CE6146
Introduction to Deep Learning
Generative Adversarial Networks

Chia-Ru Chung

Department of Computer Science and Information Engineering

National Central University

2023/11/2

20231026 Exercise

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

20231026 Exercise #6 (1/3)

- What is 'zero-padding' used for in CNNs?

  (A) To preserve the spatial dimensions of the input (B) To prevent overfitting

  (C) To speed up computation (D) To increase the depth of the feature map

- Zero-padding is the technique of adding zeros around the border of an input image or feature map.
- This is particularly useful when using convolutional layers in a neural network because convolution operations, especially with strides larger than one, tend to reduce the spatial dimensions of the input.
- Padding the input with zeros helps to preserve its dimensions through these operations.
- The spatial dimensions of the output feature map after a convolution operation are determined by the following formula:

$$Output\ Size = \left\lfloor \frac{Input\ Size - Kernal\ Size + 2 \times Padding}{Stride} \right\rfloor + 1$$

- When Padding is zero, the output size gets reduced.
- By adding Padding, the spatial reduction can be counteracted, thus preserving the dimensions.

20231026 Exercise #6 (2/3)

- Example (Stride 1): $\text{Input} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 2 & 1 \\ 1 & 2 & 5 & 2 & 1 \\ 1 & 2 & 2 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}, \text{Kernel} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

	No Padding	**Padding = 1**
Input	$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 2 & 1 \\ 1 & 2 & 5 & 2 & 1 \\ 1 & 2 & 2 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 2 & 2 & 2 & 1 & 0 \\ 0 & 1 & 2 & 5 & 2 & 1 & 0 \\ 0 & 1 & 2 & 2 & 2 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$
Output	$\begin{bmatrix} 8 & 5 & 4 \\ 5 & 9 & 5 \\ 4 & 5 & 8 \end{bmatrix}$	$\begin{bmatrix} 3 & 3 & 3 & 2 & 1 \\ 3 & 8 & 5 & 4 & 2 \\ 3 & 5 & 9 & 5 & 3 \\ 2 & 4 & 5 & 8 & 3 \\ 1 & 2 & 3 & 3 & 3 \end{bmatrix}$

20231026 Exercise #6 (3/3)

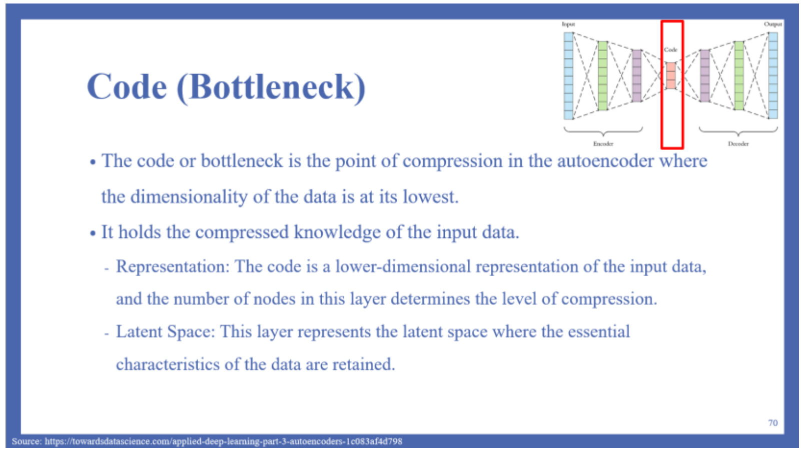
- Real-world example:
  1) **Image Segmentation in Medical Imaging:** In medical imaging, such as MRI or CT scans, the aim is often to segment different types of tissues or identify anomalies like tumors. In this scenario, every pixel in the output is significant and should correspond to a pixel in the input. Padding is essential to maintain the spatial dimensions through the network, ensuring that the output has a one-to-one mapping with the original image.
  2) **Facial Recognition:** In facial recognition systems used for secure access to devices or buildings. The subtle features of faces (like the distance between the eyes, shape of the nose, etc.) are important. Padding helps in maintaining these fine-grained details through the layers of the network.
  3) **Anomaly Detection in Manufacturing:** In manufacturing, CNNs can be used to automatically inspect parts on an assembly line for defects. The exact dimensions of the part images are often crucial for identifying minute defects, making padding a useful technique.

20231026 Exercise #7

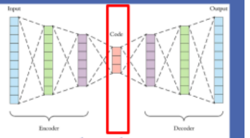
- What is the role of the bottleneck layer in an Autoencoder?

  (A) To increase the dimensionality of data
  (B) To reduce the dimensionality of data
  (C) To perform classification
  (D) To act as a database

- The bottleneck layer is the middle layer in an autoencoder and serves as a compressed representation of the input data.
- This layer has fewer neurons than the input layer, effectively reducing the dimensionality.
- Lecture 6 P.70



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.

70

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

20231026 Exercise #15

- Why might dilation NOT be suitable for fine-grained classification tasks?

  (A) Overlooks subtle differences between classes (B) Captures too small a context

  (C) Increases computational complexity (D) Reduces the depth of the network

- This question focuses on the potential drawbacks of using dilation in convolutional neural networks (CNNs) for fine-grained classification.
- Specifically, the question asks why dilation may be unsuitable for distinguishing highly similar categories on the basis of nuanced, often localized, features.
- Dilation in CNNs is used to increase the receptive field of filters.
- While this is useful for capturing global context, it can make the model overlook subtle features that are often crucial for fine-grained classification.
- The depth of the network is usually designed separately from whether dilation is used. You could have a deep or shallow network with or without dilation. Therefore, dilation itself doesn't inherently reduce the depth of the network.

20231026 Exercise #18

- What does a kernel size of three with a stride of two accomplish in a Convolutional layer?

  (A) Computes a weighted sum at every position

  (B) Computes a weighted sum at alternate positions

  (C) Computes a weighted sum at every third position

  (D) Computes a weighted sum at overlapping positions

- The term "weighted sum at alternate positions" in the context of a CNN refers to the sum obtained by performing the convolution operation at specific, non-adjacent positions across the input feature map.
- When the kernel slides across the input with a stride greater than 1, it skips over certain positions.
- The weighted sum is calculated only at those positions where the kernel lands, effectively at "alternate" positions compared to a stride of 1.

20231026 Exercise #20

- Which gating unit in an LSTM controls what information will be stored in the cell state?

  (A) Input gate (B) Forget gate

  (C) Memory gate (D) Output gate

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: $c_t = f_t \circ c_{t-1} + \boldsymbol{i_t \circ \tilde{c}_t}$

Outline

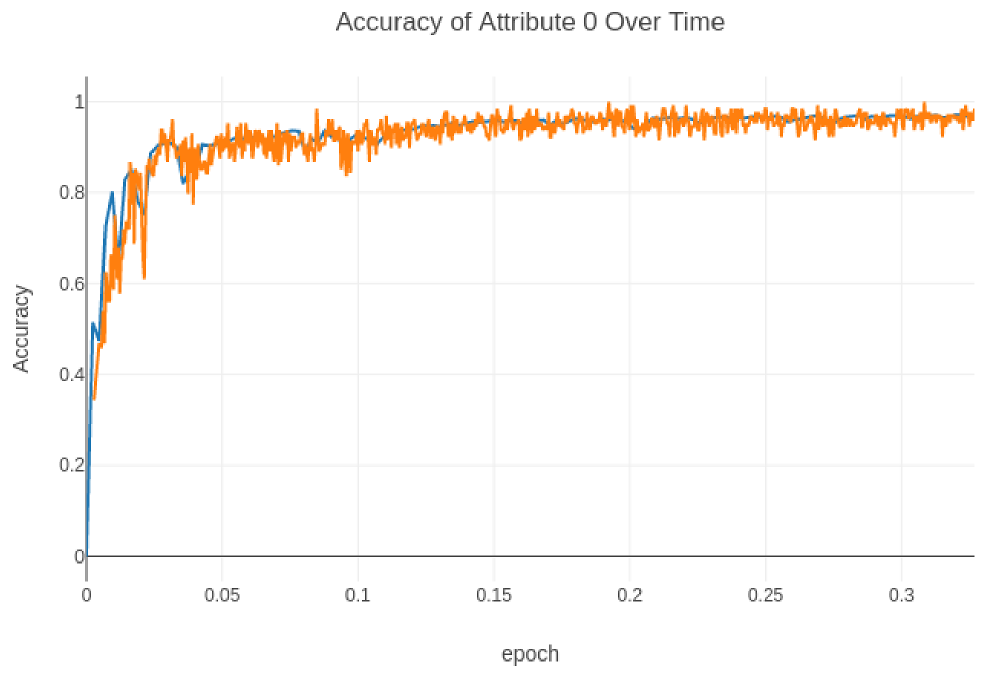
- Review
- Generative Adversarial Networks
- Hand-on

Review

- Learning Curve
- Long Short-Term Memory Networks
- Autoencoders

Learning Curve for Deep Learning (1/3)

- The term “learning curve” in the context of deep learning refers to a graphical representation that illustrates the performance of a machine learning model over time as it learns from the training data.
- A learning curve plots the value of the model's loss function for the training set against the same loss function evaluated on a validation set.



