNN layer can be seen as a vector value function or Y = f1(X) or a vector signal transforming unit (not to be confused with transformers): where Y= (y1,y2, y3, ... ym) and X= (x1,x2, x3..... xn).

Such layers can be stacked upon each other (see Fig. 7). Instead of having one hidden layer, we have several hidden layers.

What happens then is that the state vector X is transformed by the different layers of the FNN by a transformation F which is the composition of functions inherent to each layers.

In the case of 4 layers: F(X)= f4(f3(f2(f1(X)))) which is commonly expressed as:
$F = f1 \circ f2 \circ f3 \circ f4$. o being the symbol for the composition of functions.
More generally, in the case of m layers: $F = f1 \circ f2 \circ f3 \circ \cdots \circ fm$

Stacking several hidden layers has a tremendous advantage: It has been shown (Eldan and Shamir, 2016) that stacking lets you reduce dramatically, even exponentially the number of neurons, the FNN needs.

The way several hidden layers work is the following: Let us take a 1 hidden layer FNN which does not work well, i.e each neuron of the hidden layer fails to recognize meaningful characterizing features. Probably the layer is too narrow. We can then replace each of these failing neurons by our pseudo super-neurons of 2.1.2.2 and achieve satisfying pattern recognition.

But there is one drawback, the second hidden layer, comprising all the hidden layers of all our pseudo super-neurons is huge. However, remembering the shared representation of § 2.2.2, we can massively reduce its width without impairing the recognition.

In the Figure below, each (vertical) layer is a vector function fi i= 1 to n and the FNN as composition of these vector functions

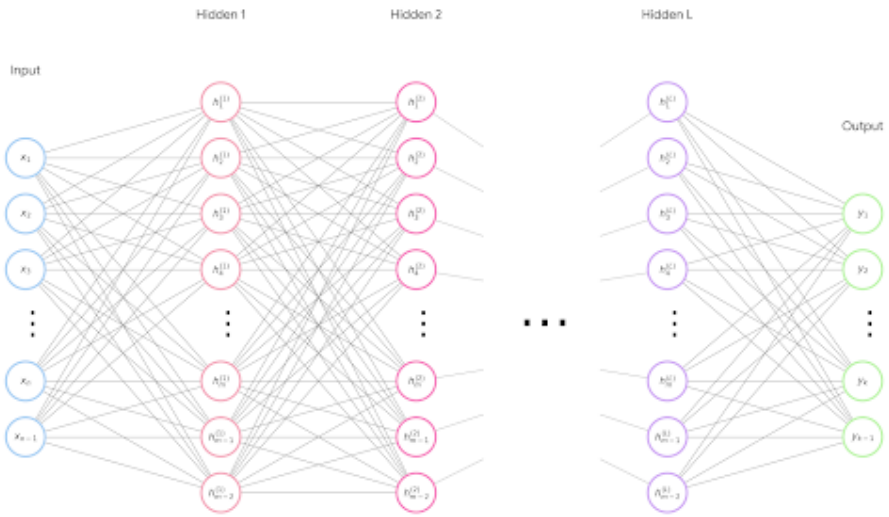


Figure 7: A FNN as composition of vector functions.

Very deep layers are used in LLMs to match the complexity of the modeling.

2.1.4 Residual networks (skip connections).

A common variant of deep feedforward networks uses residual (skip) connections. Instead of having each block learn a completely new representation, the block learns an update to its input: $x \mapsto x + F(x)$. In other words, the network represents changes (F(x)) and adds them to the existing representation via an identity shortcut. This greatly improves optimization in very deep networks and is now standard practice.

Transformers use this residual structure throughout: each attention sublayer and each feedforward (MLP) sublayer is wrapped in a residual connection (often together with normalization), unlike the plain FNN structure in the original Bengio-style model.

2.1.5 Training

We are not going to dive deep into training in this paper, but we will explain its basic principles.

2.1.5.1 Supervised learning

We have a parametrized model of a function

$$\hat{Y} = f_\theta(X)$$

where θ is the vector comprising all the parameters of the model.

For each data pair (Xi, Yi) of the data set we compute the prediction by the model: $\hat{Y}i = f(Xi)$

We then compute a function of the discrepancy between the prediction $\hat{Y}i$ and the known result Yi.

This function is called the loss per sample L.

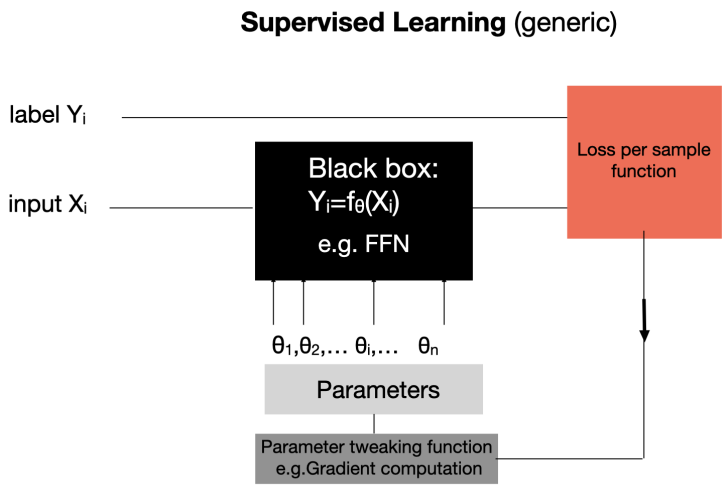


Figure 8: Supervised learning.

Training consists of minimizing the average loss over the whole dataset:

$$L_{\text{dataset}}(\theta) = \frac{1}{N} \sum_{i=1}^{N} L\big(f_\theta(X_i), Y_i\big).$$

We will explain below in 2.1.5.6 how the loss function is computed (in regression it is basically a difference), but we will first explain how this loss can be minimized:

2.1.5.2 Gradient Descent To minimize the loss function $L_{\text{dataset}}(\theta)$, we use *gradient descent*. The gradient of the loss is the vector of partial derivatives with respect to all parameters:

$$\nabla_\theta L_{\text{dataset}}(\theta).$$

It points in the direction of steepest increase of the loss. Therefore, moving in the opposite direction decreases the loss.

The parameters are updated according to:

$$\theta \leftarrow \theta - \eta \, \nabla_\theta L_{\text{dataset}}(\theta),$$

where η is the learning rate.

When this update is repeated, the parameters move toward a minimum of the loss.

Gradient descent was introduced by Cauchy (1847) under the name *method of steepest descent* (see Fig. 9 below).

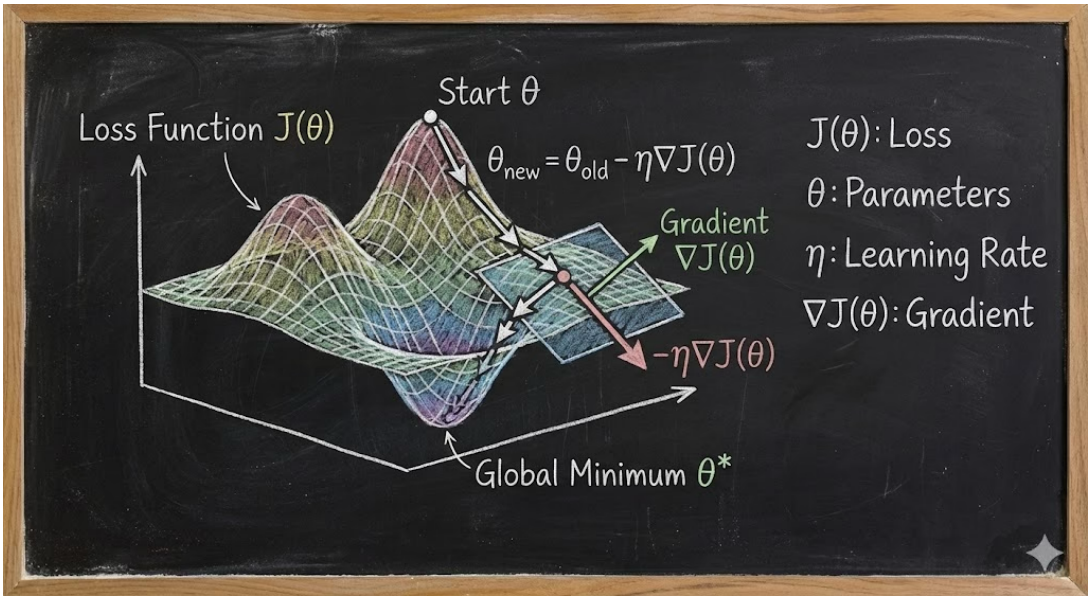


Figure 9: The gradient descent.

