International Journal of Semantic Computing
(2024) 1–44
© World Scientific Publishing Company
DOI: 10.1142/S1793351X24300085

LLMs: Their Past, Promise, and Problems

George F. Luger

Professor Emeritus, Department of Computer Science
University of New Mexico Albuquerque, NM 87113, USA
luger@cs.unm.edu
https://www.cs.unm.edu/~luger

Received 18 July 2024
Accepted 25 July 2024
Published

Transformer-based large language models are currently at the forefront of modern artificial intelligence. Their prominence followed from the seminal paper *Attention is All You Need* [1]. Vaswani and his colleagues suggested placing attention mechanisms within the encoder and decoder modules of autoencoders rather than using them to focus between these two modules. In this paper we present first the seminal insights of early AI that lead to deep learning. We then describe the mathematical tools necessary for understanding the current generation of LLMs and follow this with a brief description of the transformer architecture. We then provide examples of LLMs in action and conclude with some observations of their promise and problems.

Keywords: Large language models; attention; transformer.

1. Introduction

Neural Networks, often characterized as *neurally inspired computation*, or *parallel distributed processing*, de-emphasize the explicit use of symbols and logic-based reasoning. Neural network approaches are designed to capture relations and associations within an application domain and interpret new situations in the context of previously learned relational patterns.

The neural net philosophy conjectures that intelligence arises in systems of simple interacting components, biological or artificial neurons. This happens through a process of learning or adaptation by which the connections between components are adjusted as patterns in the world are processed. Computation in these systems is distributed across collections, or layers, of neurons. Problem solving is parallel in the sense that all the neurons within the collection or layer process their inputs simultaneously and independently. These systems also degrade gracefully since information and processing are distributed across nodes and layers and not localized to any single component of the network.

2430008.indd 1 24-09-2024 11:23:40

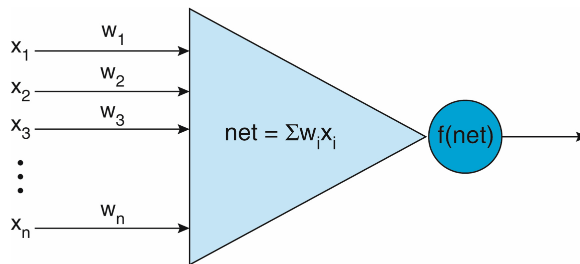


Fig. 1. An artificial neuron with input vector x_i, weights w_i for each input, and a threshold function f that determines the neurons output value. Figure adapted from [2].

The algorithms and architectures that implement connectionist techniques are usually trained or conditioned rather than explicitly programmed. This fact is a major strength of the approach, as an appropriately designed network architecture and learning algorithm can often capture invariances in the world, even in the form of strange attractors, without being explicitly programmed to recognize them.

The basis of a network is the artificial neuron, as shown in Fig. 1.

The minimal components of the artificial neuron are:

(1) *Input signals* X_i. These signals may come from the environment or from the activation of other neurons. Different models vary in the allowable range of the input values; typically, inputs are discrete, from the sets {0, 1} or {–1, 1}.
(2) A set of real-valued *weights,* w_i. These values describe connection strengths.
(3) An *activation level,* $\Sigma w_i x_i$. The neuron's activation level is determined by the cumulative strength of its input signals where each signal is scaled, or multiplied, by the connection weight associated with that input. The activation level is computed by taking the sum of the weighted inputs, that is $\Sigma w_i x_i$. The Greek sigma, Σ, indicates that these values are summed.
(4) *A threshold or a bounded nonlinear mapping function, f*. The threshold function computes the neuron's output by seeing if it is above an activation level. The nonlinear mapping function produces either an on/off or a graded response for that neuron.

Early examples of neural computing are the McCulloch–Pitts [3] neurons. The inputs of these neurons are either +1, i.e. *true*, or –1, *false*. The activation function multiplies each input by its corresponding weight and adds the results; if this sum is greater than or equal to zero, the neuron returns 1, true, otherwise, –1, false. McCulloch and Pitts showed how these neurons could be constructed to compute any logical function.

Figure 2 shows McCulloch–Pitts neurons for computing the logical functions *and* (∧) and *or* (∨). The *and* neuron, (a) on the left, has three inputs: x and y are the values to be conjoined; the third input, sometimes called a *bias*, has a constant value of +1. The input data and bias have weights of +1, +1, and –2, respectively.

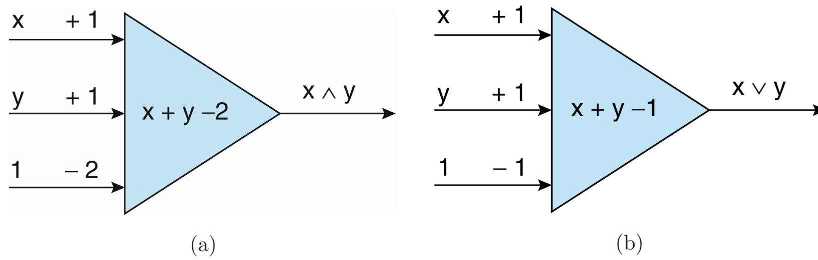


Fig. 2. McCulloch–Pitts neurons for *and* (a), and *or* (b). Figure adapted from [2].

Table 1. The McCulloch–Pitts model for computing the logical *and* of Fig. 2(a).

x	y	$x = y - 2$	Output
1	1	0	1
1	0	–1	–1
0	1	–1	–1
0	0	–2	–1

Thus, for any input values of x and y, the neuron computes $x + y - 2$. Table 1 shows that if this value is less than 0, it returns –1, false, otherwise a 1, true. The *or* neuron, (b) on the right, illustrates the neuron computing $x \vee y$. The weighted sum of input data for the $\vee$ neuron is greater than or equal to 0 unless both x and y equal –1, i.e. are false.

Although McCulloch and Pitts demonstrated the power of neural computation, interest in neural network research only began to flourish with the development of practical learning algorithms. Early learning models drew heavily on the work of the psychologist Donald Hebb [4], who speculated that learning occurred in brains through the modification, or conditioning, of synapses. Hebb stated:

> When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes place in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.

Neural physiological research has confirmed Hebb's idea that temporal proximity of the firing of connected neurons can modify their synaptic strength, albeit in a more complex fashion than Hebb's intuition of "increase in efficiency". We next demonstrate *Hebbian learning*, which belongs to the *coincidence* class of learning laws. This learning produces weight changes in response to localized events in neural processing.

Neural network learning may be unsupervised, supervised, or some hybrid combination of the two. The examples seen so far are unsupervised, as the network and its weights transformed input signals to the desired output values. We now consider an example of unsupervised Hebbian learning where each output has a weight

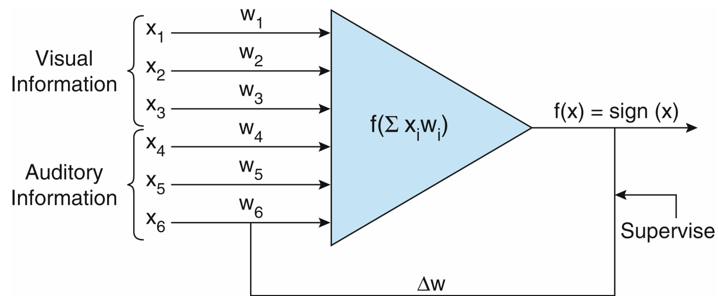


Fig. 3. A Hebbian network, with no extra-network supervision, that learns a response for an unconditioned stimulus. ΔW adjusts the weights at each iteration of the data through the network. The threshold function is shown in Fig. 7(a). Figure adapted from [2].

adjustment factor. In unsupervised learning, a critic is not available to provide the "correct" output value. As a result, the weights must be modified across multiple iterations solely as a function of the input and output values of the neuron. The training of the Hebbian network of Fig. 3 has the effect of strengthening the network's responses to patterns that it has already seen and interpreting new patterns appropriately.

Figure 3 demonstrates how Hebbian techniques can be used to model *conditioned response learning*, where an arbitrarily selected stimulus can be used to condition a desired response. Pavlov's classic 1890's experiment offers an example of a conditioned response. A dog is brought food while a bell is rung. The dog salivates in expectation of his meal. The unconditioned response of the salivating animal is the presence of food. After several instances where the arrival of food is accompanied by the ringing bell, the bell is rung without food. The dog salivates. The ringing bell produces the dog's conditioned response!

Figure 3 demonstrates how a Hebbian network can transfer a response from a primary or unconditioned stimulus to a conditioned stimulus. In Pavlov's experiments, the dog's salivation response to food is transferred to the bell. Weight adjustment, ΔW, at each network iteration, is described by the equation:

$$\Delta W = c * f(X, W) * X.$$

In this equation c is the learning constant, a small positive decimal, whose use modulates the extent of the learning at each step, as described with more detail in Fig. 3, $f(X, W)$ is the network's output at each iteration, and X_i is the input vector at that iteration.

The network of Fig. 3 has two layers, an input layer with six nodes and an output layer with one node. The output layer returns either +1, signifying that the output neuron has fired, or $a - 1$ that it has not fired. The feedback, Supervise, monitoring the network, ΔW, takes each output of the network and multiplies it by the input vector and the learning constant to produce the set of weights for the input vector at the next iteration of the network.

We set the learning constant to the small positive real number, 0.2. In this example we train the network on the pattern [1, –1, 1, –1, 1 –1] which joins the two patterns, [1, –1, 1] and [–1, 1, –1]. The pattern [1, –1, 1] represents the unconditioned stimulus and [–1, 1, –1] represents the new stimulus.

Assume that the network already responds positively to the unconditioned stimulus but is neutral with respect to the new stimulus. We simulate the positive response of the network to the unconditioned stimulus with the weight vector [1, –1, 1] exactly matching the input pattern. The neutral response of the network to the new stimulus is simulated by the weight vector [0, 0, 0]. Joining these two vectors gives the initial weight vector for the network, [1, –1, 1, 0, 0, 0].

The network is next trained on the input pattern, hoping to induce a configuration of weights that will produce a positive network response to the new stimulus. The first iteration of the network produces the result:

$$W * X = (1 * 1) + (-1 * -1) + (1 * 1) + (0 * -1) + (0 * 1) + (0 * -1) = (1) + (1) + (1) = 3, \\ \text{and } f(3) = \text{sign}(3) = 1.$$

Now the Hebbian network creates the new weight vector W^2:

$$\begin{aligned} W^2 &= [1, -1, 1, 0, 0, 0] + 0.2 * (1) * [1, -1, 1, -1, 1, -1] \\ &= [1, -1, 1, 0, 0, 0] + [0.2, -0.2, 0.2, -0.2, 0.2, -0.2] \\ &= [1.2, -1.2, 1.2, -0.2, 0.2, -0.2]. \end{aligned}$$

Next, the adjusted network sees the original input pattern with the new weights:

$$\begin{aligned} W * X &= (1.2 * 1) + (-1.2 * -1) + (1.2 * 1) + (-0.2 * -1) + (0.2 * 1) + (-0.2 * -1) \\ &= (1.2) + (1.2) + (1.2) + (0.2) + (0.2) + (0.2) \\ &= 4.2, \text{ and sign}(4.2) = 1. \end{aligned}$$

Now the Hebbian network creates the new weight vector W^3:

$$\begin{aligned} W^3 &= [1.2, -1.2, 1.2, -0.2, 0.2, -0.2] + 0.2 * (1) * [1, -1, 1, -1, 1 -1] \\ &= [1.2, -1.2, 1.2, -0.2, 0.2, -0.2] + [0.2, -0.2, 0.2, -0.2, 0.2, -0.2] \\ &= [1.4, -1.4, 1.4, -0.4, 0.4, -0.4]. \end{aligned}$$

It can now be seen that the weight vector product, $W * X$, will continue to grow in the positive direction, with the value of each element of the weight vector increasing by 0.2 in the + or − direction, at each training cycle. After 10 more iterations of Hebbian training the weight vector will be

$$W^{13} = [3.4, -3.4, 3.4, -2.4, 2.4, -2.4].$$

We use this trained weight vector to test the network's response to the two partial patterns. We would like to see if the network continues to respond to the unconditioned stimulus positively and, more importantly, if the network has now acquired a positive response to the new conditioned stimulus. We test the network first on the unconditioned stimulus [1, –1, 1]. We fill out the last three arguments

of the input vector with random 1 and –1 assignment, for example, we test the network on the vector [1, –1, 1, 1, 1, –1]:

$$\begin{aligned}\text{sign}(W^*X) &= \text{sign}((3.4^*1) + (-3.4^*-1) + (3.4^*1) + (-2.4^*1) + (2.4^*1) + (-2.4^*-1)) \\ &= \text{sign}(3.4 + 3.4 + 3.4 - 2.4 + 2.4 + 2.4) = \text{sign}(12.6) = +1.\end{aligned}$$

The network still responds positively to the original unconditioned stimulus. We next do a second test using the original unconditioned stimulus and a different random vector in the last three positions [1, –1, 1, 1, –1, –1]:

$$\begin{aligned}\text{sign}(W^*X) &= \text{sign}((3.4^*1) + (-3.4^*-1) + (3.4^*1) + (-2.4^*1) + (2.4^*-1) + (-2.4^*-1)) \\ &= \text{sign}(3.4 + 3.4 + 3.4 - 2.4 - 2.4 + 2.4) = \text{sign}(7.8) = +1.\end{aligned}$$

The second vector also produces a positive network response. With these two examples the network's sensitivity to the original stimulus has been strengthened, due to repeated exposure to that stimulus.