Source: https://www.baeldung.com/cs/learning-curve-ml

Learning Curve for Deep Learning (2/3)

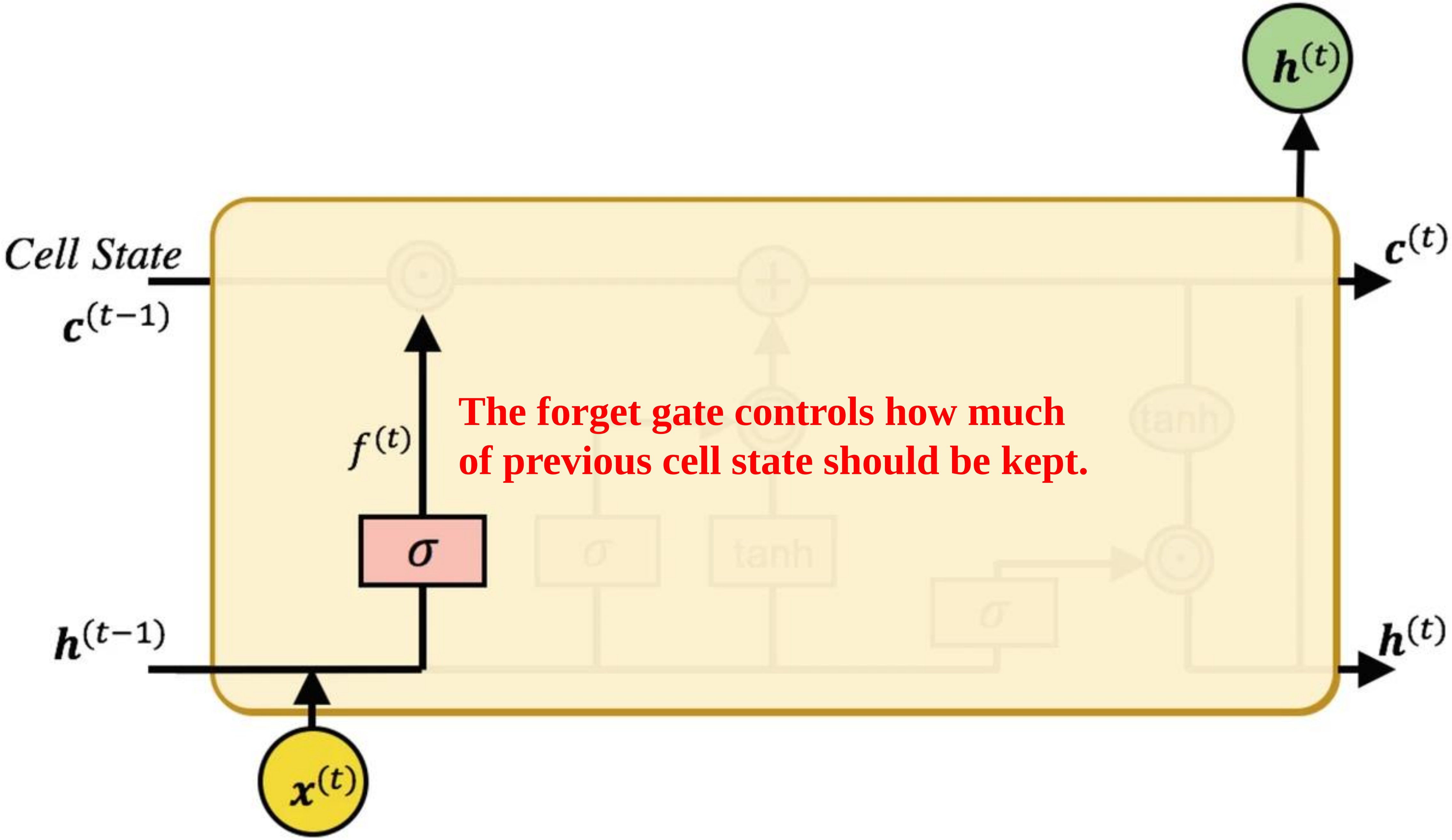
- In the context of deep learning, the x-axis of a learning curve can represent various metrics, but it most commonly represents either:
  - <u>Number of Training Samples</u>: This is useful for understanding how well the model generalizes to new data as the size of the training set increases.
  - <u>Number of Iterations or Epochs</u>: This is useful for understanding how the model's performance evolves over time during the training process.

Learning Curve for Deep Learning (3/3)

- In the context of deep learning, the y-axis of a learning curve can represent various metrics, but it most commonly represents either:
  - Loss Function: The most common metric plotted on the y-axis is the value of the loss function. The loss function quantifies how well the model's predictions match the actual data. A lower loss indicates better performance.
  - Accuracy: In classification problems, accuracy or some other performance metric like F1-score, precision, or recall can be used.
  - Error Rate: In some cases, the error rate might be plotted, which is essentially 1 - Accuracy.

