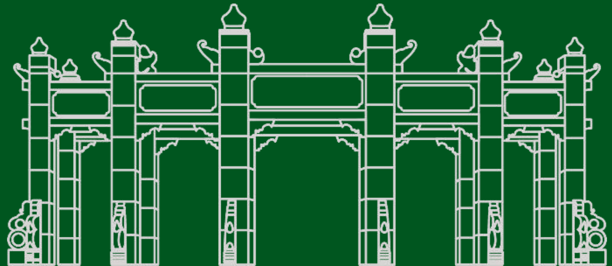
ECE371
Neural Network and Deep Learning

Lecture 2
Convolution Neural Network
Part I

Prof. Dr. Ruimao Zhang

zhangrm27@mail.sysu.edu.cn
Room N205, Engineering Building 3
School of Electronics and Communication Engineering , SYSU



Outline

Outline

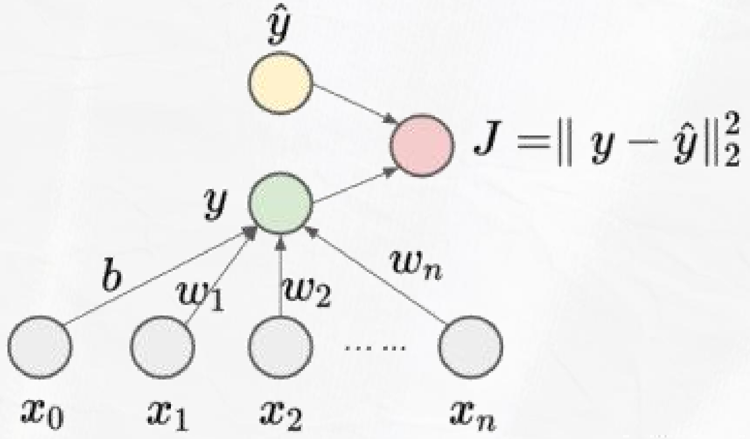
Graphical representations of linear regression

- Recall that linear regression and its cost function can be formulated as

$$y = w_1 x_1 + w_2 x_2 + \cdots w_n x_n + b$$

$$J(w_1, \cdots, w_n, b) = ||y - \hat{y}||_2^2$$

- We here represent linear regression as a computational graph
- Each input node represents one individual feature value $x_1, x_2, \cdots, x_n$ of one individual feature vector $x = \{x_1, x_2, \cdots, x_n\}$
- A constant input node $x_0 = 1$ is also utilized. Weights associated with input nodes are denoted as $w_1, w_2, \cdots, w_n$ and b
- The ground-truth target is denoted as $\hat{y}$

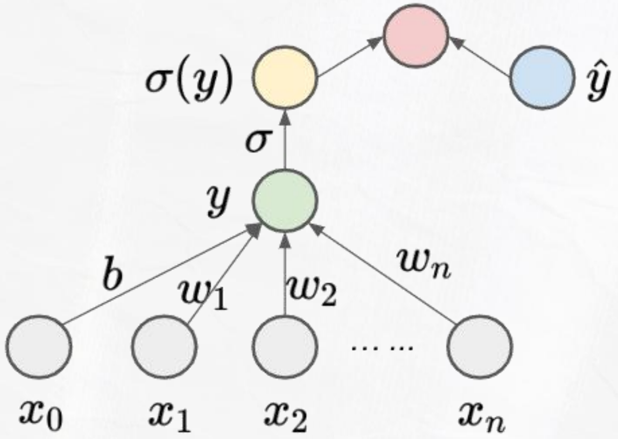


Graphical representations of logistic regression

- Similarly, the logistic classification and its cost function are

$$\sigma(y) = \sigma(w_1 x_1 + w_2 x_2 + \cdots w_n x_n + b)$$
$$J(w_1, \cdots, w_n, b) = -\hat{y} \log \sigma(y) - (1 - \hat{y}) \log(1 - \sigma(y))$$

- We here represent linear regression as a computational graph
- Each input node represents one individual feature value $x_1, x_2, \cdots, x_n$ of one individual sample $x = \{x_1, x_2, \cdots, x_n\}$
- A constant input node $x_0 = 1$ is also utilized. Weights associated with input nodes are denoted as $w_1, w_2, \cdots, w_n$ and b
- The ground-truth label (either 0 or 1) is denoted as $J = -\hat{y} \log \sigma(y) - (1 - \hat{y}) \log(1 - \sigma(y))$



Graphical representations of C-class logistic regression

- Similarly, the C-class logistic classification and its cost function are

$$y_1 = w_{11}x_1 + w_{12}x_2 + \cdots w_{1n}x_n + b_1$$
$$y_2 = w_{21}x_1 + w_{22}x_2 + \cdots w_{2n}x_n + b_2$$
$$\cdots$$
$$y_C = w_{C1}x_1 + w_{C2}x_2 + \cdots w_{Cn}x_n + b_C$$

- $y_1, y_2, \cdots, y_C$ are then normalized by the following softmax function

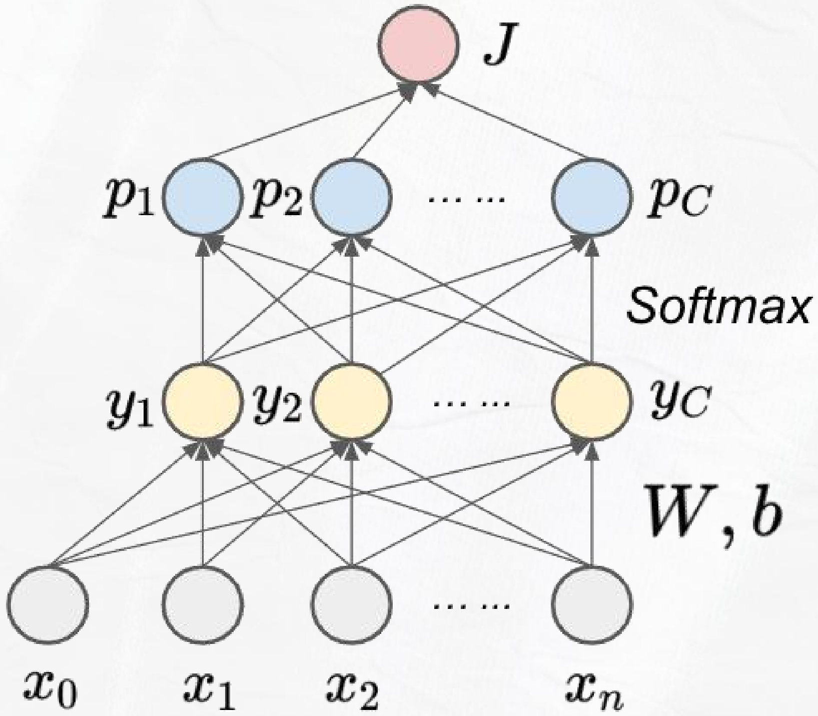
$$p_k = \frac{\exp(y_k)}{\sum_{i=1}^{C} \exp(y_i)}$$

- The loss functions are denoted as

$$J(W, b) = -\sum_{i=1}^{C} \hat{y}_i \log p_i$$

Fully-connected layer in neural networks

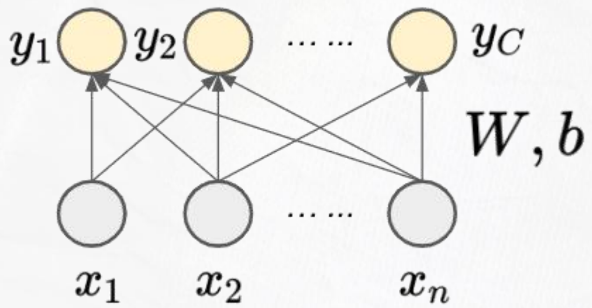
- The computational graph of multi-class (C-class) logistic classification algorithm can be drawn as



Outline

Fully-connected (linear) layer in neural networks

- The linear calculation to calculate $y_1, y_2, \cdots, y_C$ from $x_1, x_2, \cdots, x_n$ are named as ***fully-connected layer*** in neural networks
- It is one of the basic structure blocks in neural networks

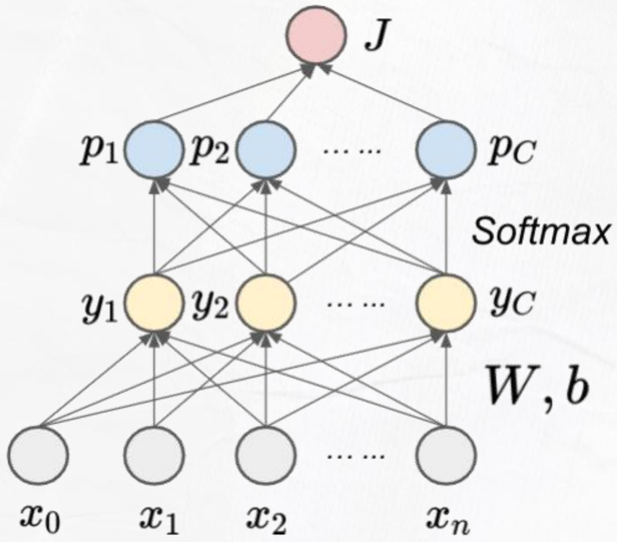


- The linear computation between x and y can be denoted as a matrix-vector multiplication $y = Wx + b$, where $W \in \mathbb{R}^{C \times n}$ and $b \in \mathbb{R}^C$ are learnable parameters and $x \in \mathbb{R}^n$ is the feature vector of one sample

$$W = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \cdots & & & \\ w_{C1} & w_{C2} & \cdots & w_{Cn} \end{bmatrix} \qquad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_C \end{bmatrix} \qquad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

Gradients of fully connected layer

- The softmax or sigmoid functions are usually called the **non-linearity (or activation)** function in neural networks
- Recall that we have the following computational graph



- Given the loss function w.r.t. W, b

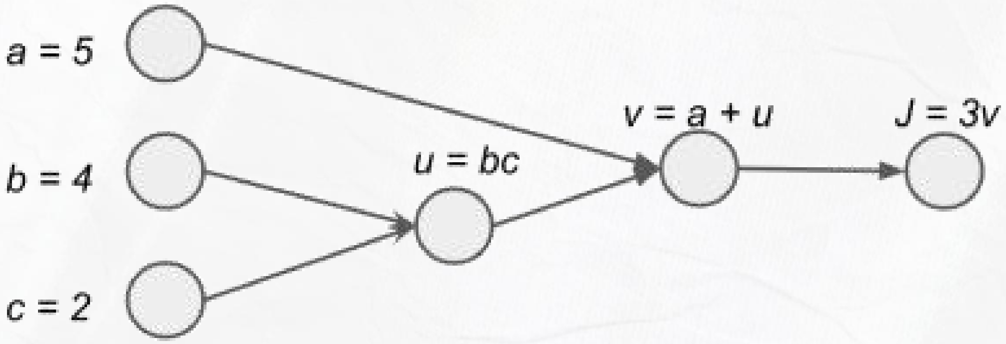
$$J(W, b) = -\sum_{i=1}^{C} \hat{y}_i \log p_i$$

- Our ultimate goal is to obtain $\frac{\partial J}{\partial W_{ij}}$ and $\frac{\partial J}{\partial b_i}$ to train the neural network

Computational graph

- Computational graph is a graphical representation of a function composition
- Example

$$u = bc, \quad v = a + u, \quad J = 3v$$



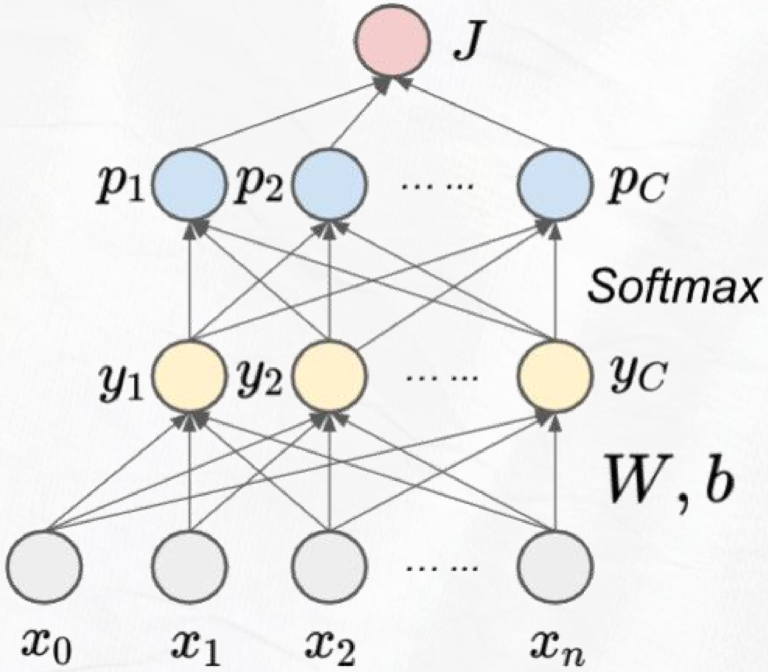
- The derivatives can be calculated backward sequentially without redundant computation

$$\frac{\partial J}{\partial v}, \quad \frac{\partial J}{\partial u} = \frac{\partial J}{\partial v}\frac{\partial v}{\partial u}, \quad \frac{\partial J}{\partial a} = \frac{\partial J}{\partial v}\frac{\partial v}{\partial a}, \quad \frac{\partial J}{\partial b} = \frac{\partial J}{\partial u}\frac{\partial u}{\partial b}, \quad \frac{\partial J}{\partial c} = \frac{\partial J}{\partial u}\frac{\partial u}{\partial c}$$

Gradients of fully connected layer

- Recall that the derivatives along computational graph can be calculated sequentially
- Eventually, we obtain

$$\frac{\partial J}{\partial p_i} \quad \frac{\partial J}{\partial y_i} \quad \frac{\partial J}{\partial W_{ij}} \quad \frac{\partial J}{\partial b_i}$$



- We therefore can calculate the following gradients sequentially and use chain rule to obtain the above gradients $\frac{\partial J}{\partial p_i} \quad \frac{\partial p_i}{\partial y_i} \quad \frac{\partial y_i}{\partial W_{ij}} \quad \frac{\partial y_i}{\partial b_i}$

- Gradients of cross-entropy loss layer $J(W,b) = -\sum_{i=1}^{C} \hat{y}_i \log p_i$ ➡ $\frac{\partial J}{\partial p_i} = \begin{cases} -\frac{1}{p_i} & \hat{y}_i = 1 \\ 0 & \hat{y}_i = 0 \end{cases}$

Gradients of softmax layer

- We are interested in calculating the gradients

$$\frac{\partial p_i}{\partial y_j} = \frac{\partial \frac{e^{y_i}}{\sum_{k=1}^{C} e^{y_k}}}{\partial y_j}$$

- We will be using quotient rule of derivatives. For $f(x) = \frac{g(x)}{h(x)}$

$$f'(x) = \frac{g'(x)h(x) - h'(x)g(x)}{[h(x)]^2}$$

- where in our case, we have

$$g = e^{y_i}, \qquad h = \sum_{k=1}^{C} e^{y_k}$$

Gradients of softmax layer

- If. $i = j$, $\frac{\partial p_i}{\partial y_j} = \frac{\partial \frac{e^{y_i}}{\sum_{k=1}^{C} e^{y_k}}}{\partial y_j} = \frac{e^{y_i}\sum_{k=1}^{C} e^{y_k} - e^{y_j}e^{y_i}}{\left(\sum_{k=1}^{C} e^{y_k}\right)^2}$

$$= \frac{e^{y_i}\left(\sum_{k=1}^{C} e^{y_k} - e^{y_j}\right)}{\left(\sum_{k=1}^{C} e^{y_k}\right)^2}$$

$$\boxed{f'(x) = \frac{g'(x)h(x) - h'(x)g(x)}{[h(x)]^2}}$$

$$\boxed{g = e^{y_i}, \qquad h = \sum_{k=1}^{C} e^{y_k}}$$

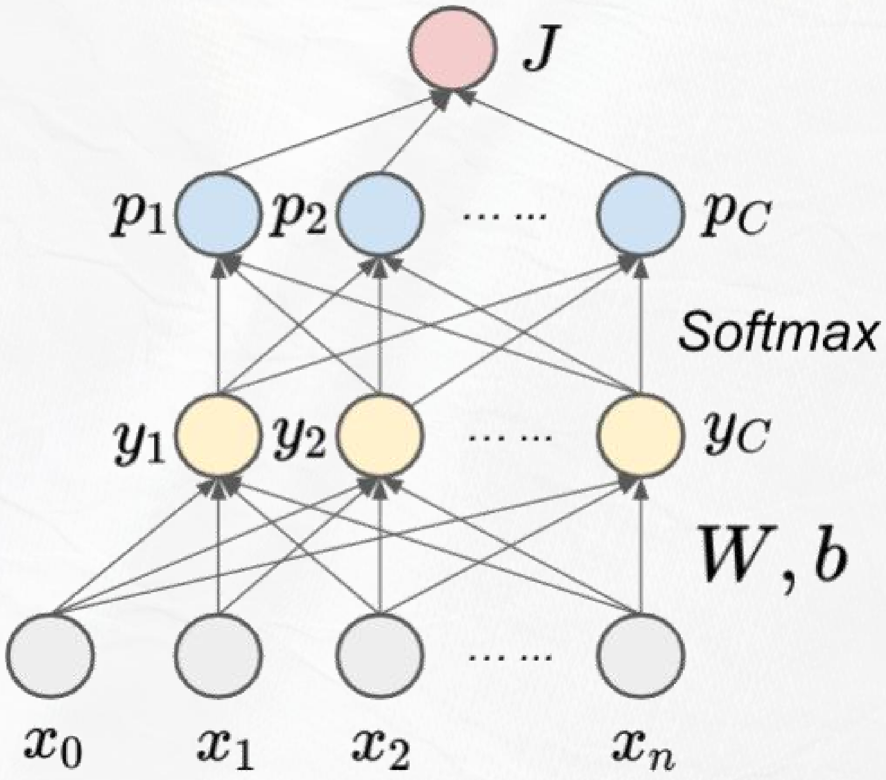
$$= \frac{e^{y_j}}{\sum_{k=1}^{C} e^{y_k}} \times \frac{\left(\sum_{k=1}^{C} e^{y_k} - e^{y_j}\right)}{\sum_{k=1}^{C} e^{y_k}}$$

$$= p_i\,(1 - p_j)$$

- If $i \neq j$, $\frac{\partial p_i}{\partial y_j} = \frac{\partial \frac{e^{y_i}}{\sum_{k=1}^{C} e^{y_k}}}{\partial y_j} = \frac{0 - e^{y_j}e^{y_i}}{\left(\sum_{k=1}^{C} e^{y_k}\right)^2}$

$$= \frac{-e^{y_j}}{\sum_{k=1}^{C} e^{y_k}} \times \frac{e^{y_i}}{\sum_{k=1}^{C} e^{y_k}}$$

$$= -p_j \cdot p_i$$



Gradients of softmax layer

- If. $i = j$, $\dfrac{\partial p_i}{\partial y_j} = \dfrac{\partial \frac{e^{y_i}}{\sum_{k=1}^{C} e^{y_k}}}{\partial y_j} = \dfrac{e^{y_i} \sum_{k=1}^{C} e^{y_k} - e^{y_j} e^{y_i}}{\left(\sum_{k=1}^{C} e^{y_k}\right)^2}$

$$= \frac{e^{y_i} \left(\sum_{k=1}^{C} e^{y_k} - e^{y_j}\right)}{\left(\sum_{k=1}^{C} e^{y_k}\right)^2}$$

$$\boxed{f'(x) = \frac{g'(x)h(x) - h'(x)g(x)}{[h(x)]^2}}$$

$$\boxed{g = e^{y_i}, \qquad h = \sum_{k=1}^{C} e^{y_k}}$$

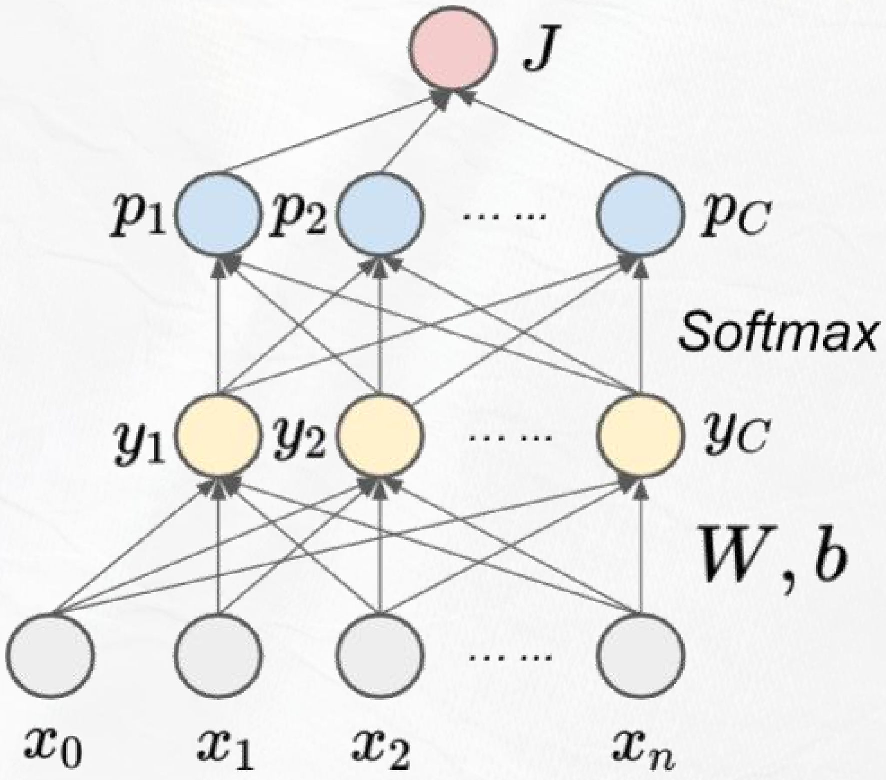
$$= \frac{e^{y_j}}{\sum_{k=1}^{C} e^{y_k}} \times \frac{\left(\sum_{k=1}^{C} e^{y_k} - e^{y_j}\right)}{\sum_{k=1}^{C} e^{y_k}}$$

$$= p_i \left(1 - p_j\right)$$

- If $i \neq j$, $\dfrac{\partial p_i}{\partial y_j} = \dfrac{\partial \frac{e^{y_i}}{\sum_{k=1}^{C} e^{y_k}}}{\partial y_j} = \dfrac{0 - e^{y_j} e^{y_i}}{\left(\sum_{k=1}^{C} e^{y_k}\right)^2}$

$$= \frac{-e^{y_j}}{\sum_{k=1}^{C} e^{y_k}} \times \frac{e^{y_i}}{\sum_{k=1}^{C} e^{y_k}}$$

$$= -p_j \cdot p_i$$



Gradients of softmax layer

- Therefore, the gradients of the softmax layer can be defined as

$$\frac{\partial p_i}{\partial y_j} = \begin{cases} p_i\left(1 - p_j\right) & \text{if} \quad i = j \\ -p_j \cdot p_i & \text{if} \quad i \neq j \end{cases}$$

- Combining gradients of the two layers, cross-entropy layer and softmax layer, the gradient of J w.r.t. y_i can be calculated as $\frac{\partial J}{\partial y_j} = \frac{\partial J}{\partial p_i}\frac{\partial p_i}{\partial y_j}$ for $\hat{y}_i = 1$

- If $i = j$ and $\hat{y}_i = 1$,

$$\frac{\partial J}{\partial y_j} = -(1 - p_i),$$

- If $i \neq j$ and $\hat{y}_i = 1$,

$$\frac{\partial J}{\partial y_j} = p_j$$

Gradients of fully-connected layer

- Recall that a fully-connected layer is calculated as

$$y = Wx + b$$

$$\begin{aligned} y_1 &= w_{11}x_1 + w_{12}x_2 + \cdots w_{1n}x_n + b_1 \\ y_2 &= w_{21}x_1 + w_{22}x_2 + \cdots w_{2n}x_n + b_2 \\ &\cdots \\ y_C &= w_{C1}x_1 + w_{C2}x_2 + \cdots w_{Cn}x_n + b_C \end{aligned}$$

- Gradients w.r.t. w_{ij} and b_i can be calculated as

$$\frac{\partial y_i}{\partial w_{ij}} = x_j \qquad \frac{\partial y_i}{\partial b_i} = 1$$

