CE6146
Introduction to Deep Learning
Autoencoders

Chia-Ru Chung

Department of Computer Science and Information Engineering

National Central University

2023/10/26

20231012 Exercise #16

- Which layer do we not have control over?

  (A) Input layer (B) Hidden layer (C) Output layer **(D) None**

- The term “control over” in the context of this question refers to the ability to modify or tweak the parameters or structure of a particular layer during the design or training phase of the neural network.
- (A) Input Layer: Initially, it may seem like we have no control over the input layer because its size and structure are dictated by the features in the dataset. However, through feature selection and feature engineering, we do exert some level of control over which features are used, and hence, indirectly control the input layer.
- (B) Hidden Layer: We have the most control over the hidden layers. We can choose how many there are, how many nodes each one has, and what activation functions to use.
- (C) Output Layer: The structure of the output layer is often determined by the specific problem we're solving (e.g., classification, regression). However, we still have control over aspects like the activation function to use.
- (D) None: This option implies that we have some level of control over all layers. We can select features for the input layer, fully design the hidden layers, and choose appropriate activation functions for the output layer based on the problem at hand.

20231012 Exercise #17

- What is dropout used for?

  (A) Speeding up training (B) **Regularization**

  (C) Activation (D) Normalization

- The primary purpose of dropout is to prevent overfitting in neural networks, making it a form of regularization.
- During training, dropout randomly sets some of the neuron outputs in a layer to zero, reducing the capacity or thinning the network during each training phase.
- By doing this, it ensures that the network can generalize well to new data. Therefore, the most appropriate answer is (B) Regularization.
- While dropout may seem like it could speed up training because it uses fewer neurons during each pass, this isn't actually the case.
- Dropout usually requires more training epochs to converge because the network is effectively “thinner” or “reduced” during each forward and backward pass.
- This requires more iterations to adequately train the model, often leading to a longer training process, not a shorter one. Hence, (A) Speeding up training would not be the correct answer.

20231019 Exercise

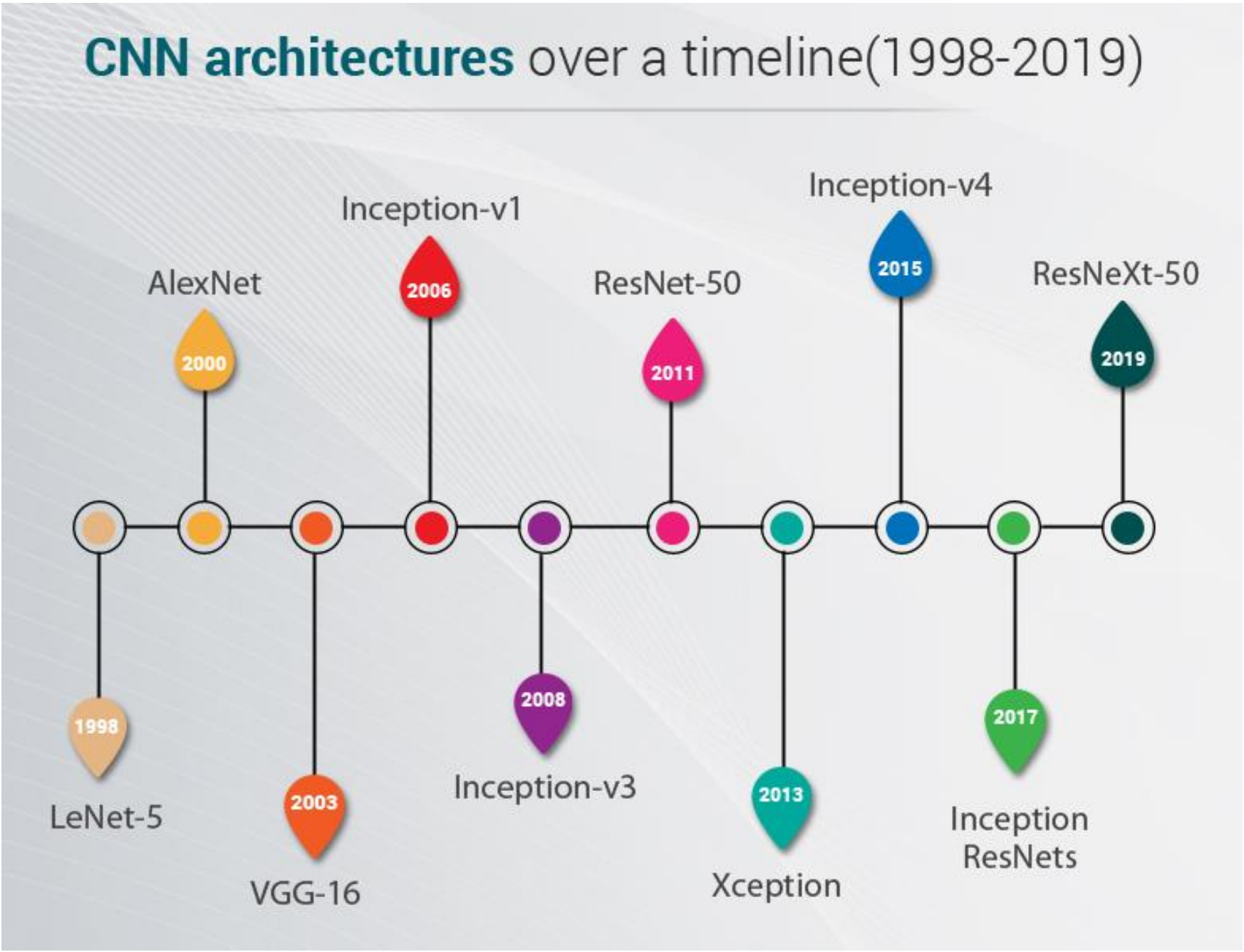
1. **C**	2. **B**	3. **D**	4. **A**	5. **D**
6. **A**	7. **D**	8. **D**	9. **B**	10. **B**
11. **C**	12. **A**	13. **D**	14. **B**	15. **C**
16. **A**	17. **D**	18. **C**	19. **D**	20. **C**

Outline

- Review
- Autoencoders
- Hand-on

Review

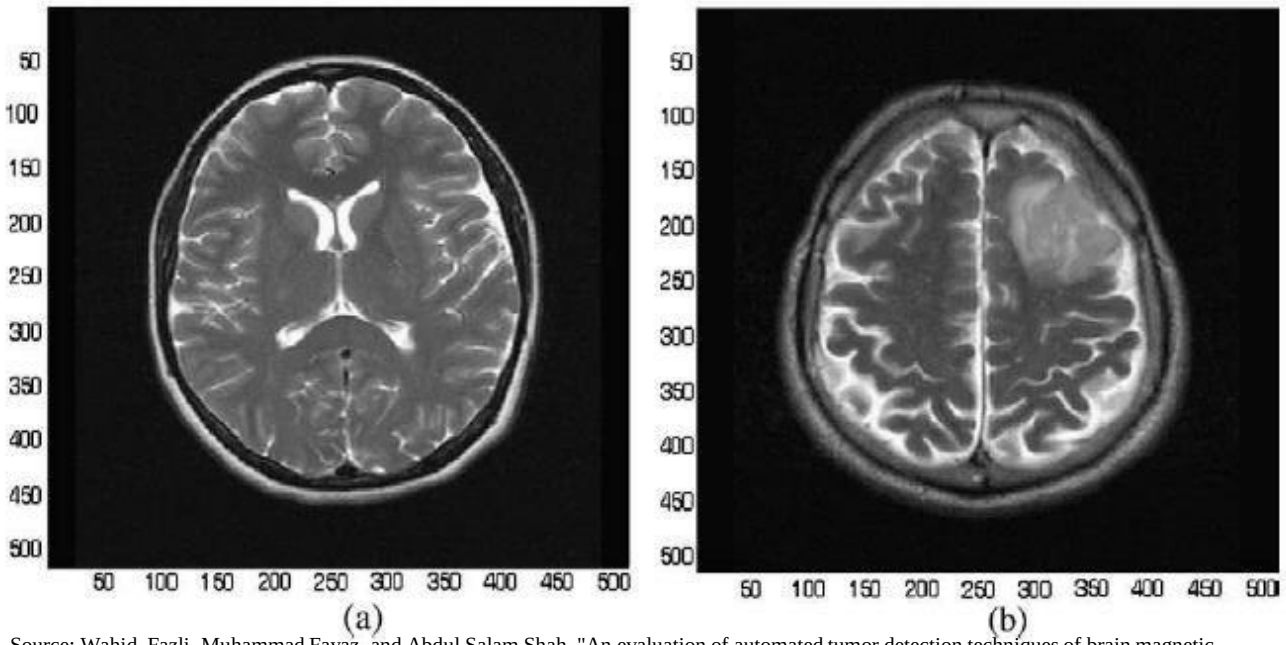
- Convolutional Neural Networks
- Recurrent Neural Networks
- Long Short-Term Memory Networks



Source: https://www.aismartz.com/blog/cnn-architectures/

Examples for Dilation

- Scenarios Suitable for Dilation:
  - Medical Imaging:
  - Example: Segmenting tumors or organs in MRI or CT scans.



Source: Wahid, Fazli, Muhammad Fayaz, and Abdul Salam Shah. "An evaluation of automated tumor detection techniques of brain magnetic resonance imaging (MRI)." International Journal of Bio-Science and Bio-Technology 8.2 (2016): 265-278.

- Scenarios Not Suitable for Dilation:
  - Fine-Grained Classification:
  - Example: Distinguishing between closely related species of birds or models of cars.

Source: Li, A. X., Zhang, K. X., & Wang, L. W. (2019). Zero-shot fine-grained classification by deep feature learning with semantics. International Journal of Automation and Computing, 16, 563-574.

RNN – Elman Network

- At each time step t, an RNN computes the hidden state h_t and the output y_t using the current input x_t, the previous hidden state h_{t-1}, and the weights and biases of the network:

$$h_t = f_1(W_{hh}h_{t-1} + W_{xh}x_t + b_h) \text{ and } y_t = f_2(W_{hy}h_t + b_y)$$

  - W_{hh}, W_{xh}, and W_{hy} are the weight matrices
  - b_h and b_y, are the bias vectors
  - f_1 and f_2 are activations functions

Source: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks

RNN – Jordan Network

- At each time step t, an RNN computes the hidden state h_t and the output y_t using the current input x_t, the previous hidden state h_{t-1}, and the weights and biases of the network:

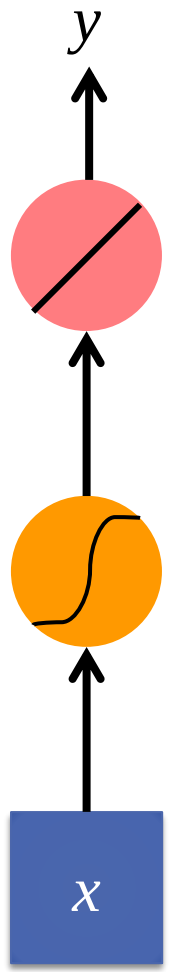
$$h_t = f_1(W_{hh}h_{t-1} + W_{xh}x_t + W_{yh}h_t + b_h) \text{ and } y_t = f_2(W_{hy}h_t + b_y)$$

  - W_{hh}, W_{xh}, W_{yh}, and W_{hy} are the weight matrices
  - b_h and b_y, are the bias vectors
  - f_1 and f_2 are activations functions

Source: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks

Example – Elman Network (1/3)

- Suppose we have a very simplified RNN with only one neuron in the hidden layer. This RNN will process a sequence of numbers one at a time, with the goal of learning to predict the next number in the sequence.
- We will use the hyperbolic tangent (tanh) activation function for the hidden state and a linear activation function for the output.
- Input sequence: (1, 2, 3, 4, …)
- Initialization: initialize all weights and biases to 1, and an initial hidden state to 0 ($W_{hh} = W_{xh} = W_{hy} = b_h = b_y = 1$ and $h_0 = 0$).



