CE6146
Introduction to Deep Learning
Recurrent Neural Networks

Chia-Ru Chung

Department of Computer Science and Information Engineering

National Central University

2023/10/19

20231012 Exercise

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

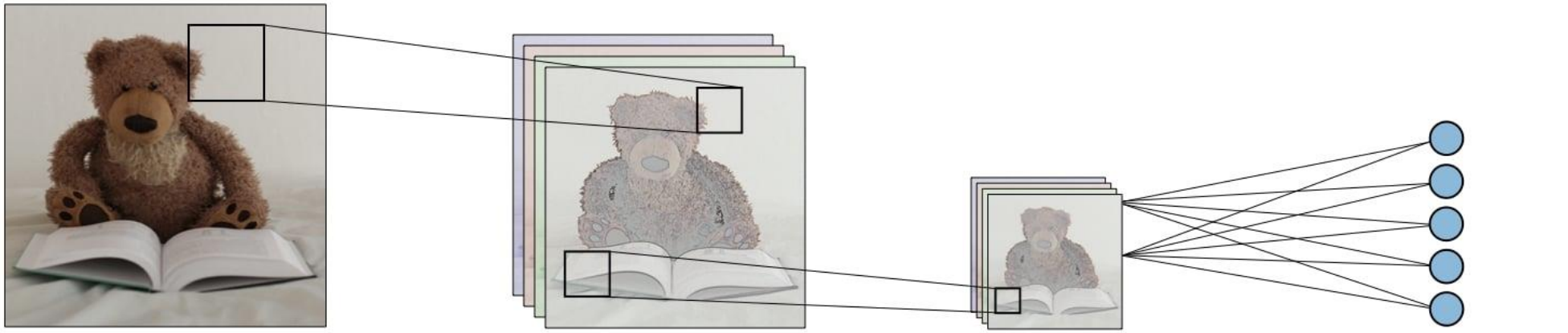
Outline

- Review
- Recurrent Neural Networks
- Hand-on

Review

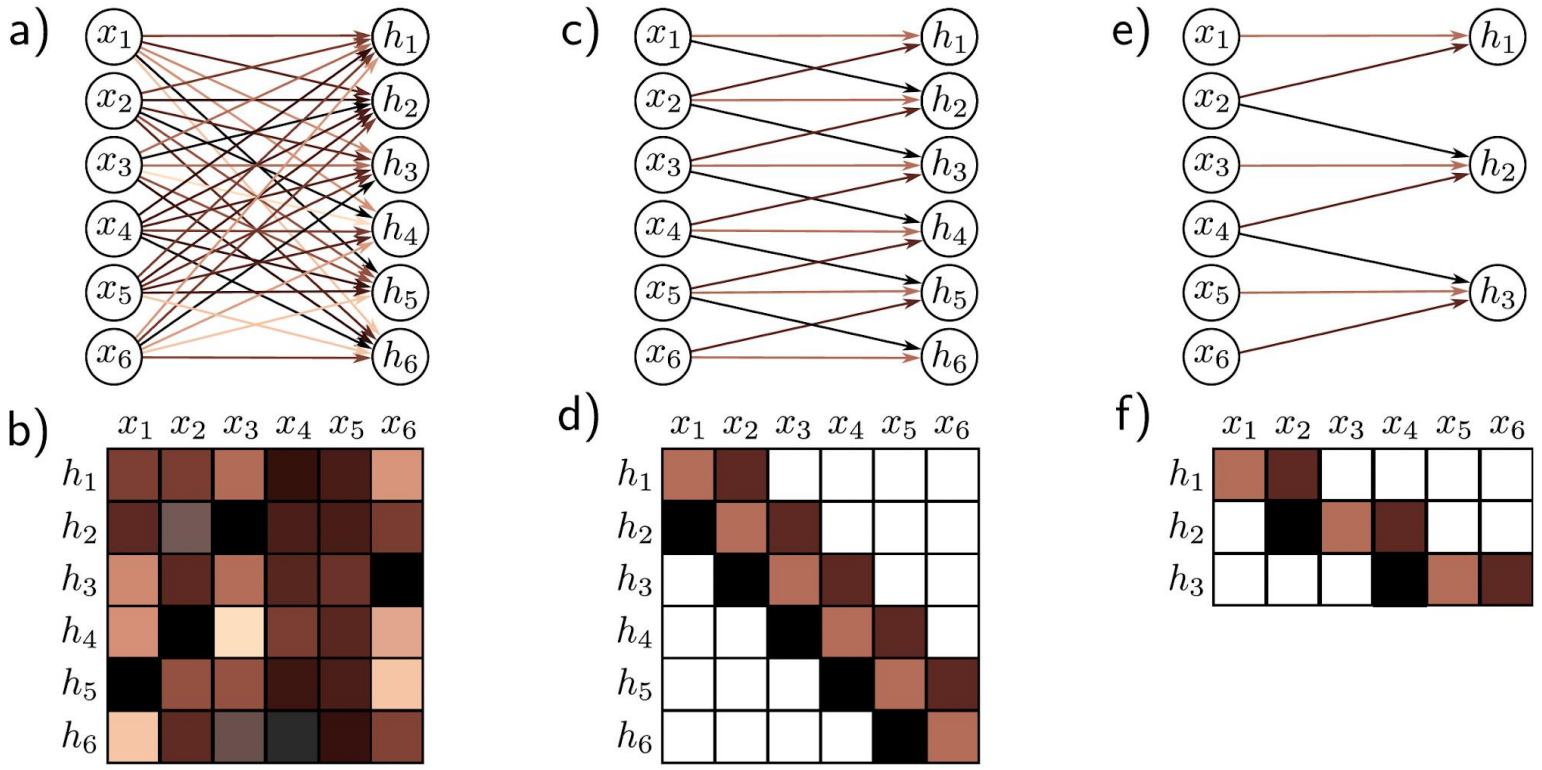
- Convolutional Neural Networks

Architecture of a Traditional CNN



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

Fully Connected vs. Convolutional Layer



a) A fully connected layer has a weight connecting each input x to each hidden unit h (colored arrows) and a bias for each hidden unit (not shown).

b) Hence, the associated weight matrix Ω contains 36 weights relating the six inputs to the six hidden units.

c) A convolutional layer with kernel size three computes each hidden unit as the same weighted sum of the three neighboring inputs (arrows) plus a bias (not shown).

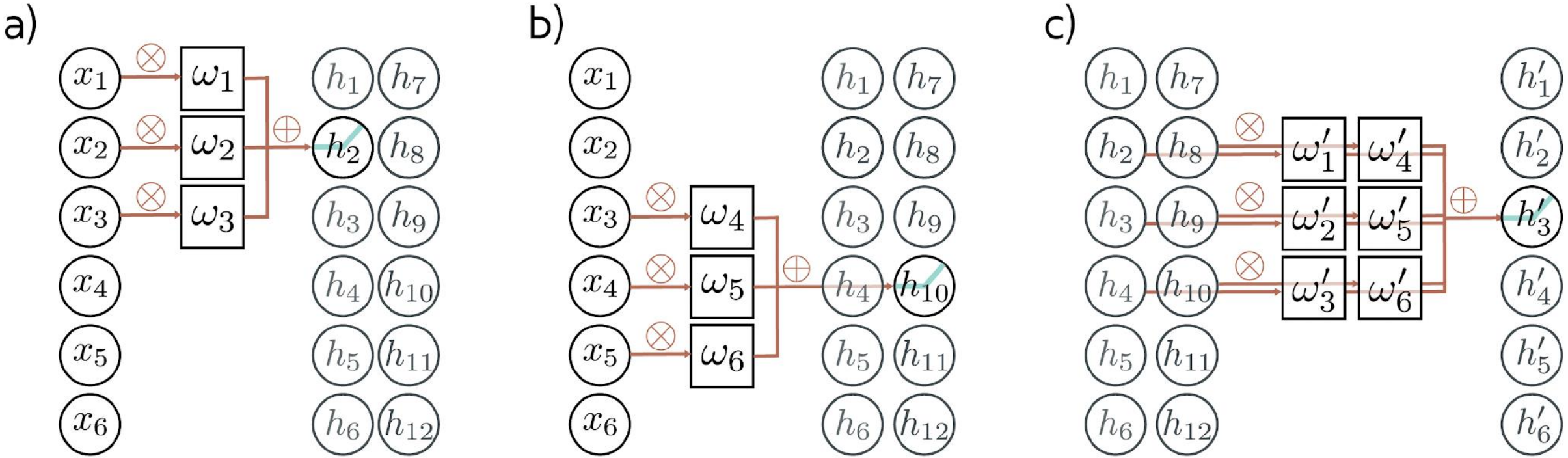
d) The weight matrix is a special case of the fully connected matrix where many weights are zero and others are repeated (same colors indicate same value, white indicates zero weight).

e) A convolutional layer with kernel size three and stride two computes a weighted sum at every other position.

f) This is also a special case of a fully connected network with a different sparse weight structure.

Source: Figure 10.4 in Understanding Deep Learning by Simon J.D. Prince.

Channels



Channels. Typically, multiple convolutions are applied to the input x and stored in channels.

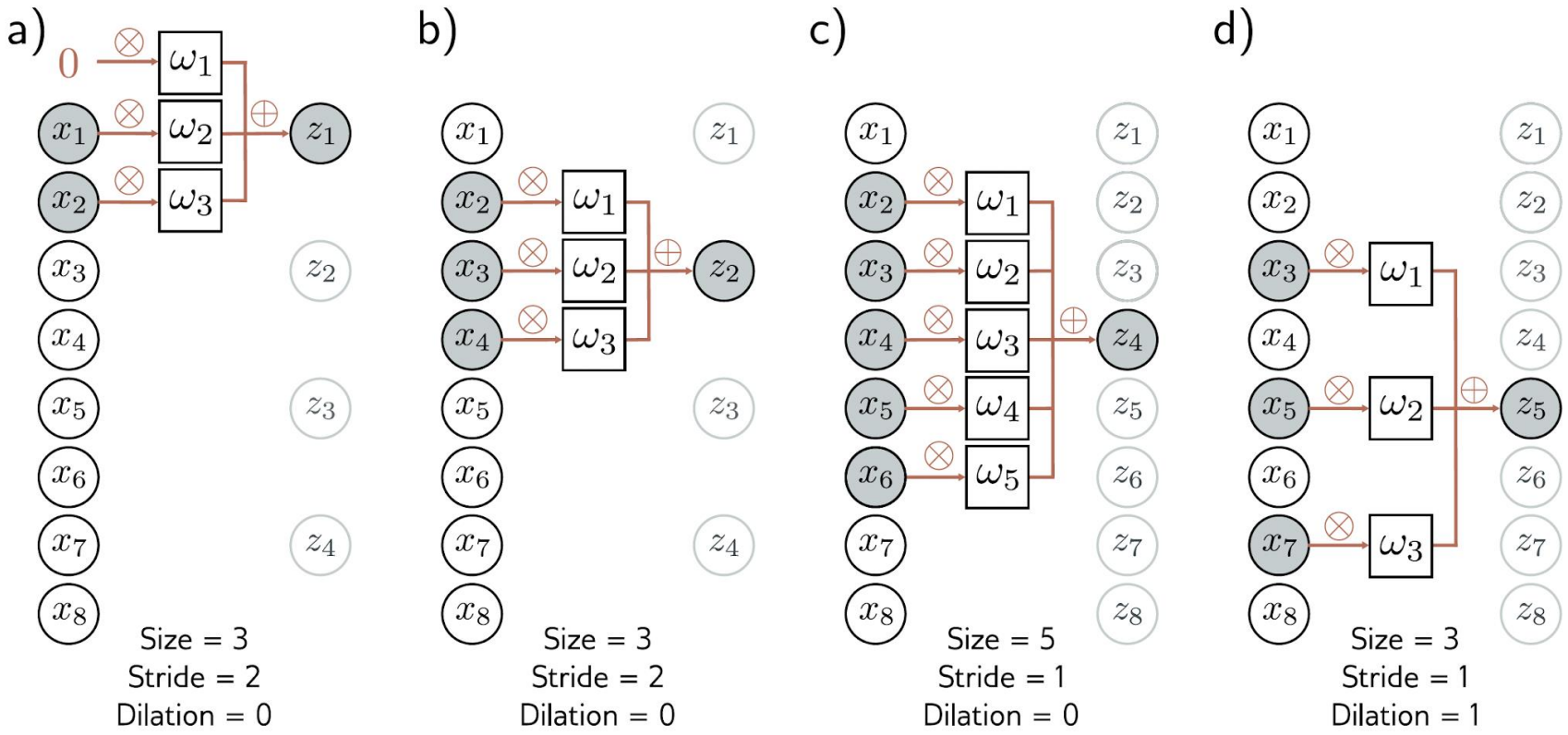
a) A convolution is applied to create hidden units h1 to h6, which form the first channel.

b) A second convolution operation is applied to create hidden units h7 to h12, which form the second channel. The channels are stored in a 2D array H1 that contains all the hidden units in the first hidden layer.

c) If we add a further convolutional layer, there are now two channels at each input position. Here, the 1D convolution defines a weighted sum over both input channels at the three closest positions to create each new output channel.