- Multiplying the gradients from the above layer according to chain rule of derivatives results in

$$\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial y_i}\frac{\partial y_i}{\partial w_{ij}} = \frac{\partial J}{\partial y_i}x_j \qquad \frac{\partial J}{\partial b_i} = \frac{\partial J}{\partial y_i}\frac{\partial y_i}{\partial b_i} = \frac{\partial J}{\partial y_i}$$

- In matrix and vector format, we have

$$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial y}x^T \text{(outer product of the two vectors)}, \qquad \frac{\partial J}{\partial b} = \frac{\partial J}{\partial y}$$

Gradients of the fully-connected layer

- We can further calculate gradients of fully-connected layers w.r.t. inputs $x_1, x_2, \cdots, x_n$

- Gradients of the fully-connected layer can be calculated as

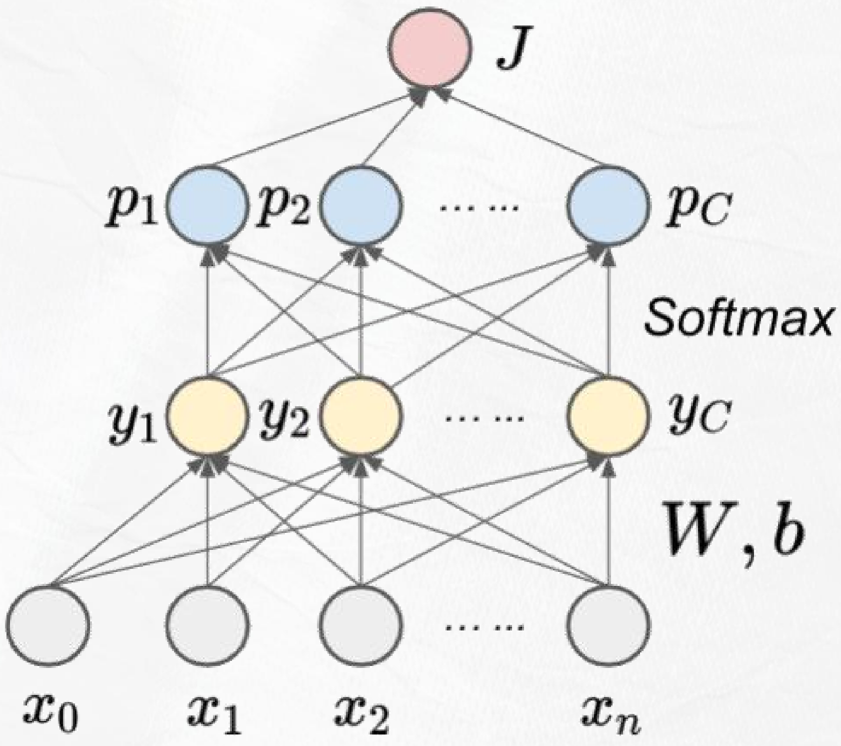
$$\frac{\partial y_i}{\partial x_j} = w_{ij}$$

- Gradients of J w.r.t. x_i therefore can be calculated as

$$\frac{\partial J}{\partial x_j} = \sum_{i=1}^{C} \frac{\partial J}{\partial y_i} \frac{\partial y_i}{\partial x_j} = \sum_{i=1}^{C} \frac{\partial J}{\partial y_i} w_{ij}$$

Converting this into a vector format, we have

$$\frac{\partial J}{\partial x} = W^T \frac{\partial J}{\partial y}$$



Forward computation and back-propagation

- Each layer's calculation can be categorized into forward and backward calculation
- Forward computation: for calculating classification probabilities from bottom layer to top layers sequentially
- Backward computation (back-propagation): for calculating gradients for parameter update from top layer to bottom layers sequentially

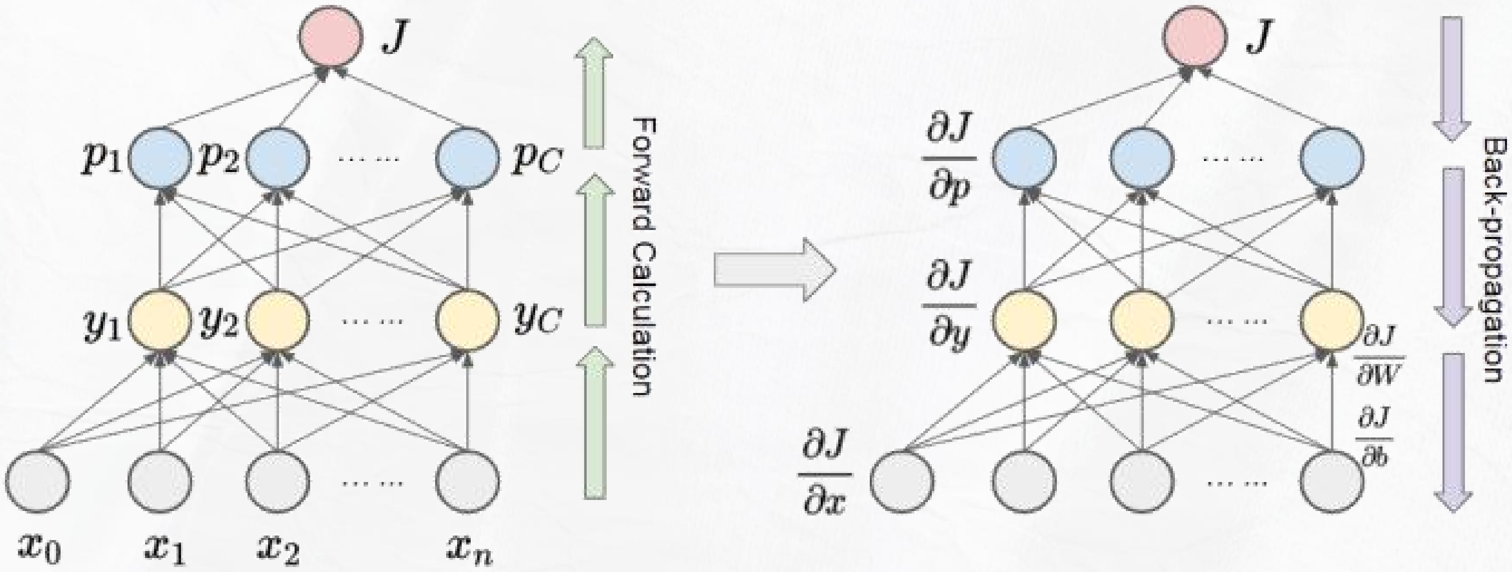
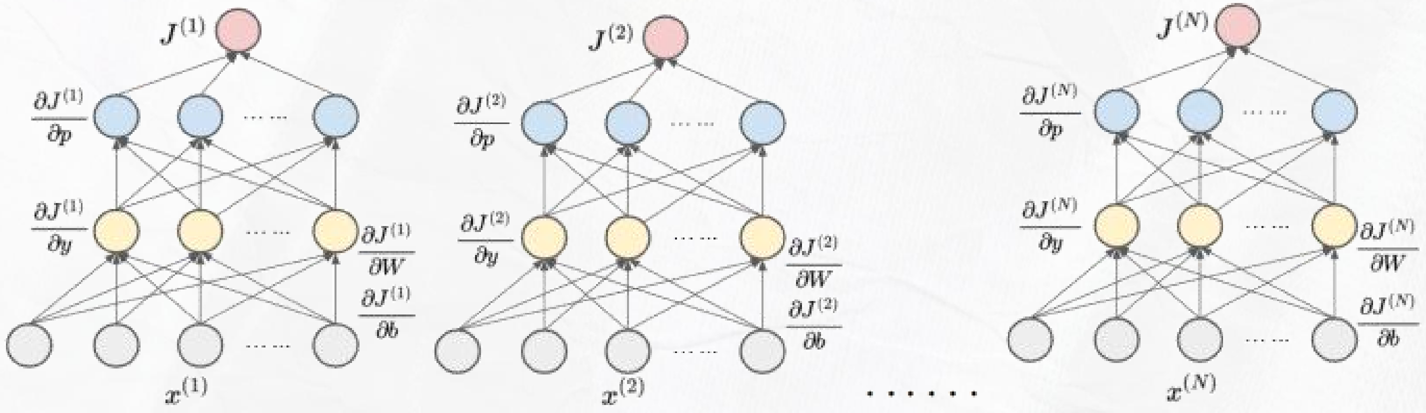


Figure: In each training iteration, (1) forward computation from bottom to top and then (2) back-propagation from top to bottom.

Gradients of a mini-batch of samples

- Recall that we mentioned that for large-scale data, the neural networks are generally trained with Stochastic Gradient Descent
- Stochastic Gradient Descent calculates derivatives J w.r.t. W, b using a mini-batch of training samples $\{x^{(1)}, x^{(2)}, \cdots, x^{(N)}\}$



- The gradients for updating parameters will be calculated as the average of the gradients of the mini-batch with batch size N,

$$J = \frac{1}{N} \sum_{i=1}^{N} \left(J^{(1)} + J^{(2)} + \cdots + J^{(N)} \right)$$

$$\frac{\partial J}{\partial W} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial J^{(i)}}{\partial W} \qquad \frac{\partial J}{\partial b} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial J^{(i)}}{\partial b}$$

Summary

- Fully-connected layer
  - ◆ Input: $x = [x_1, x_2, \cdots, x_n]$ output: $y = [y_1, y_2, \cdots, y_C]$
  - ◆ Learnable parameters: W and b
  - ◆ Forward input: x, forward output: $y = Wx + b$
  - ◆ Backward input: $\frac{\partial J}{\partial y}$
  - ◆ Backward output: $\frac{\partial J}{\partial x} = W^T \frac{\partial J}{\partial y}, \frac{\partial J}{\partial W} = \frac{\partial J}{\partial y} x^T, \frac{\partial J}{\partial b} = \frac{\partial J}{\partial y}$
- Cross-entropy loss layer
  - ◆ Input: $p = [p_1, p_2, \cdots, p_C]$, $\hat{y} = [\hat{y}_1, \hat{y}_2, \cdots, \hat{y}_C]$ output: J
  - ◆ Learnable parameters: None
  - ◆ Forward input: p, $\hat{y}$, foward output: $-\sum_{i=1}^{C} \hat{y}_i \log p_i$
  - ◆ Backward output: $\frac{\partial J}{\partial p_i} = \begin{cases} -\frac{1}{p_i} & \hat{y}_i = 1 \\ 0 & \hat{y}_i = 0 \end{cases}$
- A neural network can be considered as a structure consisting of the basic layers

Summary

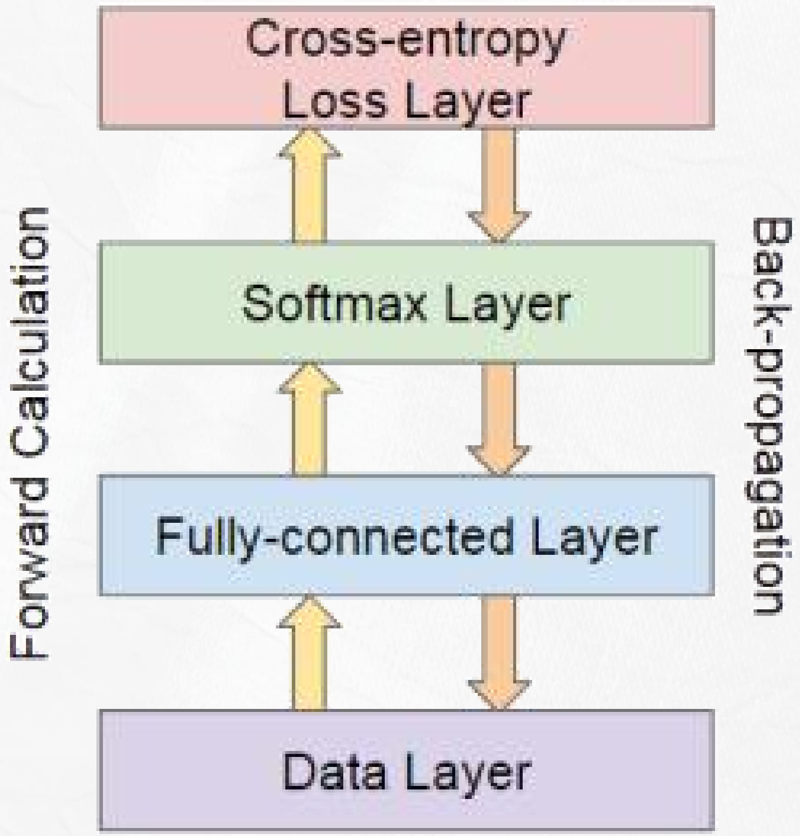
- Fully-connected layer
  - Input: $x = [x_1, x_2, \cdots, x_n]$ output: $y = [y_1, y_2, \cdots, y_C]$
  - Learnable parameters: W and b
  - Forward input: x, forward output: $y = Wx + b$
  - Backward input: $\frac{\partial J}{\partial y}$
  - Backward output: $\frac{\partial J}{\partial x} = W^T \frac{\partial J}{\partial y}, \frac{\partial J}{\partial W} = \frac{\partial J}{\partial y} x^T, \frac{\partial J}{\partial b} = \frac{\partial J}{\partial y}$
- Cross-entropy loss layer
  - Input: $p = [p_1, p_2, \cdots, p_C]$, $\hat{y} = [\hat{y}_1, \hat{y}_2, \cdots, \hat{y}_C]$ output: J
  - Learnable parameters: None
  - Forward input: p, $\hat{y}$, foward output: $-\sum_{i=1}^{C} \hat{y}_i \log p_i$
  - Backward output: $\frac{\partial J}{\partial p_i} = \begin{cases} -\frac{1}{p_i} & \hat{y}_i = 1 \\ 0 & \hat{y}_i = 0 \end{cases}$
- A neural network can be considered as a structure consisting of the basic layers

Summary

- Softmax layer:
  - Input: $y=[y_1,y_2,\cdots,y_C]$, output: $p=[p_1,p_2,\cdots,p_C]$
  - Learnable parameters: None
  - Foward input: y, forward output: $p_i=\dfrac{\exp(y_i)}{\sum_{j=1}^{C}\exp(y_j)}$
  - Backward input: $\left[\dfrac{\partial J}{\partial p_1},\dfrac{\partial J}{\partial p_2},\ldots,\dfrac{\partial J}{\partial p_C}\right]$,

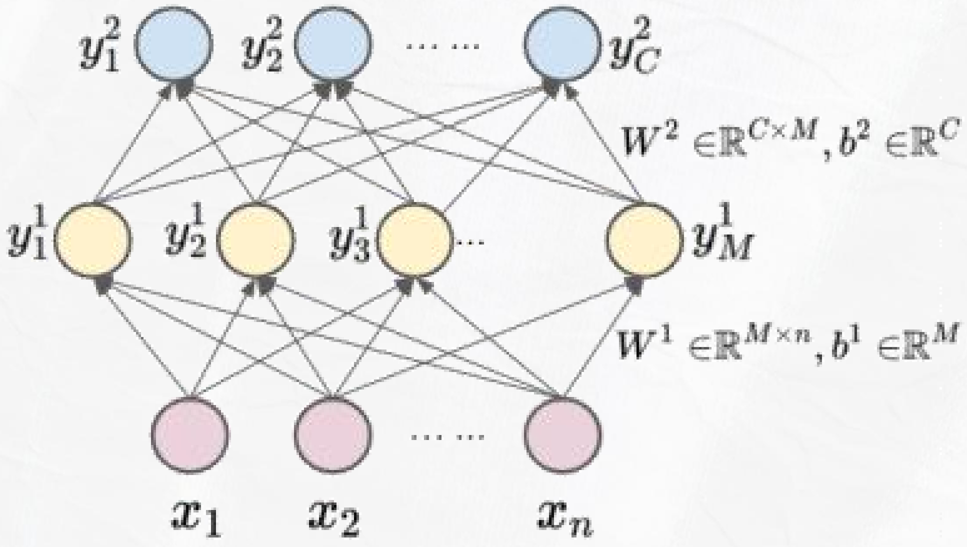
    Backward output:

$$\frac{\partial J}{\partial y_j}=\begin{cases}-\frac{\partial J}{\partial p_j}p_i(1-p_i) & \text{if } i=j\\ -\frac{\partial J}{\partial p_i}p_ip_j & \text{if } i\neq j\end{cases}$$



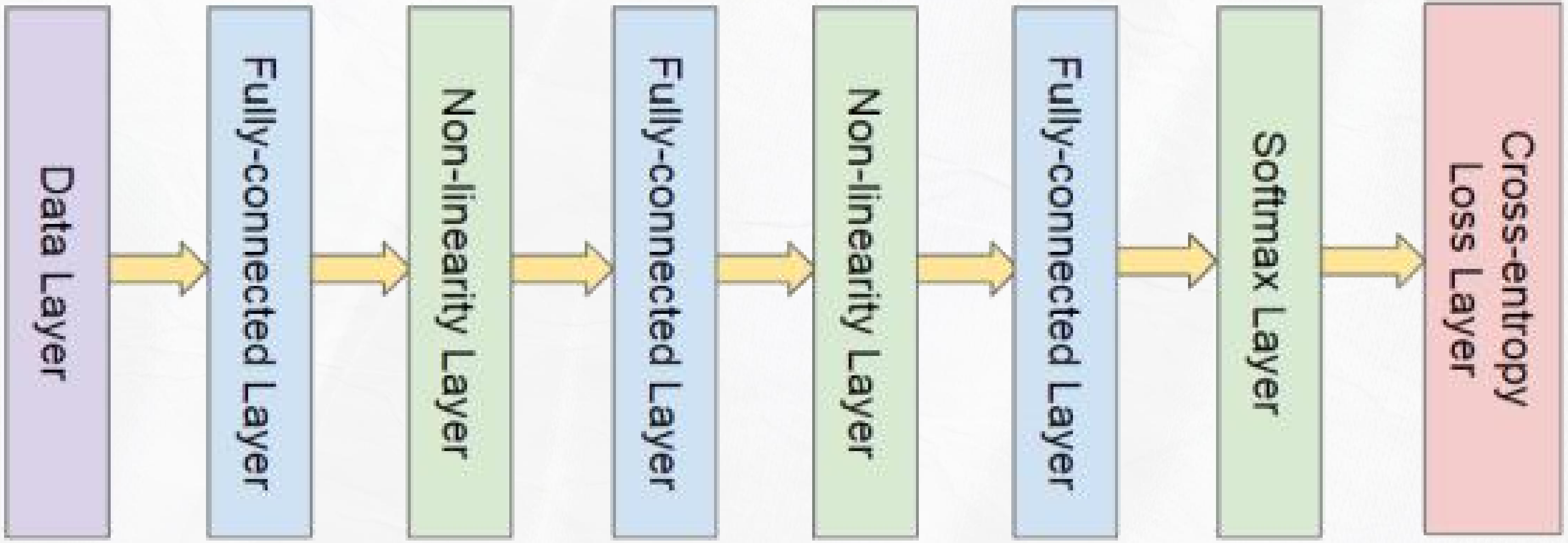
Multi-Layer Perceptron

- A neural network generally consists of multiple stacked fully-connected (linear) stacked together, where each layer has their independent parameters to learn (in general cases)
- We generally do not draw non-linearity function layers between and after fully-connected layers and do not draw $x_0, y_0^1, y_0^2, y_0^3, \cdots$
- However, the multiple fully connected layer has to be separated by non-linearity layers (e.g., softmax or sigmoid layers). Otherwise, multiple stacked fully-connected layer is equivalent to ONE fully-connected layer $y^2 = W^2(W^1x + b^1) + b^2 = [W^2W^1]\,x + [W^2b^1 + b^2]$



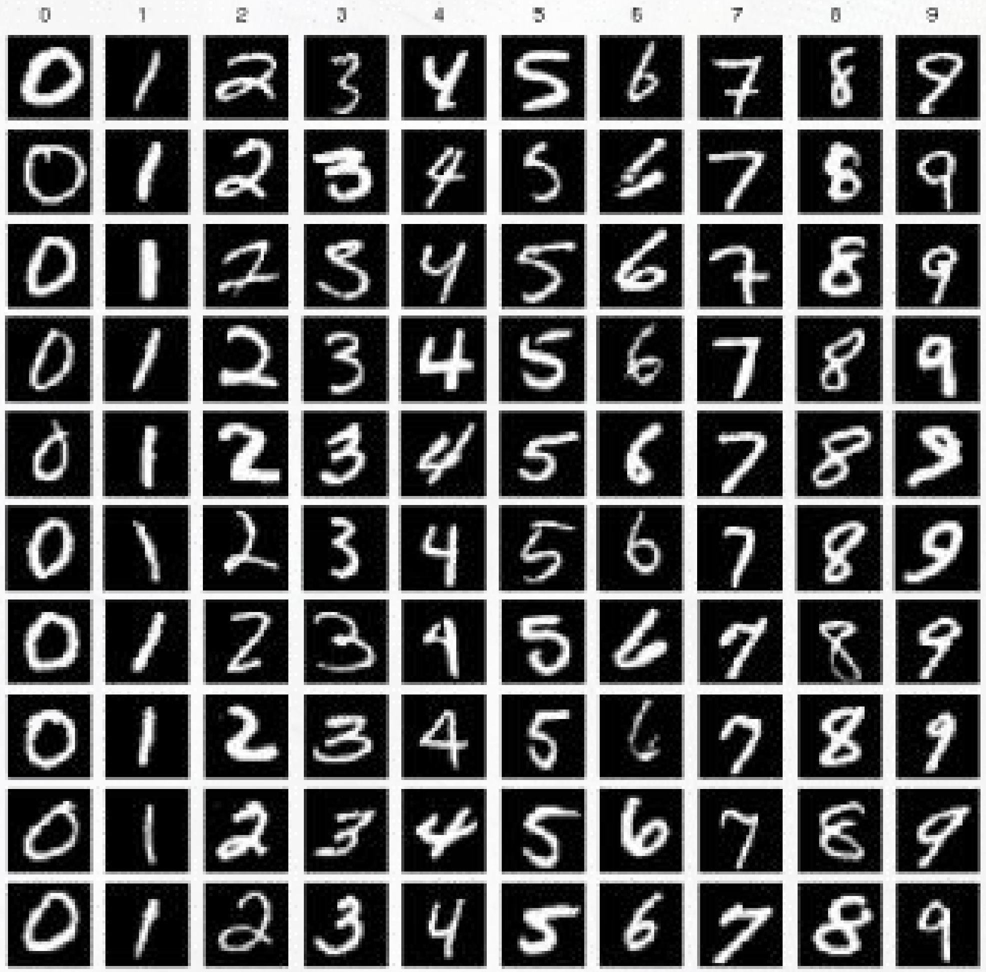
Multi-layer Perceptron

- Generally, a single linear layer with non-linearity function (e.g., logistic classification) does not have enough capacity to model the underlying function
- Neural networks with > 2 fully-connected layers can approximate any highly non-linear function
- A 3-layer Multi-Layer Perceptron (MLP) can be illustrated below