We next test the network's response to the new stimulus pattern, [–1, 1, –1], encoded in the last three positions of the input vector. We fill the first three vector positions with random assignments from the set {1, –1} and test the network on the vector [1, 1, 1, –1, 1, –1]:

$$\begin{aligned}\text{sign}(W^*X) &= \text{sign}((3.4^*1) + (-3.4^*-1) + (3.4^*1) + (-2.4^*1) + (2.4^*1) + (-2.4^*-1)) \\ &= \text{sign}(3.4 - 3.4 + 3.4 + 2.4 + 2.4 + 2.4) = \text{sign}(10.6) = +1.\end{aligned}$$

We do one final experiment, with the vector patterns slightly degraded. This could represent the stimulus situation where the input signals are altered, perhaps because a new food and/or a different sounding bell is used. We test the network on the input vector [1, –1, –1, 1, 1, –1], where the first three parameters are one digit off the original unconditioned stimulus and the last three parameters are one digit off the conditioned stimulus:

$$\begin{aligned}\text{sign}(W^*X) &= \text{sign}((3.4^*1) + (-3.4^*-1) + (3.4^*1) + (-2.4^*1) + (2.4^*1) + (-2.4^*-1)) \\ &= \text{sign}(3.4 + 3.4 - 3.4 - 2.4 + 2.4 + 2.4) = \text{sign}(5.8) = +1.\end{aligned}$$

Even this partially degraded stimulus is recognized.

What has the Hebbian learning model produced? We created an association between a new stimulus and an old response by repeatedly presenting the old and new stimuli together. The network learns to transfer its response to the new stimulus without any external supervision. This strengthened sensitivity also allows the network to respond in the same way to a slightly degraded version of the stimulus. This was accomplished by using Hebbian coincidence learning to increase the strength of the network's response to the total pattern. This increases the strength to each individual component of the pattern: an example of *self-organization* emerging from using Hebb's rule. The pattern of the secondary stimulus is also recognized!

$$= \text{sign}(10.6) = +1.$$

In 1958 Rosenblatt [5, 6] created the Perceptron, an electronic device inspired by neurologic principles. Rosenblatt was a psychologist and neuroscientist who, in

1959, became the director of Cornell's *Cognitive Systems Research Program*. The perceptron network consists of a single layer of N perceptron neurons activated by n inputs each with a weight, w_n, as shown in Fig. 1. Rosenblatt's 1962 paper describing the perceptron is titled *Principles of neurodynamics: Perceptrons and the theory of brain mechanisms*.

Interestingly, many early researchers, the precursors of neural network technology, claimed the inspiration of human neural activity for their creations. In the 19th century, Boole, the creator of the algebraic system supporting modern computation, and who's logic was first automated by McCulloch–Pitts neurons, offers a prime example. In the first chapter of *An Investigation of the Laws of Thought, on which are founded the Mathematical Theories of Logic and Probabilities* [7], Boole described his goal as

> ... to investigate the fundamental laws of those operations of the mind by which reasoning is performed: to give expression to them in the symbolical language of a Calculus, and upon this foundation to establish the science of logic and instruct its method; ... and finally to collect from the various elements of truth brought to view in the course of these inquiries some probable intimations concerning the nature and constitution of the human mind.

Perceptrons were initially greeted with enthusiasm. However, Nilsson [8] and others analyzed the limitations of the perceptron model. They demonstrated that perceptrons could not solve a certain difficult class of problems, namely problems in which the data points cannot be linearly separated in the dimensionality of the original problem statement. Although various enhancements of the perceptron model, including multilayered perceptrons, were envisioned at the time, Minsky and Papert, in their book *Perceptrons* [9], argued that the linear separability problem could not be overcome by any form of the then current perceptron network.

An example of nonlinearly separable classification is the *exclusive-or* problem of Table 2.

Consider a perceptron with two inputs, x_1, x_2, two weights, w_1, w_2, and threshold t. To learn this function, a network must find a weight assignment that satisfies the following inequalities, seen graphically in Fig. 4:

$w_1 * 1 + w_2 * 1 < t$, from line 1 of the truth table,
$w_1 * 1 + 0 > t$, from line 2 of the truth table,
$0 + w_2 * 1 > t$, from line 3 of the truth table,
$0 + 0 < t$, or t must be positive, from the last line of the table.

Table 2. The truth table for the exclusive-or operator.

x_1	x_2	Output
1	1	0
1	0	1
0	1	1
0	0	0

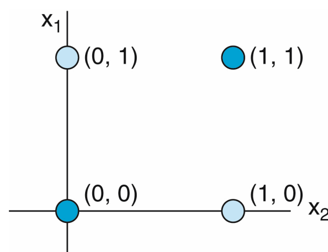


Fig. 4. The exclusive-or problem. No straight line on the two-dimensional grid can separate the (0, 1) and (1, 0) data points from (0, 0) and (1, 1). Figure adapted from [2].

This series of equations on w_1, w_2 and t has no solution, proving that a perceptron that solves exclusive-or is impossible. Although multilayer networks would eventually be built that could solve the exclusive-or problem, as we see in Sec. 2.1, the perceptron learning algorithm only worked for single layer networks. What makes exclusive-or impossible for the perceptron is that the two classes to be distinguished are not *linearly separable*. This can be seen in Fig. 4. It is impossible to draw a straight line in two dimensions that separates the data points {(0,0), (1,1)} from {(0,1), (1,0)}.

We may think of the set of data values for a network as defining a space. Each parameter of the input data corresponds to one dimension, with each input value defining a point in the space. In the exclusive-or example, the four input values, indexed by the x_1, x_2 coordinates, make up the data points of Fig. 4. The problem of learning a binary classification of the training instances reduces to that of separating these points into two groups. For a space of n dimensions, a classification is linearly separable if its classes can be separated by an $(n-1)$-dimensional hyperplane. In two dimensions an $(n-1)$-dimensional hyperplane is a line; in three dimension it is a plane, etc.

As a result of the linear separability limitation, research shifted toward work in symbol-based architectures, slowing progress in the connectionist methodology. Subsequent work in the 1980s and 1990s has shown these problems to be solvable, however; see [10–12]. We next discuss *backpropagation*, an extension of perceptron learning that works for multilayered networks.

The neurons in these networks, seen in the multi-layer perceptron of Fig. 5, are connected in layers, with units within layer n passing their activations only to neurons in layer $n+1$. Multilayer signal processing means that errors deep in the network can spread and evolve in complex, unanticipated ways throughout the layers. Thus, the analysis of the source of final output error and connecting that error back to the network nodes that produced it is complex. *Backpropagation* is an algorithm for apportioning this blame and adjusting the network's weights accordingly.

The historical emergence of networks with continuous activation functions suggested new approaches to error reduction learning. For example, the Widrow–Hoff [13] learning rule is independent of the activation function, minimizing the squared

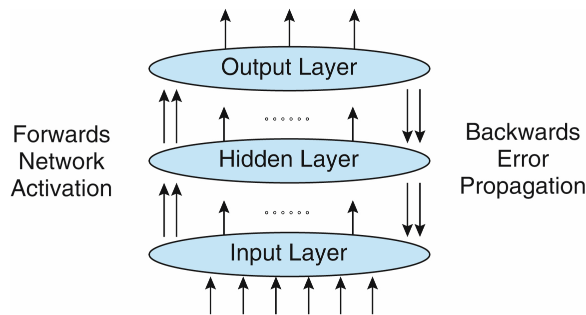


Fig. 5. A schema for a multi-hidden-layer neural network. Backward error propagation is addressed with the algorithms of Sec. 2.1. Figure adapted from [2].

error between the desired output value and the network activation, $\text{net}_i = WX_i$. A form of Widrow–Hoff became the most important learning rule for continuous activation functions. This is the *delta rule* [14].

Intuitively, the delta rule is based on the idea of an error surface, as illustrated in Fig. 6. This error surface represents cumulative error over a data set as a function of a network's weights. For example, if there were six weights to be conditioned, as in our XOR example in Sec. 2.1, the error measure would make up the 7th dimension of the error space. Each network weight configuration is represented by a point on this n-dimensional error surface. Given a particular weight configuration, we want our learning algorithm to find the direction on this surface which most rapidly reduces the error. This approach is called *gradient descent learning* because the gradient is a measure of slope, as a function of direction, from a point on a surface.

Backpropagation requires supervised data, e.g. a classifier is trained on labeled data. For example, a radiologist might have thousands of X-rays that reflect tumors while other X-rays are tumor free. Likewise, the welder may have thousands of examples of acceptable and unacceptable welds. Once these networks are trained,

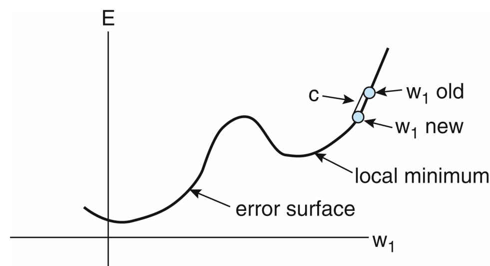


Fig. 6. An error surface in two dimensions. The dimension of a problem's error space is the number of weights involved plus the error measure. The learning constant c controls the step size taken on the error surface with each iteration of network learning. The goal is to find the value of W_1 where E is at a minimum. Figure adapted from [2].

however, they will be scanning totally new situations, examples they have never considered before. The classifier must decide each new situation and label it as either good or not.

2. What Must Be Known to Understand the Current Generation of LLMs?

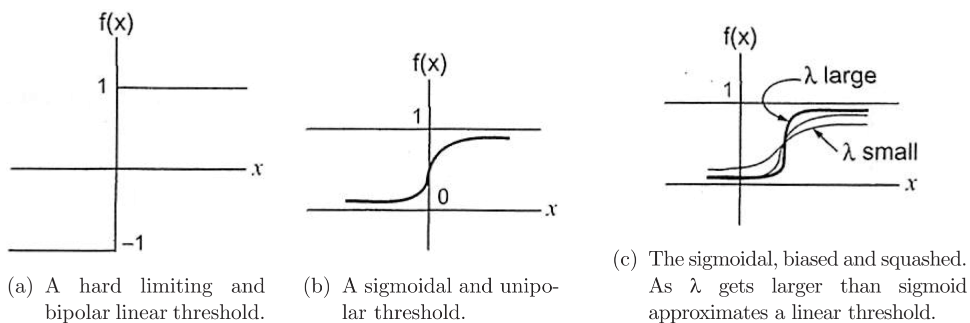
We next describe several of the skills necessary for dealing coherently with neural networks and deep learning. First, we demonstrate the backpropagation algorithm and show a solution for the XOR problem that limited the use of the first-generation perceptron. Second, we demonstrate how matrix algebra provides the medium for network computing. Finally, we note the importance of statistics and briefly describe the *softmax* transformation. Deep learning, we conjecture, can best be described as computational statistics.

2.1. *Backpropagation: Partial differential equations*

The approach taken by the backpropagation algorithm is to start at the output layer and *propagate* error backward through all the hidden layers. We know that all the information needed to update the weights on a neuron is local to that neuron, except for the amount of error. For output nodes, this error is easily computed as the difference between the desired and actual output values. For nodes in the hidden layers, it is considerably more difficult to determine the error for which a node is responsible. The activation function for backpropagation is the logistic (sigmoid) function:

$$f(\text{net}) = 1/(1 + e^{-\lambda * \text{net}}), \quad \text{where net} = \Sigma x_i w_i.$$

This function, seen in Figs. 7(b) and 7(c), is used for four reasons: First, it has the sigmoid shape giving a real-valued output. Second, as a continuous function, it has a derivative everywhere. Third, since the value of the derivative is greatest where the sigmoidal function is changing most rapidly, the assignment of the most error is attributed to those nodes whose activation was least certain. Finally,



(a) A hard limiting and bipolar linear threshold.
(b) A sigmoidal and unipolar threshold.
(c) The sigmoidal, biased and squashed. As λ gets larger than sigmoid approximates a linear threshold.

Fig. 7. Threshold functions. (a) was used in the Hebbian example. The sigmoid function, continuous and differentiable, is used with backpropagation. Figure adapted from [2].

WSPC/214-IJSC 2430008 ISSN:1793-351X 2nd Reading

the derivative of the logistic function is easily computed by a subtraction and multiplication:

$$f'(\text{net}) = (1/(1 + e^{-\lambda^*\text{net}})) = \lambda(f(\text{net}) * (1 - f(\text{net}))).$$

Backpropagation training uses the generalized delta rule. For nodes in the hidden layer, we look at their contribution to the error at the output layer. The formulas for computing the adjustment of the weight w_{ki} on the path from the kth to the ith node in backpropagation training are:

(1) $\Delta w_{ki} = -c(d_i - O_i) * O_i(1 - O_i)\ x_k$, for nodes on the output layer, and
(2) $\Delta w_{ki} = -c * O_i(1 - O_i)\ \Sigma_j\ (-\text{delta}_j * w_{ij})x_k$, for nodes on hidden layers.

