Analysis of suitable generative algorithms for the generation of safety-critical driving data in the field of autonomous driving

Nico Schick*

Machine learning algorithms are being increasingly used in the field of autonomous driving. For example, generative algorithms can be used to generate time series corresponding to safety-critical driving scenarios. This paper answers the following scientific question:

Which generative algorithms are particularly suitable for generating time series appropriate for the study of autonomous driving?

Keywords: Autonomous driving, safety-critical driving scenarios, time series, generative algorithms, Generative Adversarial Network (GAN), Restricted Boltzmann Machine (RBM), Recurrent Neural Network (RNN), Conditional Restricted Boltzmann Machine (CRBM), Factored Conditional Restricted Boltzmann Machine (FCRBM), Recurrent Temporal Restricted Boltzmann Machine (RTRBM), Variational-Autoencoder Generative Adversarial Network (VAE-GAN), Recurrent-Conditional Generative Adversarial Network (RCGAN), Time-Series Generative Adversarial Network (TimeGAN)

I. Motivation

Approximately 3700 people die in traffic accidents each day [1] [2] [3] [4] [5]. The most frequent cause of accidents is human error [6]. Autonomous driving can significantly reduce the number of traffic accidents. To prepare autonomous vehicles for road traffic, the software and system components must be thoroughly validated and tested. However, due to their criticality, there is only a limited amount of data for safety-critical driving scenarios. Such driving scenarios can be represented in the form of time series. These represent the corresponding kinematic vehicle movements by including vectors of time, position coordinates, velocities, and accelerations. There are several ways to provide such data. For example, this can be done in the form of a kinematic model. Alternatively, methods of artificial intelligence or machine learning can be used. These are already being widely used in the development of autonomous vehicles. For example, generative algorithms can be used to generate safety-critical driving data. A novel taxonomy for the generation of time series and suitable generative algorithms will be described in this paper. In addition, a generative algorithm will be recommended and used to demonstrate the generation of time series associated with a typical example of a driving-critical scenario.

*N. Schick, M. Sc. studied Applied Computer Sciences (M. Sc.) and Computer Engineering (B. Eng.) at the Esslingen University of Applied Sciences. e-mail: Nico.Schick@hs-esslingen.de

II. Time Series

Sequential data represents an ordered list of events. This type of data can be found in various use cases, such as in biology (e.g. DNA or protein sequences) or symbolic sequences (e.g. logical sequences of a customer purchase). Another important type of sequential data are time series. These contain numerical data, typically with a fixed and discrete time interval. Time series are found in a wide variety of applications. These occur in nature, in the form of temperature or climate, and in economics, in the form of stock prices. Sensor data of vehicles are also time series. With the help of time series analysis, the structure of time series can be understood. [6] [7]

III. Generative Algorithms

Generative algorithms belong to the field of machine learning and have gained significant attention in recent years. This type of algorithm can be understood as the counterpart to discriminative algorithms. They can, based on a probability distribution, generate new data. For this, information about the characteristics of the features are needed. In this regard, the characteristics of the different observations from the original data set, respectively, training data set, are transformed into a probability model. In the course of a stochastic process, the generative algorithm approximates a probability distribution of the training data set. The training phase of the probability model can be considered complete as soon as the generated data hardly differ from the original data set. Classical generative algorithms do not require labeled data sets (unsupervised learning). However, there are also mixed forms, which also depend on labels (unsupervised and supervised learning). [8] [9] A closer look at the literature reveals a higher-level taxonomy of generative algorithms in general. [8]