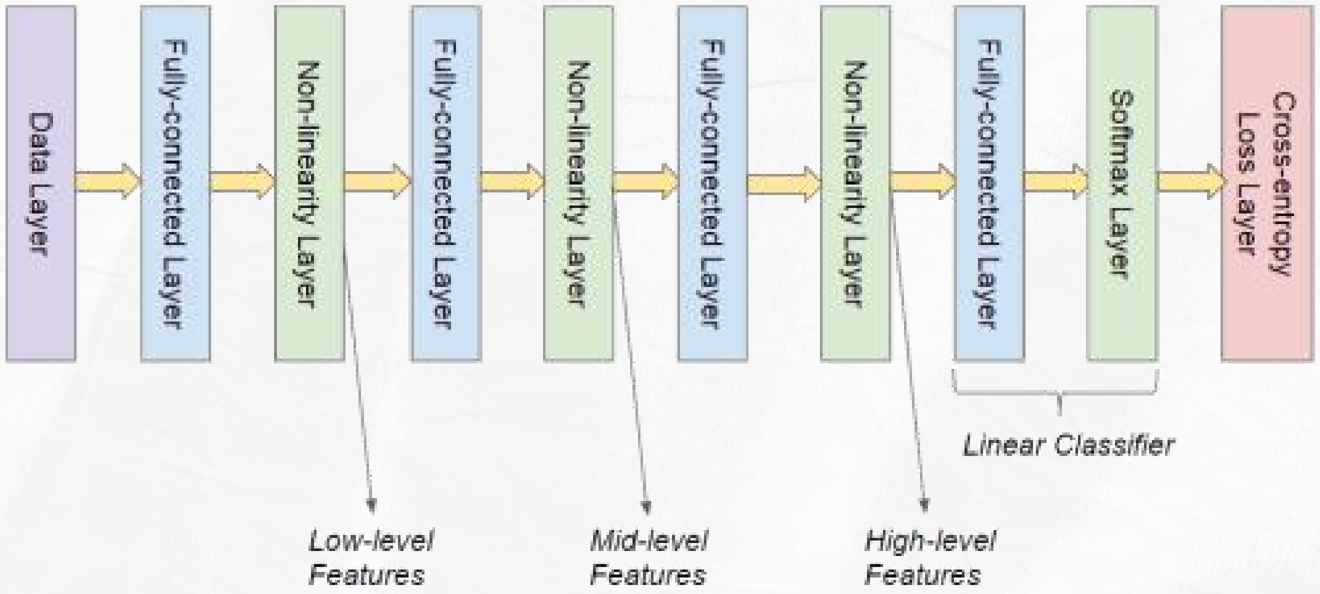
The MNIST dataset

- The MNIST dataset is a large database of handwritten digits that is commonly used for evaluating different machine learning algorithms
- It contains 60,000 training images and 10,000 testing images
- Each image is of size 32×32
- To use MLP to classify the digits, the 32×32 images can be vectorized
- into $32 \times 32 = 1024$ feature vectors as inputs



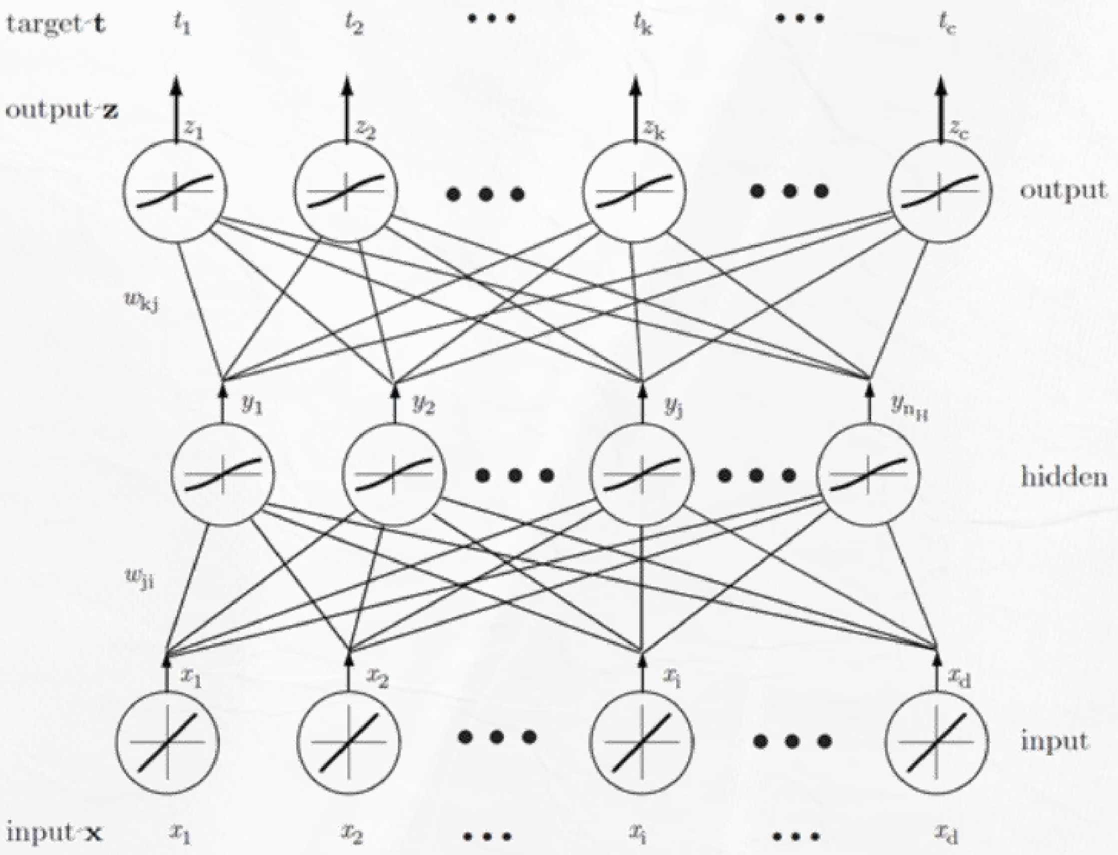
Deeply learned feature representations

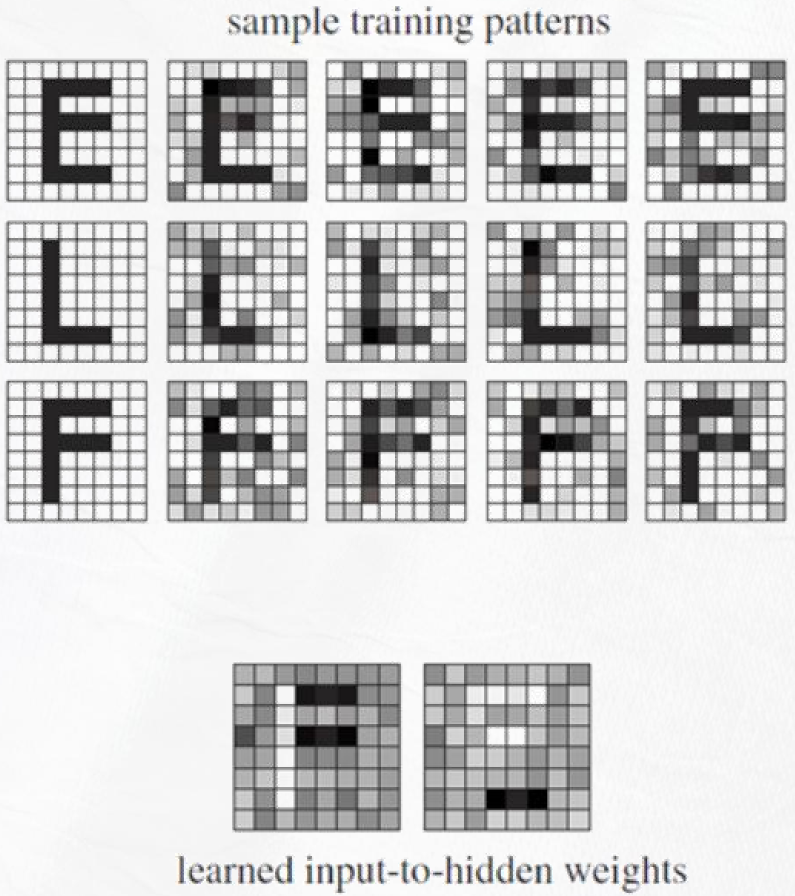
- Recall that in the begin of the course, we claimed that deep neural networks are “learning” features instead of using manually designed features
- The last fully-connected layer with the non-linearity function layer can be considered as a linear classifier
- All the previous neural layers can be considered as a series of transformations that gradually transform the input features into linearly separable features
- The low-level features captures more general information of samples of all classes
- The high-level features are closer to the final task



The learned weights

- The learned weights of each low-level neuron capture certain general patterns of all samples



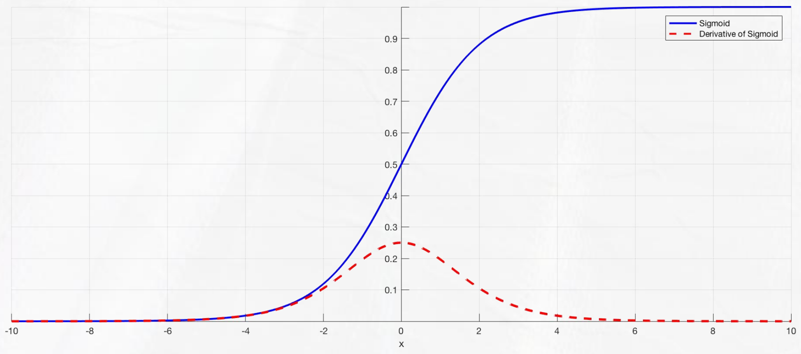


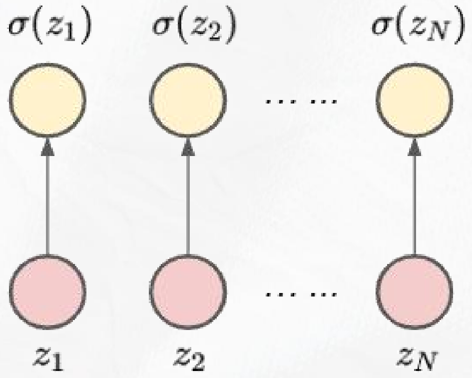
(Duda et al. Pattern Classification 2000)

Outline

Non-linearity layers

- Sigmoid (function) layer
  - Unlike softmax function, the sigmoid function only takes one value as input and output one value each time
  - Input: $z = [z_1, z_2, \cdots, z_N]$ forward output: $\sigma(z_i) = \frac{1}{1+e^{-z_i}}$
  - Backward input: $\frac{\partial J}{\partial z_i}$ backward output: $\frac{\partial J}{\partial z_i} \cdot \sigma(z_i)(1-\sigma(z_i))$
  - Use scenarios:
    - Back in 1990s-2000s, it was one of the most popular non-linearity function between fully connected layers
    - Can be used as the last layer of binary classification
    - Can be used to gate the information flow through another neuron





Non-linearity layers

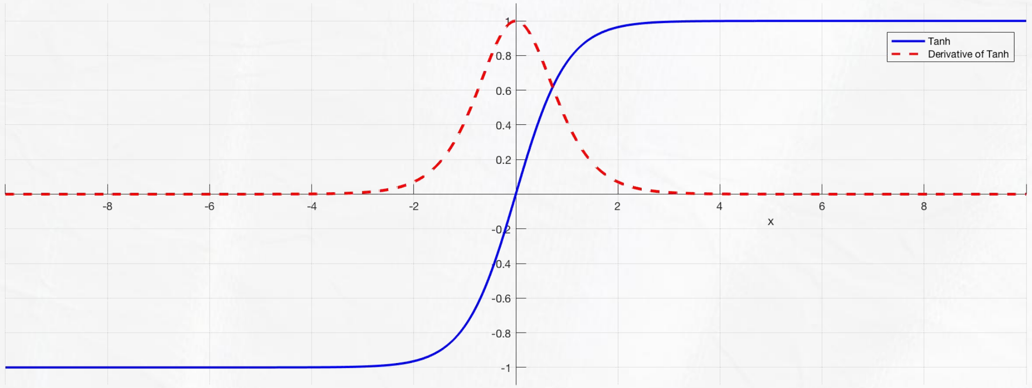
- Tanh (hyperbolic tangent function) layer
  - Sigmoid function maps $[-\infty, \infty]$ to $[0, 1]$, hyperbolic tangent function maps $[-\infty, \infty]$ to $[-1, 1]$
  - Forward input: $z = [z_1, z_2, \cdots, z_N]$, forward output:

$$g_{\tanh}(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

  - Backward input: $\frac{\partial J}{\partial z_i}$, backward output:

$$\frac{\partial J}{\partial z_i} \cdot (1 - \tanh^2(z))$$

  - It is now much less frequently used compared with sigmoid function

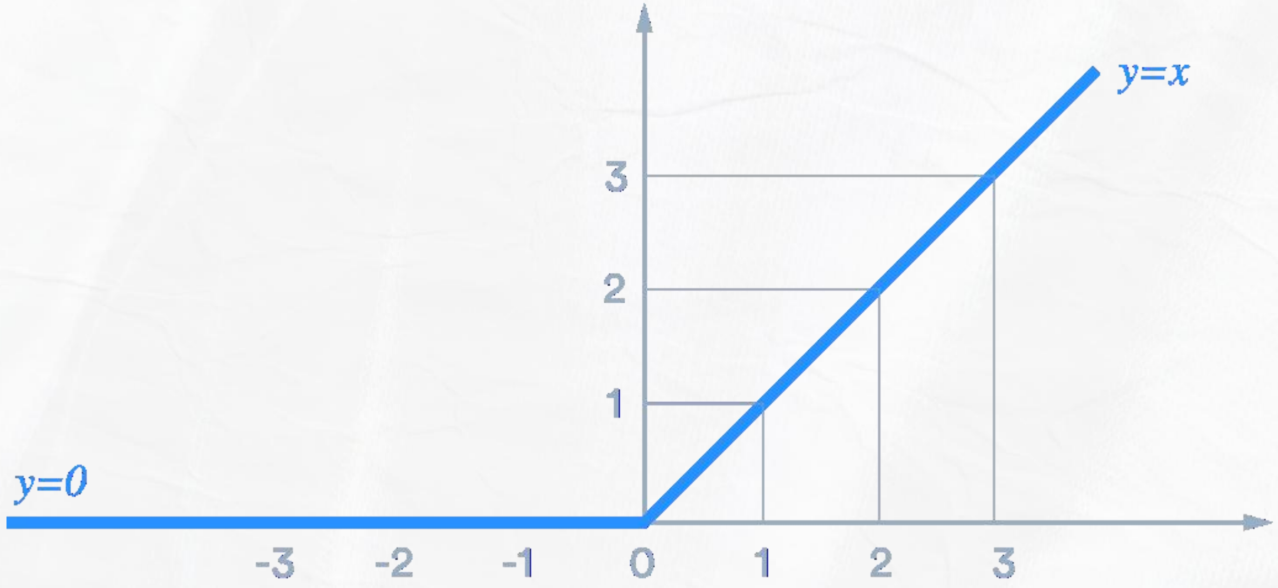


Non-linearity layers

- ReLU (Rectified Linear Unit) layer
  - One of the most frequently used non-linear function since 2012, because of its fast convergence rate
  - Forward input: $x = [x_1, x_2, \cdots, x_n]$, forward output:

$$y_i = \max(0, x_i) \text{ for } i = 1, 2, \cdots, n$$

  - Backward input: $\frac{\partial J}{\partial y_i}$, backward output: $\frac{\partial J}{\partial x_i} = \begin{cases} \frac{\partial J}{\partial y_i} & \text{if } x_i > 0 \\ 0 & \text{otherwise} \end{cases}$



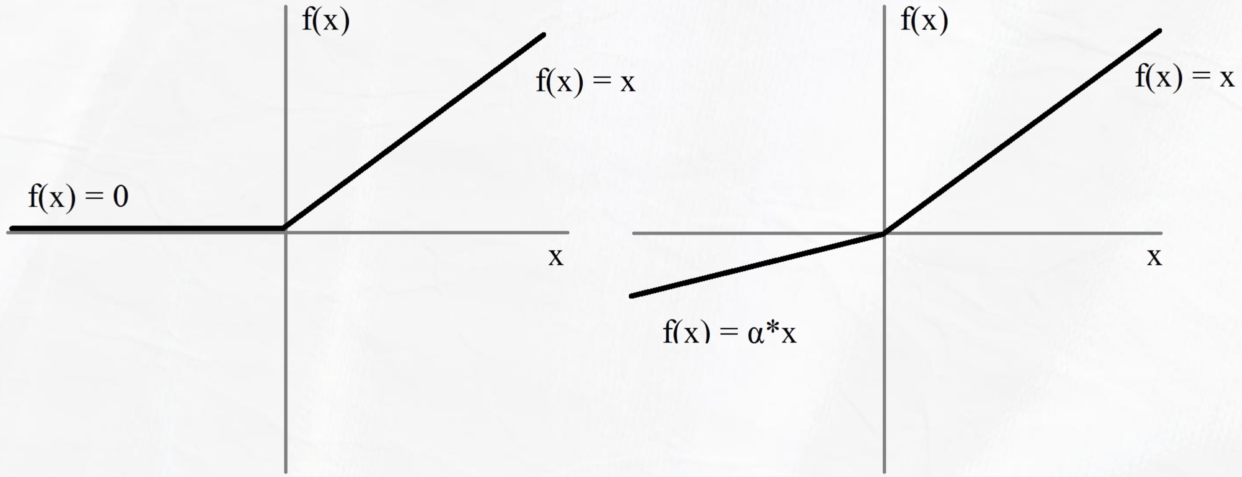
Non-linearity layers

- Leaky ReLU (Rectified Linear Unit) layer
  - Leaky ReLU is an improved version of the ReLU layer. It solves the problem of ReLU of having no gradients when the input is less than 0
  - Forward input: $x = [x_1, x_2, \cdots, x_n]$, forward output:

$$y_i = \begin{cases} \alpha x_i & \text{if } x_i < 0 \\ x_i & \text{if } x_i \geq 0 \end{cases} \quad \text{for } i = 1, 2, \cdots, n$$

  where α is a constant

  - Backward input: $\frac{\partial J}{\partial y_i}$, backward output: $\frac{\partial J}{\partial x_i} = \begin{cases} \alpha \frac{\partial J}{\partial y_i} & \text{if } x_i < 0 \\ \frac{\partial J}{\partial y_i} & \text{if } x_i \geq 0 \end{cases}$



Non-linearity layers

- PReLU layer
  - PReLU takes one step further by making the coefficient of leakage α to be learned during network training
  - Forward input: $x = [x_1, x_2, \cdots, x_n]$, forward output:

$$y_i = \begin{cases} \alpha x_i & \text{if } x_i < 0 \\ x_i & \text{if } x_i \geq 0 \end{cases} \quad \text{for } i = 1, 2, \ldots, n$$

  where α is a learnable constant

  - Backward input: $\frac{\partial J}{\partial y_i}$, backward output:

$$\frac{\partial J}{\partial x_i} = \begin{cases} \alpha \frac{\partial J}{\partial y_i} & \text{if } x_i < 0 \\ \frac{\partial J}{\partial y_i} & \text{if } x_i \geq 0 \end{cases}$$

  - Parameter gradients: $\frac{\partial J}{\partial \alpha} = \sum_{i=1}^{n} \mathbf{1}(x_i < 0) x_i \cdot \frac{\partial J}{\partial y_i}$

Loss layers

- Mean Squared Error (MSE)/L2 loss layer
  - Generally used for regression problem
  - Forward inputs: $z^{(1)}, z^{(2)}, \cdots, z^{(N)}$ and ground-truth $\hat{z}^{(1)}, \hat{z}^{(2)}, \cdots, \hat{z}^{(N)}$, forward output

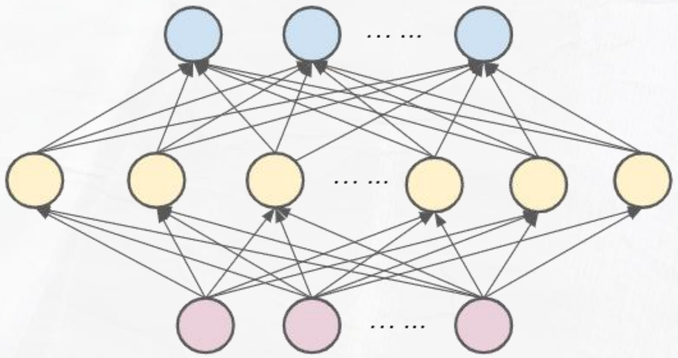
$$J = \frac{1}{2N} \sum_{i=1}^{N} \left(z^{(i)} - \hat{z}^{(i)} \right)^2$$

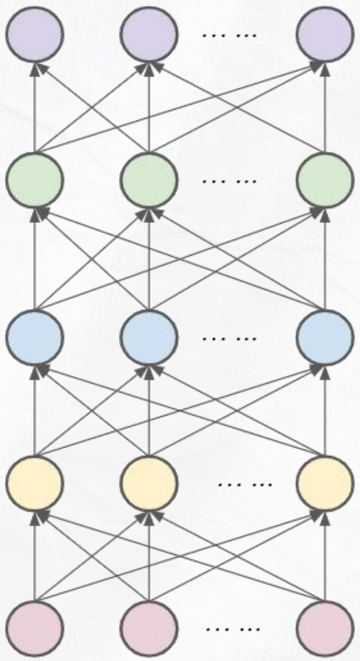
  - Backward output: $\dfrac{\partial J}{\partial z^{(i)}} = \dfrac{1}{N} \left(z^{(i)} - \hat{z}^{(i)} \right)$

- L1 loss layer
  - Also commonly used for regression problem, especially when there are many outliers
  - Forward inputs: $z^{(1)}, z^{(2)}, \cdots, z^{(N)}$ and ground-truth $\hat{z}^{(1)}, \hat{z}^{(2)}, \cdots, \hat{z}^{(N)}$, forward output
  - Backward output: $J = \dfrac{1}{N} \sum_{i=1}^{N} \left| z^{(i)} - \hat{z}^{(i)} \right|$

Why do we need “deep” neural networks

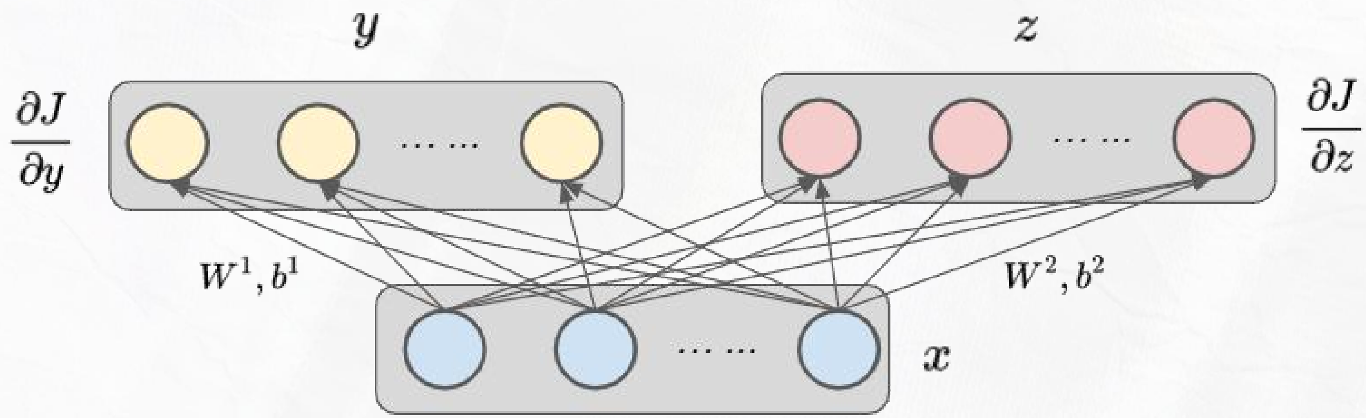
- Theoretically, a three-layer neural network can approximate any non-linear function. Logistic regression/classification can all be considered as a “shallow” three-layer neural network
- Then, why do we need “deep” neural networks?
- If the desired function is very complex, with three-layer neural networks, it might require an exponentially increasing number of neurons in the hidden layers to well approximate the function
- However, with many layers, a small number of neurons in each layer would be enough to approximate the desired function





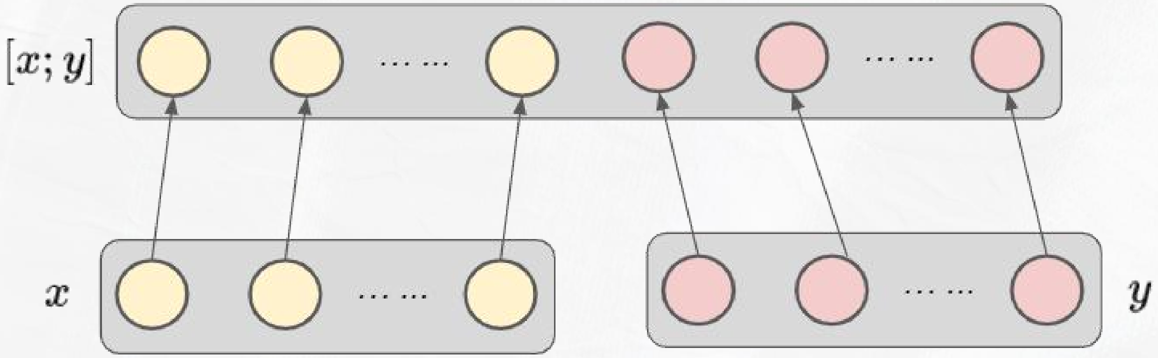
Branching and concatenation

- A group of neurons can be connected by two different fully-connected layers (branches)