In (2), j is the index of the nodes in the next layer to which i's signals fan out and

$$\text{delta}_j = -\partial\text{Error}/\partial\text{net}_j = (d - O_i) * O_i(1 - O_i).$$

We next show the derivation of these formulae. First, we derive (1), the formula for weight adjustment on nodes in the output layer. As before, what we want is the rate of change of network error as a function of change in the kth weight, w_k, of node i. We show that

$$\partial\text{Error}/\partial w_k = -((d_i - O_i) * f'(\text{net}_i) * x_k).$$

Since f, which could be any function, is now the logistic activation function, we have

$$f'(\text{net}) = f'(1/(1 + e^{-\lambda^*\text{net}})) = f(\text{net}) * (1 - f(\text{net})).$$

Recall that $f(\text{net}_i)$ is simply O_i. Substituting in the previous equation, we get

$$\partial\text{Error}/\partial w_k = -(d_i - O_i) * O_i * (1 - O_i) * x_k.$$

Since the minimization of the error requires that the weight changes be in the direction of the negative gradient component, we multiply by $-c$ to get the weight adjustment for the ith node of the output layer:

$$\Delta w_k = c(d_i - O_i) * O_i * (1 - O_i) * x_k.$$

We next derive the weight adjustment for hidden nodes. For the sake of clarity, we initially assume a single hidden layer. We take a single node i on the hidden layer and analyze its contribution to the total network error. We do this by initially considering node i's contribution to the error at a node j on the output layer. We then sum these contributions across all nodes on the output layer. Finally, we describe the contribution of the kth input weight on node i to the network error. Figure 8 illustrates this situation.

2430008.indd 11 24-09-2024 11:23:44

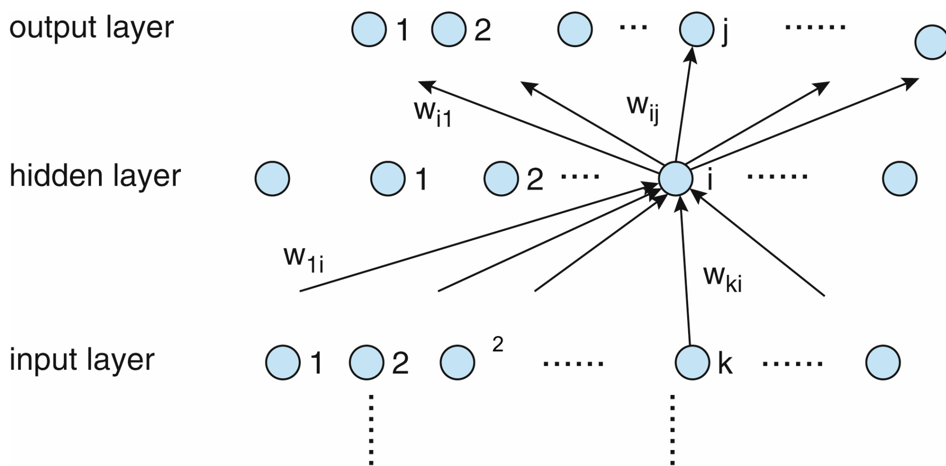


Fig. 8. $\sum_j -\text{delta}_j * w_{ij}$ is the total contribution of node i to the error at the jth output. Our derivation gives the adjustment for w_{ki}. Figure adapted from [2].

We first look at the partial derivative of the network error with respect to the output of node i on the hidden layer. We get this by applying the chain rule:

$$\partial\text{Error}/\partial O_i = \partial\text{Error}/\partial\text{net}_j * \partial\text{net}_j/\partial O_i.$$

The negative of the first term on the right-hand side, $(\delta\text{Error})/(\delta\text{net}_j)$, is called delta_j. Therefore, we can rewrite the equation as

$$\partial\text{Error}/\partial O_i = -\text{delta}_j * \partial\text{net}_j/\partial O_i.$$

Recall that the activation of node j, net_j, on the output layer is given by the sum of the product of its own weights and of the output values coming from the nodes on the hidden layer:

$$\text{net}_j = \sum_i w_{ij} O_i.$$

Since we are taking the partial derivative with respect to only one component of the sum, namely the connection between node i and node j, we get

$$\partial\text{net}_j/\partial O_i = w_{ij},$$

where w_{ij} is the weight on the connection from node i in the hidden layer to node j in the output layer. Substituting this result,

$$\partial\text{Error}/\partial O_i = -\text{delta}_j * w_{ij}.$$

Next we sum over all the connections of node i to the output layer:

$$\partial\text{Error}/\partial O_i = \sum_j -\text{delta}_j * w_{ij}.$$

This represents the sensitivity of network error to the output of node i on the hidden layer. We next determine the value of delta_i, the sensitivity of network error

to the net activation at hidden node i. This gives the sensitivity of network error to the incoming weights of node i. Using the chain rule again,

$$-\text{delta}_i = \partial\text{Error}/\partial\text{net}_i = (\partial\text{Error}/\partial O_i) * (\partial O_i/\partial\text{net}_i).$$

Since we are using the logistic activation function,

$$\partial O_i/\partial\text{net}_i = O_i * (1 - O_i).$$

We now substitute this value in the equation for delta_i to get

$$-\text{delta}_i = O_i * (1 - O_i) * \sum_j -\text{delta}_j * w_{ij}.$$

Finally, we can evaluate the sensitivity of the network error on the output layer to the incoming weights on hidden node i. We examine the kth weight on node i, w_k. By the chain rule,

$$\partial\text{Error}/\partial w_{ki} = (\partial\text{Error}/\partial\text{net}_i) * (\partial\text{net}_i/\partial w_{ki}) = -\text{delta}_i * (\partial\text{net}_i/\partial w_{ki}) = -\text{delta}_i * x_k,$$

where x_k is the kth input to node i.

We substitute into the equation the value of $-\text{delta}_i$:

$$\partial\text{Error}/\partial w_{ki} = O_i(1 - O_i) * \sum_j (-\text{delta}_j * w_{ij})x_k.$$

Since the minimization of error requires that the weight changes be in the direction of the negative gradient component, the weight adjustment for the kth weight of i is fixed by multiplying the negative learning constant:

$$\Delta w_{ki} = c * \partial\text{Error}/\partial w_{ki} = c * O_i(1 - O_i) * \sum_j (-\text{delta}_j * w_{ij})x_k.$$

For networks with more than one hidden layer, the same procedure is applied recursively to propagate the error from hidden layer n to hidden layer $n-1$.

Although it provides a solution to the problem of learning in multilayer networks, backpropagation is not without its own difficulties. Like the *hillclimbing* algorithm [2] it may converge to a local minimum, as was seen in Fig. 6. Finally, backpropagation can be very expensive to compute, especially when the network converges slowly.

Example: Backpropagation Solving the Exclusive-Or Problem

We next demonstrate the backpropagation algorithm solving the exclusive-or problem. The exclusive-or function in logic produces *true* when either of its two input values are *true*, and *false* when both input values are either *true* or *false*. It wasn't until the creation of the Boltzmann machine [12], the generalized delta rule, and the backpropagation algorithm that the exclusive-or problem was solved.

Figure 9 shows a network with two input nodes, one hidden node and one output node. The network also has two bias nodes, the first connected to the hidden node and the second to the output node. The net values for the hidden and output nodes are calculated in the usual manner, as the vector product of the input values times

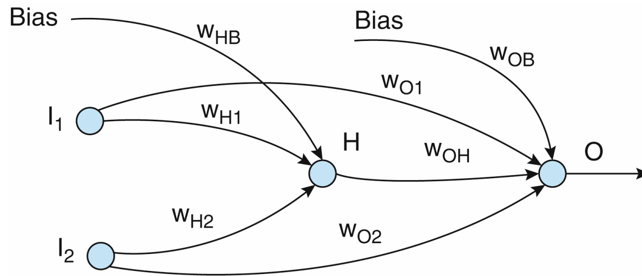


Fig. 9. One backpropagation neural network that solves the exclusive-or problem. The W_{ij} are the weights, and I the input nodes, H the hidden node, and O the output node. Figure adapted from [2].

their trained weights. The bias is added to this sum. The weights and the biases are trained using the backpropagation algorithm with the sigmoidal activation function. Note that the input nodes are also directly linked, with trained weights, to the output node. This additional linking can often let the network designer get a network with fewer nodes in the hidden layer and quicker convergence.

There is nothing unique about the network of Fig. 8. Any number of different networks could be used to compute a solution to the exclusive-or problem. This randomly initialized network was trained with multiple instances of the four patterns that represent the truth-values of the exclusive-or function. We use the symbol "→" to indicate that the value of the function is 0 or 1. These four values are

$$(0, 0) \to 0;\ (1, 0) \to 1;\ (0, 1) \to 1;\ (1, 1) \to 0.$$

A total of 1400 training cycles, using these four instances, produced the following values, rounded to the nearest tenth, for the weight parameters of Fig. 8:

$$W_{H1} = -7.0;\ W_{H2} = 2.6;\ W_{HB} = -7.0;\ W_{O1} = -5.0;\ W_{OH} = -11.0;\ W_{OB} = 7.0;\ W_{O2} = -4.0.$$

With input values (0, 0), the output of the hidden node is

$$f(0*(-7.0) + 0*(-7.0) + 1*(2.6)) = f(2.6) \to 1.$$

The output of the output node for (0, 0) is

$$f(0*(-5.0) + 0*(-4.0) + 1*(-11.0) + 1*(7.0)) = f(-4.0) \to 0.$$

With input values (1, 0), the output of the hidden node is

$$f(1*(-7.0) + 0*(-7.0) + 1*(2.6)) = f(-4.4) \to 0.$$

The output of the output node for (1, 0) is

$$f(1*(-5.0) + 0*(-4.0) + 0*(-11.0) + 1*(7.0)) = f(2.0) \to 1.$$

The input value of (0, 1) is similar. Finally, we check the network with input values of (1, 1). The output of the hidden node is

$$f(1*(-7.0) + 1*(-7.0) + 1*2.6) = f(-11.4) \to 0.$$

The output of the output node for (1, 1) is

$$f(1^*(-5.0)+1^*(-4.0)+0^*(-11.0)+1^*(7.0))=f(-2.0)\to 0.$$

The result demonstrates that the feedforward network of Fig. 8, using backpropagation learning, made a nonlinear separation of exclusive-or data points. The threshold function f is the sigmoidal of Fig. 7(c), the learned biases have translated it very slightly along the positive direction of the x-axis. We offer a matrix representation that captures the weights and input values for this problem in Sec. 2.2.

In concluding this example, it is important to understand what the backpropagation algorithm produced. The search space for our *exclusive-or* network has eight dimensions, represented by the seven weights of Fig. 9 plus the error of the output. Each of the seven weights was initialized with random values. When the initial output was produced and its error determined, backpropagation adjusted each of the seven weights to decrease this error. The seven weights are adjusted again with each iteration of the algorithm, moving toward values that minimize the error for computing the exclusive-or function. After 1400 iterations the search found values for each of the seven weights that lets the error approach zero. What has happened is that in an 8-dimensional space, backpropagation has found a 7-dimensional hyperplane that appropriately separates the four ex-or instances.

Finally, an observation is made. The exclusive-or network was trained to satisfy four exact patterns, the results of applying the exclusive-or function to true/false pairs. In modern deep learning situations training to solve exact situations is rarely the case. Take for example, a program that scans X-ray images to detect disease situations. Another example is a network that scans metal welds looking for bad metal binding. Such systems are called *classifiers* and they examine new, previously unseen situations to determine if there are potential problems.

2.2. *Matrix algebra*

We next consider how matrices are used to represent several of the neural network examples seen earlier. First consider the general description of an artificial neuron described in Fig. 1. Here we have input values $x_1, x_2, x_3, \ldots, x_n$. Each of these input values has a corresponding weight attached, $w_1, w_2, w_3, \ldots, w_n$. The calculation of the value of the output node is $\Sigma_i w_i x_i$ which is given by the product of the two matrices:

$$\begin{bmatrix} x_1 & x_2 & x_3 & \cdots & x_n \end{bmatrix} * \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} \Sigma_i w_i x_i \end{bmatrix}.$$

A threshold function, f, is then applied to the output of the node: $f\,[\Sigma_i w_i x_i]$.