Source: Figure 10.5 in Understanding Deep Learning by Simon J.D. Prince.

Stride, Kernel Size, and Dilation

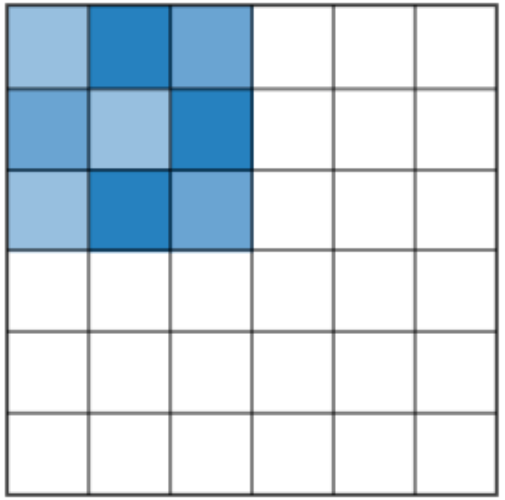


a) With a stride of two, we evaluate the kernel at every other position, so the first output z_1 is computed from a weighted sum centered at x_1, and b) the second output z_2 is computed from a weighted sum centered at x_3 and so on. c) The kernel size can also be changed. With a kernel size of five, we take a weighted sum of the nearest five inputs. d) In dilated or atrous convolution, we intersperse zeros in the weight vector to allow us to combine information over a large area using fewer weights.

Source: Figure 10.3 in Understanding Deep Learning by Simon J.D. Prince.

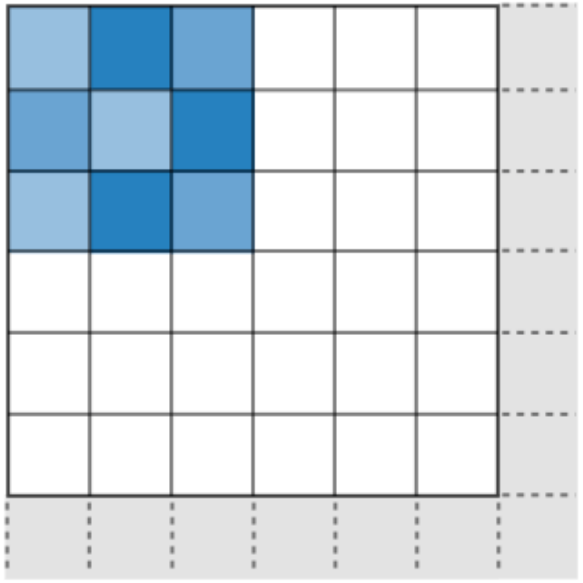
Padding

Valid



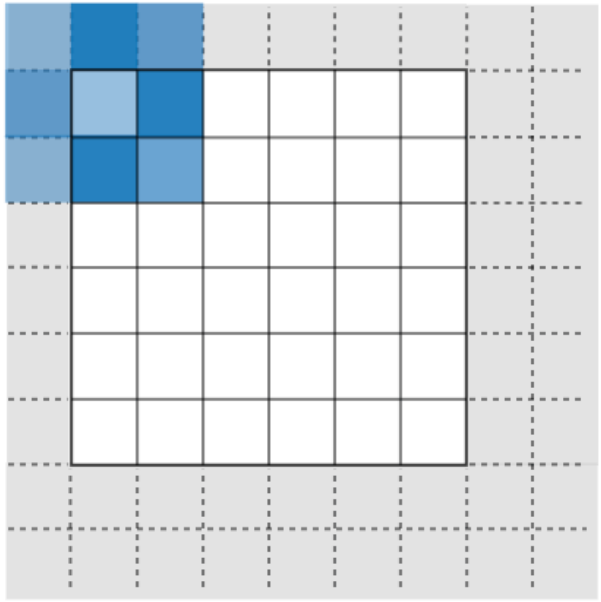
- No padding.
- Drops last convolution if dimensions do not match.

Same



- Output size is mathematically convenient.
- Also called 'half' padding.

Full



- Maximum padding such that end convolutions are applied on the limits of the input.

What is Dilation

- Dilation refers to the spacing between the values in a kernel.
- A dilation rate of 1 means adjacent values, and greater than 1 means there are gaps.
- In mathematical terms, dilation introduces zeros between the kernel values, effectively expanding its field of view without increasing the number of parameters.

Why Use Dilation

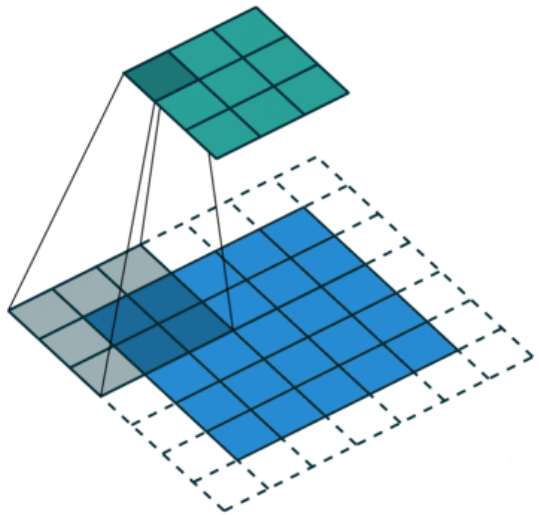
- Dilation allows the model to have a wider field of view, capturing more contextual information without increasing computational complexity.
- It is particularly useful in tasks that require understanding the context or when the objects of interest are spread out and not centered.
- Dilation can reduce the spatial dimensions of the output feature map less aggressively, preserving more spatial information.

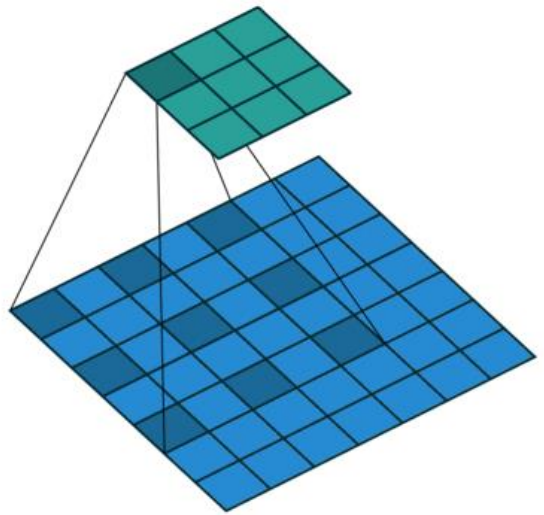
How Does Dilation Work

- Dilation in CNNs works by applying a dilated kernel to the input image or feature map.
- The kernel is applied to a larger area of the input, taking into account the dilation rate.

Practical Example of Dilation

- In a 3x3 filter with a dilation rate of 2, the filter would have the same field of view as a 5x5 filter but with only 9 parameters instead of 25.





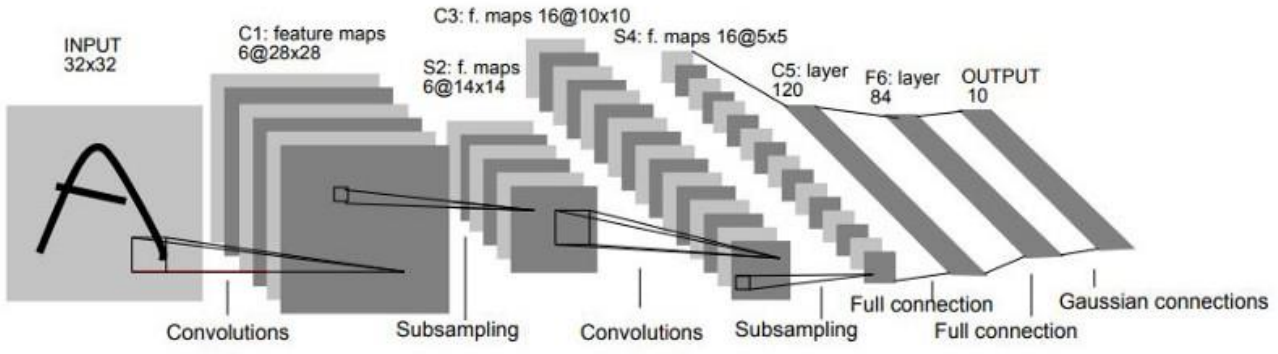
Source: https://towardsdatascience.com/review-dilated-convolution-semantic-segmentation-9d5a5bd768f5.

Calculation of The Output Size for CNN

- In a 3x3 filter with a dilation rate of 2, the filter would have the same field of view as a 5x5 filter but with only 9 parameters instead of 25.

LeNet-5 Architecture

- Developed by Yann LeCun in 1998.
- Primarily used for digit recognition tasks.
- Consists of two sets of convolutional and average pooling layers, followed by a fully connected layer.



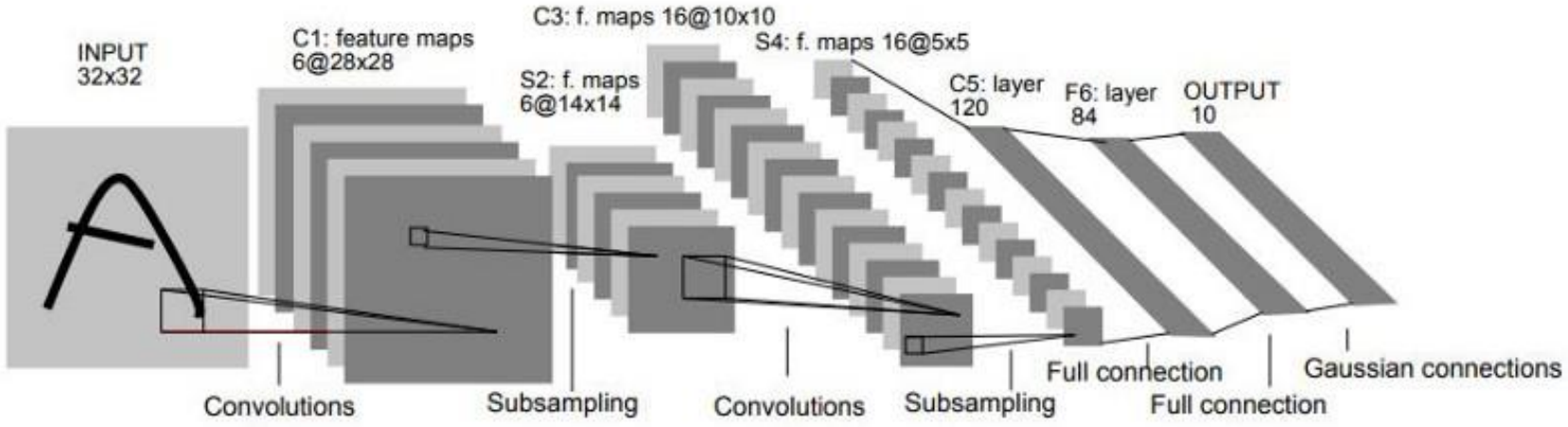
Source: https://www.datasciencecentral.com/lenet-5-a-classic-cnn-architecture/

Why LeNet-5

- LeNet-5 introduced key concepts like convolutions and pooling in a trainable architecture.
- It was one of the first successful applications of CNNs, showing superior performance in digit recognition.
- Understanding LeNet-5 can provide foundational knowledge necessary for grasping more complex architectures.