Example – Elman Network (2/3)

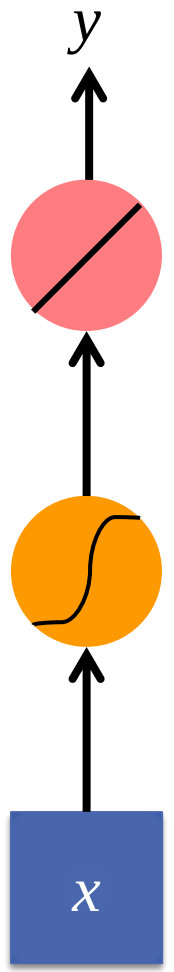
$W_{hh} = W_{xh} = W_{hy} = b_h = b_y = 1$ and $h_0 = 0$

- Time step $t = 1$:
  - Input: $x_1 = 1$
  - Compute the new hidden state:

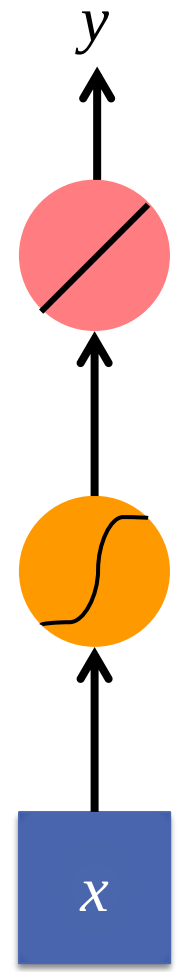
$$h_1 = \tanh(W_{hh}h_0 + W_{xh}x_1 + b_h) = \tanh(1 \cdot 0 + 1 \cdot 1 + 1) = \tanh(2) \approx 0.9640.$$

  - Compute the output:

$$y_1 = W_{hy}h_1 + b_y \approx 1 \cdot 0.9640 + 1 \approx 1.9640.$$



Example – Elman Network (3/3)



- Time step $t = 2$:

  $W_{hh} = W_{xh} = W_{hy} = b_h = b_y = 1$ and $h_1 \approx 0.9640$

  - Input: $x_2 = 2$
  - Compute the new hidden state:

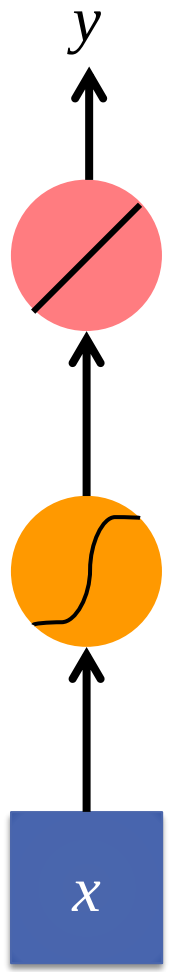
    $$h_2 = \tanh(W_{hh}h_1 + W_{xh}x_2 + b_h) \approx \tanh(1 \cdot 0.9640 + 1 \cdot 2 + 1) \approx 0.9993.$$

  - Compute the output:

    $$y_2 = W_{hy}h_2 + b_y \approx 1 \cdot 0.9993 + 1 \approx 1.9993.$$

Example – Jordan Network (1/3)

- Suppose we have a very simplified RNN with only one neuron in the hidden layer. This RNN will process a sequence of numbers one at a time, with the goal of learning to predict the next number in the sequence.
- We will use the hyperbolic tangent (tanh) activation function for the hidden state and a linear activation function for the output.
- Input sequence: (1, 2, 3, 4, …)
- Initialization: initialize all weights and biases to 1, and an initial hidden state and initial output to 0 ($W_{hh} = W_{xh} = W_{yh} = W_{hy} = b_h = b_y = 1$ and $h_0 = y_0 = 0$).



Example – Jordan Network (2/3)

- Time step $t = 1$:

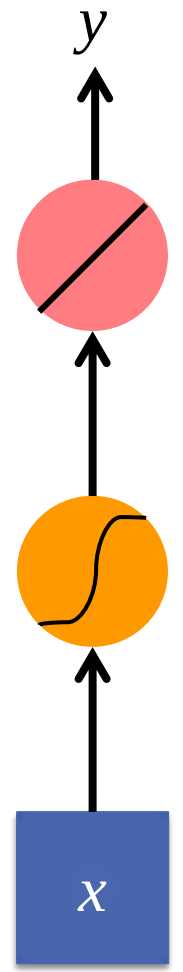
  $W_{hh} = W_{xh} = W_{yh} = W_{hy} = b_h = b_y = 1$ and $h_0 = y_0 = 0$

  - Input: $x_1 = 1$
  - Compute the new hidden state:

$$h_1 = \tanh\left(W_{hh}h_0 + W_{xh}x_1 + W_{yh}y_0 + b_h\right) = \tanh(1 \cdot 0 + 1 \cdot 0 + 1 \cdot 1 + 1)$$
$$= \tanh(2) \approx 0.9640.$$

  - Compute the output:

$$y_1 = W_{hy}h_1 + b_y \approx 1 \cdot 0.9640 + 1 \approx 1.9640.$$



Example – Jordan Network (3/3)

$W_{hh} = W_{xh} = W_{yh} = W_{hy} = b_h = b_y = 1$ and $h_1 \approx 0.9640, y_1 \approx 1.9640$

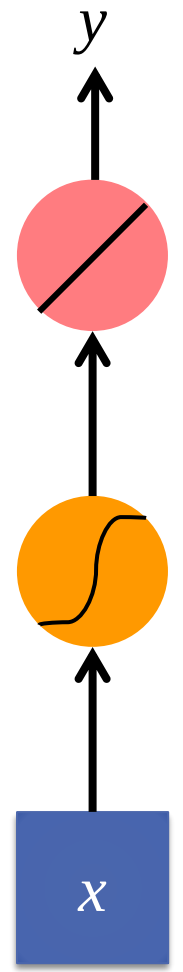
- Time step $t = 2$:
  - Input: $x_2 = 2$
  - Compute the new hidden state:

$$h_2 = \tanh\left(W_{hh}h_1 + W_{xh}x_2 + W_{yh}y_1 + b_h\right)$$

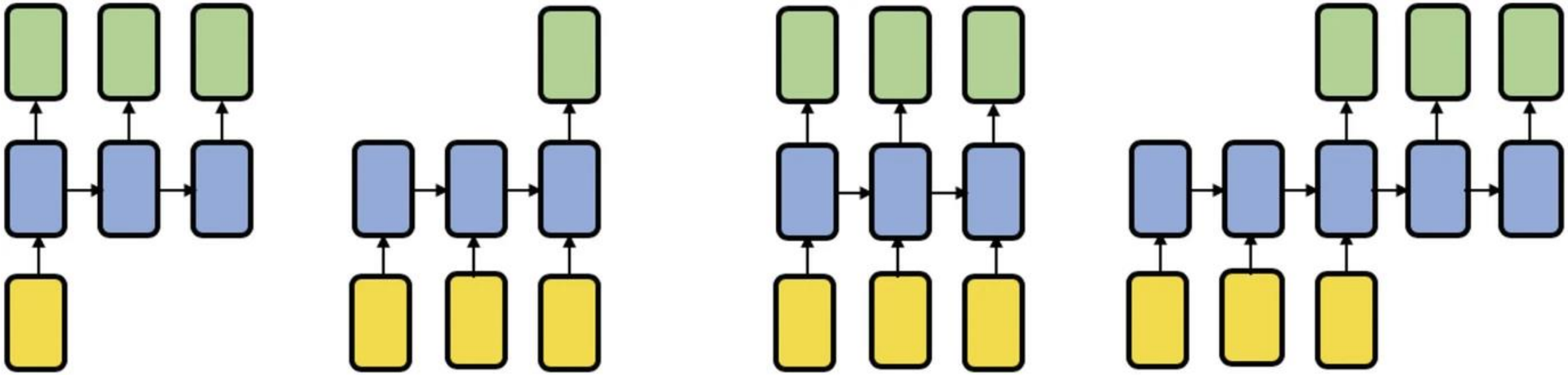
$$\approx \tanh(1 \cdot 0.9640 + 1 \cdot 2 + 1 \cdot 1.9640 + 1) \approx 0.9999.$$

  - Compute the output:

$$y_2 = W_{hy}h_2 + b_y \approx 1 \cdot 0.9999 + 1 \approx 1.9999.$$



Types of RNN



One-to-Many (e.g., image to text)

Many-to-One (e.g., sequence classification)

Many-to-Many (e.g., sequential prediction)

Many-to-Many (e.g., seq2seq)

Four types of RNN structure, where yellow box represents input layer, blue for the hidden layer, and green for output and prediction layers

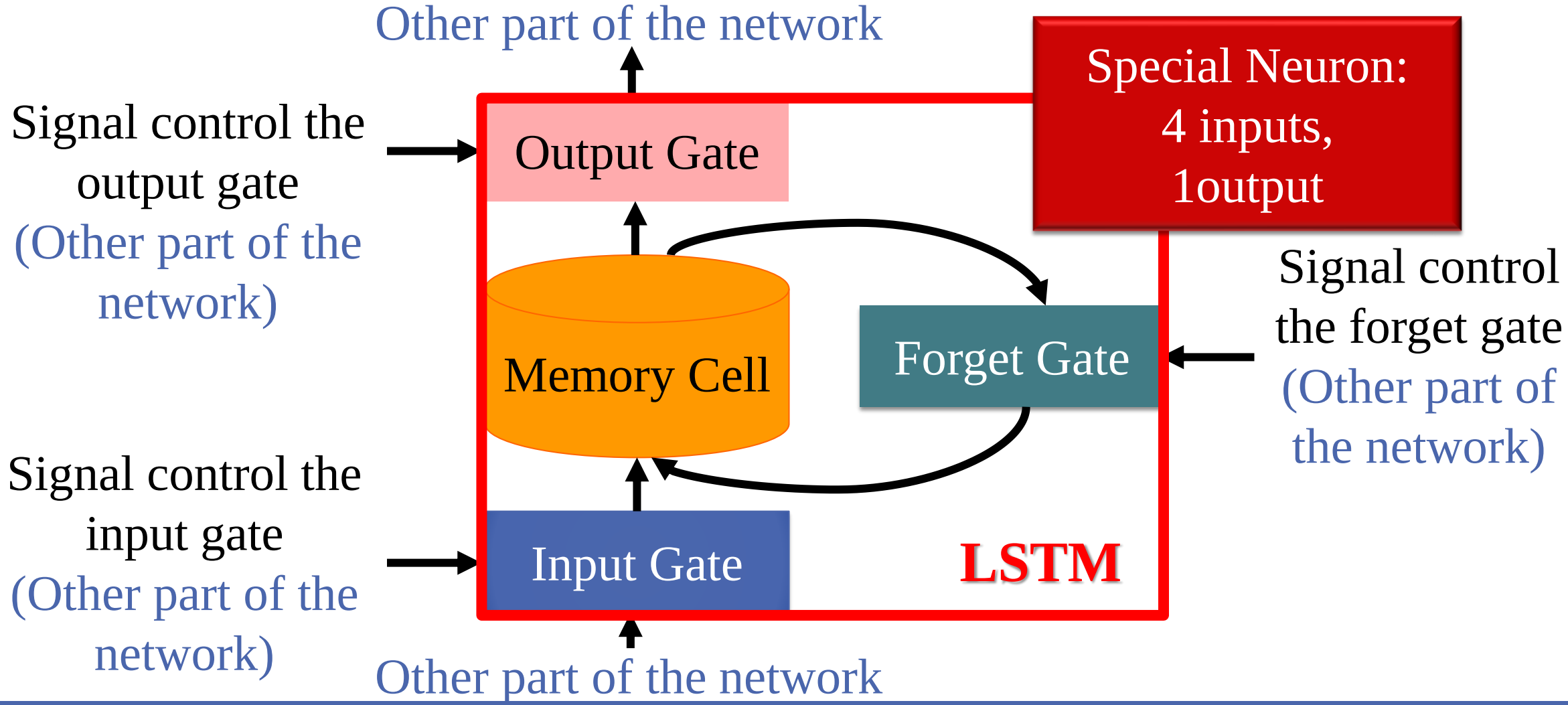
Source: https://link.springer.com/chapter/10.1007/978-3-030-82184-5_7

