UCS071Q3 - Deep Learning

Unit 2

Convolutional Neural Networks

Convolution Operation - Sparse Interactions - Parameter Sharing - Equivariance - Pooling - Convolution Variants: Strided - Tiled - Transposed and Dilated Convolutions; CNN learning: Non-linearity Functions - Loss functions - Regularization - Optimizers - Gradient Computation

* Convolution Operation

→ an operation of 2 functions on a real-valued input arguments.

→ eg In order to track a spaceship with a noisy laser

$x(t)$

Objective: Obtain a less noisy estimate of the spaceship's position using a weighted average

Let $x(t)$ = position of spaceship at time t

$w(a)$ = weighting function.

Convolution of these 2 can be represented as

$$s(t) = \int x(a)\, w(t-a)\, da \quad \rightarrow \text{continuous}$$

$$s(t) = (x * w)(t)$$

$$= \sum_{a=-\infty}^{\infty} x(a)\, w(t-a) \quad \rightarrow \text{discrete}$$

→ 2D convolution is defined as

$$S(i,j) = (I * K)(i,j) \sum_m \sum_n I(m,n)\, K(i-m, j-n)$$

$I(m,n)$ = 2D input (image)

$K(i-m, j-n)$ = 2D Kernel applied to the input

* Matrix Representation of Convolution

→ Discrete convolution can be viewed as multiplication by a Toeplitz matrix.

→ In 2D, convolution corresponds to a doubly block circulant matrix

Input

a	b	c	d
e	f	g	h
i	j	k	l

Kernel

w	x
y	z

Output

$aw+bx+ey+fz$	$bw+cx+fy+gz$	$cw+dx+gy+hz$
$ew+fx+iy+jz$	$fw+gx+jy+kz$	$gw+hx+ky+lz$

→ Every time we slide the kernel, we get one value in the output.

→ The resulting output is called a feature map.

* Sparse Interactions, Parameter Sharing & Equivariant Representations

→ Convolution leverages 3 key ideas

(i) sparse interactions

(ii) parameter sharing

(iii) equivariant representations

A. Sparse Interactions

→ Traditional neural networks use dense matrix multiplication where every output interacts with every point

→ Convolutional networks use sparse interactions by making the kernel smaller than the input

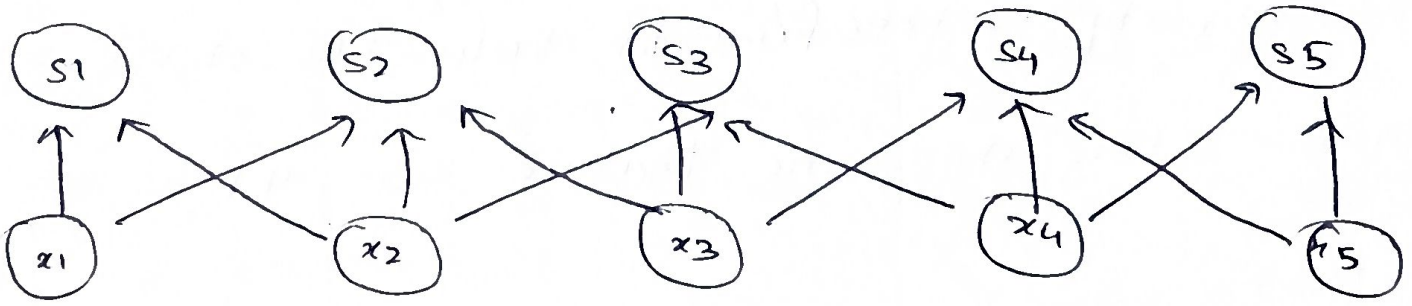
Benefits

→ Fewer parameters to store

→ reduced memory & computational requirements

→ improved statistical efficiency

Example



(rather than a fully connected layer)

B. Parameter Sharing

→ Convolutional networks use the same parameters for multiple functions in the model - called parameter sharing / tied weights

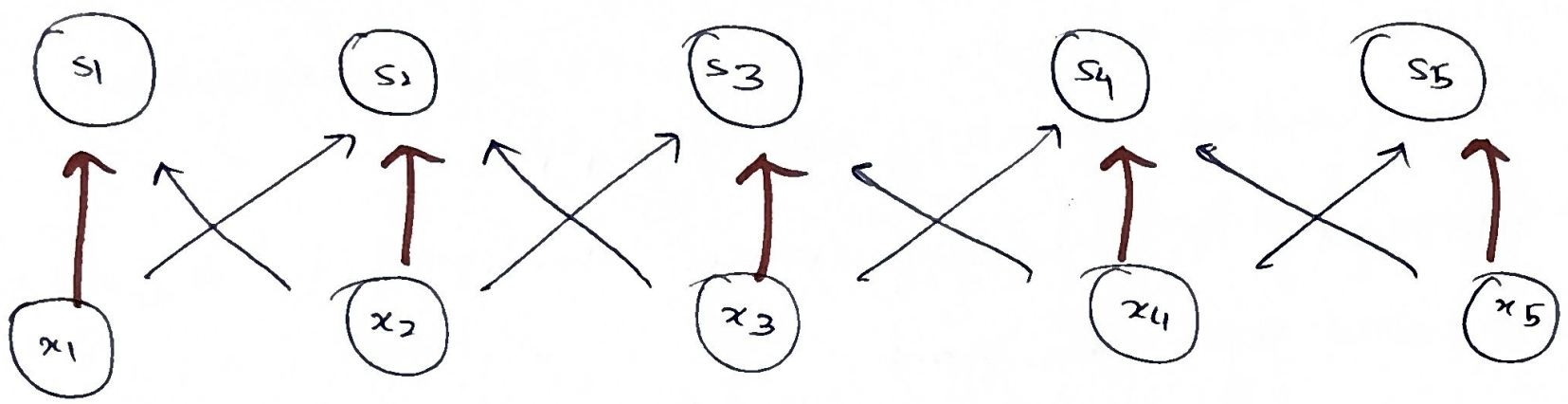
→ In traditional networks, each element of the weight matrix is used once.

