CS 412 — Introduction to Machine Learning (UIC)	April 22, 2025
Lecture 25	
Instructor: Aadirupa Saha	*Scribe(s): (Akshun Agnihotri)*

[This draft is not fully proofread. Please email any typos/errors to the instructor or directly edit the latex file.]

Overview

In the last lecture, we covered the following main topics:

1. Backpropagation (contd)
2. Recurrent Neural Networks (RNN)

This lecture focuses on:

1. The Variance Error (VE) problem in deep networks
2. Techniques to address VE: Regularization, Dropout, Gradient Clipping, ResNet
3. Long Short Term Memory (LSTM)
4. Convolution Neural Network (CNN)

1 Understanding the Variance Error (VE) Problem

Variance error arises when a model performs well on training data but fails to generalize to unseen data. This is often observed in deep neural networks trained on small datasets or using high-capacity models.
Goal: Reduce overfitting by:

- Reducing model complexity
- Adding constraints to model behavior

2 Techniques to Mitigate Variance Error

2.1 L1 and L2 Regularization

- **L1 Regularization (LASSO):** Adds $\lambda \sum |w_i|$ to the loss. Encourages sparsity.
- **L2 Regularization (Ridge):** Adds $\lambda \sum w_i^2$ to the loss. Penalizes large weights and smooths the model.

L2-Regularized Loss:

$$\text{Cost}(w, b) = \frac{1}{n} \sum_{i=1}^{n} L(y^{[i]}, \hat{y}^{[i]}) + \lambda \sum_{j} w_j^2$$

Key Effects:

- Simplifies decision boundaries
- Prevents overfitting by controlling weight magnitudes

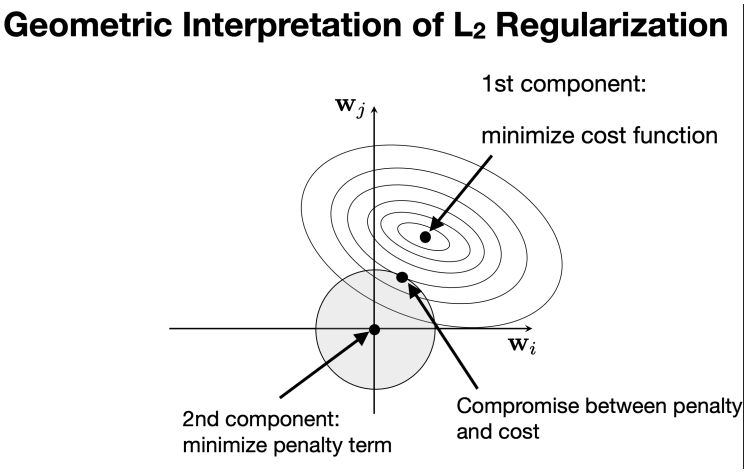

Figure 1: Geometric Interpretation of L2 Regularization. The optimum is found by balancing between minimizing the cost function and the penalty term.

2.2 Gradient Clipping

Used to prevent exploding gradients in deep or recurrent networks.
Mechanism: If $\|g\| > \tau$, then:

$$g \leftarrow \frac{\tau}{\|g\|} \cdot g$$

Impact: Stabilizes training, especially useful in LSTMs or transformers.

2.3 Dropout

Idea: Randomly deactivate a subset of neurons during training to prevent co-adaptation.
Forward pass during training:

$$a_i = a_i \cdot v_i, \quad \text{where } v_i \sim \text{Bernoulli}(1-p)$$

Forward pass during inference:

$$a_i = a_i \cdot (1-p)$$

Interpretations:

- **Co-adaptation View:** Prevents reliance on specific neurons.
- **Ensemble View:** Trains a subnetwork at each iteration; approximate model averaging at test time.

2.4 ResNet (Residual Networks)

Problem: Deep networks suffer from vanishing gradients.
Solution: Add identity (skip) connections:

$$y = \mathcal{F}(x) + x$$

Benefits:

- Improves gradient flow
- Enables training of deeper networks
- Encourages learning residual mappings instead of direct transformations