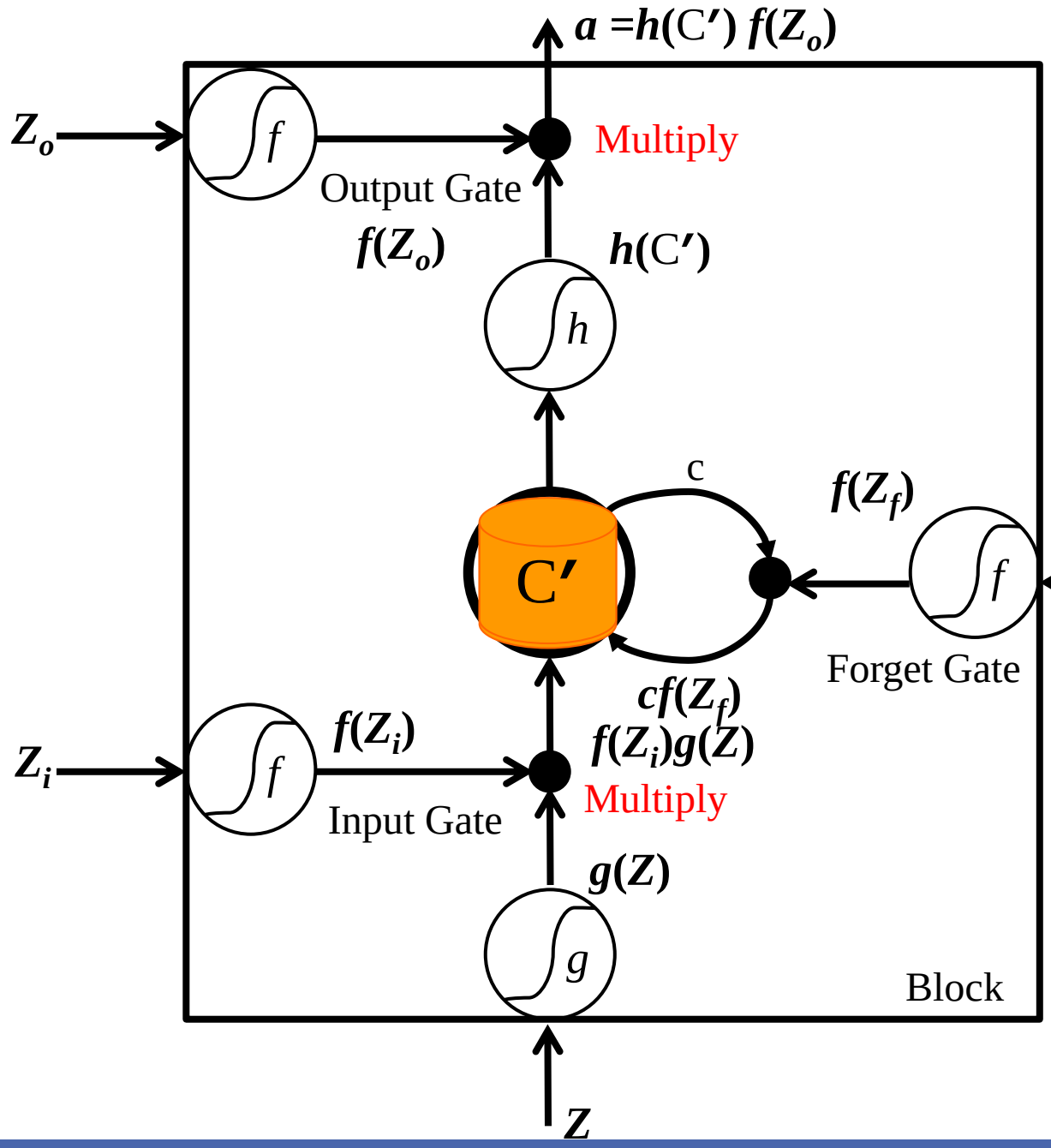
LSTM – Basic Architecture

- Input Gate: Determines the extent to which incoming data should be stored in the cell state.
- Forget Gate: Decides the extent to which previous information should be forgotten.
- Cell State: Holds the network's long-term memory.
- Output Gate: Controls the extent to which the information in the cell state is used to compute the output of the unit.
- Hidden State: Holds the network's short-term memory.

Source: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks

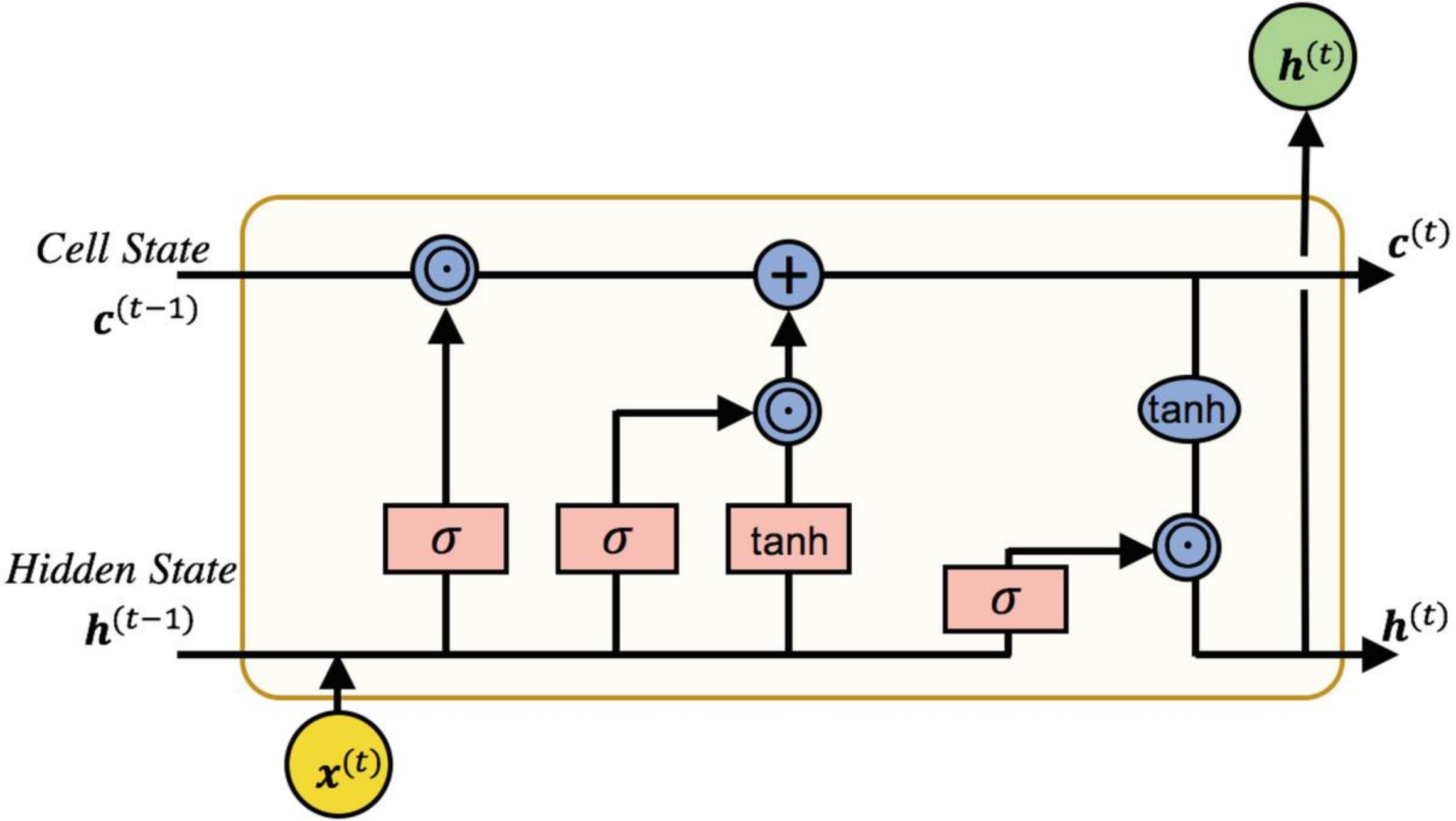
Long Short-Term Memory (LSTM)



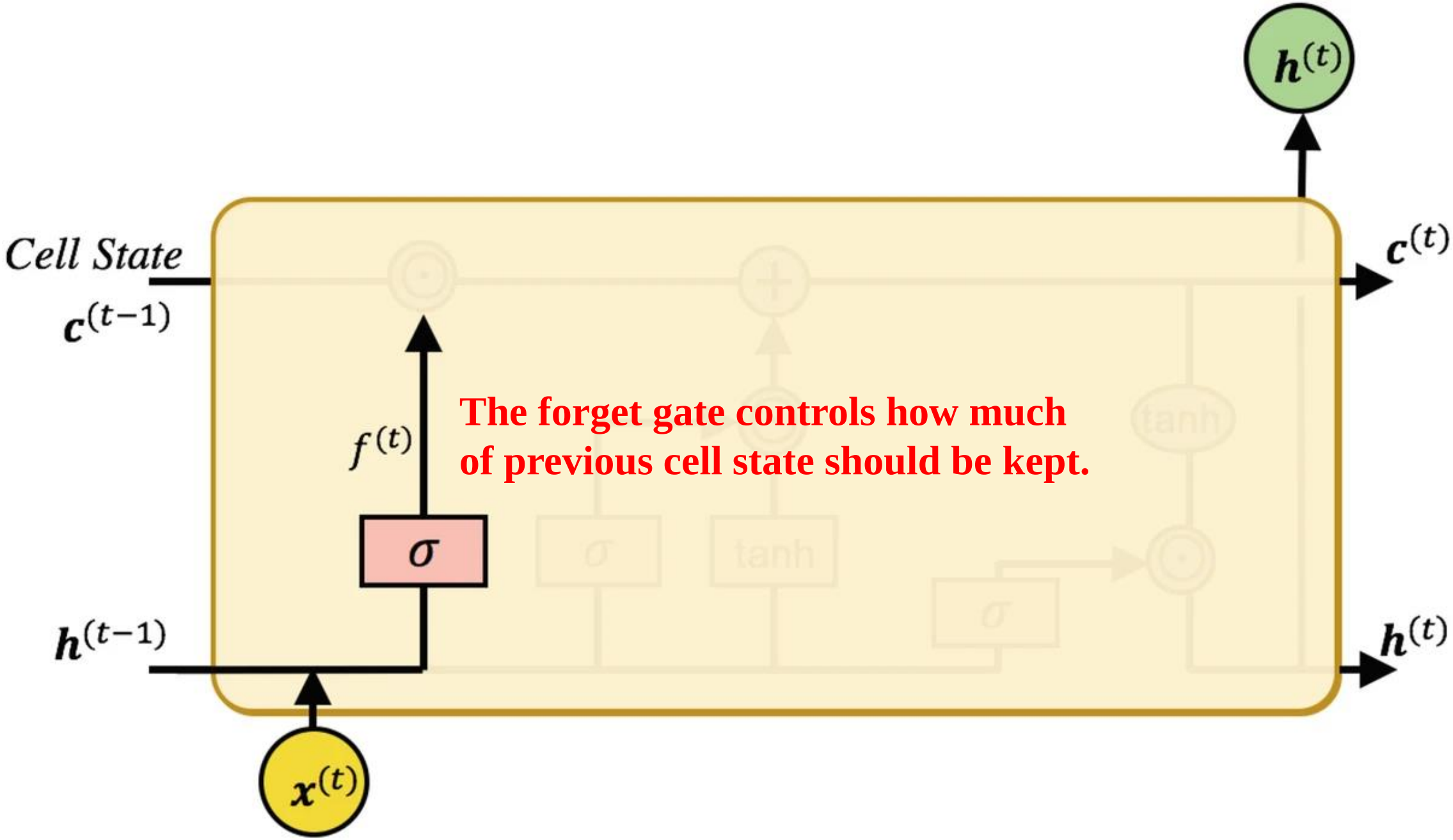


- Activation function *f* is usually sigmoid function
- Between 0 and 1
- Mimic open and close gate