(in this drawing the loss is called J instead of $L_{\text{dataset}}(\theta)$.)

2.1.5.3 Stochastic Gradient Descent (SGD) In practice, the dataset is too large to compute the full gradient $\nabla_\theta L_{\text{dataset}}(\theta)$ at every update step.

2.1.5.4 Online SGD
One variant is to update the parameters at each new input/sample (Xi,Yi).

This is called online SGD or SGD in the strict sense.

It might sound surprising that a single sample enables updating the parameters meaningfully. The key point is that, although the *true* objective is the dataset loss

$$L_{\text{dataset}}(\theta) = \frac{1}{N}\sum_{i=1}^{N} L_i(\theta), \qquad \text{with} \quad L_i(\theta) = L\big(f_\theta(X_i), Y_i\big),$$

the gradient of this average is the average of the per-sample gradients:

$$\nabla_\theta L_{\text{dataset}}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta L_i(\theta).$$

Therefore, the gradient computed from one random sample, $\nabla_\theta L_i(\theta)$, is an *unbiased estimator* of the full gradient:

$$\mathbb{E}_i[\nabla_\theta L_i(\theta)] = \nabla_\theta L_{\text{dataset}}(\theta),$$

where the expectation is over the random choice of the sample i. So each single-sample update is noisy, but it points in the correct direction *on average*. Over many steps, these noisy but unbiased estimates accumulate and drive the parameters toward a minimum.

2.1.5.5 Mini-batch SGD
Usually, we approximate the gradient using a small random subset of the dataset, called a *mini-batch*.

If the mini-batch contains B samples, the batch loss is:

$$L_{\text{batch}}(\theta) = \frac{1}{B}\sum_{i=1}^{B} L\big(f_\theta(X_i), Y_i\big),$$

and the gradient update becomes:

$$\theta \leftarrow \theta - \eta\, \nabla_\theta L_{\text{batch}}(\theta).$$

This is called *stochastic gradient descent* (SGD), because each update step is based on a random sample of the dataset. Therefore, the gradient is not exact: it is a noisy estimate of the true dataset gradient.

This randomness may seem inefficient, but it has two major advantages:

- it makes training computationally feasible,
- the noise often helps the optimization escape poor local minima and improves generalization.

An *epoch* is defined as one complete pass over the entire dataset. In practice, training consists of many epochs, each epoch containing many mini-batch updates.

Mini-batch SGD is also more computer efficient than online SGD to reach the same accuracy, because online SGD needs many more updates (and it uses hardware inefficiently), so the total training/compute time can end up higher.

2.1.5.6 The Loss As we mentioned, the FNN is trained on a dataset, which is a large collection of pairs (X_i, Y_i), where X_i is the input and Y_i is its label.

We first explain the concept of loss in the regression setting, where the goal of the network is to approximate a numerical function.

In § 2.3.4 below, we will explain how the loss is defined for probabilistic prediction. There, instead of measuring the difference between a predicted value and a label value, we measure the difference between probability distributions.

In supervised learning, the model defines a function

$$\hat{Y}_i = f_\theta(X_i),$$

where θ denotes the set of all parameters (weights and biases) of the network.

The goal of training is to choose θ such that, for all input-label pairs (X_i, Y_i), the output $\hat{Y}_i$ is as close as possible to the label Y_i.

To quantify how wrong the model currently is, we define a scalar function called the *loss per sample*:

$$L(\hat{Y}_i, Y_i).$$

A common loss per sample in regression is the mean squared error (MSE)ntroduced by Legendre (1805):

$$L_{\text{MSE}}(\hat{Y}_i, Y_i) = \frac{1}{K}\sum_{k=1}^{K}(\hat{y}_{i,k} - y_{i,k})^2 = \frac{1}{K}\|\hat{Y}_i - Y_i\|^2.$$

where K is the dimension of the vectors.

Squaring has two advantages:

- it makes all deviations positive and therefore count as errors,
- it amplifies large deviations, making the loss more sensitive to big mistakes.

This is the standard loss per sample in regression and is often called the L_2 loss per sample.

Sometimes, the squared loss per sample is too sensitive to outliers, and one uses instead the mean absolute error (MAE):

$$L_{\text{MAE}}(\hat{Y}_i, Y_i) = \frac{1}{K}\sum_{k=1}^{K}|\hat{y}_{i,k} - y_{i,k}| = \frac{1}{K}\|\hat{Y}_i - Y_i\|.$$

This is called the L_1 loss per sample.

In the rest of the article, we will just name the loss whether over the dataset, mini-batches or a single sample: the loss L.

2.1.5.7 Computing the gradient of the loss

As we might remember, the gradient is just a vector comprising as coordinates the partial derivatives of the function versus each of the coordinates. For the loss function L it looks like this:

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) = \left(\frac{\partial L}{\partial \theta_1}, \frac{\partial L}{\partial \theta_2}, \ldots, \frac{\partial L}{\partial \theta_m} \right).$$

2.1.5.8 Coordinate descent

The first idea which comes to mind (e.g. for online SGD) is to compute each coordinate of this vector numerically by increasing or decreasing each parameter separately by a small amount, leaving the other parameters unmodified, and computing the loss:

$$\frac{\partial L}{\partial \theta_j}(\theta) \approx \frac{L(\theta_1, \ldots, \theta_j + \varepsilon, \ldots, \theta_d) - L(\theta_1, \ldots, \theta_j, \ldots, \theta_d)}{\varepsilon}.$$

This works but is extremely expensive in high dimension (it costs d or 2d loss evaluations per update), which is why we use backpropagation to get all partial derivatives at once.

2.1.5.9 Backpropagation

Instead, the computation of the gradient $\nabla_\theta L_{\text{batch}}(\theta)$ is performed efficiently using *backpropagation*, which is an application of the chain rule of derivatives.

Backpropagation is essentially repeated use of the chain rule.

Chain rule (single-variable intuition). If a quantity depends on θ through intermediate steps

$$\theta \mapsto a(\theta) \mapsto b(a) \mapsto L(b),$$

then the derivative of the final loss with respect to θ is

$$\frac{dL}{d\theta} = \frac{dL}{db} \cdot \frac{db}{da} \cdot \frac{da}{d\theta}.$$

In words: *multiply the local slopes along the path from θ to L.*

Multivariable version (what networks use). In a neural network, each layer produces a vector and the loss is a scalar. For one layer

$$h = f_\theta(x), \qquad L = \ell(h),$$

the chain rule becomes

$$\nabla_\theta L = \left(\frac{\partial h}{\partial \theta} \right)^\top \nabla_h L.$$

Backpropagation computes $\nabla_h L$ for the last layer, then propagates it backward layer by layer, reusing intermediate derivatives, so that we obtain *all* partial derivatives $\frac{\partial L}{\partial \theta_j}$ efficiently.

This is the gradient of the loss with respect to the parameters. We use it to perform gradient descent: the parameters θ (and hence each θ_j) are updated so as to decrease L. In practice, this update is performed repeatedly—either after each training example (X_i, Y_i), or, more commonly, after each mini-batch—so that the dataset loss becomes smaller over time.

The detailed mechanism of backpropagation is beyond the scope of this paper but we have included their equations in the appendix B for completeness.

2.2 Classification

2.2.1 Recognition of several patterns

We explained how pattern recognition works. Classification is not far away: Instead of having one output neuron, we have several output neurons, one for each class, that is for example in animal image classification, one for rabbits, one for cats, one for elephants etc... Classification is nothing else than simultaneous recognition of several different patterns.

2.2.2 Shared representation

You may ask, why does it suffice to hook new neurons in the output layer. Should not we have one FNN of the type described above for each class?

We don't and the reason is the following: When you train such a FNN with several output neurons, one for each class, you create implicitly a library of features, residing as weight vectors in each of the neurons of the hidden layer. Not only an input corresponding to one class like the rabbit class in our previous example can be decomposed in features from the hidden layers but also inputs from other classes (cats, elephants,...) In other words, this bank of features (the $W_i s$) in the hidden layers does not enable to express only one class but all classes present in the training data set, such as cats, elephants,... and detect them with their corresponding output neurons. This is called shared representation

This also means that the output layer outputs several scalars, one for each neuron. It outputs a vector. So does the FNN. This is not exclusive to classification. This is the also the case in most applications, like function approximation.

2.3 Prediction

An application that we find mostly in some language models is to predict the next element in a stream or a long sequence of elements, from a collection of elements from which the stream is made of. In language models, it means : having an unfinished sentence, predict the next word.

