P.V.P Siddhartha Institute of Technology (Autonomous)
Department of Computer Science and Engineering
DEEP LEARNING IV B.TECH CSE SEM-I
Descriptive-I Answers Academic Year: 2025-26

Subject Code: 20CS4701A | Regulation: PVP20 | Date: 25-08-2025 | Max Marks: 15

1(a). Outline the common architectural principles of deep networks and discuss any three common activation functions in the context of deep neural networks. (CO1-L2) (2.5M)

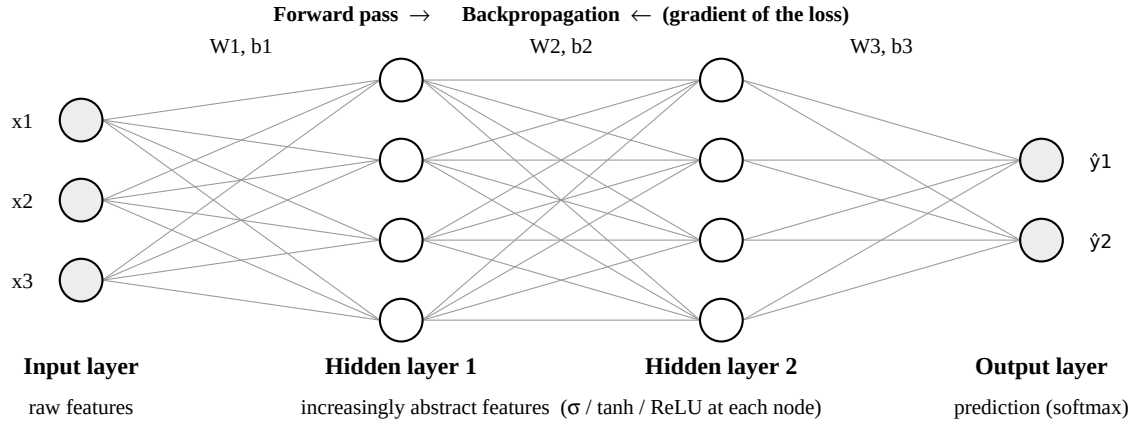
Ans:

Definition

A **deep neural network (DNN)** is an artificial neural network with an input layer, **several hidden layers** and an output layer. Each layer transforms the output of the previous layer, so the network learns a hierarchy of features directly from data. An **activation function** is a non-linear function applied to the weighted sum of a neuron's inputs; without it, any number of stacked layers would collapse into a single linear transformation.

Architecture

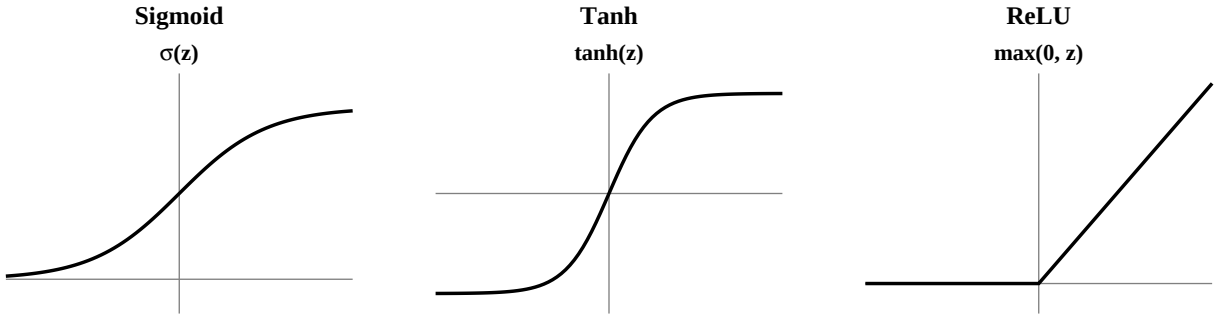
Fig. 1: General architecture of a deep neural network