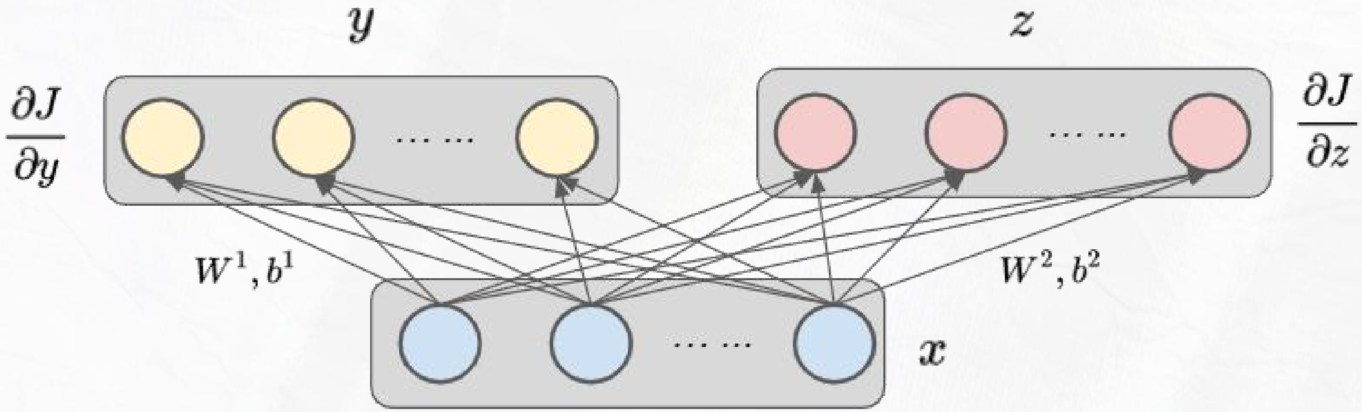
$$\frac{\partial J}{\partial x} = (W^1)^T \frac{\partial J}{\partial y} + (W^2)^T \frac{\partial J}{\partial z}$$

- Two feature vectors (branches) can also concatenate to generate a longer feature vector



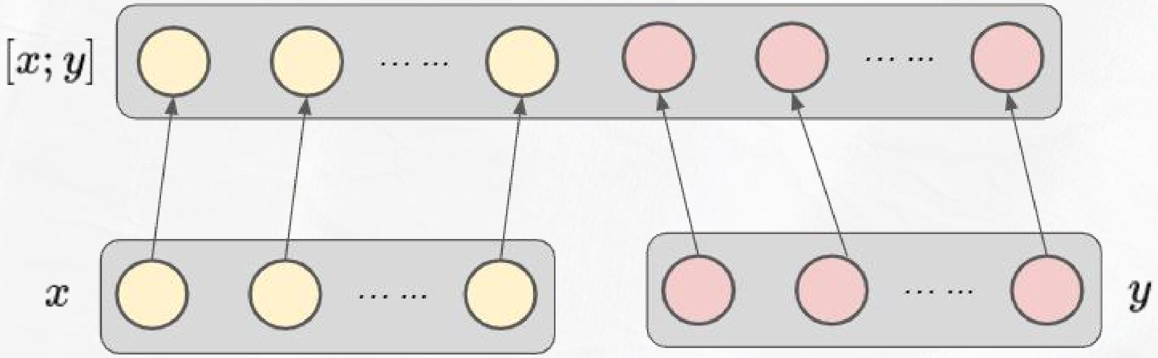
Branching and concatenation

- A group of neurons can be connected by two different fully-connected layers (branches)



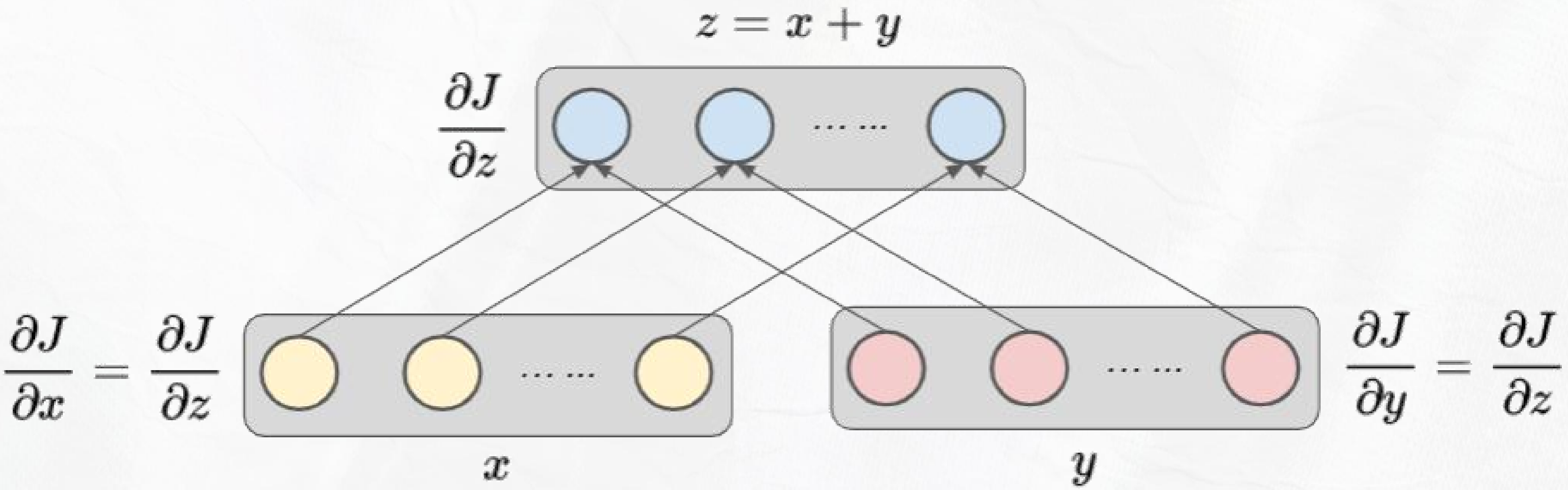
$$\frac{\partial J}{\partial x} = (W^1)^T \frac{\partial J}{\partial y} + (W^2)^T \frac{\partial J}{\partial z}$$

- Two feature vectors (branches) can also concatenate to generate a longer feature vector



Addition of two groups of neurons

- The two vectors of neurons can be added to obtain a group of neurons



Batch Normalization (BN) Layer

- Each dimension of the input feature vectors should be normalized by subtracting the mean over the entire training set and then optionally divided by the standard deviation over the entire training set

- Recall that in mini-batch gradient descent, we train neural networks with mini-batches of samples and each mini-batch might have different feature distributions (named ***covariance shift***) because of the small mini-batch size

- To handle different feature distributions in each iteration, the neural networks need to jointly handle feature distribution variations and correctly classify the training samples, which prevent the network from focusing on only learning for classification

Batch Normalization (BN) Layer (cont’d)

- The BN layer normalizes each input feature vector of a mini-batch
- Forward input: feature vector $x \in \mathbb{R}^n$ in a mini-batch $\mathcal{B}$

$$\hat{\mu}_{\mathcal{B}} \leftarrow \frac{1}{|\mathcal{B}|}\sum_{x\in\mathcal{B}} x \quad \text{and} \quad \hat{\sigma}_{\mathcal{B}}^2 \leftarrow \frac{1}{|\mathcal{B}|}\sum_{x\in\mathcal{B}} (x-\mu_{\mathcal{B}})^2 + \epsilon$$

$$\mathrm{BN}(x) = \gamma \odot \frac{x-\hat{\mu}_{\mathcal{B}}}{\hat{\sigma}_{\mathcal{B}}+\epsilon} + \beta \text{ (“}\odot\text{”: element-wise multiplication)}$$

- To address the fact that in some cases the activations may actually need to differ from standardized data, BN also introduces learnable scaling $\gamma \in \mathbb{R}^n$ and offset $\beta \in \mathbb{R}^n$
- We add a small constant $\epsilon > 0$ to the variance estimate to ensure never dividing by zero
- Training:
  - ◆ In practice, instead of estimating mean and standard deviation of each mini-batch, we keep a running estimate of the batch feature mean and standard deviation

$$\hat{x}^{(t+1)} = (1 - \text{ momentum }) \times \hat{x}^{(t)} + \text{momentum} \times \hat{x}$$

Batch Normalization (BN) Layer

- Each dimension of the input feature vectors should be normalized by subtracting the mean over the entire training set and then optionally divided by the standard deviation over the entire training set

- Recall that in mini-batch gradient descent, we train neural networks with mini-batches of samples and each mini-batch might have different feature distributions (named ***covariance shift***) because of the small mini-batch size

- To handle different feature distributions in each iteration, the neural networks need to jointly handle feature distribution variations and correctly classify the training samples, which prevent the network from focusing on only learning for classification

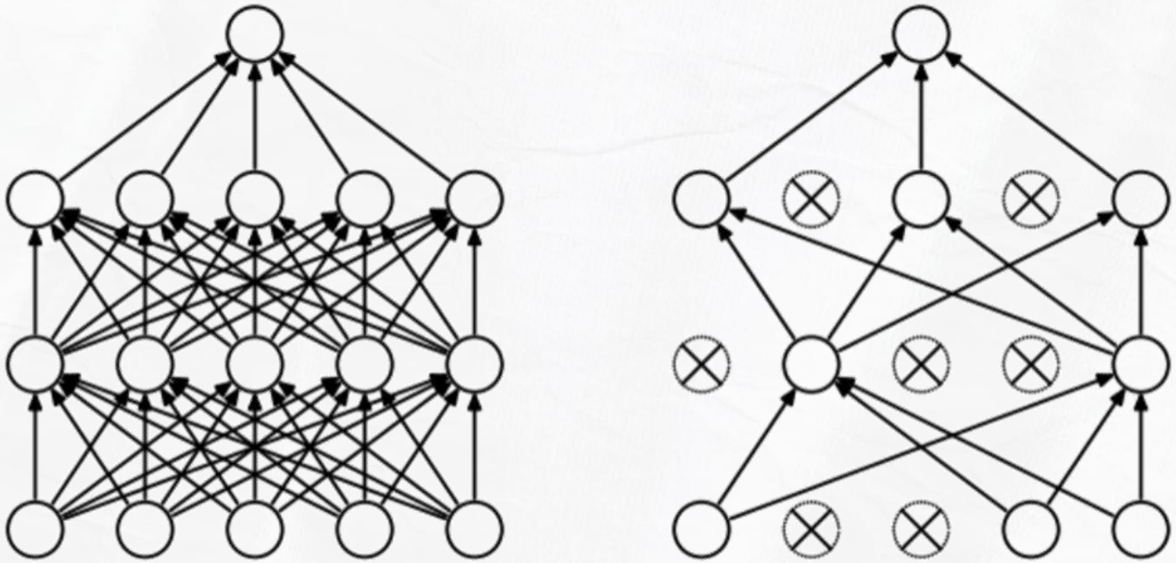
Ioffe and Szegedy,"Batch normalization: Accelerating deep network training by reducing internal covariate shift", ICML 2015

Batch Normalization (BN) Layer (cont’d)

- Testing:
  - There are three choices of mean and standard deviation during testing
    1. Calculate the mean and standard deviation from the current batch
    2. Use the running estimate of mean and standard deviation during training
    3. Calculate the mean and standard deviation from the entire training set or a relative large sub-set of the training set layer by layer
- Advantages of using BN layers
  - **Network trains faster:** Each training iteration will actually be slower because of the extra calculations. However, it should converge much more quickly, so training should be faster overall
  - **Allows higher learning rates:** Gradient descent usually requires small learning rates for the network to converge. And as networks get deeper, their gradients get smaller during back propagation so they require even more iterations. Using batch normalization allows us to use much higher learning rates, which further increases the speed at which networks train
  - **Makes weights easier to initialize:** Batch normalization seems to allow us to be much less careful about choosing our initial starting weights
  - **Makes more activation functions viable:** For instance, Sigmoids lose their gradient pretty quickly when used in neural networks

Dropout layer

- Deep neural networks can have many large model capacity because of their deep structures. They are likely to overfit on small-scale dataset
- Some neurons easily become “inactive” during training, because a small number of other neurons can perform well on the training set
- To mitigate the problem, the dropout layer randomly sets proportion of $p \in [0, 1]$ neurons to zero and force the following the layer to use the remaining neuron responses for completing the prediction task



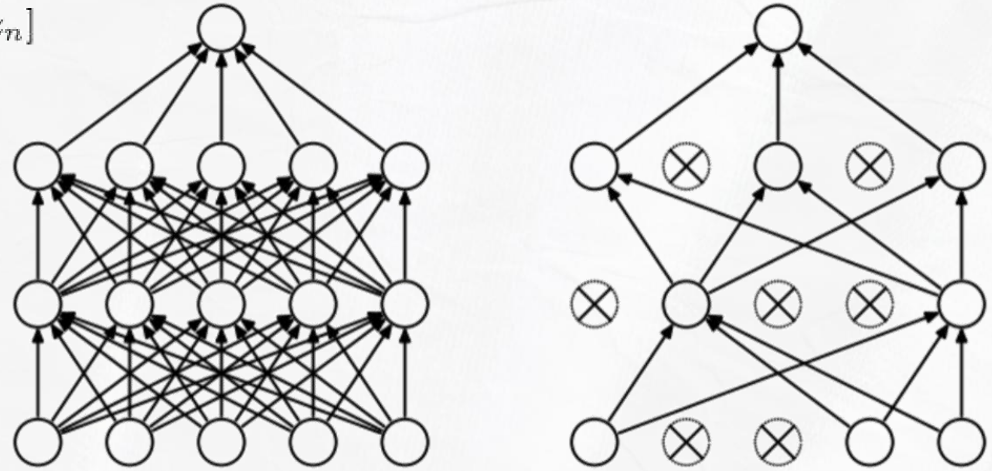
(a) Standard Neural Net (b) After applying dropout.

Dropout layer

- Training:
  - Forward input: dropout ratio *p*, input feature vector $z = [z_1, z_2, \cdots, z_n]$, forward output: randomly set proportion *p* of feature values in $[z_1, z_2, \cdots, z_n]$ to zero to obtain *y*, then multiplied by 1/(1-*p*)
  - Backward input: $\frac{\partial J}{\partial y}$, backward output:

$$\frac{\partial J}{\partial z_i} = \begin{cases} \frac{\partial J}{\partial z_i} & \text{if } z_i \text{ is not dropped out in forward computation} \\ 0 & \text{if } z_i \text{ is dropped out in forward computation} \end{cases}$$

- Testing/Inference:
  - Forward input: dropout ratio *p*, input feature vector $z = [z_1, z_2, \cdots, z_n]$;
  - forward output: $[z_1, z_2, \cdots, z_n]$



(a) Standard Neural Net (b) After applying dropout.

Dropout layer

- In general, when using dropout layers, training errors (losses) will **INCREASE**
- For small-scale datasets, dropout layers are effective and **decrease** testing errors
- However, since the dropout layer is designed to prevent overfiting, it shows **LESS to NONE** effectiveness on large-scale datasets

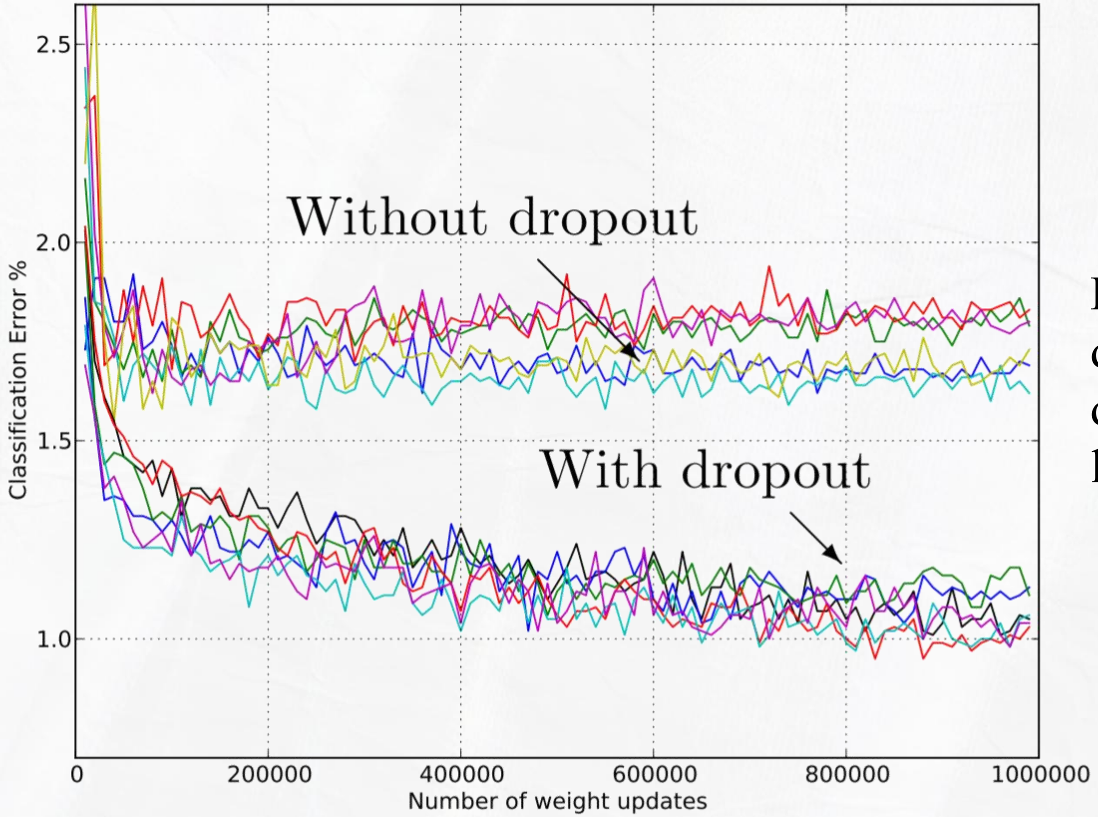
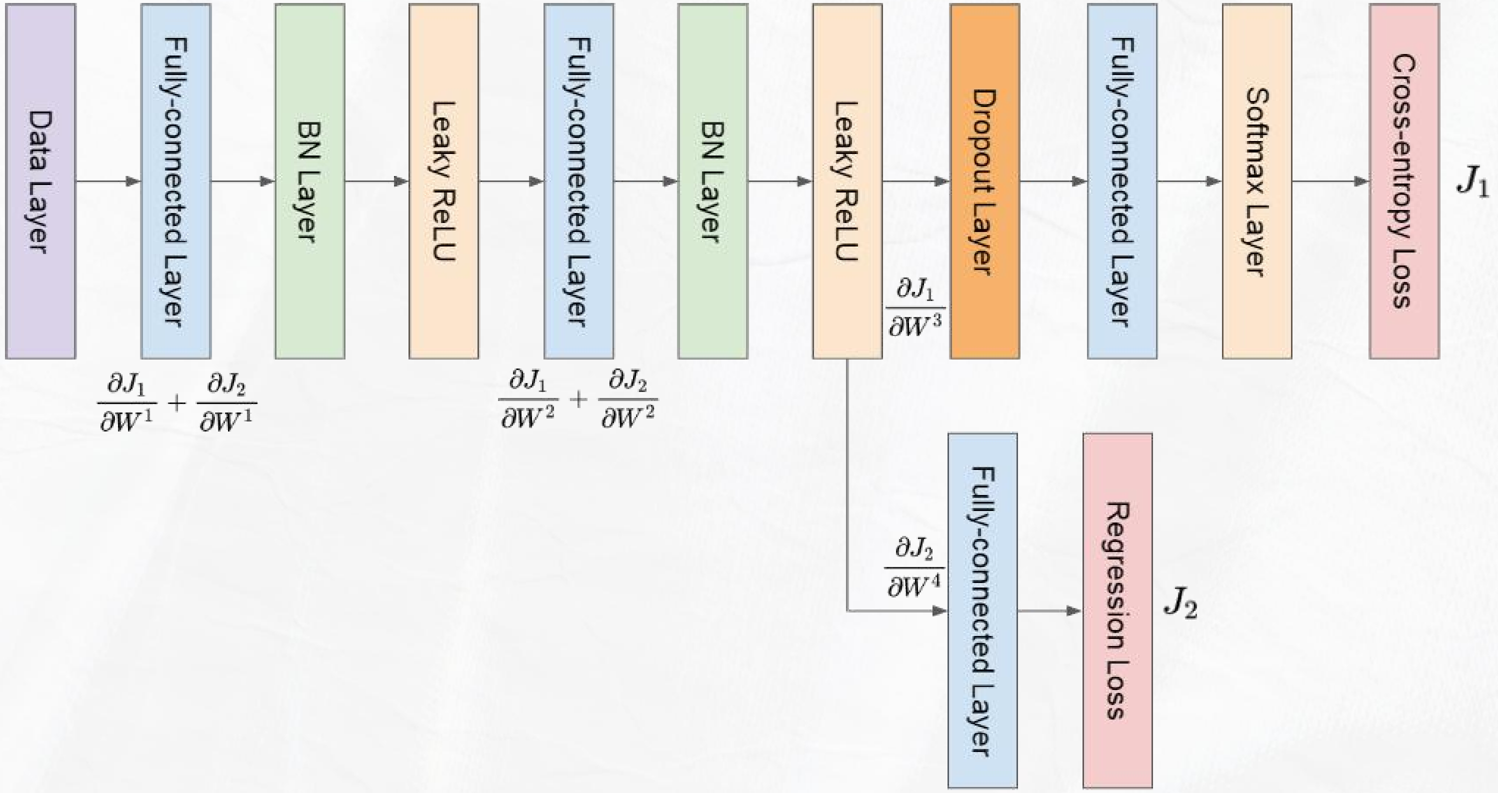


Figure: Test error on MINIST datasets for different architectures with and without dropout. The networks have 2 to 4 hidden layers each with 1024 to 2048 units.