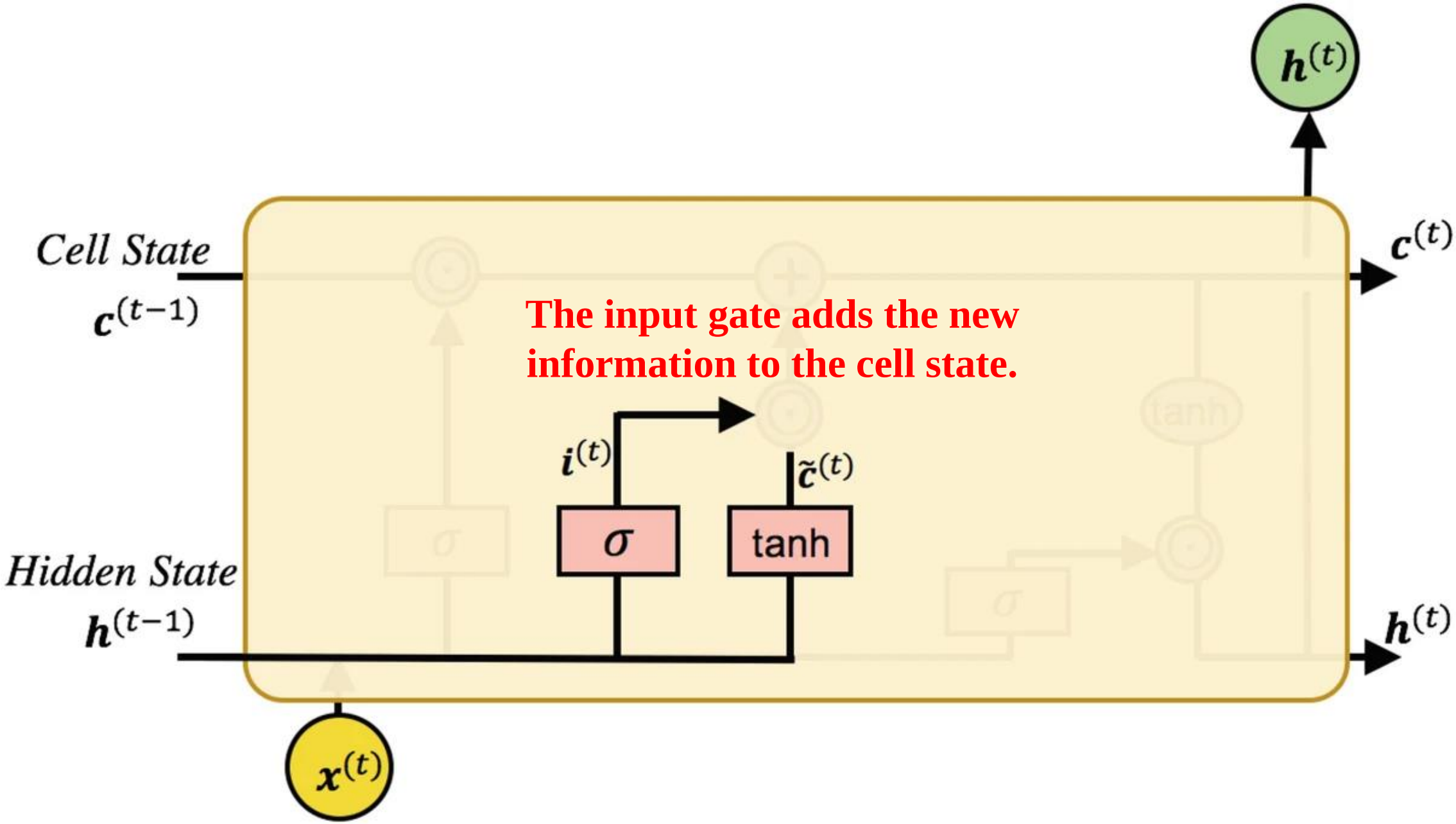
$$C' = f(Z_i)g(Z) + cf(Z_f)$$



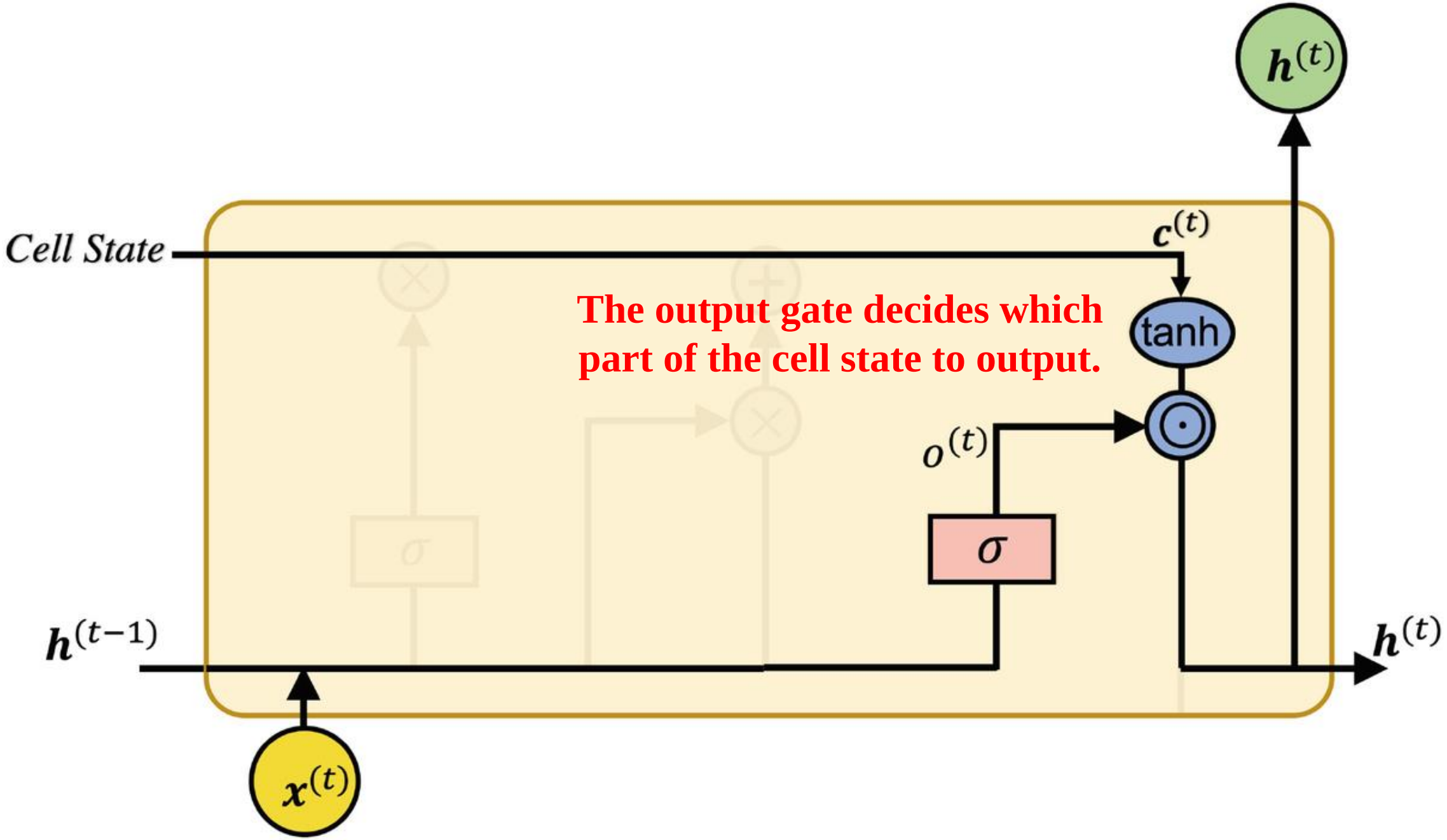
Source: https://link.springer.com/chapter/10.1007/978-3-030-82184-5_7



Source: https://link.springer.com/chapter/10.1007/978-3-030-82184-5_7



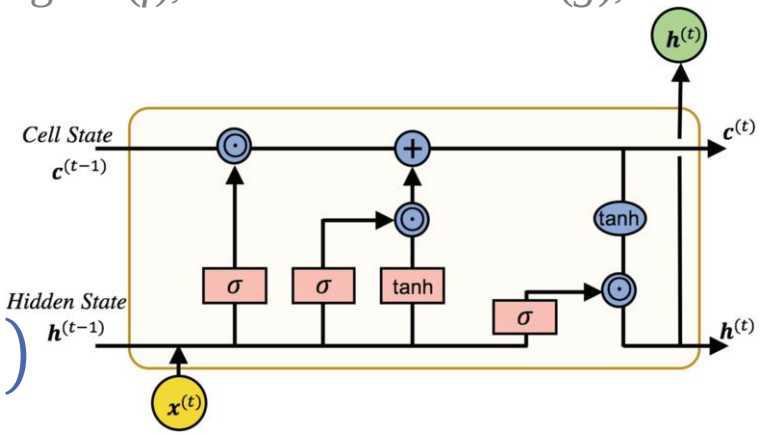
Source: https://link.springer.com/chapter/10.1007/978-3-030-82184-5_7



Source: https://link.springer.com/chapter/10.1007/978-3-030-82184-5_7

LSTM

- σ: The sigmoid activation function.
- W_{mn} and b_{mn}: Weight matrices.
  The first subscript denotes whether the weights are related to the input (i) or the previous hidden state (h). The second subscript denotes which part of the LSTM the weights are associated with: the input gate (i), the forget gate (f), the cell candidate (g), or the output gate (o).



- At each time step t
  - Input Gate: $i_t = \sigma(W_{hi}h_{t-1} + b_{hi} + W_{ii}x_t + b_{ii})$
  - Forget Gate: $f_t = \sigma\left(W_{hf}h_{t-1} + b_{hf} + W_{if}x_t + b_{if}\right)$
  - Cell Candidate: $\tilde{c}_t = \tanh\left(W_{hg}h_{t-1} + b_{hg} + W_{ig}x_t + b_{ig}\right)$
  - New Cell State: $c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$
  - Output Gate: $o_t = \sigma(W_{ho}h_{t-1} + b_{ho} + W_{io}x_t + b_{io})$
  - New Hidden State: $h_t = o_t \circ \tanh(c_t)$

Source: https://link.springer.com/chapter/10.1007/978-3-030-82184-5_7

Example – LSTM (1/2)

- Suppose we have a very simplified LSTM with only one neuron in the hidden layer. This LSTM will process a sequence of numbers one at a time, with the goal of learning to predict the next number in the sequence.
- We will use the hyperbolic tangent (tanh) activation function for the hidden state and a linear activation function for the output.
- Input sequence: (1, 2, 3, 4, …)
- Initialization: initialize all weights and biases to 1, and an initial hidden state and initial output to 0 ($W_{hi} = W_{ii} = W_{hf} = W_{if} = W_{hg} = W_{ig} = W_{ho} = W_{io} = b_{hi} = b_{ii} = b_{hf} = b_{if} = b_{hg} = b_{ig} = b_{ho} = b_{io} = 1$ and $h_0 = c_0 = 0$).