Common architectural principles of deep networks

Principle	Meaning
1. Layered, hierarchical structure	Input → many hidden layers → output. Early layers learn simple patterns (edges); deeper layers combine them into complex concepts (objects).
2. Neurons with weights and bias	Each neuron computes $z = \Sigma\, w_i x_i + b$. Weights and biases are the learnable parameters.
3. Non-linear activation	$a = f(z)$ lets the network model non-linear relationships.
4. Depth and width	More layers (depth) and more units per layer (width) increase representational power, at the cost of more computation and data.
5. End-to-end training	A loss function measures error; backpropagation computes gradients and gradient descent (SGD/Adam) updates all weights together.
6. Regularization and normalization	Dropout, weight decay, early stopping and batch normalization reduce overfitting and stabilize training.
7. Parameter sharing and modularity	Reusing weights (convolution, recurrence) and stacking reusable blocks gives efficiency; design suits the data type (CNN, RNN, etc.).

Three common activation functions



Function	Formula	Range	Advantages	Limitations
Sigmoid	$\sigma(z) = 1 / (1 + e^{-z})$	(0, 1)	Smooth; output read as probability; used in binary output layer	Vanishing gradient for large |z|; not zero-centred; exponential is costly
Tanh	$\tanh(z) = (e^{z} - e^{-z}) / (e^{z} + e^{-z})$	$(-1, 1)$	Zero-centred; usually trains better than sigmoid	Still saturates → vanishing gradient
ReLU	$f(z) = \max(0, z)$	$[0, \infty)$	Very cheap; no saturation for $z > 0$; fast convergence; sparse activations	"Dying ReLU": neurons stuck at 0 for $z < 0$ (fixed by Leaky ReLU)

Example (step by step)

Take one neuron with inputs **x = (1, 2)**, weights **w = (0.5, −0.25)** and bias **b = 0.1**.

Step 1: Weighted sum: $z = (0.5)(1) + (-0.25)(2) + 0.1 = 0.5 - 0.5 + 0.1 =$ **0.1**

Step 2: Apply each activation to z = 0.1:

Activation	Calculation	Output a
Sigmoid	$1 / (1 + e^{-0.1})$	0.5250
Tanh	tanh(0.1)	0.0997
ReLU	max(0, 0.1)	0.1000

Step 3: Behaviour on other inputs:

z	−2	0	2
Sigmoid	0.1192	0.5000	0.8808
Tanh	-0.9640	0.0000	0.9640
ReLU	0	0	2

Sigmoid and tanh flatten near the extremes (small gradient), whereas ReLU stays linear for positive z.

Result and use

A deep network = stacked layers + learnable weights + non-linear activations + gradient-based training. Sigmoid is used for binary-output layers, tanh in some recurrent units, and ReLU as the default choice in hidden layers of modern deep networks.

1(b). Compare any 5 differences between Machine Learning and Deep Learning. (CO1-L2) (2.5M)

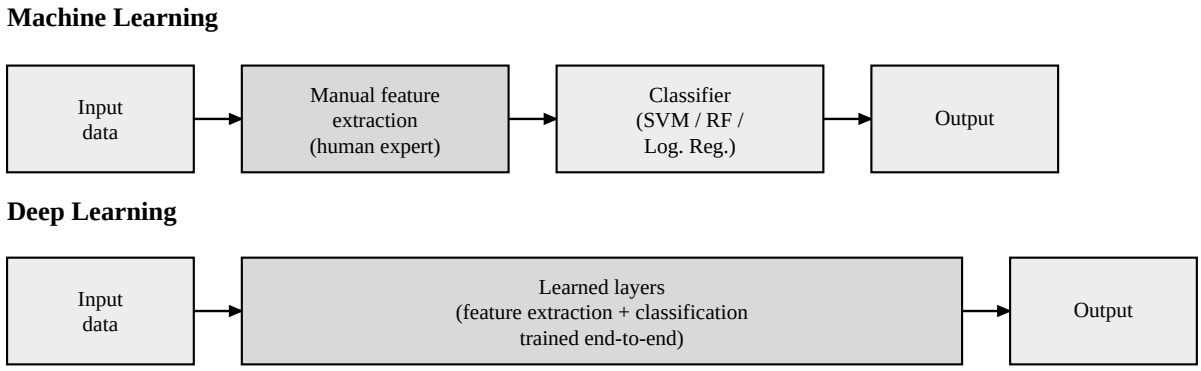
Ans:

Definition

Machine Learning (ML) is a branch of AI in which algorithms learn patterns from data to make predictions; the useful input features are usually designed by humans. **Deep Learning (DL)** is a subset of ML that uses multi-layer neural networks to learn features and the prediction rule together, directly from raw data.