→ In convolutional networks, each ~~fe~~ input member is applied at every position of the input

Benefits

(i) reduces model parameters

(ii) maintains computational efficiency ??



c.) Equivariant Representations

→ Convolution provides equivariance to translation.

→ If the input shifts, the output shifts in the same way.

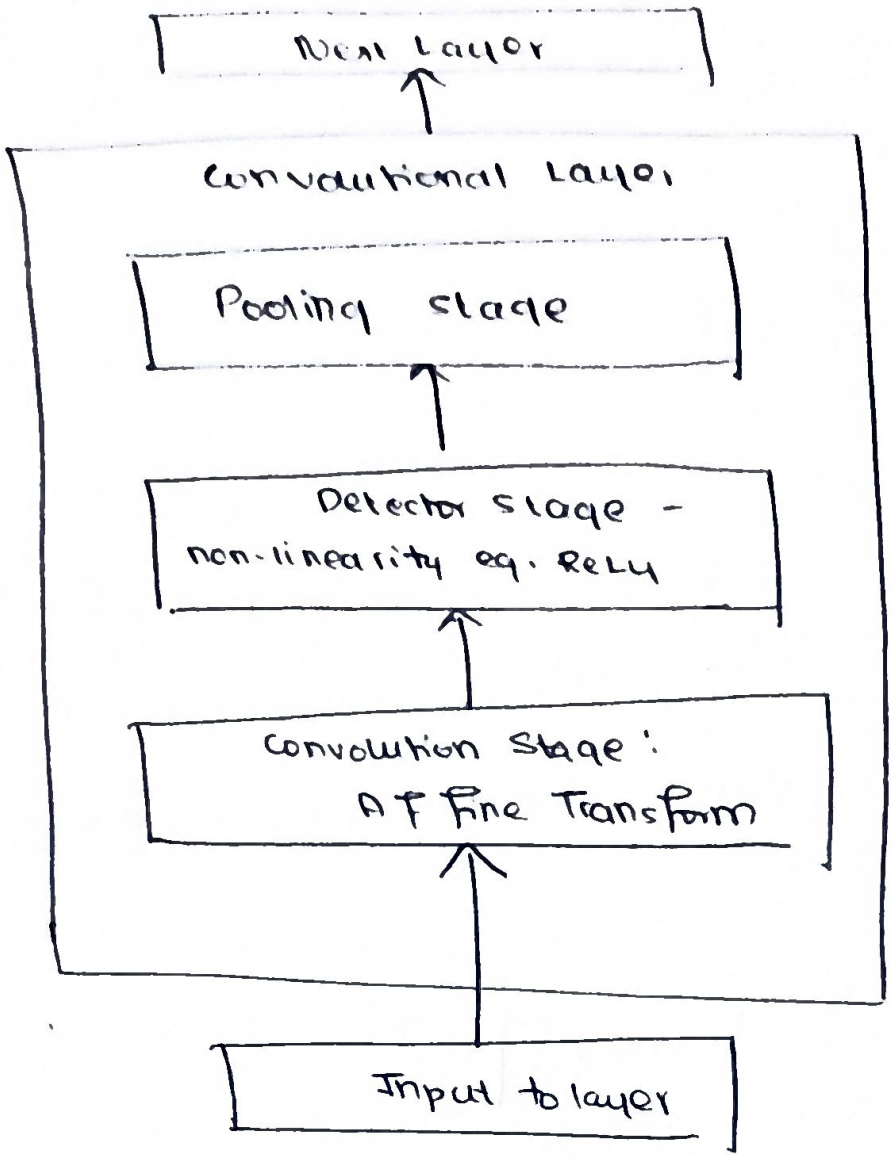
→ This property is beneficial for tasks like object detection in images, where the same feature can appear in different locations

* Overview of a Convolutional Layer

→ A typical layer consists of 3 stages:

(i) convolution
(ii) non-linear activation (detector stage)
(iii) pooling

→ Pooling replaces the output with a summary statistic of nearby outputs



* Pooling? → replace output with a summary statistic
  - → max pool
  - → L2 normal
  - → weighted avg

(i) **Max Pooling**

→ reports the maximum output within

(a) 224 × 224 × 64 —pool→ 112 × 112 × 64

(stride = 2, size = 2) → 2 × 2 ⇓ 1/2 size

downsampling

(b)

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

x ↑, → y

max pool with 2 × 2 filters 2 stride 2

6	8
3	4

⇓ Pooling with Downsampling

(ii) **L2 Norm Pooling**

→ calculate the square root of the sum of the squares of the values in the pooling region

→ It is expressed as:

$$L2 = \sqrt{\sum_{i=1}^{N} xi^2}$$

Benefits - captures overall energy in a pooling region, not just max. value

- can be more stable than max pooling, especially when values are close

eg. pooling region = [1, 2, 3]

$$L2 = \sqrt{1^2+2^2+3^2} = \sqrt{14} = 3.74$$

(iii) **Weighted Average Pooling**

→ calculates the average of values in a pooling region, with each value weighted by a specific function

→ A common wt. function is based on distance from the center of the pooling region

Benefits - allows more flexibility in how features are pooled, giving more importance to central pixels

- can be tailored to specific tasks where certain regions are more imp.