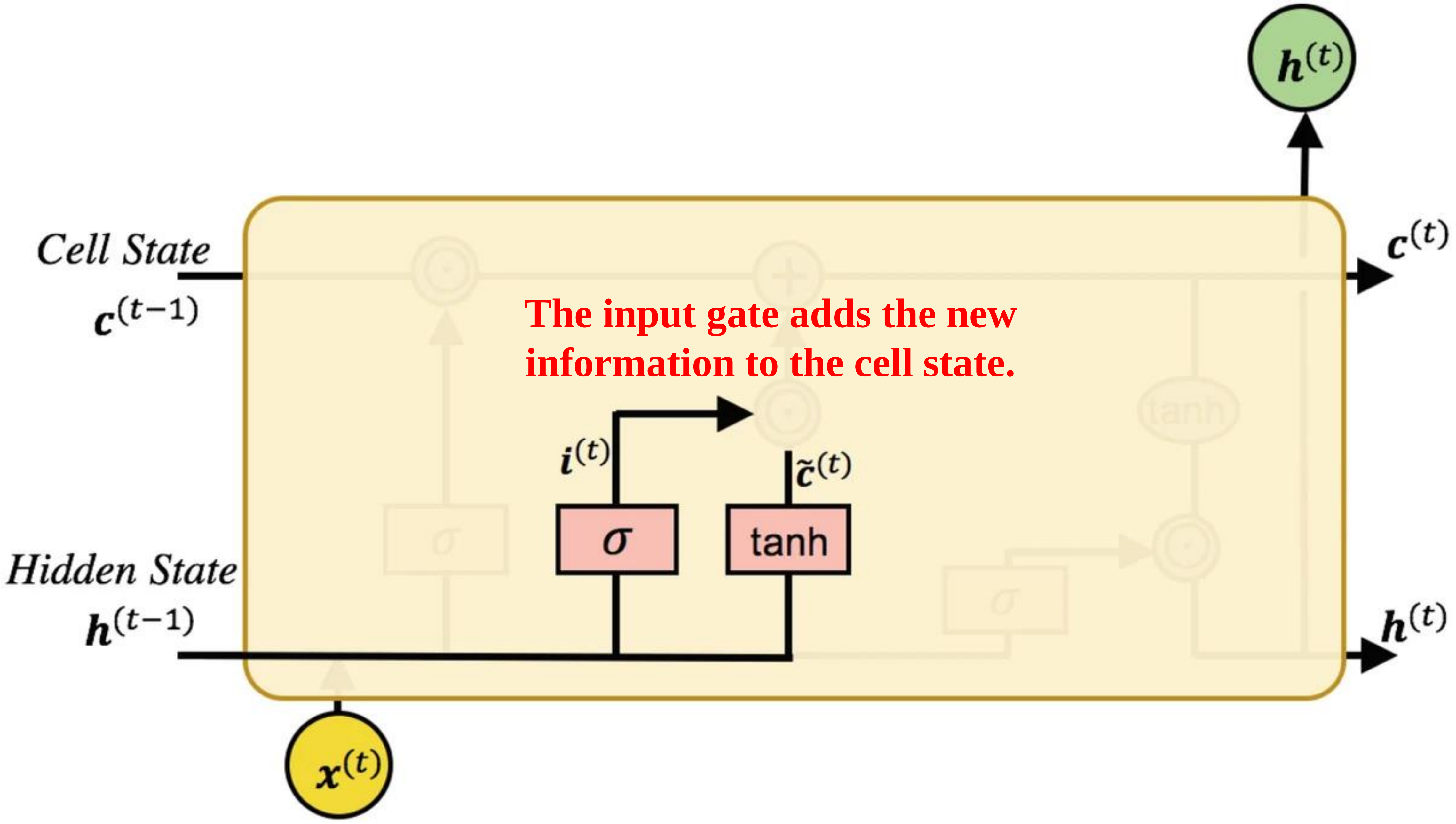
LSTM – Basic Architecture

- Input Gate: Determines how much of the incoming information should be stored in the memory cell.
- Forget Gate: Decides what information should be discarded from the memory cell.
- Cell State: Holds the network's long-term memory.
- Output Gate: Controls what information should be outputted based on the memory cell's current state.
- Hidden State: Holds the network's short-term memory.

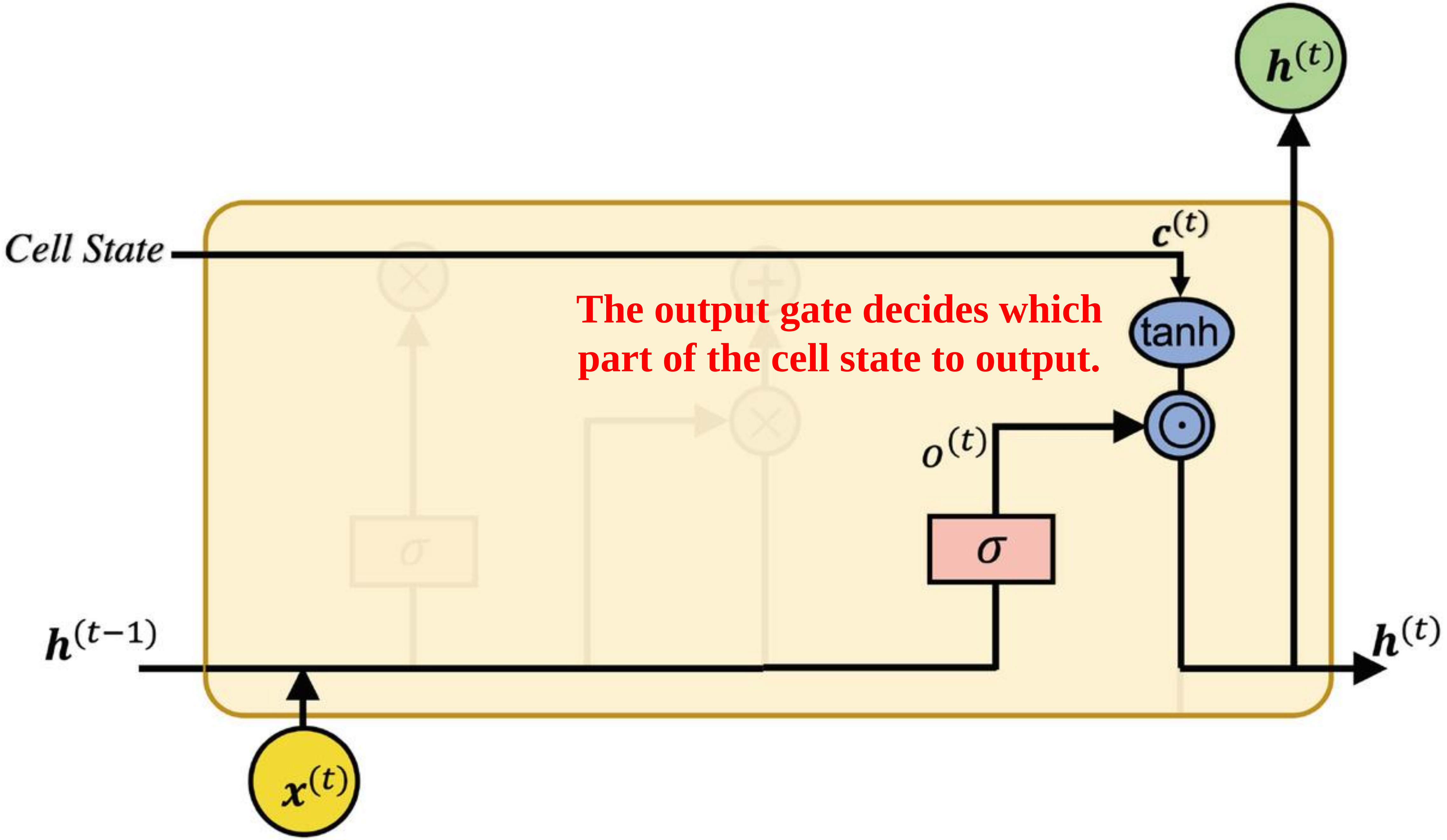


The forget gate controls how much of previous cell state should be kept.

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



The input gate adds the new information to the cell state.

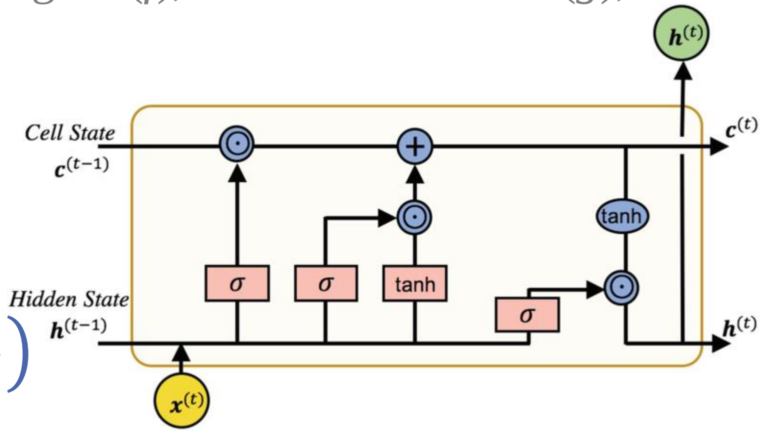


The output gate decides which part of the cell state to output.

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

Introduction

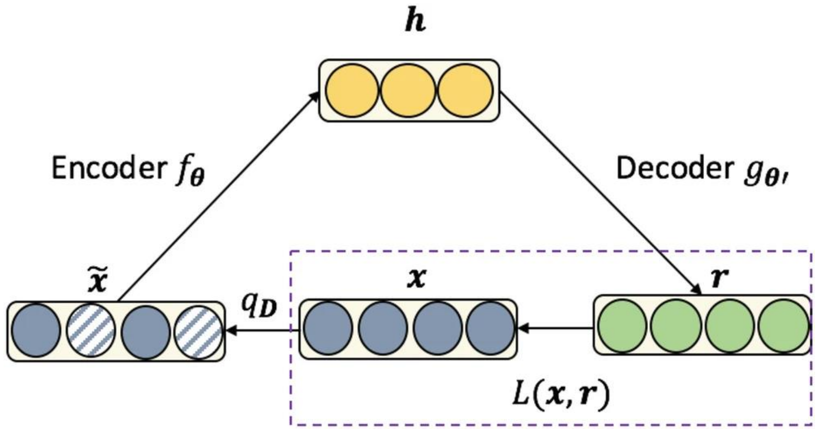
- Autoencoders are a specific type of neural network used in unsupervised machine learning, where the objective is to learn a representation (encoding) for a set of data, typically for the purpose of dimensionality reduction, noise reduction, or generative modeling.
- The primary goal of an autoencoder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

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.