Our second example considers the weight calculations for the Hebbian stimulus conditioning network of Sec. 1. First the network is run with the original input and weights:

$$[x_1\ x_2\ x_3\ \cdots\ x_6]*\begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_6 \end{bmatrix} = [\Sigma_i w_i x_i].$$

The weight adjustment, ΔW, for the second iteration of the net is created by multiplying the original input array by the learning rate constant, c, times the original network output, either a 1 or -1:

$$\Delta W = X * c * f(X,\ W).$$

The new set of weights is produced by scalar multiplication on the original input array:

$$[x_1\ x_2\ x_3\ \cdots\ x_6] * c * (1 \text{ or} -1) = [2w_1\ 2w_2\ 2w_3\ \cdots\ 2w_6].$$

Next, these new weights are multiplied by the transposed array of the next input values, $[2x_1,\ 2x_2,\ 2x_3, \ldots,\ 2x_6]$ to produce

$$[2w_1\ 2w_2\ 2w_3\ \cdots\ 2w_6]*\begin{bmatrix} 2x_1 \\ 2x_2 \\ 2x_3 \\ \vdots \\ 2x_6 \end{bmatrix} = [\Sigma_i 2w_i 2x_i].$$

The output of the network is the sign of $\Sigma_i 2w_i 2x_i$, either +1 or -1. The next weight adjustment, $[3w_1\ 3w_2\ 3w_3\ \cdots\ 3w_6]$, is like the most recent:

$$[2x_1\ 2x_2\ 2x_3\ \cdots\ 2x_6] * c * (1 \text{ or } -1) = [3w_1\ 3w_2\ 3w_3\ \cdots\ 3w_6].$$

The Hebbian unsupervised network continues, with numeric details as shown in Sec. 1.

2.3. *Statistical measures*

Traditional machine learning in AI has always been an exercise in computational statistics. For example, the ID3 algorithm and its descendants [15, Sec. 10.3] use information theory to measure how pieces of data from a large collection of data correlate. Principle component analysis is often used for dimensionality reduction of the very large matrices used in *latent semantic analysis*. A knowledge of the implicit parallelism possibilities within matrix algebra has led to computational efficiencies in deep learning.

WSPC/214-IJSC 2430008 ISSN:1793-351X 2nd Reading

We earlier demonstrated how multi-layered feedforward networks propagated output error back across the hidden layers of the network. This error propagation is called *gradient descent learning*. We also noted that the output nodes of these networks were activated through using the summed values of their weighted inputs. We next describe the *softmax* equation that transforms network output values into distributions. This transformation takes the raw numbers produced by the network's output layer and transforms them into a probability distribution. The term "softmax" comes from a "softening" of the traditional *max* function which selects the maximum of a set of given values.

As we will see in Sec. 3, deep learning algorithms represent their words/tokens as small real numbers. This is done for many reasons, including keeping partial derivatives within acceptable bounds. Further, since the output of these networks reflects the correlations found between the words/tokens in the learned model, it is important to characterize the output values of many matrix computations as probability distributions.

A probability distribution is a set of non-negative numbers that sum to 1.0. The *softmax* function s supports this transformation of network output values into a probability distribution:

$$\sigma(x_j) = e^{x_j}/\Sigma_i\, e^{x_i}.$$

In this formula, the Softmax of each output x_j is e to the power of that output divided by the sum of all outputs as a power of e. Even if x_j is a negative number, e to that power is positive. For an example, $s[1, 0, -1]$ is approximately [0.665, 0.244, 0.090]. It should be recognized that softmax transforms each output value to be part of a probability distribution while retaining the overall relationships between the original output's values. Figure 10 augments Fig. 5 by adding a Softmax layer. The result of the softmax processing is then used with backpropagation to reduce the errors in the weights of the nodes on previous layers of the network.

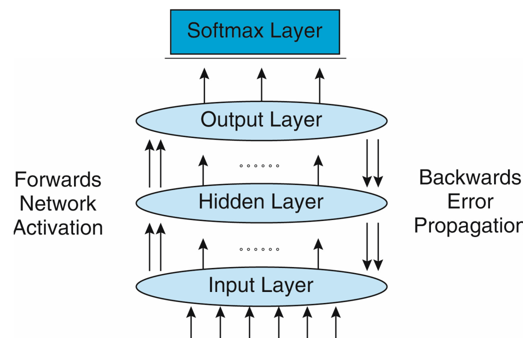


Fig. 10. A revision of Fig. 5 to add a Softmax layer. The results of Softmax are then used to adjust the weights of the hidden layers. Figure adapted from [2].

Figure 10 shows softmax used on the network output layer, however it can be used on the output values of sets of nodes anywhere in the network. One example of this will be shown with the attention component of the transformer model in Sec. 4.

There are several further issues that arise when using softmax. One is the problem of trying to process large vectors with big numbers. But even with small vectors softmax can produce exaggerated results. Consider, for example, the Softmax of the output vector [1 5 1 1]. softmax produces the vector [0.0174 0.9479 0.0174 0.174], where the second element of the vector gets a very high value, only 0.0521 less than 1, while the smaller probabilities are close to 0.

This issue is called *saturation* and only gets worse when some number is much larger than the others: a result where the larger number gets even closer to 1 and the smaller values go to 0. This issue is critical because the resulting softmax vector is intended to be used for gradient descent error reduction, which can lead to slow convergence and high variance in the training process. There are remedies for saturation problems, such as normalizing the output vector by dividing each element by the square root of the vector's length, as we will see in later sections.

Finally, creating networks based on reasoning using probability distributions affects the entire design of the network. The word/token embeddings input to the network are represented as small real numbers, or *floats*, between, and including, one and zero. Weight vectors are randomly initialized as real numbers near zero. We see this in more detail with the large language models and transformers of Sec. 3.

Each output of the output layer of the neural network, before its softmax transformation, is called a *logit*, pronounced "LOW-jit". Using a probability distribution to determine reward/punishment measures for weights in the network is called calculating the *cross-entropy loss*. Entropy is a term from information theory that measures the degree of disorder or randomness in a system. Loss of entropy indicates the reduction of disorder or error. When the Softmax equation is used this error reduction is measured probabilistically across the network's output states. *Cross-entropy* is a term used for describing the difference between two distributions, i.e. the distributions of the current and of the desired outputs.

Decreasing cross entropy loss, a positive number, is measured using the negative logarithmic probability assigned to the Softmax output. softmax computes the probabilities of all the outputs and provides an ordering of the "best" results for minimizing loss. We represent the cross-entropy loss, CEL, x, as the negative logarithmic probability of that result:

$$\mathrm{CEL}(f,x) = -\ln p_f(a_x).$$

In this equation, (f,x) represents the Softmax equation, ϕ, applied to x which produces a_x.

We next consider why the negative logarithmic probability is used. First, our error estimate from softmax is a positive number so we want to decrease that error. Second, the logarithmic function has an important property between 1, our

WSPC/214-IJSC 2430008 ISSN:1793-351X 2nd Reading

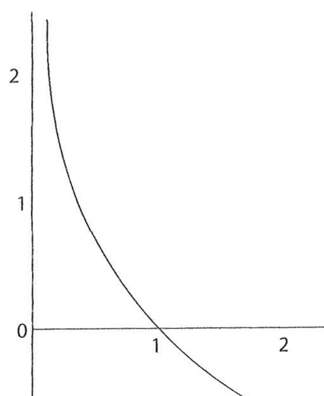


Fig. 11. The graph of –ln(x) as x approaches 0.

maximum probability, and 0: –ln(x) gets larger as x approaches 0 which is important because this range is where most of our focus will be. This logarithmic function of x is shown in Fig. 11.

3. Building Large Language Models

Consider the sentence "What does a language model model?" How many times in normal English communication does the word model follow itself? Not really that often. On the other hand, how many times will the word "hand" follow the three words "On the other"? Quite regularly, as drawing comparisons using binary metaphors is rather common in normal English communication. Both these situations are captured in language models.

A language model is designed to represent how a particular language is used. Current language models developed for deep networks are stochastic, where each query to the model produces a probability distribution of language patterns that satisfy that query. The data to support these models can be from written or oral language usage. Large language models are a probabilistic representation of the corpora of language usage on which they are trained. We focus next on building models for written English word/sentence communication.

As we will see, discovering what the next word should be in an expression is important for creating new expressions. In fact, language models also support finding the most likely previous word for a set of words, and the most likely words to fit between language strings. This missing word generation process supports the design and use of many interesting language-based applications. These can include:

(A) Translation between languages.
(B) The classification of texts into different categories.
(C) The recognition of "sentiment" such as joy or anger in expressions.
(D) Information retrieval from databases.

(E) Answering questions, as in various standardized tests.
(F) The generation of news articles.
(G) Composing essays on various topics.

There are several components of the creation of large written English language models. We discuss these next.

3.1. *Preparing the text*

Representing an all-English text, even for a very large neural network, is a complex task. For one thing, there are more than an infinite number of numbers each of which can be part of a piece of text. No finite state machine can handle an uncountably infinite number of tokens! Thus, the first task in representing written language for a neural network is to break the text up into reasonable finite number of pieces, that we will refer to as "tokens". We then describe how these tokens will be the input for the network. We can start by considering the example of the infinite number of numbers and replace each of these numbers by the token UNK. This token, indicating an "unknown" word in English, will also be used to replace any component of the text that is not in the final set of vocabulary tokens that will be the input to the network.

Because even very large neural networks have a certain fixed size, we next consider what an "acceptable" number of vocabulary tokens might be. Current language models have a word vocabulary of 30,000, with BERT [16], to 50,000 for GPT [17]. Although this sounds like a rather large number of possible tokens for the network, the Oxford English Dictionary estimates there are more than 170,000 English words currently in use. The OED also contains about 50,000 obsolete words that are used at different times. GPT is, at the present time, trained on a corpus of English sentences that contain more than a half trillion words from 45 terabytes of text data.

The task, therefore, once the training corpora is identified, is to reduce the total number of individual words in the training corpus to the specific set of tokens that the network can process. There are several methods for accomplishing this task:

(A) Individual words must be isolated and "recognized". This isolation can be assisted by recognizing the blank spaces that separates words. Punctuation, such as a period or semicolon, can also separate words. This word "recognition" enables *word embeddings*, described below.
(B) Punctuation symbols are "recognized" using rules that describe their roles in English text. The punctuation symbols themselves are placed in the "UNK" category.
(C) Sentences are usually padded, in that a specific symbol, say "STOP" is added after every sentence. This STOP captures the difference between periods that end sentences and those that are parts of sentences, e.g., e.g.
(D) New paragraphs, new pages, misspelled and hyphenated words must also be recognized and tokenized, possibly as UNK.
(E) And so on...

As can be understood with very little thought, this process of making rules to break up text into individual word/tokens is complex as well as tedious to apply. But computers are excellent at complex and tedious tasks. Further details on the English text tokenization process can be found in computational linguistics texts or on the internet.

The next task is to determine the number of unique words or tokens, i.e. the vocabulary V, for the language model. We also calculate the number of times each of these words/tokens is used in the training corpus. Once words are cataloged and their occurrences calculated, further reduction is a function of the vocabulary's limitations in the network. Reduction is done in several ways, including removing non-word symbols from the text. Besides numbers, periods, commas, and other punctuation are removed. Chemical, biological, or mathematical expressions are also often removed. Rare and infrequently used words can also be removed.

About 2017 and with the emergence of GPT-1, a further refinement was made on language tokenization. All words, including the previous UNK, were decomposed further to create partial-word tokens. This process breaks all words into their constituent syllables or, if it is voiced speech, into phonemes. This process makes sense since there are far fewer syllables or phonemes than there are individual words in a language. GPT now uses about 50,000 of these language tokens. Once the size of the set of tokens appropriate for the constraints of the network, the V, is determined, the mapping from tokens to network representations is possible. This is called creating the *word* or *token embeddings*.

3.2. *Creating word/token embeddings*

Because a language model will be represented as a probability distribution over word/token use in a language, word/token encodings, randomly generated, are usually taken from the set of real numbers between –1 and 1. Each token will be represented as a vector of real numbers, or *floats*, called the *word embeddings*. The number of floats, f, used for word embeddings is a hyperparameter of the network. A typical value for f is 100, although much larger vectors are often used for token embeddings.

The size of the array of token embeddings is $|V| \times f$, where $|V|$ is the size of the vocabulary and f the number of floats used in the model. The integers 1 to $|V|$ serve as the index for the array of token embeddings. Thus, if the token "butter" has index 6, then the 6th row of the embedding array is the vector representing "butter". This array of word embeddings is a parameter of the learning network. When the network is trained on English word/token patterns and, for example, is looking for the most appropriate next word/token in a string of word/tokens, back propagation conditions all word embeddings.

To be precise, the input to the language model is the index of the current word/token in question which is immediately translated into its embedding. Continuing from that point, all the operations of the network are on the embeddings. The output of the network is the probability that each of the tokens in the array of

embeddings is the next most likely token in an expression. The backpropagation conditioning is cross-entropy loss, described by the function $-\ln P(x)$ of Fig. 7. This is the negative natural log of the probability of x, where x is an instance of an actual correct next word. In the backpropagation stage, once the loss function is determined the token embeddings of that loss are modified.

3.3. *Bi-grams and tri-grams*

N-gram technology has long been a staple of natural language understanding and the development of language models. The general question asks what the probability is of an expression being part of a language. In probabilistic terms, described in more detail in [2] PART VII, the probability, P, of the expression "It is hot today" can be represented using the *chain rule*:

$$P(\text{It is hot today}) = P(\text{It}) * P(\text{is}\,|\,\text{It}) * P(\text{hot}\,|\,\text{It is}) * P(\text{today}\,|\,\text{It is hot}).$$

The expression $P(X|Y)$ is interpreted as "the probability that X is true given that Y is true". "It is hot today" is obviously a very short sentence. It is not difficult to imagine the calculations and network training necessary to determine the probabilities of longer expressions, e.g. 15 words. The point, besides the calculation of the probabilities themselves, is that the lack of a corpus sufficient to condition all the probabilities that make up the members of this product of probabilities is a serious limitation on learning.