Architecture (workflow comparison)

Fig. 2: ML pipeline versus DL pipeline



Differences between Machine Learning and Deep Learning

S.No	Basis	Machine Learning	Deep Learning
1	Definition / scope	Broader field; includes regression, trees, SVM, k-NN, etc.	Subset of ML based on deep neural networks
2	Feature engineering	Manual: expert designs the features (e.g. HOG, TF-IDF)	Automatic: features learned by hidden layers
3	Data requirement	Works well with small to medium datasets	Needs very large datasets to perform well
4	Hardware	Runs on an ordinary CPU	Needs GPUs/TPUs for heavy matrix computation
5	Training time	Short (seconds to minutes)	Long (hours to days)
6	Interpretability	Easier to explain (e.g. decision trees)	Black box; harder to interpret
7	Data type	Best on structured / tabular data	Excels on unstructured data: images, speech, text
8	Problem solving	Often splits the problem into sub-parts, then combines	Solves the problem end-to-end

Example (step by step) — classifying cat vs. dog images

Using ML: Step 1 collect about 1,000 labelled images. **Step 2** a human designs features (colour histogram, edge/HOG descriptors, ear-shape measurements). **Step 3** feed the feature vectors to an SVM or random forest. **Step 4** evaluate; accuracy is limited by how good the hand-made features are.

Using DL: Step 1 collect a large labelled set (e.g. 100,000 images). **Step 2** pass raw pixels into a CNN. **Step 3** early layers learn edges, middle layers learn textures and parts (ears, eyes), deep layers learn whole faces; no manual features are needed. **Step 4** train on GPU with backpropagation; accuracy is usually higher when data is plentiful.

Result and use

Choose ML for small, structured data and when explanation matters (credit scoring, sales forecasting). Choose DL for large unstructured data and complex tasks (image recognition, speech, language translation).

2(a). Construct the principle of Restricted Boltzmann Machine with neat architecture. (CO2-L3) (2.5M)

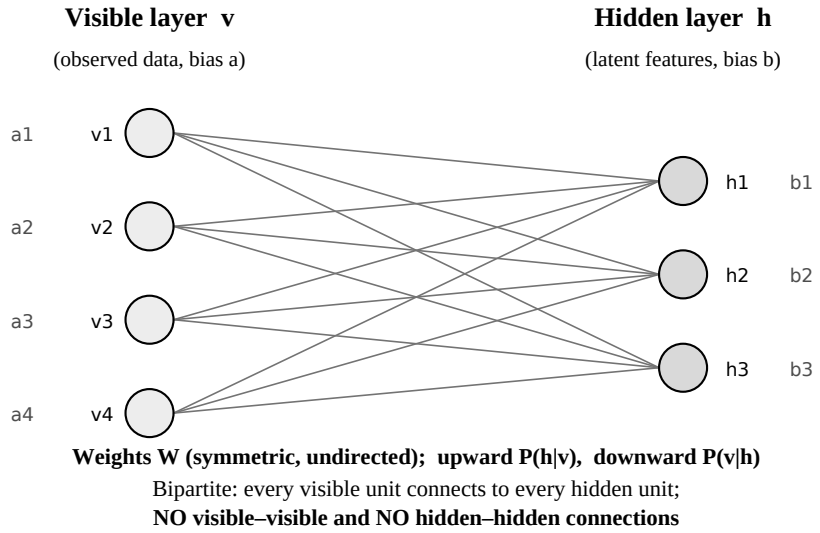
Ans:

Definition

A **Restricted Boltzmann Machine (RBM)** is a generative, energy-based stochastic neural network with two layers: a **visible layer** (the observed data) and a **hidden layer** (latent features). It is "restricted" because units are connected only **between** layers, never within a layer. It learns the probability distribution of the input data without labels (unsupervised learning), and many RBMs can be stacked to form a Deep Belief Network.

