Deep Neural Networks for Classifications

Department of Mathematics and Statistics
SUNY Albany

Outline

- logistic regression, softmax ouput, cross entropy
- neural network, dense layer, non-linear activation, ReLU
- convolutional layer, pooling layer
- accelerated SGD

Multi-class classification

- i-th sample in training dataset $\mathcal{D}$:
  - input: data $\mathbf{x}^{(i)} \in \mathbb{R}^d$
  - expected output: class label $y^{(i)} \in \{1, \ldots, k\}$
- learn a vector-valued function with tunable model parameters θ

$$f(\cdot; \theta) : \mathbb{R}^d \to (0, 1)^k$$

  - $f(\mathbf{x}^{(i)}; \theta)_c$ is the **predicted probability** that input data $\mathbf{x}^{(i)}$ belongs to Class c, for each $c = 1, \ldots, k$
  - tune θ using certain optimization algorithm in the hope that, for all samples $1 \le i \le n$,

$$\operatorname*{argmax}_{1 \le c \le k} f(\mathbf{x}^{(i)}; \theta)_c = y^{(i)}$$

Multi-class logistic regression

- f is composition of linear and softmax functions
- input data $\mathbf{x} \in \mathbb{R}^d$, fed into a linear function z
- **pre-activation** vector $z(\mathbf{x}) = \mathbf{W}\mathbf{x} + b \in \mathbb{R}^k$; $\mathbf{W} \in \mathbb{R}^{k \times d}$: **weight matrix**, $\mathbf{b} \in \mathbb{R}^k$: **bias vector**
- model outputs the **predicted probabilities** for the k classes:

$$f(\mathbf{x}; \mathbf{W}, \mathbf{b}) = \text{softmax}(z(\mathbf{x})) = \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b}) = \begin{bmatrix} \frac{\exp(\tilde{\mathbf{w}}_1^\top \mathbf{x} + b_1)}{\sum_{c=1}^k \exp(\tilde{\mathbf{w}}_c^\top \mathbf{x} + b_c)} \\ \vdots \\ \frac{\exp(\tilde{\mathbf{w}}_k^\top \mathbf{x} + b_k)}{\sum_{c=1}^k \exp(\tilde{\mathbf{w}}_c^\top \mathbf{x} + b_c)} \end{bmatrix} \in (0,1)^k$$

where $\tilde{\mathbf{w}}_c^\top$ is the c-th row of $\mathbf{W}$, $c = 1, \ldots, k$

- find parameters $\theta = (\mathbf{W}, \mathbf{b})$ that minimize the training loss function
  - the **cross entropy loss** for a given training sample $\mathbf{x}^{(i)} \in \mathcal{D}$, with $f(\mathbf{x}^{(i)}; \theta) \in (0,1)^k$, $1 \leq y^{(i)} \leq k$:

    $$\ell(f(\mathbf{x}^{(i)}; \theta), y^{(i)}) = -\log f(\mathbf{x}^{(i)}; \theta)_{y^{(i)}} \geq 0 \quad \text{(note that } f(\mathbf{x}^{(i)}; \theta)_{y^{(i)}} \in (0,1))$$

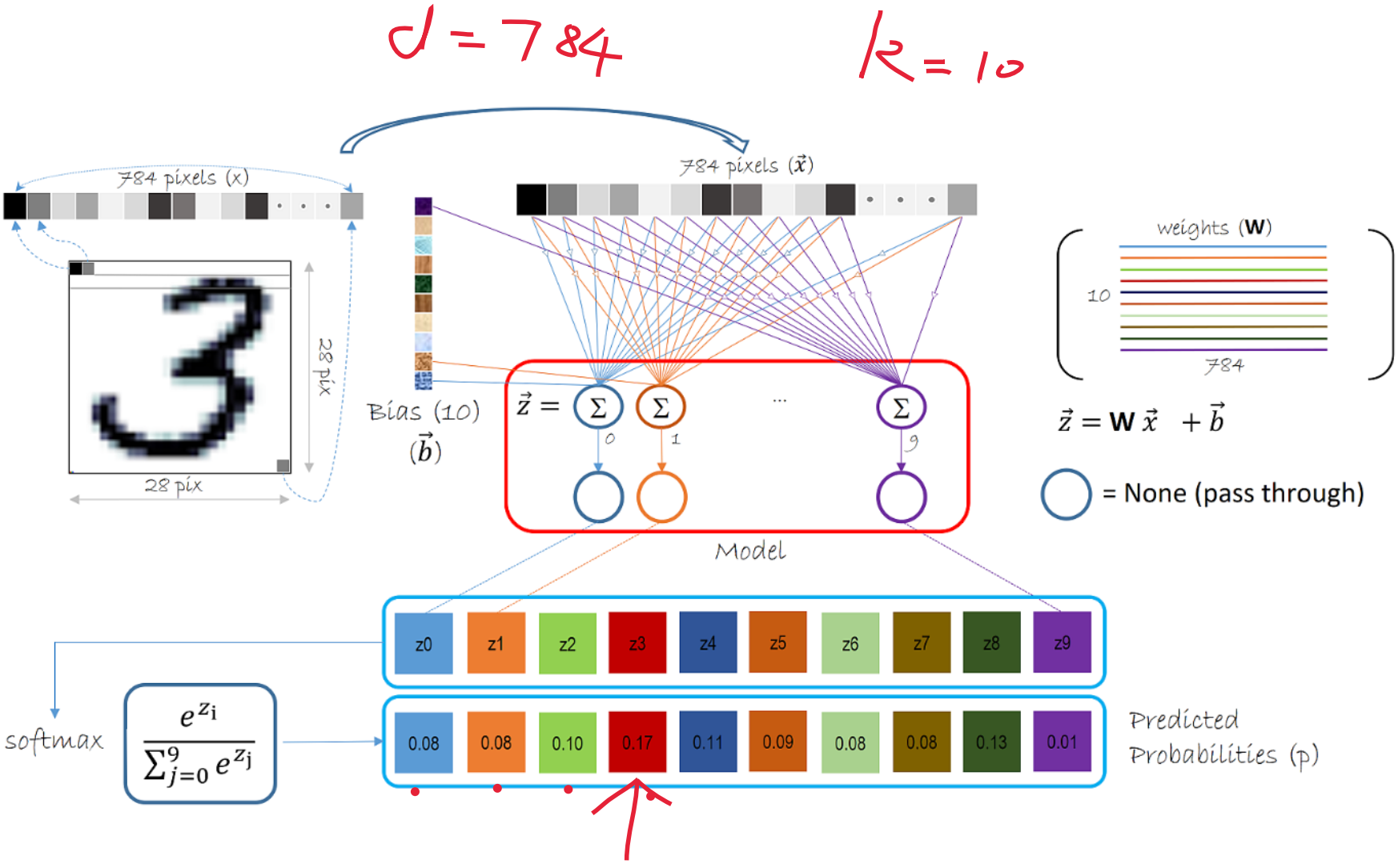
  - for each sample, want to make $f(\mathbf{x}^{(i)}; \theta)_{y^{(i)}}$ (i.e., the predicted probability for Class $y^{(i)}$) as large (close to 1) as possible
  - the training loss to be minimized is the cross entropy loss on the full training set:

    $$\min_{\theta} \; L(\theta; \mathcal{D}) = -\frac{1}{n} \sum_{i=1}^{n} \log f(\mathbf{x}^{(i)}; \theta)_{y^{(i)}}$$

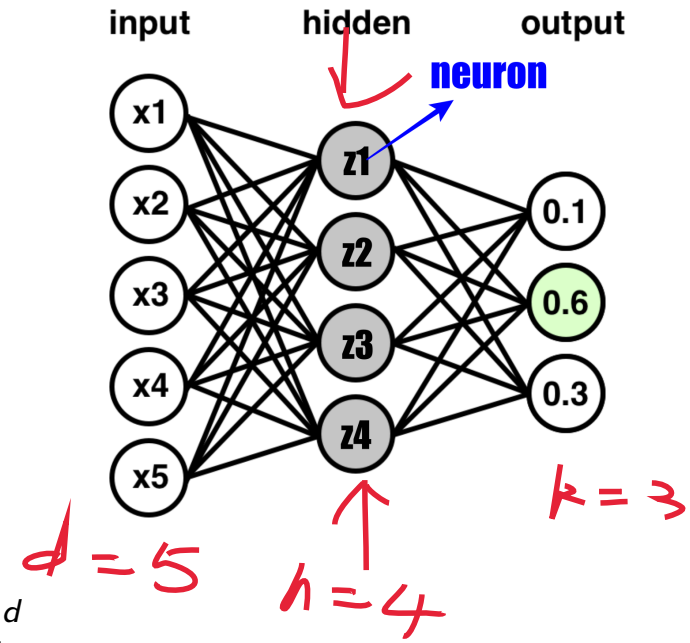
  - usually add a penalty term for reducing overfitting:

    $$\min_{\theta} L(\theta; \mathcal{D}) = -\frac{1}{n} \sum_{i=1}^{n} \log f(\mathbf{x}^{(i)}; \theta)_{y^{(i)}} + \lambda \|\mathbf{W}\|^2$$

MNIST classification



One-hidden-layer network



- input: data $\mathbf{x} \in \mathbb{R}^d$
- hidden neurons: $\mathbf{z}(\mathbf{x}) = g(\mathbf{W}^1\mathbf{x} + \mathbf{b}^1) \in \mathbb{R}^h$; here $\mathbf{W}^1 \in \mathbb{R}^{h \times d}$, $\mathbf{b}^1 \in \mathbb{R}^h$: weight matrix and bias vector for (dense) linear layer; $g : \mathbb{R} \to \mathbb{R}$: **element-wise** (non-linear) activation function: $z_j = g(o_j)$, where $\mathbf{o} = \mathbf{W}^1\mathbf{x} + \mathbf{b}^1$ is the output from linear layer
- output: $f(\mathbf{x}; \theta) = \text{softmax}(\mathbf{W}^2 z(\mathbf{x}) + \mathbf{b}^2) = \text{softmax}(\mathbf{W}^2 g(\mathbf{W}^1\mathbf{x} + \mathbf{b}^1) + \mathbf{b}^2) \in (0, 1)^k$; $\mathbf{W}^2 \in \mathbb{R}^{k \times h}$, $\mathbf{b}^1 \in \mathbb{R}^k$: weights and biases for the second linear layer
- $\theta = \{\mathbf{W}^1, \mathbf{b}^1, \mathbf{W}^2, \mathbf{b}^2\}$ are model parameters from linear layers

Dense linear layer

Dense layer defines a linear mapping, specified by the weights **W** and bias **b** (i.e., parameters)

- both input and output of the layer are **vectors**; if input is not, then vectorize it;
- each output o_j is a weighted average of inputs **z** plus a bias term:

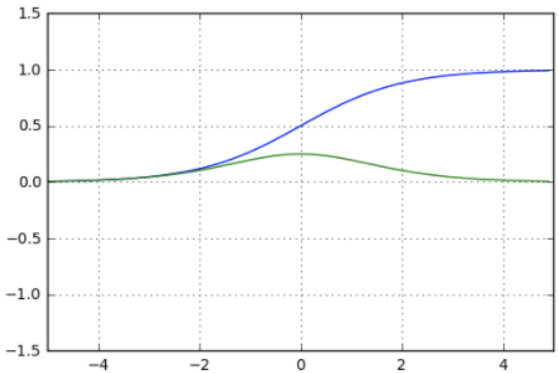
$$o_j = \mathbf{w}_j^\top \mathbf{z} + b_j$$

  $\mathbf{w}_j^\top$ is the j-th row of **W**
- **full connectivity** between the input and output components, justifying the name 'dense'; hence, also known as fully-connected layer
- another common type of linear layer is convolutional layer (later slides)