Example - pooling region [1, 2, 3]

wts = [0.1, 0.3, 0.6]

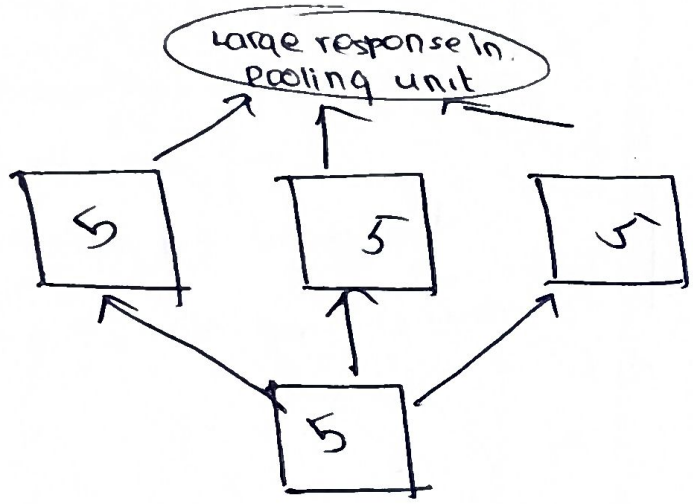
$$\text{wt. avg} = \frac{(1)(0.1) + (2)(0.3) + (3)(0.6)}{0.1 + 0.3 + 0.6} = 2.9$$

* Pooling for Learned Invariances

→ pooling over outputs from separately parametrized convolutions

→ features learn which transformations to become invariant to

eg. invariance to rotation



Large response in detector unit 1

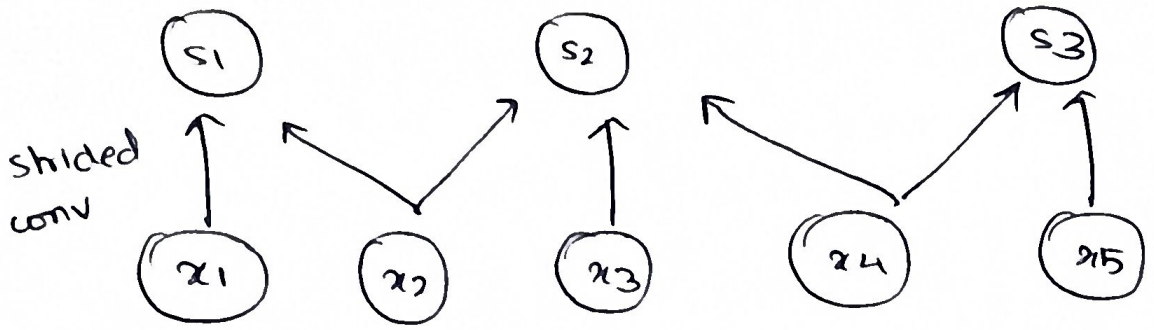
* Convolution Variants - Strided and Tiled

A. Stride

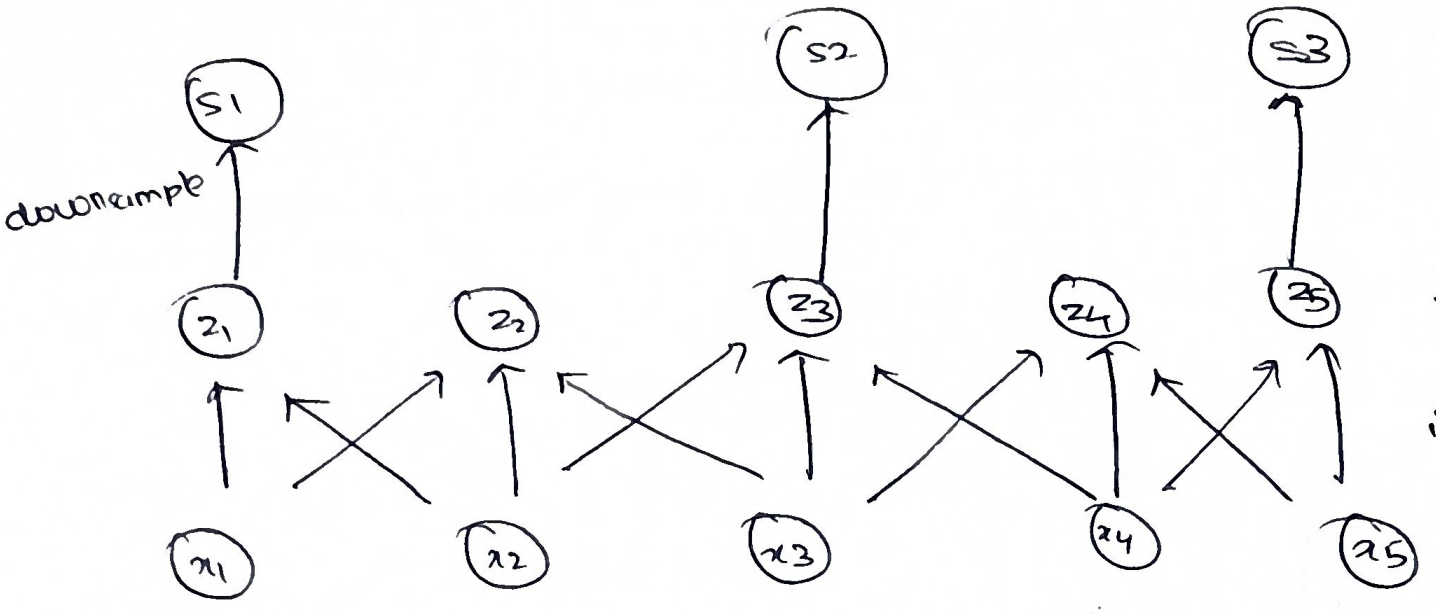
→ Stride s controls the step size for moving the convolutional kernel