Architecture

Fig. 3: Architecture of an RBM (bipartite graph)



Parameters: weight matrix W (visible × hidden), visible biases a, hidden biases b. Units are binary (0/1) and stochastic.

Principle of working

1. Energy function. The network assigns an energy to every joint configuration; low energy means high probability:

$$E(v, h) = -\Sigma_i a_i v_i - \Sigma_j b_j h_j - \Sigma_i \Sigma_j v_i w_{ij} h_j$$

2. Probability. $P(v, h) = e^{-E(v,h)} / Z$, where Z is the partition function (sum over all configurations).

3. Conditional independence. Because of the bipartite structure, all hidden units are independent given v, and all visible units are independent given h:

$$P(h_j = 1 \mid v) = \sigma(b_j + \Sigma_i v_i w_{ij}) \quad P(v_i = 1 \mid h) = \sigma(a_i + \Sigma_j w_{ij} h_j)$$

4. Learning by Contrastive Divergence (CD-1). Positive phase: clamp the data v and compute h. Negative phase: reconstruct v′ from h, then compute h′. Update:

$$\Delta w_{ij} = \eta\,(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{reconstruction}) = \eta\,(v\,h^T - v'\,h'^T)$$

Training lowers the energy of the training data and so raises its probability.

Example (step by step)

RBM with 3 visible and 2 hidden units, all biases = 0, learning rate $\eta = 0.1$, and weights

$$W = [[0.5, -0.3], [0.2, 0.4], [-0.1, 0.6]] \text{ (rows = v1,v2,v3; columns = h1,h2)}$$

Training sample **v = (1, 0, 1)**.

Step 1 (positive phase): h1: $0.5 + (-0.1) = 0.4 \rightarrow \sigma(0.4) =$ **0.5987**; h2: $-0.3 + 0.6 = 0.3 \rightarrow \sigma(0.3) =$ **0.5744**. Sample h = (1, 0).

Step 2 (reconstruction): v1: W row1 · h = 0.5 → σ = 0.6225; v2: 0.2 → 0.5498; v3: −0.1 → 0.4750. Sampled reconstruction v′ = (1, 1, 0).

Step 3 (second hidden pass): h1′: 0.5 + 0.2 = 0.7 → 0.6682; h2′: −0.3 + 0.4 = 0.1 → 0.5250.

Step 4 (weight update): Δw = 0.1 × (v h^T − v′ h'^T):

Weight	v·h	v′·h′	Δw	New w
w11	1×0.5987 = 0.5987	1×0.6682 = 0.6682	−0.0070	0.4930
w12	1×0.5744 = 0.5744	1×0.5250 = 0.5250	+0.0049	−0.2951
w21	0	1×0.6682 = 0.6682	−0.0668	0.1332
w22	0	1×0.5250 = 0.5250	−0.0525	0.3475
w31	1×0.5987 = 0.5987	0	+0.0599	−0.0401
w32	1×0.5744 = 0.5744	0	+0.0574	0.6574

Step 5: Repeat for all samples over many epochs until the reconstruction error is small.

Result and use

The RBM learns hidden features that can regenerate the data. Applications: dimensionality reduction, feature learning, collaborative filtering (recommender systems), and layer-wise pre-training of Deep Belief Networks.