3 Main Content - 2: Long Short-Term Memory (LSTM)

3.1 Motivation for LSTM

LSTM stands for Long Short-Term Memory, and it is a type of recurrent neural network (RNN) architecture that is commonly used in natural language processing, speech recognition, and other sequence modeling tasks. Unlike a traditional RNN, which has a simple structure of input, hidden state, and output, an LSTM has a more complex structure with additional memory cells and gates that allow it to selectively remember or forget information from previous time steps.
An LSTM cell consists of several components:
Input gate: This gate controls the flow of information from the current input and the previous hidden state into the memory cell. Forget gate: This gate controls the flow of information from the previous memory cell to the current memory cell. It allows the LSTM to selectively forget or remember information from previous time steps. Memory cell: This is the internal state of the LSTM. It stores information that can be selectively modified by the input and forget gates. Output gate: This gate controls the flow of information from the memory cell to the current hidden state and output. During the forward pass, the LSTM takes in a sequence of inputs and updates its memory cell and hidden state at each time step. The input gate and forget gate use sigmoid functions to decide how much information to let into or out of the memory cell, while the output gate uses a sigmoid function and a tanh function to produce the current hidden state and output.
The LSTM's ability to selectively remember or forget information from previous time steps makes it well-suited for tasks that require modeling long-term dependencies, such as language translation or sentiment analysis.
LSTM (Long Short-Term Memory) addresses this by introducing:

- A memory cell C^t that preserves information
- Gating mechanisms to regulate what to keep, forget, and output

3.2 LSTM Cell Architecture

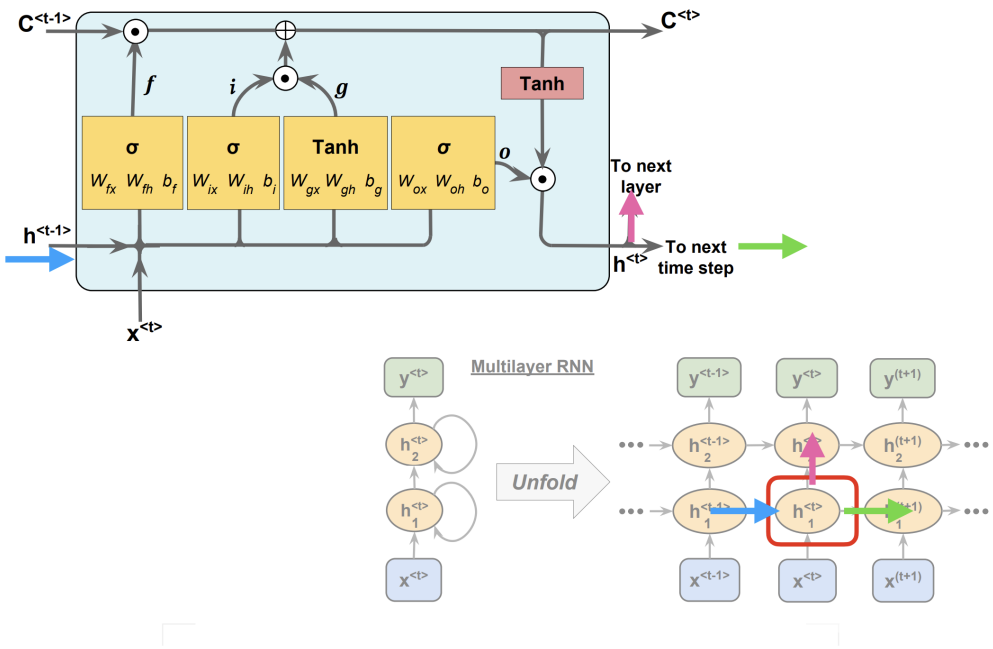

Figure 2: LSTM architecture with gating mechanisms. The diagram shows how the cell state and hidden state flow through time and across layers.

3.3 Mathematical Formulation of LSTM Gates

Let:

- x^t: Input at time t
- h^{t-1}: Hidden state from previous time step
- C^{t-1}: Cell state from previous time step

Then, each component of the LSTM cell is computed as follows:

- **Forget Gate**:
$$f^t = \sigma(W_{fx}x^t + W_{fh}h^{t-1} + b_f)$$
- **Input Gate**:
$$i^t = \sigma(W_{ix}x^t + W_{ih}h^{t-1} + b_i)$$
- **Candidate Gate** (Cell proposal):
$$g^t = \tanh(W_{gx}x^t + W_{gh}h^{t-1} + b_g)$$
- **Cell State Update**:
$$C^t = f^t \odot C^{t-1} + i^t \odot g^t$$

- **Output Gate**:
$$o^t = \sigma(W_{ox}x^t + W_{oh}h^{t-1} + b_o)$$
- **Hidden State Update**:
$$h^t = o^t \odot \tanh(C^t)$$

Here, $\odot$ represents element-wise (Hadamard) product and σ is the sigmoid activation function.