Example – LSTM (2/2)

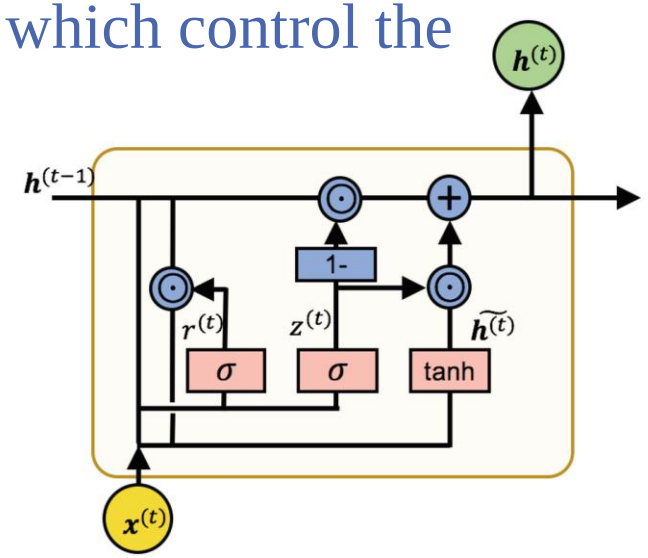
- Time step $t = 1$:
  - Input: $x_1 = 1$

$W_{hi} = W_{ii} = W_{hf} = W_{if} = W_{hg} = W_{ig} = W_{ho} = W_{io} = b_{hi} = b_{ii} = b_{hf} = b_{if} = b_{hg} = b_{ig} = b_{ho} = b_{io} = 1$ and $h_0 = c_0 = 0$

  - Input gate: $i_1 = \sigma(W_{hi}h_0 + b_{hi} + W_{ii}x_1 + b_{ii}) = \sigma(1 \cdot 0 + 1 + 1 \cdot 1 + 1) = \sigma(3) \approx 0.9526$
  - Cell candidates: $\tilde{c}_1 = \tanh\left(W_{hg}h_0 + b_{hg} + W_{ig}x_1 + b_{ig}\right) = \tanh(1 \cdot 0 + 1 + 1 \cdot 1 + 1) = \tanh(3) \approx 0.9951$
  - Forget gate: $f_1 = \sigma\left(W_{hf}h_0 + b_{hf} + W_{if}x_0 + b_{if}\right) = \sigma(1 \cdot 0 + 1 + 1 \cdot 1 + 1) = \sigma(3) \approx 0.9526$
  - New cell state: $c_1 = f_1 \cdot c_0 + i_1 \cdot \tilde{c}_1 \approx 0.9526 \cdot 0 + 0.9526 \cdot 0.9951 = 0.9479$
  - Output gate: $o_1 = \sigma(W_{ho}h_0 + b_{ho} + W_{io}x_1 + b_{io}) = \sigma(1 \cdot 0 + 1 + 1 \cdot 1 + 1) = \sigma(3) \approx 0.9526$
  - New hidden state: $h_1 = o_1 \cdot tanh(c_1) \approx 0.9526 \cdot tanh(0.9479) = 0.7038$

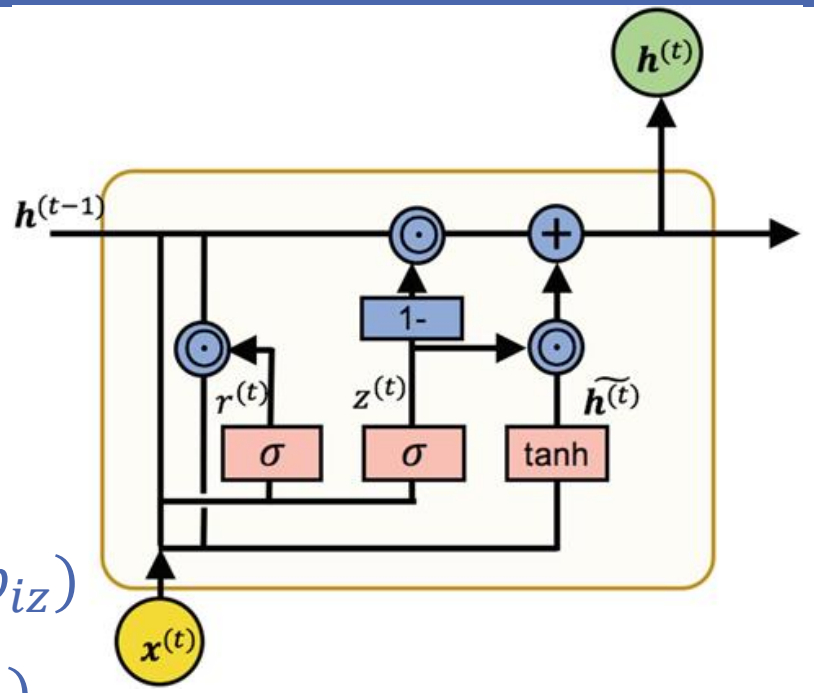
Gated Recurrent Unit (GRU)

- The Gated Recurrent Unit (GRU) is a type of recurrent neural network that aims to solve the vanishing gradient problem, similar to the LSTM, but with a simpler structure.
- The GRU has two gates, a reset gate and an update gate, which control the flow of information through the network.



Source: https://link.springer.com/chapter/10.1007/978-3-030-82184-5_7

GRU



- At each time step t
  - Update Gate: $z_t = \sigma(W_{hz}h_{t-1} + b_{hz} + W_{iz}x_t + b_{iz})$
  - Reset Gate: $r_t = \sigma(W_{hr}h_{t-1} + b_{hr} + W_{ir}x_t + b_{ir})$
  - Candidate Activation:: $\tilde{h}_t = \tanh(r_t \circ (W_{hh}h_{t-1} + b_{hh}) + W_{ih}x_t + b_{ih})$
  - New Hidden State: $h_t = (1 - z_t) \circ h_{t-1} + z_t \circ \tilde{h}_t$

Source: https://link.springer.com/chapter/10.1007/978-3-030-82184-5_7

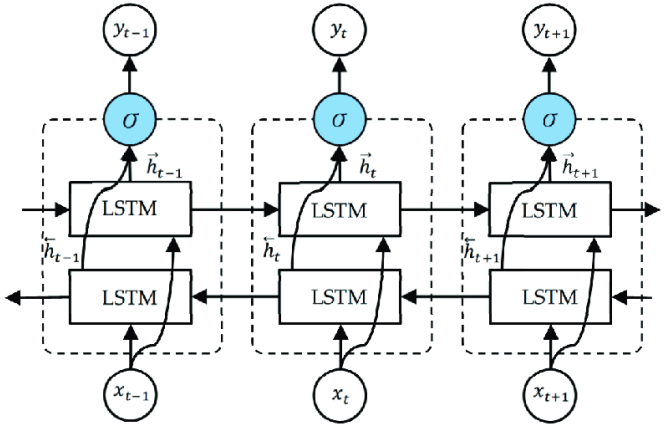
LSTM vs. GRU (1/2)

LSTM	GRU
Input Gate: Controls how much of the newly computed information from the current input will be stored in the cell state. **Forget Gate**: Determines the extent to which information from the previous cell state is retained or forgotten. **Output Gate**: Modulates the amount of information from the current cell state that will be used to compute the hidden state, which can then be output or passed to the next time step.	**Update Gate**: Serves a dual purpose: decides how much of the previous hidden state to retain and how much of the new candidate activation to use for updating the hidden state. It plays a role similar to the combined function of the input and forget gates in LSTM. **Reset Gate**: Determines how to combine the new input with the previous hidden state to compute the candidate activation. There isn't a direct equivalent in LSTM, though its function is somewhat akin to how the input gate modulates the influence of the new input in LSTM.

LSTM vs. GRU (2/2)

- LSTM has three gates (input, forget, and output) that control the flow of information into, within, and out of the cell, enabling a finer-grained control compared to GRU.
- GRU simplifies the gating mechanism with only two gates (update and reset), making it computationally more efficient but potentially less expressive than LSTM.
- The Update Gate in GRU performs a function similar to a combination of the Input and Forget Gates in LSTM.
- There isn't a direct counterpart in GRU for the Output Gate in LSTM. Instead, the function of controlling the exposure of the internal state is embedded within the formula for computing the new hidden state in GRU.
- The Reset Gate in GRU doesn't have a direct counterpart in LSTM, but it plays a somewhat similar role to the Input Gate in terms of modulating the influence of new input.

Bidirectional LSTM



- Bidirectional Long Short-Term Memory Networks (Bi-LSTMs) are an extension of traditional LSTMs that can improve model performance by providing access to future context.
- A Bi-LSTM consists of two LSTMs: a forward LSTM and a backward LSTM that process the input sequence in opposite directions.
- The key idea is to capture information from both past and future states.

Source: https://www.researchgate.net/figure/The-unfolded-architecture-of-Bidirectional-LSTM-BiLSTM-with-three-consecutive-steps_fig2_344751031

Backpropagation Through Time

- Backpropagation Through Time (BPTT) is the training algorithm used for RNNs. It's a variant of the standard backpropagation algorithm adapted for sequences.
- In BPTT, the network is unrolled over time, and the error is computed at each step based on the difference between the predicted output and the actual output. The error is then propagated backward through the network from the final step to the first, updating the weights to minimize the error.

BPTT Steps (1/4)

1. Forward Pass:

   - Process the entire sequence through the RNN, computing the hidden states and outputs at each time step *t*.

2. Loss Calculation:

   - Compute the loss at each time step *t* using the chosen loss function (e.g., Mean Squared Error).

BPTT Steps (2/4)

3. Backward Pass:

- Starting from the final time step and moving backward through the sequence, compute the gradients of the loss with respect to the network parameters (weights and biases) at each time step *t*.
- Accumulate these gradients across all time steps in the sequence.

BPTT Steps (3/4)

4. Parameter Update:

- Once the gradients have been accumulated across all time steps, update the network parameters using a learning rate α.
- The updates are performed using the following formulas:

$$W \leftarrow W - \alpha \frac{\partial Loss}{\partial W} \text{ and } b \leftarrow b - \alpha \frac{\partial Loss}{\partial b}$$

- Here, W represents the weights and b represents the biases of the network.

BPTT Steps (4/4)

- This process ensures that the learning from the entire sequence is taken into account when updating the parameters, which is crucial for training RNNs to learn temporal dependencies in sequence data.
- The parameter updates are done after the gradients have been computed and accumulated across the entire sequence, not at each individual time step.

Example – BPTT (1/4)

- Problem Setup:
  - A simplified demonstration using a sequence (1,2,3,4).
  - Suppose we have a simple RNN with one hidden unit and the goal is to predict the next number in the sequence based on the current number.

$h_t = W_h h_{t-1} + W_h x_t + b_h$ and $y_t = W_y h_t + b_y$

Example – BPTT (2/4)

- Forward Pass:
  - We unroll the RNN through the sequence (1,2,3,4), calculating the hidden state and output at each time step *t*:

    $h_t = \sigma(W_h h_{t-1} + W_h x_t + b_h)$ and $y_t = W_y h_t + b_y$
- Loss Calculation:
  - Suppose we use Mean Squared Error (MSE) as the loss function.

    $Loss_t = (y_t - x_{t+1})^2$

Example – BPTT (3/4)

$$\frac{\partial Loss_t}{\partial y_t} = 2(y_t - x_{t+1})$$

$$\frac{\partial Loss_t}{\partial W_y} = \frac{\partial Loss_t}{\partial y_t} \cdot \frac{\partial y_t}{\partial W_y}$$

$$\frac{\partial Loss_t}{\partial W_h} = \frac{\partial Loss_t}{\partial y_t} \cdot \frac{\partial y_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial W_h}$$

- BPTT Pass:

  Now we will propagate the gradients of the loss back through the network to update the weights and biases.

  1) Compute Gradients of Loss with respect to Network Outputs.
  2) Compute Gradients of Loss with respect to Network Weights and Biases.
  3) Accumulate Gradient: Accumulate the gradients across all time steps in the sequence.
  4) Update Weights and Biases.

Example – BPTT (4/4)

- Repeat:

  Repeat the forward pass, loss calculation, BPTT, and parameter update steps for a number of epochs until the loss converges to a minimum value.

Backpropagation vs. BPTT (1/4)

- Sequence Handling
  - Backpropagation: Does not inherently handle sequences or temporal dependencies as it processes the data in a single forward pass followed by a backward pass.
  - BPTT: Explicitly designed to handle sequences and temporal dependencies by unrolling the network through time and applying backpropagation on this unfolded structure.

Backpropagation vs. BPTT (2/4)

- Computational Complexity
  - Backpropagation: Generally more computationally efficient as it only requires a single forward and backward pass through the network.
  - BPTT: Can be computationally intensive and memory-demanding due to the necessity to unroll the network through time, which effectively increases the depth of the network with each time step.

Backpropagation vs. BPTT (3/4)

- Error Propagation
  - Backpropagation: Error gradients are propagated backward from the output layer to the input layer in a straightforward manner.
  - BPTT: Error gradients are propagated backward both through layers and through time, which can lead to challenges like vanishing or exploding gradients, especially over long sequences.