2(b). Construct the architecture of a Generative Adversarial Network (GAN) and provide a real-world example. (CO2-L3) (2.5M)

Ans:

Definition

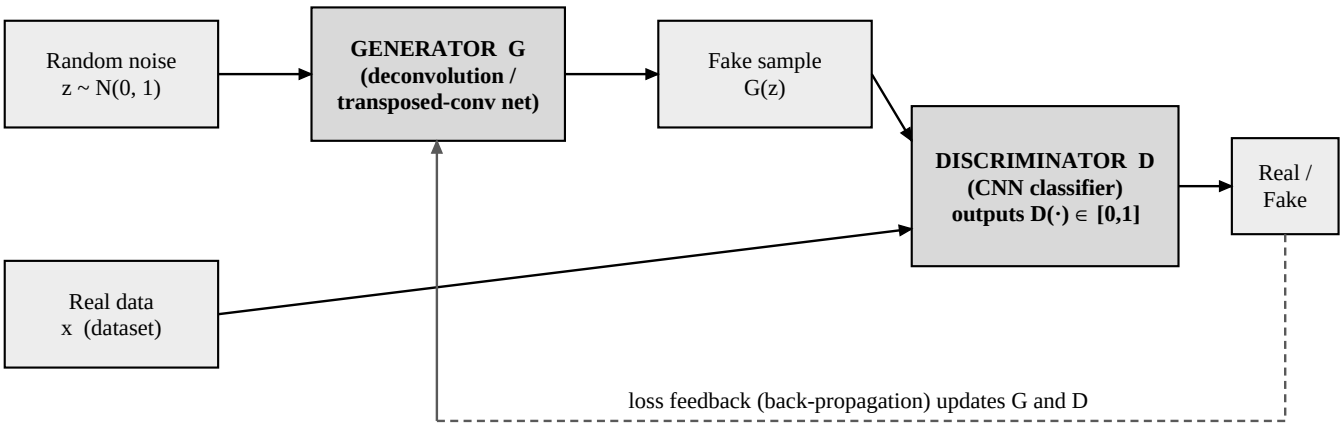
A **Generative Adversarial Network (GAN)**, introduced by Ian Goodfellow (2014), is a generative model made of two neural networks trained against each other: a **Generator (G)** that creates fake samples from random noise, and a **Discriminator (D)** that decides whether a sample is real or fake. The competition drives G to produce data that is hard to distinguish from real data.

Architecture

Fig. 4: Architecture of a GAN

min over G max over D V(D, G)

G tries to fool D; D tries to catch G



Component	Input	Output	Goal
Generator G	Noise vector z	Fake sample G(z)	Make D label fake samples as real
Discriminator D	Real x or fake G(z)	Probability D(·) that input is real	Label real as 1, fake as 0

Objective (minimax game):

$$\min_G \max_D V(D,G) = \mathbb{E}_x[\log D(x)] + \mathbb{E}_z[\log(1 - D(G(z)))]$$

Here $\mathbb{E}_x$ is the average over real samples x and $\mathbb{E}_z$ the average over noise vectors z.

Example (step by step) — training procedure

Step 1: Sample a mini-batch of noise z and pass it through G to obtain fake data G(z).

Step 2: Take a mini-batch of real data x. Feed both real and fake samples to D.

Step 3 (train D, G fixed): maximize $\log D(x) + \log(1 - D(G(z)))$. Suppose $D(x) = 0.9$ and $D(G(z)) = 0.2$: D-loss = $-[\ln 0.9 + \ln 0.8] = 0.1054 + 0.2231 =$ **0.3285**.

Step 4 (train G, D fixed): G minimizes $-\log D(G(z))$ (non-saturating form). With $D(G(z)) = 0.2$: G-loss = $-\ln 0.2 =$ **1.609** (high, so G gets a strong gradient to improve).

Step 5: Back-propagate and update G and D alternately. After many iterations the fakes improve: D(G(z)) rises toward 0.5, giving G-loss = $-\ln 0.5 =$ **0.693**.

Step 6 (equilibrium): G's distribution matches the real distribution and D can only guess, D = 0.5 everywhere.

Real-world example

Realistic face generation (NVIDIA StyleGAN, "This Person Does Not Exist"): the generator takes a random vector and produces a photo-realistic human face that does not belong to any real person, while the discriminator is trained on thousands of real face photos. Other uses: image super-resolution, text-to-image translation, medical-image augmentation (synthetic MRI/X-ray for scarce data), style transfer, and generating training data.

Result and use

GANs learn to generate new data resembling the training data without explicit probability modelling. Known challenges: training instability, mode collapse and difficult evaluation.