The *Markov assumption* simplifies this lengthy calculation, and constrains the probabilities that make up the chain, by assuming that the probability of a word is only conditioned by the probability of the word that immediately precedes it. Our previous example, simplified by using the Markov assumption, is

$$P(\text{It is hot today}) = P(\text{It}) * P(\text{is}\,|\,\text{It}) * P(\text{hot}\,|\,\text{is}) * P(\text{today}\,|\,\text{hot}).$$

Using the Markov assumption to determine the probability of word combinations is called the *bi-gram*, or two-word model. These bi-gram probabilities can be developed in either direction: either taking a word and determining the most likely next word or taking a word and determining the probability of the previous word. One can understand how determining bi-gram probabilities of word pairs is essential for generating new sequences of coherent word patterns.

Tri-gram probabilities are even more powerful in capturing patterns in language use. Tri-gram, or three word models, describe the probability of a word as a function of the probability of the two words that precede it or the two words that follow it. It takes a larger language corpus to train tri-gram models than that required for training bi-gram models.

3.4. *Training the language model*

Many of the details that make up the current generation of large language models remain company confidential. It takes internet searches and personal

WSPC/214-IJSC 2430008 ISSN:1793-351X 2nd Reading

Table 3. Results summarized from [18, Chap. 4], using a bi-gram-based language model.

Word	Cosine similarity
the	–0.160
a	0.127
recalls	0.479
says	0.553
computer	0.249
machine	0.333

communications to determine the best estimates of how these models are composed and trained. There is, however, information describing the time and costs of LLM training: For example, it took Google four days for 64 specially configured tensor processor to train BERT [16]. For the more general purpose GPT-3, it is estimated that training took 1,024 A100 computers with graphic processing units about 34 days. It is also estimated that it would cost about $4.6M to train GPT-3 using the lowest cost GPU cloud provider [17]. The details on Gemini, Google's DeepMind LLM, are not available.

Fortunately for end users, once these models are trained, they can then be reused with minimal retraining by outside groups for their own special purpose needs. The idea is that the final layer of the network can be replaced, and the learned patterns of the hidden layers can be reconditioned to address new tasks. This new training can take about 2 h on a normal laptop using graphics processors. As a result, much less time is needed for the general user to employ the BERT or GPT environments for specific uses. We describe this re-purposing, or *fine-tuning*, of LLMs in more detail in Sec. 5.

It is interesting to consider some of the computations produced by a language model. Charniak [18] in Chapter 4 describes the results of running such a model. His language model was trained on a corpus of about 1 million words and had a vocabulary size of 7500 words. The length of the word embeddings was 30. Selected results from Charniak's model are presented in Table 3. Cosine similarity measures word embedding vectors of length 30. Note that words that are used in similar situations have close cosine values. It is interesting that words that are appropriate for fitting into a sequence of words can also have similar meanings. For example, *recalls* or *says* could each follow he or she of a person's name in a sentence.

We next consider the design of the *attention-based* large language models called *transformers*.

4. Toward Transformer-Based Large Language Models

The precursors of *transformer-based language models*, before the release of GPT-1 in 2018, were *convolutional* and *recurrent* neural networks and *autoencoders*. Convolutional networks are feedforward networks that learn image features through filter optimization. Recurrent networks support feedback within layers so that

WSPC/214-IJSC 2430008 ISSN:1793-351X 2nd Reading

new input for some nodes is modulated by previous node output. This "memory device" facilitates their use in automated handwriting and speech recognition tasks. Autoencoders are unsupervised networks for learning efficient representations of unlabeled data. The decoder element then translates this internal representation back into some useful form. For example, the autoencoder can be generative, translating a string of words into a different language.

The architectures underlying convolutional and recurrent networks are scaled-up implementations of ideas several decades older. Models resembling classical convolutional neural networks gained *state-of-the-art* status in computer vision and models resembling the original design for the LSTM recurrent neural network came to dominate applications in natural language processing [19, 20]. Through the 2010s, the rapid emergence of deep learning successes based on CNN and RNN technology is also attributable to the availability of computational resources, including innovations in parallel computing and massive data farms, i.e. cheap storage and multiple internet service providers.

There were, however, fundamental limitations with RNNs including the use of LSTM recurrent networks. First, recurrent networks have difficulty addressing long range dependencies in sentences. Second, RNNs had issues with very large and very small learning gradients. Finally, the sequential nature of the recurrent network "roll out" limited parallel evaluation and required substantially increased training times. Attention-based models address each of these issues.

One further contribution of the early 2010s, and an important step in the evolution of transformer-based language models, was the creation of the *generative adversarial network*, or *GAN*. The GAN was originally developed by Goodfellow and his colleagues [21]. The GAN pits two neural networks "against" each other, the adversarial component, in a zero-sum competition. Given a training set, the GAN technique can generate new data with the same statistical profile as its training data. For example, a GAN trained on image data generates new images that can look authentic to human viewers (url 1). Although originally proposed as a method for unsupervised learning, the GAN is also useful for semi-supervised [22] and [23] learning. It can also support reinforcement learning [24].

When first proposed, the *attention* mechanism produced improvement to the recurrent networks previously used for machine translation, performing better than the earlier encoder–decoder sequence-to-sequence approaches [25]. Using attention, the decoder receives as input a context vector that consists of a weighted representation of the input at each time step. Researchers noted that some important qualitative insights often emerged when using attention weights. In translation tasks, for example, attention could suggest choices between synonyms for generating words in the target language. In Fig. 12 attention enhances the autoencoder–decoder model.

Starting with OpenAI's GPT-1, available in 2018, the *transformer* has become the predominant network architecture for generative AI. Nearly all language processing and vision analysis tasks are currently based on the transformer architecture.

2430008.indd 24 24-09-2024 11:24:08

WSPC/214-IJSC 2430008 ISSN:1793-351X 2nd Reading

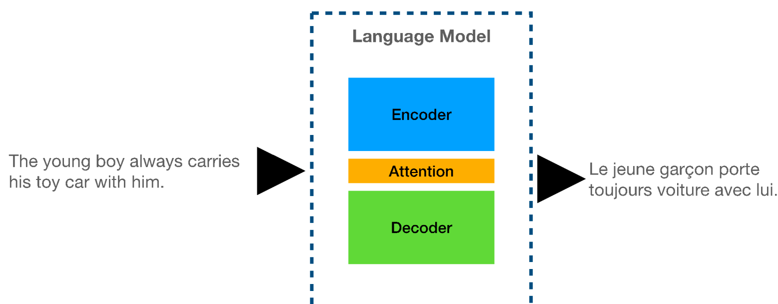


Fig. 12. Bahdanau attention [25] mediates between the input to the encoder and the decoder output in an example of English to French translation.

Besides GPT, examples include BERT [16], RoBERTa [27], Longformer [28], and Gemini [29, url 18.16].

OpenAI's large language models can develop "conversations" using the GPT transformer [17]. Further, their vision transformer is emerging as the model for multiple vision tasks, including the recognition of images and detection of objects, as well as creating semantic supported [27, 30]. Transformers are also used for speech recognition [31] and reinforcement learning [18]. Transformer models use *autoencoders* containing *self-attention* mechanisms. We consider these next.

4.1. *Transformer models*

The paper entitled *Attention is All You Need* [1] was a revolutionary contribution to the design of generative models. The idea that supports the *transformer* design is the *self-attention* mechanism. Figure 12, adapted from [1], reflects the transformer model with an attention mechanism, called *self-attention*, in each of the encoder–decoder components.

Attention mechanisms enhance transformer models because they support selective focusing on multiple input elements. This improves both decoding accuracy and computational efficiency. Attention mechanisms prioritize and emphasize relevant information, acting as a "focus" that enhances overall model performance. This overcomes one of the problems of the previous recurrent models where long-range dependencies within a sentence or across several sentences were often lost.

As can be seen in Table 4, there are currently multiple transformer-based large language models available for exploration. Many details of these transformers are not available for public perusal. We will confine our comments to the publications available. In particular, we focus on the original transformer architecture proposed by Vaswani *et al.* [1], seen in Fig. 13, and on transformer descriptions offered by Zhang and his colleagues [32].

The encoder maps an input sequence of token embeddings, $(x_1, x_2,..., x_n)$ to a sequence of continuous representations $(z_1, z_2,..., z_n)$. Consuming Z, the decoder than

Table 4. Currently, July 2024, available software for generative AI. Note that information on several models is company confidential. Interested readers should search these software tools for more current information as AI companies are known to change names, merge, or simply dissolve. Table adapted from [2].

Model	Capabilities	Parameters	Training data	URL
BERT	Question answering, finds semantic similarity	345 million	3.3 billion words	url 2
PaLM 2	Generates text, essays, and reports, answers questions, uses desired style and tone. Tuned to follow instructions	340 billion	3.6 trillion tokens	url 3
ChatGPT, powered by GPT	Generates text, essays, and reports, answers questions, uses desired style and tone. Tuned to follow instructions.	175 billion	300 billion tokens	url 4
Gemini created by Google DeepMind	Generates multimodal output from multimodal input. Generates documents with both text and images.	Information not public	Information not public	url 16 url 17
Llama 2	Generates text, essays, and reports, answers questions using desired style and tone.	70 billion	2 trillion tokens	url 5
T5	Advanced multilingual text generation	220 million	29 trillion characters in 107 languages	url 6
Stable diffusion	Generate images from text, image translation.	860 million	5 billion image text annotated pairs	url 7
Imagen	Generate images from text, image translation, advanced image editing.	Information not public	Information not released publicly	url 8
DALL E	Generate images from text, advanced image modification	12 billion	650 million text image pairs	url 9
Phenaki	Generate videos from text descriptions	Information not public	Information not released publicly	url 10
Music Gen	Generate music from text descriptions	Information not public	Information not released publicly	url 11

generates, one at a time, an output sequence $(y_1, y_2,..., y_m)$. At each step, when generating an output, the decoder can utilize all the previously generated symbols as additional input. This property is called being *auto-regressive*.

The encoder for the original transformer was a stack of six identical layers. As seen in Fig. 12, there are two components in each layer, the first is a multi-head self-attention mechanism and the second a fully connected feed-forward perceptron network. There are *add* and *norm* connections for each component to ensure inter-component and inter-layer communication.

The original decoder was also a stack of six identical layers. Besides the two subcomponents identical to those of the encoder, there is a second multi-head attention mechanism, masked, and *positional encoding* ensuring that the output embeddings are offset by one position. This insures that the output values at any time also

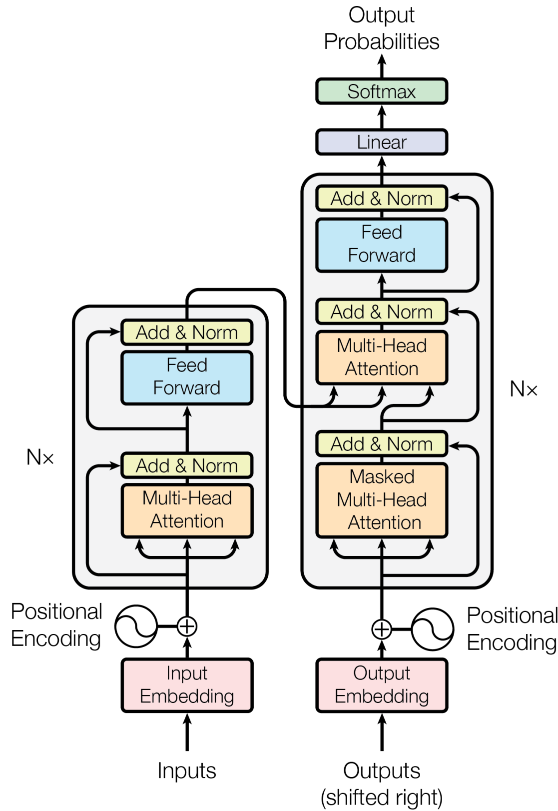


Fig. 13. A transformer model, reflecting the encoder and decoder using self-attention. This is one layer of the original six-layer stack architecture. Figure adapted from [1].

depend on previous output values. The decoder's components also have *add* and *norm* mechanisms supporting inter-component and inter-layer communication.

4.2. *Attention*

Figure 12 presented an example where the encoder and decoder were focused on a string of tokens to translate that string into French. We begin this section with two examples of the self-attention mechanism that supports transformer processing. In the transformer, self-attention is a network function that assists the encoder and decoder in determining how tokens within a string of tokens are related to each other; examples are shown in Fig. 14. In both figures the bottom string is identical to the top string and attention should be seen as a search for relationships within the string itself. In Fig. 14, the focus is on the it token, searching for an appropriate reference; on the left attention focuses on animal and on the right on street.

As a second example, consider the sentence of Fig. 15, The young boy always carries his toy car with him. If the word him is masked, and the model doesn't remember boy from the beginning of the sentence, it will not know which pronoun

Fig. 14. Attention: finding the relevant tokens supporting use of the pronoun "it" in two different but syntactically related sentences. Figure adapted from [2].