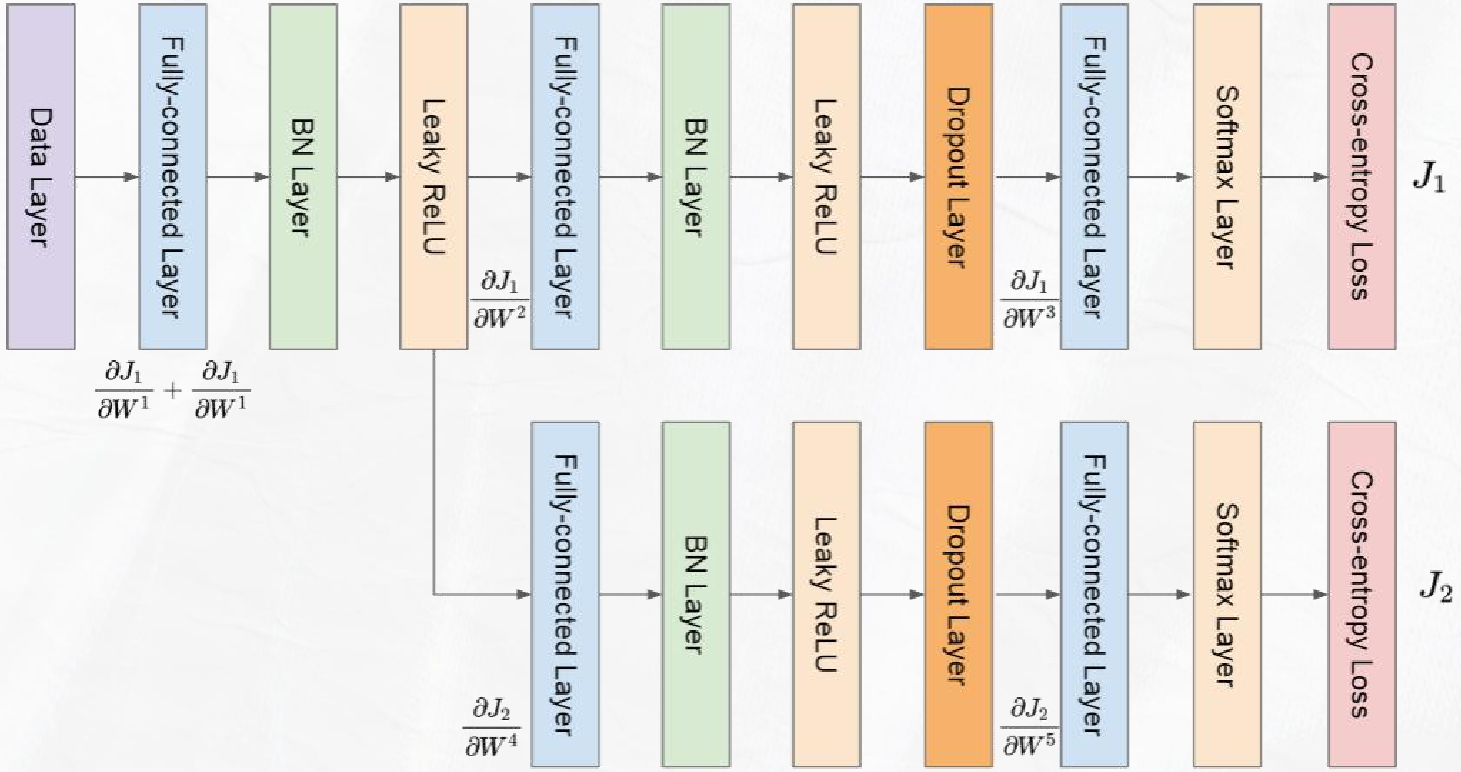
Modern MLPs

- A modern MLP can consist of several fully-connected layers, each of which is followed by a BN layer and then a PReLU or Leaky ReLU non-linearity layer
- Each dimension of the input feature dimension should be normalized by first subtracting the mean and then dividing by the standard deviation
- An MLP can have multiple losses either all at the topmost layer or at different layers



Modern MLPs (cont’d)

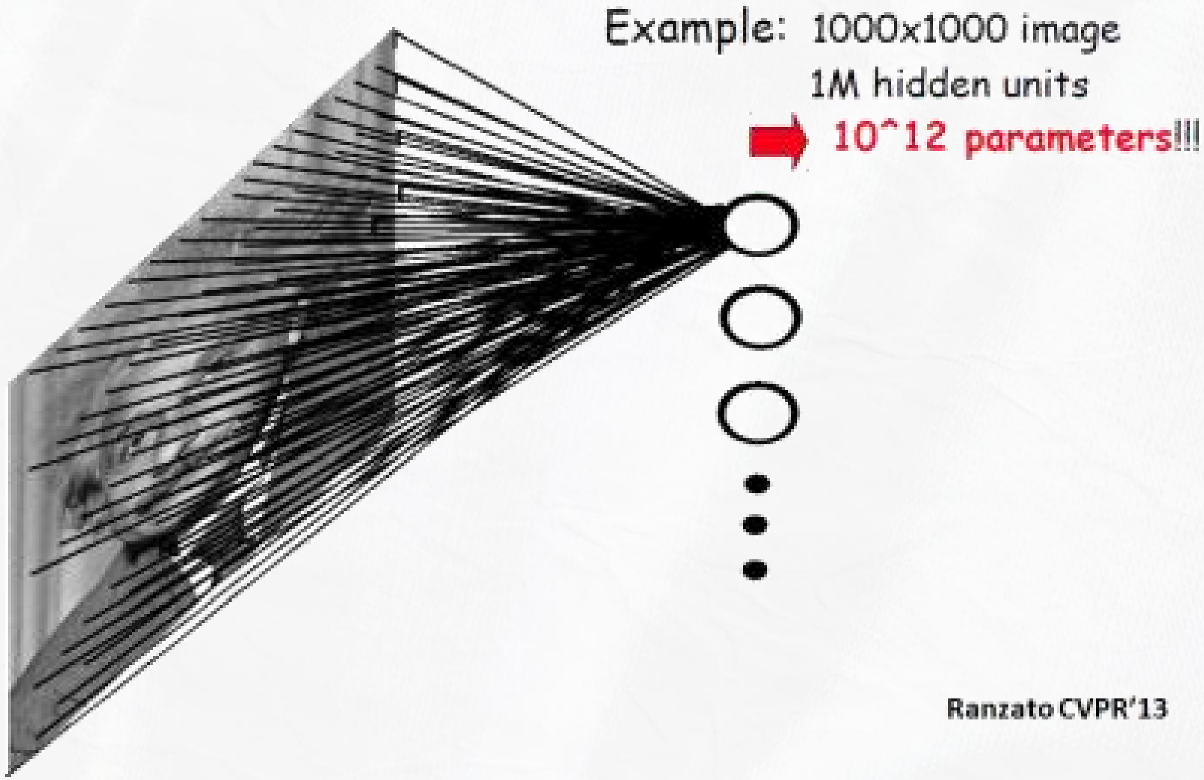
- A modern MLP can consist of several fully-connected layers, each of which is followed by BN layer and then PReLU or Leaky ReLU non-linearity layer
- Each dimension of the input feature dimension should be normalized by first subtracting the mean and then dividing by the standard deviation
- An MLP can have multiple losses either all at the topmost layer or at different layers



Outline

Motivation of convolutional neural networks (CNNs)

- Although deeper neural networks have larger representation capability and better generalization, it is difficult to extend the multi-layer feed-forward neural networks to very deep, since every layer is fully connected.
- For example, given a large image, the number of parameters could be very large. How to alleviate such limitation?
- People resort to two tricks:
  - Sparse connection
  - Shared parameters

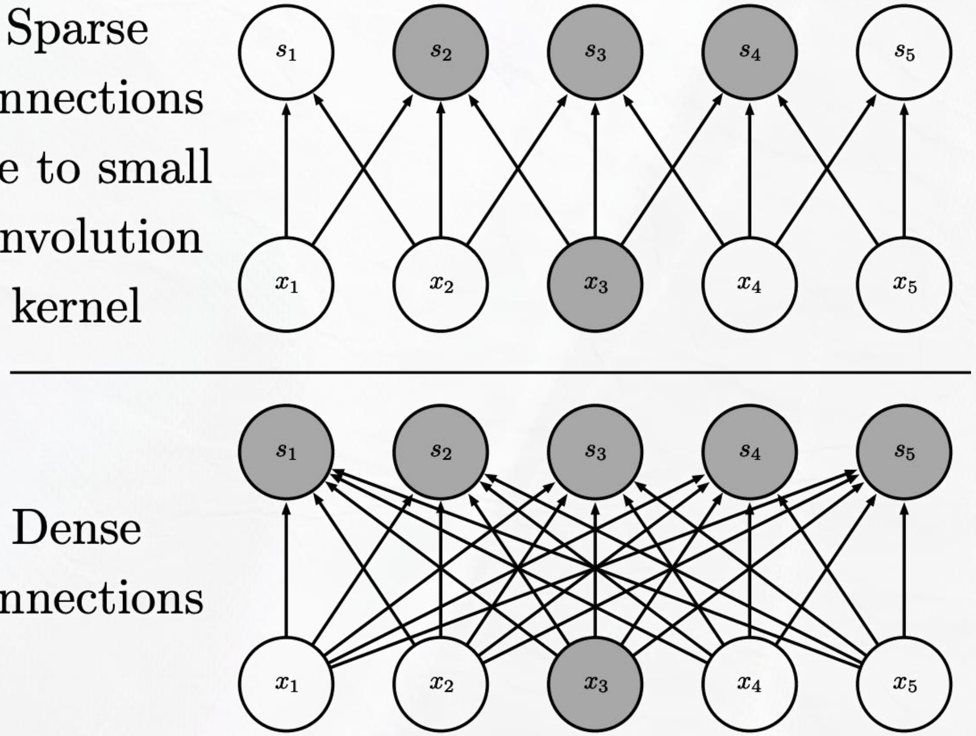


Sparse connection

- Each input neuron only connects to partial output neurons
- Each output neuron only connects to a few neighboring input neurons. And the range of input neurons is called **receptive field.**

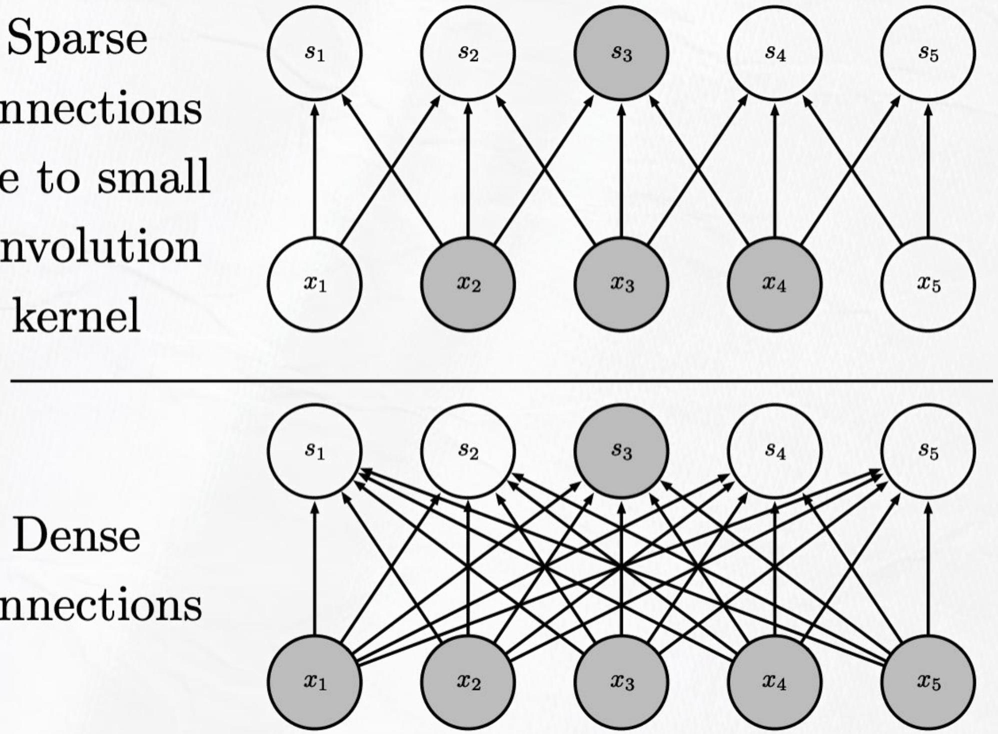
Sparse connections due to small convolution kernel

Dense connections



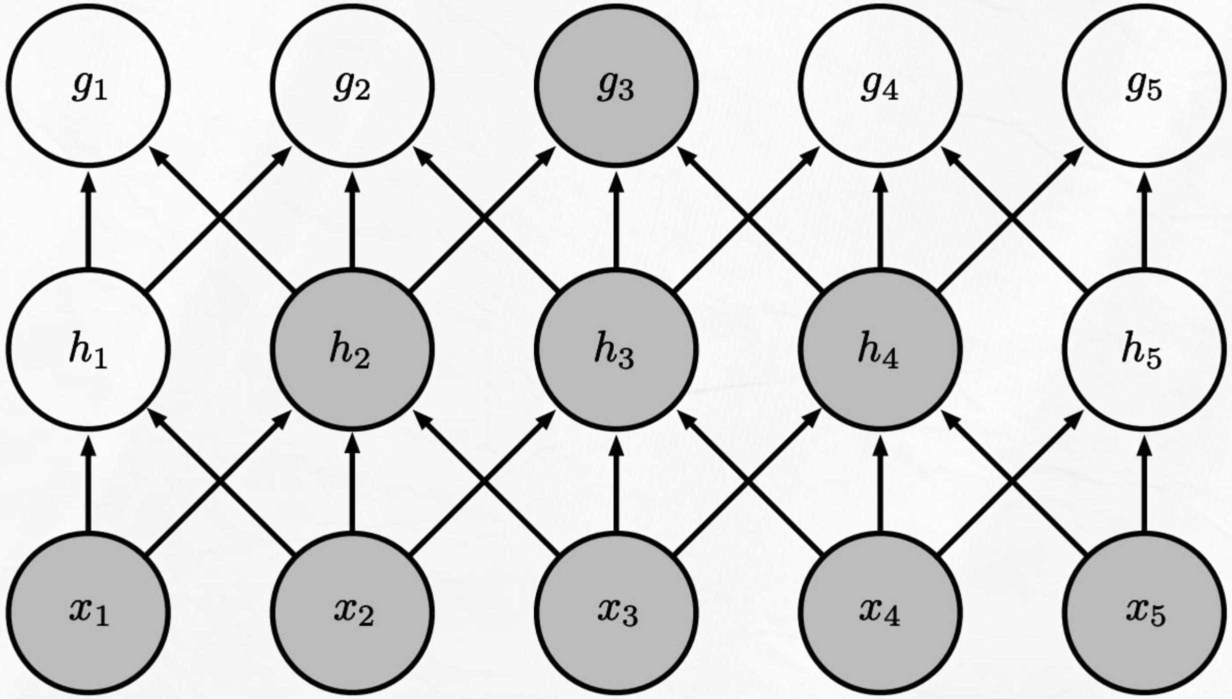
Sparse connections due to small convolution kernel

Dense connections



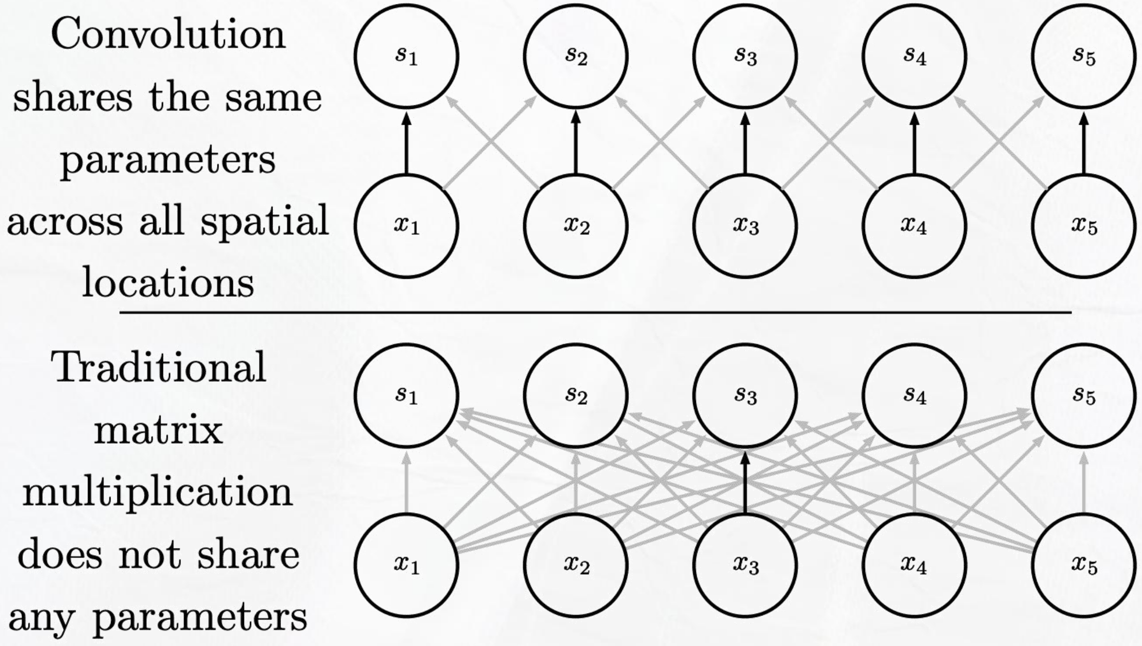
Growing receptive field

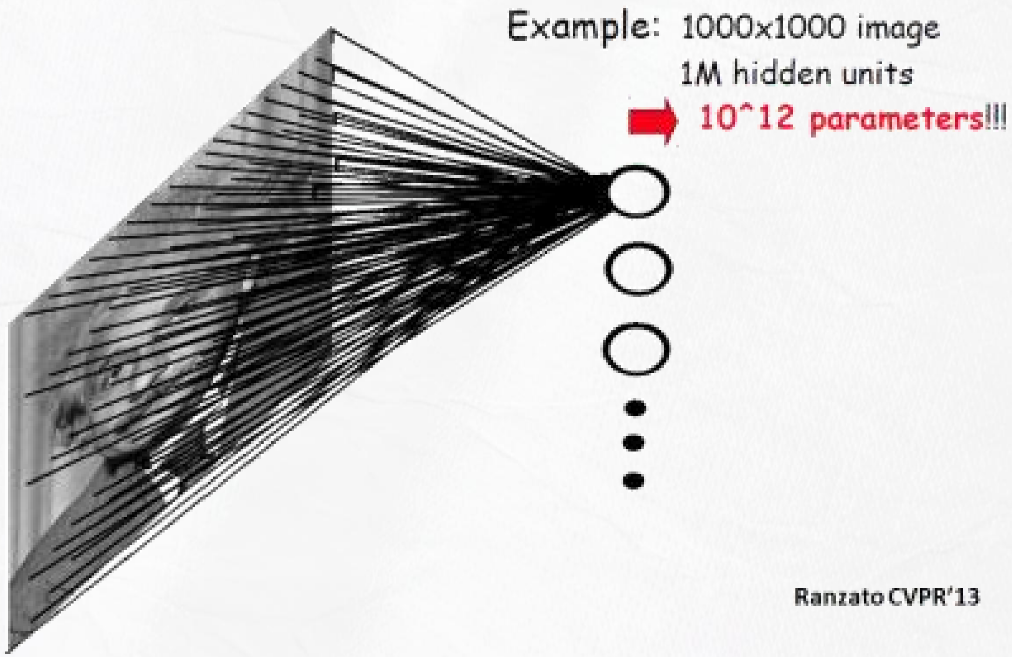
- **Receptive field** will increase along the layer goes deeper.



Shared parameters

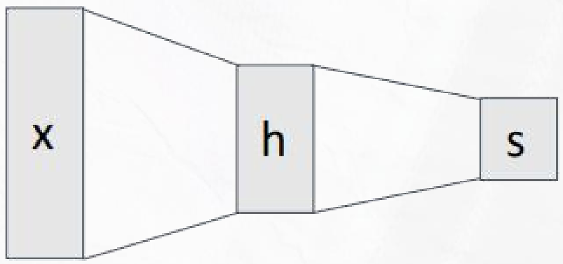
- Parameters at different spatial locations are shared.
- Consequently, as shown in the following example, the number of parameters in the convolution filter is 3, while 5 × 5 = 25 in the fully connected layer.



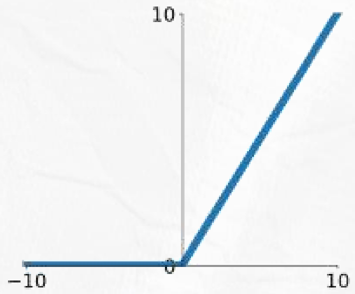


Components of a Convolutional Network

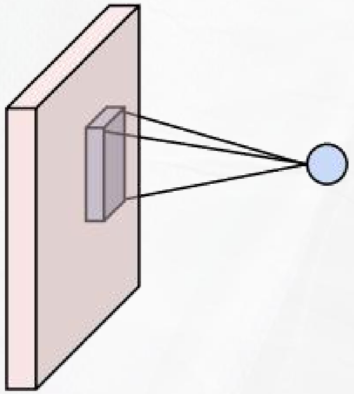
Fully-Connected Layers



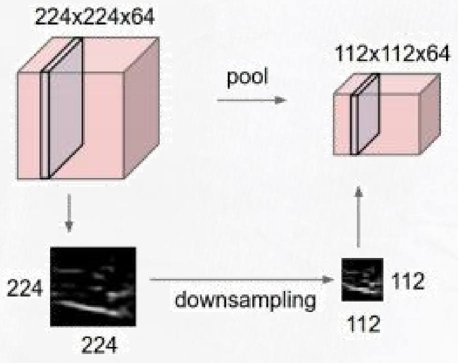
Activation Function



Convolution Layers



Pooling Layers

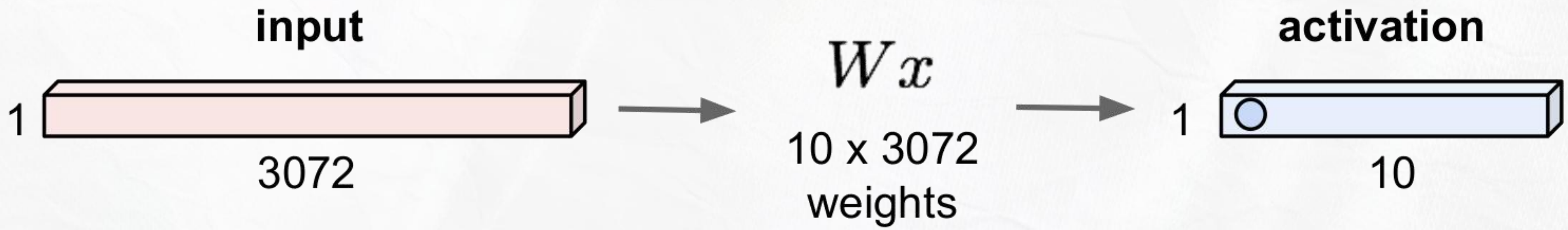


Normalization

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

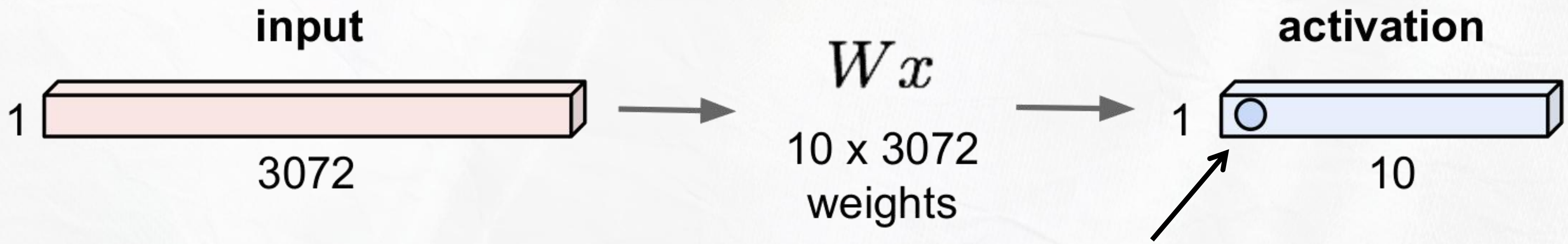
Fully Connected Layer

$32 \times 32 \times 3$ image $\rightarrow$ stretch to 3072×1



Fully Connected Layer

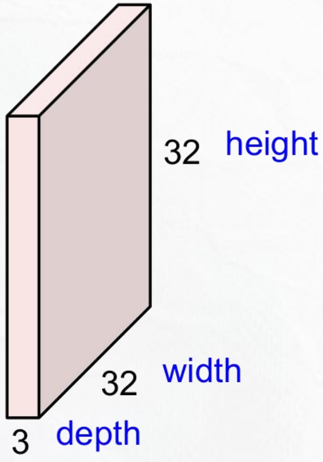
$32 \times 32 \times 3$ image $\rightarrow$ stretch to 3072×1

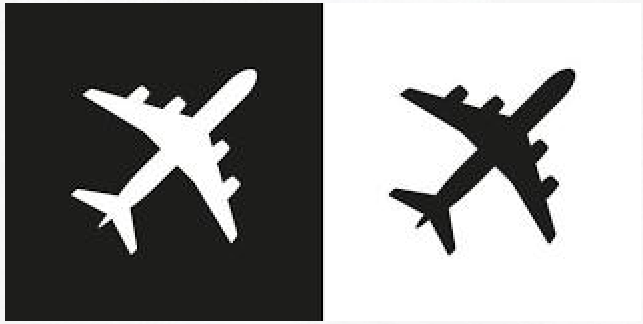


1 number:
the result of taking a dot product between a row of W and the input (a 3072-dimensional dot product)

Convolution Layer

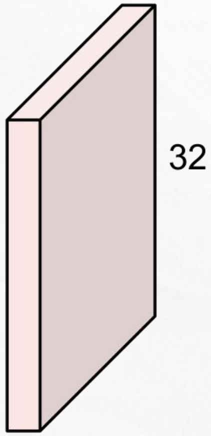
$32 \times 32 \times 3$ image → preserve spatial structure



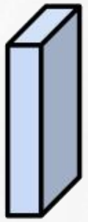


Convolution Layer

$32 \times 32 \times 3$ image



$5 \times 5 \times 3$ filter

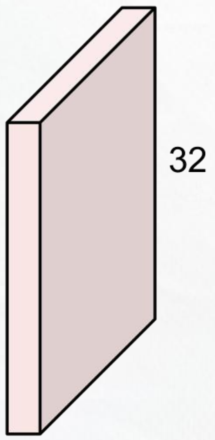


Convolve the filter with the image i.e. "slide over the image spatially, computing dot products"