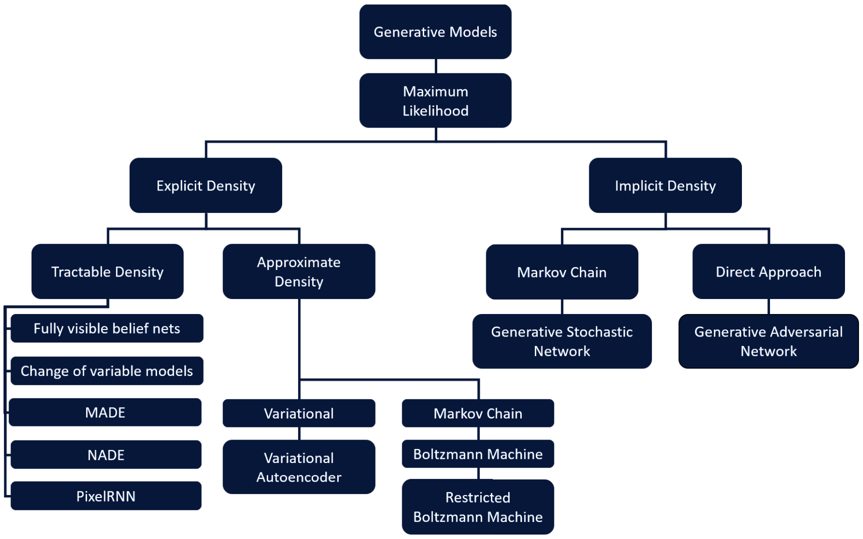

Fig. 1 Taxonomy of generative models

As seen in Figure 1, all generative algorithms are derived from the maximum likelihood method. A basic classification into explicit and implicit densities is made. Here, density refers to the probability density function (PDF) of the data set. The term ‘explicit density’ refers to an explicit or a closed expression of such a PDF. The implicit density method, on the other hand, estimates the PDF during the training phase. The explicit approach distinguishes between tractable densities and approximate densities. Generative algorithms, which belong to the group of tractable density, can be executed in polynomial time. These include algorithms such as Fully visible belief nets, Neural Autoregressive Distribution Estimation (NADE), Masked Autoencoder for Distribution Estimation (MADE), Pixel Recurrent Neural Networks (PixelRNN) or change of variable models. With generative algorithms, which instead belong to the group of approximate densities, a known PDF is approximated during the training phase. In particular, the Variational Autoencoder (VAE) as well as the Restricted Boltzmann Machine (RBM) are to be mentioned in this regard. VAEs were one of the first known generative neural networks of their kind. They are used in the field of image processing and feature reduction. RBMs are an extension of the Boltzmann Machines. They are used, for example, in the generation of human motion sequences. With the implicit approach, a subdivision between Generative Stochastic Networks (GSN) and Generative Adversarial Networks (GAN) takes place. GSNs are Markov chains and mimic the Gipps sampler of a Depp Boltzmann Machine. They can handle sequential data, but are not suitable for time series. [8] [9]

A. Generative Adversarial Networks

Generative Adversarial Networks (GAN) have gained great popularity in recent years. Moreover, they have been able to achieve continual improvement in their quality. The range of applications for GANs is diverse. GANs are used in different domains, such as image, video, audio, text, speech processing as well as time series. Goodfellow et al. developed the first concept of GAN in 2014. According to Figure 1, GANs determine a PDF in an implicit-direct manner. In more detail, a stochastic procedure is defined that is capable of generating synthetic data directly. The generation of new synthetic data is based on existing data sets. The basic structure of a GAN model consists of two artificial neural networks (ANNs), the so-called generator (G) and the discriminator (D), which perform a zero-sum game. [8] [9] The structure of the model is illustrated in Figure 2.

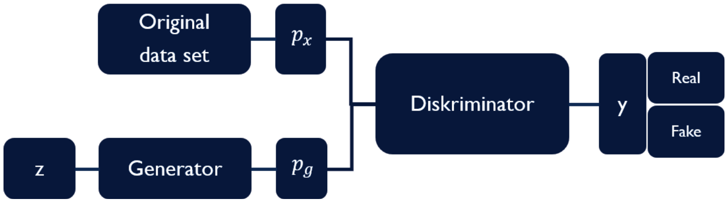

Fig. 2 Structure of a GAN model [8]

The task of the generator is to generate new synthetic data, whereas these data are evaluated or criticized by the discriminator. At the beginning, a probability distribution p_x is generated from an original data set, respectively, a training data set. It is used for representing the features of the original data set x in a latent space (sub-space). At first, the generator is initialized with a latent random vector z. The goal of the generator is to learn to generate synthetic data, according to an approximate distribution (p_g). The discriminator, on the other hand, tries to distinguish between the generated data and the original data. Here, y indicates whether x comes from p_x (with $y = 1$) or from p_g (with $y = 0$). GANs include white noise at the beginning of the training exclusively. Both networks interact with each other until the generator generates synthetic data that cannot be distinguished by the discriminator from the original data. This process is also known as a minimax game (also a zero-sum game), where the loss function of the discriminator is maximized and that of the generator is minimized. The total loss function $V(G, D)$ is defined as follows: [8] [9]

$$\min_G \max_D V(G, D) = \mathbb{E}_{x\sim p_{data}(x)} \left[\log D(x)\right] + \mathbb{E}_{z\sim p_z(z)} \left[\log(1 - D(G(z)))\right] \tag{1}$$