Element-wise activation function

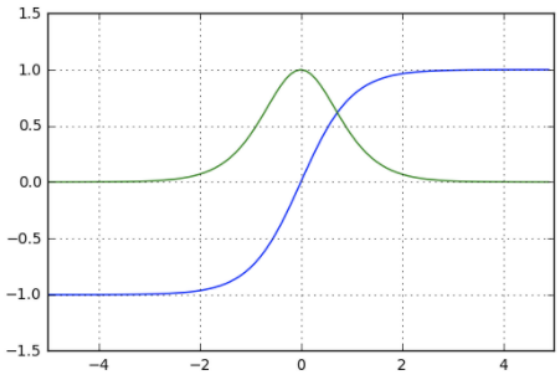
Element-wise activation: for any $\mathbf{z} \in \mathbb{R}^h$

$$g(\mathbf{z})_j = g(z_j),\ j = 1, \ldots, h$$



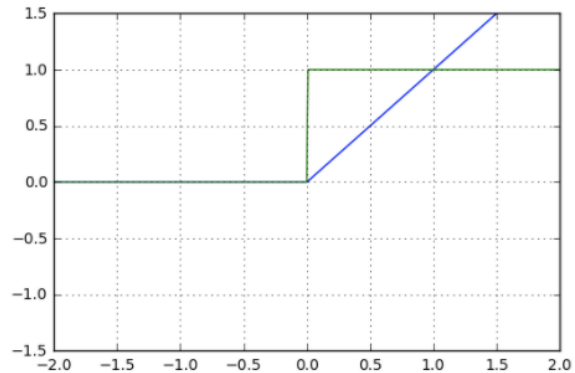
$$\mathrm{sigm}(x) = \frac{1}{1+e^{-x}}$$

$$\mathrm{sigm}'(x) = \mathrm{sigm}(x)(1-\mathrm{sigm}(x))$$



$$\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$$

$$\tanh'(x) = 1 - \tanh(x)^2$$

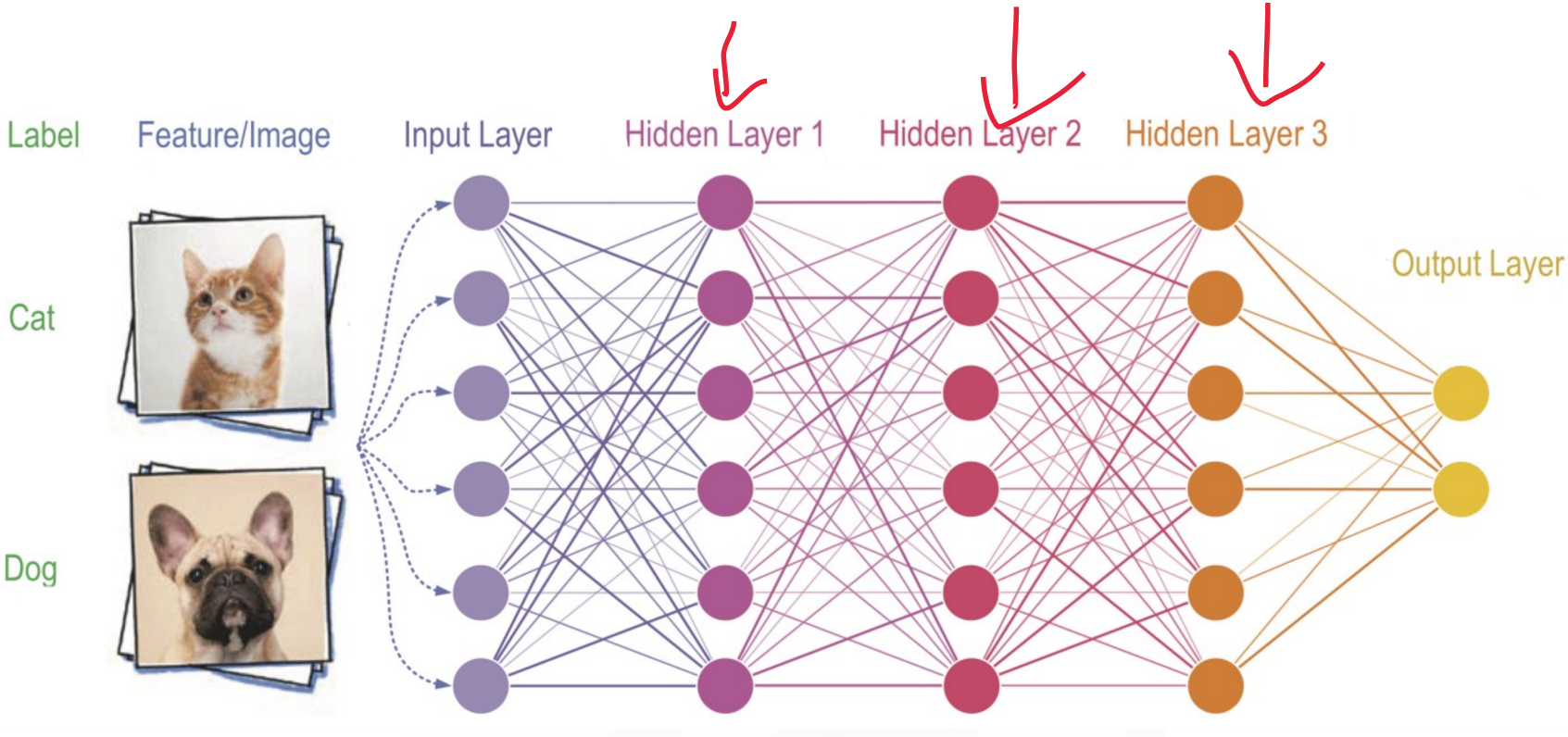


$$\mathrm{relu}(x) = \max(0, x)$$

$$\mathrm{relu}'(x) = 1_{x>0}$$

- sigm: sigmoid; ReLU: rectified linear unit
- blue: activation function
- green: derivative of activation function

Deep neural network (DNN)



- multiple layers of hidden neurons: linear layers with each followed by element-wise non-linear activation
- parameters are collection of weights and biases from linear layers
- softmax output layer, cross entropy loss

Universal approximation theorem

Why DNN models?

- in classification tasks, ML aims to learn an underlaying decision function $\hat{f}$ that maps any data to its correct label
- DNN models can approximate a decision function $\hat{f}$ with arbitrary precision, given sufficient depth

Theorem (Universal Approximation Theorem)

For any Lebesgue-integrable function $\hat{f} : \mathbb{R}^d \to \mathbb{R}$ *and any* $\epsilon > 0$, *there* **exists** *a deep dense (fully-connected) ReLU network* $\mathcal{A}$ *with width* $\leq d + 4$, *such that the function* $F_{\mathcal{A}}$ *represented by the network satisfies*

$$\int_{\mathbb{R}^n} |\hat{f}(\mathbf{x}) - F_{\mathcal{A}}(\mathbf{x})| \mathrm{d}\mathbf{x} < \epsilon$$

- on computational side, learning (near-)optimal DNNs can be difficult due to highly non-convex optimization

Stochastic gradient descent: epoch training

Training neural network (empirical risk minimization):

$$\min_{\theta} L(\theta; \mathcal{D}) = -\frac{1}{n} \sum_{i=1}^{n} \log f(\mathbf{x}^{(i)}; \theta)_{y^{(i)}}$$

- initialize θ randomly
- for m epochs do:
  - randomly select a small batch of samples $\mathcal{B} \subset \mathcal{D}$ (without replacement)
  - compute gradient of loss functions restricted to batch $\mathcal{B}$: $\nabla L(\theta; \mathcal{B})$, where

$$\nabla L(\theta; \mathcal{B}) = -\frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_{\theta} \log f(\mathbf{x}^{(i)}; \theta)_{y^{(i)}} \approx \nabla L(\theta; \mathcal{D})$$

  - update the parameters using learning rate $\eta > 0$:

$$\theta \leftarrow \theta - \eta \nabla L(\theta; \mathcal{B})$$

- stop when reaching criterion
  - entropy loss stops decreasing

step size

Remarks

- each epoch is a **complete pass** through the training dataset $\mathcal{D}$
- the number of SGD iterations for one epoch is $\lceil \frac{n}{|\mathcal{B}|} \rceil$
- choice of learning rate η is crucial; in theory, should decay to 0 for convergence guarantee
  - start with a large value first: $\eta = 0.1$ or even $\eta = 1$; divide by 10 and retry in case of divergence
  - monitor training loss and divide η by 2 or 10 when no progress
- batch gradient is computed by applying **chain rule** for composite functions
  - automatic differentiation packages are available for implementation
- the SGD with chain rule for gradient evaluations is also known as back-propagation algorithm (Hinton '86)

State-of-the-art optimizers

- SGD with Nesterov momentum (Nesterov '83):

$$\nu^t = \theta^{t-1} + \gamma(\theta^{t-1} - \theta^{t-2})$$
$$\theta^t = \nu^t - \eta_t \nabla f(\nu^t; \mathcal{B}_t)$$

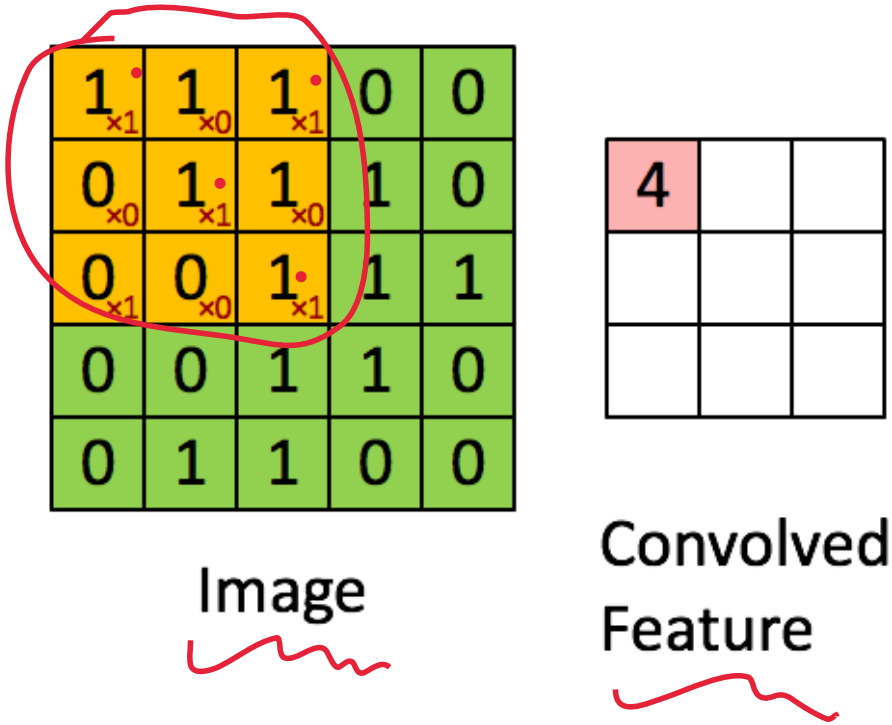
  - the momentum γ is typically set to 0.9
  - accelerate the standard SGD; simple to implement
  - similar to SGD, need learning rate scheduling

- Adam (Kingma & Ba '15):
  - update each parameter with an individual learning rate
  - only need to set a initial learning rate; the learning rates adapt themselves during training

Convolutional neural network

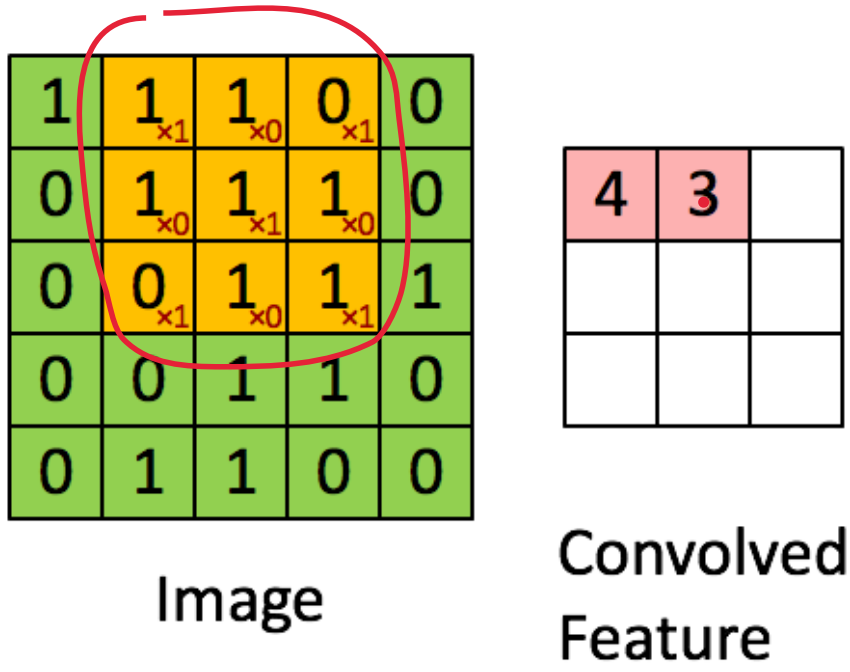
Convolutional neural network features convolutional (linear) layers:

- take image or features output from hidden layers as input in its **original shape** without vectorization
- better capture the **spatial information** in an image
- trainable weights are typically 3-D or 4-D tensors, called filters
- **weights sharing** and **sparse connectivity** in contrast to full connectivity in dense layers
- much more compact in terms of model size (i.e., less parameters)



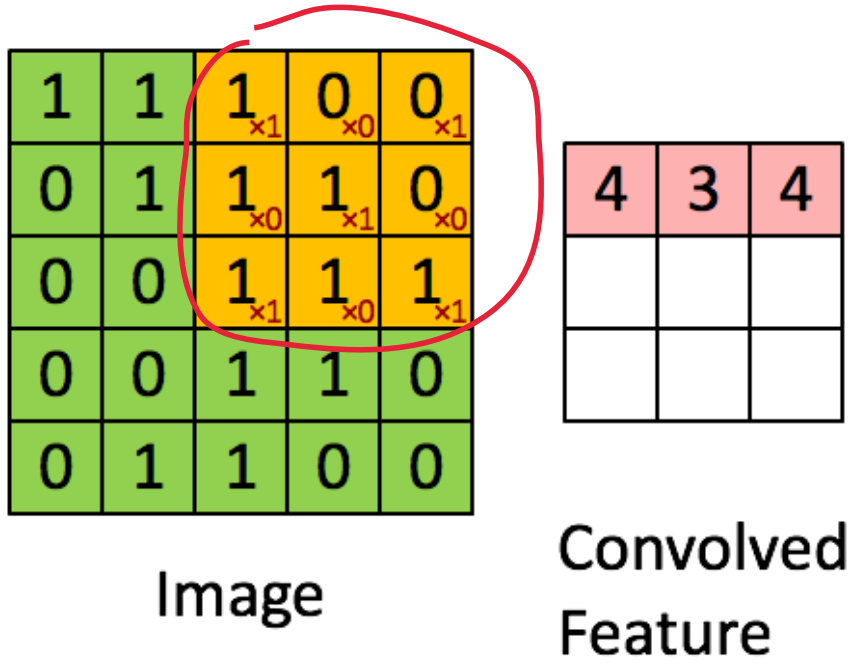
3×3 window dot product with the 3×3 filter

- input: 5×5 matrix
- a filter of size 3×3: $\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$, stride $= 1$ (shift the window by one grid)
- output: 3×3 matrix (5 (input dimension) -3 (filter dimension) $+1$(stride) $=3$ (output dimension))



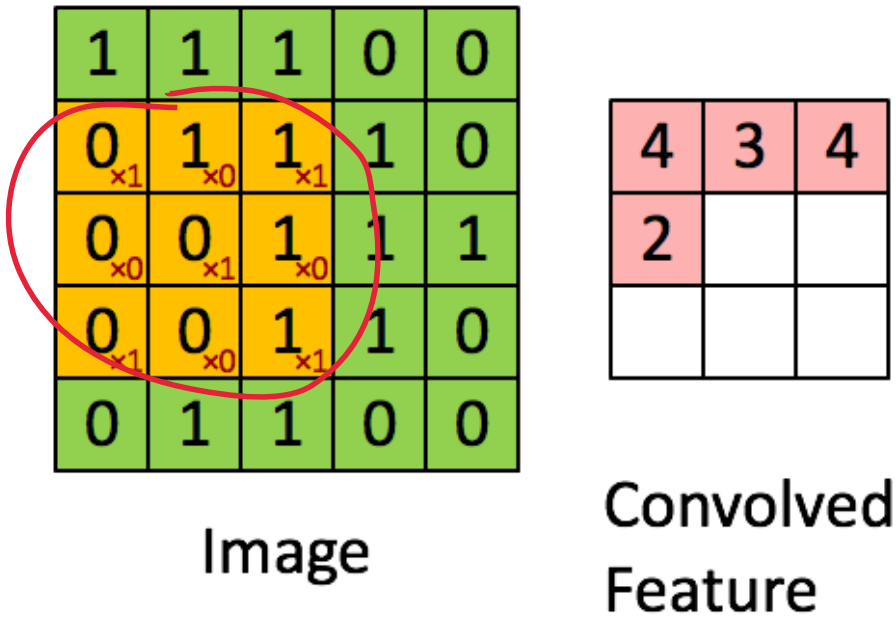
3×3 window dot product with the 3×3 filter

- input: 5×5 matrix
- a filter of size 3×3: $\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$, stride $= 1$ (shift the window by one grid)
- output: 3×3 matrix (5 (input dimension) -3 (filter dimension) $+1$(stride) $=3$ (output dimension))



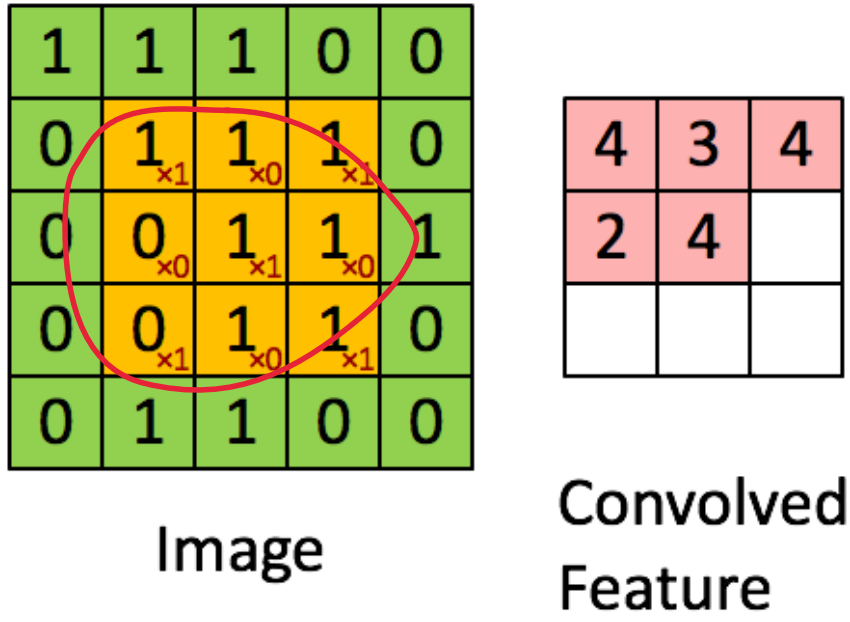
3×3 window dot product with the 3×3 filter

- input: 5×5 matrix
- a filter of size 3×3: $\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$, stride $= 1$ (shift the window by one grid)
- output: 3×3 matrix (5 (input dimension) -3 (filter dimension) $+1$(stride) $=3$ (output dimension))



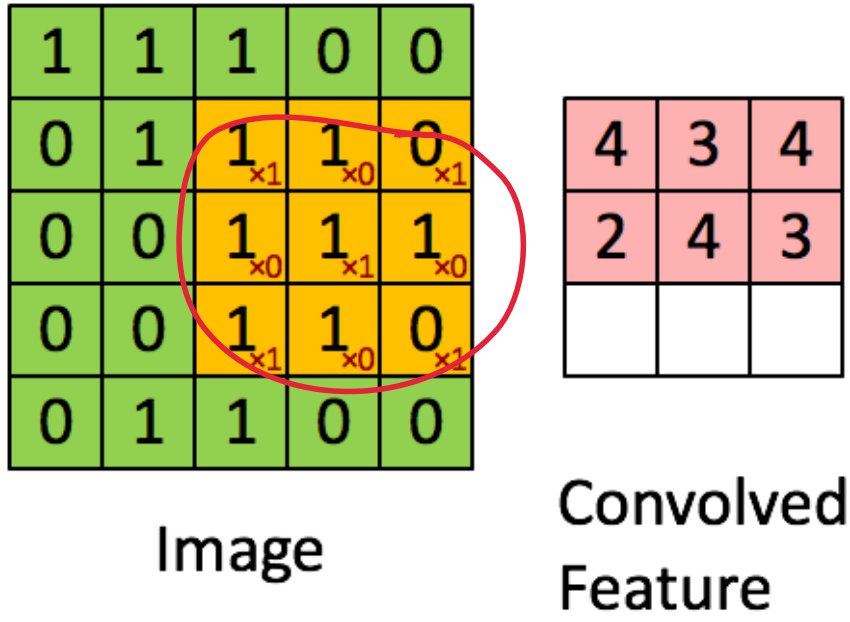
3×3 window dot product with the 3×3 filter

- input: 5×5 matrix
- a filter of size 3×3: $\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$, stride $= 1$ (shift the window by one grid)
- output: 3×3 matrix (5 (input dimension) -3 (filter dimension) $+1$(stride) $=3$ (output dimension))



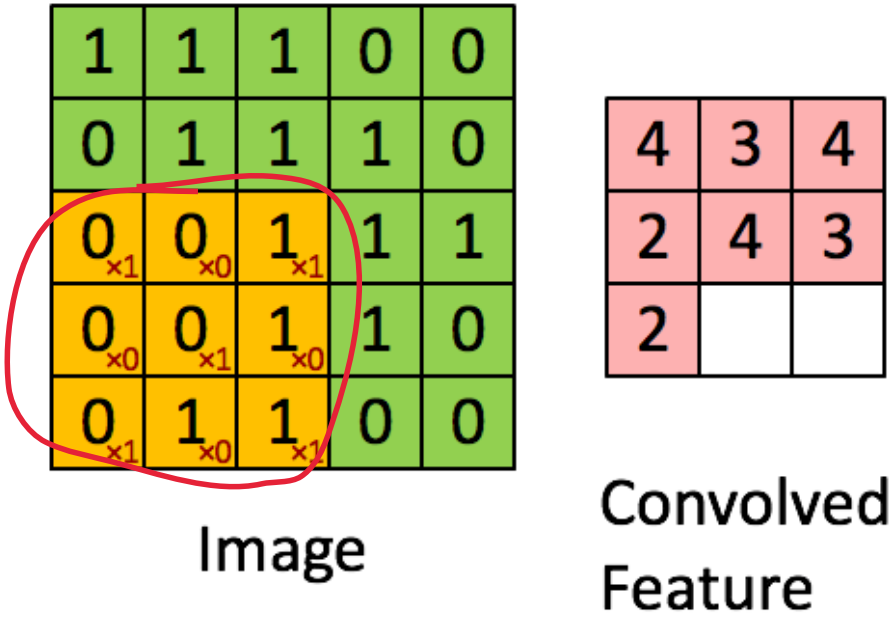
3×3 window dot product with the 3×3 filter

- input: 5×5 matrix
- a filter of size 3×3: $\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$, a stride $= 1$ (shift the window by one grid)
- output: 3×3 matrix (5 (input dimension) -3 (filter dimension) $+1$(stride) $=3$ (output dimension))