Backpropagation vs. BPTT (4/4)

- Gradient Accumulation
  - Backpropagation: Computes gradients and updates parameters based on a single pass of data.
  - BPTT: Accumulates gradients over all time steps in a sequence before updating the parameters, which helps in learning coherent patterns across the entire sequence.

Comparisons of RNN, LSTM, and GRU (1/7)

- Architecture Complexity:
  - RNN: Simplest architecture among the three with a basic loop for passing information from one step in the sequence to the next.
  - LSTM: More complex architecture with a memory cell and three gating mechanisms (input, forget, and output gates) to control the flow of information.
  - GRU: Intermediate complexity with a simplified gating mechanism having two gates (reset and update gates).

Comparisons of RNN, LSTM, and GRU (2/7)

- Ability to Handle Long-term Dependencies:
  - RNN: Struggles with long-term dependencies due to the vanishing gradient problem.
  - LSTM: Designed specifically to handle long-term dependencies by maintaining a separate memory cell.
  - GRU: Also capable of handling long-term dependencies, albeit with a simpler gating mechanism.

Comparisons of RNN, LSTM, and GRU (3/7)

- Parameter Count:
  - RNN: Fewest parameters, making it computationally the most efficient but often the least expressive.
  - LSTM: Highest number of parameters due to its three gates and separate memory cell.
  - GRU: Fewer parameters than LSTM due to its simplified gating mechanism, providing a balance between computational efficiency and expressiveness.

Comparisons of RNN, LSTM, and GRU (4/7)

- Training Efficiency:
  - RNN: Fastest to train due to its simplicity but often yields the poorest performance on tasks requiring the modeling of long-term dependencies.
  - LSTM: Slower to train due to its complexity and higher parameter count.
  - GRU: Generally faster to train than LSTM while often achieving comparable performance.

Comparisons of RNN, LSTM, and GRU (5/7)

- Performance:
  - RNN: Typically underperforms LSTMs and GRUs on tasks with long-term dependencies.
  - LSTM: Often yields strong performance on a wide range of tasks, especially those involving long-term dependencies.
  - GRU: Usually performs comparably to LSTM on many tasks while being more computationally efficient.

Comparisons of RNN, LSTM, and GRU (6/7)

- Use Cases:
  - RNN: Suitable for simple tasks or when computational resources are limited.
  - LSTM: Preferred for more complex tasks involving long-term dependencies, such as machine translation or speech recognition.
  - GRU: A good choice for tasks requiring the modeling of long-term dependencies but with constrained computational resources.

Comparisons of RNN, LSTM, and GRU (7/7)

- The choice between RNN, LSTM, and GRU depends on the specific task, the complexity of the data, and the computational resources available.
- LSTMs and GRUs are often favored for more complex tasks due to their ability to model long-term dependencies, with the choice between them depending on the trade-off between computational efficiency and architectural complexity.

Recent RNN Architectures (1/2)

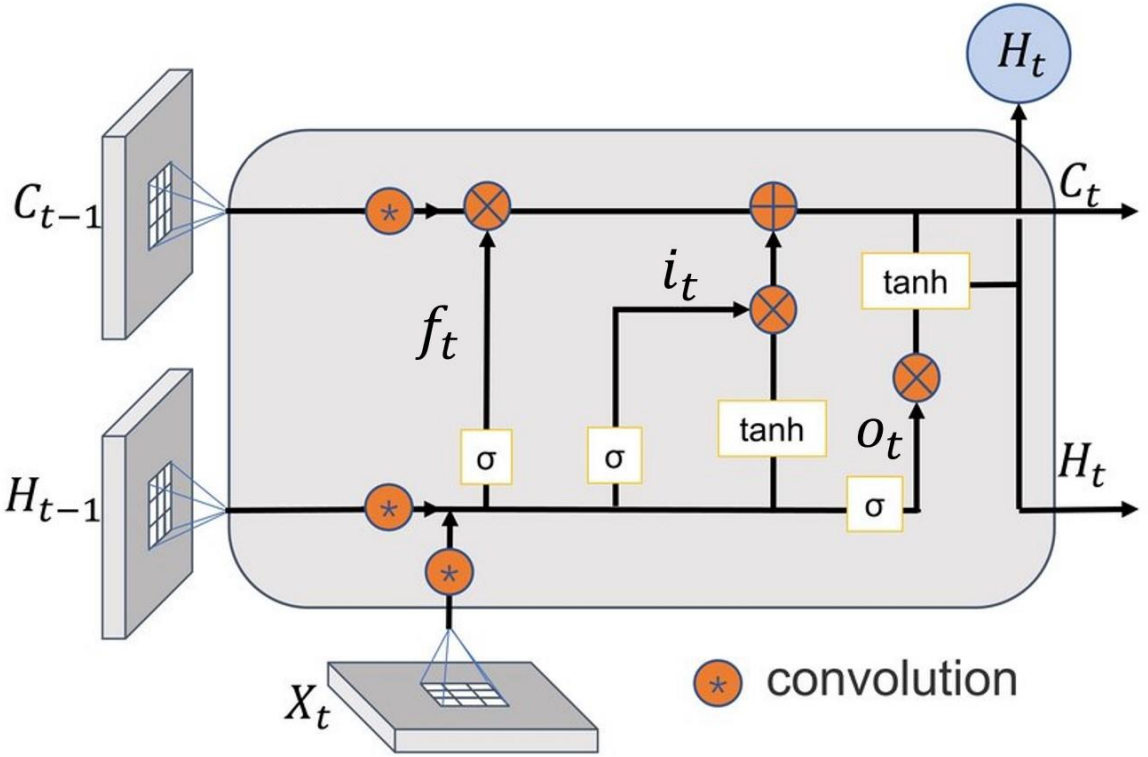
- Training-free Neural Architecture Search (NAS):
  - NAS is an automated method for finding the best neural network architecture.
  - Recent studies have explored training-free NAS metrics for RNNs and BERT-based transformer architectures, specifically for language modeling tasks.
  - A new metric called hidden covariance has been developed to predict the trained performance of an RNN architecture without the need for training, which could potentially speed up the architecture search process significantly.

Recent RNN Architectures (2/2)

- Advanced Gated RNN Architectures:
  - Gated RNN architectures, which incorporate internal loops regulated by gates, have been identified as the most advanced form of RNNs.
  - These architectures enhance the capability to maintain input-to-state stability (ISS) and Incremental Input-to-State Stability, which are crucial for effective training and reliable performance of RNNs in various applications.

ConvLSTM

- The Convolutional Long Short-Term Memory (ConvLSTM) is an extension of the traditional LSTM to better handle spatial-temporal data.
- The ConvLSTM successfully blends the strengths of CNNs and LSTMs to process spatial-temporal data effectively, leading to improved performance in relevant applications.



Source: Shi, X., Chen, Z., Wang, H., Yeung, D. Y., Wong, W. K., & Woo, W. C. (2015). Convolutional LSTM network: A machine learning approach for precipitation nowcasting. Advances in neural information processing systems, 28.

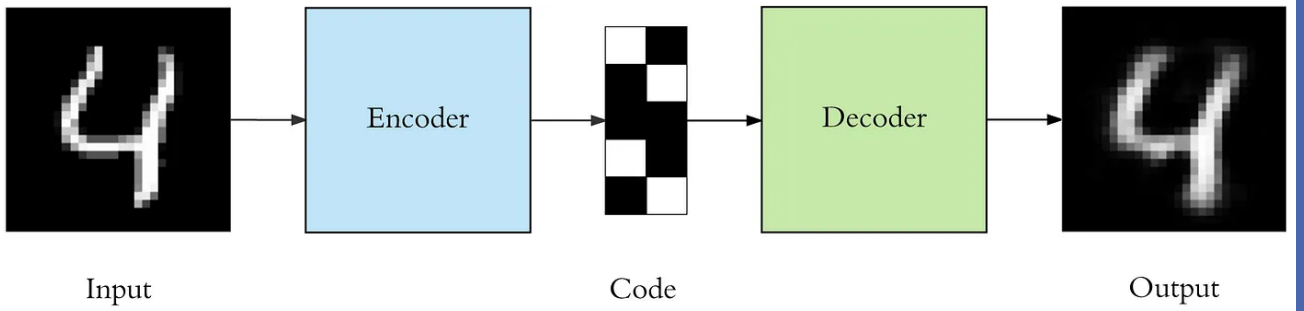
Autoencoders

- Introduction
- Core Concepts

Introduction (1/2)

- Autoencoders are a specific type of neural network used in unsupervised machine learning, where the objective is to learn a representation (encoding) for a set of data, typically for the purpose of dimensionality reduction, noise reduction, or generative modeling.
- The primary goal of an autoencoder is to minimize the reconstruction error, i.e., the difference between the original input data and the reconstructed data.

Introduction (2/2)



- Autoencoders consist of two main parts:
  - Encoder: This part compresses the input data into a lower-dimensional representation. It's essentially learning a function that maps the input data to a point in a lower-dimensional space.
  - Decoder: This part attempts to reconstruct the original data from the lower-dimensional representation obtained from the encoder. It learns a function that maps the lower-dimensional representation back to the original input space.

Source: https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

Importance in Various Domains (1/3)

- Dimensionality Reduction:
  - Autoencoders are highly effective for dimensionality reduction tasks.
  - In some scenarios, they outperform Principal Component Analysis (PCA) because unlike PCA, which only learns linear transformations of the features, autoencoders can learn non-linear transformations thanks to their neural network-based architecture.

Importance in Various Domains (2/3)

- Noise Reduction:
  - They can be employed to denoise data; the autoencoder is trained on noisy data while being supervised by the clean data.
  - Through this process, the autoencoder learns to recover the original, noise-free data, thereby acting as a denoising tool.

Importance in Various Domains (3/3)

- Generative Modeling:
  - Autoencoders, particularly Variational Autoencoders (VAEs), are used in generative modeling.
  - They can learn to generate new data that's similar to the training data, which is an essential aspect of generative modeling.

Key Properties

- Unsupervised Learning:
  - Autoencoders are trained in an unsupervised manner as they don't require labeled data. They learn to reconstruct the input data based on inherent structures and patterns in the data.
- Data-specific:
  - The representations learned by autoencoders are data-specific, meaning they might not generalize well to unseen data or different tasks.
- Lossy Compression:
  - The process of encoding and decoding is a form of lossy compression where some information might be lost during the compression.