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.
- The concept is straightforward: the autoencoder is trained on a dataset where the input is a noisy version of the data, while the target output is the clean, original data.



- A schematic representation of denoising autoencoders.
- The input data is stochastically corrupted via q_D to $\tilde{x}$.
- It then be mapped to h via encoder f_θ and tries to reconstruct x via decoder and produces reconstruction r.
- The reconstruction error is measured by $L(x, r)$

Denoising Autoencoders – How It Works

- Input Corruption:
  - During training, random noise is added to the input data to create a corrupted version. The level and type of noise can vary depending on the application.
- Encoder:
  - This part of the network compresses the corrupted input into a latent representation, attempting to capture the underlying structure of the data while ignoring the noise.
- Decoder:
  - The decoder takes the compressed, latent representation and reconstructs the data, aiming to produce the original, clean data.
- Loss Function:
  - The difference between the reconstructed output and the original, clean data is calculated, usually using Mean Squared Error (MSE) or a similar metric. The model then adjusts its weights to minimize this loss.

Denoising Autoencoders – Real-world Applications

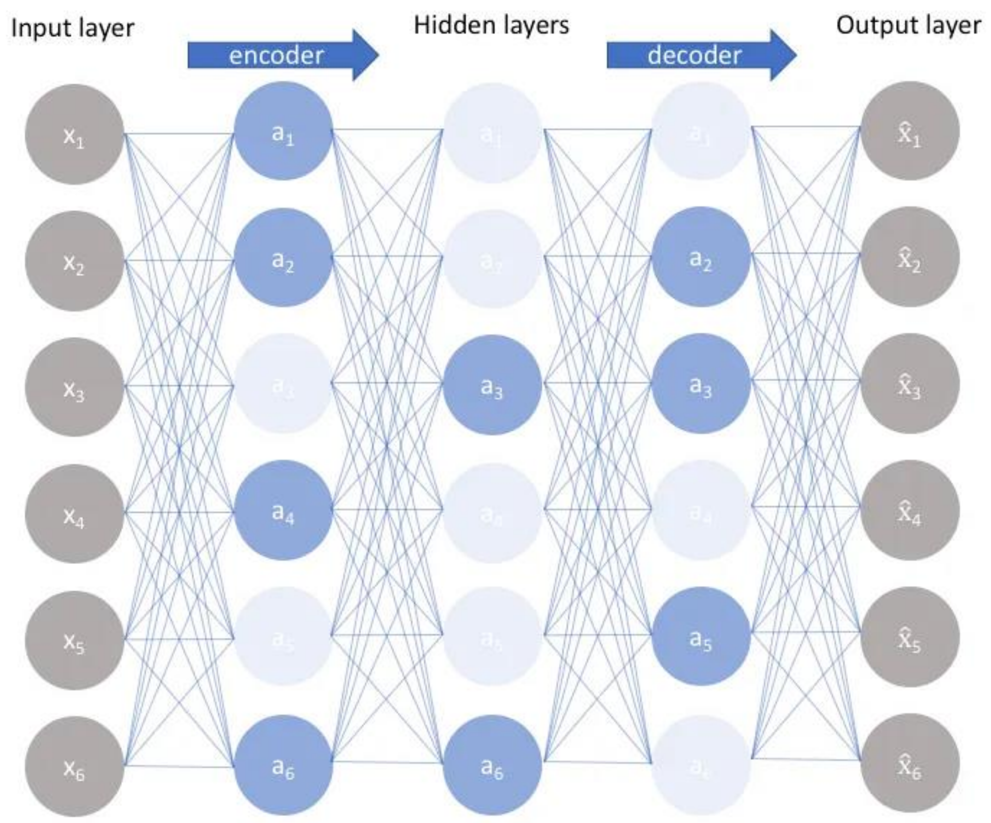
- Image Denoising:
  - Denoising autoencoders can be used to <u>clean up images</u> that have been corrupted by random noise, such as salt-and-pepper noise in old photographs or Gaussian noise in medical images.
- Signal Processing:
  - In telecommunications, denoising autoencoders can be used to <u>remove noise from audio signals to improve the clarity of voice calls</u>.

Denoising Autoencoders – Limitations

- Overfitting:
  - If not carefully regularized, denoising autoencoders can learn to simply memorize the training data, leading to poor generalization.
- Computational Cost:
  - Training a denoising autoencoder, especially on high-dimensional data, can be computationally expensive.
- Hyperparameter Sensitivity:
  - The performance of denoising autoencoders can be sensitive to the choice of hyperparameters such as the type and level of noise, the architecture of the network, and the regularization techniques used.

Sparse Autoencoders

- The sparsity constraint in autoencoders aims to ensure that only a small subset of neurons are activated, encouraging the model to learn more informative and compact representations.
- While it offers advantages like better feature learning and regularization, it also introduces challenges such as increased computational cost and hyperparameter sensitivity.



Source: https://medium.com/@syoya/what-happens-in-sparse-autencoder-b9a5a69da5c6

Sparse Autoencoders – How It Works

- Sparsity Constraint:
  - In addition to the usual loss term (often Mean Squared Error between the reconstructed and original data), a sparsity penalty term is added to the loss function. This term discourages the activation of multiple neurons at the same time.
- Encoder:
  - The encoder compresses the input into a latent representation, but the sparsity constraint ensures that only a subset of neurons in the hidden layer are activated for any given input.
- Decoder:
  - As in standard autoencoders, the decoder tries to reconstruct the original data from the latent representation.
- Optimization:
  - The objective is to minimize the combined loss, which includes both the reconstruction error and the sparsity penalty.

Sparse Autoencoders – Sparsity Constraint (1/2)

- The sparsity constraint in autoencoders is a form of regularization technique that aims to enforce that only a small subset of neurons in the hidden layers become active when presented with inputs.
- The main idea is to penalize the model if too many neurons are activated at the same time, encouraging it to learn a more compact and informative representation of the data.
- The mathematical formulation for sparse autoencoders is designed to ensure that only a small subset of neurons in the hidden layer are activated for a given input.
- A commonly used sparsity constraint is based on the Kullback-Leibler (KL) divergence.

Sparse Autoencoders – Sparsity Constraint (2/2)

- The sparsity penalty term Ω_j for each neuron j is often calculated using the KL divergence between a predefined sparsity parameter ρ and the observed average activation $\hat{\rho}_j$ of that neuron over the training batch.

$$\Omega_j = \rho \log\left(\frac{\rho}{\hat{\rho}_j}\right) + (1-\rho)\log\left(\frac{1-\rho}{1-\hat{\rho}_j}\right)$$

- Average activation $\hat{\rho}_j = \frac{1}{m}\sum_{i=1}^{m} a_j^{(i)}$, where $a_j^{(i)}$ is the activation of neuron j for the ith training sample, and m is the number of training samples.
- The overall loss function $Loss_{new} = Loss + \lambda \sum_j \Omega_j$, where λ is a hyperparameter that controls the weight of the sparsity penalty term.