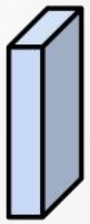
Convolution Layer

Filters always extend the full depth of the input volume

32 × 32 × 3 image

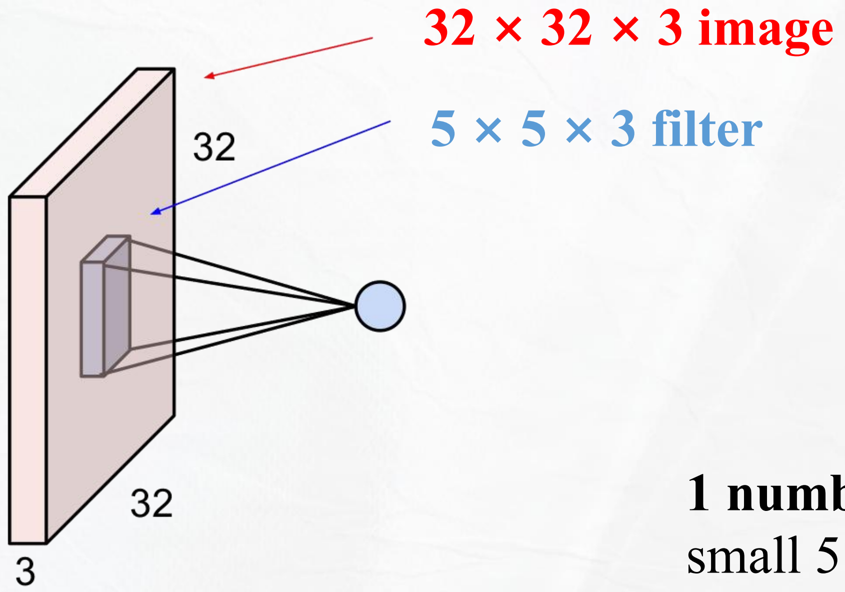


5 × 5 × 3 filter



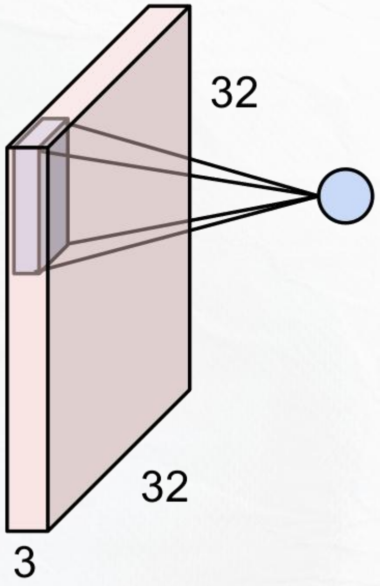
Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

Convolution Layer

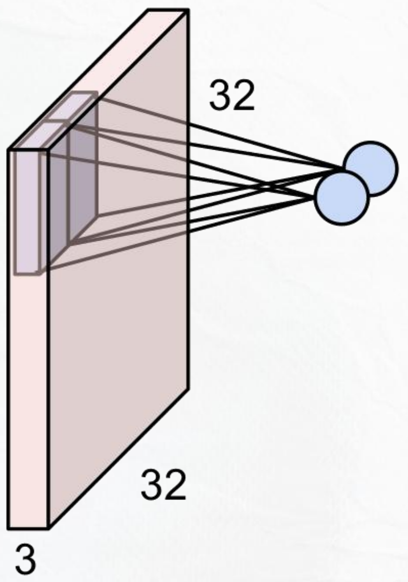


1 number: the result of taking a dot product between the filter and a small $5 \times 5 \times 3$ chunk of the image (i.e. $5 \times 5 \times 3 = 75$-dimensional dot product + bias)

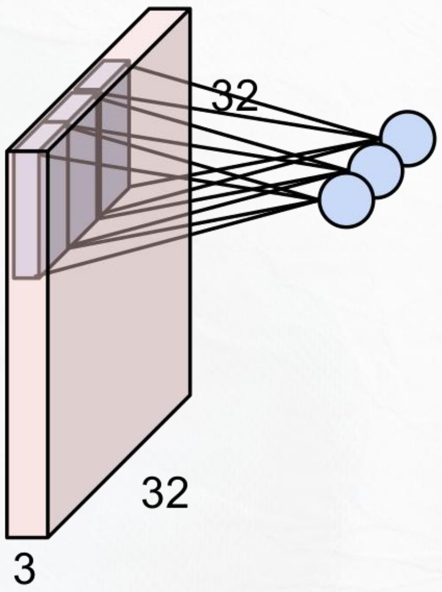
Convolution Layer



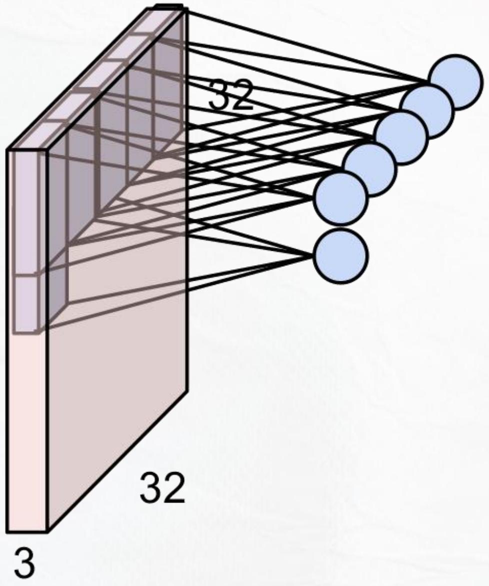
Convolution Layer



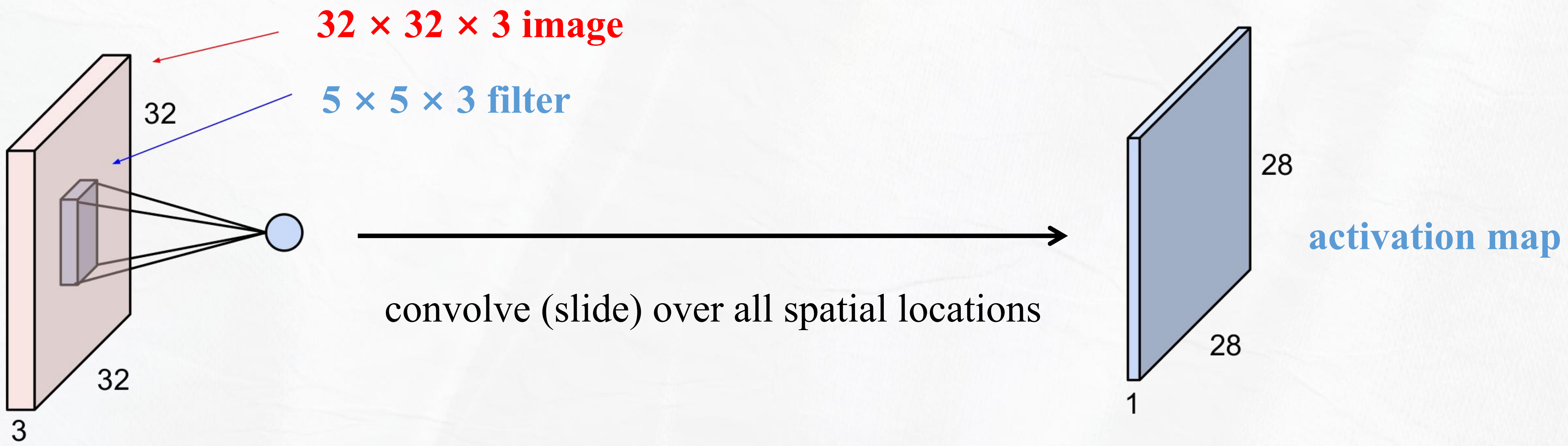
Convolution Layer



Convolution Layer

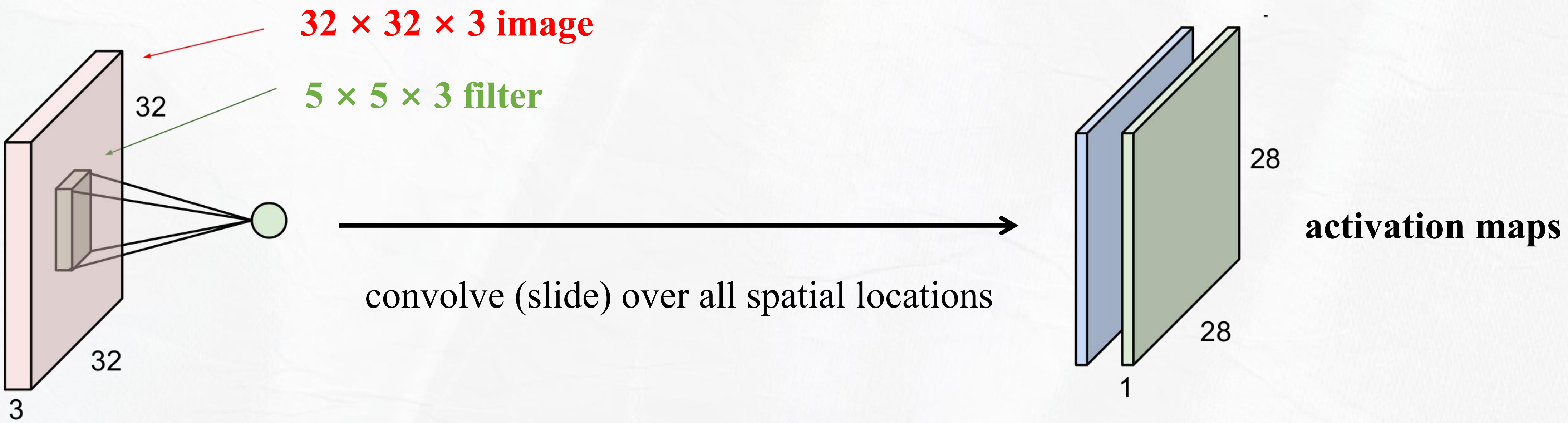


Convolution Layer



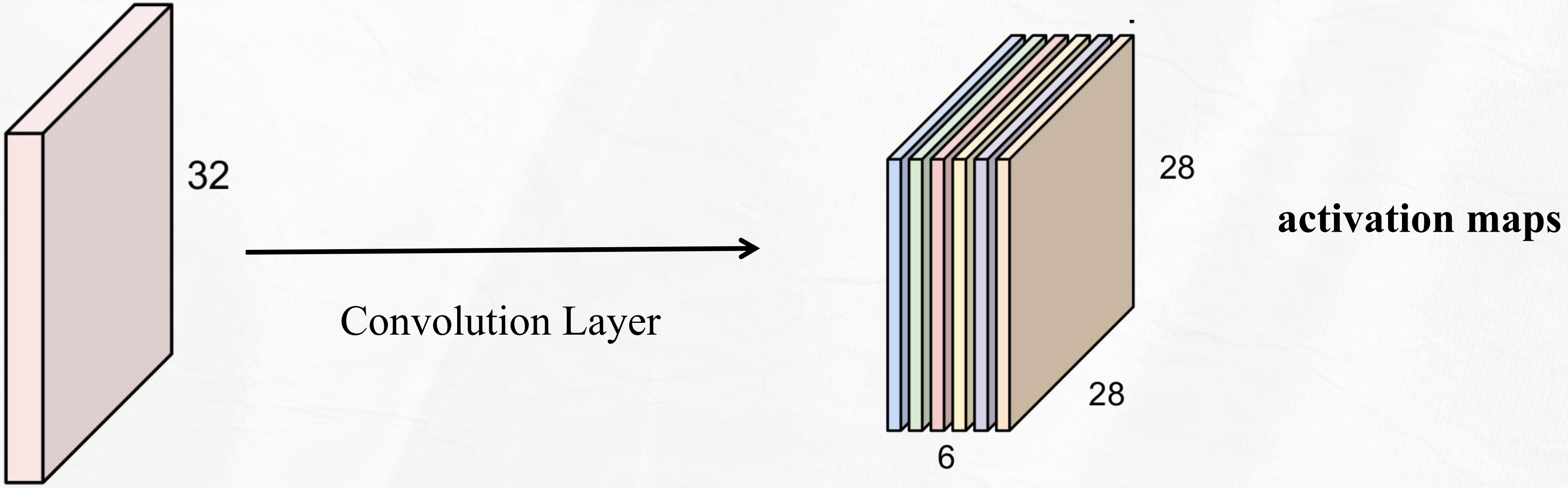
Convolution Layer

consider a second, green filter



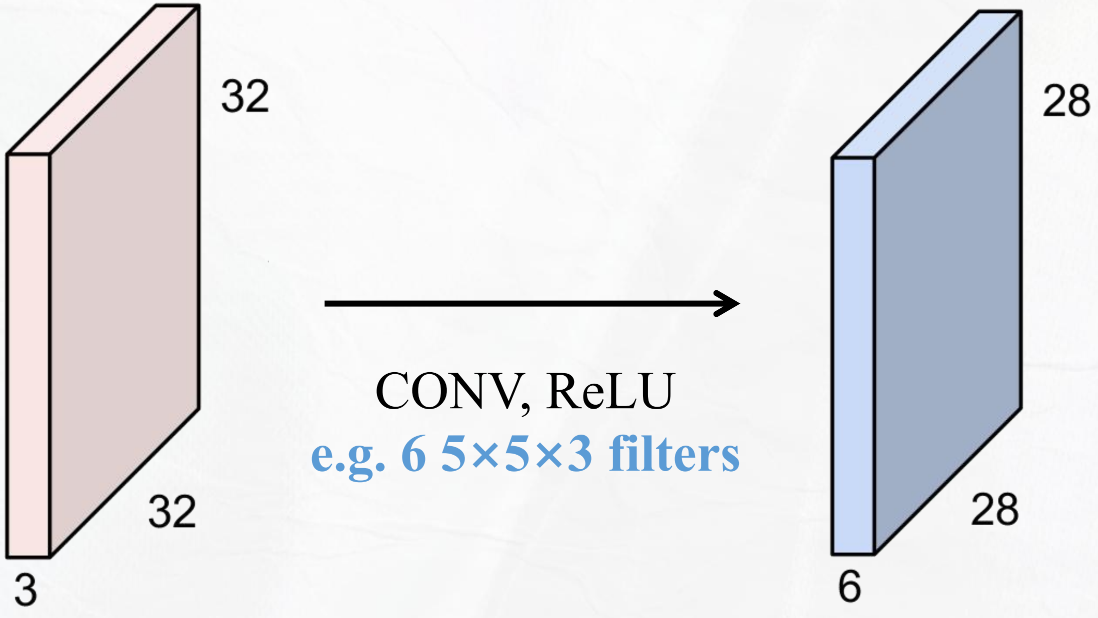
Convolution Layer

For example, if we had 6 5×5 filters, we'll get 6 separate activation maps:

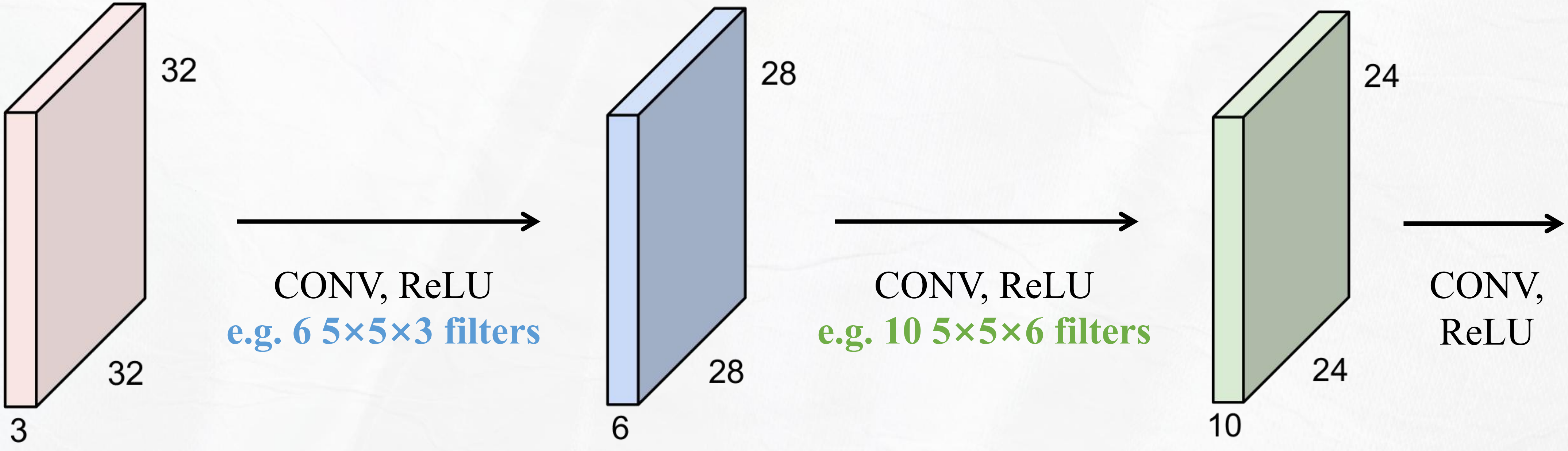


We stack these up to get a "new image" of size $28 \times 28 \times 6$!

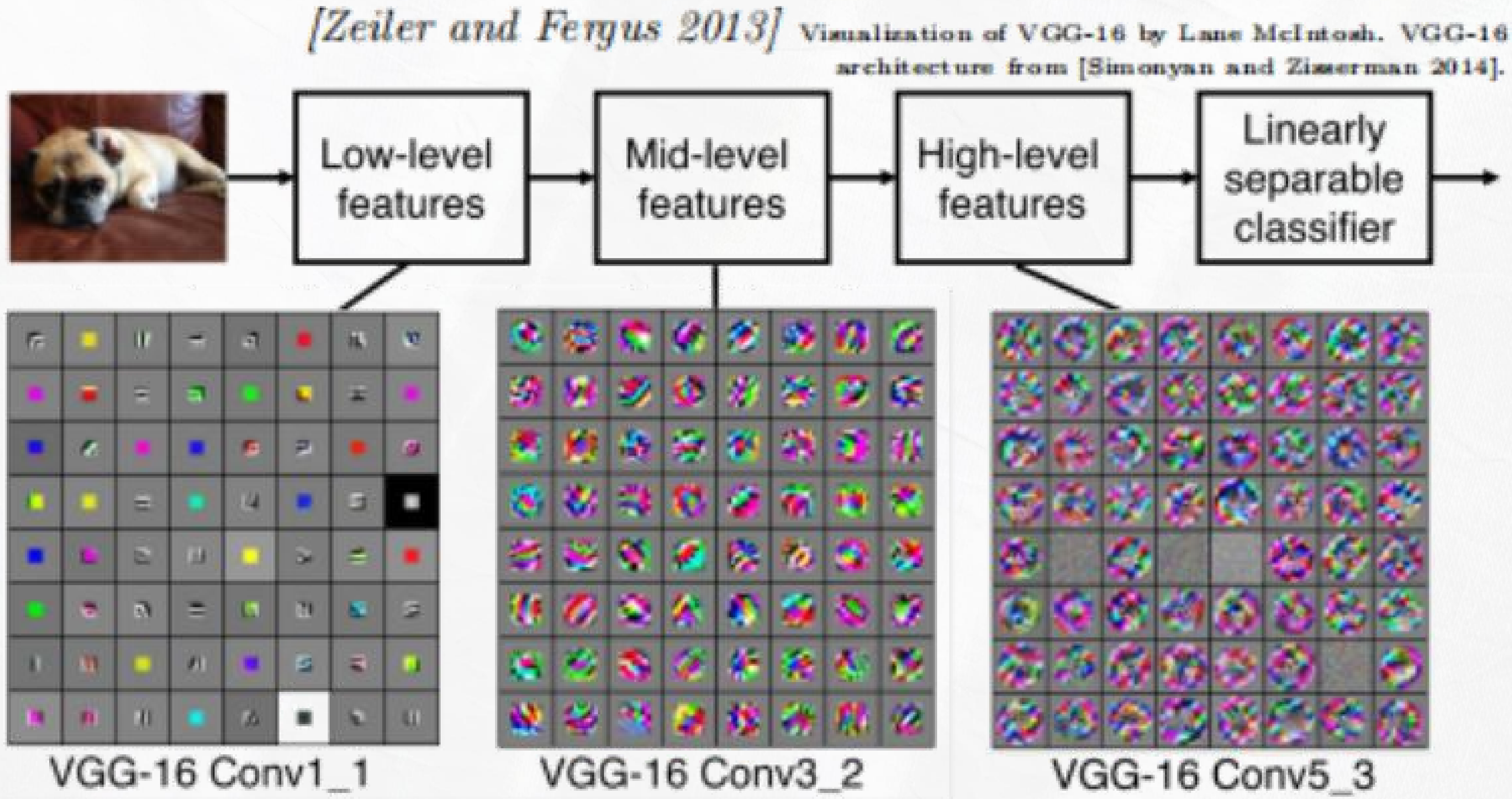
ConvNet is a sequence of Convolution Layers, interspersed with activation functions



ConvNet is a sequence of Convolution Layers, interspersed with activation functions