Historical Context

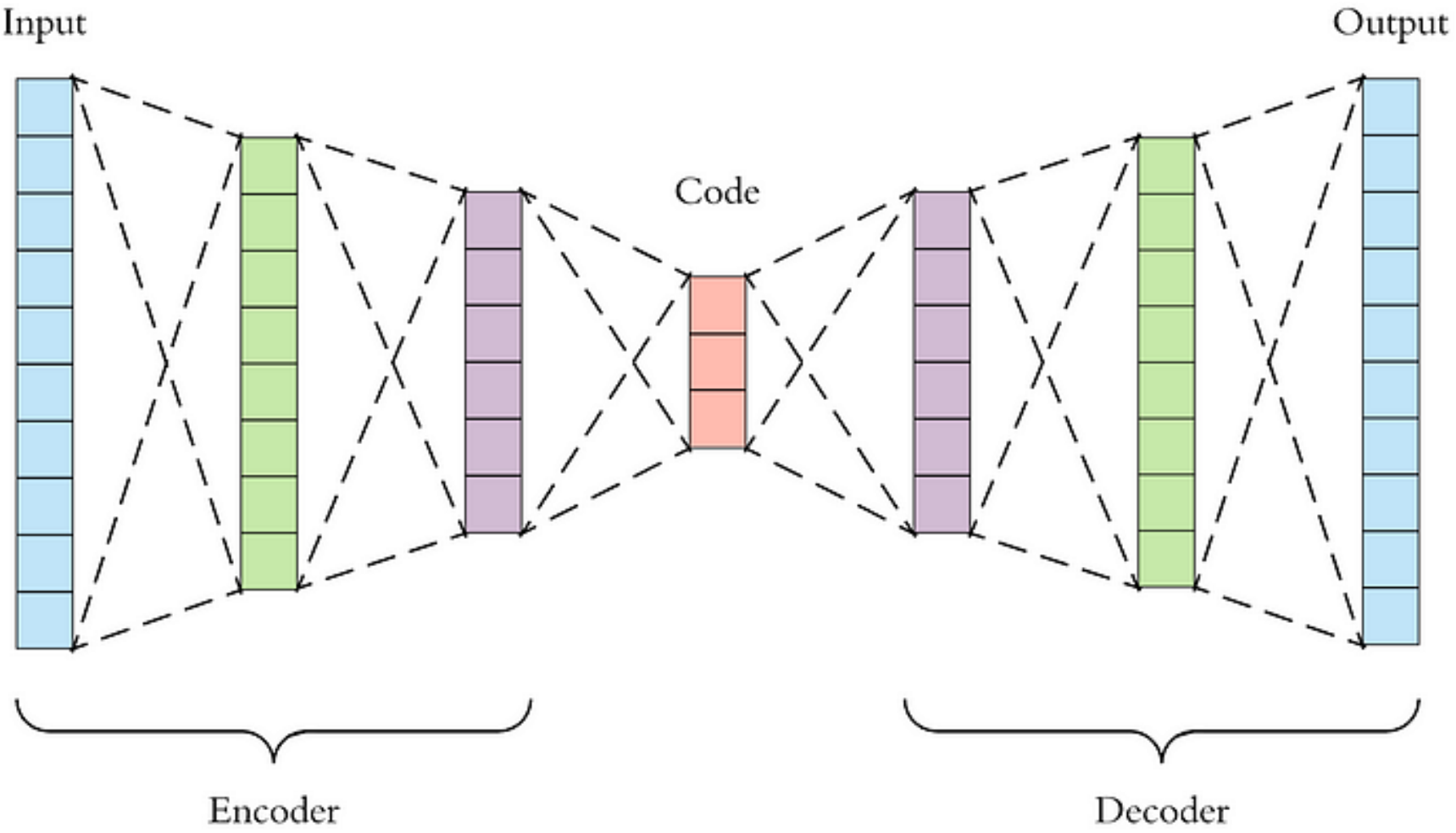
- The concept of autoencoders has been explored since the 1980s, initially for dimensionality reduction and feature learning.
- With the advent of deep learning in the mid-2000s, autoencoders gained prominence and evolved with new variants like Variational Autoencoders, Convolutional Autoencoders, and Sequence-to-Sequence Autoencoders, extending their applications to a broader range of tasks and data types.

Why Are Autoencoders Important (1/2)

- **Unsupervised Learning:** Autoencoders are crucial for learning from unlabeled data, which is abundant compared to labeled data. They help in extracting valuable insights and representations from such data without manual labeling.
- **Dimensionality Reduction:** They are effective in reducing the dimensionality of data, which is critical for handling high-dimensional data in various machine learning tasks.

Why Are Autoencoders Important (2/2)

- **Noise Reduction:** Autoencoders can recover original, noise-free data from noisy data, making them essential for denoising tasks.
- **Generative Modeling:** Variants like Variational Autoencoders (VAEs) are used for generative modeling, helping in the generation of new data samples similar to the training data.
- **Feature Learning:** They can learn meaningful representations of data, aiding in feature engineering for other machine learning tasks.
- **Anomaly Detection:** By identifying instances that deviate significantly from the learned representation, autoencoders can be utilized for anomaly detection.



Source: https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

How Does an Autoencoder Work (1/3)

1. Training:
   - The autoencoder is trained to minimize the reconstruction error between the original input data and the reconstructed data.
   - It learns to compress the data in the encoding phase and to reconstruct the original data in the decoding phase.
2. Encoding:
   - The encoder processes the input data and compresses it into a lower-dimensional representation.
   - This representation is often referred to as the “bottleneck” as it's a compressed knowledge of the input data.

How Does an Autoencoder Work (2/3)

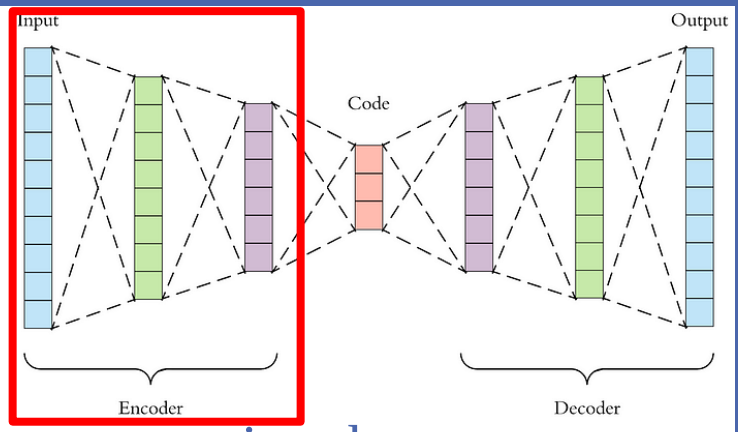
3. Decoding:
   - The decoder takes the encoded representation and attempts to recreate the original input data.
   - It learns to undo the compression, reconstructing the data as closely as possible to the original input.
4. Loss Calculation:
   - A loss function, often Mean Squared Error (MSE) or Binary Cross-Entropy, measures the difference between the original data and the reconstructed data.
   - The optimizer minimizes this loss by adjusting the weights in the network using backpropagation.

How Does an Autoencoder Work (3/3)

5. Iterative Learning:

- The process of encoding, decoding, and loss minimization is iteratively performed until the network achieves a low reconstruction error, or until a specified number of epochs is reached.

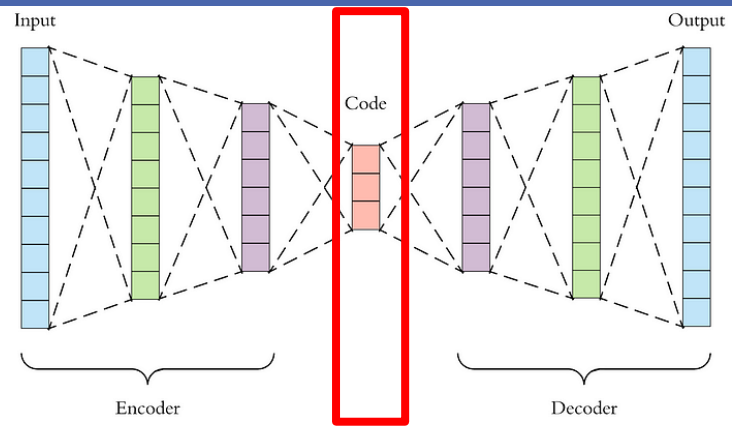
Encoder



- The encoder is the first half of the autoencoder, responsible for compressing the input data into a lower-dimensional representation.
- It consists of one or more layers of neurons in a neural network.
  - Input Layer: The input layer receives the input data. The number of nodes in this layer equals the number of features in the input data.
  - Hidden Layers: Following the input layer are one or more hidden layers that process the input data. Each successive layer may have fewer nodes, thereby reducing the dimensionality of the data progressively.
  - Activation Functions: Activation functions introduce non-linearity into the system, enabling the encoder to learn from the error backpropagation process.

Source: https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

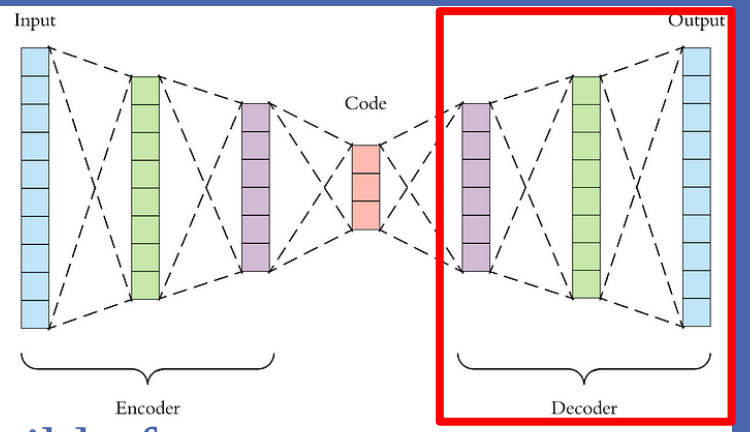
Code (Bottleneck)



- The code or bottleneck is the point of compression in the autoencoder where the dimensionality of the data is at its lowest.
- It holds the compressed knowledge of the input data.
  - Representation: The code is a lower-dimensional representation of the input data, and the number of nodes in this layer determines the level of compression.
  - Latent Space: This layer represents the latent space where the essential characteristics of the data are retained.

Source: https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

Decoder

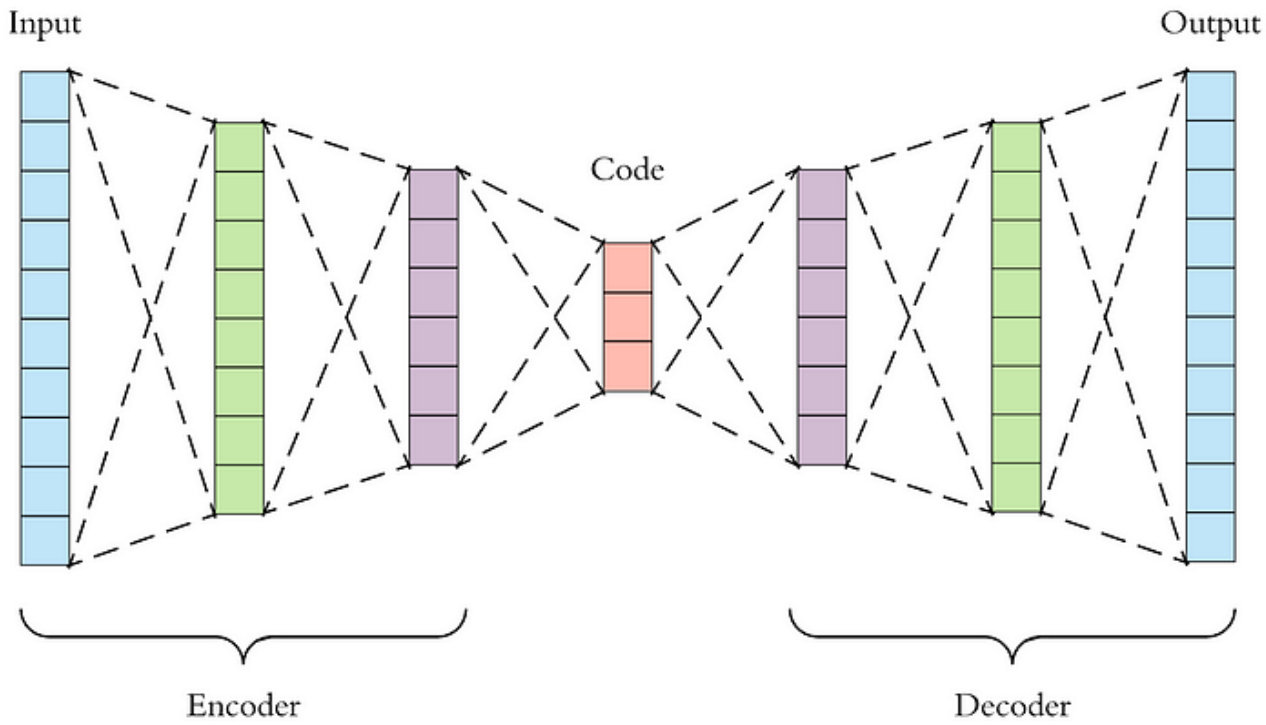


- The decoder is the second half of the autoencoder, responsible for reconstructing the original input data from the lower-dimensional representation obtained from the encoder.
  - Hidden Layers: Like the encoder, the decoder has hidden layers, but each successive layer in the decoder may have more nodes, expanding the data back towards its original dimensionality.
  - Output Layer: The final layer in the decoder is the output layer. The number of nodes in the output layer equals the number of features in the original input data, and it produces the reconstructed data.
  - Activation Functions: Activation functions are used to introduce non-linearity, similar to the encoder.