Overview of LeNet-5 Architecture

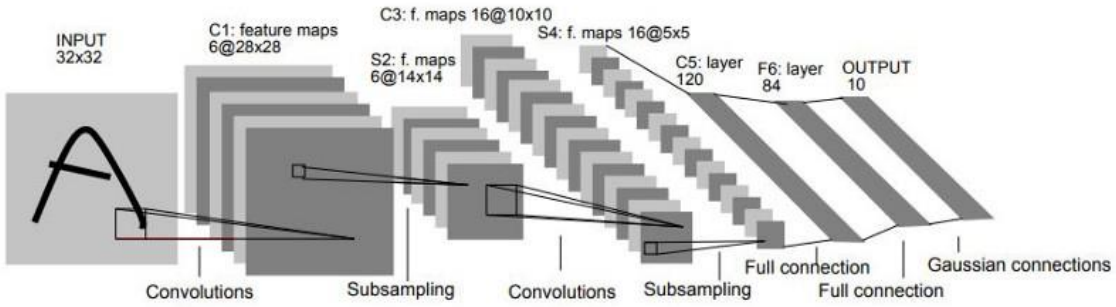
- The architecture has a total of 7 layers (excluding the input layer).
- The layers are organized as follows: [Input -> C1 -> S2 -> C3 -> S4 -> C5 -> F6 -> Output]



Source: https://www.datasciencecentral.com/lenet-5-a-classic-cnn-architecture/

Detailed Architecture and Size Settings (1/3)

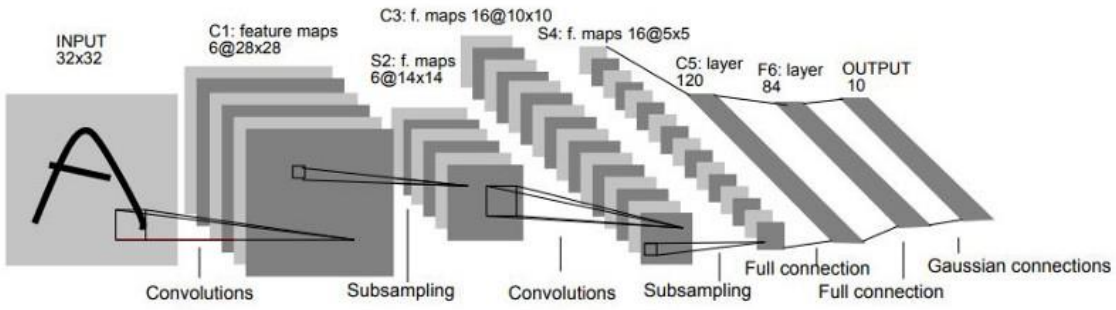
- Input Layer:
  - Size: 32×32
  - Depth: 1 (Grayscale)
- C1 (Convolutional Layer 1):
  - Filters: 6
  - Kernel size: 5×5
  - Output size: 28×28×6



Source: https://www.datasciencecentral.com/lenet-5-a-classic-cnn-architecture/

Detailed Architecture and Size Settings (2/3)

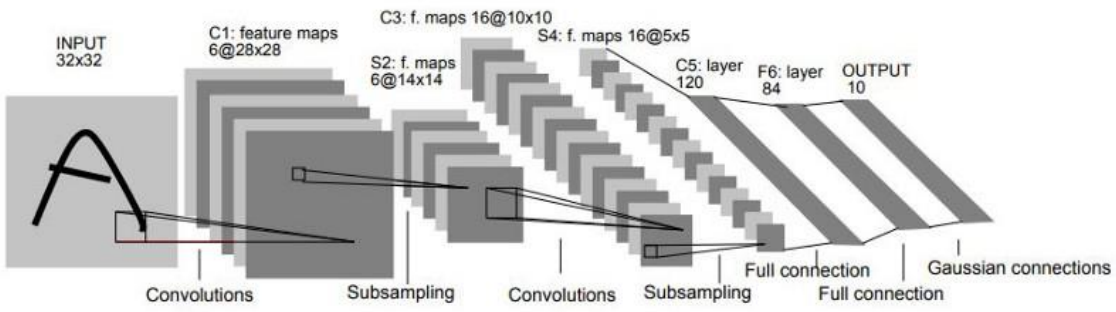
- S2 (Pooling Layer):
  - Filters: 6
  - Pooling size: 2×2
  - Output size: 14×14×6
- C3 (Convolutional Layer 3):
  - Filters: 16
  - Kernel size: 5×5
  - Output size: 10×10×16



Source: https://www.datasciencecentral.com/lenet-5-a-classic-cnn-architecture/

Detailed Architecture and Size Settings (3/3)

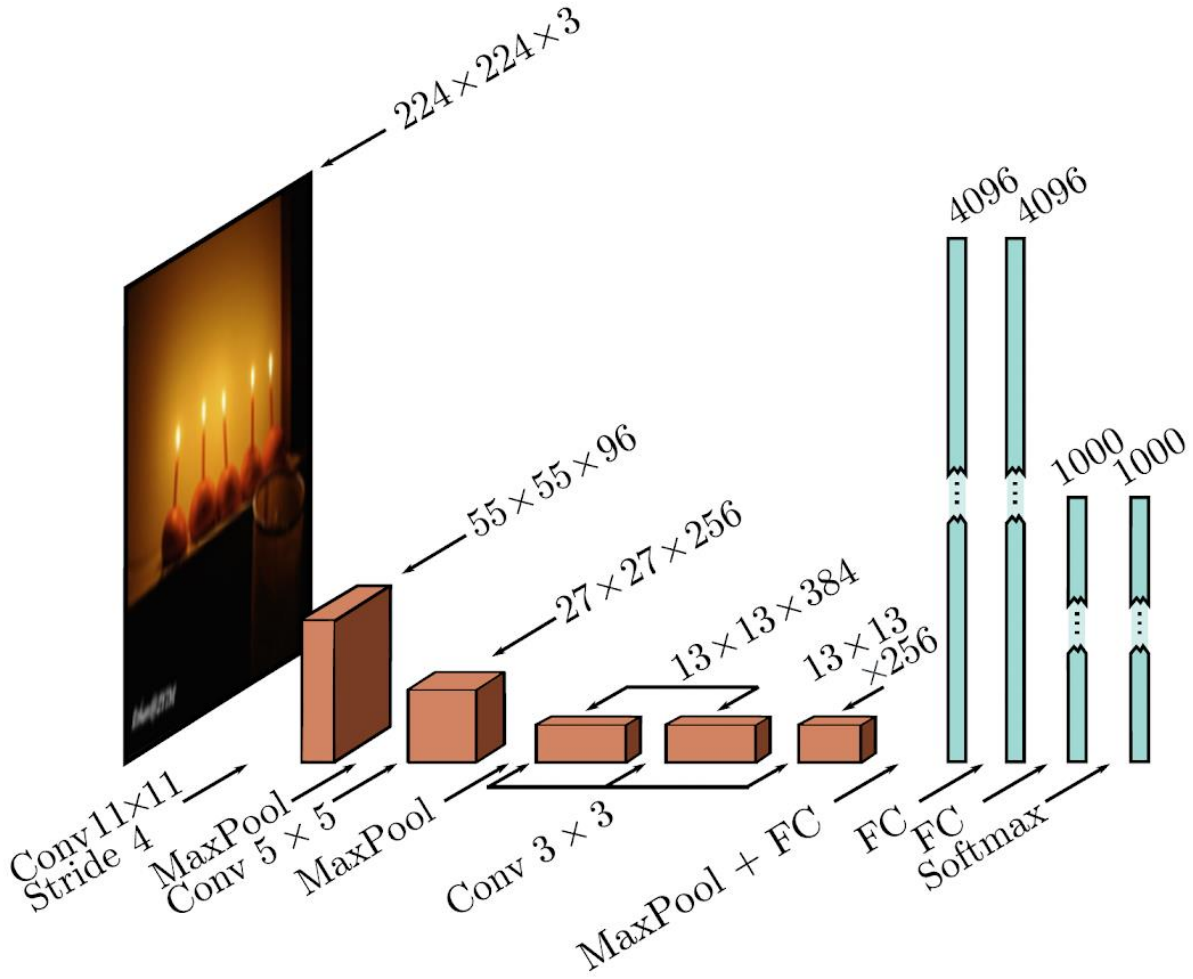
- S4 (Pooling Layer):
  - Filters: 16
  - Pooling size: 2×2
  - Output size: 5×5×16



- C5 (Convolutional Layer 5:
  - Filters: 120
  - Kernel size: 5×5
  - Output size: 1×1×120
- F6 (Fully Connected Layer):
  - Neurons: 84
- Output Layer:
  - Neurons: 10 (for digit recognition)

Source: https://www.datasciencecentral.com/lenet-5-a-classic-cnn-architecture/

AlexNet (Krizhevsky et al., 2012)



The network maps a 224×224 color image to a 1000-dimensional vector representing class probabilities. The network first convolves with 11×11 kernels and stride 4 to create 96 channels. It decreases the resolution again using a max pool operation and applies a 5×5 convolutional layer. Another max pooling layer follows, and three 3×3 convolutional layers are applied. After a final max pooling operation, the result is vectorized and passed through three fully connected (FC) layers and finally the softmax layer.

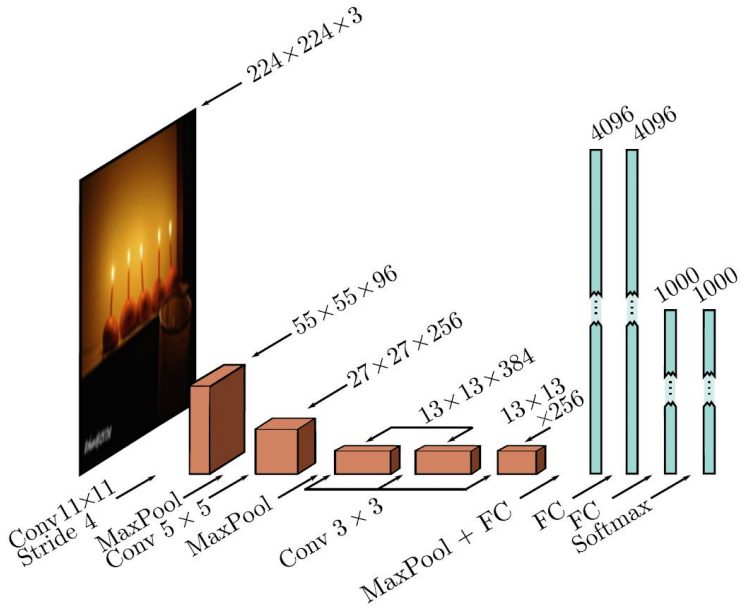
Source: Figure 10.16 in Understanding Deep Learning by Simon J.D. Prince.

Why AlexNet

- AlexNet demonstrated the viability and superiority of deep convolutional neural networks for large-scale image classification.
- It introduced novel techniques like ReLU activations, dropout, and data augmentation that are standard practices today.
- The architecture serves as a base for many modern CNN architectures, making it essential foundational knowledge.

Overview of AlexNet Architecture

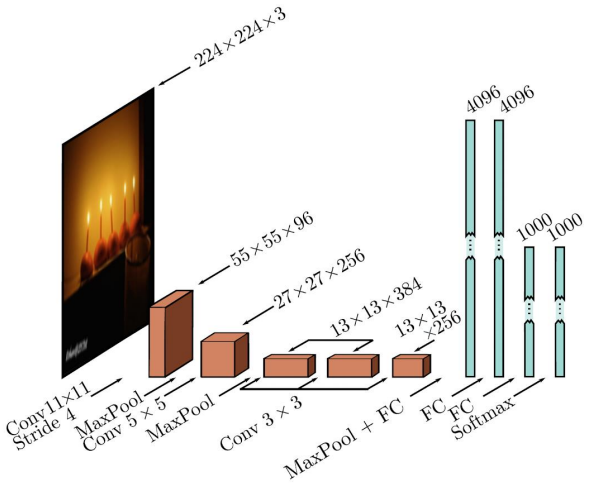
- The architecture has a total of 8 layers.
- 5 convolutional layers followed by 3 fully connected layers.



Source: Figure 10.16 in Understanding Deep Learning by Simon J.D. Prince.

Detailed Architecture and Size Settings (1/5)

- Input Layer:
  - Size: 32×32
  - Depth: 3 (RGB)

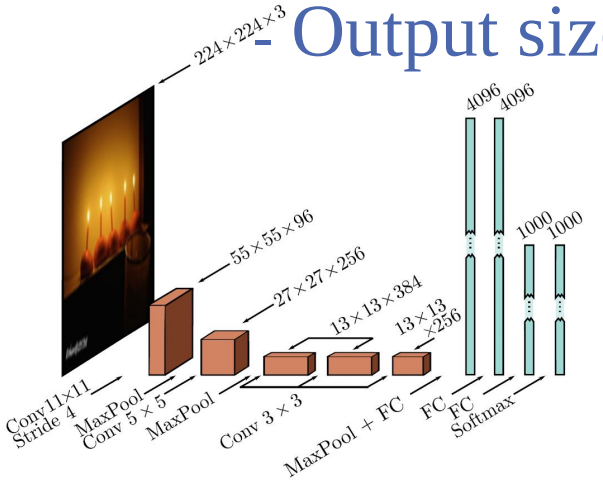


- Convolutional Layer 1 (Conv1):
  - Filters: 96
  - Kernel size: 11×11
  - Stride: 4
  - Output size: 55×55 × 96

Source: Figure 10.16 in Understanding Deep Learning by Simon J.D. Prince.