Sparse Autoencoders – Real-world Applications

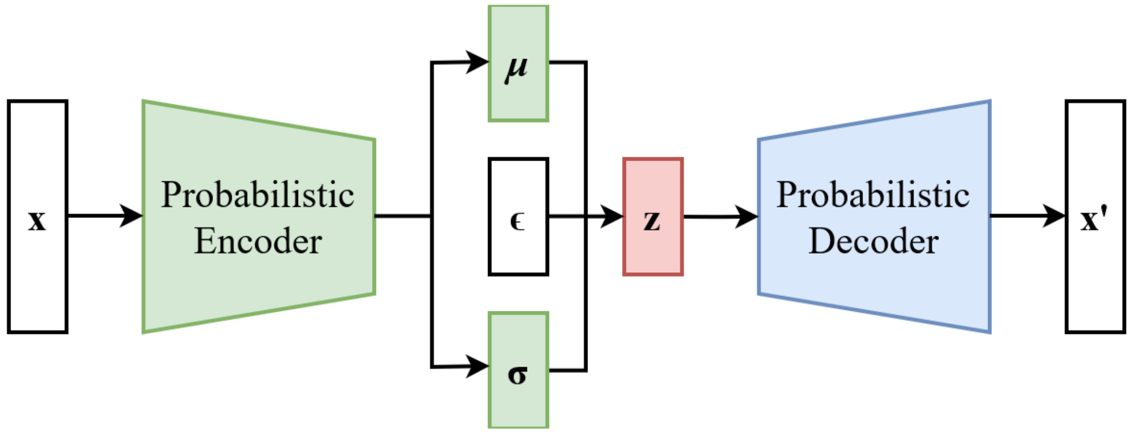
- Anomaly Detection:
  - Sparse autoencoders can be used to detect anomalies in datasets by training on the “normal” data and using the reconstruction error to identify outliers.
- Text Mining:
  - In natural language processing, sparse autoencoders can extract meaningful features from text data, which can be useful for tasks like document clustering and topic modeling.
- Image Recognition:
  - In computer vision, sparse autoencoders can learn to identify key features in images that can be useful for object recognition or segmentation tasks.
- Compressed Sensing:
  - In scenarios where data collection is expensive, sparse autoencoders can reconstruct a complete dataset from a small set of features.

Sparse Autoencoders – Limitations

- Computational Complexity:
  - Adding a sparsity constraint makes the optimization problem more complex and can increase training time.
- Hyperparameter Sensitivity:
  - The effectiveness of the constraint is highly dependent on the hyperparameters, requiring careful tuning.
- Local Optima:
  - The added complexity increases the chances of the optimization getting stuck in local optima.

Variational Autoencoders (VAEs)

- Variational Autoencoders (VAEs) are a specific type of autoencoder that aim to learn not just a deterministic function mapping inputs to outputs, but a probabilistic mapping from inputs to a distribution in a latent space.
- This makes VAEs part of the family of generative models, allowing them to generate new data that's similar to the training data.



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

VAEs – How It Works

- Encoder:
  - Unlike traditional autoencoders that output a single point in the latent space, the encoder in a VAE outputs parameters of a probability distribution (usually Gaussian). These parameters often include a mean μ and a standard deviation σ for each dimension of the latent space.
- Reparameterization Trick:
  - To make the network differentiable, a sample z from the distribution is obtained using a reparameterization trick: $z = \mu + \sigma \odot \epsilon$, where ϵ is random noise.
- Decoder:
  - The decoder takes the sampled latent variable z and maps it back to the original data space, reconstructing the input.
- Loss Function:
  - The loss function has two terms:

    1) Reconstruction Loss: Measures how well the reconstructed output matches the original input.

    2) KL Divergence: Measures how much the learned latent variable distribution diverges from a prior (usually a standard Gaussian distribution).

Note. The notation $\odot$ represents element-wise multiplication, also known as the Hadamard product. In this operation, each element in one matrix is multiplied by the corresponding element in another matrix of the same dimensions. $\sigma \odot \epsilon$ means that each element in the σ vector is multiplied by the corresponding element in the ϵ vector to produce the sample z from the latent distribution.

VAEs – Real-world Applications

- Image Generation:
  - VAEs are often used for generating realistic images for tasks like art creation, game design, and data augmentation.
- Anomaly Detection:
  - In industrial and financial applications, VAEs can be used to model 'normal' behavior and flag anomalies based on reconstruction error and latent space distribution.
- Drug Discovery:
  - VAEs can generate molecular structures for new drugs that are likely to have desired properties.
- Natural Language Processing:
  - Though less common, VAEs have been adapted to work with text data for tasks like text generation and topic modeling.

VAEs – Limitations

- Complexity:
  - VAEs are more complex to implement and train compared to basic autoencoders due to the added probabilistic layer and more complex loss function.
- Blurred Reconstructions:
  - VAEs often produce blurrier reconstructions compared to deterministic autoencoders because they average over multiple sampled latent variables.
- Optimization Challenges:
  - The dual nature of the loss function makes the optimization more challenging and may require careful tuning of hyperparameters.

Generative Adversarial Networks

- Introduction
- Theoretical Foundations
- Key Architectures and Variants of GANs
- Training GANs

Introduction (1/3)

- Generative Adversarial Networks (GANs) are a class of artificial intelligence algorithms used in unsupervised machine learning
- A GAN consists of two neural networks, namely the Generator and the Discriminator, which are trained simultaneously.
- The Generator aims to produce data that mimics some distribution (often the distribution of the training data), while the Discriminator aims to distinguish between genuine and generated data.

Introduction (2/3)

- Generator:
  - This network aims to produce data that is indistinguishable from some real data.
  - It starts with a random noise and gradually refines its output as the training process advances.
- Discriminator:
  - This network tries to differentiate between real and fake (generated) data.
  - It takes in both real and fake samples and assigns a probability that a given sample is real.

Introduction (3/3)

- The Generator and Discriminator are in essence co-adversaries, engaged in a kind of cat-and-mouse game, often described mathematically as a minimax game or a zero-sum game.
- In a zero-sum game, one player's gain or loss is exactly balanced by the losses or gains of another player.
- In the context of GANs, the Generator and Discriminator are essentially playing a zero-sum game against each other.

Importance of GANs

- **Data Generation:** One of the most obvious utilities of GANs is the generation of new data that mimics the distribution of the training data. This is useful in domains like artwork creation, where GANs can create realistic images, or in natural language processing, where they can generate human-like text.
- **Data Augmentation:** For tasks where the dataset is limited, GANs can create additional data to augment the existing dataset. This is particularly useful in fields like healthcare, where obtaining more real data is often difficult or expensive.
- **Anomaly Detection:** Because GANs learn the distribution of the training data, they can be used to identify data points that do not fit this distribution. This is useful in fields like fraud detection and network security.
- **Complexity:** GANs are capable of capturing complex, high-dimensional distributions. This makes them powerful tools for tasks that involve understanding the underlying distribution of complex datasets.
- **Representation Learning:** While the Generator is obviously useful, the Discriminator also learns useful features of the data in the process of trying to distinguish between real and fake. These learned features can often be used for other machine learning tasks, a practice known as transfer learning.
- **Research and Innovation:** GANs have become a cornerstone in the machine learning research community, inspiring a wide range of variations and applications, from image-to-image translation (like turning satellite images into maps) to drug discovery.

Historical Background

- GANs were introduced by Ian Goodfellow and his colleagues in 2014. The concept revolutionized the way we think about generating data in an unsupervised manner.
- Evolution
  - DCGAN (Deep Convolutional GAN): Improved stability in training.
  - CycleGAN: For unpaired image-to-image translation.
  - WGAN (Wasserstein GAN): Introduced Wasserstein loss for more stable training.
  - BigGAN: GANs that can generate high-resolution and high-quality images.
  - StyleGAN & StyleGAN2: Introduced style-based generators.

Source: https://github.com/hindupuravinash/the-gan-zoo

Core Components – Generator (G)

- Definition: A neural network that takes a random noise vector as input and produces synthetic samples as output.
- Objective: To generate data that is indistinguishable from real data.
- Terminology:
  - Latent Space: The high-dimensional space from which random vectors are sampled as inputs for the generator.
  - Upsampling: The process of increasing the dimensions of the data, often used in the generator to transform a low-dimensional random vector into a high-dimensional synthetic sample.

Upsampling in Generator

- Upsampling in the context of the Generator refers to the process of increasing the spatial dimensions (height and width) of the input tensor, often achieved through operations like deconvolution or transposed convolution.
- Mechanism:
  - Transposed Convolution: This operation essentially reverses the effect of a regular convolution. It maps a low-resolution input to a higher resolution output.
  - Pixel Shuffle: Another technique that rearranges elements in a tensor to expand its dimensions.
  - Nearest Neighbor / Bilinear Interpolation: Simple yet effective techniques to increase tensor dimensions by filling in new values based on neighboring values.

Why Upsampling Is Important

- Data Generation: The Generator starts with a low-dimensional noise vector. To produce complex, high-dimensional output such as images, upsampling is essential.
- Detail Recovery: Upsampling helps in generating finer details as the tensor goes through the network, enabling the production of high-quality synthetic data.