3.4 Explanation of Diagram Terms

- σ: Sigmoid function, squashes values into $[0, 1]$, used in gates.
- $\tanh$: Hyperbolic tangent function, outputs in $[-1, 1]$, used in cell proposal and final output.
- C^t: Cell state — carries memory across time steps.
- h^t: Hidden state — output of the LSTM at time t.
- $\odot$: Element-wise multiplication (shown as in diagram).
- $\oplus$: Element-wise addition (shown as in diagram).
- Blue arrows: Inputs from the previous time step x^t, h^{t-1}, C^{t-1}
- Green arrow: Outputs h^t, C^t to the next time step
- Pink arrow: Output h^t to the next layer (for stacked LSTMs)

Theorem 3.1: Key Properties of LSTM Architecture

The LSTM architecture effectively handles long-term dependencies through:

1. Persistent cell memory via additive updates, reducing vanishing gradients.
2. Gating mechanisms that allow the model to selectively store, erase, or expose information.
3. Learned temporal control, enabling better generalization across time-dependent tasks.

Exercise 3.1: Understanding LSTM Gates

Answer the following based on the LSTM gate behavior:

1. What happens if $f^t \approx 0$ and $i^t \approx 1$?
2. What does a value of $o^t \approx 0$ imply for the hidden state h^t?
3. How would setting all gate values to 1 reduce the LSTM to a vanilla RNN?

4 Gated Recurrent Neural Networks (Gated RNNs)

4.1 What Are Gated RNNs?

Vanilla RNNs face challenges like:

- **Vanishing gradients**, which prevent learning long-term dependencies
- **Exploding gradients**, which destabilize training

To address this, researchers introduced **Gated RNNs** — a broad class of recurrent architectures that use gating mechanisms to regulate the flow of information across time steps.
Examples of Gated RNNs:

- LSTM (Long Short-Term Memory)
- GRU (Gated Recurrent Unit)
- Peephole LSTM, MGU, and other variants

Note: GRU is a specific architecture within the gated RNN family. It simplifies LSTM while preserving its advantages.

5 Main Content - 4: Gated Recurrent Unit (GRU)

5.1 Motivation for GRU

LSTM uses multiple gates and a separate cell state, which increases model complexity. GRU addresses this by:

- Merging the hidden state and cell state
- Using only two gates (reset and update)
- Having fewer parameters

5.2 GRU Cell Architecture

Explanation of GRU Architecture Diagram

- $x[t]$: Input at time step t
- $h[t-1]$: Hidden state from previous time step
- $r[t]$: **Reset gate** — decides how much of the past to forget when computing the new memory content
- $z[t]$: **Update gate** — controls how much of the new candidate $\hat{h}[t]$ should overwrite the old hidden state
- $\hat{h}[t]$: **Candidate hidden state** — intermediate proposal to update $h[t]$, computed using $x[t]$, $h[t-1]$, and $r[t]$

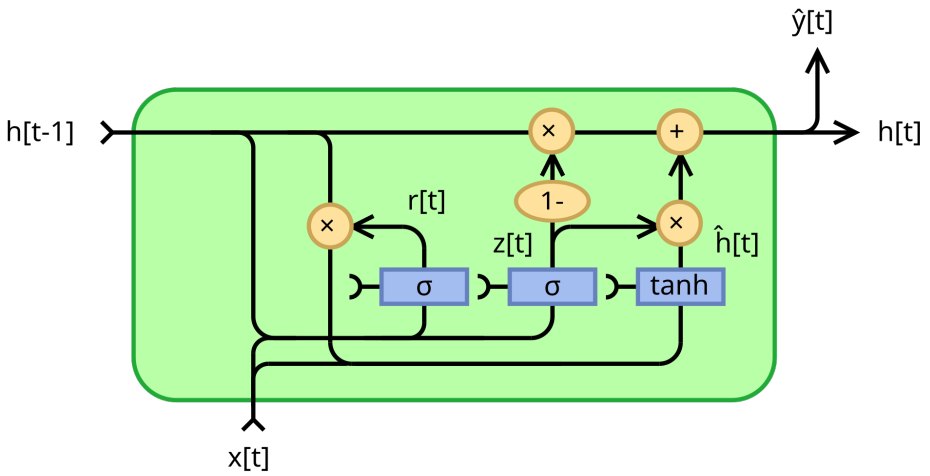

Figure 3: GRU architecture. The model uses reset and update gates to control information flow. Unlike LSTM, GRU does not maintain a separate memory cell.