2.3.1 Prediction as selection

If you look at it under the right angle, prediction is nothing else than classification with a subsequent selection of the class, even if the class appears a bit unusual. In the following we will focus on language as an example because language models is the focus of our article, but it applies to any type of prediction.

Let us take our favorite example, this time with a context window of 4 words. Consider the input: "I love you so", the possible next words could be : "much", "badly", "intensively",... These represent classes.

Other inputs that belong to class "much": " I hate you so", "well I am very","it is simply too", because they can all be followed by "much". Now, inputs that could belong to the class "badly": "He thought it would", "he claimed he could" , etc... And so on with class "intensively"...

The FNN is designed with an output neuron for each word of the vocabulary. That is it will give a likeness score for each word of the vocabulary for any input, meaning it will output a score telling how well the input sequence belongs to this class/word. It does this for all words of the vocabulary.

You can then make the model identify the highest score for a given input e.g. : "I love you so". Let the highest score correspond to the class: "much", the model has then just picked up "much" as its choice. It has selected the next word from the vocabulary.

2.3.2 Spikiness

We can view an FNN in classification as follows:

The aim of a FNN in classification (including the class output layer) is through multiple re-representations of the state vector to arrive at a spiky representation where one coordinate is much higher than the other ones (especially visible when your subtract a reference logit).

2.3.3 From alike to likely

Most of us did not have any problem with the above method of selecting the next word based on geometric likeness/similarity (geometric because dot products are geometry).

But some of us might feel ill at ease: A quasi-geometric similarity suddenly becomes the likelihood or likeliness of an event happening, a score akin to probability? Can we change from a geometric space to a probability-like space just like that?

This unease roots in the fact that we intuitively associate probability with counting, witj frequencies and histograms (like the Shanon N-grams of §3.1) but none of this happened in our FNN. This unease is therefore perfectly legitimate.

But another take is present in our intuition. We might catch ourselves thinking or saying: this score is the highest, it is therefore more likely that it is a rabbit. This score is so low, it is very improbable that it is an elephant.

We suddenly shift conceptually from likeness (geometry/score) to likelihood/likeliness (probability/frequencies), we change space, some of us perhaps without thinking or noticing it, others puzzled and resisting for the reasons mentioned above. How does this happen ?

It comes from our perception and experience: Something difficult to attain will be rare. Something easy to attain will be frequent. This is ingrained in our intuition, for good reasons, because this is how nature often functions.

The best and most famous example (though not the first) was discovered by Boltzmann in his study of gas molecules at various temperatures.

2.3.3.1 The Boltzmann-Gibbs assumption Boltzmann discovered that molecules don't spread uniformly across all possible configurations. Instead, they concentrate in lower-energy states with a probability P(E) which is proportional to: $\exp(-E/K_b \cdot T)$, where E is the energy, T, the temperature, Kb the Boltzmann constant.

Boltzmann showed that a mapping existed in nature between: —quantity z derived from geometry (geo=world, metry= measure), that is from measurements of the state S of objects in the world, e.g. :position, speed, energy etc... and — their probability of being in that state:

It is a mapping form Geometry to Probability.

He discovered that this mapping was exponential: the probability of the state being at level/score z P(S=z) is proportional to exp(z). In his case the measure or score z was : $-E/K_b \cdot T$.

He was not the first one to discover such mapping. Gauss did it 70 years earlier observing planets motions. Inspired by Galileo who noted that small errors in astronomical measurements occurred much more often than big errors. While Gauss described the probability of errors (distance from truth), Boltzmann described the probability of states (energy levels). Both found that nature favors the path of least resistance.

Gauss realized that the probability P of an error x in measurement to happen was proportional to the exponential of a score $z(= -(x-\mu)^2/2\sigma^2))$, also geometric because derived from how much the error was away from the average of errors μ, or if you defined the error state by its score z:

$P(S = z)$ is proportional to $\exp(z)$. Again geometry to probability.

This reveals a fundamental principle in nature:

What is more stable, easier to reach (lower cost/energy) or closer to an average tends to be more probable.

This is why stair steps are more worn in the middle (where walking is natural and easy) following a bell curve, with less wear at the edges (where walking is awkward).

Boltzmann and Gibbs seem to be more referred to in Machine Learning (ML) than Gauss, more for historical reason. Perhaps also because the expression of the score or exponent in the Boltzmann formula comprises a constant called the Temperature which can tweak the flatness of the distribution which will be computed with it. See §9.1.6.

We may call this the Boltzmann-Gibbs assumption and it is the second key insight to understand LLMs.

We make this assumption for most data we process in machine learning and particularly for prediction with neural networks:

If we view the task as maximizing similarity between input and pattern, then: A high score z indicates this task was easy to accomplish - the input naturally aligns well with the pattern. A low score z means the alignment is difficult, forced. By the principle observed in Boltzmann's or Gauss's work - that what is easier to reach is more probable - we interpret high similarity as indicating high probability.

In other words we posit a Boltzmann-Gibbs distribution as the relationship between geometry and probability. This assumption is often shoved under the rug but it is the fundamental hypothesis that enables neural networks which processes geometrical signals, to output probabilities.

This hypothesis is not mathematically proven for language or images, but empirically. Natural data follows this peaked distribution structure - typical patterns are frequent, extreme ones rare - just like Boltzmann's molecules and Gaussian distributions.

2.3.3.2 Softmax: From likelihood to probability distribution Using this hypothesis we can suddenly do more with our likeness scores for each word of the vocabulary, than just predict the most likely word with selection of the highest score as in 2.3.1 above. We can suddenly estimate a probability for each word of the vocabulary to be the next. In other words we can compute a probability distribution for each word of the vocabulary to be the next.

Since the sum of all probabilities must equal 1 which is inherent to the definition of probabilities, we have to divide the exp(z) by the sum of all exponentials of all possible states i: sum (exp(zi)) : Now we don't have just a proportionality rule but an exact rule:

$P(\text{state scores } z_i)$ or $P(z_i) = \frac{\exp(z_i)}{\sum \exp(z_i)}$

This function is called Softmax and its use to compute probability distributions of classes in classification is attributed to John S. Bridle[4] in 1990.

Since we turn scores to probabilities with this function, we can turn probabilities to scores with logarithms. This is why the scores are (called) logits in the ML/FNN jargon.

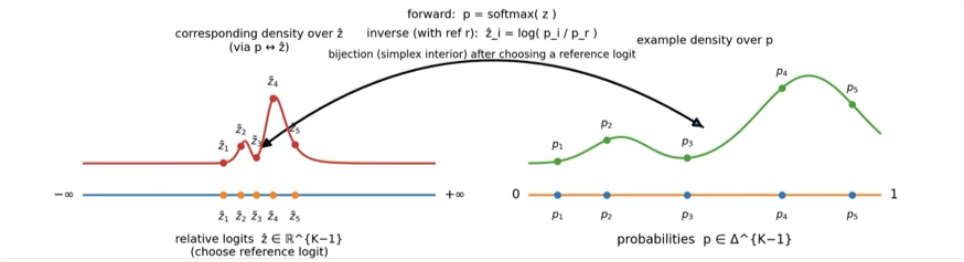


Fig.6: Softmax
The Logits to Probabilities mapping

Figure 10: Softmax: From Logits to Probabiliies

2.3.4 The Loss when comparing probability distributions

In §7.2.3.1 above, we defined the loss using the approximation (also called regression) of the inputs (the outputs of the model) and their labels.

We can apply this to probability distributions, that is compare them and derive a loss to use in the (supervised learning) training. The first distribution is the output of our predicting network and is the discrete probability distribution: $\hat{P} = (\hat{p}_1, p_2, \hat{p}_3, \ldots, \hat{p}_n)$.

The "label" distribution to compare it with is the vector: $Q = (q_1, q_2, \ldots q_i, \ldots q_n)$(e.g. 0,0,0. . . .,1,0,0,0,0,0,0), as wide as the vocabulary (e.g 128 000 !), When i s the right token, $q_i = 1$ and $q_{i\neq 1} = 0$ for all other tokens.

This kind of vector is called a one-hot vector in the jargon (This is how the tokens are also input in the model just before being embedded as mentioned in 1.3 above).

This label vector can be viewed as hard discrete probability distribution of the right token: It gives the highest probability (1) to the right token and (0) to all other tokens and the sum of the probabilities is 1 (hard because it is one or zero).

So we could compute the loss as:
— the mean average estimate: $MAE = \sum(|q_i - p_i|)/N$ for $i = 1$ to N or
— the mean squared estimate: $MSE = \sum((q_i - p_i)^2)/N$ for $i = 1$ to N
just like in approximation (regression), see 2.1.5.6 above.