During training, the GAN model is optimized alternately. This involves fixing the weights of one network while optimizing the weights of the other one, and vice versa. At the beginning of the training, the generator is fixed first, to adjust the weights of the discriminator. Then, the weights of the discriminator are fixed, to update those of the generator. This process is repeated at each training iteration until the distribution of the synthetic data set p_g resembles the original data set p_x. After several training steps, the generator and discriminator reach a state where neither can improve. In this way, the global optimum is reached as soon as p_g and p_x are identical. This means that the discriminator can no longer distinguish between those two distributions. This is achieved as soon as $D(x) = 0.5$. Afterwards, the training phase is terminated and the discriminator is discarded. The discriminator has fulfilled its task by helping the generator to reach its target distribution ($p_g = p_x$) in order to be able to generate new data based on the approximation p_g. GANs are characterized by the high quality of the generated data. The diversity of applications is to be emphasized also. Due to the explosive nature of GANs, it is expected that this type of algorithm will continue to improve. There are already many publications about GANs. The tendency is still increasing. The training phase of a GAN is considered as a major challenge. It is mathematically complex and can lead to consequent numerical instabilities. The reason for this is the already mentioned minimax problem of the loss function. The evaluation of the generated data of a GAN is difficult to implement. Currently, expert knowledge (qualitative consideration) is used in this regard, for example. Alternatively, similarity measures (quantitative consideration) of underlying PDFs can be applied also. [8] [9]

B. Restricted Boltzmann Machines

Restricted Boltzmann Machines (RBM) belong to the group of generative algorithms that represent an explicit probability model. An RBM is an ANN consisting of a visible layer and a hidden layer. They have the ability to learn a probability distribution of the original data set. RBMs were developed in 1985 by Hinton and Sejnowski. They can be used for dimensionality reduction, classification, regression, collaborative filtering, feature learning, topic modeling, as well as sequences. They are an extension of Boltzmann machines and are constrained with respect to the connections between the visible and hidden nodes. [8] The abstract structure of an RBM model is shown in Figure 3.

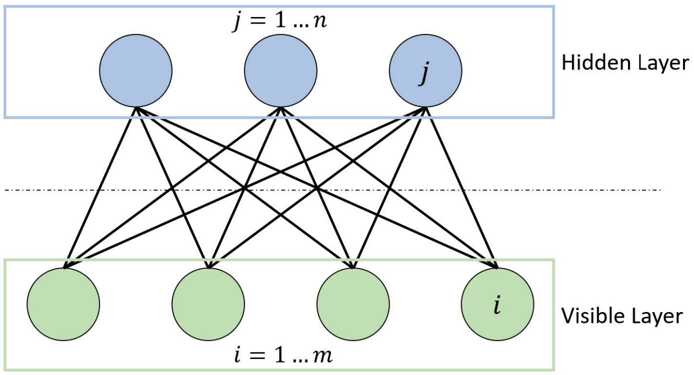

Fig. 3 Structure of an RBM model [8]

The graph illustrates a bipartite undirected graph. This consists of m visible units as well as n hidden units. Each node in the visible layer is connected to each node in the hidden layer. However, due to the constraint, nodes in the same layer are not connected to each other. This allows the use of efficient and numerically more stable training algorithms, such as the gradient-based Contrastive Divergence (CD) algorithm. RBMs approximate a distribution function stochastically. The joint probability distribution of the model is given by the Gibbs distribution [8]

$$p(v, h) = \frac{1}{Z} e^{-E(v,h)} \tag{2}$$

with the loss fuction [8]

$$E(v, h) = -\sum_{i=1}^{n} \sum_{j=1}^{m} w_{ij} h_i v_j - \sum_{j=1}^{m} b_j v_j - \sum_{i=1}^{n} c_i h_i \tag{3}$$

Here w_{ij} denotes the weight of the connection between units v_i and h_j. The visible units v_i represent observable data, whereas the hidden units capture the dependencies between the observed variables. There are two different bias units (hidden and visible biases). The hidden biases c_i are used in the forward pass and the visible biases b_j to reconstruct the input during the backward pass. The variable Z denotes the distribution function and is given by summing over all possible pairs of visible and hidden vectors: [8]

$$Z = \sum_{v,h} e^{-E(v,h)} \tag{4}$$

The forward pass respectively the training of the network is shown in Figure 4 with several inputs x.