- $1 - z[t]$: Weight for retaining the old hidden state
- $h[t]$: Final hidden state — linear interpolation between $h[t-1]$ and $\hat{h}[t]$
- σ: Sigmoid activation — squashes gate values between 0 and 1
- `tanh`: Hyperbolic tangent — scales intermediate state between -1 and 1
- $\odot$: Element-wise multiplication (Hadamard product)
- $+$: Element-wise addition (used to combine retained memory and candidate update)

5.3 GRU Equations and Internal Flow

Let: - x^t: input at time step t - h^{t-1}: previous hidden state - h^t: updated hidden state
Then:

- **Update Gate**:
$$z^t = \sigma(W_z x^t + U_z h^{t-1} + b_z)$$
- **Reset Gate**:
$$r^t = \sigma(W_r x^t + U_r h^{t-1} + b_r)$$
- **Candidate Hidden State**:
$$\tilde{h}^t = \tanh(W_h x^t + U_h(r^t \odot h^{t-1}) + b_h)$$
- **Final Hidden State**:
$$h^t = (1 - z^t) \odot h^{t-1} + z^t \odot \tilde{h}^t$$

5.4 Explanation of Gates

- z^t: Controls how much of the past to retain
- r^t: Controls how much of the past to reset
- $\tilde{h}^t$: Proposed update to the hidden state
- h^t: Final hidden state — a blend of old and new

5.5 Unrolled GRU and Comparison to LSTM

Theorem 5.1: Key Properties of GRU

The Gated Recurrent Unit simplifies long-term sequence modeling through:

1. Two gates: reset and update — simplifying design
2. Fewer parameters compared to LSTM
3. Faster training with similar accuracy on many tasks

Exercise 5.1: GRU vs LSTM

1. What effect does a large update gate $z^t \approx 1$ have on the hidden state update?
2. How does GRU maintain memory without a separate cell state?
3. In what types of tasks might LSTM outperform GRU, and why?

6 Convolutional Neural Networks (CNNs)

6.1 Motivation

CNNs reduce the number of parameters and allow the network to focus on local patterns (like edges, textures, shapes), making them highly effective in image classification and vision tasks.

6.2 Convolution Operation

Convolution is widely used in signal processing for bandpass filtering and noise reduction. This operation is particularly effective for filtering because it allows for the extraction of specific frequency components from a signal while suppressing others. For instance, in bandpass filtering, convolution enables the isolation of frequency bands of interest by convolving the input signal with a suitable filter kernel. The convolution of a kernel g over an input image f combines local information by adding and combining elements.

$$(f * g)(x) = \int_{-\infty}^{\infty} f(t)g(x-t)\,dt$$

In matrix form, if f is $n \times n$ and kernel g is $f \times f$, the output size is:

$$(n - f + 1) \times (n - f + 1)$$

Theorem 6.1: Convolution Output Size (No Padding)

For an input of size $n \times n$ and kernel size $f \times f$, with no padding and stride 1, the output feature map has size:

$$(n - f + 1) \times (n - f + 1)$$

Exercise 6.1: Convolution Output

Given a 7×7 input and a 3×3 filter with stride 1 and no padding, calculate the output size.

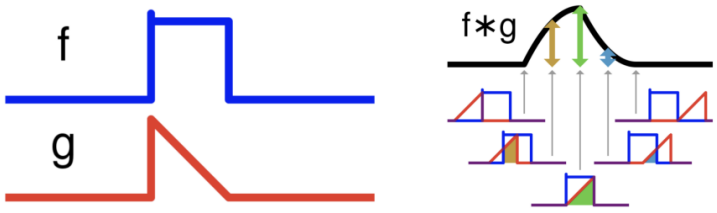

Figure 14.2: Convolution Operation

Figure 4: Example of 2D convolution using a 3×3 kernel sliding over a 5×5 input.

6.3 Example: Grayscale Matrix and Edge Detection

Let grayscale matrix:

$$f = \begin{bmatrix} 1 & 20 & 20 & 20 & 1 \\ 20 & 1 & 1 & 1 & 20 \\ 20 & 1 & 1 & 1 & 20 \\ 1 & 20 & 1 & 20 & 1 \\ 1 & 1 & 20 & 1 & 1 \end{bmatrix}$$

Vertical Edge Kernel:

$$g = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \quad \Rightarrow \quad f * g = \begin{bmatrix} 19 & 0 & -19 \\ 38 & 0 & -38 \\ 0 & 0 & 0 \end{bmatrix}$$

6.4 Padding

To retain edge information and maintain input size, we apply zero-padding.