→ Larger strides reduce computational cost by downsampling the output

→ Strides can be defined separately for each spatial direction



⟹ stride of 2 (single operation)



⇒ convolution with a stride > 1 is equivalent to convolution w/ unit stride followed by downsampling

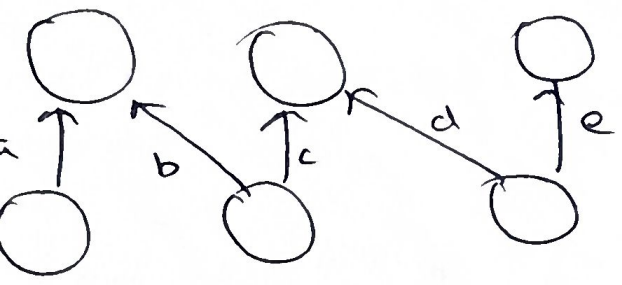
→ computationaly wasteful

B. Tiled Convolution

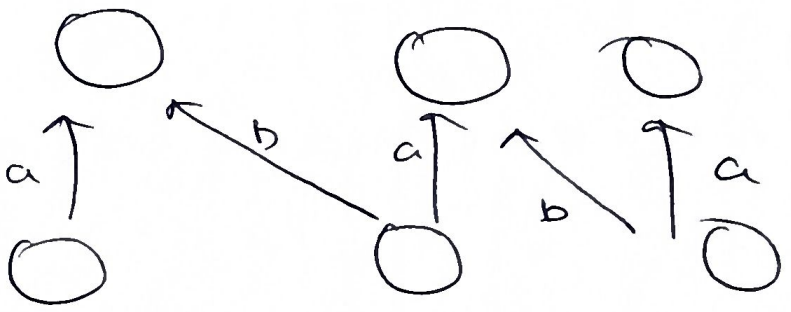
Some preface

* Locally connected Layers vs. Convolutional Layers

↑ no parameter sharing, each connection has its own weight



↑ parameters are shared across spatial locations (worst case = fully connected layers)



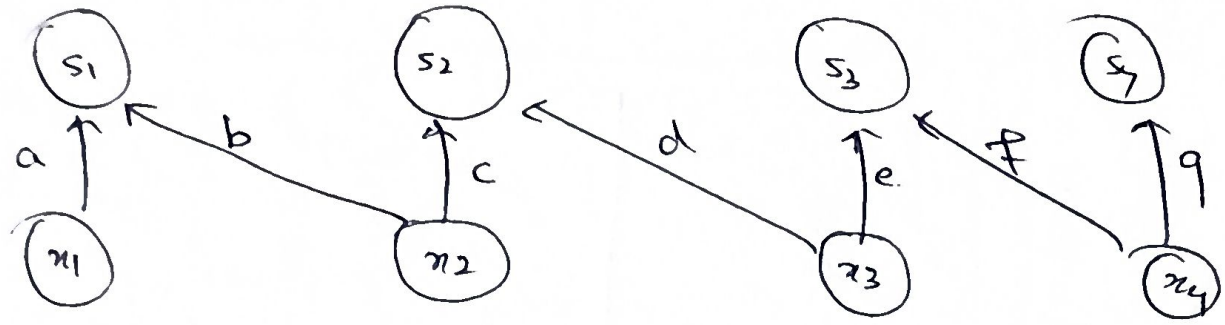
→ Tiled convolution rotates through a set of kernels rather than applying a single kernel everywhere

→ provides a balance between local and convolutional layers

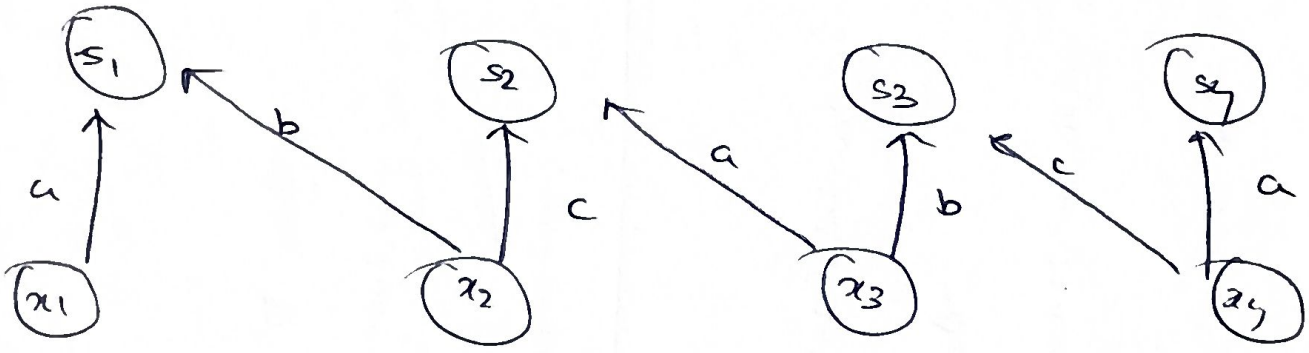
→ memory requirements grow by a factor of the kernel size rather than the full output feature map