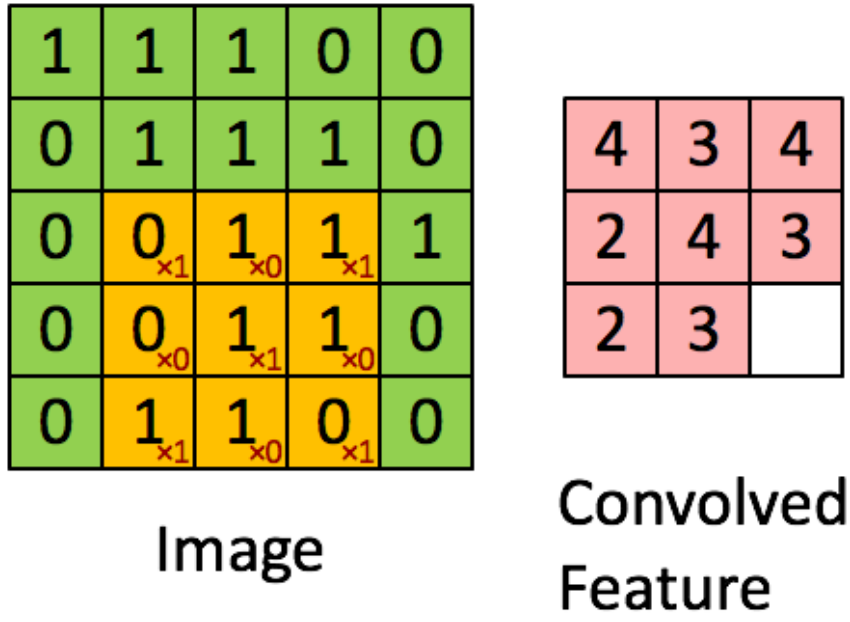
3×3 window dot product with the 3×3 filter

- input: 5×5 matrix
- filter of size 3×3: $\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$, stride $= 1$ (shift the window by one grid)
- output: 3×3 matrix (5 (input dimension) -3 (filter dimension) $+1$(stride) $=3$ (output dimension))



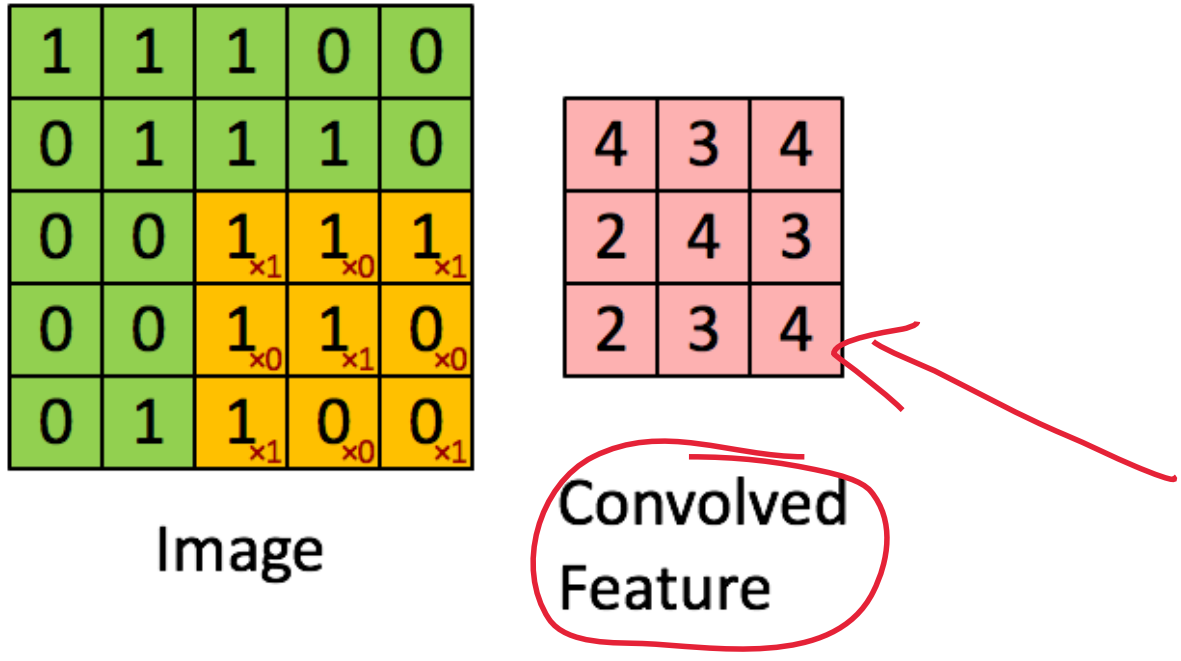
3×3 window dot product with the 3×3 filter

- input: 5×5 matrix
- a filter of size 3×3: $\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$, stride $= 1$ (shift the window by one grid)
- output: 3×3 matrix (5 (input dimension) -3 (filter dimension) $+1$(stride) $=3$ (output dimension))



3×3 window dot product with the 3×3 filter

- input: 5×5 matrix
- a filter of size 3×3: $\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$, stride $= 1$ (shift the window by one grid)
- output: 3×3 matrix (5 (input dimension) -3 (filter dimension) $+1$(stride) $=3$ (output dimension))



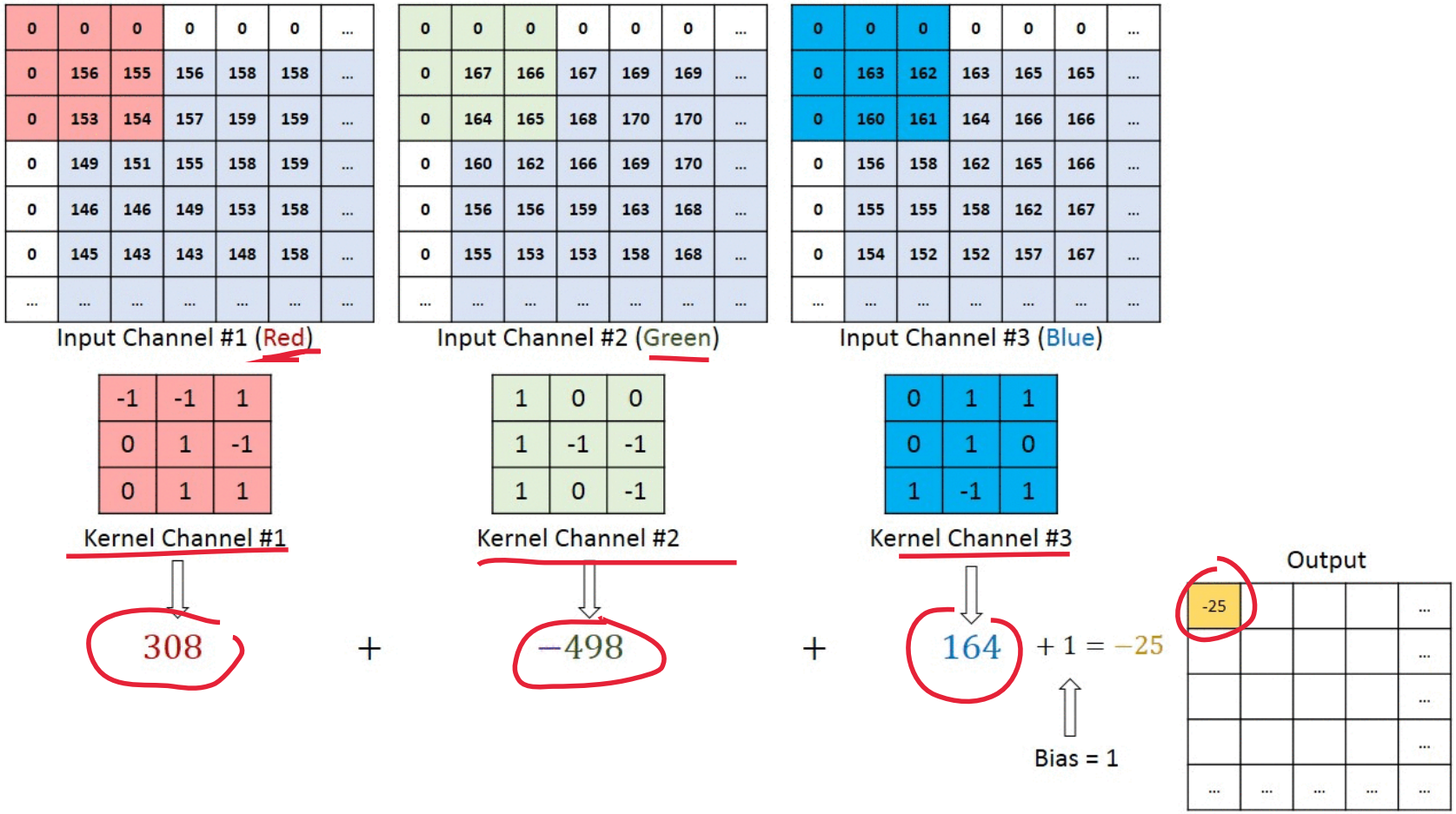
3×3 window dot product with the 3×3 filter

- input: 5×5 matrix
- a filter of size 3×3: $\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$, stride $= 1$ (shift the window by one grid)
- output: 3×3 matrix (5 (input dimension) -3 (filter dimension) +1(stride) =3 (output dimension))