Theorem 6.2: Padding to Preserve Output Size

To preserve spatial dimensions with a kernel of size $f \times f$, set padding p as:

$$p = \frac{f-1}{2} \quad \text{(only for odd-sized kernels)}$$

Exercise 6.2: Padding Calculation

You are convolving a 32×32 image with a 5×5 filter. What padding should you use to preserve size?

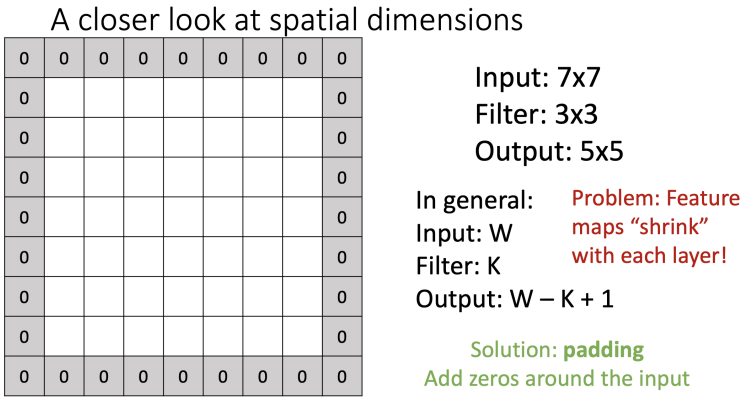

Figure 5: Zero-padding around a 5×5 input image to preserve output size during convolution.

6.5 Stride

Stride s determines how many pixels we skip after applying the filter.

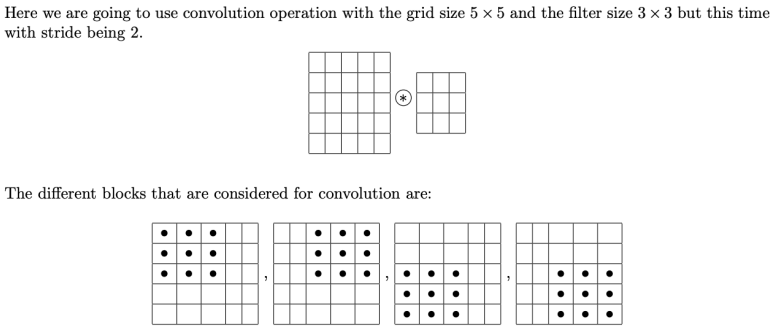

Figure 6: Stride of 2 applied over a 5×5 input with a 3×3 filter. Only blocks where filter fits completely are used.

Theorem 6.3: Output Size with Stride

For input $n_1 \times n_2$, kernel size f, and stride s, the output size is:

$$\left\lfloor \frac{n_1 - f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n_2 - f}{s} + 1 \right\rfloor$$

Exercise 6.3: Stride Application

Given a 5×5 image and a 3×3 kernel with stride 2 and no padding, compute the output size.

6.6 Pooling: Max Pooling

Max pooling reduces dimensionality by taking the maximum value from sub-regions of the feature map. The Max Pooling Layer is used to reduce the size of the input. In this, we take only the max valued pixel as per grayscale at every evaluation so that, our image sharpens/contrast increase. Even after padding and striding of the input image we may get a filtered image that is large, so we use the max pooling layer to reduce the size. Often the image pixel values changes gradually. Given this we don't need to apply the filter to consecutive blocks, instead we can skip some of the blocks which is what we call as "striding". This way we can decrease the size of the grid obtained after convolution while at the same time we are fetching the information the filter was supposed to fetch. Formally, stride is the amount of shift of the filter between two consecutive evaluations. It should be noted that striding has to be applied both vertically and horizontally.

$$\text{MaxPool} = \begin{bmatrix} m_1 & m_2 \\ m_3 & m_4 \end{bmatrix} \quad \Rightarrow \quad \text{Output} = \begin{bmatrix} \max(m_1) & \max(m_2) \\ \max(m_3) & \max(m_4) \end{bmatrix}$$

Theorem 6.4: Max Pooling Properties

For a $f \times f$ pooling filter with stride $s = f$, the output size is reduced by a factor of f^2. Max pooling retains the most salient feature in each subregion.

Exercise 6.4: Max Pooling Calculation

Apply 2×2 max pooling with stride 2 to a 4×4 matrix. What is the output size? What happens if stride = 1 instead?