These losses collapse to:
— $1 - p_i$ for MAE and
— $(1 - p_i)^2$ for MSE
due to the fact that Q is a hard distribution (all zeros except 1).

This (use of the MSE) has been done by G. Hinton(1986), in his groundbreaking paper: "Learning representations by back-propagating errors"[1]. However this was abandoned because of the tendency of the training to be too slow and even get stuck, a problem called the vanishing gradient (MAE is even worse).

2.3.5 The vanishing gradient

One remembers that the gradient (the derivative representing the "steepest slope") is used in gradient descent to update the model parameters recurrently towards their optimum, as seen in § 2.1.5.2.

Mathematically, calculating this gradient relies on the Chain Rule. To see the "mechanical miracle" of Cross-Entropy, we must look at the gradient with respect to the model's raw output scores, called logits (z):

$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial p} \cdot \frac{\partial p}{\partial z}$

where L is the loss, p the probability of the model to predict the correct class.

We thus see that the gradient decomposes into the slope of the loss: $\partial L/\partial p$ and the slope of the model $\partial p/\partial z$, where z (Logit) is the raw score the model calculates and p the probability derived from that score (via Softmax).

The Problem: The issue arises when the model is wrong: it assigns a negligible probability ($p \to 0$) to the correct class. In this region, the Softmax curve becomes flat. Consequently, its slope $\partial p/\partial z$ vanishes toward zero (specifically, it scales with p). If we used a standard loss function, this tiny term would dominate, causing the total gradient to vanish and leaving the model stranded in error.

The Solution: To counter this, we need the "Slope of the Loss" ($\partial L/\partial p$) to act as a counterweight. It must grow toward infinity exactly as fast as the Softmax slope shrinks to zero.
The Logarithm ($L = -\log p$) is the specific function that achieves this.
— Softmax Slope: Shrinks to zero (proportional to p)
— Log Slope: Grows to infinity (proportional to 1/p). See Fig.11 — The Cancellation:

The Gradient is approximately equal to $(1/p) \cdot p = 1$. The infinite scream of the Logarithm perfectly cancels the vanishing whisper of the Softmax. The result is a clean, linear gradient $(p - 1)$ that flows back to the parameters, ensuring that "completely wrong" receives a strong, constant corrective shove.

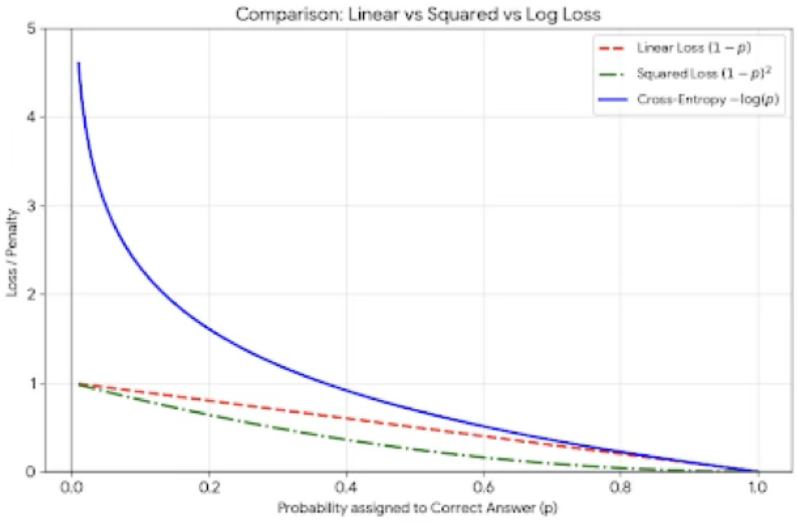


Figure 11: MAE, MSE, LOG Losses

Historical Note: This concept, Cross-Entropy, is rooted in Information Theory, originating from the work of Shannon (1948) and Kullback & Leibler (1951). For LLMs, we treat it as a mechanical necessity: it is the only loss function capable of repairing the vanishing gradient caused by Softmax saturation.

2.3.6 A note about the unembedding layer

Since we embedded tokens, that is mapped a word index to a vector, we might expect to "undo" this at the end. But the hidden state h is not an embedded token; it is a contextual vector. The output layer therefore does not invert the embedding. Instead it applies a linear map that produces one score (logit) per vocabulary item: $z = W.h + b$, where z is real and contains one logit per token in the vocabulary. Softmax turns these logits into a probability distribution over the vocabulary. Only after argmax or sampling do we obtain a single token index. We can view this output projection as the model's unembedding step. In our FNN/Bengio model, the output layer takes the role of the unembedding layer.

2.3.7 Soft prediction or sampling

Historically, at least for (mutually non exclusive) class predictions, the networks included this conversion of logits into probabilities (Bridle[4]). It is later that one developed networks without Softmax, classifying only with the logits, the Energy Based Models (EBM, Yann Lecun) for example. The incentive to avoid computing Softmax is to save computing resources.

Our first approach, predicting by selecting the highest logit (score) is a hard prediction. In language model, it means: this token/word and no other. This would be and has shown a bit too severe for language models, they would show no variation or flexibility in the recurrent predictions inherent to their model. They would also get stuck in a loop, eg.: I think that I think that I think ...

However, using Softmax and the probability distribution alleviates this problem. We can design mechanism to choose slightly lower probabilities than the maximum one. We have the choice. and Bengio's model comprises Softmax.

This is done by flattening the distribution through lowering the temperature in the exponents of the exponentials in softmax (see § 2.3.3.2) and random sampling the tokens in a fixed interval.

We have now arrived at what we promised in 1.4.4 above: the explanation of how a FNN predicts the next token/word (in a probabilistic sense) in the Bengio language model. This is the cornerstone of how LLMs work.

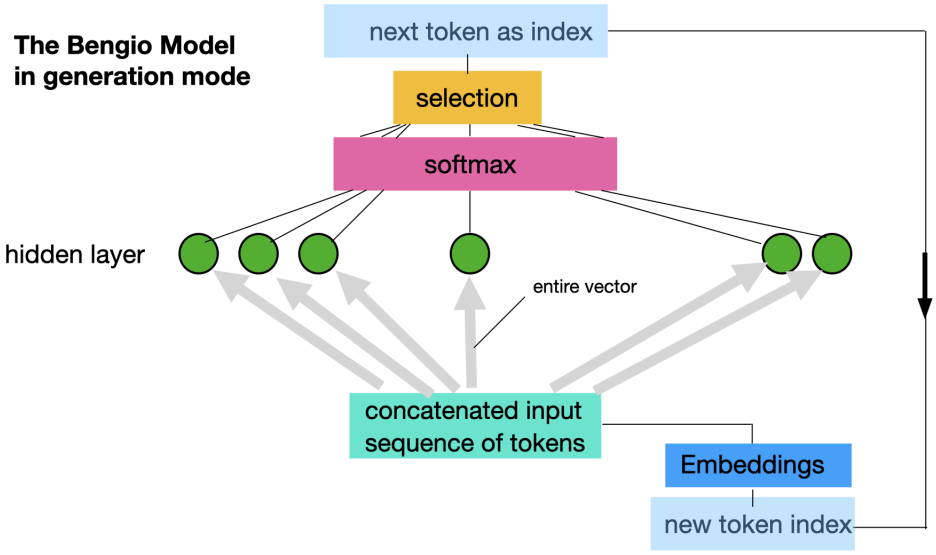


Figure 12: The Bengio model

3 Transformers

3.1 Transformers

Although the state of the art before Transformers (2017) was RNN, CNN, LSTMs and Sequence2Sequence networks, we will build on our FNN Bengio model to introduce Transformers. The reason is that the Bengio model comprises the foundation of the architecture of prediction, which we will remind here.

3.1.1 The backbone of Transformers: the Bengio Small Language Model

In (supervised) training, a dataset is input as long streams of data (coherent text parsed into tokens). The stream is viewed sequentially through a context window: at each new token we form a pair {label, input}:

$$\{\text{the new token},\ \text{the preceding sequence (of the window size)}\}.$$

The network soft-classifies the input versus the label. That is: for each possible next token, the model learns a scoring function that says how compatible that token is with its preceding sequence.

Hence, in inference (use of the network), when you input a sequence, the network computes a score for each possible next word and selects the next word among those with the highest scores (typically by argmax or sampling).

In short the network is a classifier. In FNN classifiers like our Bengio model, the process is:

- The tokens of the input sequence are turned into vectors by embeddings, and combined into a state vector h.
- The state vector h representing the input sequence is re-represented by successive layers until it reaches a more spiky shape (one coordinate is much higher than the others) at the last layer, the output projection layer (before softmax).
- This spikiness enables detection of the highest score and thus classification.
- In most systems, softmax is applied to produce a probability distribution and eases soft-sampling (flexibility in the choice of the prediction).