Detailed Architecture and Size Settings (2/5)

- Pooling Layer 1 (Pool1)
  - Stride: 2
  - Pooling size: 2×2
  - Output size: 27×27 × 96

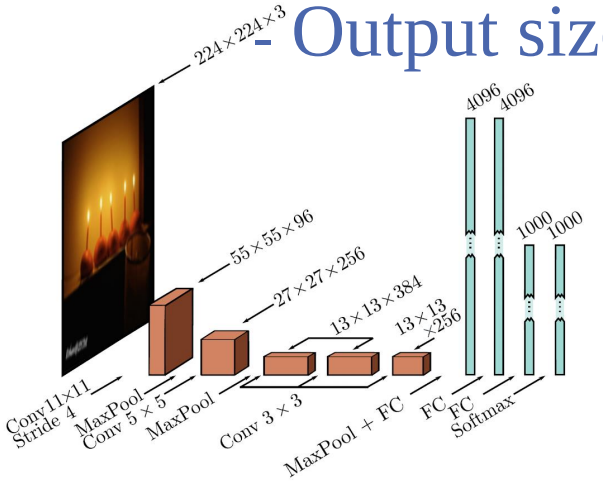


- Convolutional Layer 2 (Conv2):
  - Filters: 256
  - Kernel size: 5×5
  - Padding: 2
  - Output size: 27×27 ×256

Source: Figure 10.16 in Understanding Deep Learning by Simon J.D. Prince.

Detailed Architecture and Size Settings (3/5)

- Pooling Layer 2 (Pool2)
  - Stride: 2
  - Pooling size: 3×3
  - Output size: 13×13×96



- Convolutional Layer 3 (Conv3):
  - Filters: 384
  - Kernel size: 3×3
  - Padding: 1
  - Output size: 13×13 × 384

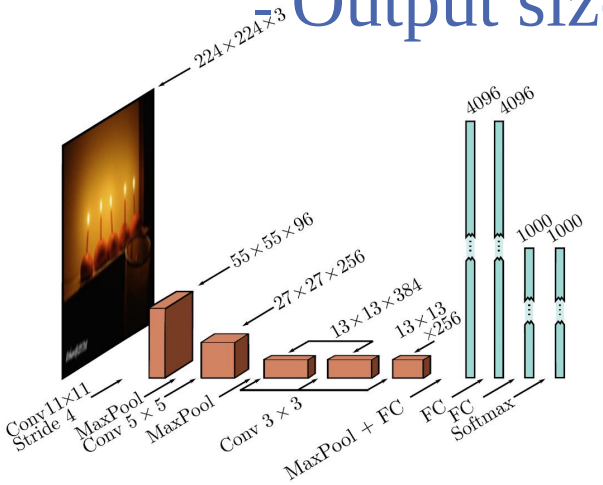
Source: Figure 10.16 in Understanding Deep Learning by Simon J.D. Prince.

Detailed Architecture and Size Settings (4/5)

- Convolutional Layer 4 (Conv4):
  - Filters: 384
  - Kernel size: 3×3
  - Padding: 1
  - Output size: 13×13 × 384
- Convolutional Layer 5 (Conv5):
  - Filters: 256
  - Kernel size: 3×3
  - Padding: 1
  - Output size: 13×13 × 256

Detailed Architecture and Size Settings (5/5)

- Pooling Layer 3 (Pool3):
  - Stride: 2
  - Pooling size: 3×3
  - Output size: 6×6×256

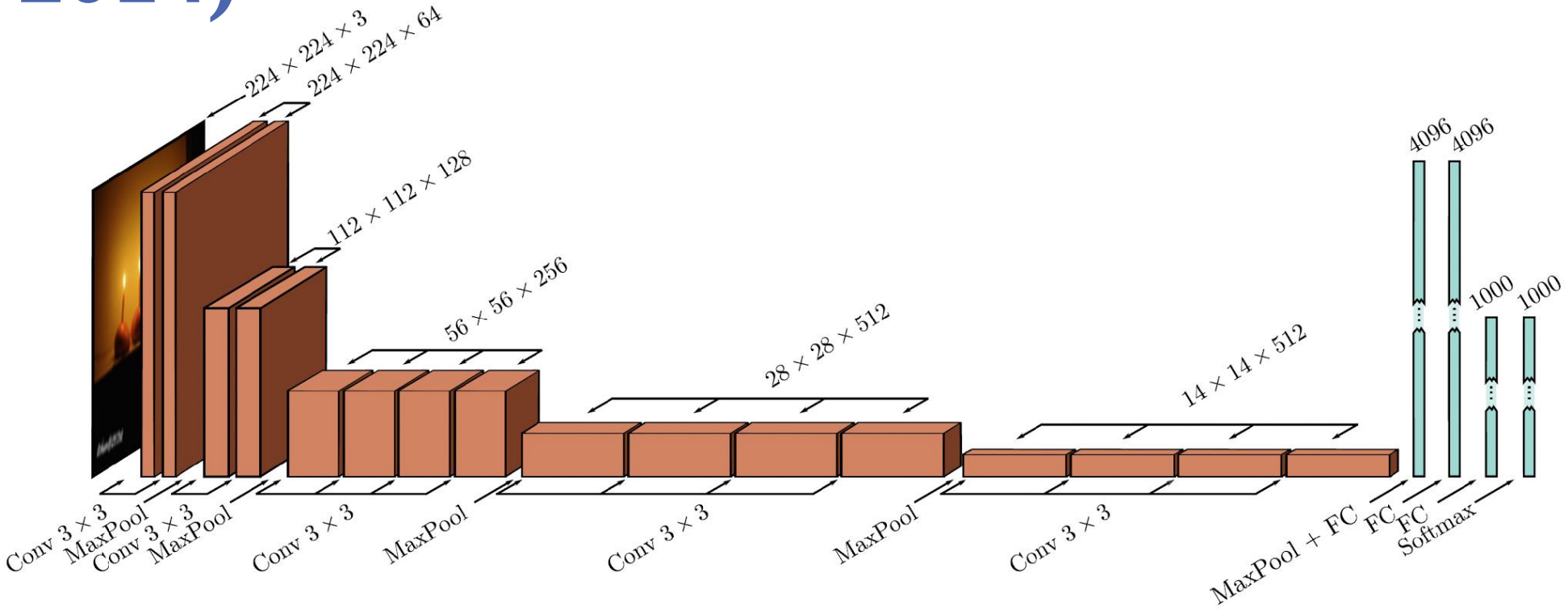


- Fully Connected Layer 1 (FC1):
  - Neurons: 4096
- Fully Connected Layer 2 (FC2):
  - Neurons: 4096
- Output Layer (FC3):
  - Neurons: 1000

    (for ImageNet classes)

Source: Figure 10.16 in Understanding Deep Learning by Simon J.D. Prince.

VGG Network (Simonyan & Zisserman, 2014)



This network consists of a series of convolutional layers and max pooling operations, in which the spatial scale of the representation gradually decreases, but the number of channels gradually increases. The hidden layer after the last convolutional operation is resized to a 1D vector and three fully connected layers follow. The network outputs 1000 activations corresponding to the class labels that are passed through a softmax function to create class probabilities.

Source: Figure 10.17 in Understanding Deep Learning by Simon J.D. Prince.

Why VGG Network

- Simplicity and uniformity: VGG uses only 3×3 convolutional layers stacked on top of each other in increasing depth.
- Transferability: Features learned by VGG generalize well to other tasks.
- Benchmark: Served as a strong baseline for many subsequent architectures.

Overview of VGG Architecture

- Two main variants: VGG-16 and VGG-19, denoting the number of weight layers.
- Architecture: Conv layers with small receptive fields, followed by FC layers.

Recurrent Neural Networks

- Introduction
- Core Concepts
- Architectures

Example – Slot Filling

- Slot filling is a task in Natural Language Processing (NLP) where the goal is to identify and extract specific pieces of information from a given text and categorize them into predefined slots or categories.
- This task is part of a broader field known as Information Extraction (IE).

I will arrive in Taoyuan on November 10th.

In this scenario, the slots to be filled might include:

- Arrival City: Taoyuan
- Departure Date: November 10th

Solve Slot Filling by FNN

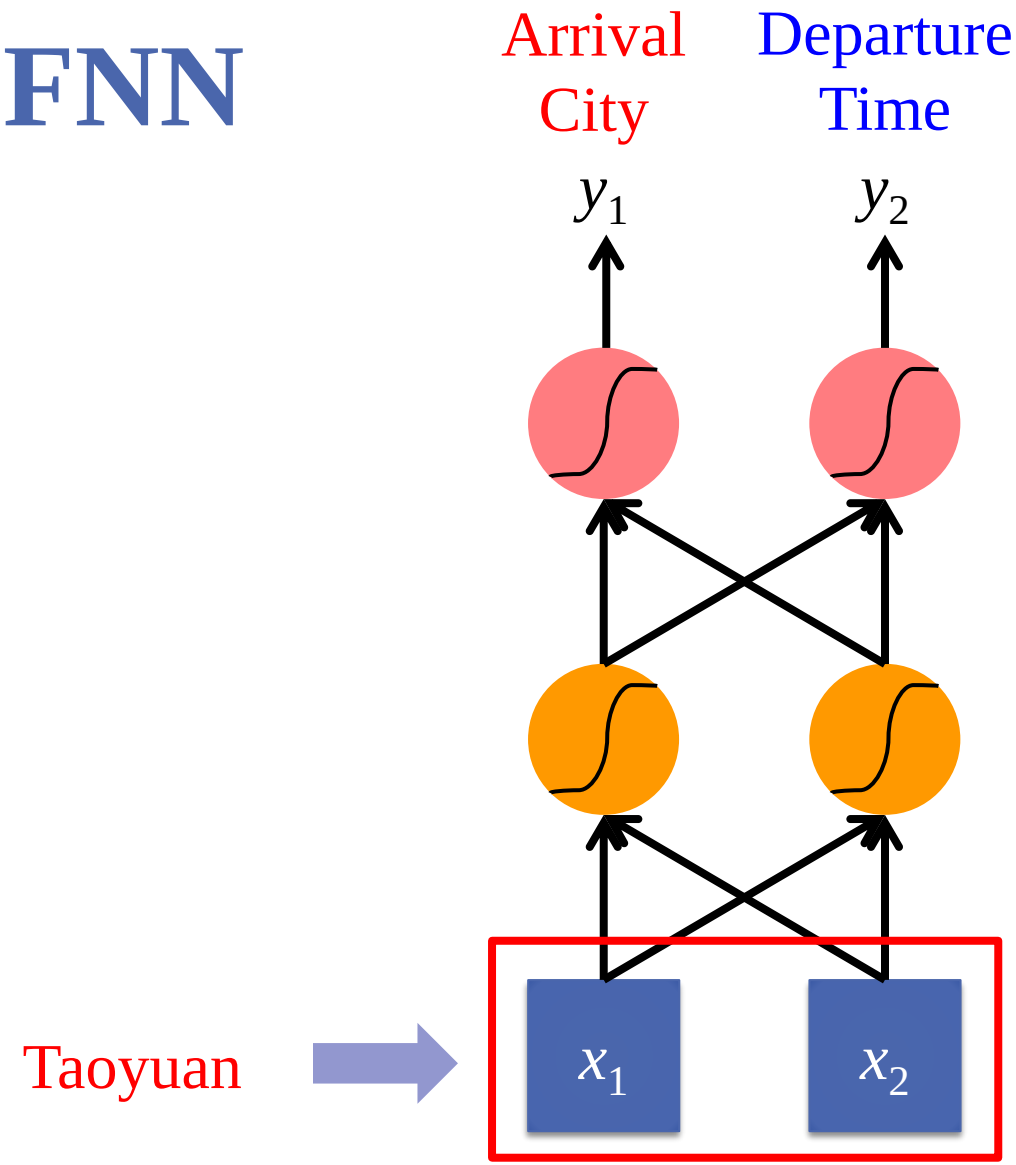
- Input:

  a word

  (Each word is represented as a vector)

- Output:

  Probability distribution that the input word belonging to the slots



Solve Slot Filling by FNN

I will arrive in Taoyuan on November 10th.