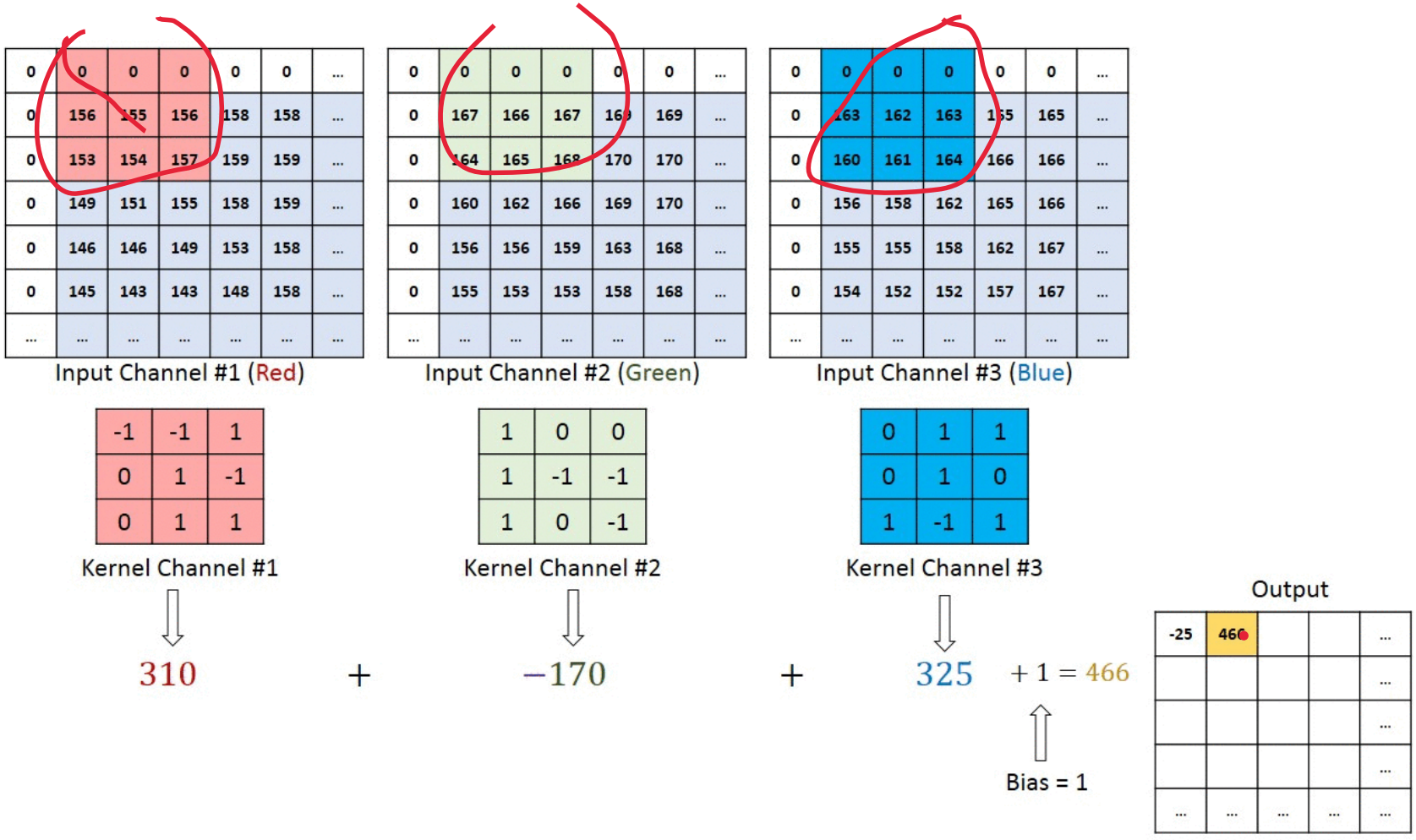
Remarks for 2-D convolutions

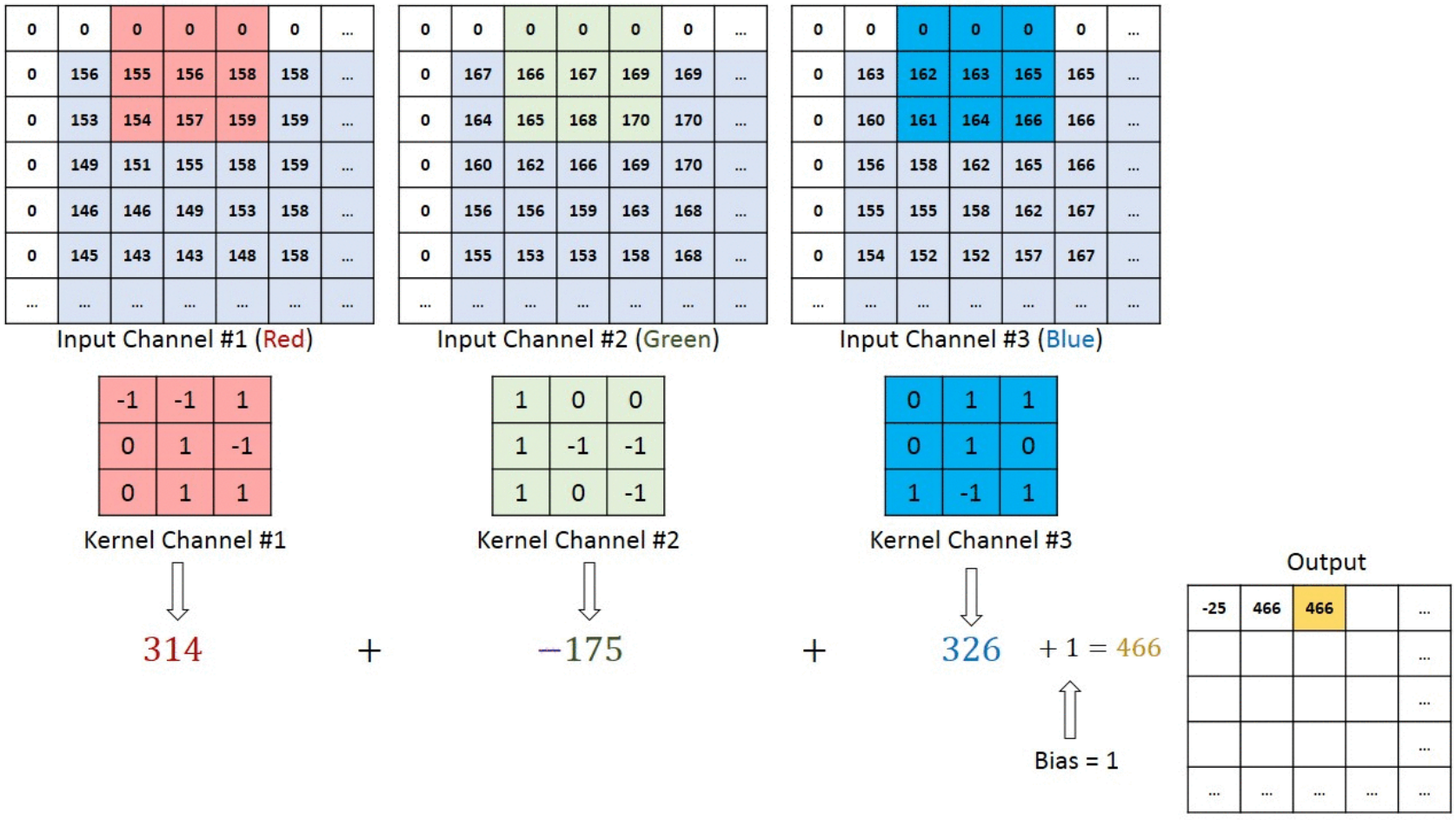
2-D convolution works for matrix input

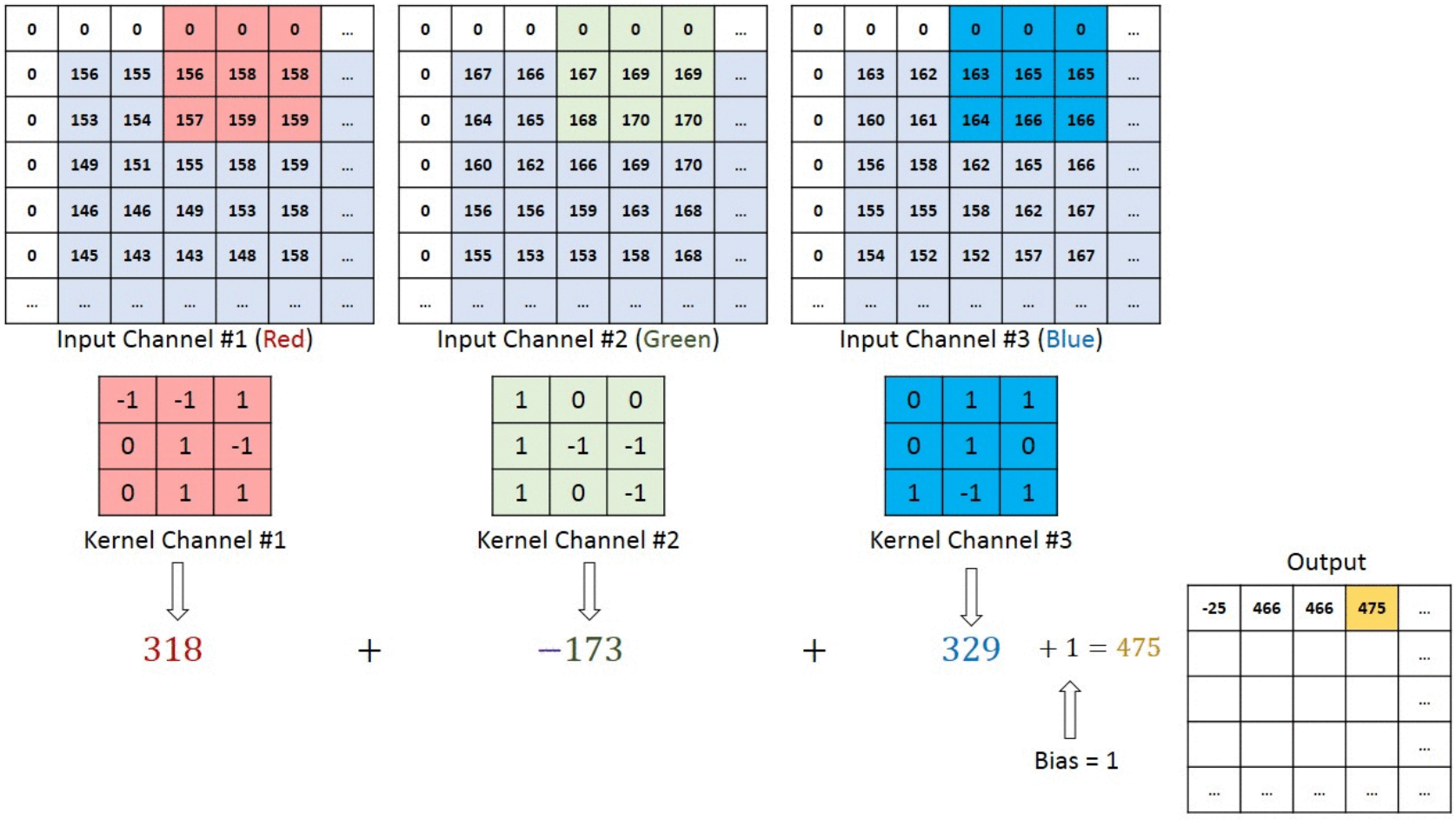
- one convolution filter of (kernel) size 3×3 (with just 9 weights) transforms 5×5 input into 3×3 output
- in contrast, a dense layer would have $9 \times 25 = 225$ weights!
- could have bias term
- could have multiple filters of (kernel) size 3×3 (say, 4 filters with 4×9 weights), then the output is a $4 \times 3 \times 3$ tensor (each filter gives a 3×3 slice of the 3-D tensor)
- other typical choices of kernel size are 5×5 and 7×7