This spikiness is limited by the empirical distribution of the data (irreducible uncertainty/entropy).

These principles and architecture are also present in the Transformer architecture since they are the backbone of prediction. However, in the original paper about Transformers—which is very good for the engineering design—this backbone is mentioned but not explained. This is the reason why many readers get lost and don't get an end-to-end picture of how it works. This missing link has been provided by this article. But before we dive into Transformers, we have to remind or introduce three concepts:

3.2 Preliminary

3.2.1 Distributed representation

Neural Networks process distributed representation of information expressed as vectors. Each neuron of a layer performs a dot product between the input vector and the neuron weight vector (followed by an activation function that suppresses weak outputs). The outputs of each neuron of a layer form again a vector. These are all operations on vectors.

Neural Networks are not tailored to process information well when it is input as a single scalar. They can process scalars, but most of their capacity comes from high-dimensional feature mixing.

In other words a single scalar is not the right vocabulary for expressing information in the language of neural network; they speak vectors, distributed representation of information.

This is why the tokens of the input sequence are transformed from one-hot vectors (non-distributed) to a distributed representation by the embeddings. These vectors represent word-tokens comprising semantics and syntax. They are learned during training, just like weights of the neurons are learned using training. They live in a space of reasonable dimension (in Bengio's model it was around 100, in LLMs it is from 1024 to 8192). We never really know how the model has decided to do this (after training), that is,

we cannot really put a precise name on most of each of the dimensions of this vector, but we know the representation is consistent and works: word-tokens can really be represented by vectors. They are used both in Bengio's model and in Transformers.

3.2.2 A word about adding meaning

In LLMs, one often hears and reads the expression "adding meaning" from another word-token, etc. Since tokens are represented as vectors, meaning can be combined by a natural operation: vector addition.

Adding meaning looks like this in the famous intuitive example:

$$\text{King} - \text{Man} + \text{Woman} = \text{Queen}.$$

We can also weight these additions:

$$\text{Queen} + 0.2 \times \text{Man} - 0.2 \times \text{Woman} = \text{Viril Queen}.$$

We could add to meaning something different than semantic, a bit of syntax; Man + object complement, meaning the word *man* as an object complement instead of subject.

One last example which will be relevant for rendering the context with attention (a key component of transformers): depending on whether the word "bank" is close to "river" or to "rob" or to "sit", it will have a different meaning although it is the same word. It could therefore be meaningful to add the context and create different state/vector representations:

$$\text{bank} + \text{river}, \quad \text{bank} + \text{rob}, \quad \text{bank} + \text{sit}$$

Adding vectors to add meaning is a fundamental hypothesis in LMs and LLMs. It is particularly present and used in Transformers. This addition seems to blur and loose the individual components in the sum, but the network does not need to recover and know the individual components; it is precisely their modified pattern which is going to be used for scoring and classification. This is the whole sense of vector addition.

3.2.3 Positional encoding

Because attention modifies (the meaning of) each token according to which tokens are in the sequence, one needs position encoding of the tokens within the sequence to know how far each of these tokens is from the considered token (if "river" is far away, i.e. 100 pages before, it is likely that it should not be related and modify the current word-token "bank"; if it is in the same sentence it is much more likely).

For this reason, it is needed to give each token an address in the sequence. Distributed representation of the position is used instead of just adding a dimension to the tokens comprising its numerical address exactly for the reason mentioned above. This is done by adding a position vector to the token, of the same dimension. This position vector is a unique vector for each position in the sequence (common encodings comprise sine and cosine functions with frequency increasing with the position).

3.3 From Bengio's model to Transformers

In the Bengio model, since the entire input sequence is used, the state vector's width is the length of sequence D times the dimension d of the tokens: $d \times D$. This can make the number of connections in the FNN huge (dense mixing) since each neuron has many more connections and weights. Especially with deeper networks, needed for longer input sequences. It renders this art of representation of the past—a concatenation of tokens—impossible to maintain over longer sequences.

A solution has been sought to avoid parameter explosion caused by dense mixing over a concatenated context vector, present in the Bengio architecture. This solution is attention.

Conceptually it goes like this: At each layer l, instead of taking a concatenated input sequence

$$u(1,l), u(2,l), u(3,l), \ldots, u(n,l), u(n+1,l)$$

and propagating it through the next layer, the idea is to extract all the relevant meaning (for the prediction) from the prefix sequence

$$u(1,l), u(2,l), u(3,l), \ldots, u(n,l)$$

and add it to the last state $u(n+1, l)$. This extraction and transfer is what attention is about.

Thus, if this succeeds, we keep at each layer l, $n+1$ state vectors of fixed dimension d_{model}, and apply the same layers to each position (and all the others until prediction, see right network of Figure 13 below). This prevents the parameter number from exploding as with a huge sequence vector and we are no longer limited by parameter explosion as the sequence grows; the remaining limitation becomes runtime/memory (attention cost), not model size.
This is the fundamental difference and advantage of transformers over the Bengio model.

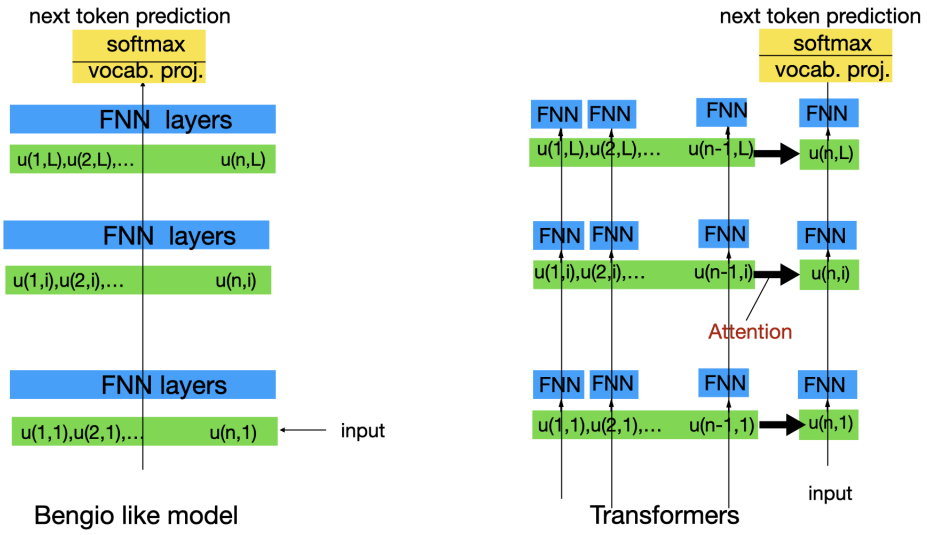


Figure 13: From Bengio's model to Transformers.

3.4 The attention mechanism

3.4.1 An attention head

We will now describe the mechanism implemented in a so called "attention head". Later we will see that attention is computed with several attention heads, each paying a different attention to the past sequence.

3.4.1.1 Query, Key, Value The Q, K, V model is a generic model for the transmission of information between vectorized elements E. In a language model, these elements are word-tokens (or token states).

The terms *query/key/value* echo database and information-retrieval language: a query matches keys to retrieve corresponding values (here learned continuous vectors, not discrete records).

3.4.1.2 The mathematical hypothesis Each element E_i is represented by a vector (its coordinates), i.e. by its state representation. To each element E_i we associate three vectors: a query vector q_i, a key vector k_i, and a value vector v_i.

- The query vector q_i specifies what kind of information the receiver element E_i is receptive to.
- The key vector k_j specifies what kind of information a source element E_j is able to offer.
- The value vector v_j specifies what information the source element E_j will effectively make available as transmissible payload.

These three vectors are assumed to be functions of the element state:

$$q_i = f_Q(E_i), \qquad k_j = f_K(E_j), \qquad v_j = f_V(E_j).$$

Since these objects are multi-dimensional, the simplest non-trivial model for such functions is a linear map, implemented by matrix multiplication. Therefore, we posit the existence of three trainable projection matrices:

$$q_i = E_i W_Q, \qquad k_j = E_j W_K, \qquad v_j = E_j W_V.$$

The compatibility between a receiver E_i and a source E_j is measured by a dot product score:

$$s_{i,j} = q_i^\top k_j,$$

(optionally scaled by $1/\sqrt{d_k}$ in practice, d_k being the dimension of the k and q vectors).

A high value of $s_{i,j}$ means that what E_i is currently receptive to matches well what E_j offers.