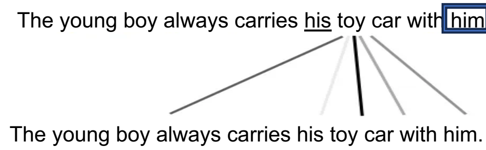


Fig. 15. Self-attention for his. Line shading indicates the amount of attention the word his pays to other words in the sequence. Using *masked,* or *auto-regressive* attention, only the words before the word under consideration are visited, in this example, the three leftmost lines. Figure adapted from [2].

to use at the end of the sentence: him, her, or it. Extending this example to a translation task, the transformer needs to use an appropriately translated pronoun when referring to boy.

To define the attention mechanism, we consider three components. First, a query, q, that can probe a tensor, T, of m tuples, for specific information. This probe queries tensor locations, or keys, to access the information at that specific location. The tensor location we refer to as the key, k, and the content of that location as the value v, the focus of the query. The query, keys, values, and output are all tensors.

Attention and self-attention are defined:

$$\text{Attention}(q, T) = \Sigma_i a(q, k_i) v_i,$$

where q is the token query, i ranges over the n key-value tuples of tensor T and the a are the positive real number attention weights.

With self-attention, the tensor T contains the query, q, the key k, and the value v, i.e. attention is focused on the vector containing the query.

To summarize results of using the attention mechanism:

(1) The attention query, q, process can operate on tensors, T, of any size.
(2) A single query, q, will receive different responses, v, depending on the keys of tensor, T.
(3) The query, q, operating on T can be flexible, i.e. asking for exact or approximate matches.

(4) The a weights on queries are positive real numbers. The attention process forms a positive cone, C. i.e. if v_i and v_j are outputs of the attention procedure and c and d are positive scalars, then all $c\,v_i + d\,v_j$ also belong to cone C.

To assure that all attention weights are non-negative and sum to 1, we apply a form of the *softmax* function:

$$a(q, k_j) = \frac{e^{a(q,k_j)}}{\Sigma_i e^{a(q,k_i)}}.$$

Attention supports the aggregation, or pooling, of information, given query q, over multiple key-value pairs. The attention procedure itself provides a continuous and differentiable function for the feed-forward neural network to determine which elements best suit the construction of further weighted representations. The attention weights attached to each query–key pair are trained by the values accessed by the key in the key–query pair. Figure 16 offers an example of attention-pooling network processing.

We next describe several metrics that support the query–key analysis, or how the query-based key-value relationship, $a(q,\ k_i)$ might be trained for each v_i. One commonly used metric is the Gaussian. As an example of using the Gaussian metric, the Nadaraya–Watson estimator [33, 34] is employed in regression analysis where the query represents the location for making the regression, the keys are the locations of observed previous data, and the values correspond to the regression values. For the Gaussian regression example, the $a(q,\ k)$ measure is calculated:

$$a(q, k_i) = e^{(-\frac{1}{2}||q-k_i||^2)},$$

where $||q - k_i||$ is the normed vector difference between q and each key, k_i.

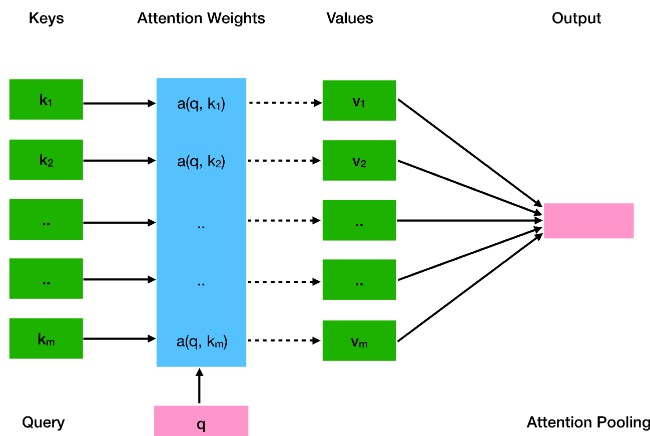


Fig. 16. The attention procedure, with pooling, computes a continuous and differentiable function of weight values over the v_i. Weights are trained for $(q,\ k_i)$ by the values v_j. Figure adapted from [2].

There are alternative choices for determining $a(q, k_i)$. The most common is the scaled dot-product used in the Vaswani *et al.* [1] original transformer. The dot-product is the scalar value created by the component-wise multiplication of two vectors and then the summation of these products. To ensure that the magnitude of the dot-product does not become over large, it is usually normalized by the square root of the dimension of the key vector, k_i. The dot-product attention measure is

$$a(q, k_i) = qk_i^T/\sqrt{d},$$

where k_i^T indicates the transpose of the key vector, k_i, and d is the length of the vector k_i.

We can simplify the results of this equation by using the softmax equation:

$$a(q, k_i) = \text{softmax}\, a(q, k_i) = e^{qk_i^T/\sqrt{d}}/\Sigma_j e^{a(qk_i^T)/\sqrt{d}},$$

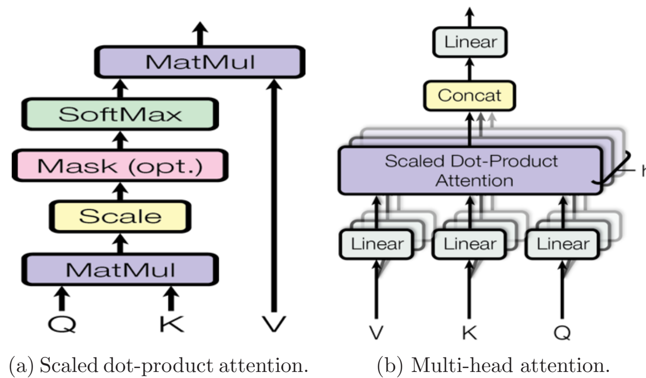
where k_i^T and k_j^T are the transposes of k_i and k_j and d is the length of vector k_i.

Most current transformers, including the original Vaswani *et al.* [1] transformer, use the scaled dot-product with softmax for calculating attention. Figure 17(a) represents the scaled dot-product attention mechanism. As just noted, scaled dot-product attention calculates using the matrix representations Q and K for all the individual q and k. From a matrix perspective, the attention mechanism is computed on a set of queries combined into a matrix Q. The keys and values are also combined into matrices K and V. The matrix output then is

$$\text{Attention}(Q, K, V) = \text{softmax}\frac{(QK^T)}{\sqrt{d_k}}V,$$

where T is the transpose of K and d_k is the dimension of the key vectors.

An alternative to the scaled dot-product attention mechanism is to use the dot-product mechanism without the scaling factor. Scaling is used to prevent the dot-product from getting too large and helps stabilize the learning process. Dividing



(a) Scaled dot-product attention. (b) Multi-head attention.

Fig. 17. (a) The standard scaled dot-product attention mechanism. (b) shows the multi-head scaled dot-product attention mechanism. Figure adapted from [1].

WSPC/214-IJSC 2430008 ISSN:1793-351X 2nd Reading

by the square root of the dimension of the key vector does not affect the overall distribution of the attention scores.

Research continues determining quality attention mechanisms. In fact, one research group [35] proposes improving the transformer architecture with the weights of a multiple level perceptron. A more common approach, however, is that of the original Vaswani *et al.* [1] attention mechanism that uses multi-head attention processing, as seen in Fig. 17(b).

Instead of using the scaled dot-product attention function with model-based dimensional keys, values, and queries, they found it helpful to linearly project the queries, keys, and values h times with different learned linear projections. The dimension of the key vector was used for the query and key and the value used its own dimension. In their original transformer architecture projection, h was 8. On each of the projected versions of queries, keys, and values the attention function is performed in parallel.

The attention output values are then concatenated together and again projected, producing the final values, as seen in Fig. 16(b). The multi-head attention mechanism supports the notion that the model can attend to different representation subspaces at different positions all in parallel, a function not possible with a single attention head of Fig. 17(a). Vaswani *et al.* [1] present several of these parallel multi-head attention results graphically.

Returning to the description of Fig. 13, the encoder contains self-attention layers, where the queries, keys and values come from the output of the previous layer. Each position of the encoder can attend to any position in the previous layer. Similarly, the self-attention layers in the decoder attend to all positions up to and including the current position which is masked to enforce the auto-regressive property.

In the encoder/decoder attention layers, the queries come from the previous decoder layer and the keys and values come from the output of the encoder. This enables each position in the decoder to attend to all positions in the encoder sequence. This gives the attention mechanism a functionality like the earlier pre-transformer sequence-to-sequence models seen in Fig. 12.

For the attention mechanism to utilize the order in the sequences there must be information about the relative order and position of tokens in each sequence. To accomplish this, positional encodings are added to the input embeddings at the bottoms of the encoder and decoder stacks, as seen in Fig. 12. Finally, each layer of the encoder and decoder contains a fully connected feed-forward network which is applied to the output of each of the attention mechanisms. The training process for the translation tasks of the Vaswani *et al.* [1] transformer is described in their paper.

We conclude this subsection with Fig. 18, a representation of the calculation of the attention mechanism.

For a high-level description of attention, compare it with normal database retrieval. In database retrieval, a query, q, is used to search for a key, k, that is

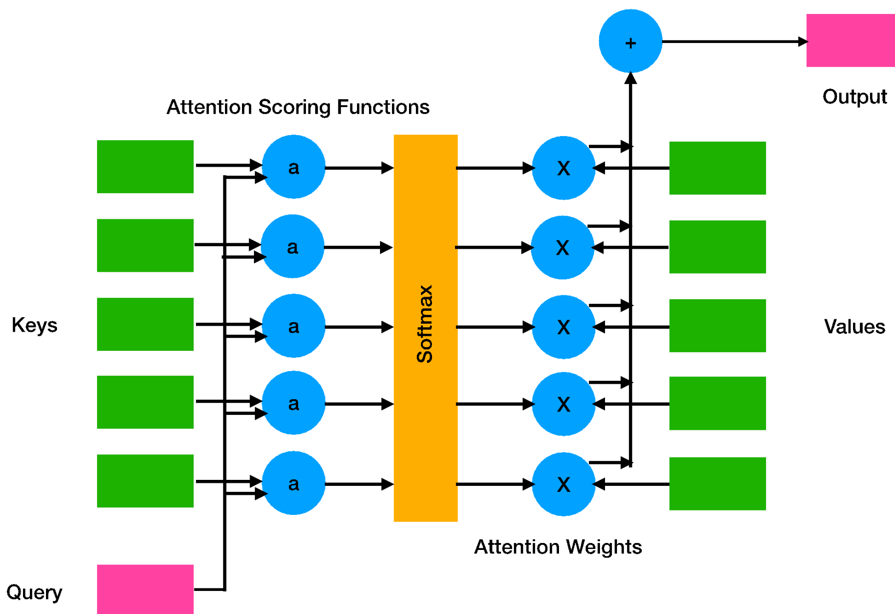


Fig. 18. The output of attention pooling is a weighted average of values where weights, a, are computed using the attention procedure with softmax. Figure adapted from [2].

a pointer to a value, v, that we wish to retrieve. With the attention mechanism, instead of returning a particular key–value pair, we return a probability distribution, based on our query, of the appropriateness of all possible key–value pairs. A more formal description for many of the transformer mechanisms described this section can be found in [36].

Since 2018 the transformer architecture with attention has become the predominant methodology used in building large language models. The practice in using transformers is to pretrain these large-scale models on enormous corpora to optimize self-supervised learning. After pretraining, the models can then be fine-tuned by their users with data appropriate for the user's application needs, as we see in Sec. 5 and Fig. 19. When using this pretraining approach, the original attention-based transformers are referred to as *foundation models* [26]. We consider these issues in Sec. 5.

5. The Transformer in Practice

Our final section first considers how the transformer-based model is trained to operate in specialized environments. Then we demonstrate *fine-tuning* the LLM to train it for new domains and then we present *prompt engineering*, a real-time technique that conditions the LLM to further focus its performance. Finally, we describe several important application areas for generative AI.

WSPC/214-IJSC 2430008 ISSN:1793-351X 2nd Reading

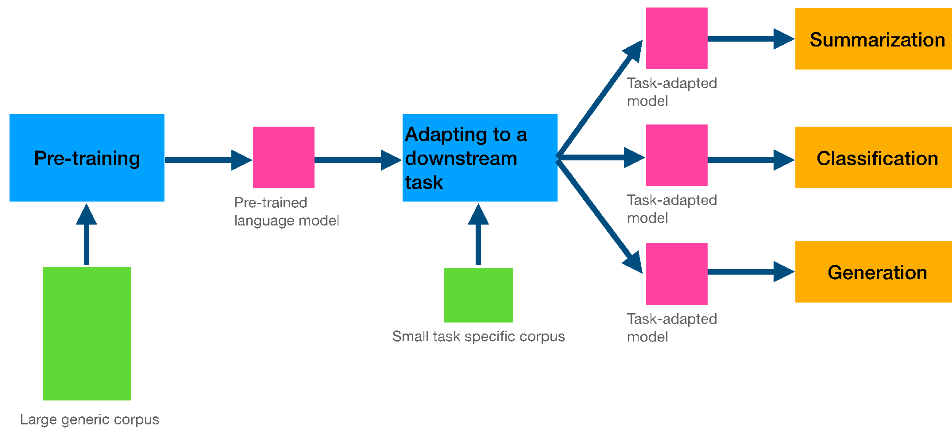


Fig. 19. The pretrained transformer can transfer its knowledge to related domains. The new domain is conditioned by using a smaller task-specific dataset. Figure adapted from [2].