3(a). Apply the pooling in CNNs, and how does it contribute to feature extraction and dimensionality reduction? (CO3-L3) (2.5M)

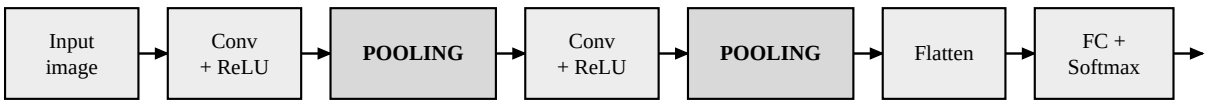
Ans:

Definition

Pooling (sub-sampling) is a CNN layer that slides a small window (e.g. 2×2) over each feature map and replaces every window with **one summary value** (maximum or average). It has no learnable weights and works on each channel independently.

Architecture

Fig. 5: Position of pooling layers in a CNN



Pooling follows convolution: it shrinks each feature map (H × W) while keeping the depth (channels) unchanged.
Feature maps shrink: 32×32 → 16×16 → 8×8 (example with 2×2 stride-2 pooling)

Output-size formula: $O = \lfloor (n - f) / s \rfloor + 1$, where n = input size, f = pool window, s = stride. Types: **max pooling** (largest value), **average pooling** (mean), **global pooling** (one value per whole map).

Example (step by step) — 2×2 pooling with stride 2 on a 4×4 feature map

Step 1: Input feature map (shading shows the four 2×2 windows):

Input 4×4

1	3	2	1
4	6	5	2
7	2	1	0
3	8	4	9

Max pooling →

6	5
8	9

Average pooling →

3.5	2.5
5	3.5

Step 2 (output size): O = (4 − 2)/2 + 1 = 2, so the output is 2×2.

Step 3 (max pooling): top-left window {1,3,4,6} → 6; top-right {2,1,5,2} → 5; bottom-left {7,2,3,8} → 8; bottom-right {1,0,4,9} → 9.

Step 4 (average pooling): (1+3+4+6)/4 = 3.5; (2+1+5+2)/4 = 2.5; (7+2+3+8)/4 = 5; (1+0+4+9)/4 = 3.5.

Step 5 (reduction): 16 values → 4 values, a **75% reduction**. A 32×32×16 feature map pooled with 2×2/stride 2 becomes 16×16×16 (1024 → 256 values per channel group, i.e. 4× fewer).

Contribution to feature extraction and dimensionality reduction

Contribution	Explanation
Dimensionality reduction	Reduces height and width, so fewer activations, fewer parameters in later layers, less memory and faster computation.
Feature extraction	Max pooling keeps the strongest response (edge, texture) in each region and discards weak, irrelevant activations; average pooling keeps the overall presence of a feature.
Translation invariance	Small shifts of the object leave the pooled output nearly unchanged (e.g. 6 remains 6 wherever it sits in its window).
Larger receptive field	Each later neuron “sees” a bigger part of the image, helping detect larger patterns.
Reduces overfitting	Fewer parameters and a summarized representation improve generalization.

Result and use

Pooling progressively compresses the spatial size while retaining the most important features, which is why CNNs such as LeNet, AlexNet and VGG alternate convolution and pooling layers. Modern networks also use global average pooling before the classifier. Limitation: pooling discards exact position information.

3(b). Construct a convolutional network to demonstrate the effect of striding and padding styles with an example. (CO3-L3) (2.5M)

Ans:

Definition

Stride (s) is the number of pixels by which the filter moves at each step. **Padding (p)** is the addition of border pixels (usually zeros) around the input. Together with the filter size f they control the output size:

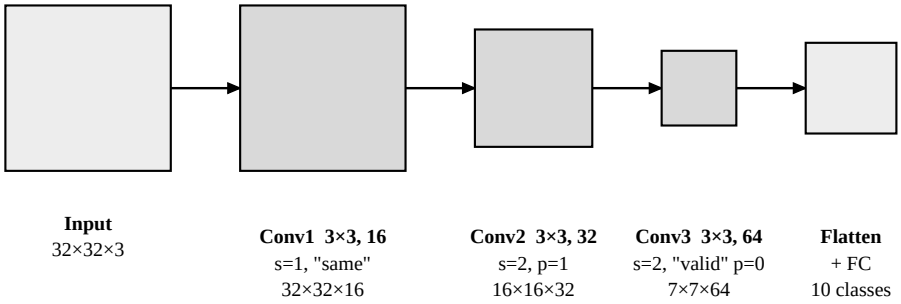
$$O = \lfloor (n + 2p - f) / s \rfloor + 1$$

Padding style	Meaning	Effect on size (s = 1)
Valid	No padding (p = 0)	Output shrinks to n − f + 1
Same	Zero padding p = (f − 1)/2	Output size = input size
Full	Padding p = f − 1	Output grows to n + f − 1

Architecture (convolutional network with different strides and paddings)

Fig. 6: CNN demonstrating stride and padding

Padding keeps the size ("same"); stride 2 halves it; "valid" (no padding) shrinks it



Layer	Filter / Stride / Padding	Calculation	Output
Input	—	—	32×32×3
Conv1	3×3, 16 filters, s = 1, p = 1 (same)	$(32 + 2 - 3)/1 + 1 = 32$	32×32×16
Conv2	3×3, 32 filters, s = 2, p = 1	$\lfloor(32 + 2 - 3)/2\rfloor + 1 = 15 + 1 = 16$	16×16×32
Conv3	3×3, 64 filters, s = 2, p = 0 (valid)	$\lfloor(16 - 3)/2\rfloor + 1 = 6 + 1 = 7$	7×7×64
Flatten + FC	7×7×64 = 3136 values → 10 classes	—	10

Example (step by step) — 5×5 input, 3×3 vertical-edge filter

Step 1: Input image and filter:

Input 5×5

1	2	0	1	3
0	1	3	1	2
2	1	0	2	1
1	0	2	3	0
3	1	1	0	2

Filter 3×3

1	0	-1
1	0	-1
1	0	-1

Step 2 (valid, s = 1): $O = (5 - 3)/1 + 1 = 3$. First value = $(1\cdot1 + 2\cdot0 + 0\cdot(-1)) + (0\cdot1 + 1\cdot0 + 3\cdot(-1)) + (2\cdot1 + 1\cdot0 + 0\cdot(-1)) = 1 - 3 + 2 =$ **0**. Output 3×3:

Valid, s = 1 (3×3)

0	0	-3
-2	-4	2
3	-3	0

Step 3 (same, s = 1, p = 1): the 5×5 input becomes 7×7 with a zero border; $O = (5 + 2 - 3)/1 + 1 = 5$. The size is preserved and edge pixels are used as often as centre pixels:

Same, s = 1, p = 1 (5×5)

-3	-2	1	-2	2
-4	0	0	-3	4
-2	-2	-4	2	6
-2	3	-3	0	5
-1	1	-2	1	3

Step 4 (valid, s = 2): the filter jumps 2 pixels; $O = \lfloor (5 - 3)/2 \rfloor + 1 = 2$:

Valid, s = 2 (2×2)

0	-3
3	0

Step 5 (same padding, s = 2, p = 1): $O = \lfloor (5 + 2 - 3)/2 \rfloor + 1 = 3$:

p = 1, s = 2 (3×3)

-3	1	2
-2	-4	6
-1	-2	3

Step 6 (summary of effect):

Setting (n = 5, f = 3)	Stride	Padding	Output size	Effect
Valid	1	0	3×3	Shrinks; border information partly lost
Same	1	1	5×5	Size preserved; border pixels retained
Valid	2	0	2×2	Strong down-sampling, fewer computations
Padded	2	1	3×3	Moderate down-sampling with border kept

Result and use

Stride controls down-sampling: larger stride gives a smaller output, less computation and less overlap between filter positions (can replace pooling). **Padding** controls border handling: “same” padding preserves spatial size and lets deep networks stack many layers, while “valid” padding shrinks the maps at each layer. Choosing them correctly is essential when designing CNN architectures.