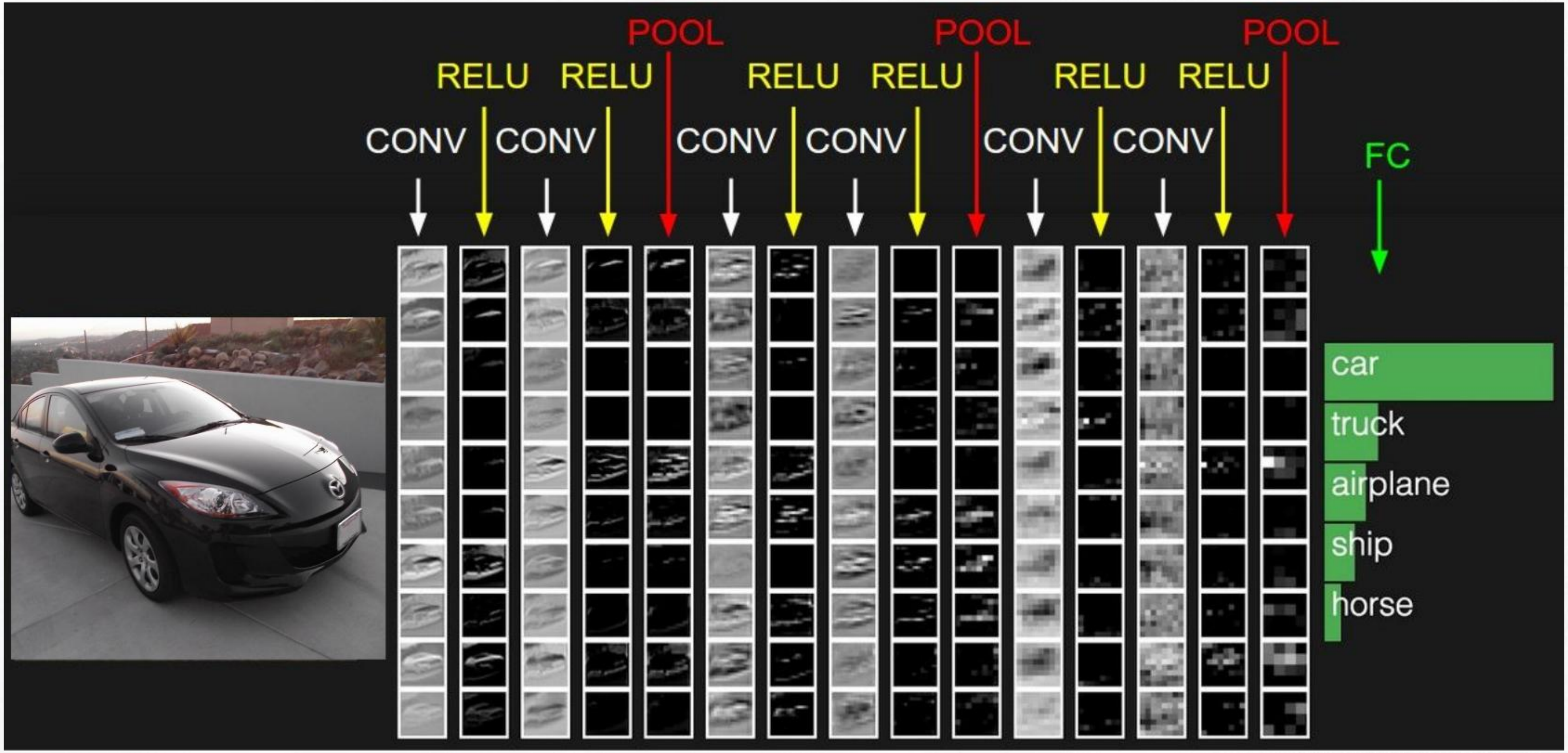


Preview



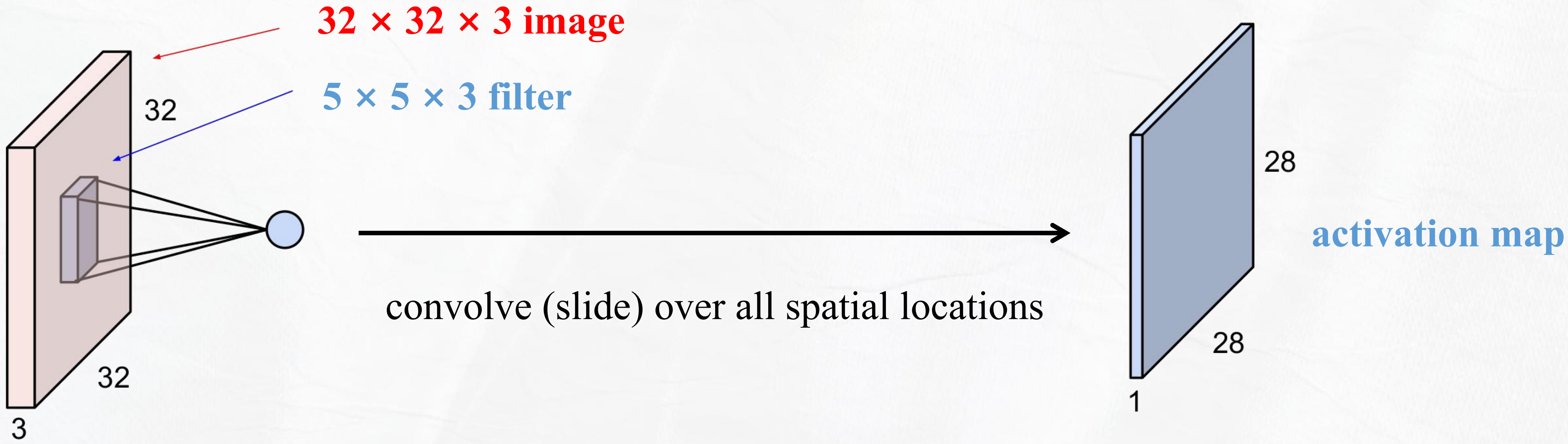
[Zeiler and Fergus 2013]
Visualization of VGG-16 by Lane McIntosh. VGG-16 architecture from [Simonyan and Zisserman 2014].
Low-level features
Mid-level features
High-level features
Linearly separable classifier
VGG-16 Conv1_1
VGG-16 Conv3_2
VGG-16 Conv5_3

Preview

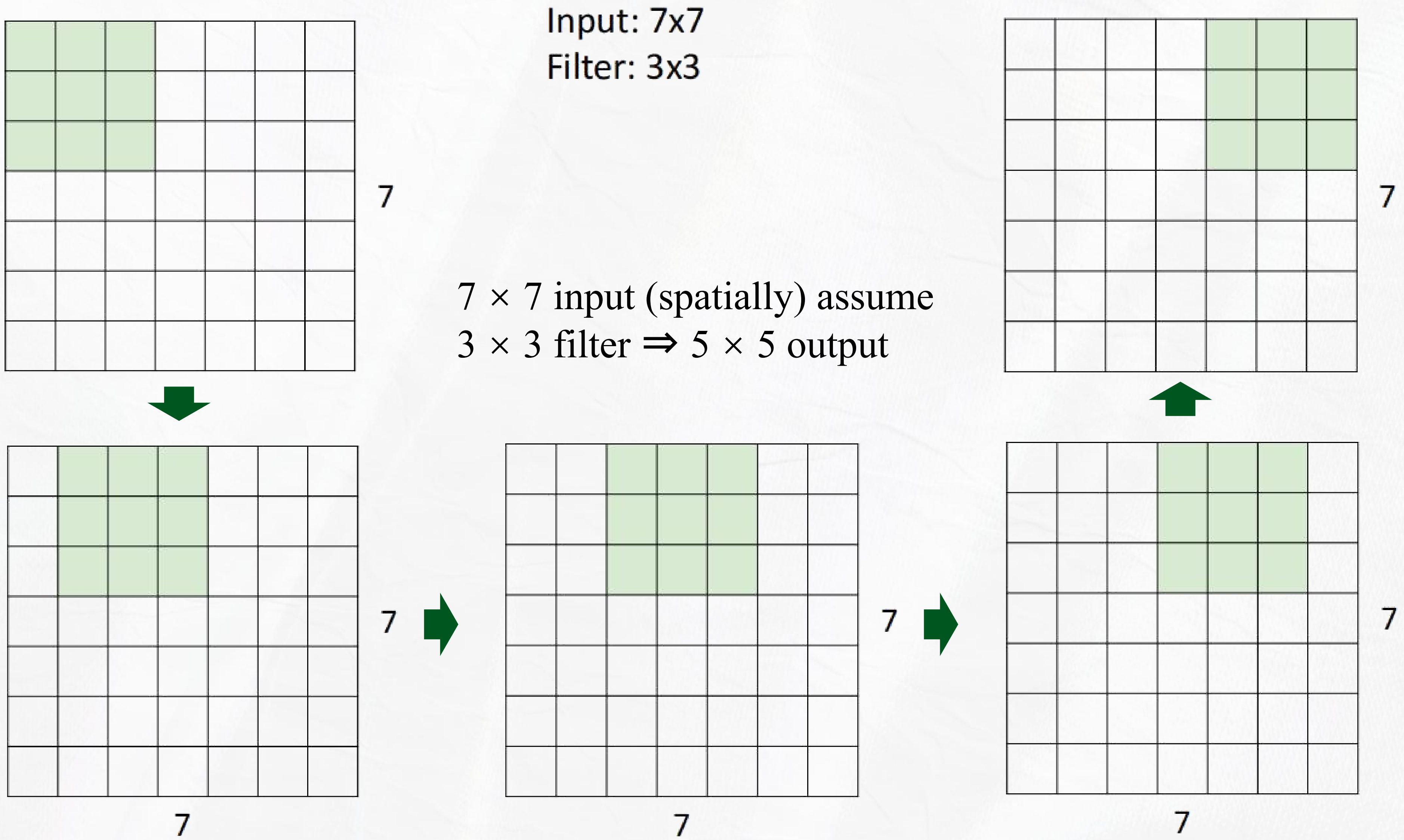


POOL
RELU
RELU
CONV
CONV
RELU
RELU
POOL
CONV
CONV
RELU
RELU
POOL
CONV
CONV
FC
car
truck
airplane
ship
horse

A closer look at spatial dimensions:

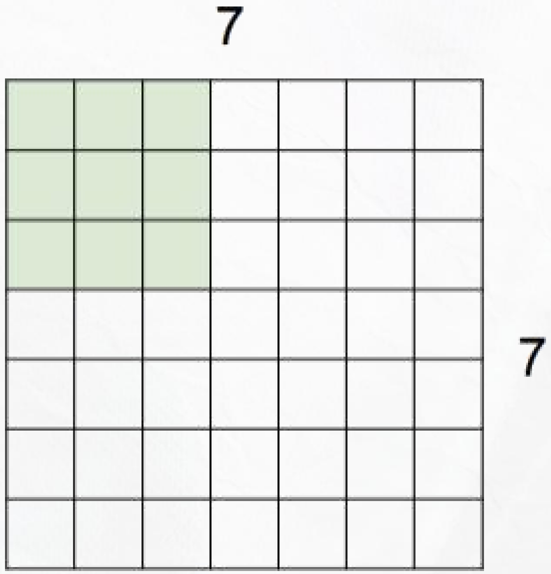


A closer look at spatial dimensions:



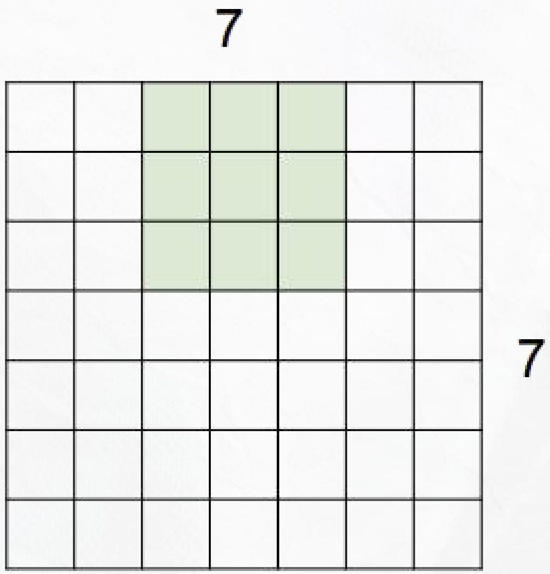
7×7 input (spatially) assume 3×3 filter $\Rightarrow$ 5×5 output

A closer look at spatial dimensions:



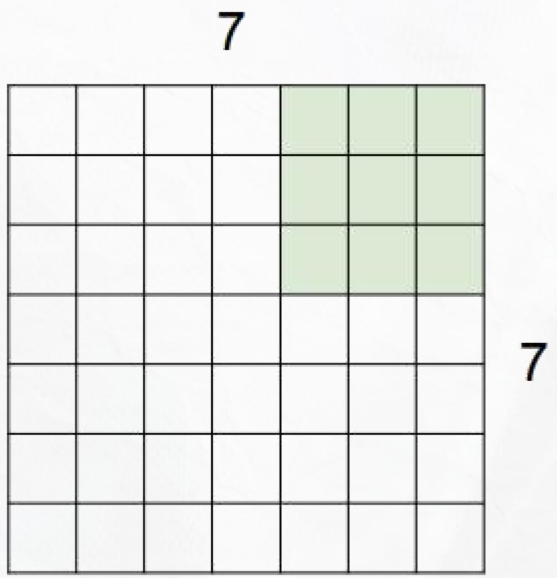
7×7 input (spatially) assume 3×3 filter applied with **stride 2**

A closer look at spatial dimensions:



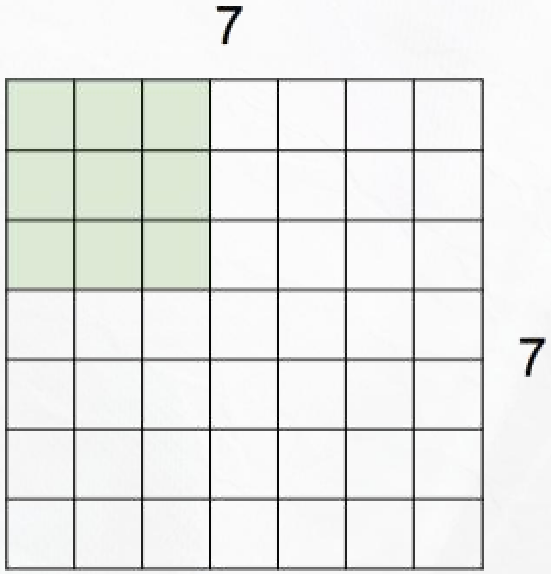
7×7 input (spatially) assume 3×3 filter applied with **stride 2**

A closer look at spatial dimensions:



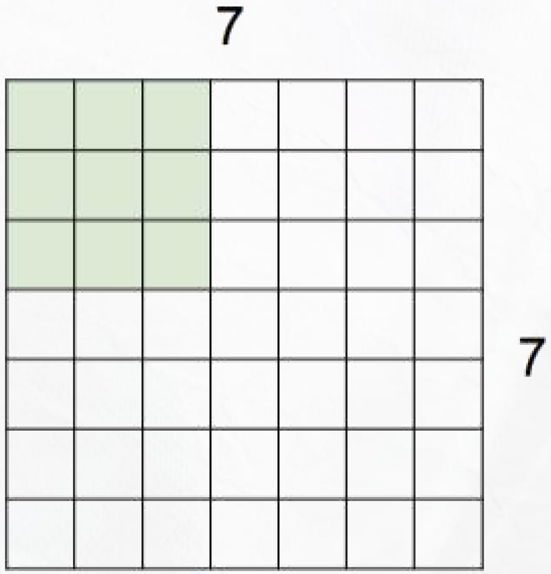
7 × 7 input (spatially) assume 3 × 3 filter applied with **stride 2 => 3 × 3 output!**

A closer look at spatial dimensions:



7×7 input (spatially) assume 3×3 filter applied **with stride 3** ?

A closer look at spatial dimensions:

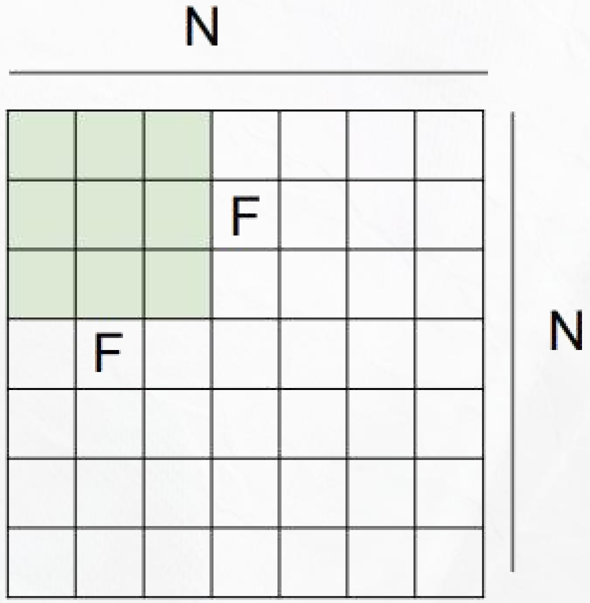


7×7 input (spatially) assume 3×3 filter applied **with stride 3** ?

doesn't fit!

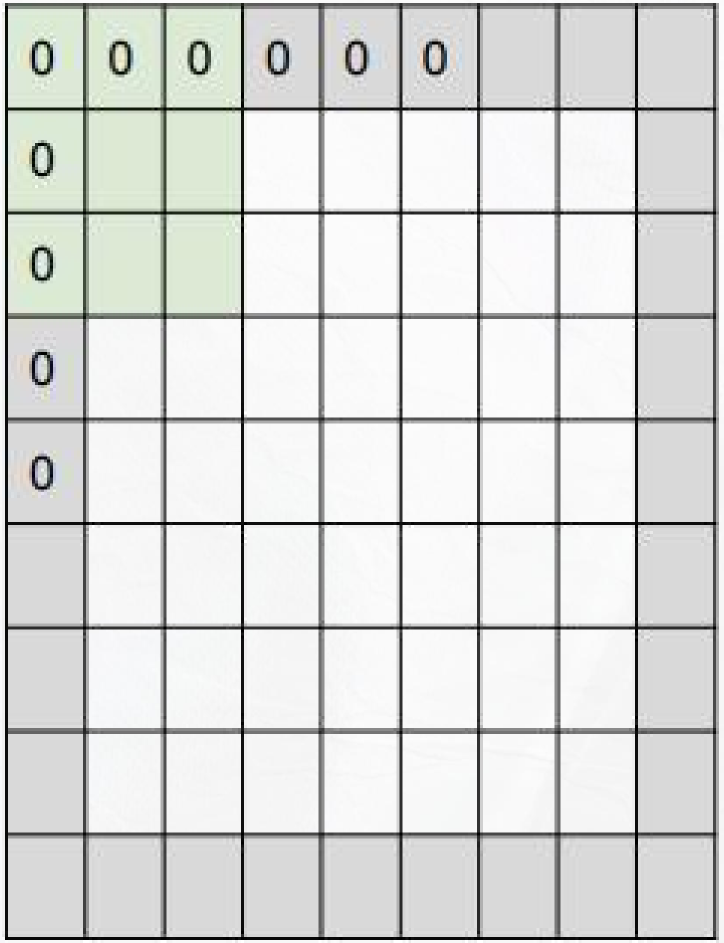
cannot apply 3×3 filter on 7×7 input with stride 3.

A closer look at spatial dimensions:



Output size: (N - F) / stride +1 e.g. N = 7, F = 3 :
stride 1 ⇒ $(7 - 3)/1 + 1 = 5$
stride 2 ⇒ $(7 - 3)/2 + 1 = 3$
stride 3 => $(7 - 3)/3 + 1 = 2.33$

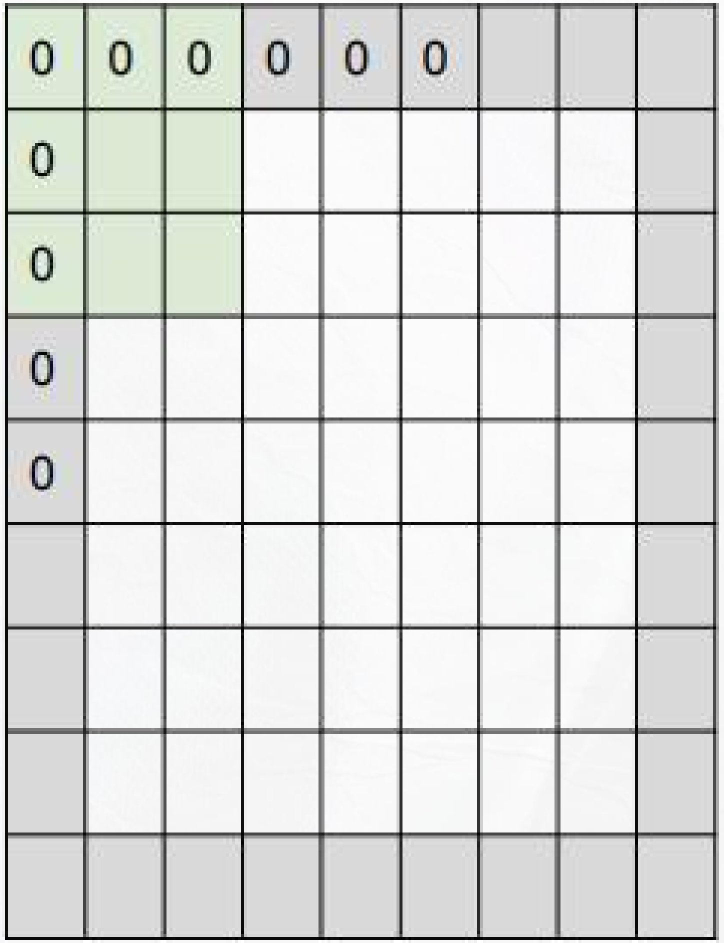
In practice: Common to zero pad the border



e.g.
input 7×7, 3×3 filter,
applied with stride 1, pad with 1 pixel border
=>
what is the output?

(recall:)
$(N - F)$ / stride + 1

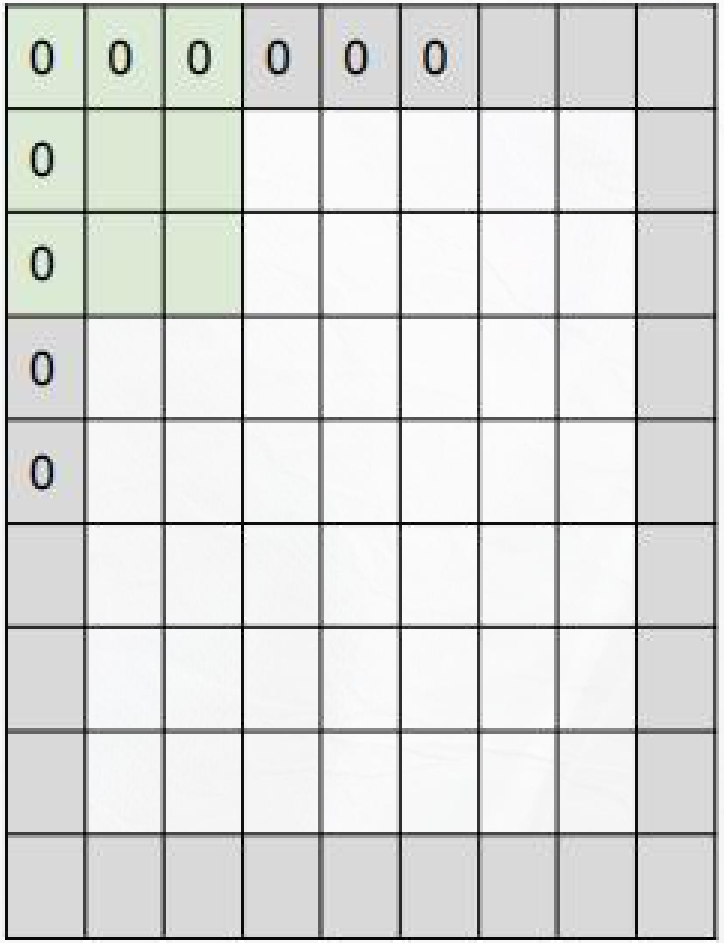
In practice: Common to zero pad the border



e.g.
input 7 × 7, 3 × 3 filter,
applied with stride 1, pad with 1 pixel border
=>
what is the output?
7 × 7 output!

(recall:)
(N + 2P− F) / stride + 1

In practice: Common to zero pad the border

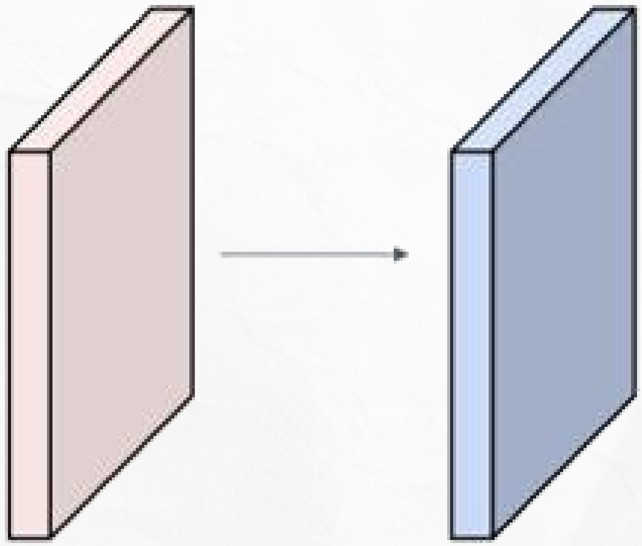


e.g.
input 7×7, 3×3 filter,
applied with stride 1, pad with 1 pixel border
=>
what is the output?
7×7 output!

In general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with $(F-1)/2$. (will preserve size spatially)

e.g. $F = 3 \Rightarrow$ zero pad with 1
$F = 5 \Rightarrow$ zero pad with 2
$F = 7 \Rightarrow$ zero pad with 3

Practice: about the model size

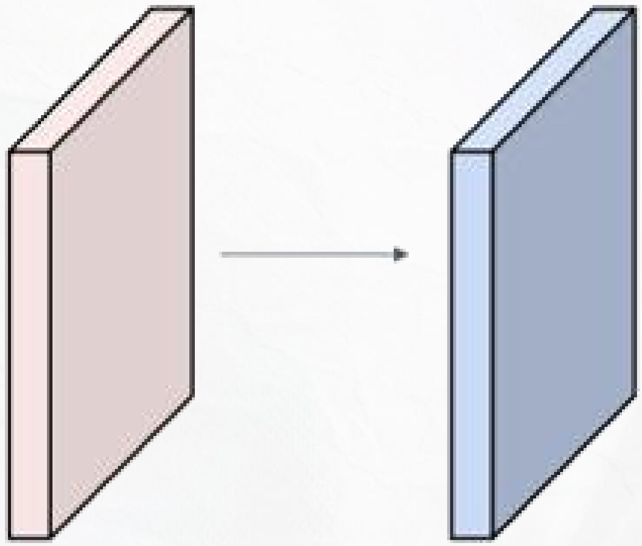


Input volume : 3×32 ×32

10 5 × 5 filters with stride 1 , pad 2

Output volume size: ?