6.7 Final Image from Slide 67 (Umich Notes)

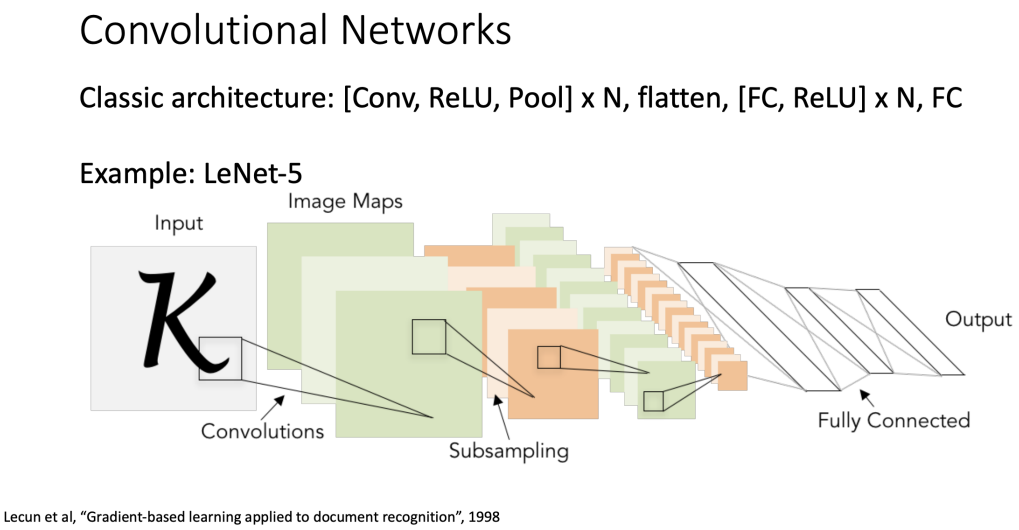

Figure 7: Final feature maps and layer visualization from slide 67 of Umich notes.

6.8 Slide 76 Example (Umich Notes)

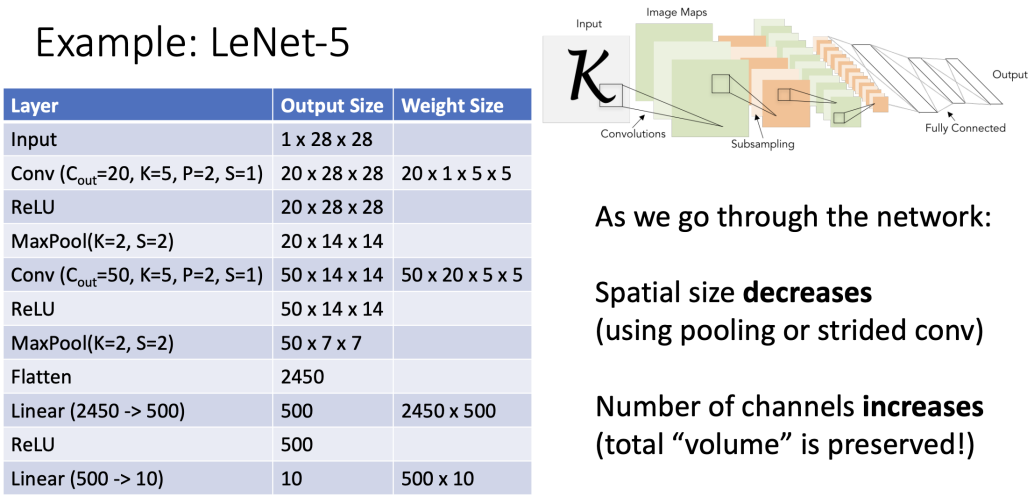

Figure 8: Example from slide 76 showing edge activation and pooling.

Next Lecture

- Properties of (Multivariate) Gaussian Distribution
- Gaussian Processes (GP): A Bayesian Regression Technique

References:

1. Lecture 14, Lecture Notes from CS217: Artificial Intelligence and Machine Learning, IIT Bombay (2024) [Link]
2. Gated Recurrent Unit (GRU) – Wikipedia [Link]
3. Introduction to Long Short-Term Memory (LSTM) – Medium [Link]
4. Long Short-Term Memory (LSTM) – CMU Deep Learning Reading [Link]
5. LSTM Explained – YouTube [Link]
6. Regularization and LSTM Training – Lecture Notes by Sebastian Raschka [Link]
7. CNN in Medical Imaging – Springer [Link]
8. CNN Lecture Slides (Slide 67 and 76) – University of Michigan [Link]
9. CNN Backpropagation – YouTube [Link]