Core Components – Discriminator (D)

- Definition: A neural network that tries to distinguish between real and generated samples.
- Objective: To correctly classify a given sample as real or fake.
- Terminology:
  - Downsampling: The process of reducing the dimensions of the data, often used in the discriminator to transform a high-dimensional sample into a scalar classification output.
  - Real Score: The output of the discriminator when given a real sample, ideally close to 1.

Downsampling in Discriminator

- Downsampling refers to the process of reducing the spatial dimensions of the input tensor. In GANs, this is typically accomplished using strided convolutions in the Discriminator.
- Mechanism:
  - Strided Convolution: A convolutional operation with a stride greater than 1, which effectively reduces the dimensions of the output tensor.
  - Pooling Layers: Operations like MaxPooling or AveragePooling can also be used, though they are less common in modern GAN architectures.
  - Global Average Pooling: Sometimes used to convert a 2D tensor into a 1D tensor before a fully connected layer for classification.

Why Downsampling Is Important

- Classification: The Discriminator needs to classify the input data as real or fake. Downsampling helps in reducing the computational complexity and focuses on essential features for classification.
- Feature Extraction: As the dimensions reduce, the Discriminator learns more abstract and complex features of the data, aiding in better classification.

Advanced Concepts (1/2)

- Mode Collapse:
  - Definition: A phenomenon where the generator starts to produce very similar or identical samples for different input vectors.
  - Solution: Techniques like minibatch discrimination, spectral normalization, etc., are employed to combat mode collapse.
- Nash Equilibrium:
  - Definition: A state in which neither the generator nor the discriminator can improve their performance given the current state of the other.
  - Significance: It is considered the optimal stopping point for GAN training, although reaching true Nash equilibrium is difficult in practice.

Advanced Concepts (2/2)

- Stability and Convergence:
  - Definition: Describes the behavior of the GAN during the training process.
  - Challenges: GANs are notorious for being difficult to train; they may not converge or may become unstable.
  - Solutions: Various architectural tweaks and training techniques have been proposed, such as gradient clipping, learning rate annealing, and different normalization methods.

Nash Equilibrium

- Nash Equilibrium is a concept from game theory that describes a situation in which each player's strategy is optimal given the strategy of their opponent(s).
- In simpler terms, in a Nash Equilibrium, no player can improve their payoff by unilaterally deviating from their current strategy, assuming the other players keep their strategies unchanged.
- In GAN training, reaching a Nash Equilibrium means that the Generator has become so good at generating fake data that the Discriminator can no longer distinguish between real and fake data.
  - At this point, both networks are said to be in equilibrium because neither can improve without the other changing.

Nash Equilibrium – Mathematical Representation

- In the context of GANs, the Nash Equilibrium can be represented by the minimax function:

$$\min_{G} \max_{D} V(D, G)$$

- Here, V(D,G) is the value function that both the Generator (G) and the Discriminator (D) are trying to optimize.
- The Generator aims to minimize this function, while the Discriminator aims to maximize it.
- In a Nash Equilibrium, neither can improve their objective function value without the other also changing their strategy.

Nash Equilibrium – Challenges

- Computational Complexity:

  Finding the Nash Equilibrium in a high-dimensional, non-convex game like GANs is computationally intensive.

- Mode Collapse:

  Before reaching Nash Equilibrium, GANs can suffer from issues like mode collapse, where the Generator produces limited varieties of samples.

Researchers often aim for a “local” Nash Equilibrium, where the networks are relatively stable, and the generated data is of high quality, even if it's not a perfect equilibrium.

Training Dynamics – Adversarial Training

- The fundamental idea behind GANs is adversarial training, where the Generator and the Discriminator are trained simultaneously but with opposing objectives.
  - Objective of the Generator (G): To generate data that is indistinguishable from real data.
  - Objective of the Discriminator (D): To correctly identify whether the given data is real or generated by the Generator.
- The training process iteratively updates both G and D, ideally until they reach a Nash equilibrium, where neither can improve without the other changing as well.

Training Dynamics – Loss Function

- The loss function, usually a form of Binary Cross-Entropy loss, measures how well each network is doing its job. It's common to see the following setup:
  - The first term encourages D to correctly classify real samples.
  - The second term encourages D to correctly classify fake samples.

- Discriminator Loss: $L_D = -\log\big(D(x)\big) - \log\big(1 - D(G(z))\big)$
- Generator Loss: $L_G = -\log(D(G(z)))$
  - This term encourages G to generate samples that D classifies as real.

Components in Generator (G)

- Input Layer: Takes a random noise vector *z* sampled from a latent space.
- Hidden Layers: Comprises a series of fully connected, convolutional, or transposed convolutional layers.
- Output Layer: Produces the synthetic data, typically matching the dimensions of the real data.

Operations in Generator (G)

- Upsampling: Uses transposed convolutional layers or other techniques to transform a low-dimensional noise vector into a high-dimensional synthetic sample.
- Activation Functions: Commonly uses ReLU or Leaky ReLU in hidden layers and tanh or Sigmoid in the output layer, depending on the data type.

Components in Discriminator (D)

- Input Layer: Takes in data samples, which could either be real or generated by the Generator.
- Hidden Layers: Typically comprises a series of convolutional layers and fully connected layers.
- Output Layer: Outputs a scalar value representing the probability that the input sample is real.

Operations in Discriminator (D)

- Downsampling: Employs strided convolutions or pooling layers to reduce the dimensions of the input data.
- Activation Functions: Often uses Leaky ReLU in hidden layers and Sigmoid in the output layer to output probabilities.

Data Flow in GAN

1. Random noise vectors are sampled from the latent space.
2. These vectors are fed into the Generator.
3. Upsampling layers in the Generator transform these vectors into high-dimensional data.
4. This generated data and real data are fed into the Discriminator.
5. Downsampling layers in the Discriminator reduce the data dimensions.
6. The Discriminator then classifies the downsampled data as real or fake.

Latent Space in GANs

- The latent space is a high-dimensional space from which the random noise vectors are sampled as input to the Generator.
- It serves as a compact representation of the data's inherent features.
- In most GANs, you randomly sample from a predefined distribution (usually a Gaussian distribution) to generate noise vectors.
- The latent space evolves implicitly as the Generator learns to map these random vectors to realistic data.

How to Train a GAN (1/4)

1. Initialize Networks: Initialize the Generator (G) and Discriminator (D) with random weights.
2. Preprocess Data: Prepare your real data samples, often involving normalization and data augmentation.

How to Train a GAN (2/4)

3. Training Loop: Typically involves the following steps:

Step 1: Train the Discriminator

1.1 Sample a mini-batch of real data x from the data distribution.

1.2 Generate a mini-batch of fake data G(z), where z is a random noise vector.

1.3 Compute the Discriminator's loss on real and fake data. Commonly used loss is Binary Cross-Entropy: $L_D = -\log\big(D(x)\big) - \log\big(1 - D(G(z))\big)$

1.4 Update the Discriminator's weights using backpropagation to minimize L_D.

How to Train a GAN (3/4)

3. Training Loop: Typically involves the following steps:

   Step 2: Train the Generator

   2.1 Generate a new mini-batch of fake data G(z).

   2.2 Compute the Generator's loss. The aim is to fool the Discriminator into thinking the fake samples are real: $L_G = -\log(D(G(z)))$

   2.3 Update the Generator's weights using backpropagation to minimize L_G.

How to Train a GAN (4/4)

3. Training Loop: Typically involves the following steps:

   Step 3: Check Convergence

   3.1 Monitor the losses L_D and L_G.

   3.2 Optionally, evaluate using metrics like Fréchet Inception Distance (FID) or Inception Score (IS).

   3.3 If the networks have converged or met specific criteria, exit the loop; otherwise, return to Step 1.

Evaluation Metrics for GANs (1/3)

- Fréchet Inception Distance (FID):
  - FID measures the similarity between the generated data and real data distributions in the feature space of a pre-trained Inception network.
  - The FID between two multivariate Gaussians $N(\mu_1, \Sigma_1)$ and $N(\mu_2, \Sigma_2)$ is defined as: $\text{FID} = ||\mu_1 - \mu_2||^2 + \text{Tr}(\Sigma_1 + \Sigma_2 - 2(\Sigma_1\Sigma_2)^{\frac{1}{2}})$. μ_1, Σ_1 are the mean and covariance of features from the real data, and μ_2, Σ_2 are from the generated data.