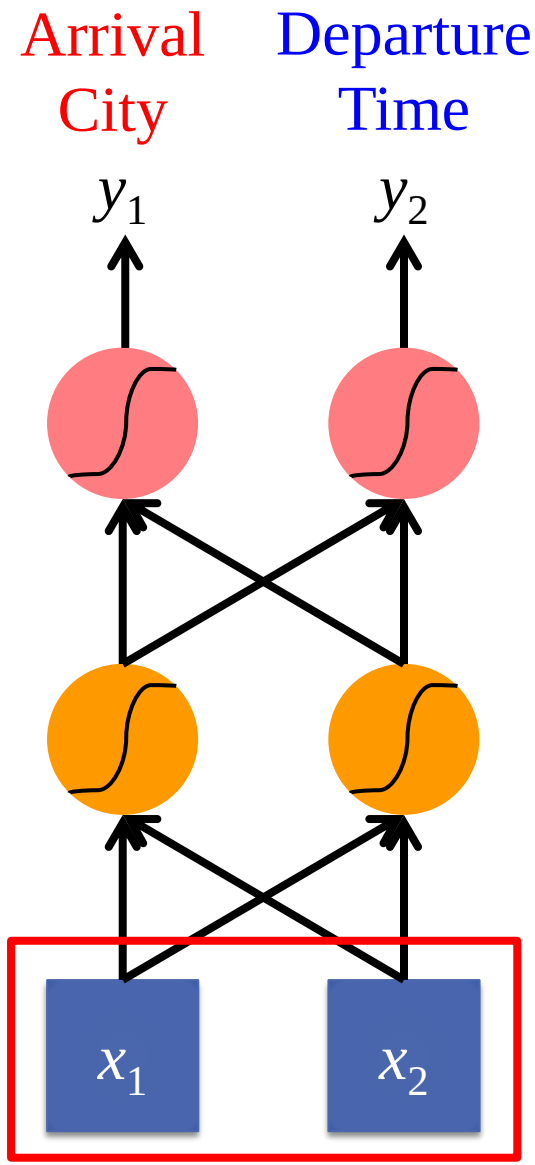
Other | Arrival City | Departure Time | Departure Time

Problem?

I want to leave Taoyuan on November 10th.

Departure City

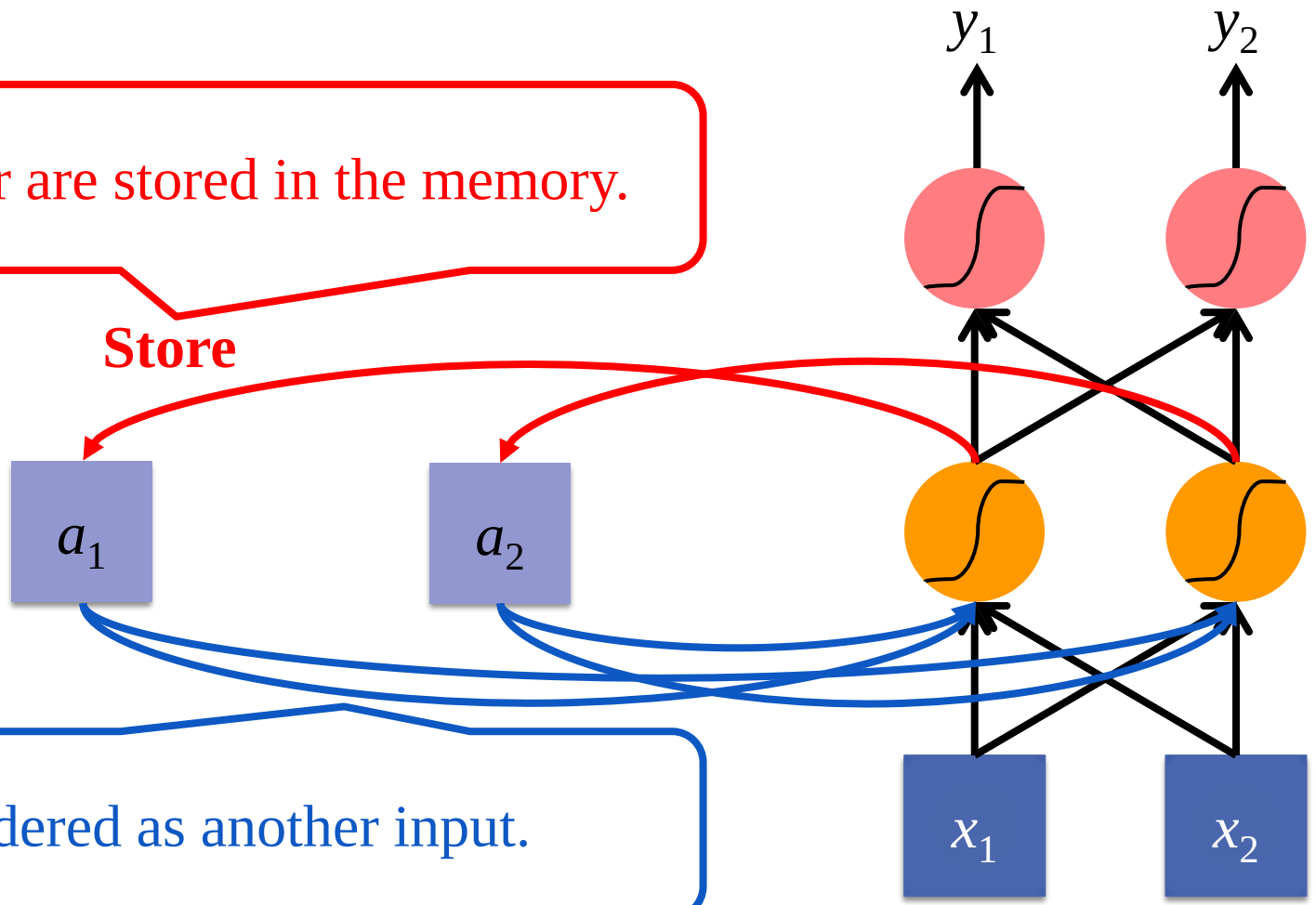
Neural network needs memory!



Taoyuan

Recurrent Neural Network (RNN)

The output of hidden layer are stored in the memory.



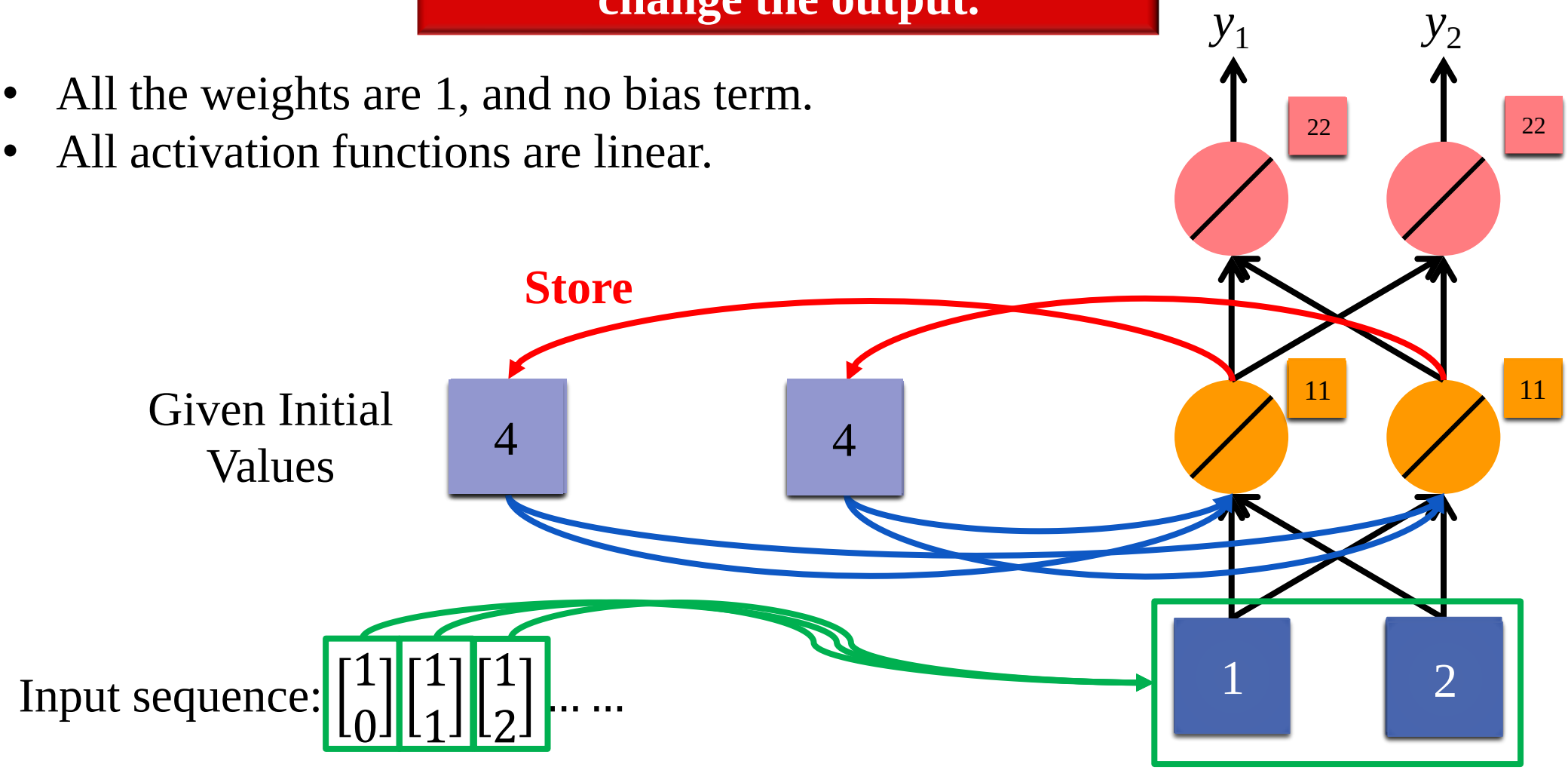
Memory can be considered as another input.

Output sequence: $\begin{bmatrix}2\\2\end{bmatrix}\begin{bmatrix}8\\8\end{bmatrix}\begin{bmatrix}22\\22\end{bmatrix}$

Example

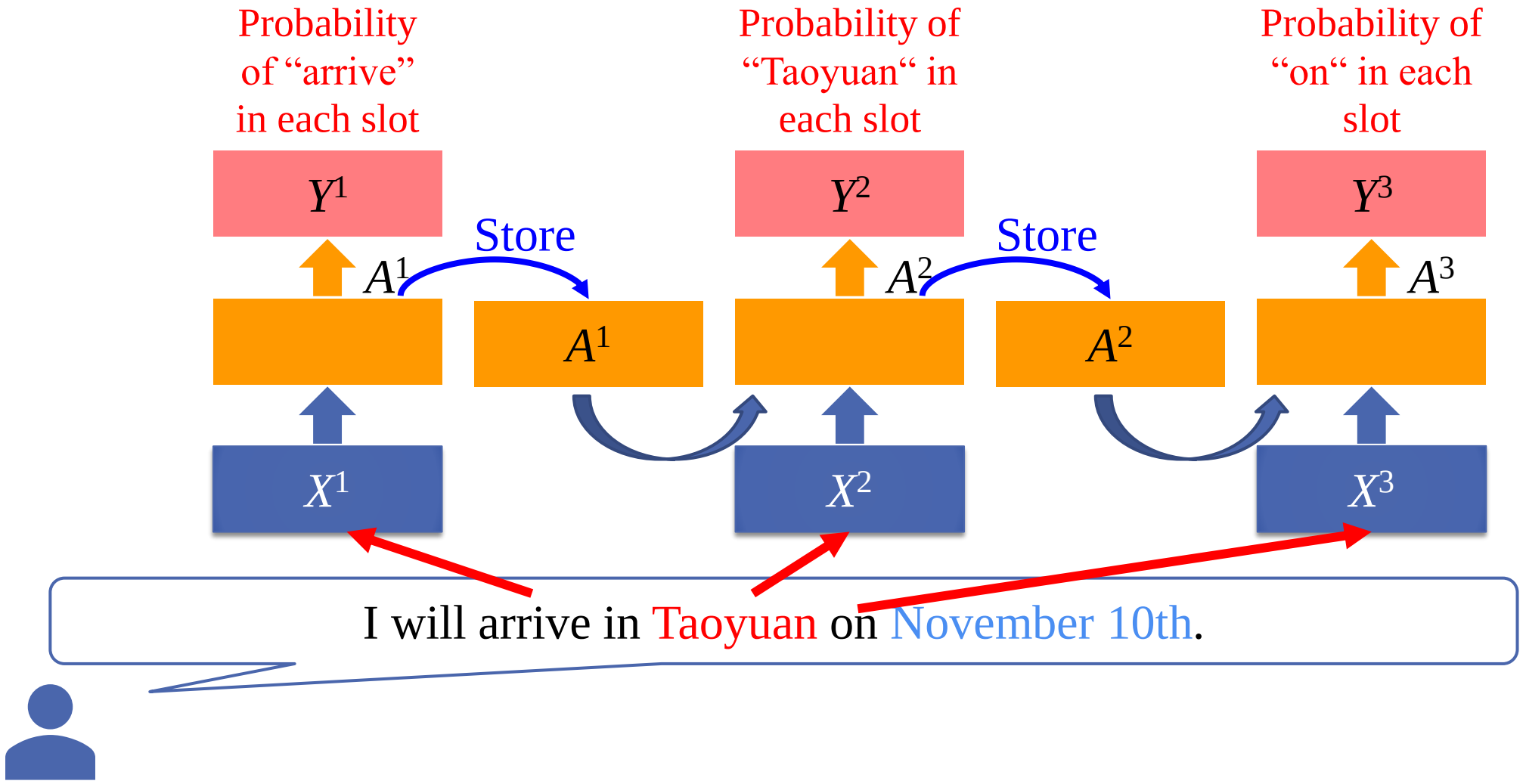
Changing the sequence order will change the output.

