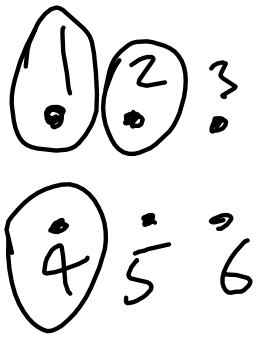
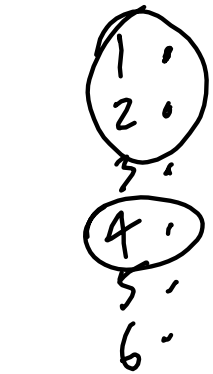
Convolutional Neural Networks





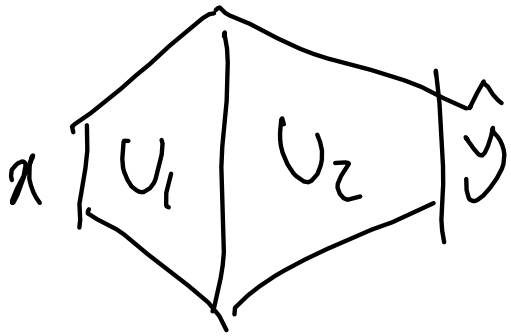
Deep Learning (ENGG*6600*07)

School of Engineering,
University of Guelph, ON, Canada

Course Instructor: Benyamin Ghojogh
Fall 2023

Convolutional Neural Networks

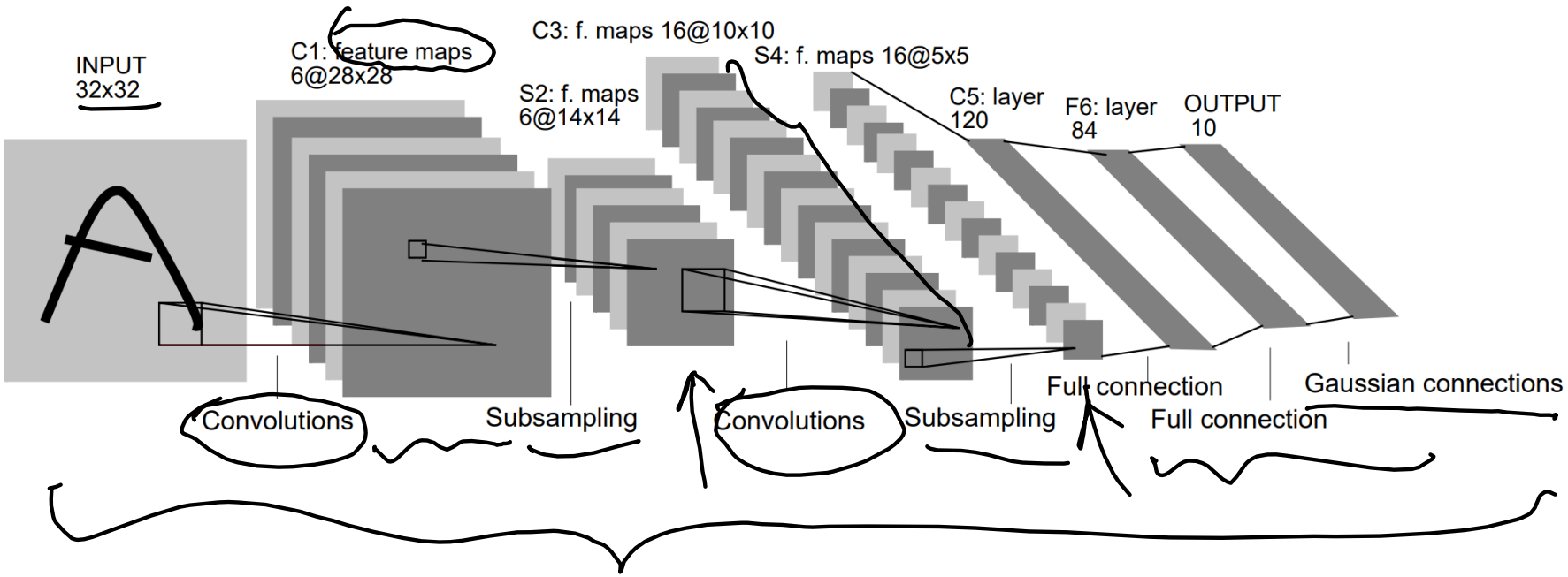
- Consider an image with its pixels as the data for neural network. If the neural network is fully connected, it cannot capture the 2D spatial correlation of nearby pixels. This is because in fully connected neural networks (or multilayer Perceptrons), we should reshape the input image to become a vector.
- Therefore, **Convolutional Neural Network (CNN)** was proposed to get the images as they are and not in reshaped form. In this way, the neural network can consider the 2D correlation of pixels.
- CNN is a neural network which uses convolution in place of general matrix multiplication in at least one of the its layers. A CNN might or might not have one or several fully connected layers in addition to convolutional layer(s).