Comparison with Local & Convolutional layers

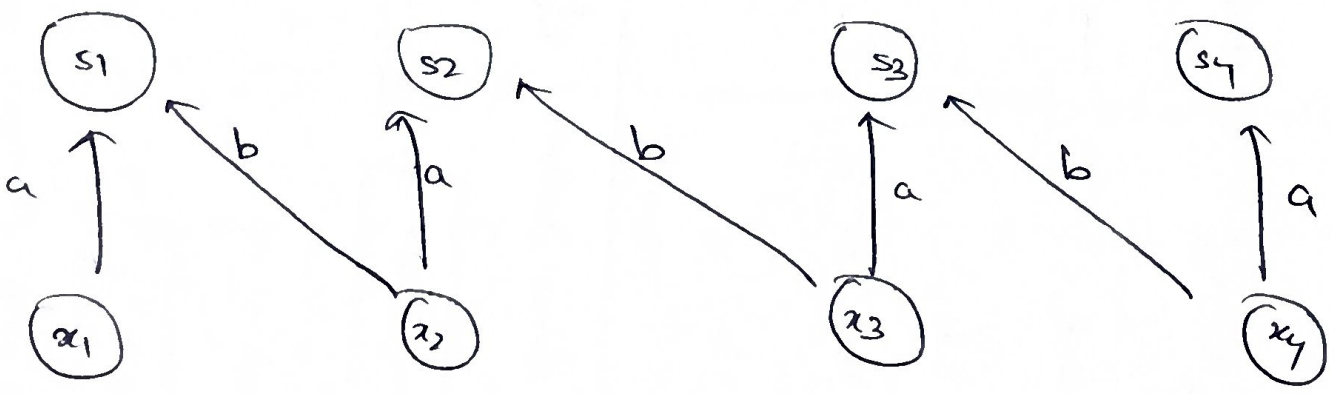
Local



Tiled



Convolution



Locally Connected Layers	Tiled Convolution	Standard Convolution
→ no parameter sharing, no weight reuse ⇒ larger no. of params → spatially distinct weights, offer high flexibility, but at a computational cost	→ partial parameter sharing → use t different kernels that are applied in a tiled pattern → Kernel cycles – cycles through t steps → parameter sharing across tiles	→ full parameter sharing → equivalent to tiled convolution w/ t=1 → same kernel applied everywhere → dramatically reduces the no. of parameters & computational cost → Homogenous behaviour - all spatial locations share the same set of parameters → less flexibility, more consistency

Two more kinds of convolution are

3. **Transpose convolution** → used to upsample, increase spatial dimensions of input

4. **Dilated convolution** → a convolution with gaps (dilations) 'inserted' between filter elements to allow it to capture a larger receptive field.

* Zero Padding in Convolutional Networks

→ used to control output size & prevent shrinking

Valid Convolution - no padding

- output size decreases by k-1

Same convolution - padding to keep input & output sizes the same

Full convolution | Custom Padding - padding so every pixel is visited by the kernel

* Training Convolutional Networks

→ need to minimize the loss - done by computing the gradient

→ loss function $J(V, K)$

↑ input ↑ kernel

→ Gradients are needed for forward propagation, backpropagation to weights, and backpropagation to inputs.

→ During backpropagation, transpose convolution is done.

→ This operation is necessary to backpropagate error derivatives through a convolutional layer.

→ The transpose convolution must be coordinated w/ the forward propagation in terms of padding & stride (2 eqns from slide)

* Non-Linearity Functions and Loss Functions

* Non-Linearity in CNNs

→ The weight layers in a CNN are often followed by a non-linear activation function

→ The activation function takes a real-valued input & squashes it within a small range such as $[0,1]$ or $[-1,1]$

→ Non-linearities allow a neural network to learn non-linear mappings

→ In the absence of non-linearities, a stacked network of weight layers is equivalen to a linear mapping from the input domain to the output domain

* Activation Functions

→ A non-linear function can be understood as a switching or a selection mechanism

→ decides whether a neuron will fire or not given the inputs

→ Activation functions in deep networks are differentiable to enable error back propagation

Common Activation Functions

(i) Sigmoid $[0,1]$

$$f(x) = \frac{1}{1+e^{-x}}$$

(ii) Tanh : $[-1,1]$ $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

(iii) Algebraic Sigmoid : $[-1,1]$ $f(x) = \frac{x}{\sqrt{1+x^2}}$

(iv) ReLu → map to zero if -ve

keeps value unchanged if positive

$f(x) = \max(0, x)$

- Noisy Relu - adds noise - $f(x) = \max(0, x + \varepsilon_p)$

$\varepsilon_p \sim N(0, \sigma(x))$

- Leaky ReLU $f(x) = \begin{cases} x, & x > 0 \\ cx, & x \leq 0 \end{cases}$

- Parametric ReLU : $f(x) = \left.\begin{cases} x, & x > 0 \\ ax, & x \leq 0 \end{cases}\right\}$ leak parameter 'a' is trained

- Randomized Leaky ReLU :

(RReLU) $f(x) = \begin{cases} x & , x > 0 \\ ax & , x \leq 0 \end{cases}$

$a \sim U(l, u)$

randomly select leak factor from a uniform distribution

* CNN Loss Functions

→ estimate the quality of predictions

→ used during training to minimize the error between the predicted & true labels

① Cross Entropy Loss

$$L(p,y) = -\sum_{n} y_n \log(p_n) \quad n \in \{1, 2, \ldots N\}$$

p_n = softmax function

$$p_n = \frac{\exp(p_{On})}{\sum \exp(p_{Ok})}$$

→ cross entropy loss measures the performance of a classification model whose output is a probability value is between 0 & 1.