Evaluation Metrics for GANs (2/3)

- Inception Score (IS)
  - IS uses the Inception network to classify generated images into different classes and computes a score based on the distribution of these classes.
  - For a set of N generated samples, the IS is given by: $\text{IS} = \exp(\frac{1}{N}\sum_{i=1}^{N} KL(p(y|x_i)||p(y))$. $p(y|x_i)$ is the conditional class distribution for sample x_i, and $p(y)$ is the marginal class distribution.

Evaluation Metrics for GANs (3/3)

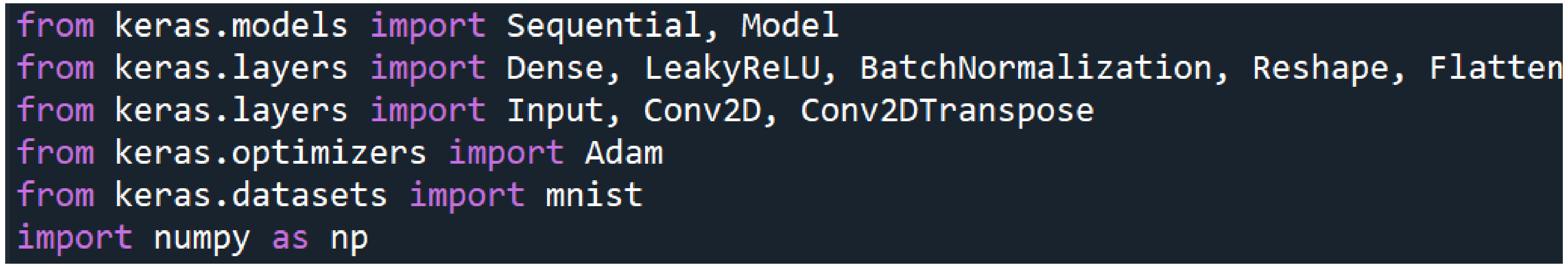
- Wasserstein Distance (For WGANs)
  - This metric is specific to Wasserstein GANs (WGANs) and measures the Earth Mover's Distance between the real and generated data distributions.
  - In the context of WGANs, the Wasserstein distance is given by the Discriminator‘s loss: $W = \max_D \mathbb{E}_{x \sim P_{\text{real}}} [D(x)] - \mathbb{E}_{x \sim P_{\text{fake}}} [D(x)]$.$\mathbb{E}$ represents the expected value, and P_{real}, P_{fake} are the real and fake data distributions..

Hand-on

- A simple GAN using Keras

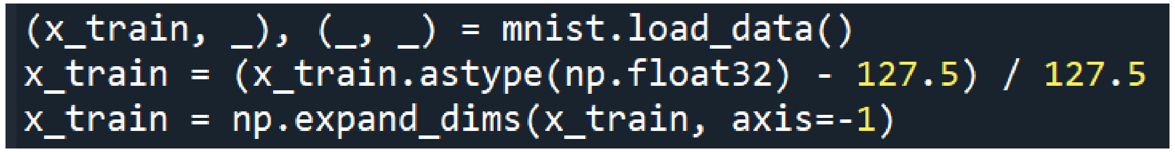
A simple GAN using Keras (1/6)

1. Import Required Libraries



```python
from keras.models import Sequential, Model
from keras.layers import Dense, LeakyReLU, BatchNormalization, Reshape, Flatten
from keras.layers import Input, Conv2D, Conv2DTranspose
from keras.optimizers import Adam
from keras.datasets import mnist
import numpy as np
```

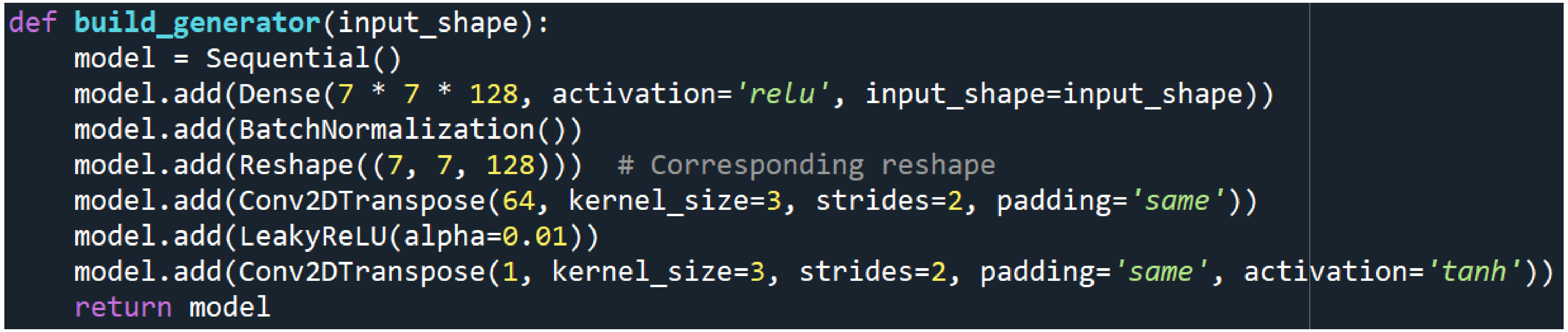
2. Load and Preprocess the Data



```python
(x_train, _), (_, _) = mnist.load_data()
x_train = (x_train.astype(np.float32) - 127.5) / 127.5
x_train = np.expand_dims(x_train, axis=-1)
```

A simple GAN using Keras (2/6)

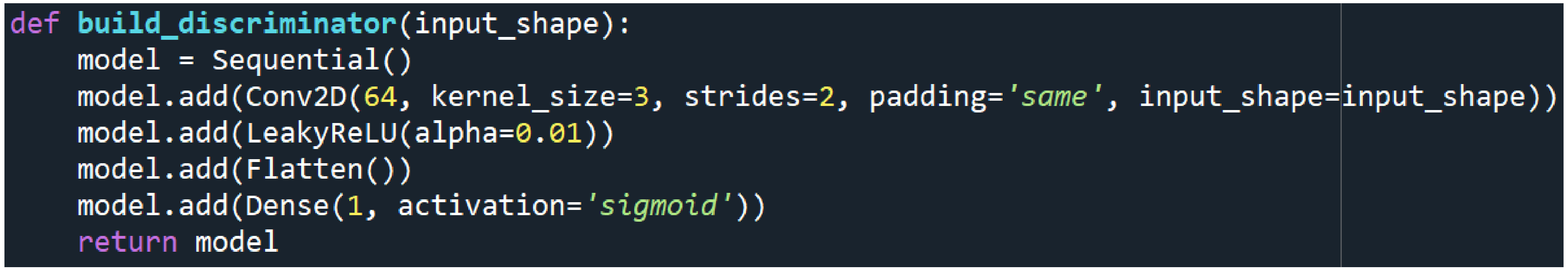
3. Define the Generator Model



```
def build_generator(input_shape):
    model = Sequential()
    model.add(Dense(7 * 7 * 128, activation='relu', input_shape=input_shape))
    model.add(BatchNormalization())
    model.add(Reshape((7, 7, 128)))  # Corresponding reshape
    model.add(Conv2DTranspose(64, kernel_size=3, strides=2, padding='same'))
    model.add(LeakyReLU(alpha=0.01))
    model.add(Conv2DTranspose(1, kernel_size=3, strides=2, padding='same', activation='tanh'))
    return model
```

A simple GAN using Keras (3/6)