- All the weights are 1, and no bias term.
- All activation functions are linear.

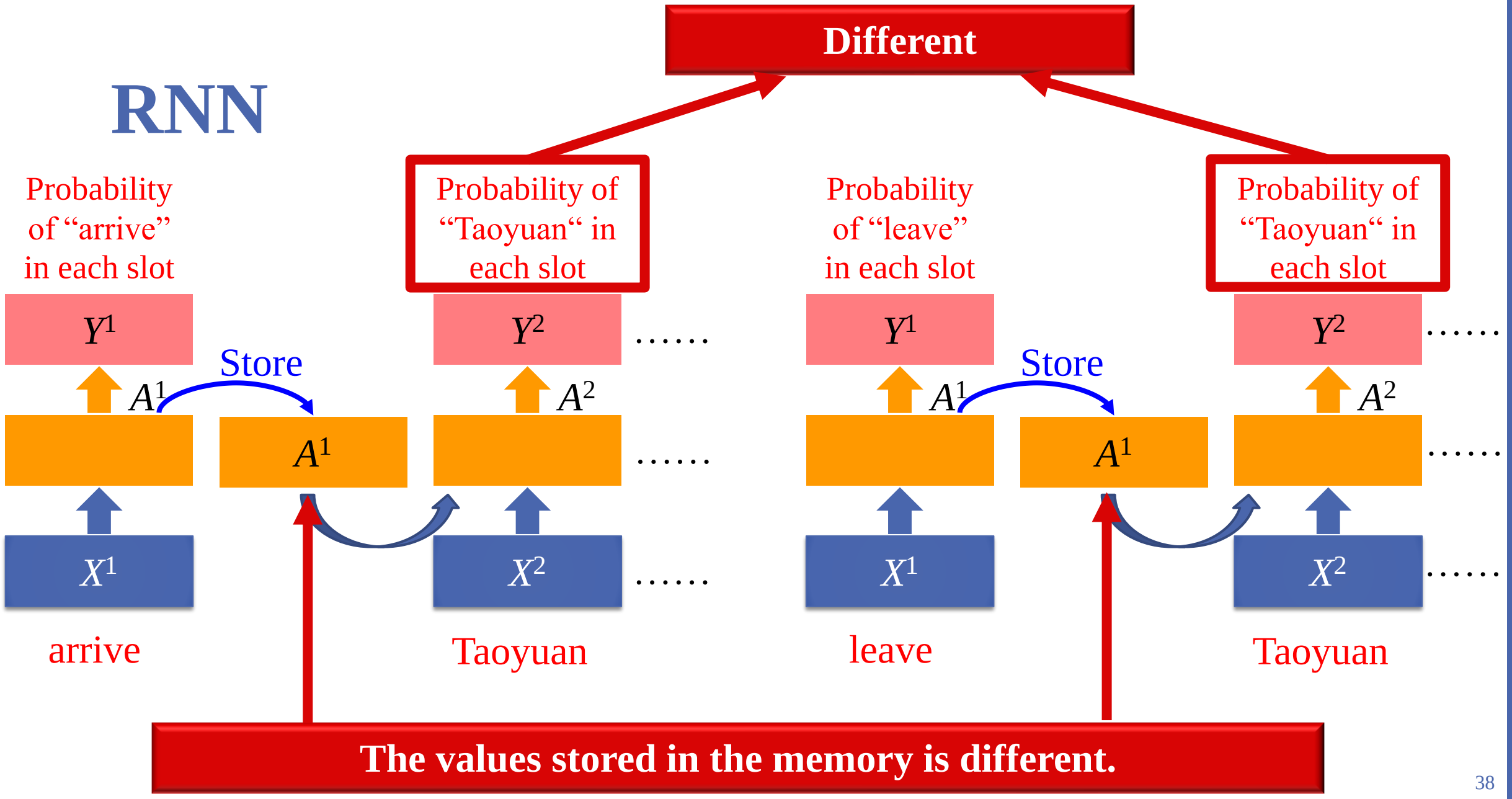


RNN

The same network is used again and again.



RNN


Different
Probability of "arrive" in each slot
Probability of "Taoyuan" in each slot
Probability of "leave" in each slot
Probability of "Taoyuan" in each slot
Y^1
Y^2
Y^1
Y^2
Store
Store
A^1
A^2
A^1
A^1
A^2
X^1
X^2
X^1
X^2
arrive
Taoyuan
leave
Taoyuan
The values stored in the memory is different.

Introduction (1/2)

- Recurrent Neural Networks (RNNs) are a class of artificial neural networks designed to recognize patterns in sequences of data, such as text, genomes, handwriting, and spoken words.
- Unlike traditional neural networks, RNNs possess loops that allow information to be carried across neurons while processing sequences, thereby enabling a form of memory to retain past information.

Introduction (2/2)

- The primary motivation behind the development of RNNs is to process sequential and temporal data, where the order and context of data points play a crucial role in understanding the underlying patterns.
- Traditional neural networks lack the capability to handle such dependencies, making them unsuitable for tasks like time series prediction or natural language processing where sequential information is crucial.

Differences between Feedforward Neural Networks (FNNs) and RNNs (1/4)

- Data Flow
  - FNNs: Data flows in one direction from the input layer to the output layer, with no cycles or loops.
  - RNNs: They have loops that allow information to flow from one step in the sequence to the next, enabling a form of memory.

Differences between Feedforward Neural Networks (FNNs) and RNNs (2/4)

- Handling of Sequential Data:
  - FNNs: They are not designed to handle sequential data and treat each input independently.
  - RNNs: They are specifically designed to work with sequential data by maintaining a hidden state that can capture information about previous steps.

Differences between Feedforward Neural Networks (FNNs) and RNNs (3/4)

- Memory:
  - FNNs: Lack an internal memory to capture information about previous steps in a sequence.
  - RNNs: Have an internal memory which allows them to capture and utilize information from earlier in the input sequence.

Differences between Feedforward Neural Networks (FNNs) and RNNs (4/4)

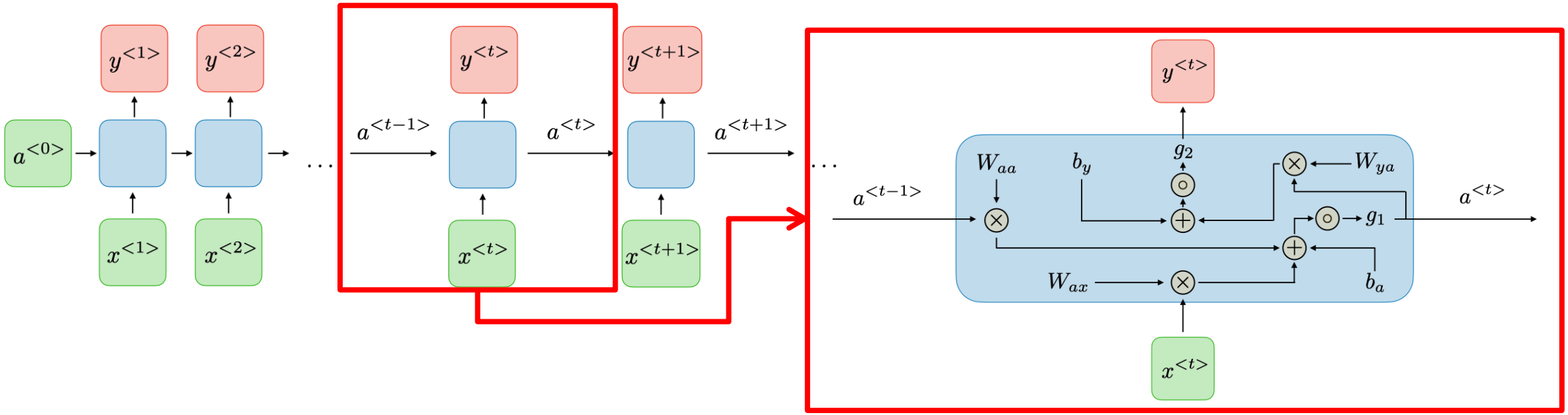
- Training Algorithm:
  - FNNs: Trained using standard backpropagation.
  - RNNs: Trained using Backpropagation Through Time (BPTT), a variant of backpropagation suited for sequences.

	Feedforward Neural Networks	Convolutional Neural Networks	Recurrent Neural Networks
Primary Usage	General-purpose	Image and video processing	Sequential data processing
Data Flow	One-way (from input to output)	One-way (from input to output)	Loops (feedback connections)
Memory	None	None	Internal state (sequence memory)
Layer Connectivity	Fully connected	Locally connected with shared weights	Connections through time
Training Algorithm	Backpropagation	Backpropagation	Backpropagation Through Time (BPTT)
Key Feature	Simplicity	Convolutional layers and pooling	Recurrent loops
Examples	Basic classification tasks	Image recognition, object detection	Text generation, sentiment analysis

Applications of RNNs

- Natural Language Processing (NLP):
  - Text Generation: Generating text based on a given seed text.
  - Sentiment Analysis: Analyzing the sentiment of text data.
  - Machine Translation: Translating text from one language to another.
  - Speech Recognition: Converting spoken language into written text.
- Time Series Analysis:
  - Predicting stock prices, weather forecasting, and any other time-dependent phenomena.
- Healthcare:
  - Analyzing sequential medical data to predict disease onset or other health-related predictions.