Source: https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

Working Mechanism

1. Forward Pass
   - Encoding
   - Decoding
2. Reconstruction Loss
   - Loss Function
   - Error Calculation
3. Backward Pass
   - Gradient Calculation
   - Weight Update



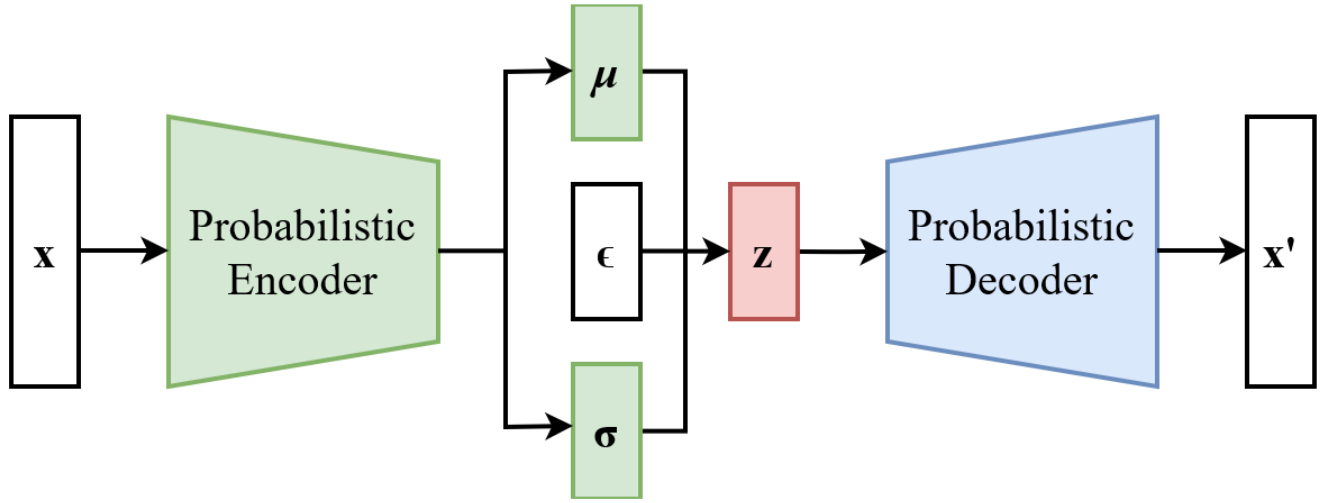
Source: https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

Limitations of Basic Autoencoders

- Lack of Generative Capability:
  Traditional autoencoders are not designed to generate new data points that were not in the training set. They are primarily geared towards reconstruction.
- Inability to Handle Noise:
  They are not robust to noise in the input data. Noise can significantly degrade the quality of the reconstructed output.
- Sparse Representations:
  Basic autoencoders do not naturally enforce sparsity in the learned representations, which could be desirable for certain applications like feature selection or dimensionality reduction.
- Fixed Dimensionality:
  The dimensionality of the bottleneck layer is fixed and predetermined, which could be a limitation if the optimal dimensionality is unknown a priori.
- Lack of Interpretability:
  The latent space learned by basic autoencoders may not be interpretable, which might be a disadvantage in scenarios requiring understanding and interpretation of the learned representations.

Variational Autoencoders (VAEs)

- Variational Autoencoders (VAEs) are a type of probabilistic autoencoder that not only learn to compress and decompress the data but also generate new, similar data.
- Probabilistic Framework
- Reparameterization Trick
- KL-Divergence Regularization
- Generative Capability



Source: https://en.wikipedia.org/wiki/Variational_autoencoder

Sparse Autoencoders

- Sparse Autoencoders are designed to learn sparse representations of the input data, which can be beneficial for various machine learning tasks.
- They are useful in scenarios where data is high-dimensional and redundant, and where discovering a compact, sparse representation can lead to better performance in downstream tasks.
- A sparsity constraint is imposed on the activations of the neurons in the hidden layers, often through a regularization term in the loss function that penalizes non-zero activations.
- By learning sparse representations, sparse autoencoders can capture the most salient features of the data while ignoring irrelevant details, which can be useful for feature extraction and dimensionality reduction.

Source: https://en.wikipedia.org/wiki/Variational_autoencoder

Denoising Autoencoders

- Denoising Autoencoders are designed to learn to remove noise from data, and are trained by adding noise to the input data while using the clean data as the target for reconstruction.
- They are valuable in scenarios where data is naturally noisy or corrupted, and where recovering clean, denoised data is the objective.
- Noise Injection: Noise is added to the input data either by corrupting the data values or by masking some of the data values.
- By learning to recover the original, clean data from the noisy input, denoising autoencoders learn a robust representation of the data that's resilient to noise.

Source: https://en.wikipedia.org/wiki/Variational_autoencoder

Example – VAEs

- Generative Modeling:
  - If you're working on a project to generate new data that resembles a given dataset, say, generating new faces based on a dataset of celebrity faces, VAEs would be a suitable choice due to their generative capabilities.
- Semi-Supervised Learning:
  - In a scenario with a small amount of labeled data and a large amount of unlabeled data, VAEs can be used for semi-supervised learning to leverage the unlabeled data to improve performance on a classification task.
- Anomaly Detection:
  - If you're tasked with identifying fraudulent transactions in a dataset, VAEs can model the normal transactions and thereby identify the outliers or anomalies.

Source: https://en.wikipedia.org/wiki/Variational_autoencoder

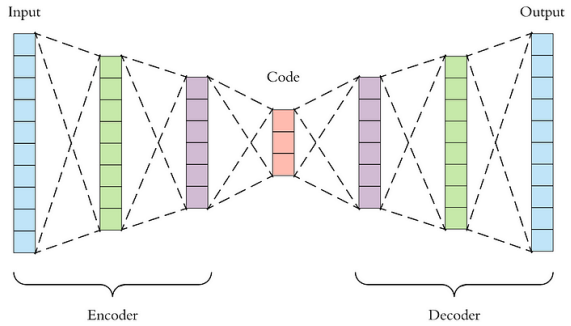
Example – Sparse Autoencoders

- Feature Selection or Extraction:
  - If you're dealing with high-dimensional data, say gene expression data with thousands of features, and you want to identify a smaller set of important features, a Sparse Autoencoder can help in extracting a sparse set of meaningful features.
- Dimensionality Reduction:
  - In a scenario where visualization of high-dimensional data is required, Sparse Autoencoders can be utilized to reduce the dimensions of the data to a more manageable size while retaining the essential structure of the data.
- Denoising:
  - Despite not being as robust as Denoising Autoencoders, Sparse Autoencoders can also be used for denoising purposes by learning a sparse representation that captures the underlying structure of the data, ignoring the noise.

Example – Denoising Autoencoders

- Noise Reduction:
  - If you're working on image restoration, say removing noise from scanned images or photographs, Denoising Autoencoders can be trained to map noisy images to clean images.
- Robust Feature Learning:
  - In a scenario where the input data for a classification task is known to be noisy or corrupted, training a Denoising Autoencoder on the data first can help in learning robust features which can then be used for classification.
- Pre-training for Deep Networks:
  - If you have a large and potentially noisy dataset for a supervised learning task, a Denoising Autoencoder can be used for pre-training layers of a deep network which can then be fine-tuned for the specific task.

Guideline for Autoencoders (1/4)



1. Determine the Objective
   - Understand the primary goal (e.g., dimensionality reduction, denoising, etc.) to choose an appropriate type of autoencoder.
2. Input and Output Dimensions
   - The number of neurons in the input and output layers should match the dimensionality of your data.
3. Number of Layers:
   - Shallow Autoencoders: For simpler tasks or smaller datasets, a single hidden layer for the encoder and the decoder may suffice.
   - Deep Autoencoders: For more complex tasks or larger datasets, consider using multiple hidden layers (e.g., 3 to 5 layers in both the encoder and decoder).

Guideline for Autoencoders (2/4)

4. Number of Neurons:
   - Encoder: Typically, each subsequent layer in the encoder has fewer neurons to achieve data compression. A common practice is to halve the number of neurons at each layer, though this rate can be adjusted based on the desired level of compression and the complexity of the data.
   - Latent Layer: The number of neurons in the latent layer (bottleneck) depends on the desired level of compression. There's no hard and fast rule, and it may require experimentation to find an optimal size.
   - Decoder: Mirror the encoder's architecture in the decoder, doubling the number of neurons at each layer until reaching the original data dimensionality.

Guideline for Autoencoders (3/4)

5. Activation Functions:
   - ReLU (Rectified Linear Unit) is commonly used in hidden layers due to its ability to handle vanishing gradient issues and its computational efficiency.
   - For the output layer, use a sigmoid activation function if your data is normalized between 0 and 1, or a linear activation function for unbounded data.
6. Regularization:
   - If you’re building a Sparse Autoencoder, include a sparsity regularization term in your loss function, and experiment with the sparsity parameter.

Guideline for Autoencoders (4/4)

7. Initialization:
   - Use techniques like He initialization or Xavier initialization to set the initial weights of the layers, which can help in faster convergence.
8. Batch Size and Learning Rate:
   - Select an appropriate batch size and learning rate for training. Smaller batch sizes and learning rates can lead to more stable convergence but may require more training epochs.
9. Experimentation and Evaluation:
   - Experiment with different architectures and evaluate the model's performance on a validation dataset to find an optimal architecture.
   - Visualization tools can help to analyze the learned representations in the latent space and the quality of the reconstruction.

Hand-on

- **Basic Autoencoder**

Basic Autoencoder (1/3)

1. Import Necessary Libraries

```
import tensorflow as tf
from tensorflow.keras import layers, models
import numpy as np
import matplotlib.pyplot as plt
```

2. Load and Preprocess the Data

```
# Load the MNIST dataset
(x_train, _), (x_test, _) = tf.keras.datasets.mnist.load_data()

# Normalize the data to [0, 1] range
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.

# Flatten the images as the autoencoder will be fully connected
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
```

Basic Autoencoder (2/3)

3. Define the Autoencoder Architecture

```
encoding_dim = 32  # Size of the encoded representations

# Define the encoder
input_img = tf.keras.Input(shape=(784,))
encoded = layers.Dense(encoding_dim, activation='relu')(input_img)

# Define the decoder
decoded = layers.Dense(784, activation='sigmoid')(encoded)

# Combine the encoder and the decoder into an autoencoder model
autoencoder = models.Model(input_img, decoded)
```

4. Compile the Model

```
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
```

Basic Autoencoder (3/3)

5. Train the Model

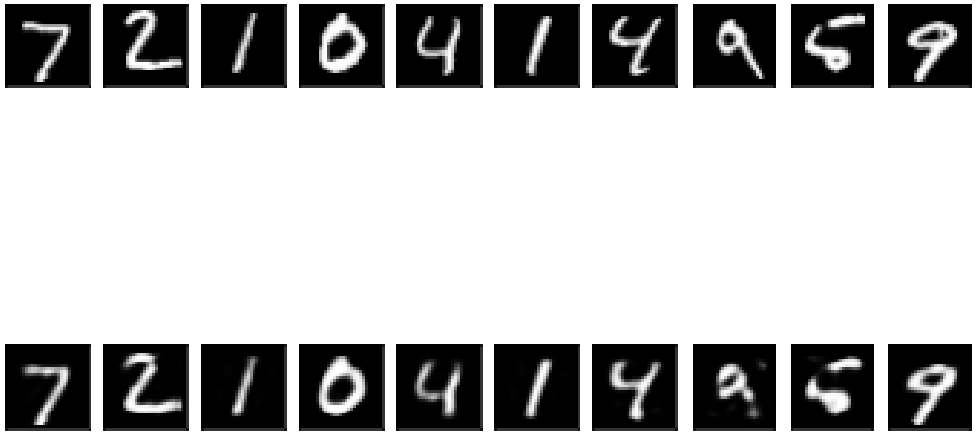
```
history = autoencoder.fit(x_train, x_train,
                          epochs=50,
                          batch_size=256,
                          shuffle=True,
                          validation_data=(x_test, x_test))
```

5. Evaluate the Model

```
# Obtain the reconstructed images
decoded_imgs = autoencoder.predict(x_test)

# Plot original and reconstructed images
n = 10
for i in range(n):
    # Display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
```



Thank you