The final information transmitted from E_j to E_i is modeled by weighting the value v_j according to the normalized score:

$$m_i = \sum_j \alpha_{i,j}\, v_j,$$

where the weights are given by softmax:

$$\alpha_{i,j} = \mathrm{softmax}_j(s_{i,j}) = \frac{e^{s_{i,j}}}{\sum_r e^{s_{i,r}}}.$$

This defines a trainable mechanism for selecting and aggregating information from many sources into one receiver.

3.4.1.3 The intuitive example To make this model more intuitive, imagine people at a party, each represented by an element E_j. We are interested in what John (the receiver element E_i) will obtain from his interactions with the other people present.

3.4.1.3.1 Queries. John is receptive to certain kinds of information: cinema, AI, philosophy, politeness, humor, empathy, etc. These dimensions define the components of his query vector q_i. The coefficients indicate how strongly he is currently receptive to each of these topics.

3.4.1.3.2 Keys. Now consider another person, Jack (a source element E_j). Jack may have interests, competences, or traits that he can offer in conversation:: gardening, music, humor, extreme political opinions, expertise in AI, politeness, etc. These dimensions define the components of his key vector k_j. The coefficients indicate how strongly Jack embodies or expresses these features.

3.4.1.3.3 Matching. If John talks to Jack, the potential for meaningful interaction depends on how well John's receptivity matches Jack's offer. This is modeled by the dot product:

$$s_{i,j} = q_i^\top k_j.$$

If the score is high, John and Jack match strongly (for example in AI, humor, and politeness).

3.4.1.3.4 Values. However, even if Jack *could* provide certain information, he may not actually transmit it fully. The effective payload depends on Jack's current state and willingness to communicate. For example: is he tired, does he like John, does he want to discuss AI at a party, does he want to be discreet, etc.

This is what the value vector v_j represents: it is the mixture of information that Jack actually makes available to be transmitted.

Thus the information John receives from Jack is not only determined by the match score $s_{i,j}$, but also by the content v_j. We will see how in the next paragraph:

3.4.1.3.5 Aggregation: softmax again. John does not interact with only one person. He may speak with many people E_j. He will therefore receive many contributions, but not all with equal importance. Some interactions will be strong and memorable, others weak and negligible.

To model this selection effect, we normalize the scores using softmax:

$$\alpha_{i,j} = \frac{e^{s_{i,j}}}{\sum_r e^{s_{i,r}}}.$$

The final aggregated information John receives is then:

$$A(i) = \sum_j \alpha_{i,j}\, v_j.$$

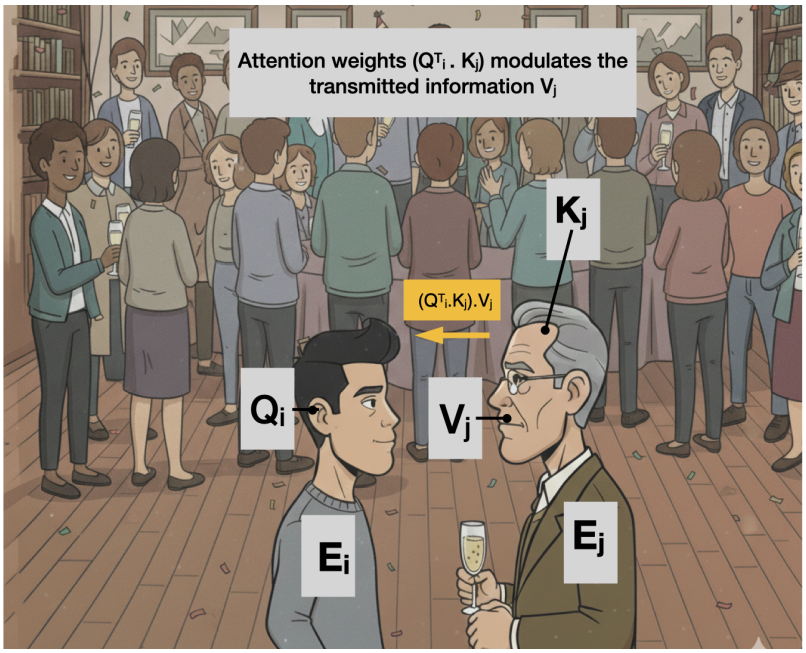


Figure 14: An Attention Party

3.4.1.3.6 Disclaimer. This example is not meant as a faithful model of real human conversation. It is only an intuitive analogy to make the generic Q, K, V mechanism understandable. In practice, the network discovers its own internal features during training.

In our party analogy we only use $A(i)$ as an abstract takeaway of what John gathered from his conversations. In Transformers, the attention output is position- and layer-dependent, $A(i, l)$, and is added to the state $u(i, l)$ via a residual connection. In generation mode (as in our figure), we only track the newest token position, so the index i is fixed and we write simply $A(l)$.

This mechanism is the essence of attention: it defines, from vector representations, a general and trainable rule for routing information from many sources into one receiver.

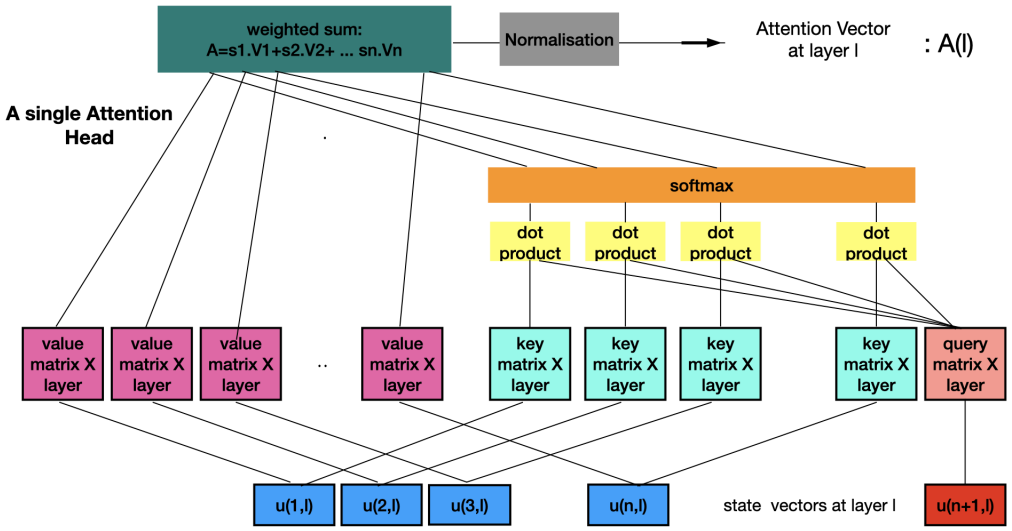


Figure 15: An Attention Head

3.4.2 Multi-head attention

It has been shown advantageous instead of having one attention head attending all dimensions of the token representations of the past sequence, to have several heads, typically 16. Each head thereby projects on different subspaces of the representation and recombines their outputs by concatenation and projection onto a layer of the dimension of the state vector. This allows different perspectives to be taken.

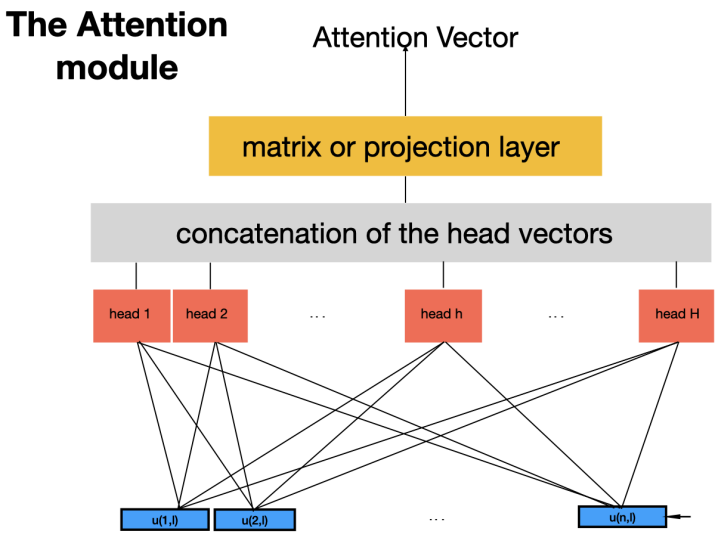


Figure 16: Multihead Attention

This procedure is very similar to what is done in FNNs where an input vector is projected onto the weight/feature vectors of the neurons of the following layer to extract features.

3.5 The Transformer block

After the attention contribution vector has been added to the residual state vector $u(n, l)$, the state vector is normalized and input into a 2-layer FNN for re-representation (as in the Bengio model). Both the attention module and the FNN constitute what is called a Transformer block. It plays the role of a dense hidden layer in a Bengio-like model. Its output, the FNN contribution, is added to the residual state vector $u(n, l)$ and the sum is normalized before entering the next Transformer block.