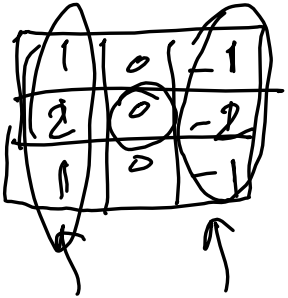


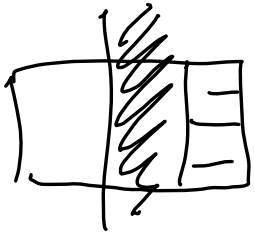
$$\hat{y} = \sigma_2\left(U_2^T \sigma_1\left(U_1^T x\right)\right)$$

LeNet

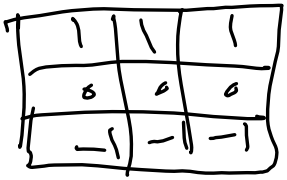
- CNN was initially proposed by Kunihiko Fukushima from Japan in 1980 [1]. That network was named **neocognitron** in that paper.
- The idea of CNN was adopted and slightly modified for the task of **handwritten digit recognition** by a team led by Yann LeCun in around 1989 [2].
- Later in 1998, it was improved by Yann Lecun and Yoshua Bengio *et al.* and named **LeNet** [3] after the name of LeCun. The paper [3] was the paper which proposed the Modified NIST (MNIST) dataset which is a handwritten digit dataset widely used as a benchmark in machine learning nowadays.
- The structure of LeNet is shown in this figure:

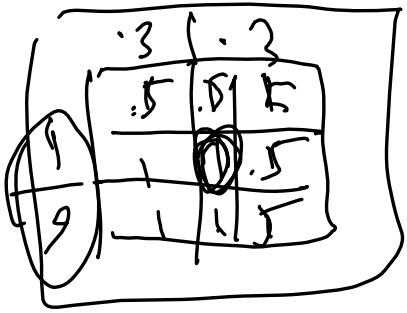


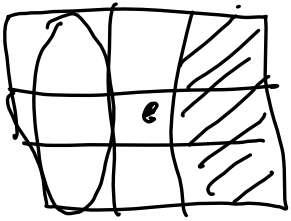

1 0 -1
2 0 -2
1 0 -1




bright


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


.3 .3
.5 .5 .5
1 .5
1 1 .5


e

The Stages in a Convolutional Layer

- Every convolutional layer has three stages:
  - convolution stage
  - detector stage (activation function)
  - pooling stage
- We talk about all these stages.

Convolution Stage

Convolution Stage

- Recall filtering in image processing. We have various kernel filters for different tasks such as edge detection, corner detection, smoothing, sharpening, etc.
- For example, these two filter kernels in Prewitt filter (1970) [4] detect the horizontal and vertical edges of the image:

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}, \quad \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}, \tag{1}$$

  respectively.
- These specific kernels are used for edge detection (feature extraction for edges). There are specific filters for specific tasks (feature extraction for specific tasks).
- But for a general machine learning task, such as classification or regression, we may need different filter kernels for feature extraction. The kernel differs for various machine learning tasks and also different dataset. Even for the same task, we may need different filter kernels for different datasets.
- Therefore, let us learn the best filter kernel values, using optimization, for our task and our dataset. This is the idea of convolutional layer. A convolutional layer considers the values of the filter kernel as learnable weights of that layer which can be trained by the backpropagation algorithm [5].