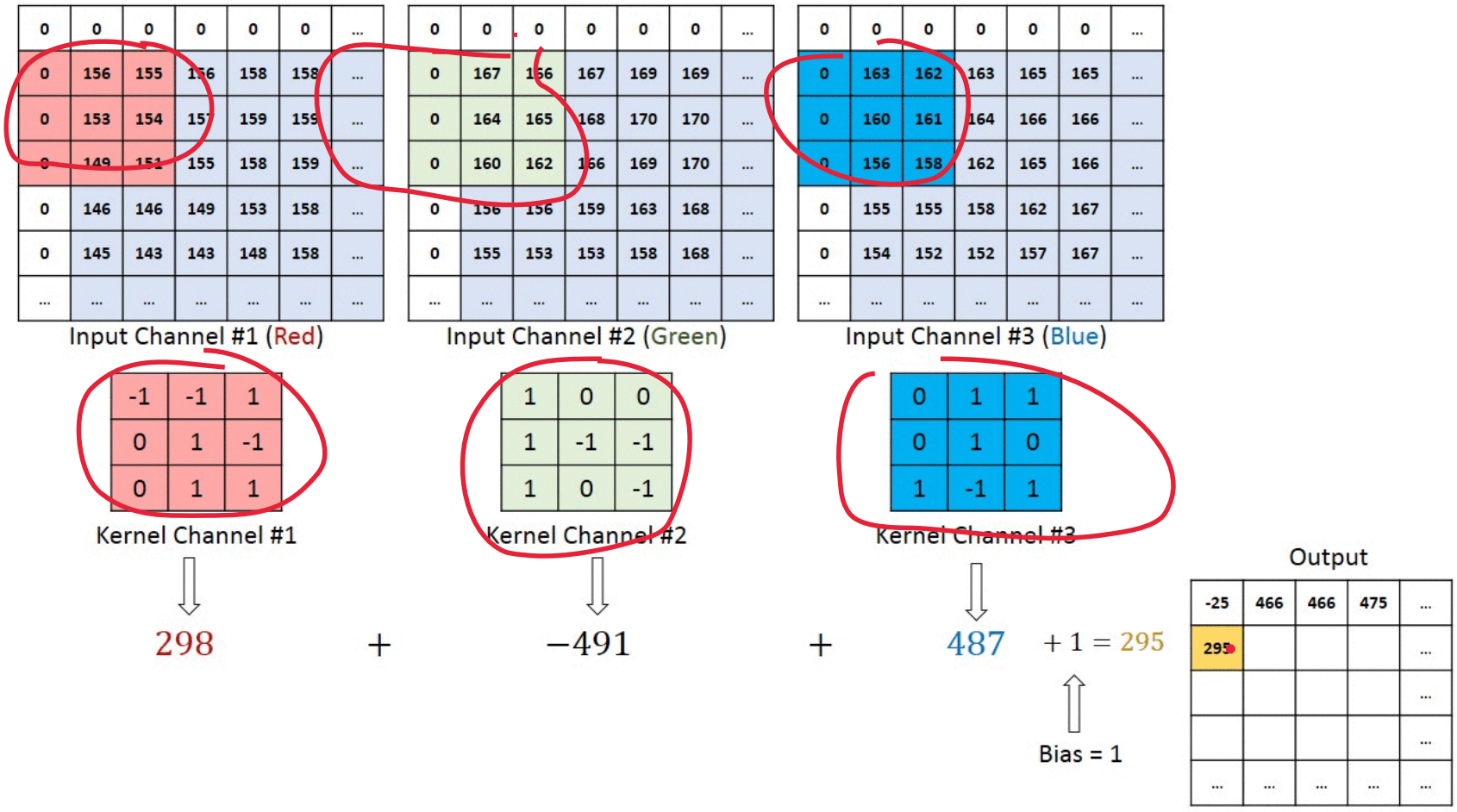


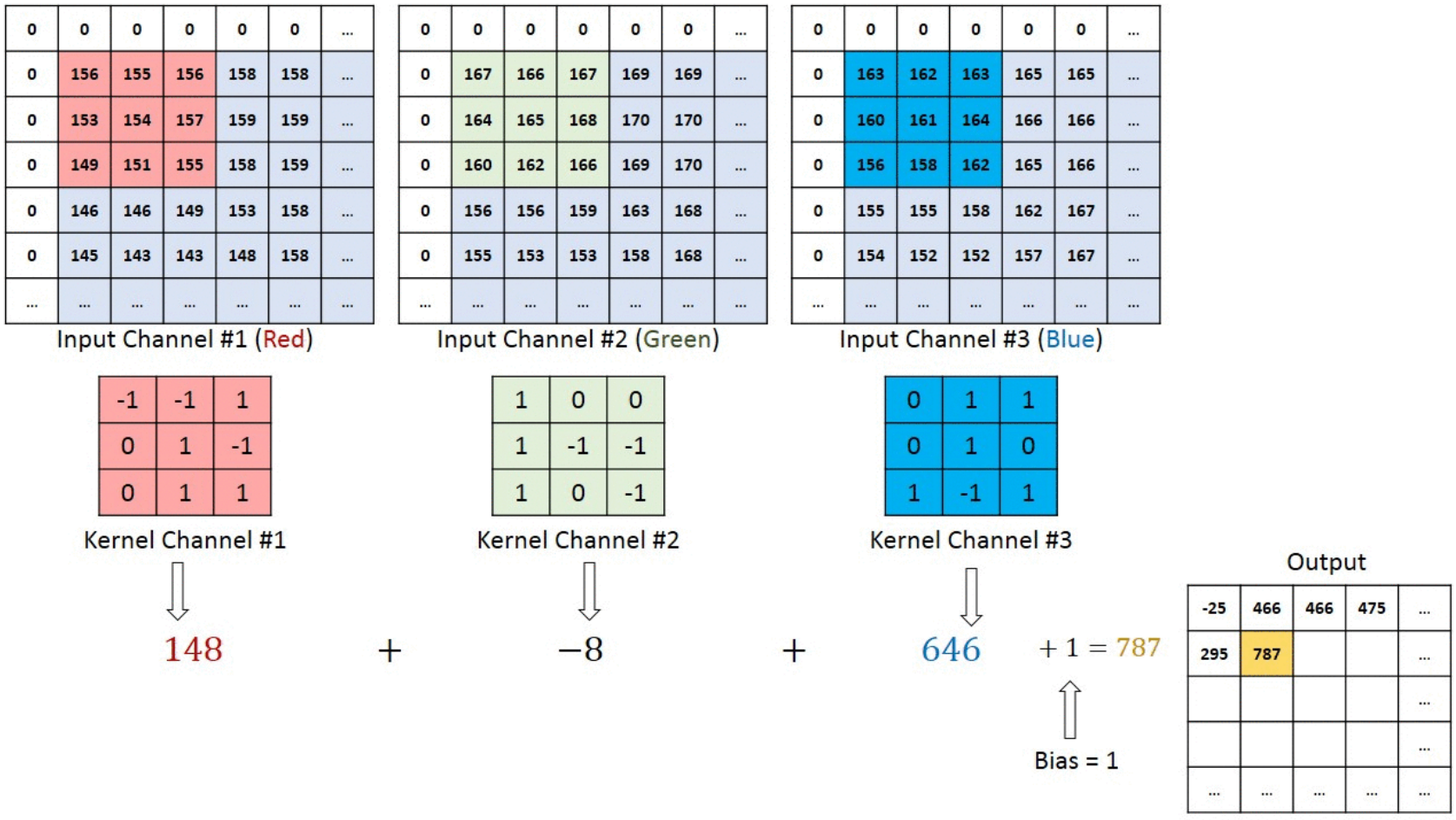
- input size: $3 \times d \times d$
- filter size: $3 \times 3 \times 3$ (with kernel size 3×3), stride $= 1$
- output size : $(d-2) \times (d-2)$

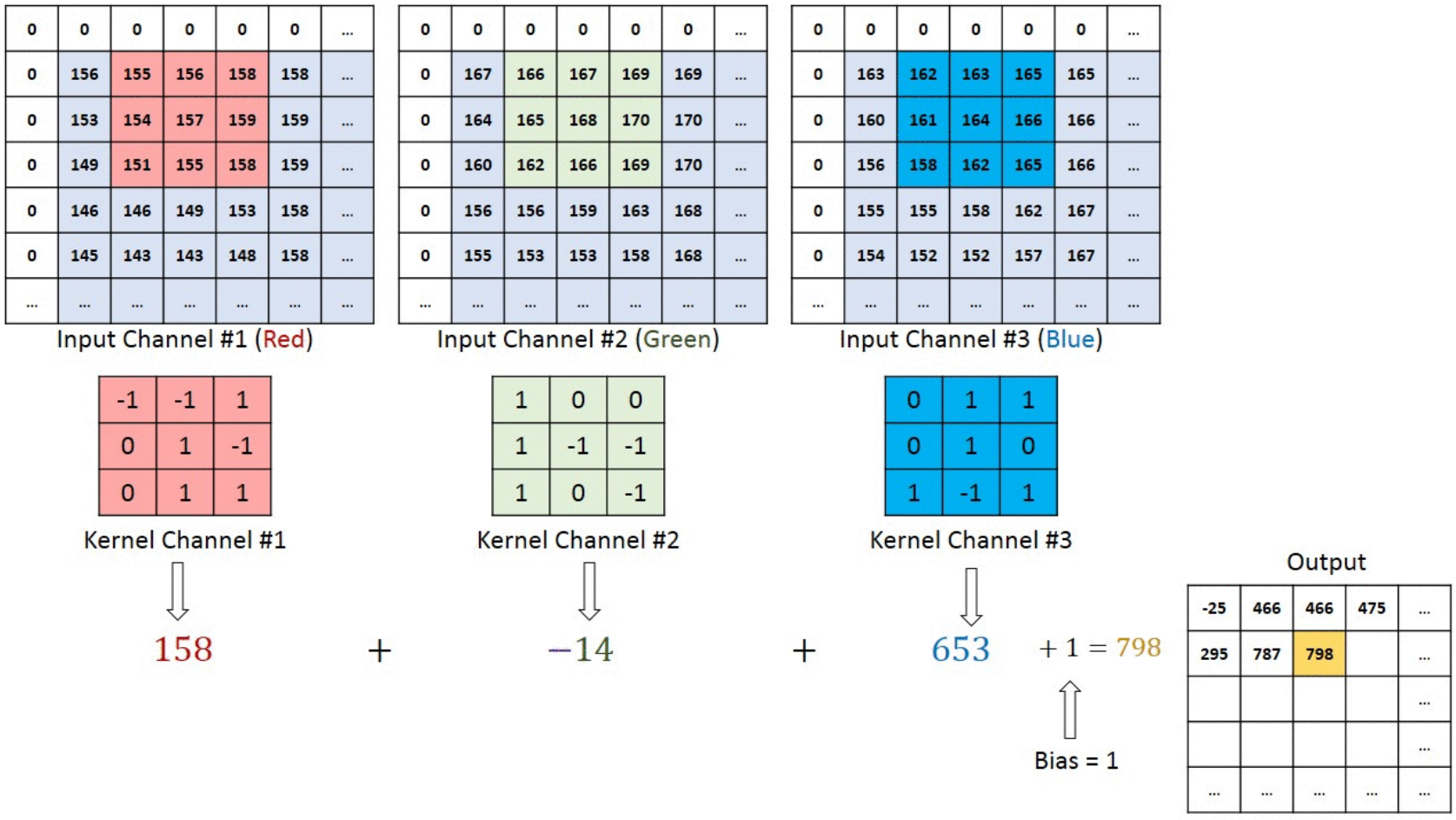

0 0 0 0 0 0 ...
0 156 155 156 158 158 ...
0 153 154 157 159 159 ...
0 149 151 155 158 159 ...
0 146 146 149 153 158 ...
0 145 143 143 148 158 ...
... ...
Input Channel #1 (Red)
0 0 0 0 0 0 ...
0 167 166 167 169 169 ...
0 164 165 168 170 170 ...
0 160 162 166 169 170 ...
0 156 156 159 163 168 ...
0 155 153 153 158 168 ...
... ...
Input Channel #2 (Green)
0 0 0 0 0 0 ...
0 163 162 163 165 165 ...
0 160 161 164 166 166 ...
0 156 158 162 165 166 ...
0 155 155 158 162 167 ...
0 154 152 152 157 167 ...
... ...
Input Channel #3 (Blue)
-1 -1 1
0 1 -1
0 1 1
Kernel Channel #1
1 0 0
1 -1 -1
1 0 -1
Kernel Channel #2
0 1 1
0 1 0
1 -1 1
Kernel Channel #3
310
+
−170
+
325
+ 1 = 466
Bias = 1
Output
-25 466 ... ...
... ...
... ...
... ...
... ...
... ...