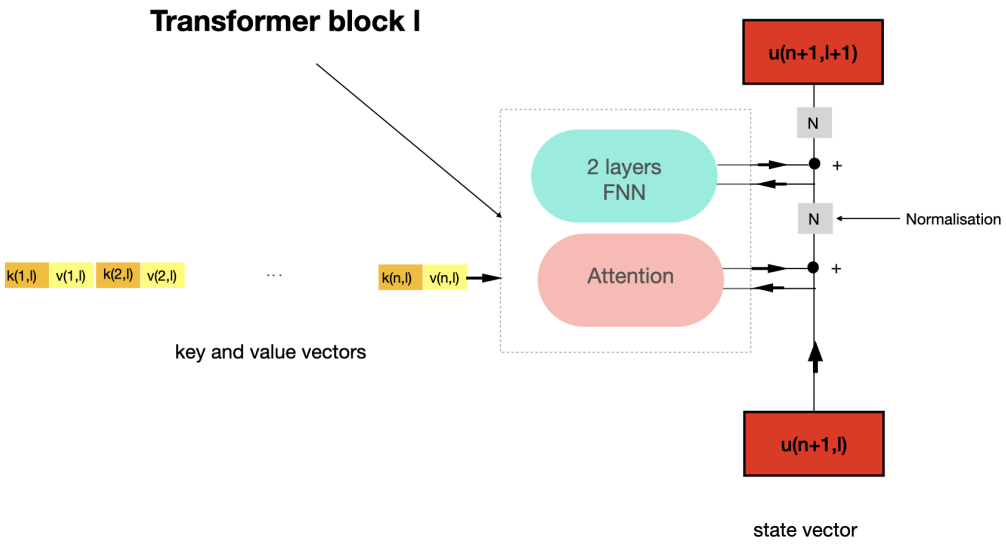


Figure 17: The Transformer block

3.5.1 Why so many Transformer blocks?

In the Bengio model, each layer mixes information from all context tokens because the dense wiring spans the entire concatenated input; therefore, a scalable replacement (like Transformers) must also mix across tokens at multiple representational stages. Transformers recover this by alternating attention (horizontal mixing) and MLPs (vertical re-representation) through depth.

The reason there are so many Transformer blocks stacked upon each other lies in the fact that the attention mechanism cannot transfer (horizontally) as much information to the state vector as the Bengio model does across token states. In the Bengio model all neurons of a layer can input in all neurons of the next layer; this is the maximum on transfer and connectivity, vertically and horizontally. In comparison, the attention mechanism is a bottleneck since it can only transmit:

$$m_i = \sum_j \alpha_{i,j} \, (u_j W_V),$$

which is much more constrained.

This stack of Transformer blocks is called a Transformer.

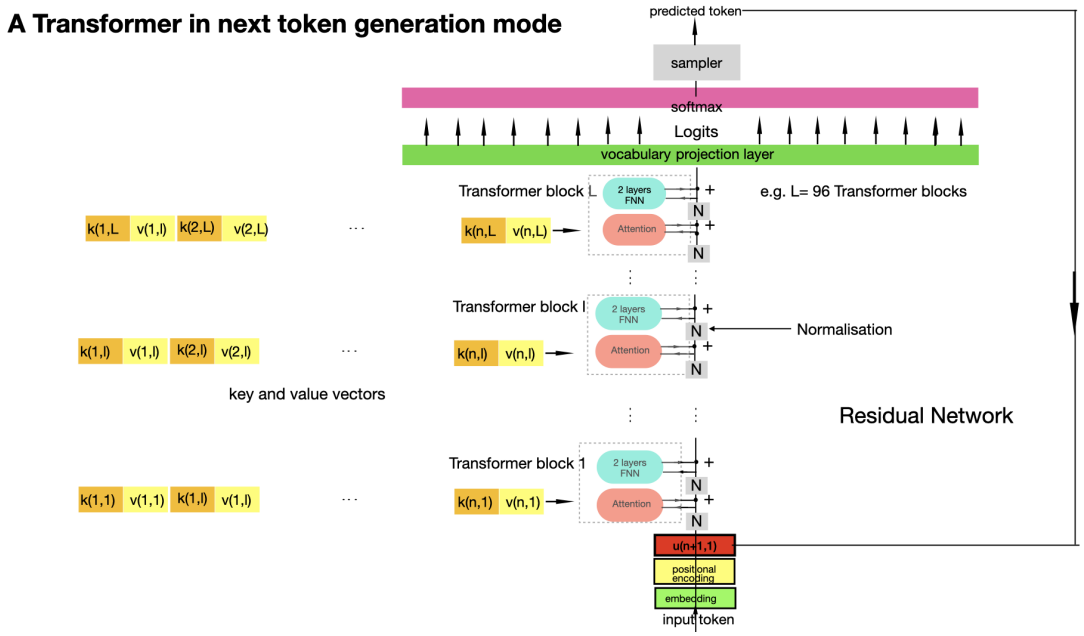


Figure 18: A Transformer

In generation mode all previous key vectors are stored at each layer and used to compute attention for the states of the new token u(n+1,l), l=1 to L. Only the newly updated u(n+1,l) state vector is propagated through the FNNs.

In prompt and training mode, the entire prompt or training sequence of the n vector states is propagated through n parallel identical FNNs.

3.5.2 Why prediction starts at the new token?

In Transformers, the state at the predicting position is always a compound of the newest token's current representation and information imported from the prefix. This composition is implemented by the residual addition:

$$u_n^{(l)} = u_n^{(l-1)} + \mathrm{Attn}_n^{(l)}\big(u^{(l-1)}\big) + \cdots$$

Because attention enters as an added term, it is often described as an "update," with the current state vector as the "base." This interpretation is natural in upper layers, where $u_n^{(l-1)}$ has already become largely a predictive vector: it already contains a distilled summary of the past, and attention refines it.

At the very bottom, however, this language can sound wrong: the newest token embedding cannot plausibly be the "reference" for predicting the next token. The resolution is to remember that, as a sum, the residual decomposition is not a statement about conceptual priority. At early layers, the functional base for prediction is the prefix contribution carried by attention, while the newest token acts more like a conditioning cue or anchor. As depth increases, the balance shifts: the state at the last position progressively internalizes more of the prefix and becomes increasingly prediction-shaped, so attention becomes more and more a refinement term relative to an already predictive state.

3.6 Conclusion on Transformers

Though the design of Transformers might still appear somehow empirical, we hope to have uncovered their logic and lifted this mysterious and impressing veil around them. They removed the limitations of the Bengio model and enabled Large Language Models such as ChatGPT.

4 Post-training

While training the model on billions of lines of text creates a model capable of generating coherent text, there is still one part missing to arrive at what we are accustomed to nowadays with ChatGPT and similar LLM chatbots: responding precisely and helpfully to human prompts. This is achieved through post-training.

The first phase is Supervised Fine-Tuning (SFT), in which the model is trained on interaction-style data — prompts, questions, instructions, and desired answers written or curated by human annotators. The objective is no longer only to predict the most likely next token from the internet distribution, but to teach the model to follow instructions, structure its answers, and provide responses that match human expectations.

The second phase is Reinforcement Learning from Human Feedback (RLHF), in which human annotators rank multiple model outputs for the same prompt. These preferences are used to train a reward model, which the language model is then optimized against using reinforcement learning algorithms such as PPO or GRPO.

The full details of these methods go beyond the scope of this article and are mentioned here only for completeness. The result of this pipeline is that the raw next-token predictor learns to align with user intent, avoid harmful outputs, and behave as a useful conversational assistant.

5 Conclusion

We are at the end of our introductory journey into LLMs, and I hope you now have a basic understanding of them, which I would define as a compressed repository of the human intelligence present in the billions of lines of text they were trained on — more a mirror of human linguistic intelligence than an intelligence in its own right.

It is worth remembering that their architecture and principles are not limited to language, but find application in image, video, and music generation; robotics; prediction of protein properties; medical diagnosis; drug discovery; genomics; and many other areas still to be explored. Any stream of data that has some structure — and there are many in nature — can be modeled by these principles, enabling prediction or generation. This is what makes them so powerful, and the center of current AI.

We are perhaps only at the beginning of what these principles can achieve.

EOS

A print edition of this work will be available on Amazon in March 2026.

Appendices

A Linear Algebra Notation Used in This article

This appendix summarizes the linear algebra notation used throughout the article. We focus on matrices as arrays of coefficients and on how multiplications are defined.