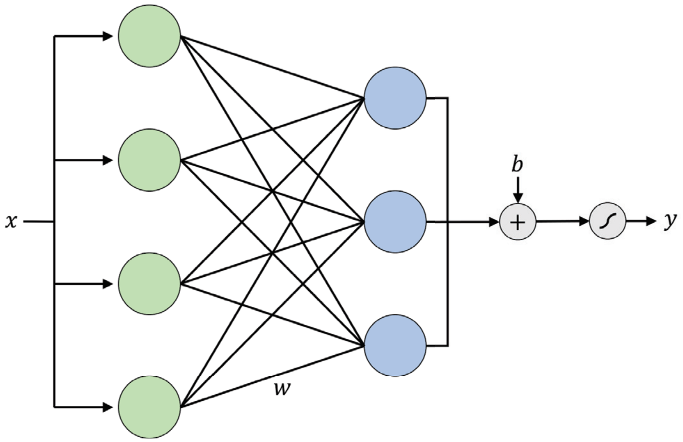

Fig. 4 Forward pass of an RBM model [8]

In the first step, the inputs x are multiplied by the weights w_i and then added with the bias b_j. The output is then passed through a sigmoid activation function. The output determines whether the hidden node is activated or not (binary activation). The weights represent a matrix, with the number of input nodes (number of rows), and the number of hidden nodes (number of columns). Before the bias value is added to the first hidden node, a vector multiplication of the inputs multiplied by the first weight column is applied. [8]

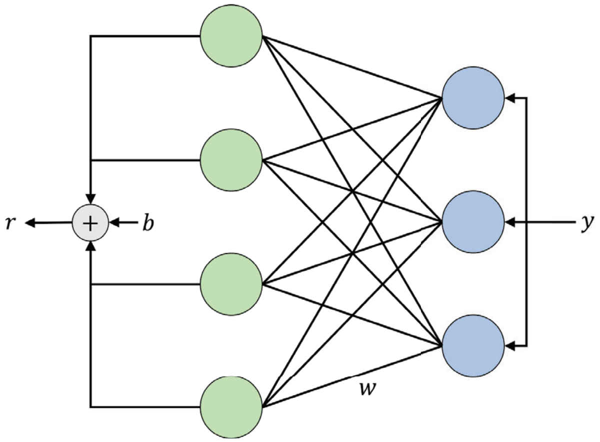

Fig. 5 Backward pass of an RBM model [8]

Once the training has been completed, the backward pass follows. Figure 5 shows its structure. While the RBM uses inputs to predict node activations during the forward pass, it tries to estimate the probability of inputs during the backward pass based on given activations. The difference between the reconstructions and the original input is due to the random initialization of the weights applied at the beginning. This is also referred to as reconstruction error, and is the difference between the reconstructed values r and the original input values x. In an iterative learning process, this error is used to optimize the weights of the network until a minimum error is reached. To measure the distance between two PDFs, RBMs use the Kullback–Leibler divergence. The CD is used to minimize the distance of both probability distributions accordingly. RBMs are more efficient and numerically stable compared to the unconstrained Boltzmann machine, due to their constraints as well as in conjunction with the CD algorithm applied. Those RBMs which contain recurrent nodes use LSTM cells mostly to build-up the recurrent behaviour accordingly. This promotes the information flow of the hidden units. [8]

IV. Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are known in the modeling of sequence data due to their recurrent characteristics. By preserving past information over time, RNNs are able to learn the temporal properties of the original data set. RNNs have applications in language translation, speech generation, handwriting recognition, image-to-text translation, and machine translation. [8] [9] The general structures of a standard neural network and those of an RNN model are illustrated in the following.

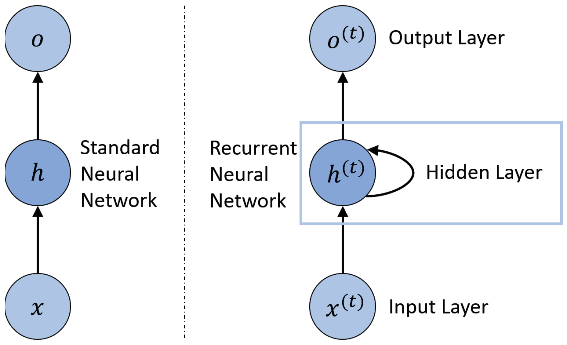

Fig. 6 General structures of standard neural network and of an RNN model [8]

In Figure 6 both networks have only one hidden layer. In this simplified representation, not all units of the input layer x, hidden layer h, and the output layer o are illustrated. In contrast to a standard feed forward neural network, in an RNN the hidden layer receives two inputs. These include the values from the current input layer as well as from the hidden layer (from the previous time step $t - 1$). Such an information flow is represented in the model as a loop, which is also referred to as a recurrent edge. [8] [9] For a closer look at the information flow, an unfolded representation of an RNN model is illustrated in Figure 7.