Architecture of a Traditional RNN



For each timestep t, the activation a<t> and the output y<t> are expressed as follows:

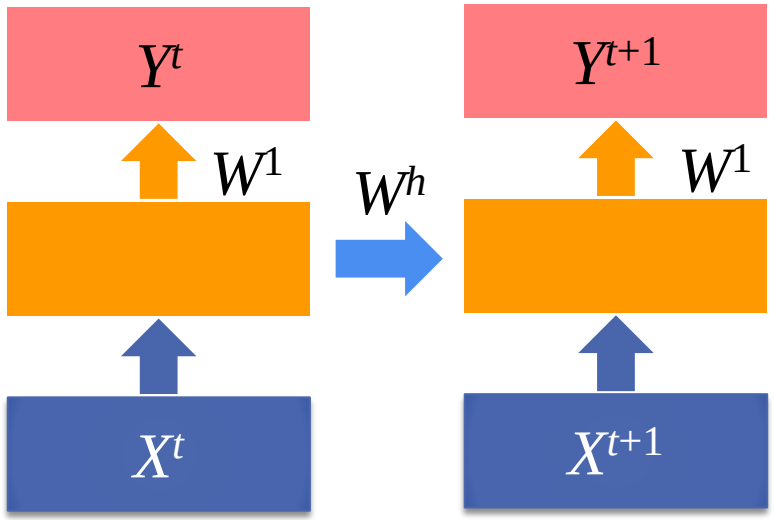
$$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}a^{<t>} + b_a \text{ and } y^{<t>} = g_2(W_{ya}a^{<t>} + b_y$$

where W_{ax}, W_{aa}, W_{ay}, b_a, b_y, are coefficients that are shared temporally and g_1, and g_2 activations functions

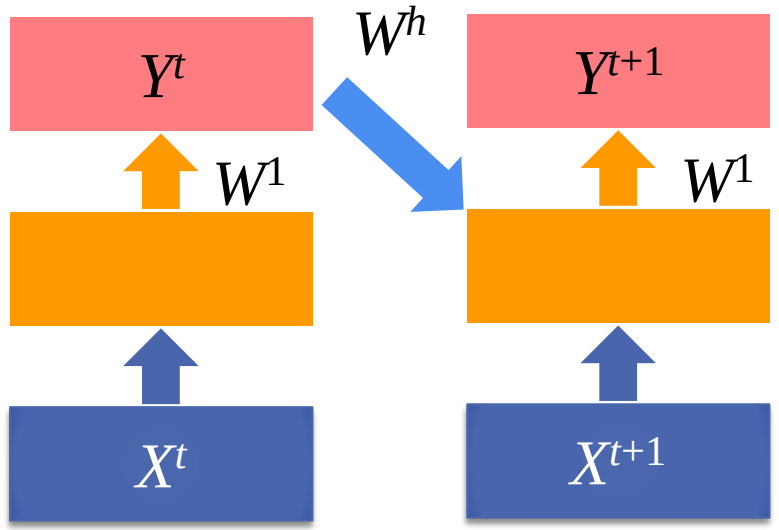
Source: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks

Elman Network & Jordan Network

Elman Network



Jordan Network



Backpropagation Through Time (1/7)

- 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.

Backpropagation Through Time (2/7)

- Sequence Processing:
  - RNNs process sequences of data one step at a time, maintaining a hidden state that carries information from one step to the next.
  - Each step involves computing an output and *a new hidden state* based on the current input and the previous hidden state.

Backpropagation Through Time (3/7)

- Error Calculation:
  - Like other neural networks, RNNs are trained to minimize the error between their predicted outputs and the actual target outputs for a given set of inputs.
  - The error for a sequence is often computed as the sum of the errors at each step of the sequence.

Backpropagation Through Time (4/7)

- Unrolling the Network:
  - Before applying BPTT, the RNN is “unrolled” over time, creating a separate copy of the network for each step of the sequence.
  - This unrolled network provides a clear structure for computing and backpropagating the error.

Backpropagation Through Time (5/7)

- Forward Pass:
  - The forward pass involves processing the input sequence through the unrolled network, computing the output and hidden state at each step.
  - The predicted output and the hidden state at each step are stored as they will be used during the backward pass.

Backpropagation Through Time (6/7)

- Error Backpropagation:
  - The backward pass in BPTT involves computing the gradient of the error with respect to the network's weights, starting from the final step of the sequence and working backward to the first step.
  - The gradients are computed based on the chain rule of calculus, which involves propagating the error gradients back through the network, step by step, and layer by layer.

Backpropagation Through Time (7/7)

- Weight Updates:
  - Once the gradients have been computed, the weights of the network are updated to minimize the error.
  - The weight updates can be performed at each step, or they can be accumulated across the sequence and applied at the end.

Vanishing/Exploding Gradient

- In the vanishing gradient problem, the gradients of the loss function with respect to the model's parameters become very small, often approaching zero.
- In the exploding gradient problem, the gradients become very large, which can cause the model parameters to be updated with very large values.

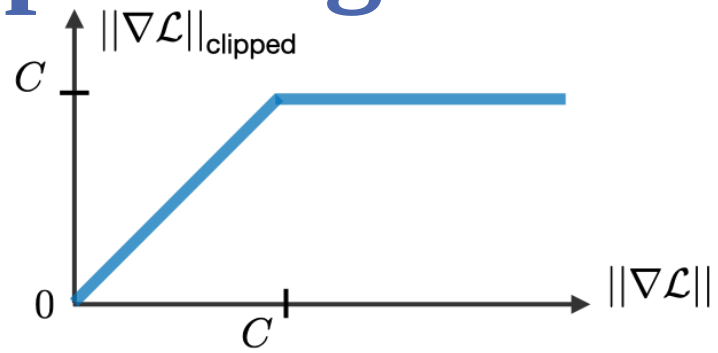
Why does Vanishing/Exploding Gradient Occur in RNN (1/2)

- Sequential Dependency: RNNs are trained to capture dependencies across sequences of data. When sequences are long, the network needs to backpropagate gradients through many time steps, which can lead to vanishing or exploding gradients.
- Activation Functions: The choice of activation function can contribute to the vanishing gradient problem. For instance, traditional activation functions like sigmoid or tanh have regions where their gradients are very small.

Why does Vanishing/Exploding Gradient Occur in RNN (2/2)

- Weight Initialization: If weights are initialized improperly, they can either squash the gradients (leading to vanishing gradients) or amplify them (leading to exploding gradients) as they are backpropagated through time.
- Recurrent Nature: The recurrent nature of RNNs involves repeatedly applying the same set of weights at each time step. If these weights are such that they squash or amplify the gradients, the problem gets compounded over many time steps.

How to Mitigate Vanishing/Exploding Gradient in RNN (1/4)



- Gradient Clipping:
  - For exploding gradients, a common remedy is gradient clipping, where gradients are scaled down if they exceed a certain threshold, preventing them from growing too large.
- Improved Initialization:
  - Proper initialization of the weights can also prevent gradients from vanishing or exploding initially.

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

How to Mitigate Vanishing/Exploding Gradient in RNN (2/4)

- Gated Recurrent Units (GRUs) and Long Short-Term Memory (LSTM) Networks:
  - These are special types of RNNs designed to handle long-term dependencies by maintaining a more stable hidden state across time steps.
  - They introduce gating mechanisms that control the flow of information, making it easier for the network to carry information across many time steps, thus mitigating the vanishing gradient problem.

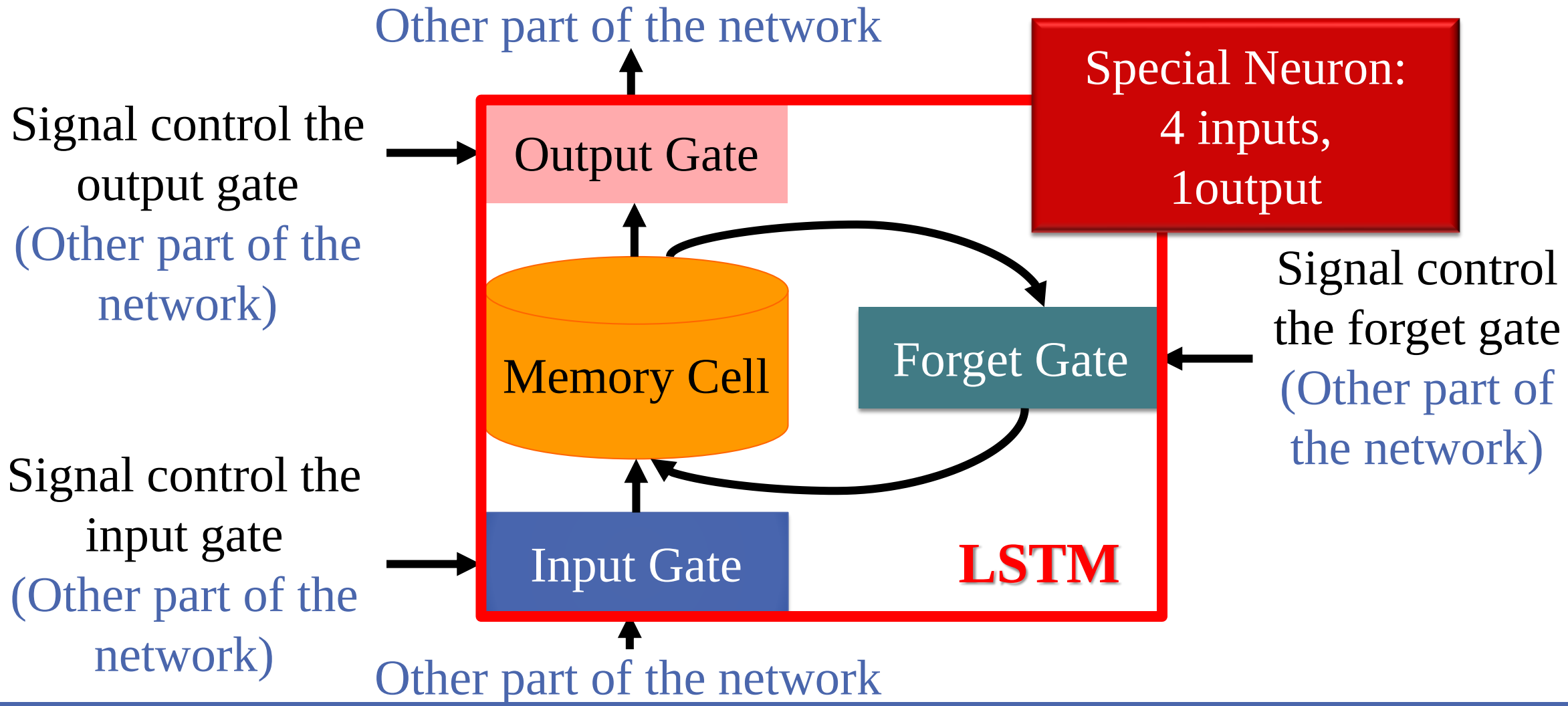
How to Mitigate Vanishing/Exploding Gradient in RNN (3/4)

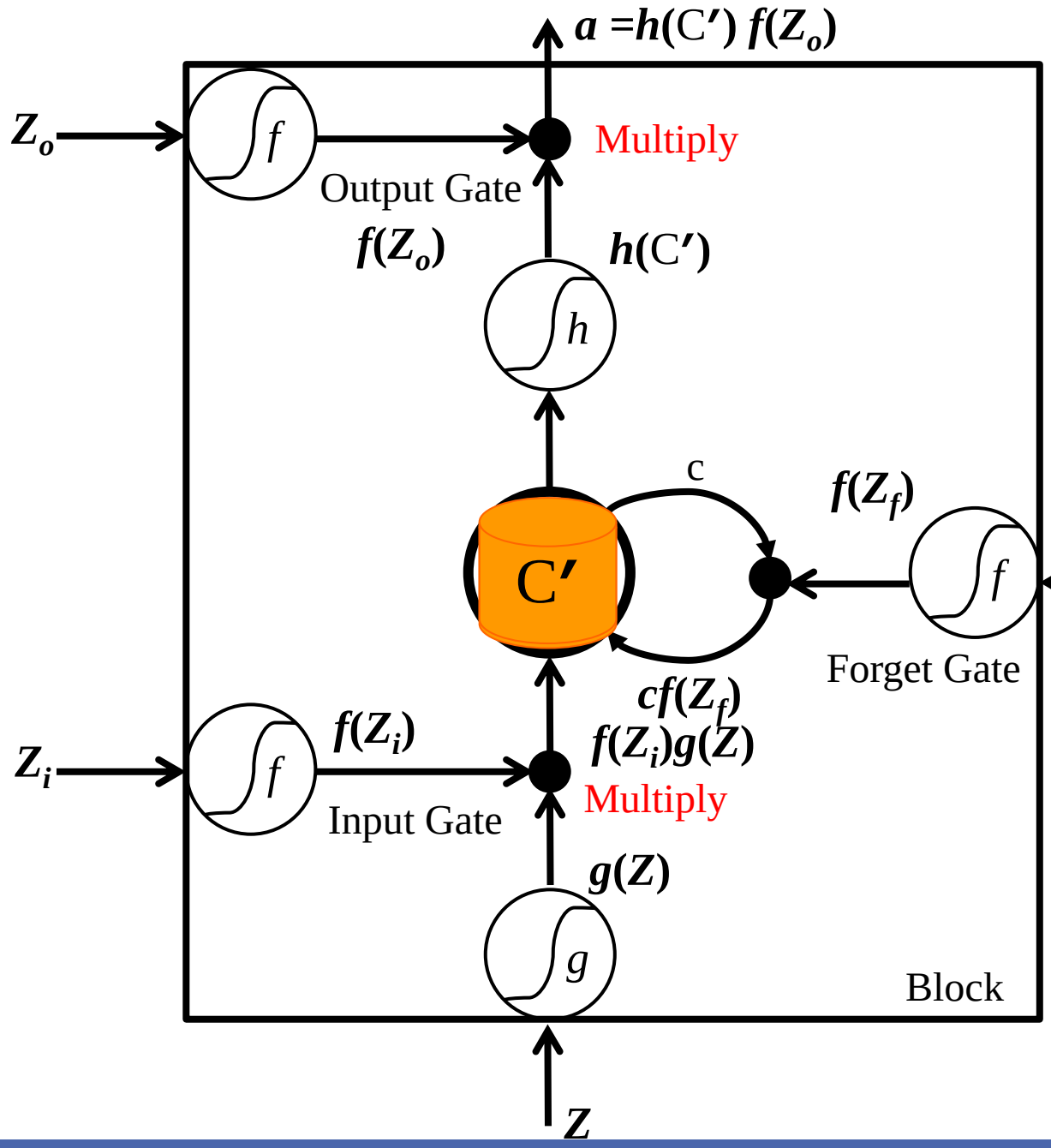
- Identity Initialization:
  - Initializing the recurrent weights as identity matrices or orthogonal initialization can help in reducing the vanishing and exploding gradient problems by ensuring that the recurrent layer starts off as approximately linear.
- Using ReLU Activation Function:
  - ReLU and its variants (like Leaky ReLU, Parametric ReLU) can mitigate the vanishing gradient problem as they do not squash the gradients.