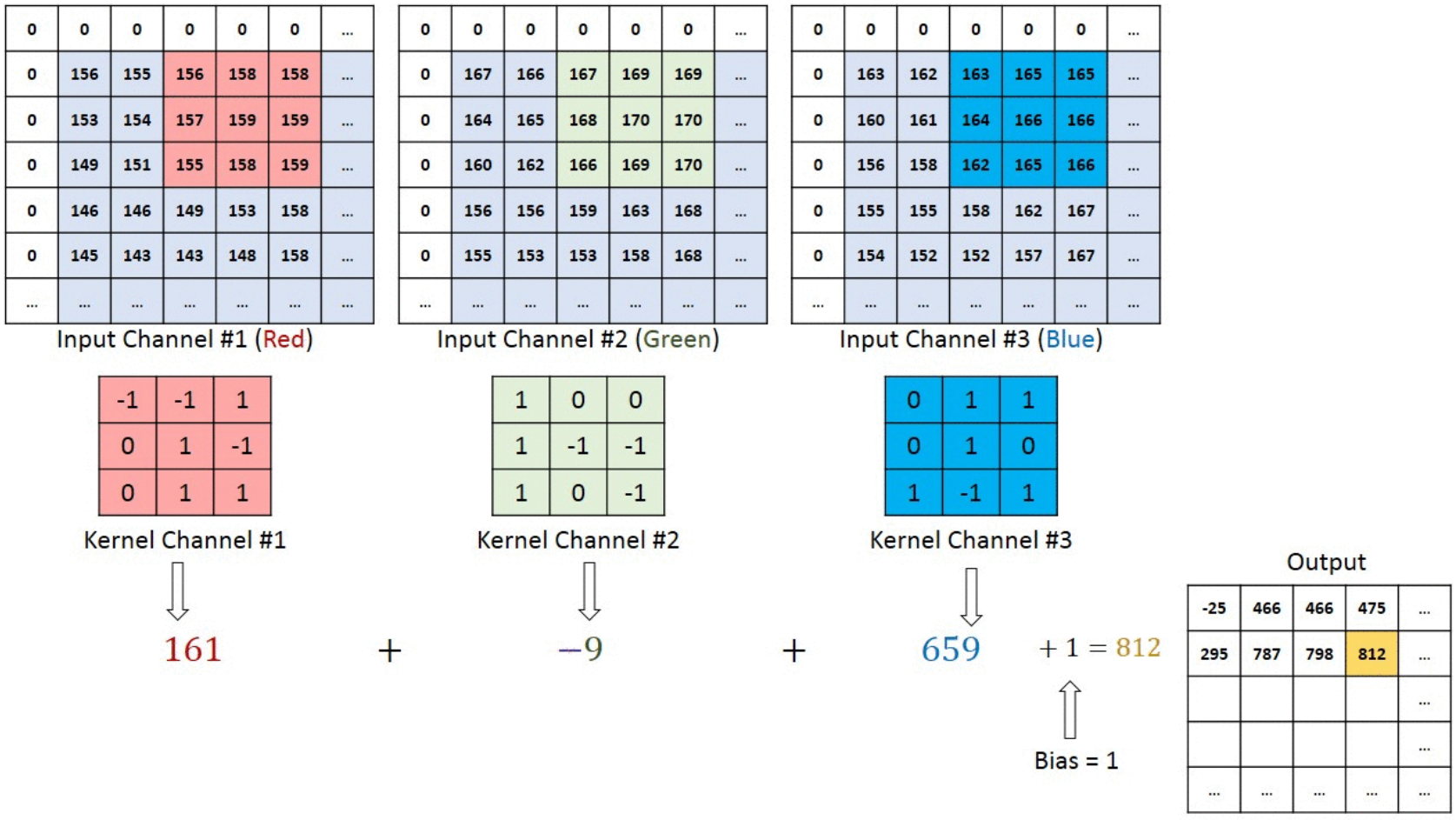

0 0 0 0 0 0 ...
0 156 155 156 158 158 ...
0 153 154 157 159 159 ...
0 149 151 155 158 159 ...
0 146 146 149 153 158 ...
0 145 143 143 148 158 ...
... ...
Input Channel #1 (Red)
0 0 0 0 0 0 ...
0 167 166 167 169 169 ...
0 164 165 168 170 170 ...
0 160 162 166 169 170 ...
0 156 156 159 163 168 ...
0 155 153 153 158 168 ...
... ...
Input Channel #2 (Green)
0 0 0 0 0 0 ...
0 163 162 163 165 165 ...
0 160 161 164 166 166 ...
0 156 158 162 165 166 ...
0 155 155 158 162 167 ...
0 154 152 152 157 167 ...
... ...
Input Channel #3 (Blue)
-1 -1 1
0 1 -1
0 1 1
Kernel Channel #1
1 0 0
1 -1 -1
1 0 -1
Kernel Channel #2
0 1 1
0 1 0
1 -1 1
Kernel Channel #3
314
+
−175
+
326
+ 1 = 466
Bias = 1
Output
-25 466 466 ...
... ...
... ...
... ...
... ...


0 0 0 0 0 0 ...
0 156 155 156 158 158 ...
0 153 154 157 159 159 ...
0 149 151 155 158 159 ...
0 146 146 149 153 158 ...
0 145 143 143 148 158 ...
... ...
Input Channel #1 (Red)
0 0 0 0 0 0 ...
0 167 166 167 169 169 ...
0 164 165 168 170 170 ...
0 160 162 166 169 170 ...
0 156 156 159 163 168 ...
0 155 153 153 158 168 ...
... ...
Input Channel #2 (Green)
0 0 0 0 0 0 ...
0 163 162 163 165 165 ...
0 160 161 164 166 166 ...
0 156 158 162 165 166 ...
0 155 155 158 162 167 ...
0 154 152 152 157 167 ...
... ...
Input Channel #3 (Blue)
-1 -1 1
0 1 -1
0 1 1
Kernel Channel #1
1 0 0
1 -1 -1
1 0 -1
Kernel Channel #2
0 1 1
0 1 0
1 -1 1
Kernel Channel #3
318
+
−173
+
329
+ 1 = 475
Bias = 1
Output
-25 466 466 475 ...
... ...

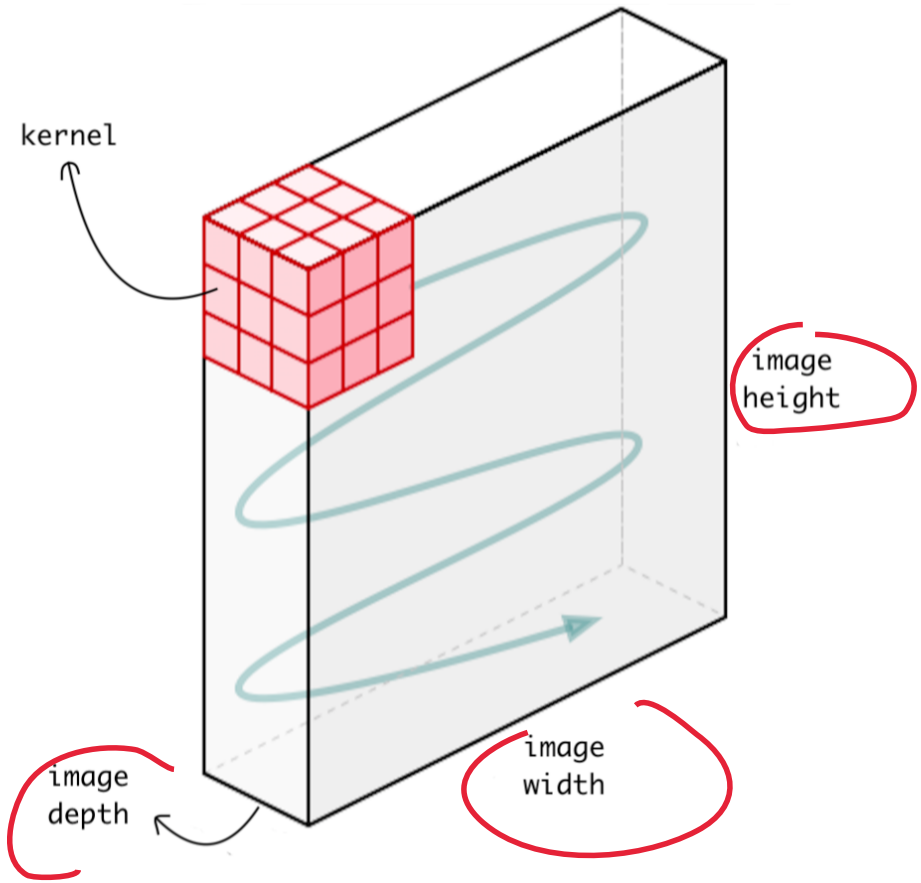

Input Channel #1 (Red)
Input Channel #2 (Green)
Input Channel #3 (Blue)
Kernel Channel #1
Kernel Channel #2
Kernel Channel #3
298
+
−491
+
487
+ 1 = 295
Bias = 1
Output


0 0 0 0 0 0 ...
0 156 155 156 158 158 ...
0 153 154 157 159 159 ...
0 149 151 155 158 159 ...
0 146 146 149 153 158 ...
0 145 143 143 148 158 ...
... ...
Input Channel #1 (Red)
0 0 0 0 0 0 ...
0 167 166 167 169 169 ...
0 164 165 168 170 170 ...
0 160 162 166 169 170 ...
0 156 156 159 163 168 ...
0 155 153 153 158 168 ...
... ...
Input Channel #2 (Green)
0 0 0 0 0 0 ...
0 163 162 163 165 165 ...
0 160 161 164 166 166 ...
0 156 158 162 165 166 ...
0 155 155 158 162 167 ...
0 154 152 152 157 167 ...
... ...
Input Channel #3 (Blue)
-1 -1 1
0 1 -1
0 1 1
Kernel Channel #1
1 0 0
1 -1 -1
1 0 -1
Kernel Channel #2
0 1 1
0 1 0
1 -1 1
Kernel Channel #3
148
+
−8
+
646
+ 1 = 787
Bias = 1
Output
-25 466 466 475 ...
295 787 ... ...
... ...
... ...
... ...


0 0 0 0 0 0 ...
0 156 155 156 158 158 ...
0 153 154 157 159 159 ...
0 149 151 155 158 159 ...
0 146 146 149 153 158 ...
0 145 143 143 148 158 ...
... ...
Input Channel #1 (Red)
0 0 0 0 0 0 ...
0 167 166 167 169 169 ...
0 164 165 168 170 170 ...
0 160 162 166 169 170 ...
0 156 156 159 163 168 ...
0 155 153 153 158 168 ...
... ...
Input Channel #2 (Green)
0 0 0 0 0 0 ...
0 163 162 163 165 165 ...
0 160 161 164 166 166 ...
0 156 158 162 165 166 ...
0 155 155 158 162 167 ...
0 154 152 152 157 167 ...
... ...
Input Channel #3 (Blue)
-1 -1 1
0 1 -1
0 1 1
Kernel Channel #1
1 0 0
1 -1 -1
1 0 -1
Kernel Channel #2
0 1 1
0 1 0
1 -1 1
Kernel Channel #3
158 + −14 + 653 + 1 = 798
Bias = 1
Output
-25 466 466 475 ...
295 787 798 ... ...


0 0 0 0 0 0 ...
0 156 155 156 158 158 ...
0 153 154 157 159 159 ...
0 149 151 155 158 159 ...
0 146 146 149 153 158 ...
0 145 143 143 148 158 ...
Input Channel #1 (Red)
0 0 0 0 0 0 ...
0 167 166 167 169 169 ...
0 164 165 168 170 170 ...
0 160 162 166 169 170 ...
0 156 156 159 163 168 ...
0 155 153 153 158 168 ...
Input Channel #2 (Green)
0 0 0 0 0 0 ...
0 163 162 163 165 165 ...
0 160 161 164 166 166 ...
0 156 158 162 165 166 ...
0 155 155 158 162 167 ...
0 154 152 152 157 167 ...
Input Channel #3 (Blue)
-1 -1 1
0 1 -1
0 1 1
Kernel Channel #1
1 0 0
1 -1 -1
1 0 -1
Kernel Channel #2
0 1 1
0 1 0
1 -1 1
Kernel Channel #3
161
+
−9
+
659
+ 1 = 812
Bias = 1
Output
-25 466 466 475 ...
295 787 798 812 ...
...
...
... ...

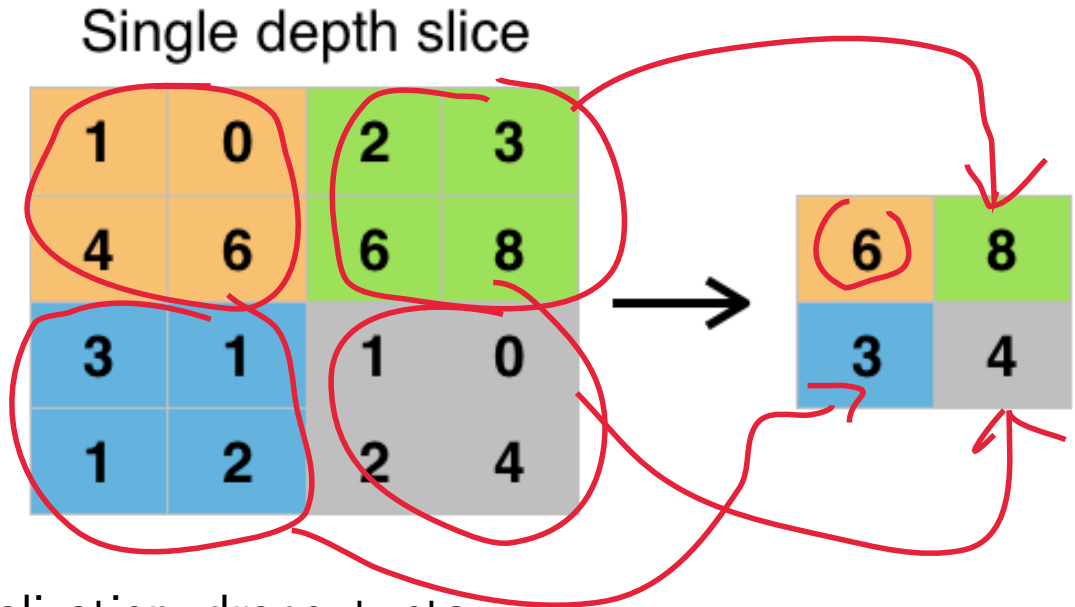
3-D inputs to convolutional layer



Convolve $d_{in} \times w_{in} \times h_{in}$ input features with d_{out} filters of size $d_{in} \times d_{ker} \times d_{ker}$ and stride $= 1$, then size of output features is $d_{out} \times (w_{in} - d_{ker} + 1) \times (h_{in} - d_{ker} + 1)$