→ used in multi-class classification problems

KL Divergence Relation

$$KL(p||y) = L(p,y) - H(p)$$

② SVM Hinge Loss

$$L(p,y) = \sum_{n} \max(0, m - (2y_n - 1) p_n)$$

→ used in SVMs & focuses on maximizing the margin between different classes

→ penalizes misclassified points that fall within the margin

③ Squared Hinge Loss

$$L(p,y) = \sum_n \max(0, m - (2y_n - 1)p_n))^2$$

→ modifies the standard hinge loss by squaring the margin error

→ makes it more sensitive to margin violations, penalizing errors more severely

→ useful in scenarios where emphasis is placed on correcting large margin violations

→ applied in tasks requiring robust margins, such as image classification with noisy data.

④ Euclidean loss

$$L(p,y) = \frac{1}{2N} \sum_n (p_n - y_n)^2$$

→ calculates the average of the squared differences between predicted & actual values

→ used in regression tasks where the goal is to predict continuous values

(5) L1 Error

$$L(p,y) = \frac{1}{N}\sum_n |p_n - y_n|$$

→ called MAE, measures the average magnitude of the errors without considering their direction

→ less sensitive to outliers compared to Euclidean loss

→ preferred in regression tasks where robustness to outliers is desired

(6) Contrastive loss

$$d = \|f_a - f_b\|_2$$

$$L(p,y) = \frac{1}{2N}\sum_n \left(y d^2 + (1-y)\max(0, m-d)^2\right)$$

→ used to train models that learn representations by ensuring that similar inputs are close together in the feature space & dissimilar ones are far apart.

→ common in tasks involving similarity learning

→ applied in Siamese networks for matching / comparing pairs of items

(7) Expectation Loss

$$L(p,y) = \sum_n \left| y_n - \frac{\exp(p_n)}{\sum_k \exp(p_k)} \right|$$

→ minimizes the expected misclassification probability by adjusting predicts

→ aims to maximize the probability of the true label

→ less commonly used in deep neural networks due to optimization challenges

→ could be useful in cases where robustness to outliers is a priority

⑧ Structural Similarity Measure (SSIM)

$$L(p,y) = 1 - SSIM(n)$$

$$SSIM(n) = \frac{(2\mu_{p_n}\mu_{y_n} + c_1)(2\sigma_{p_n y_n} + c_2)}{(\mu^2_{p_n} + \mu^2_{y_n} + c_1)(\sigma^2_{p_n} + \sigma^2_{y_n} + c_2)}$$

→ measures the perceptual similarity between images by comparing luminance, contrast & structure

→ used in image quality assessment & enhancement tasks, image compression, denoising & super-resolution tasks

Example on SVM hinge loss

$p = [1.2, 2.6, 2.0]$

correct class = 2 $m = 1$

The hinge loss is calculated as

$$\max(0, m + p_i - p_c)$$

$L_1 = \max(0, 1 + 1.2 - 2.6) = \max(0, -0.6) = 0$

$L_3 = \max(0, 1 + 2.0 - 2.6) = \max(0, 0.2) = 0.2$

Total loss = $L_1 + L_3 = 0 + 0.2 = \underline{0.2}$

* Regularization

→ Regularization is a technique used to improve model generalization by preventing overfitting. Leads to: model simplification, feature selection, complexity control

→ It involves adding a penalty term, $\Omega(\theta)$ to the objective function $J(\theta; X, y)$

→ The regularized objective function is given by:

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha \Omega(\theta)$$

α is a hyperparameter that controls the strength of the regularization.

A. L2 Regularization

→ also known as weight decay, adds a penalty proportional to the square of the weights.

→ The L2 penalty is given by

$$\Omega(\theta) = \frac{1}{2} \|w\|^2 = \frac{1}{2} w^T w$$

→ The regularized objective function becomes:

$$\tilde{J}(w, X, y) = \frac{\alpha}{2} w^T w + J(w; X, y)$$

$J(w; X, y)$ = original cost fn.
α = regularization param
$w^T w = \|w\|_2^2$ = squared Euclidean norm of the weight vector

Gradient Descent with L2 Regularization

The objective function can be written as:

$$\tilde{J}(w, X, y) = \frac{\alpha}{2} \|w\|_2^2 + J(w, X, y)$$

Differentiate:

$$\nabla_w \tilde{J}(w; x, y) = \alpha w + \nabla_w J(w; x, y)$$

→ The weight update rule in gradient descent is modified to:

$$w \leftarrow (1 - \epsilon_p \alpha) w - \epsilon_p \nabla_w J(w; x, y) \rightarrow (A)$$

→ L2 regularization shrinks the weights towards the origin, reducing model complexity.

Derivation for Weight Update Rule

$$\nabla_w \tilde{J}(w; x, y) = \nabla_w J(w; x, y) + \alpha w \quad \rightarrow (1)$$

The weight update rule is:

$$w \leftarrow w - \epsilon_p \nabla_w \tilde{J}(w; x, y)$$

$$w \leftarrow w - \epsilon_p \left(\nabla_w J(w; x, y) + \alpha w \right) \rightarrow \text{Sub } (1)$$

$$w \leftarrow w - \epsilon_p \nabla_w J(w; x, y) - \epsilon_p \alpha w$$

Factor out w

$$w \leftarrow \underbrace{(1 - \epsilon_p \alpha) w}_{\text{Decay Term}} - \underbrace{\epsilon_p \nabla_w J(w; x, y)}_{\text{Gradient Descent Term}}$$

↳ This is Equation (A)

Decay Term → acts as the shrinkage factor, happens when α is +ve (the whole term becomes -ve)

Gradient Descent Term → moves the weights in the direction that reduces the cost function

Effects of L2 Regularization

(1) Regularization Effect - The shrinkage term reduces the magnitude of the weights, helping to prevent overfitting by discouraging large weights.

(2) Balance Between Fit and Complexity - By controlling the regularization parameter α, we can balance the model's fit to the training data against the complexity of the model.