How to Mitigate Vanishing/Exploding Gradient in RNN (4/4)

- Regularization:
  - Applying regularization techniques can also help in keeping the magnitude of the weights and hence the gradients in check.

Long Short-Term Memory (LSTM)

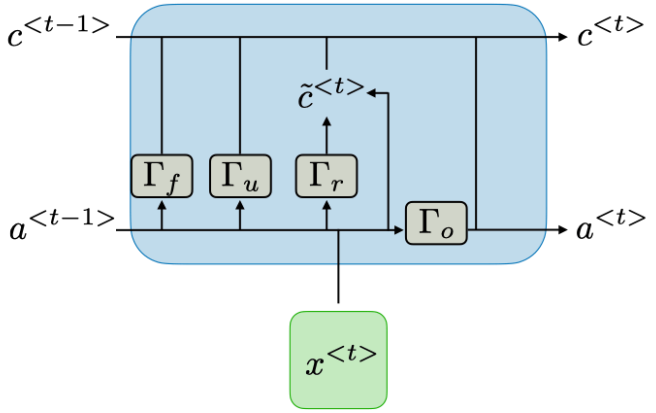




- Activation function *f* is usually sigmoid function
- Between 0 and 1
- Mimic open and close gate

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

What is LSTM



- Long Short-Term Memory (LSTM) is a type of Recurrent Neural Network (RNN) designed to overcome the vanishing gradient problem, which hampers the training and performance of traditional RNNs, especially over long sequences of data.
- LSTMs have a more complex cell structure that includes a memory cell and three multiplicative gates: the input, forget, and output gates.

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

Why is LSTM Needed (1/2)

- Vanishing Gradient Problem:
  - Standard RNNs struggle with the vanishing gradient problem during training, which makes learning long-term dependencies in data sequences challenging.
  - This problem arises due to the repeated multiplication of gradients through many time steps during backpropagation, leading to exponentially shrinking gradients as the sequence length increases.

Why is LSTM Needed (2/2)

- Long-term Dependencies:
  - In many real-world sequence processing tasks like language modeling, information from earlier parts of a sequence may be crucial for processing later parts.
  - Standard RNNs find it hard to maintain and use such long-term dependencies due to the vanishing gradient problem.

How Does LSTM Work (1/4)

- Memory Cell:
  - At the heart of the LSTM is a memory cell that can maintain its state over time, providing a way to represent and carry forward information through the sequence.

How Does LSTM Work (2/4)

- Gates:
  - Input Gate: Controls how much of the incoming information should be stored in the memory cell.
  - Forget Gate: Determines how much of the existing memory should be discarded.
  - Output Gate: Controls how much of the internal state should be exposed to the next layers in the network.

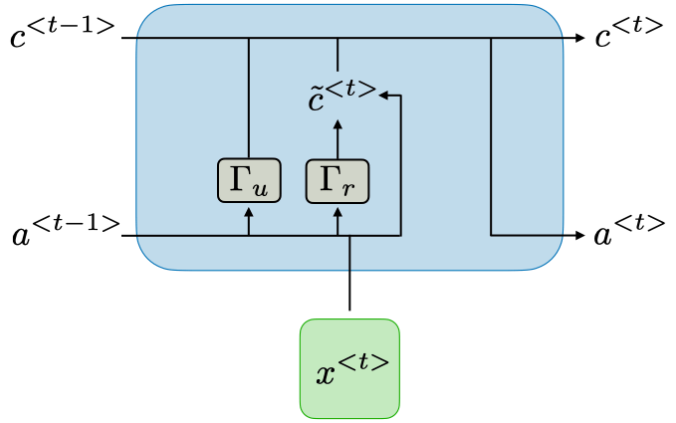
How Does LSTM Work (3/4)

- Gate Operations:
  - The operations of these gates are learned from the data during training, allowing the LSTM to learn how to use its memory to carry forward relevant information through time.
- Cell State Update:
  - The cell state gets updated in a way that avoids the vanishing gradient problem, allowing for learning over longer sequences.

How Does LSTM Work (4/4)

- Training:
  - LSTMs are trained using Backpropagation Through Time (BPTT), similar to standard RNNs, but their special architecture makes the backpropagation process more effective, even over long sequences.

What is GRU



- Gated Recurrent Units (GRU) are a variant of Recurrent Neural Networks (RNNs) designed to address the vanishing gradient problem and make it easier to model long-term dependencies in sequence data.
- GRUs simplify the LSTM architecture while retaining its ability to handle long-term dependencies.

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

Why is GRU Needed (1/2)

- Vanishing Gradient Problem:
  - Traditional RNNs suffer from the vanishing gradient problem, which hinders their ability to learn from long sequences of data.
- Long-term Dependencies:
  - In tasks like language modeling or time series analysis, it's crucial to capture dependencies across long spans of the input data. Traditional RNNs struggle with this, leading to the development of gated architectures like LSTMs and GRUs.

Why is GRU Needed (2/2)

- Simplification of LSTMs:
  - While LSTMs are effective, their architecture is relatively complex. GRUs provide a simpler alternative that retains much of the capability of LSTMs with fewer parameters.

How Does GRU Work (1/5)

- Gating Mechanism:
  - Reset Gate: The reset gate helps the model to decide how much of the past information to forget.
  - Update Gate: The update gate helps the model to determine how much of the new information to store.

How Does GRU Work (2/5)

- Hidden State:
  - Unlike LSTMs, which have a separate cell state and hidden state, GRUs have a single hidden state that captures the relevant information from past inputs.

How Does GRU Work (3/5)

- Formulation:
  - The computation of the new hidden state in a GRU involves a combination of the reset gate, the previous hidden state, and the current input.
  - The update gate determines the mix of the previous hidden state and the candidate hidden state to produce the new hidden state.

How Does GRU Work (4/5)

- Training:
  - Like LSTMs, GRUs are trained using the Backpropagation Through Time (BPTT) algorithm. The gating mechanisms ensure that gradients can flow effectively through long sequences during training.

How Does GRU Work (5/5)

- Efficiency:
  - The simpler architecture of GRUs makes them computationally more efficient than LSTMs, often leading to faster training times.
  - They also require fewer parameters, which could lead to a more straightforward optimization landscape.

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.

Hand-on

- **RNN**
- **LSTM**
- **GRU**

RNN

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

# Generate synthetic data
np.random.seed(0)
x = np.linspace(0, 100, 1000)
y = np.sin(x)

# Prepare input and output data for training
input_data = [y[i:i+10] for i in range(len(y)-10)]
output_data = y[10:]

input_data = np.array(input_data)
output_data = np.array(output_data)

# Reshape input_data to match RNN input shape (batch_size, sequence_length, feature_dim)
input_data = np.reshape(input_data, (input_data.shape[0], input_data.shape[1], 1))

# Manually split data into training and testing sets
split_idx = int(0.8 * len(input_data))  # 80% training, 20% testing
x_train, x_test = input_data[:split_idx], input_data[split_idx:]
y_train, y_test = output_data[:split_idx], output_data[split_idx:]

# Build the RNN model
model = keras.Sequential([
    layers.SimpleRNN(10, activation='relu', input_shape=(10, 1)),
    layers.Dense(1)
])

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
history = model.fit(x_train, y_train, epochs=10, batch_size=32, validation_data=(x_test, y_test))

# Evaluate the model on the testing set
y_pred = model.predict(x_test)
test_mse = np.mean((y_test - y_pred)**2)
print(f'Test Mean Squared Error: {test_mse}')
```

LSTM

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

# Generate synthetic data
np.random.seed(0)
x = np.linspace(0, 100, 1000)
y = np.sin(x)

# Prepare input and output data for training
input_data = [y[i:i+10] for i in range(len(y)-10)]
output_data = y[10:]

input_data = np.array(input_data)
output_data = np.array(output_data)

# Reshape input_data to match RNN input shape (batch_size, sequence_length, feature_dim)
input_data = np.reshape(input_data, (input_data.shape[0], input_data.shape[1], 1))

# Manually split data into training and testing sets
split_idx = int(0.8 * len(input_data))  # 80% training, 20% testing
x_train, x_test = input_data[:split_idx], input_data[split_idx:]
y_train, y_test = output_data[:split_idx], output_data[split_idx:]

# Build the LSTM model
model = keras.Sequential([
    layers.LSTM(10, activation='relu', input_shape=(10, 1)),
    layers.Dense(1)
])

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
history = model.fit(x_train, y_train, epochs=10, batch_size=32, validation_data=(x_test, y_test))

# Evaluate the model on the testing set
y_pred = model.predict(x_test)
test_mse = np.mean((y_test - y_pred)**2)
print(f'Test Mean Squared Error: {test_mse}')
```

GRU

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

# Generate synthetic data
np.random.seed(0)
x = np.linspace(0, 100, 1000)
y = np.sin(x)

# Prepare input and output data for training
input_data = [y[i:i+10] for i in range(len(y)-10)]
output_data = y[10:]

input_data = np.array(input_data)
output_data = np.array(output_data)

# Reshape input_data to match RNN input shape (batch_size, sequence_length, feature_dim)
input_data = np.reshape(input_data, (input_data.shape[0], input_data.shape[1], 1))

# Manually split data into training and testing sets
split_idx = int(0.8 * len(input_data))  # 80% training, 20% testing
x_train, x_test = input_data[:split_idx], input_data[split_idx:]
y_train, y_test = output_data[:split_idx], output_data[split_idx:]

# Build the GRU model
model = keras.Sequential([
    layers.GRU(10, activation='relu', input_shape=(10, 1)),
    layers.Dense(1)
])

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
history = model.fit(x_train, y_train, epochs=10, batch_size=32, validation_data=(x_test, y_test))

# Evaluate the model on the testing set
y_pred = model.predict(x_test)
test_mse = np.mean((y_test - y_pred)**2)
print(f'Test Mean Squared Error: {test_mse}')
```

RNN with Regularization

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

# Generate synthetic data
np.random.seed(0)
x = np.linspace(0, 100, 1000)
y = np.sin(x)

# Prepare input and output data for training
input_data = [y[i:i+10] for i in range(len(y)-10)]
output_data = y[10:]

input_data = np.array(input_data)
output_data = np.array(output_data)

# Reshape input_data to match RNN input shape (batch_size, sequence_length, feature_dim)
input_data = np.reshape(input_data, (input_data.shape[0], input_data.shape[1], 1))

# Manually split data into training and testing sets
split_idx = int(0.8 * len(input_data))  # 80% training, 20% testing
x_train, x_test = input_data[:split_idx], input_data[split_idx:]
y_train, y_test = output_data[:split_idx], output_data[split_idx:]

# Build the RNN model with regularization
model = keras.Sequential([
    layers.SimpleRNN(10, activation='relu', input_shape=(10, 1),
                     kernel_regularizer=regularizers.l2(0.01),  # L2 regularization
                     recurrent_regularizer=regularizers.l2(0.01),
                     dropout=0.2,  # Dropout regularization
                     recurrent_dropout=0.2),
    layers.Dense(1)
])

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
history = model.fit(x_train, y_train, epochs=10, batch_size=32, validation_data=(x_test, y_test))

# Evaluate the model on the testing set
y_pred = model.predict(x_test)
test_mse = np.mean((y_test - y_pred)**2)
print(f'Test Mean Squared Error: {test_mse}')
```

Thank you