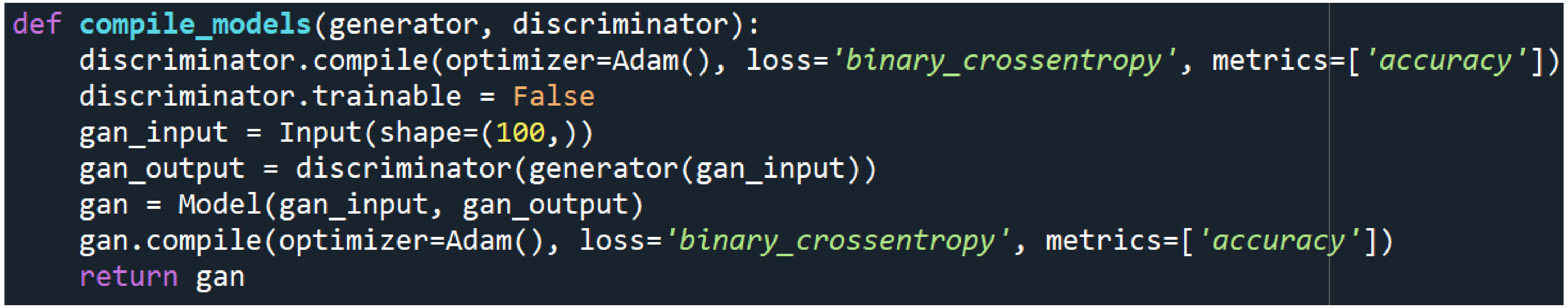
4. Define the Discriminator Model



```
def build_discriminator(input_shape):
    model = Sequential()
    model.add(Conv2D(64, kernel_size=3, strides=2, padding='same', input_shape=input_shape))
    model.add(LeakyReLU(alpha=0.01))
    model.add(Flatten())
    model.add(Dense(1, activation='sigmoid'))
    return model
```

A simple GAN using Keras (4/6)

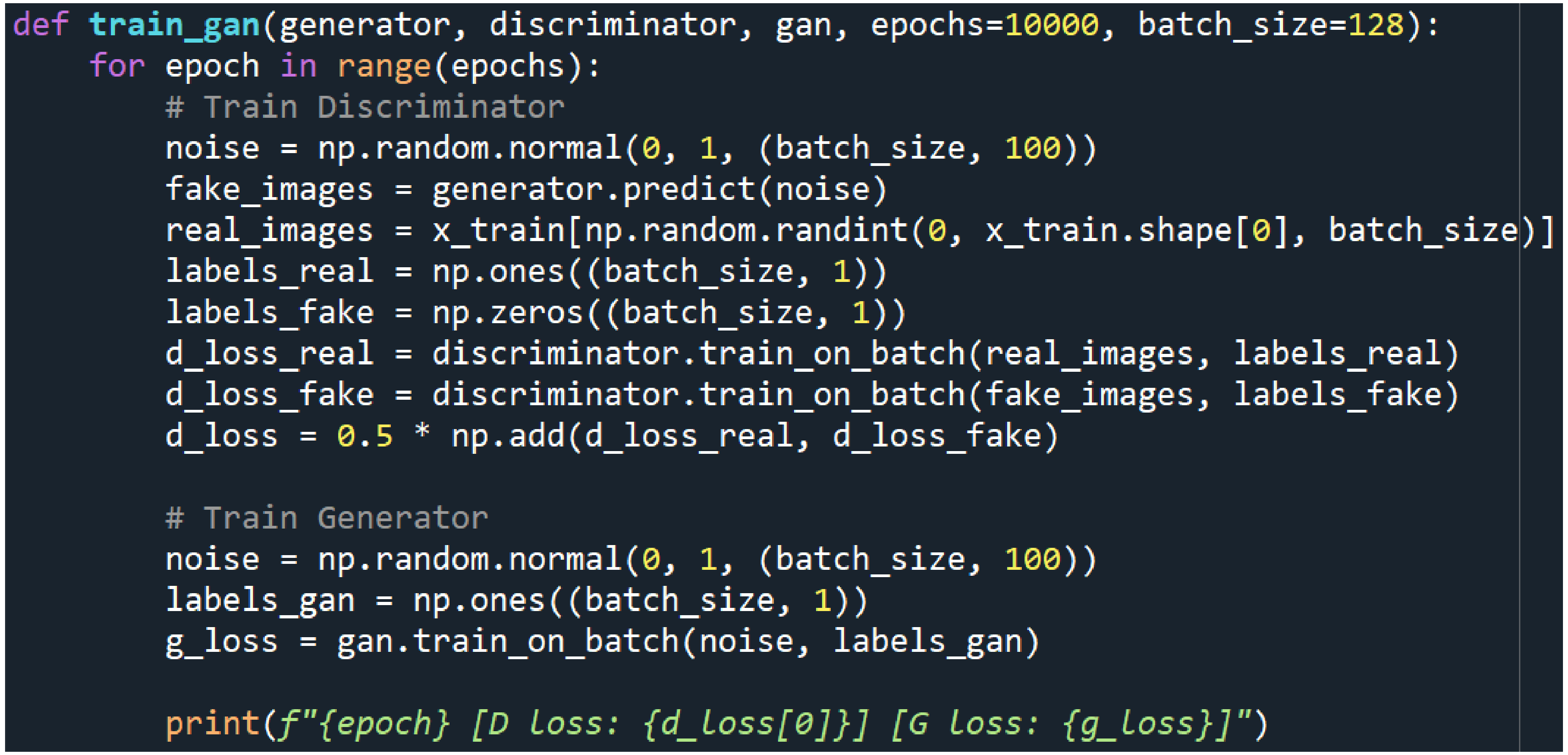
5. Compile the Models



```
def compile_models(generator, discriminator):
    discriminator.compile(optimizer=Adam(), loss='binary_crossentropy', metrics=['accuracy'])
    discriminator.trainable = False
    gan_input = Input(shape=(100,))
    gan_output = discriminator(generator(gan_input))
    gan = Model(gan_input, gan_output)
    gan.compile(optimizer=Adam(), loss='binary_crossentropy', metrics=['accuracy'])
    return gan
```

A simple GAN using Keras (5/6)

6. Train the GAN



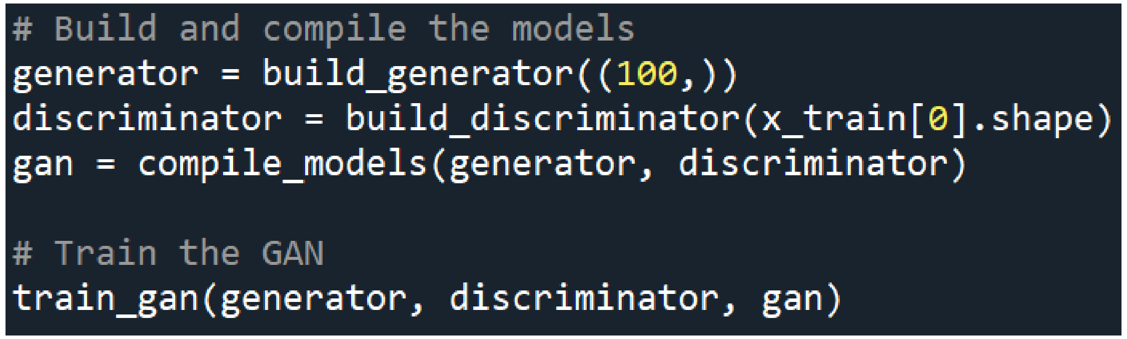
```python
def train_gan(generator, discriminator, gan, epochs=10000, batch_size=128):
    for epoch in range(epochs):
        # Train Discriminator
        noise = np.random.normal(0, 1, (batch_size, 100))
        fake_images = generator.predict(noise)
        real_images = x_train[np.random.randint(0, x_train.shape[0], batch_size)]
        labels_real = np.ones((batch_size, 1))
        labels_fake = np.zeros((batch_size, 1))
        d_loss_real = discriminator.train_on_batch(real_images, labels_real)
        d_loss_fake = discriminator.train_on_batch(fake_images, labels_fake)
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

        # Train Generator
        noise = np.random.normal(0, 1, (batch_size, 100))
        labels_gan = np.ones((batch_size, 1))
        g_loss = gan.train_on_batch(noise, labels_gan)

        print(f"{epoch} [D loss: {d_loss[0]}] [G loss: {g_loss}]")
```

A simple GAN using Keras (6/6)

7. Execute the Training



```
# Build and compile the models
generator = build_generator((100,))
discriminator = build_discriminator(x_train[0].shape)
gan = compile_models(generator, discriminator)

# Train the GAN
train_gan(generator, discriminator, gan)
```

Thank you