5.1. *Pretraining: The corpora*

A large language model is first pre-trained on corpora of text data and on specific language modeling related tasks. During pre-training, the LLM tries to learn and understand general language patterns found in the relationships between words. Some examples of a suitable corpus for LLM pre-training:

- English Wikipedia — a collection of articles from the English version of Wikipedia, the free online encyclopedia. It contains a range of topics and writing styles, making it a representative sample of English language usage. The English component of Wikipedia contains about 2.5 billion words.
- The BookCorpus — a large collection of fiction and non-fiction books. It was created by scraping book text from the web and includes a range of genres, from romance and mystery to science fiction and history. The books in this corpus were required to have a minimum length of 2000 words and to be written in English. BookCorpus contains approximately one billion words.

5.2. *Fine-tuning the LLM*

Transfer learning is a technique that leverages the knowledge gained from one task to improve performance on another task. Transfer learning for LLMs involves taking a pretrained LLM and fine-tuning it for a specific "new" task, such as text classification or text generation. Fine-tuning updates the model's parameters using the new task-specific data, as in Fig. 19. Fine-tuning consists of four steps:

(1) Determine the model to be tuned and its parameters, e.g. the learning rate.
(2) Aggregate new training data, where format and other parameters depend on the model.

(3) Compute losses, the error measure, and gradients, to change the model to minimize error.
(4) Update the model through backpropagation.

5.3. *Prompt engineering*

Prompt engineering is the practice of querying the trained LLM with specific pieces of information to elicit the most appropriate responses from the model. There are several approaches to prompt engineering. *Zero-shot* queries request information that is not part of the model's training; the model will, however, generate a result. This technique makes LLMs useful for many different tasks. *Few-shot* prompting is a strategy where the model is given several task-specific examples before presenting the actual query. Few-shot queries enable the model to generalize over the queries. To summarize:

Zero-Shot Prompting: Used when the task is self-explanatory, requiring no specific examples.

One-Shot Prompting: Ideal for tasks requiring a specific format or context, where one example can guide the output. For example, give the LLM a job description and then ask it to "write a similar job description for a Data Analyst position".

Few-Shot Prompting: Used for complex tasks requiring multiple examples that provide broader context or to handle more nuanced queries. For example, after giving several labeled product reviews, to "Predict the sentiment of the following review".

Chain of Thought Prompting: It breaks down large problems into intermediate steps, allowing language models to tackle complex tasks not solved with standard prompting techniques.

Chain-of-thought prompting is a style of few-shot prompting, where prompts contain a series of intermediate reasoning steps. Chain-of-thought prompting encourages the model to reason the way that the prompts are proposed, i.e. in a series of steps. Surprisingly, the answers from chain-of-thought prompting are often more accurate and interpretable than the answers from other prompts. Chain-of-thought prompting also discourages the model from generating quick easy answers.

There are now several suggestions for organizing the process just described for moving from the foundation model through fine tuning to prompt engineering. This process is called LLM alignment, see [37] for a survey of approaches. One of these is called RLHF or Reinforcement Learning from Human Feedback [37]. There are still major questions about the utility of these approaches [38].

We next demonstrate our own use of prompt engineering to change LLM's responses. For our examples we use BERT (url 2) and GPT (url 4). The user gives ***Prompt:*** and the LLM replies with ***Output:***. We begin with an example of a zero-shot prompt:

WSPC/214-IJSC 2430008 ISSN:1793-351X 2nd Reading

Prompt: The sky is
Output: blue

The LLM replies with a continuation of "The sky is" string, which may not be appropriate for the user's needs. We next try for an improved result:

Prompt: Complete the sentence: The sky is
Output: so beautiful today.

Since we told the LLM to "complete the sentence", the result looks different as the LLM follows what it is told to do. Prompts contain any of the following elements and the format depends on the task:

Instruction — a description of a specific task you want the model to perform.
Context — further information or a particular context that directs the model to a response.
Input Data — the input or question that you are interested in asking.
Output Indicator — the type or format of the output, for example, a poem.

We can also design prompts for various tasks by using commands that instruct the model what we want to achieve. These prompts include: "Write", "Classify", "Summarize", "Translate", "Order", etc.

Prompt: Instruction: Translate the text to Spanish:
Text: "hello! "
Output: ¡Hola!

We can add contextual constraints to the prompt to give information to the model or to restrict the boundaries of the responses to the prompt. Context can be either instructions that specify how the model should behave or information that the model uses or references to generate a response. For example:

Prompt: Marbles:
Color: red
Number: 12
Color: blue
Number: 28
Color: yellow
Number: 15
Color: green
Number: 17
How many green marbles are there?
Output: There are 17 green marbles.

WSPC/214-IJSC 2430008 ISSN:1793-351X 2nd Reading

We can also add examples to enrich the context.

Prompt: Classify the following.
Options:
- red wine
- white wine
Text: Chardonnay
The answer is: white wine
Text: Cabernet
The answer is: red wine
Text: Moscato
The answer is: white wine
Text: Riesling
The answer is:
Output: white wine.

To demonstrate the utility of few-shot prompting, consider *sentiment analysis*: We take paragraphs of different opinions and label them with a sentiment classification. Then we condition the model to take a paragraph as input and generate a classification as output. Before transformer-based LLMs, recurrent neural networks were not very adaptive. Adding a new classification to be considered or asking the model not to classify but to summarize a paragraph required that the model needed to be modified and retrained.

A large transformer language model, however, does not need to be retrained. We can ask the model to classify a paragraph or summarize it when we know how to ask correctly. The model cannot classify a paragraph into categories A or B when the specifications for A and B are unclear. As we see next, the LLM can classify "positive" or "negative" sentiment once the model is told what "positive" and "negative" are. During training, the model learns these words and acquires the ability to follow simple instructions.

Prompt: Classify the text into positive, neutral, or negative:
Text: That musical performance was enjoyable.
Classification:
Output: The text "That musical performance was enjoyable" can be classified as positive.

The response was "positive". This is correct and concise. The model will also be able to classify "awesome" as positive. Identifying sentiment results from the beginning instruction: Classify the text into positive, neutral, or negative.

Prompt: Text: The room was very hot.
Classification: Negative
Text: The lecture was very intense.
Classification:

2430008.indd 36 24-09-2024 11:24:52

WSPC/214-IJSC 2430008 ISSN:1793-351X 2nd Reading

Output: The text "The lecture was very intense" can be classified as neutral. It describes the lecture's intensity without inherently conveying a positive or negative sentiment.

Prompt: Text: Today the weather is fantastic
Classification: Pos
Text: The furniture is small.
Classification: Neg
Text: I don't like your attitude
Classification: Neg
Text: That shot selection was awful
Classification:

Output: The text "That shot selection was awful" can be classified as negative, as it expresses strong disapproval or criticism of the shot selection.

Next, chain of thought reasoning allows models to decompose complex problems into intermediate steps that are solved individually. Moreover, the language-based nature of chain of thought makes it applicable to most tasks people can solve using language.

Prompt: Which is a faster way to get home?
Option 1: Take a 10 min bus, then a 40 min bus, and finally a 10 min train.
Option 2: Take a 90 min train, then a 45 min bike ride, and finally a 10 min bus.
Option 1 will take $10 + 40 + 10 = 60$ min.
Option 2 will take $90 + 45 + 10 = 145$ min.
Since Option 1 takes 60 min and Option 2 takes 145 min, Option 1 is faster.
Which is a faster way to get to work?
Option 1: Take a 1000 min bus, then a half hour train, and a 10 min bike ride.
Option 2: Take an 800 min bus, then an hour train, and finally a 30 min bike ride.

Output: Option 1 will take $1000 + 30 + 10 = 1040$ min.
Option 2 will take $800 + 60 + 30 = 890$ min.
Since Option 2 takes 890 min and Option 1 takes 1040 min, Option 2 is faster.

In Fig. 20, we request the LLM to create two images, the first from a word description and the second from an image plus a word description.

5.4. *Sample applications of generative AI*

Using the fundamental concepts supporting LLM training and prompt engineering, several interesting applications have emerged. Generative AI models are used to write novels, pass the legal bar exam, create marketing campaign materials, automatically regenerate financial reports, and for many other interesting tasks. LLMs have applications beyond the domain of text generation. They can be used to generate images from text, or text description of images, or, with instructions, to transform images.

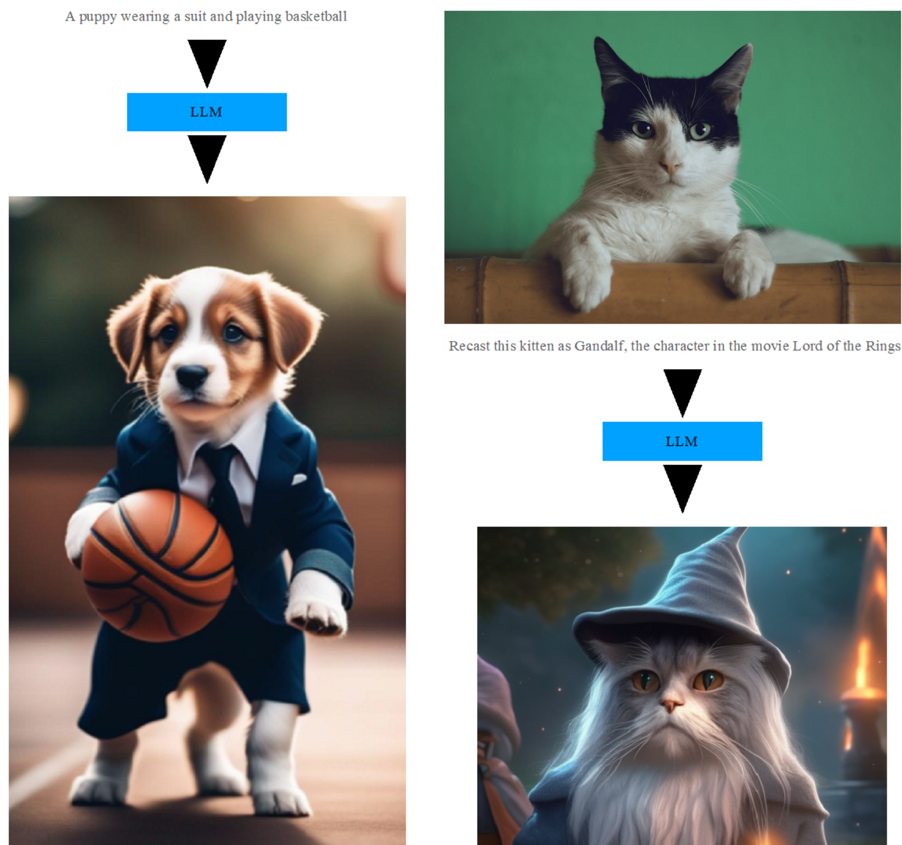


Fig. 20. The LLM creates an image of a suit-wearing basketball-playing puppy (with 5 legs?). The second image transforms a kitten into a Gandalf clone. Figure adapted from [2].

Image to Image Conversion, (url 1).

Image conversion includes transforming the external elements of an image, such as its color, medium, or form, while preserving its constitutive elements, as in Fig. 20. One example would be turning a daylight image into a nighttime image. This type conversion is also used to manipulate the attributes of an image, such as a face, colorize them, or change their style. Another example is to take images in the style of one painter and transform them in the style of another. Image conversion has also been used in advanced photo editing applications, e.g. to add or remove details, figures, or objects from pictures. All these tasks are performed using carefully designed prompts.

Text to Images and Images to Text, (url 4, url 10).

Generative AI models are also trained to generate images from text descriptions under a special class of models called *diffusion*. Diffusion models are a class of generative models that simulate the data generation process by transforming a simple starting distribution into the desired complex data distribution through a

sequence of invertible operations. These models can be used in applications including answering questions about an image, creating descriptions of images, and editing an image.

Text to Video

Interestingly, not only can transformers generate text to images, as we saw in Fig. 20, but current research is exploring possibility of transforming text to video. For example, "Show me a woman in a red parka skiing down a resort ski run" or "A man and women in black clothes are doing ice dancing". Many important questions remain, including how best to train transformers with video materials [40, 41]; see also (url 15).

Music Production, (url 6).

Using recurrent neural networks and variational autoencoders, generative AI has been trained to automatically create music from text descriptions. Models can learn melodic motifs, chord progression, as well as rhythmic elements. Applications include automatically synthesizing music from different genres, tweaking compositions, teaching music theory, and provoking creativity in music makers.

Code Generation, (url 2, url 12).

Generative AI can be used to create computer code based on prompts. This code can be generated in most programming languages using only a natural language description of the problem. Applications support more accurate code completion suggestions, generating unit tests for a function, automated debugging, generating documentation, teaching programming, and answering questions about pieces of code.

Protein Design and Generation, (url 13).