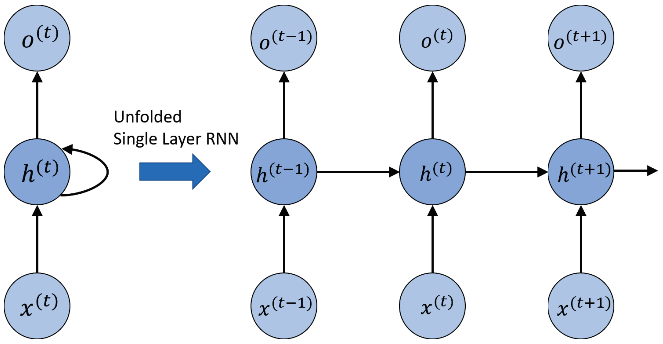

Fig. 7 Unfolded representation of an RNN [8]

At the beginning of the training, the hidden units are initialized with zeros or small random values. In Figure 8, different weight matrices of a standard RNN are shown.

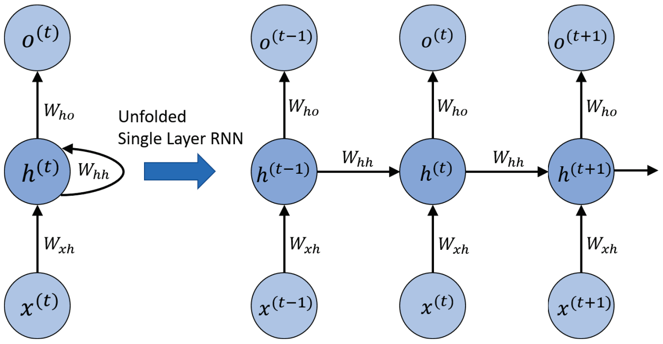

Fig. 8 Weight matrices of an RNN [8]

The directed edges between units are associated with weights, making up a weight matrix. These weights are independent of time. There are several different weight matrices in a single-layer RNN, as listed below: [8] [9]

- W_{xh}: The weight matrix between the input layer and the hidden layer.
- W_{hh}: The weight matrix associated with the recurrent edge.
- W_{ho}: The weight matrix between the hidden layer and the output layer.

The calculation of the activations of an RNN model is described in more detail below. For the hidden layer, the network input z_h (pre-activation) is calculated by a linear combination from adding the sum of the multiplications of the weight matrices with the corresponding vectors to the bias vector b_h: [8] [9]

$$z_h^{(t)} = W_{xh}x^{(t)} + W_{hh}h^{(t-1)} + b_h \tag{5}$$

The activations of the hidden units at time step t are calculated as follows: [8] [9]

$$h^{(t)} = \phi_h \left(z_h^{(t)} \right) = \phi_h \left(W_{xh} x^{(t)} + W_{hh} h^{(t-1)} + b_h \right) \tag{6}$$

The size b_h is the bias vector for the hidden units and ϕ_h is the activation function of the hidden layer. Once the activations of the hidden units at the current time step are calculated, the activations of the output units are determined as follows: [8] [9]

$$o^{(t)} = \phi_o \left(W_{ho} h^{(t)} + b_o \right) \tag{7}$$

The variable ϕ_o is the activation function and b_o is the bias vector of the output layer. The training phase of an RNN is a major challenge in general. This refers especially to the recurrent edges. During the computation of the gradients, the so-called vanishing gradient problem and exploding gradient occur. These are shown by the examples in Figure 9.