B. **L1 Regularization**

→ L1 regularization adds a penalty proportional to the absolute value of the weights:

$$\Omega(\theta) = \|w\|_1 = \sum_i |w_i|$$

→ The regularized objective function becomes:

$$\tilde{J}(w; X, y) = \alpha \|w\|_1 + J(w; X, y)$$

Gradient Descent with L1 Regularization

→ The gradient of the regularized objective function is:

$$\nabla_w \tilde{J}(w; X, y) = \alpha \, \text{sign}(w) + \nabla_w J(w; X, y)$$

→ L1 regularization introduces sparsity by shrinking some weights to exactly zero.

Effect of L1 Regularization

→ Consider a diagonal Hessian $H = \text{diag}([H_{1,1}, \ldots, H_{n,n}])$

→ The solution for each weight is given by:

$$w_i = \text{sign}(w_i^*) \max\left(|w_i^*| - \frac{\alpha}{H_{i,i}}, 0\right)$$

→ L1 regularization can shrink weights to 0, leading to a sparse solution.

* Comparison of L1 vs. L2 regularization

→ L2 regularization results in a solution with non-zero weights.

$$\tilde{w}_i = \frac{H_{i,i}}{H_{i,i} + \alpha} \cdot w_i^*$$

→ L1 regularization can produce sparse solutions where some $w_i = 0$

→ L1 is used for feature selection (eg LASSO)

* Dataset Augmentation

Purpose: Improve model generalization by training on more data

Method: →
- → Create fake data via transformations of existing data
- → Make sure the created data is invariant to transformations, and that class labels are not altered.
- → Perform translation, scaling, rotation of images to create more samples.

* Noise Robustness

Different Kinds include:

(i) Input Noise: Adds random noise to inputs to improve robustness

(ii) Hidden Unit Noise: Augments dataset at multiple abstraction levels

(iii) Weight Noise: Stochastic implementation reflecting weight uncertainty

(iv) Output Target Noise: Handles label errors using noise injection.

* Early Stopping

→ a widely used regularization process - involves stopping the training process when the validation error starts increasing, even if the training error is still decreasing.

→ This method helps prevent overfitting, and can be seen as a hyperparameter optimization technique.

Procedure

① Monitor Validation Error - Regularly evaluate the model's performance on the validation set during training.

② Save the best model - Store the model parameters, whenever the validation error improves (decreases)

③ Stop Training - Stop the training process when the validation error begins to increase, indicating overfitting.

④ Return the best parameters - Use the model parameters corresponding to the lowest validation error, ensuring the best generalization.

Algorithm

n → no. of steps between evaluations

p → patience - no. of times the validation error is allowed to worsen before stopping training.

θ → model parameters being trained, initialized to θ_0

θ^* → best parameters - corresponding to the lowest validation error

i → current no. of training steps

j → counter for worsening validation error

v → current best validation error

i^* → best no. of training steps

Algo

Initialize : $\theta \leftarrow \theta_0$ $\quad v \leftarrow \infty$

$i \leftarrow 0$ $\quad \theta^* \leftarrow \theta$

$j \leftarrow 0$ $\quad i^* \leftarrow i$

While $j < p$ do

① Update θ by running the training algorithm for n steps.

② $i \leftarrow i+n$

③ $v' \leftarrow$ validationSetError(θ)

④ If $v' < v$ then

$j \leftarrow 0$

$\theta^* \leftarrow \theta$

$i^* \leftarrow i$

$v \leftarrow v'$

⑤ Else

① $j \leftarrow j+1$

⑥ End while

⑥ Best parameters are θ^*, best number of training steps is i^*.

* Reusing Data after Early Stopping

→ Early stopping requires a validation set, which means some training data is not used for training

→ Two strategies for utilizing this data:

A. Retrain the model from scratch on the entire dataset for the optimal number of steps determined by early stopping.

B. Continue training with the obtained parameters using all data, monitoring until the validation loss reaches the early stopping point.

A. **Retraining after early stopping**

1. Split the dataset into subtrain and validation sets
2. Run early stopping on the subtrain set to determine the

Optimal number of steps i*

3 Reinitialize the model parameters

4. Train the model on the entire training set for i* steps.

B. Continuing Training After Early Stopping?

1. Split the training set into subtrain and validation sets

2. Run early stopping on the subtrain set to update parameters θ and determine the overfitting objective value ϵ_f.

3. Continue training on the entire training set until the validation loss reaches ϵ_f.

* Early Stopping as a Regularizer

→ Early stopping can be seen as equivalent to L2 regularization

→ By stopping training early, the model's capacity is effectively reduced, which prevents overfitting.

→ This model restricts the parameter space, the model can explore, similarly to how L2 regularization penalizes large weights

* Dropout

→ A regularization technique that is computationally feasible for large datasets and neural networks.

→ It has 2 phases:

(i) Training phase – during training, neurons are randomly dropped out or ignored with probability p

(ii) Inference phase – All neurons are used, but their outputs are scaled to account for the dropout applied during training.

→ During training, the output of a neuron is either:

$$z = \begin{cases} 0 & , \text{ with probability } p \\ \frac{a}{1-p} & , \text{ with probability } 1-p \end{cases}$$

where a is the activation before dropout.