Trained AI models are used to generate new protein sequences. The training consists of several amino acid sequences of different proteins and then fine-tuning with a smaller subset of sequences that have contextual descriptions. Using language prompts for the desired properties of the required protein, the model automatically generates several proteins whose amino acid sequences meet those properties. This technology has been used to speed up the process of drug design for new medications.

Continued Extensions of Google's DeepMind Problem Solvers, (url 14).

Google's DeepMind research group designed the Alpha family of game playing programs. Their Alpha Zero program played go, chess, and shogi all at world class levels. A recent extension, AlphaGeometry [42], combines symbolic AI with deep learning language model technology to generate proofs in Euclidean geometry. Symbolic AI, along the line of traditional mathematical theorem proving programs [43, 44], suggests possible proof procedures. These partial solutions are then passed to deep learning models to explore symbolic and construction-based methods

for completing the proofs. The resulting program surpassed the geometry proof skills of previous computers and approaches the performance of the International Mathematical Olympiad competitors.

Finally, Table 4 presents the currently most used large language models, along with several of their design parameters. In the table, "tokens" refers to the piece of information used to train the model, syllables, words, components of images, etc. Having more parameters or training tokens does not always support having a better model, as code design and computational efficiency always remain important.

6. Summary and Conclusions

We have presented an overview of the evolution of neural network research and practice, beginning in the 1940s with models of human cortical processing and ending with transformer-based large language models. The complete story of this evolution requires a much deeper analysis of the algorithms and engineering practices than we have presented here [17, 16, 36]. Although the current application of these technologies is very impressive, as we saw in Sec. 5, and offers entirely new opportunities for AI practitioners, their remain important issues that society must address.

First, there is a lack of mathematical integrity and support for many of the engineering practices of modern AI. That engineers create programs "because they seem to work" is not sufficient justification for software tools that are coming to play important roles in modern life. Adding to this issue is the fact that many of our current LLM creators are keeping their engineering practices "company confidential". Protecting profit for corporate investment is important, but accountability for algorithm use and engineering decisions is critical for society's responsible use of this technology. A step in this direction would be to have more university collaboration, peer-reviewed publications, and open analysis of the current generation of AI practices. And linked to this, of course, is better education of the public to both the promise and problems of the current generation of AI problem solvers.

An important first step here is to acknowledge that LLMs reflect the data that they are trained on. If the data is racist and/or sexist, so also will be its product. If the model is trained on data of a certain date and location, its results will reflect this. The fine-tuning and prompt engineering of LLMs to extract useful results is more of a black art than a science, although several research groups are addressing this [37, 38].

Large language models are relatively knowledge free. They don't know what they know and, even worse, they don't know what they don't know. But they will always offer the user a response. Consider again the five-legged dog of Fig. 20. Their "knowledge" is how words/tokens are associated in large corpora, with attention networks assisting in building more complex associations. These models, in the human sense, do not "know" anything.

A further critique, explained by the absence of knowledge within the system, is a lack of transparency and explainability of LLMs and their products. When an LLM produces a product, it is close to impossible for it to produce any justification

for that result. For most of society's important decisions, justifications and explanations are required. Why was I approved for that bank loan at a particular interest rate? Why do you think this tumor is malignant? Why is this stock purchase recommended? Humanly responsible decision-making requires transparency and explanations.

The United States government (url 20, url 21, url 22) and the European Union have each proposed guidelines for AI use (url 23) as have major companies including IBM (url 19) and Microsoft (url 18). Professional societies including IEEE [45, 46] as well as a modern AI textbook [2] that contains chapters on the ethical use of modern AI technology.

We noted in our introduction that many of the neural network neuroscientist pioneers, including McColloch and Pitts, Hebb, and Rosenblatt, felt that their creations emulated aspects of human neuronal processing. That vision is no longer part of the prospectus of the current generation of LLMs [47].

URLs

url 1: Image to image translation GAN: https://github.com/eriklindernoren/PyTorch-GAN
url 2: BERT: https://github.com/google-research/bert
url 3: Bard, now Gemini: https://bard.google.com
url 4: ChatGPT: https://chat.openai.com
url 5: https://ai.meta.com/llama/
url 6: https://huggingface.co/docs/transformers/model_doc/t5
url 7: Stable diffusion: https://stablediffusionweb.com
url 8: https://imagen.research.google/
url 9: https://openai.com/dall-e-2
url 10: https://sites.research.google/phenaki/, https://phenaki.video
url 11: MusicGen: https://huggingface.co/spaces/facebook/MusicGen
url 12: Copilot: https://github.com/features/copilot
url 13: https://doi.org/10.1038/s42256-022-00532-1
url 14: https://www.nature.com/articles/s41586-023-06747-5
url 15: https://huggingface.co/tasks/text-to-video
url 16: https://storage.googleapis.com/deepmindmedia/gemini_v1_5_report.pdf
url 17: https://blog.google/technology/ai/google-gemini-ai/
url 18: https://www.microsoft.com/en-us/ai/responsible-ai?activetab=pivot1%3aprimaryr6
url 19: https://research.ibm.com/topics/trustworthy-ai
url 20: https://www.nist.gov/artificial-intelligence-safety-institute
url 21: https://www.cisa.gov/ai
url 22: https://www.commerce.gov/news/press-releases/2024/04/us-commerce-secretary-gina-raimondo-announces-expansion-us-ai-safety
url 23: https://www.aepd.es/sites/default/files/2019-12/ai-ethics-guidelines.pdf

ORCID

George F. Luger https://orcid.org/0009-0001-8164-5964

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser and I. Polosukhin, Attention is all you need, in *Advances in Neural Information Processing Systems* (Curran Associates Inc., San Francisco CA, 2017), pp. 5998–6008.

[2] G. F. Luger, *Artificial Intelligence: Principles and Practice* (Springer Nature, New York, 2024).

[3] W. S. McCulloch and W. Pitts, A logical calculus of the ideas immanent in nervous activity, *Bull. Math. Biophys.* **5** (1943) 115–133.

[4] D. O. Hebb, *The Organization of Behavior* (Wiley, New York, 1949).

[5] F. Rosenblatt, The perceptron: A probabilistic model for information storage and organization in the brain, *Psychol. Rev.* **65** (1958) 386–408.

[6] F. Rosenblatt, *Principles of Neurodynamics* (Spartan, New York, 1962).

[7] G. Boole, *An Investigation of the Laws of Thought* (Walton & Maberly, London, 1854).

[8] N. J. Nilsson, *Learning Machines* (McGraw-Hill, New York, 1965).

[9] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry* (MIT Press, Cambridge, MA, 1969).

[10] D. H. Ackley, G. E. Hinton and T. J. Sejnowski, A learning algorithm for Boltzmann machines, *Cogn. Sci.* **9** (1985) 147–169.

[11] G. E. Hinton and T. J. Sejnowski, Analyzing cooperative computation, in *Proc. 5th Annual Congress of the Cognitive Science Society*, 1983.

[12] G. E. Hinton and T. J. Sejnowski, Learning and relearning in Boltzmann machines, in *Parallel Distributed Processing*, eds. J. L. McClelland *et al.* (MIT Press, Cambridge MA, 1986), pp. 282–317.

[13] B. Widrow, and M. E. Hoff, Adaptive switching circuits, in *1960 IRE WESCON Convention Record* (IEEE, New York, 1969), pp. 96–104.

[14] D. E. Rumelhart, J. L. McClelland and The PDP Research Group, *Parallel Distributed Processing* (MIT Press, Cambridge, MA, 1986).

[15] G. F. Luger, *Artificial Intelligence: Structures and Strategies for Complex Problem Solving* (Addison Wesley-Pearson, New York, 2009).

[16] J. Devlin, M. Chen, K. Lee and K. Toutanova, BERT: Pre-training of deep bidirectional transformers for language understanding, preprint (2018), https://arxiv.org/abs/1810.04805.

[17] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei and I. Sutskever, Language models are unsupervised multitask learners, *OpenAI Blog*, **1**(8) (2019) 9.

[18] E. Charniak, *Introduction to Deep Learning* (MIT Press, Cambridge MA, 2019).

[19] Y. LeCun and Y. Bengio, Convolutional networks for images, speech, and time series, in *The Handbook for Brain theory and Neural Networks* (MIT Press, Cambridge MA, 1995), pp. 255–258.

[20] Z. C. Lipton, J. Berkowitz and C. Elkan, A critical review of recurrent neural networks for sequence learning, preprint (2015), arXiv:1506.00019.

[21] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville and Y. Benjio, Generative adversarial nets, in *Advances in Neural Information Processing Systems* (Curran Associates Inc., San Francisco CA, 2014), pp. 2672–2680.

WSPC/214-IJSC 2430008 ISSN:1793-351X 2nd Reading

[22] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford and X. Chen, Improved techniques for training GANs, preprint (2016), arXiv:1606.03498.
[23] P. Isola, J. Y. Zhu, T. Zhou and A. A. Efros, Image-to-image translation with conditional adversarial nets,preprint (2016), arXiv:1611.07004.
[24] J. Ho and S. Ermon, Generative adversarial imitation learning, in *Advances in Neural Information Processing Systems*, Vol. 29, (Curran Associates Inc., San Francisco CA), pp. 4565–4573.
[25] D. Bahdanau, K. Cho and Y. Bengio, Neural machine translation by jointly learning to align and translate (2014). ArXiv:1409.0473.
[26] R. Bommasani *et al.*, On the opportunities and risks of foundation models, preprint (2021), arXiv:2108.07258.
[27] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen and V. Stoyanov, RoBERTa: A robustly optimized BERT pretraining approach, preprint (2019), arXiv:1907.11692.
[28] I. O. Beltagy, M. E. Peters and A. Cohan, Longformer: The long-document transformer, preprint (2020), arXiv:2004.05150v2.
[29] Gemini Team: Google DeepMind, Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context (2024), https://blog.google/technology/ai/google-gemini-ai/.
[30] A. Dosovitskiy *et al.*, An image is worth 16×16 words: Transformers for image recognition at scale, in *Int. Conf. Learning Representations*, 2021, pp. 1–22.
[31] A. Gulati *et al.*, Conformer: Convolution-augmented transformer for speech recognition, in *Proc. Interspeech 2020*, 2020, pp. 5036–5040.
[32] A. Zhang, Z. C. Lipton, M. Li and A. J. Smola, *Deep Dive into Deep Learning* (Cambridge University Press, Cambridge, 2023).
[33] E. A. Nadaraya, On estimating regression, *Theory Probab. Appl.* **9**(1) (1964) 141–142.
[34] G. S. Watson, Smooth regression analysis, *Sankhyā: Indian J. Stat. Ser. A* (1964) 359–372.
[35] A. Morsali, M. Heidari, S. Heydarian and T. Abdeini, MLP-Attention: Improving transformer architecture with MLP attention weights, in *Int. Conf. Learning Representations (ICLR-23)*, 2023, pp. 1–5.
[36] M. Phuong and M. Hutter, Formal algorithms for transformers, preprint (2022), arXiv:2207.09238v1.
[37] T. Shen, R. Jin, Y. Huang, C. Liu, W. Dong, Z. Guo, X. Wu, Y. Liu and D. Xiong, Large language model alignment: A survey, preprint (2023), arXiv:2309.15025v1.
[38] R. Rafailov, S. Archit, E. Mitchell, E. Stephano, C. D. Manning and C. Finn, Direct preference optimization: Your language model is secretly a reward model, in *37th Conf. Neural Information Processing, NeurIPS-23*, 2023 (Curran Associates Inc., San Francisco CA), arXiv:2305.18290.
[39] S. Casper *et al.*, Open problems and fundamental limitations on reinforcement learning from human feedback, preprint (2023), arXiv:2307.1517.
[40] W. Hong, M. Ding, W. Zheng, X. Liu, and J. Tang, CogVideo: Large-scale pretraining for text-to-video generation via transformers, preprint (2022), arXiv.2205.15868.
[41] G. Chen, A simple text to video model via transformer, preprint (2023), arXiv:2309.14683v1.
[42] T. H. Trinh, Y. Wu, Q. V. Le, H. He and T. Luong, Solving Olympiad geometry without human demonstrations, *Nature* **625** (2024) 476-482.
[43] H. Gelernter and N. Rochester, Intelligent behavior in problem-solving machines, *IBM J. Res. Develop.* **2**(4), (1958) 336–345.

2430008.indd 43 24-09-2024 11:24:54

[44] A. Bundy, L. Byrd, G. Luger, C. Mellish, R. Milne and M. Stone, Solving mechanics problems using meta-level inference, in *Proc. Sixth Int. Joint Conf. Artificial Intelligence*, *IJCAI-79*, 1979, pp. 1017–1027.

[45] C. Huang, Z. Zhang, B. Mao and X. Yao, An overview of artificial intelligence ethics, *IEEE Trans. Artif. Intell.* (2022), doi:10.1109/TAI.2022.3194503.

[46] K. Shahriari and M. Shahriari, IEEE standard review — Ethically aligned design: A vision for prioritizing human wellbeing with artificial intelligence and autonomous systems, in *IEEE Canada Int. Humanitarian Technology Conference (IHTC)*, 2017, pp. 197–201.

[47] G. F. Luger, *Knowing Our World: An Artificial Intelligence Perspective* (Springer Nature, New York, 2021)