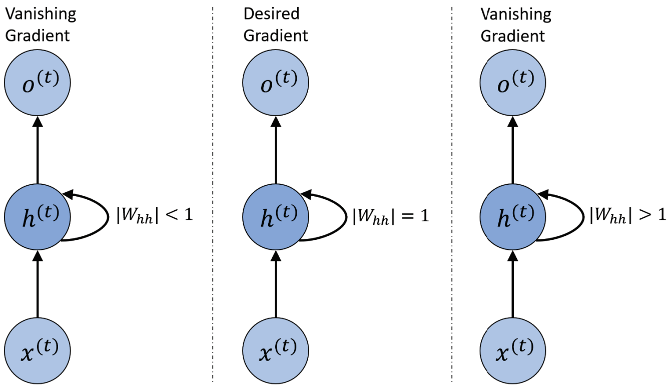

Fig. 9 Vanishing and Exploding problem [8]

Basically, the network has $t - k$ multiplications. Therefore, multiplying the weight w by itself, results in a factor w_{t-k} for $t - k$ times. When $|w| < 1$, this factor takes a very small value (vanishing) if $t - k$ is large. On the other hand, in case the weight of the recurrent edge is $|w| > 1$, then the factor w_{t-k} becomes very large (exploding) if $t - k$ is large also. Such a high value of $t - k$ is present in case the underlying data have larger lengths. A solution to avoid vanishing or exploding gradients can be achieved by ensuring $|w| = 1$. There are at least three different approaches, as listed below: [8] [9]

- Gradient Clipping (GC)
- Truncated Backpropagation over Time (TBPTT)
- Long Short Term Memory (LSTM)

GC is used to specify a threshold value for the gradients: gradients that exceed this value are rescaled to produce a more plausible behaviour of the gradient descent. In contrast, TBPTT limits the number of time steps that the signal can backpropagate after each forward pass. However, such a truncation approach limits the number of steps that the gradient can effectively flow back and update the weights properly. [8] [9]

LSTM cells, on the other hand, are able to successfully counteract such gradient problems efficiently, also when dealing with larger data sets. The building block of an LSTM include a memory cell that essentially replaces the hidden layer of a standard RNN. In each memory cell, there is a recurrent edge that has the desired weight $w = 1$. The values associated with this recurrent edge are collectively referred to as the cell state. [8] [9] The unfolded structure of a modern LSTM cell is shown in Figure 10.

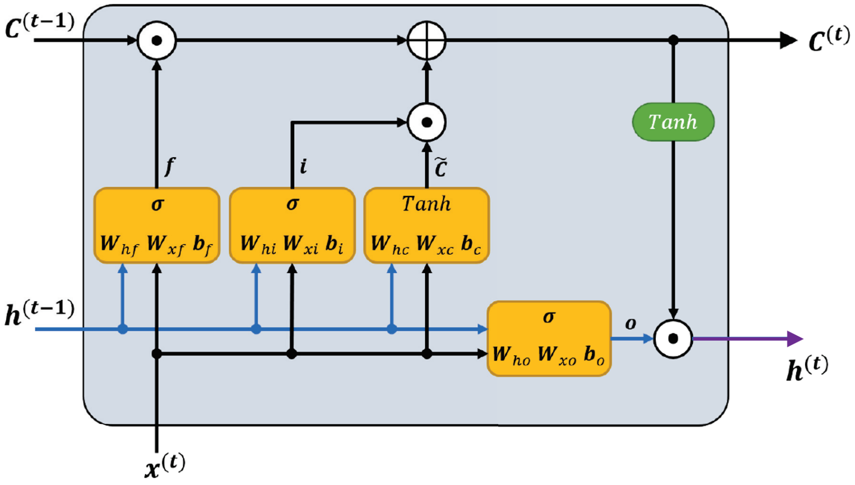

Fig. 10 LSTM cell of the hidden unit of an RNN [8]

To obtain the cell state at the time step $C^{(t)}$, the cell state from the previous time step $C^{(t-1)}$, without being directly multiplied by a weight factor, is updated. The information flow in this memory cell is affected by several computational units. In Figure 10, the sign $\odot$ refers to element-wise multiplication and $\oplus$ to element-wise addition. The size $x^{(t)}$ refers to the input data at time t and $h^{(t-1)}$ refers to the hidden units at time $t-1$. The four yellow blocks are each characterized by either a sigmoid or tanh activation function, as well as a series of weights. These computational units carry out a linear combination by performing matrix–vector multiplications on their inputs $h^{(t-1)}$ and $x^{(t)}$. They are likewise referred to as gates. In an LSTM cell, there are three different types of gates: forget gate, input gate, and output gate. [8] [9]

A forget gate f_t allows the memory cell to reset the cell state without growing indefinitely. More specifically, a forget gate decides what kind of information to pass through and what kind of information to suppress. It is calculated as follows: [8] [9]

$$f_t = \sigma\left(W_{xf}x^{(t)} + W_{hf}h^{(t-1)} + b_f\right) \quad (8)$$