Scalars, vectors, matrices

A *scalar* is a single real number $s \in \mathbb{R}$.

A *column vector* $x \in \mathbb{R}^n$ is an ordered list of n real numbers:

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}.$$

In "shape" terms, x is an $n \times 1$ object.

A *row vector* is a $1 \times n$ array, typically obtained by transposition:

$$x^\top = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \end{pmatrix}.$$

A *matrix* $A \in \mathbb{R}^{m \times n}$ is an array of coefficients a_{ij} arranged in m rows and n columns:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}.$$

The index i refers to the row, and j to the column.

Dot product (written out)

Let

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \qquad y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \in \mathbb{R}^n.$$

The *dot product* of x and y is the scalar

$$x \cdot y \;=\; x^\top y \;=\; x_1 y_1 + x_2 y_2 + \cdots + x_n y_n \;=\; \sum_{j=1}^{n} x_j y_j.$$

So it is a sum of products: multiply the first components, then the second, etc., and add everything.

Matrix–vector multiplication (written out, with standard notation)

Let

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \qquad x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}.$$

Then the product is written in the standard compact form

$$y = Ax,$$

where $y \in \mathbb{R}^m$ is the output vector:

$$y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} = \begin{pmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{pmatrix}.$$

In words: each y_i is the dot product of row i of A with the vector x.

Vector–matrix multiplication (written out)

Most of the time in this article we use column vectors, and linear layers are written as $y = Ax$. However, in some derivations (for example when writing dot products, or when expressing derivatives compactly), it is convenient to write vectors as *row vectors*. A row vector is simply the transpose of a column vector. In that notation, the corresponding linear operation is written $y^\top = x^\top A$. It is the same idea, just written with the vector on the left.

Let $x^\top \in \mathbb{R}^{1\times n}$ be a row vector and $A \in \mathbb{R}^{n\times m}$ a matrix:

$$x^\top = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \end{pmatrix}, \qquad A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix}.$$

Then the product is written compactly as

$$y^\top = x^\top A,$$

where

$$y^\top = \begin{pmatrix} y_1 & y_2 & \cdots & y_m \end{pmatrix} = \begin{pmatrix} x_1a_{11} + x_2a_{21} + \cdots + x_na_{n1} & x_1a_{12} + x_2a_{22} + \cdots + x_na_{n2} & \cdots & x_1a_{1m} + x_2a_{2m} + \cdots + x_n$$

So each output component is

$$y_k = x_1a_{1k} + x_2a_{2k} + \cdots + x_na_{nk}.$$

In words: each y_k is the dot product of the row vector $x^\top$ with column k of A.

Matrix–matrix multiplication (written out)

Let

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \in \mathbb{R}^{m\times n}, \qquad B = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix} \in \mathbb{R}^{n\times p}.$$

Then the product is written compactly as

$$C = AB,$$

where $C \in \mathbb{R}^{m\times p}$ is

$$C = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}.$$

Each coefficient is computed as a dot product:

$$c_{ik} = a_{i1}b_{1k} + a_{i2}b_{2k} + \cdots + a_{in}b_{nk}.$$

So c_{ik} is the dot product of row i of A with column k of B.

Transpose of a matrix (written out)

Let

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \in \mathbb{R}^{m \times n}.$$

The *transpose* $A^\top$ is obtained by turning rows into columns (swapping indices):

$$A^\top = \begin{pmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{pmatrix} \in \mathbb{R}^{n \times m},$$

so that $(A^\top)_{ij} = A_{ji}$.

Two identities that are used constantly are:

$$(A^\top)^\top = A, \qquad (AB)^\top = B^\top A^\top.$$

Gradients and shapes

If L is a scalar loss and $x \in \mathbb{R}^n$, then the gradient of L with respect to x is

$$\nabla_x L = \begin{pmatrix} \frac{\partial L}{\partial x_1} \\ \frac{\partial L}{\partial x_2} \\ \vdots \\ \frac{\partial L}{\partial x_n} \end{pmatrix} \in \mathbb{R}^n.$$

So $\nabla_x L$ has the same dimension as x.

Similarly, if $W \in \mathbb{R}^{m \times n}$, then

$$\frac{\partial L}{\partial W} \in \mathbb{R}^{m \times n},$$

meaning that the gradient has the same shape as the parameter matrix. This simple "shape logic" is extremely useful when checking whether an equation in backpropagation is consistent.

B Backpropagation (multi-layer, mini-batches, Jacobians)

Consider an L-layer feed-forward network. For a mini-batch of size B, we stack the examples column-wise:

$$A^{(0)} = X \in \mathbb{R}^{d_0 \times B}.$$

For each layer $l = 1, \dots, L$ we compute

$$Z^{(l)} = W^{(l)} A^{(l-1)} + b^{(l)} \mathbf{1}^\top, \qquad A^{(l)} = \phi^{(l)}\Big(Z^{(l)}\Big),$$

where $W^{(l)} \in \mathbb{R}^{d_l \times d_{l-1}}$, $b^{(l)} \in \mathbb{R}^{d_l}$, $\mathbf{1} \in \mathbb{R}^B$ is the all-ones vector, and $\phi^{(l)}$ is applied elementwise.

Let the batch loss be

$$\mathcal{L} = \frac{1}{B} \sum_{i=1}^{B} \ell\Big(a_i^{(L)}, y_i\Big),$$

where $a_i^{(L)}$ is the i-th column of $A^{(L)}$.

Backpropagated sensitivities.

Define the *error signal* (sometimes called "delta") at layer l by

$$\Delta^{(l)} \;=\; \frac{\partial \mathcal{L}}{\partial Z^{(l)}} \in \mathbb{R}^{d_l \times B}.$$

Then, by the chain rule in matrix (Jacobian) form,

$$\frac{\partial \mathcal{L}}{\partial A^{(l-1)}} \;=\; \Big(W^{(l)}\Big)^\top \Delta^{(l)}.$$

Moreover, because $A^{(l)} = \phi^{(l)}(Z^{(l)})$ elementwise, we have

$$\Delta^{(l)} \;=\; \frac{\partial \mathcal{L}}{\partial A^{(l)}} \odot \phi^{(l)\,\prime}\Big(Z^{(l)}\Big),$$

where $\odot$ denotes elementwise (Hadamard) multiplication.

Backward recursion (layer by layer).

Starting from the last layer, we obtain $\Delta^{(L)}$, and then for $l = L-1, \dots, 1$:

$$\Delta^{(l)} \;=\; \Big(\Big(W^{(l+1)}\Big)^\top \Delta^{(l+1)}\Big) \odot \phi^{(l)\,\prime}\Big(Z^{(l)}\Big).$$

Gradients with respect to parameters.

For each layer l, the gradients of the batch loss are

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} \;=\; \frac{1}{B}\, \Delta^{(l)} \Big(A^{(l-1)}\Big)^\top, \qquad \frac{\partial \mathcal{L}}{\partial b^{(l)}} \;=\; \frac{1}{B}\, \Delta^{(l)} \mathbf{1}.$$

These are exactly the quantities needed to update the parameters by gradient descent (or a variant such as Adam).

Softmax + cross-entropy (common output layer).

If the last layer produces logits $Z^{(L)} \in \mathbb{R}^{K \times B}$, probabilities $P = \mathrm{softmax}(Z^{(L)})$, and one-hot targets $Y \in \mathbb{R}^{K \times B}$, then the derivative simplifies to

$$\Delta^{(L)} \;=\; \frac{1}{B}\,(P - Y),$$

which makes the final-layer gradients particularly simple.

***

References

[1] David E. Rumelhardt, Geoffrey E. Hinton & Ronald J.Williams, *Learning representations by back-propagating errors* Geoffrey Hinton's University of Toronto Page https://www.cs.toronto.edu/~hinton/absps/naturebp.pdf

[2] Y. Bengio, R. Ducharme, P. Vincent, C. Jauvin, *A Neural Probabilistic Language Model*, Journal of Machine Learning Research, 2003. https://dl.acm.org/doi/pdf/10.5555/944919.944966

[3] C. E. Shannon, *A Mathematical Theory of Communication*, Harvard University, 1948.Reprinted with corrections from The Bell System Technical Journal, Vol. 27, pp. 379–423, 623–656, July, October, 1948. https://people.math.harvard.edu/~ctm/home/text/others/shannon/entropy/entropy.pdf

[4] John S. Bridle, *Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition*, 1990. https://link.springer.com/chapter/10.1007/978-3-642-76153-9_28

[5] A. Vaswani et al., *Attention Is All You Need*, NeurIPS, 2017. https://arxiv.org/abs/1706.03762