Other layers

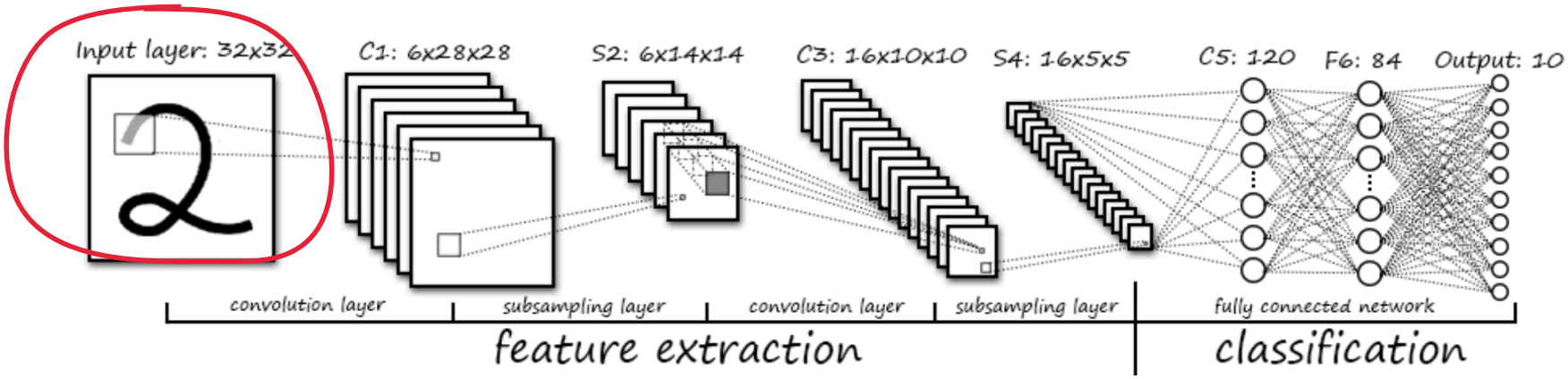
- Pooling or subsampling layers reduce the spatial size of features to reduce the amount of parameters; operate on each feature slice independently
  - 2×2 max pooling (with stride 2) is commonly used



- batch-normalization, dropout, etc.

LeNet-5 for MNIST classification

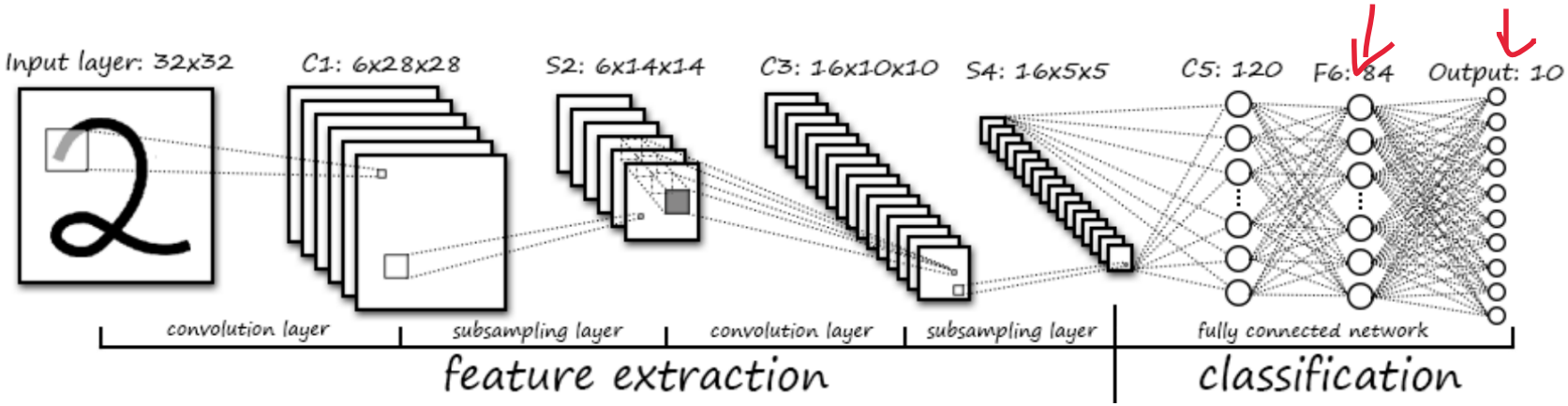
The original LeNet-5 for handwritten digits recognition (LeCun et al. '98)



- pad images with zeros and increase the size to 32×32
- convolutional layer C_1 with 6 filters of size 5×5 (followed by tanh activation layer), then 2×2 max-pooling layer S_1
  - **ReLU activation is used nowadays for better accuracy**
- convolutional layer C_3 with 16 filters of size $6 \times 5 \times 5$ (followed by tanh activation), then 2×2 max-pooling layer S_4

$32 - 5 + 1 = 28$

$14 - 5 + 1 = 10$



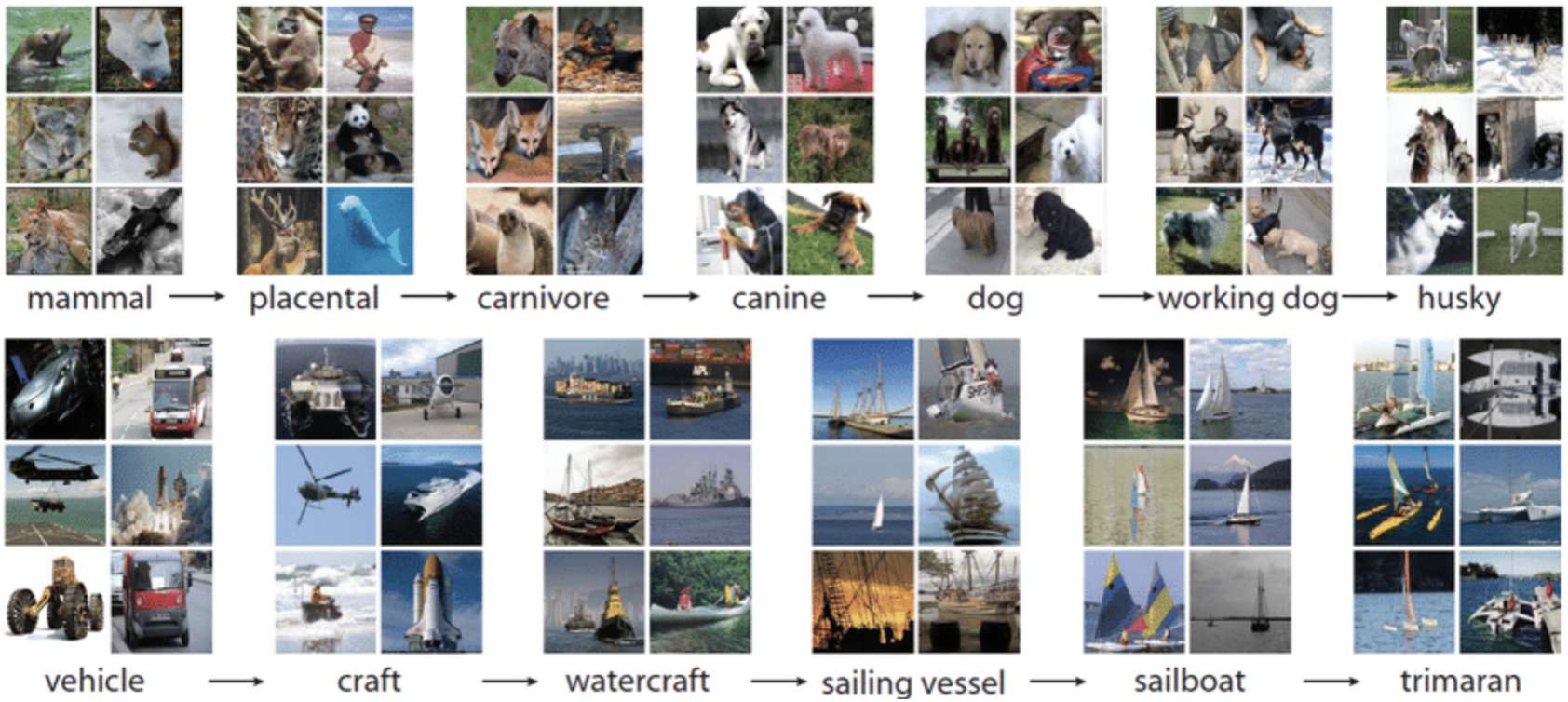
- convolutional layer C_5 with 120 filters of size $16 \times 5 \times 5$, then flatten into 120-D vector
- dense layer of filter size 84×120 (followed by tanh activation)
- dense layer of filter size 10×84 followed by softmax output layer

In total 61,706 trainable parameters including weights and biases

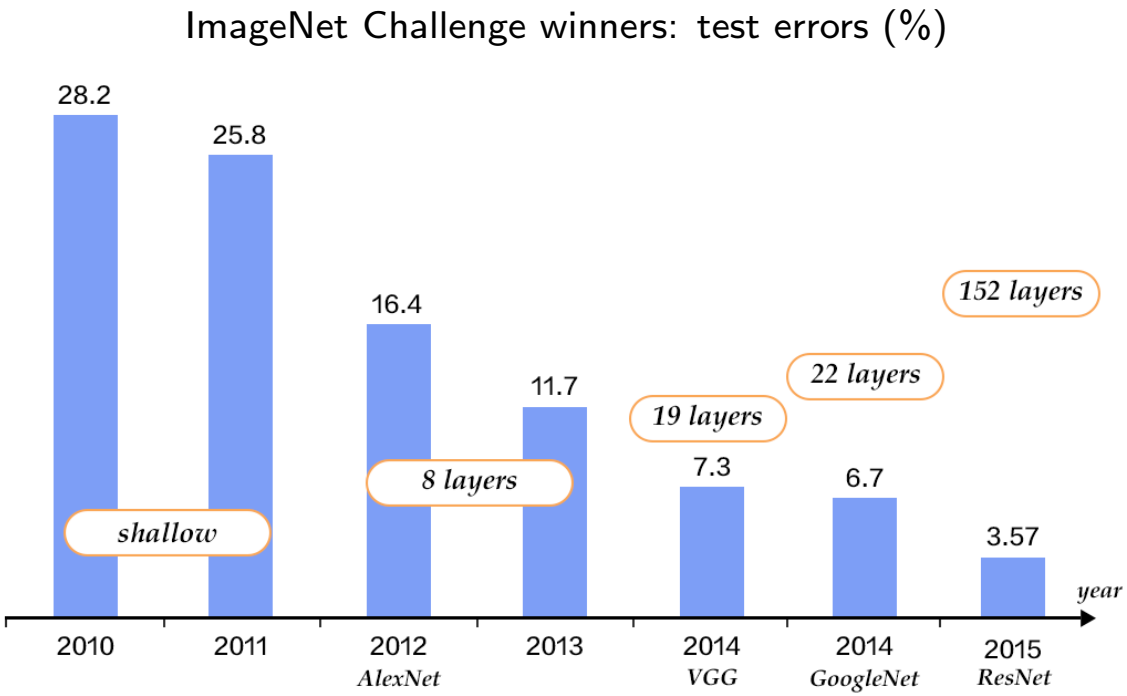
- easily $> 99\%$ test accuracy

ImageNet Classification Challenge

ImageNet dataset has $\sim$ 1.2M color images of size 256×256 for training and $\sim$ 100K for testing with $\sim$ 1000 object classes.



Models are Getting Deeper and Larger



ResNet (He et al. '15)

Super-human precision

152 layers

60 million parameters

224 megabyte memory

11.3 GFLOPs per prediction

Computations of training large scale networks are made parallelizable by using GPUs, an essential ingredient of the deep learning revolution

Summary

- deep learning is a class of machine learning methods, which uses multiple layers to progressively extract higher level features from the raw input
- DNNs as a class of functions, has great expressive power to approximate any decision function
- convolutional architectures are good at capturing spatial information and are popular for processing images
- SGD based optimization is the workhorse for the training of DNNs
- The computations of training process are made parallelizable and fast on GPUs