Practice: about the model size



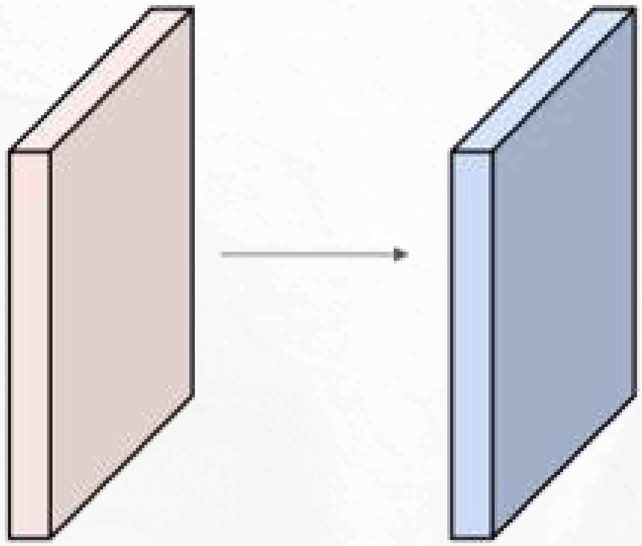
Input volume : 3×32 ×32

10 5 × 5 filters with stride 1 , pad 2

Output volume size:

$(32 + 2\times2 - 5)\ /\ 1 + 1 = 32$ spatially, so 10×32×32

Practice: about the model size



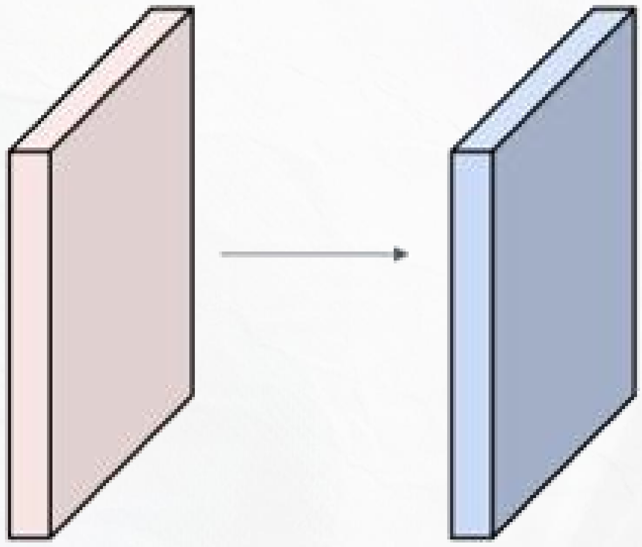
Input volume : $3\times32\times32$

10 5×5 filters with stride 1 , pad 2

Output volume size: $10\times32\times32$

Number of learnable parameters: ?

Practice: about the model size



Input volume : $3 \times 32 \times 32$

10 5×5 filters with stride 1 , pad 2

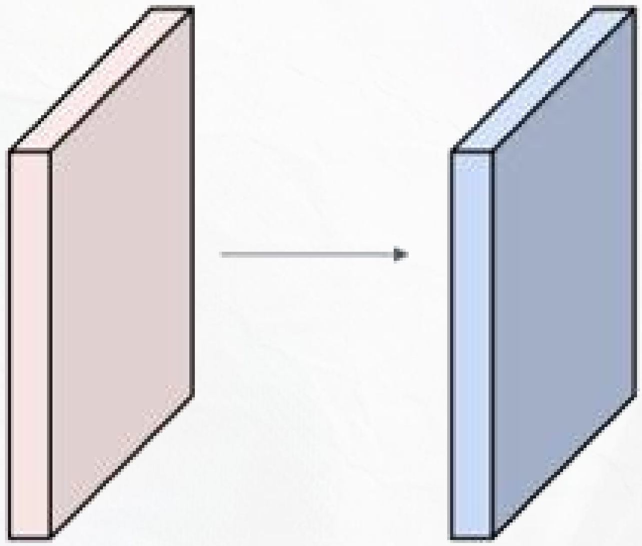
Output volume size: $10 \times 32 \times 32$

Number of learnable parameters: **760**

Parameters per filter: $3 \times 5 \times 5 + 1$ (for bias) $= 76$

10 filter, so total is $10 \times 76 =$ **760**

Practice: about the model size



Input volume : $3 \times 32 \times 32$

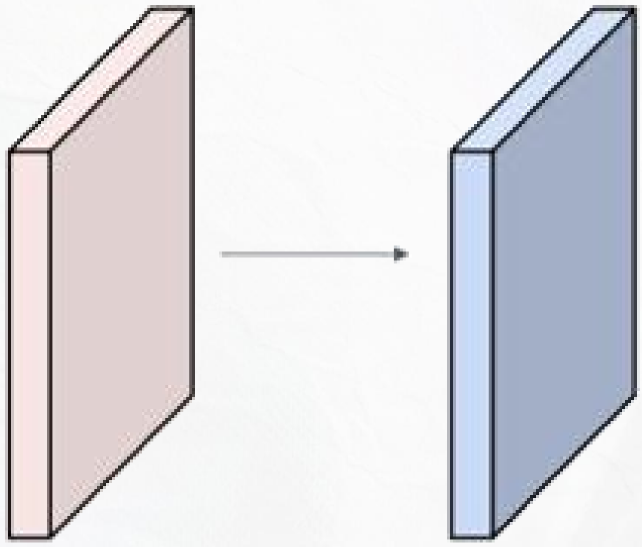
10 5×5 filters with stride 1 , pad 2

Output volume size: $10 \times 32 \times 32$

Number of learnable parameters: 760

Number of multiply-add operations: ?

Practice: about the model size



Input volume : $3 \times 32 \times 32$

10 5×5 filters with stride 1 , pad 2

Output volume size: $10 \times 32 \times 32$

Number of learnable parameters: 760

Number of multiply-add operations: **768,000**

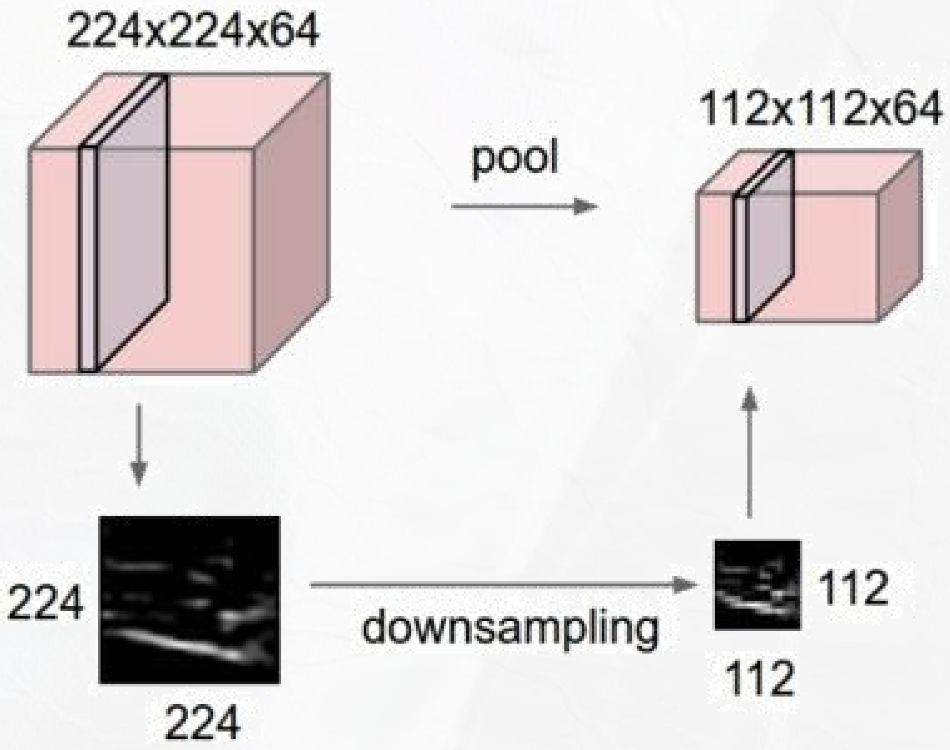
$10 \times 32 \times 32$ = 10,240 outputs;

each output is the inner product of two $3 \times 5 \times 5$ tensors

(75 elems); total = 75×10240 = **768K**

Pooling Layer

- Another way to downsample



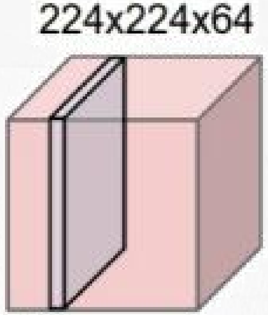
Hyperparameter:

Kernel Size

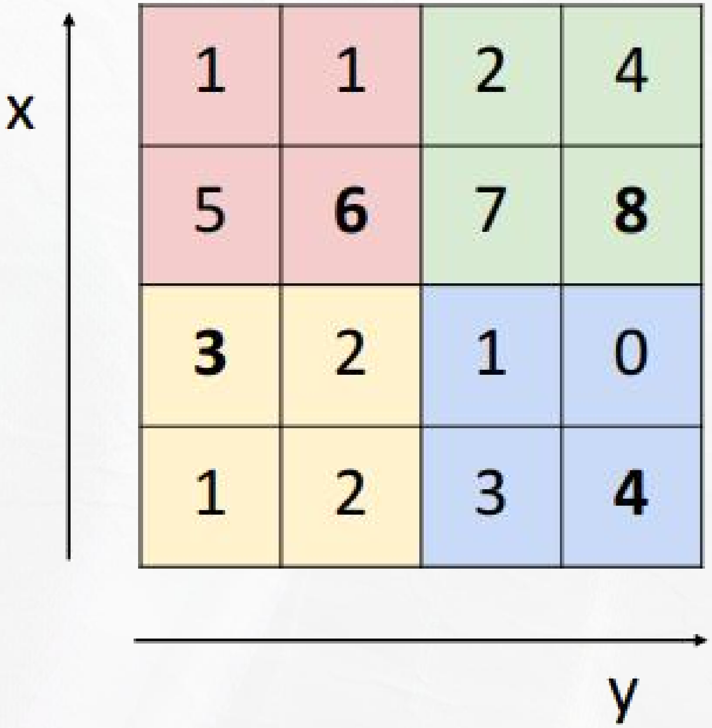
Stride

Pooling function

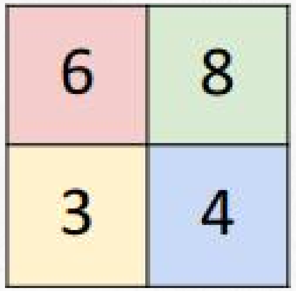
Max Pooling



Single depth slice



Max pooling with 2x2 kernel size and stride 2



Introduces **invariance** to small spatial shifts

No learnable parameters!

Pooling Layer Summary

- **Input:** C×H×W
- **Hyperparameters:**
  - kernel size: K
  - Stride: S
  - Pooling function (max, avg)
- **Output:** C×H'×W' where
  - H' = (H-K) / S + 1
  - W' = (W-K) / S + 1
- **Learnable parameters:** None!

Common settings:

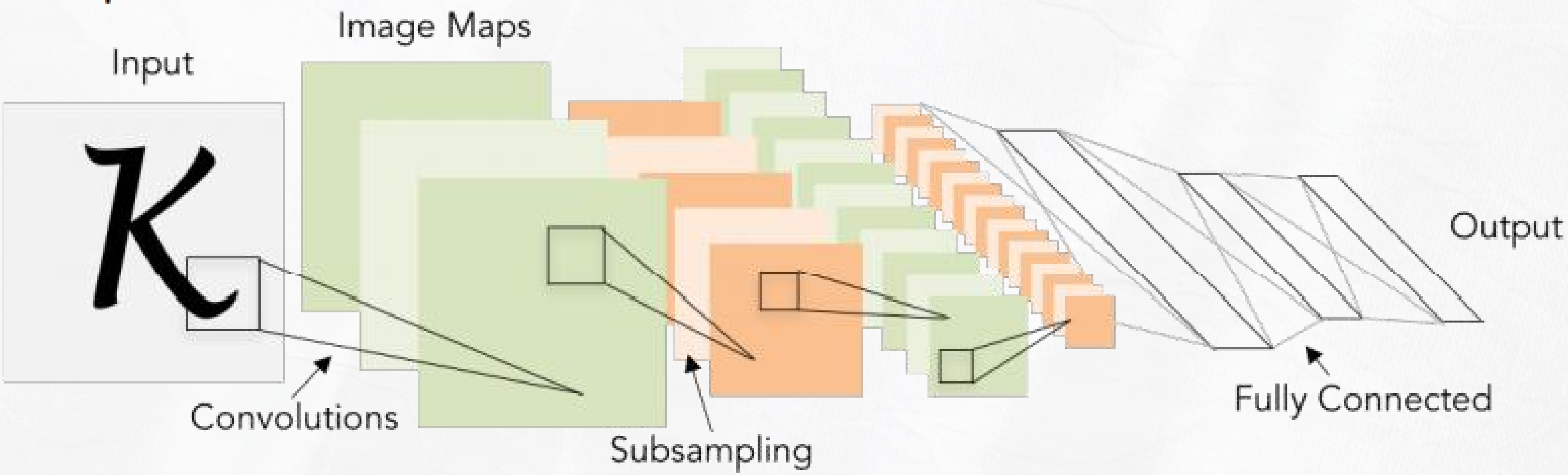
max, K=2, S=2

max, K=3, S=2 (AlexNet)

Convolutional Networks Example:

Classic architecture: [Conv, ReLU, Pool] x N, flatten, [FC, ReLU] x N, FC

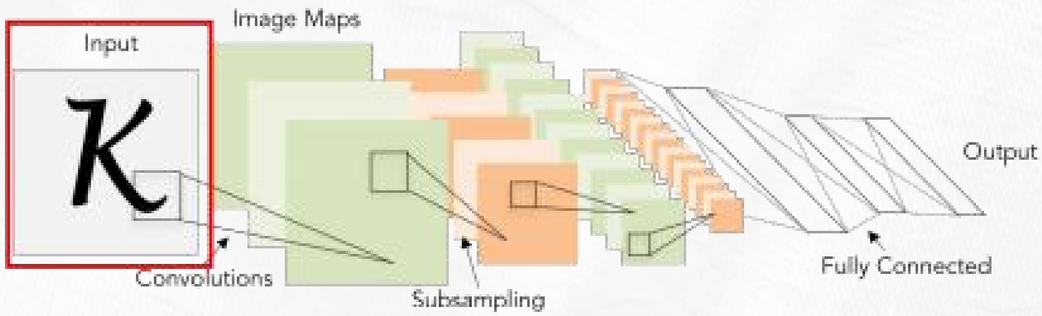
Example: LeNet-5



Convolutional Networks Example:

Example: LeNet-5

Layer	Output Size	Weight Size
Input	1 x 28 x 28	



Convolutional Networks Example:

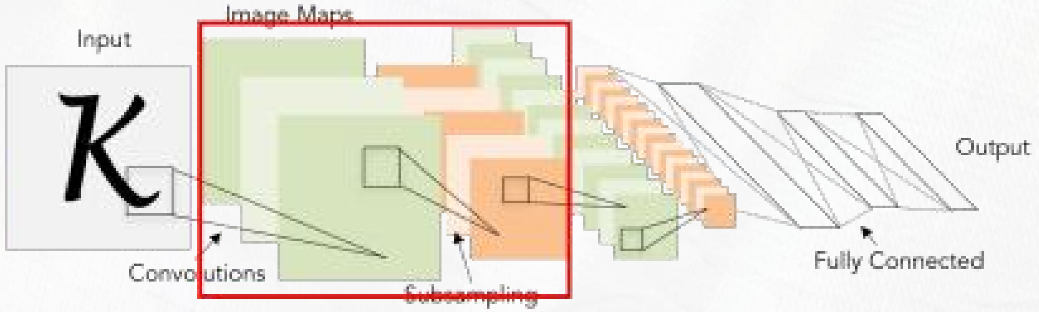
Example: LeNet-5

Layer	Output Size	Weight Size
Input	1 x 28 x 28	
Conv (C_{out}=20, K=5, P=2, S=1)	20 x 28 x 28	20 x 1 x 5 x 5
ReLU	20 x 28 x 28	

Convolutional Networks Example:

Example: LeNet-5

Layer	Output Size	Weight Size
Input	1 x 28 x 28	
Conv (C_{out}=20, K=5, P=2, S=1)	20 x 28 x 28	20 x 1 x 5 x 5
ReLU	20 x 28 x 28	
MaxPool(K=2, S=2)	20 x 14 x 14	



Convolutional Networks Example:

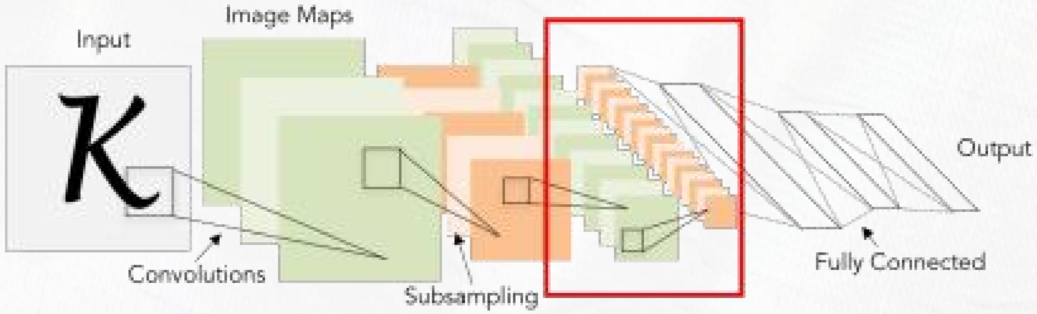
Example: LeNet-5

Layer	Output Size	Weight Size
Input	1 x 28 x 28	
Conv (C_{out}=20, K=5, P=2, S=1)	20 x 28 x 28	20 x 1 x 5 x 5
ReLU	20 x 28 x 28	
MaxPool(K=2, S=2)	20 x 14 x 14	
Conv (C_{out}=50, K=5, P=2, S=1)	50 x 14 x 14	50 x 20 x 5 x 5
ReLU	50 x 14 x 14	

Convolutional Networks Example:

Example: LeNet-5

Layer	Output Size	Weight Size
Input	1 x 28 x 28	
Conv (C_{out}=20, K=5, P=2, S=1)	20 x 28 x 28	20 x 1 x 5 x 5
ReLU	20 x 28 x 28	
MaxPool(K=2, S=2)	20 x 14 x 14	
Conv (C_{out}=50, K=5, P=2, S=1)	50 x 14 x 14	50 x 20 x 5 x 5
ReLU	50 x 14 x 14	
MaxPool(K=2, S=2)	50 x 7 x 7	

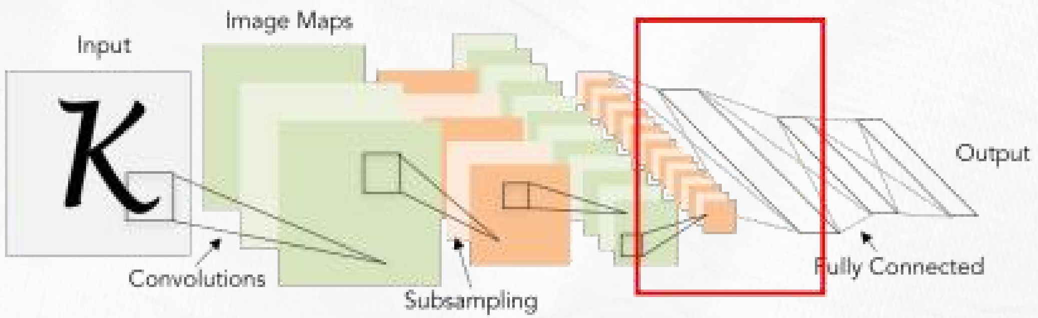


Convolutional Networks Example:

中山大學 SUN YAT-SEN UNIVERSITY

Example: LeNet-5

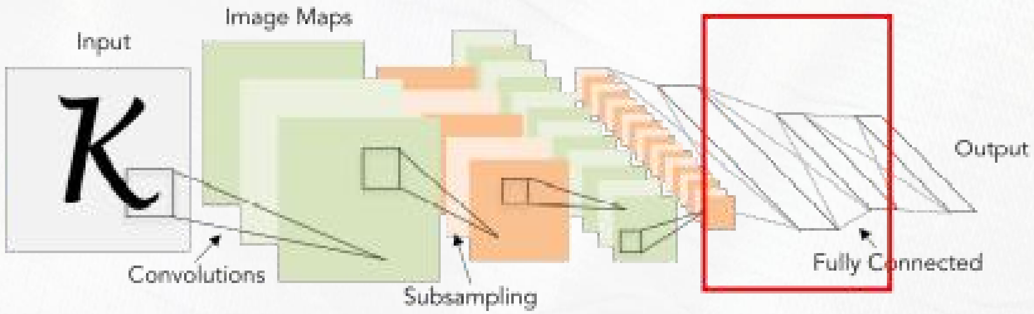
Layer	Output Size	Weight Size
Input	1 x 28 x 28	
Conv (C_{out}=20, K=5, P=2, S=1)	20 x 28 x 28	20 x 1 x 5 x 5
ReLU	20 x 28 x 28	
MaxPool(K=2, S=2)	20 x 14 x 14	
Conv (C_{out}=50, K=5, P=2, S=1)	50 x 14 x 14	50 x 20 x 5 x 5
ReLU	50 x 14 x 14	
MaxPool(K=2, S=2)	50 x 7 x 7	
Flatten	2450	



Convolutional Networks Example:

Example: LeNet-5

Layer	Output Size	Weight Size
Input	1 x 28 x 28	
Conv (C_{out}=20, K=5, P=2, S=1)	20 x 28 x 28	20 x 1 x 5 x 5
ReLU	20 x 28 x 28	
MaxPool(K=2, S=2)	20 x 14 x 14	
Conv (C_{out}=50, K=5, P=2, S=1)	50 x 14 x 14	50 x 20 x 5 x 5
ReLU	50 x 14 x 14	
MaxPool(K=2, S=2)	50 x 7 x 7	
Flatten	2450	
Linear (2450 -> 500)	500	2450 x 500
ReLU	500	

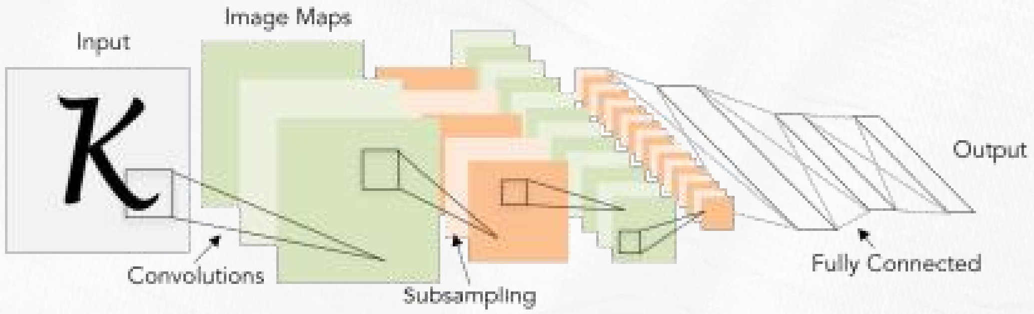


Convolutional Networks Example:

中山大學 SUN YAT-SEN UNIVERSITY

Example: LeNet-5

Layer	Output Size	Weight Size
Input	1 x 28 x 28	
Conv (C_{out}=20, K=5, P=2, S=1)	20 x 28 x 28	20 x 1 x 5 x 5
ReLU	20 x 28 x 28	
MaxPool(K=2, S=2)	20 x 14 x 14	
Conv (C_{out}=50, K=5, P=2, S=1)	50 x 14 x 14	50 x 20 x 5 x 5
ReLU	50 x 14 x 14	
MaxPool(K=2, S=2)	50 x 7 x 7	
Flatten	2450	
Linear (2450 -> 500)	500	2450 x 500
ReLU	500	
Linear (500 -> 10)	10	500 x 10



As we go through the network:

Spatial size **decreases**
(using pooling or strided conv)

Number of channels **increases**
(total “volume” is preserved!)

中山大學
SUN YAT-SEN UNIVERSITY

Thanks for Listening

SUN YAT-SEN UNIVERSITY 2025

Prof. Dr. Ruimao Zhang

Room N-205,　Engineering Building 3

zhangrm27@mail.sysu.edu.cn