→ The expected output is

$$E[z] = (1-p) \cdot \frac{a}{1-p} + p \cdot 0 = a$$

→ During inference, to maintain this expectation, scale the activations by 1-p

Scaled Output = $(1-p) \cdot a$

eg. if $p = 0.5$

During training $\frac{1}{1-p} = 2$, during inference, scale by $1-p = 1-0.5 = 0.5$

Deep Learning Unit 2 Questions

1. Consider the following class labels and their corresponding predicted probabilities from a classification model:

Label	Predicted Probability
Cat	[0.7, 0.2, 0.1]
Dog	[0.1, 0.6, 0.3]
Rabbit	[0.2, 0.3, 0.5]

Each label corresponds to one-hot encoded ground truth labels:

- Cat → [1, 0, 0]
- Dog → [0, 1, 0]
- Rabbit → [0, 0, 1]

Task:

1. Apply the cross-entropy loss for each instance using the predicted probabilities and the ground truth labels.
2. Compute the total cross-entropy error by averaging the individual losses.

The formula for **cross-entropy loss** is:

$$\text{Loss} = -\sum_{i=1}^{C} y_i \log(p_i)$$

where:

- y_i: ground truth (one-hot encoded)
- p_i: predicted probability for class i
- C: number of classes

Step-by-Step Calculation:

1. **For Cat ([1, 0, 0]):**

$$\text{Loss}_{\text{Cat}} = -[1 \cdot \log(0.7) + 0 \cdot \log(0.2) + 0 \cdot \log(0.1)] = -\log(0.7)$$

$$\text{Loss}_{\text{Cat}} = -\log(0.7) \approx 0.357$$

2. **For Dog ([0, 1, 0]):**

$$\text{Loss}_{\text{Dog}} = -[0 \cdot \log(0.1) + 1 \cdot \log(0.6) + 0 \cdot \log(0.3)] = -\log(0.6)$$

$$\text{Loss}_{\text{Dog}} = -\log(0.6) \approx 0.511$$

3. **For Rabbit ([0, 0, 1]):**

$$\text{Loss}_{\text{Rabbit}} = -[0 \cdot \log(0.2) + 0 \cdot \log(0.3) + 1 \cdot \log(0.5)] = -\log(0.5)$$

$$\text{Loss}_{\text{Rabbit}} = -\log(0.5) \approx 0.693$$

4. **Total Loss:** Average the individual losses:

$$\text{Total Loss} = \frac{\text{Loss}_{\text{Cat}} + \text{Loss}_{\text{Dog}} + \text{Loss}_{\text{Rabbit}}}{3}$$

$$\text{Total Loss} = \frac{0.357 + 0.511 + 0.693}{3} \approx 0.520$$

Final Answer:

- Individual losses:
  $\text{Loss}_{\text{Cat}} = 0.357, \text{Loss}_{\text{Dog}} = 0.511, \text{Loss}_{\text{Rabbit}} = 0.693$
- Total cross-entropy loss: ≈ 0.520

2. CAT 1 12-mark question on Cross Entropy Loss

Consider a fully connected neural network with:

- **3 inputs**: x_1, x_2, x_3
- **2 hidden layers**, each having **4 neurons** with **Sigmoid activation functions**.
- **Output layer** with a **Softmax activation function**.

Assumptions:

1. All weights in the network are set to 1.
2. All biases are set to 0.

Tasks:

1. Apply the activation functions and calculate the output of the network for an input $x = [x_1, x_2, x_3]$.
2. Compute the **cross-entropy loss** using the network's output and the given target label.
   - Let t_i be the target label and y_i the predicted probability for the i^{th} class.

Step 1: Forward Pass

1. **Input to the Network:**

$$x = [x_1, x_2, x_3]$$

2. **Hidden Layer 1:** Each neuron in the first hidden layer receives a weighted sum of the inputs:

$$z_{h1} = x_1 + x_2 + x_3$$

Since all weights are 1 and biases are 0, this applies to all four neurons.

Apply the Sigmoid activation function:

$$\text{Sigmoid}(z) = \frac{1}{1 + e^{-z}}$$

Output of all four neurons in the first hidden layer:

$$o_{h1} = \frac{1}{1 + e^{-(x_1+x_2+x_3)}} \quad \text{(same for all 4 neurons since weights are equal).}$$

3. **Hidden Layer 2:** The input to each neuron in the second hidden layer is the sum of outputs from the first hidden layer:

$$z_{h2} = o_{h1} + o_{h1} + o_{h1} + o_{h1} = 4 \cdot o_{h1}$$

Apply the Sigmoid activation function:

$$o_{h2} = \frac{1}{1 + e^{-4 \cdot o_{h1}}}$$

Output of all four neurons in the second hidden layer is the same:

$$o_{h2} = \frac{1}{1 + e^{-4 \cdot o_{h1}}}.$$

4. **Output Layer:** Each neuron in the output layer receives the sum of the outputs from the second hidden layer:

$$z_o = o_{h2} + o_{h2} + o_{h2} + o_{h2} = 4 \cdot o_{h2}$$

Apply the Softmax activation function for 3 output classes:

$$y_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Since $z_i = z_o$ for all output neurons:

$$y_i = \frac{e^{z_o}}{3 \cdot e^{z_o}} = \frac{1}{3}, \quad \text{for all } i.$$

Step 2: Cross-Entropy Loss

The cross-entropy loss for the target label t_i is:

$$\text{Loss} = -\sum_i t_i \log(y_i)$$

For a one-hot encoded target label (e.g., $t = [1, 0, 0]$), the loss becomes:

$$\text{Loss} = -\log(y_{\text{target}}) = -\log\left(\frac{1}{3}\right)$$

Compute the loss:

$$\text{Loss} = -\log\left(\frac{1}{3}\right) = \log(3) \approx 1.099$$

Final Answer:

1. **Output of the network:**

$$y_1 = y_2 = y_3 = \frac{1}{3}.$$

2. **Cross-Entropy Loss:**

$$\text{Loss} \approx 1.099.$$