Convolution Stage

- The discrete convolution is defined as:

$$a[t] = (x * k)[t] = \sum_{a=-\infty}^{\infty} x[a]w[t-a]. \tag{2}$$

- We often use convolution in more than one axis at a time for a 2D image I. Let the kernel of 2D filter be denoted by K. The 2D convolution is:

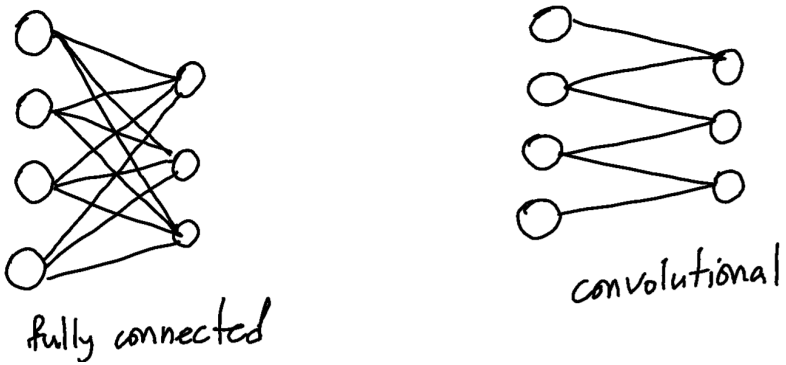
$$s[i,j] = (I * K)[i,j] = \sum_{m}\sum_{n} I[m,n]K[i-m, j-n], \tag{3}$$

  where the summation is over the number of rows and columns of the filter kernel. In other words, in a convolutional layer, we do not consider convolution in all range $(-\infty, \infty)$ but only in the range of kernel size. We can assume that the summations in convolution are in all range $(-\infty, \infty)$ but our signal is zero after the size of the filter kernel.
- Convolution flips one of the signals such as the kernel filter. Usually, we do not flip the filter kernel in the convolutional layer, as also done in image processing. This does not make any difference because if we flip the kernel, the network will learn the flipped weights of the kernel, and if we do not flip it, the network will learn the non-flipped weights of the kernel. The non-flipped convolution is **cross correlation** and is formulated as:

$$s[i,j] = (I * K)[i,j] = \sum_{m}\sum_{n} I[m,n]K[m-i, n-j]. \tag{4}$$

Convolution Stage

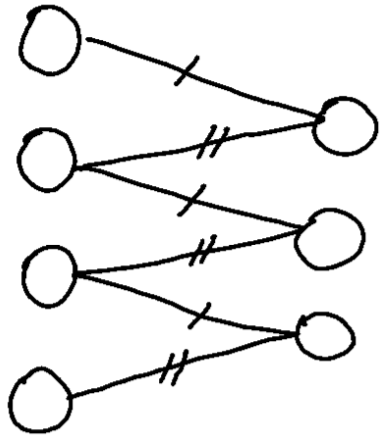
- Compare a dense (fully connected) layer and a convolutional layer:



- As seen in this figure, the convolutional layer is **sparse** and not dense like the fully connected layer. In a fully connected layer, every output neuron interacts with every input neuron. However, in a convolutional layer, there is a sparse connectivity and thus sparse learnable weights.
- This sparsity is beneficial for the "betting on sparsity principal" [6, 7], the Occam's razor [8], and also being low-rank (the weight matrix will probably have lower rank compared to a dense layer) [9].
- As obvious in the above figure, this sparsity occurs because the kernel is usually smaller than the input size. If the kernel size is the same as the input size, the layer becomes a dense (fully connected) layer.
- If the number of input and output neurons of a layer are m and n, respectively, the run-time, the required space, and the number of parameters of a fully connected layer is $O(m \times n)$. That is while, if the kernel size is k (where $k < m$), the run-time, the required space, and the number of parameters of a convolutional layer is $O(k \times n)$. Therefore, both time complexity and space complexity of a convolutional layer is better than a dense layer.

Convolution Stage

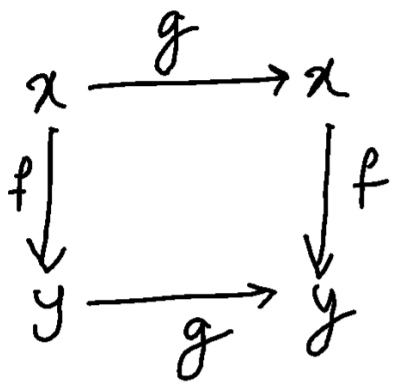
- There is another constraint in a convolutional layer and that is **parameter sharing**. the parameters (wights) of the kernels are shared. In other words, we sweep the same kernel over the neurons in a convolutional layer.
- In a traditional fully connected layer, every element of the weight matrix is multiplied by one element of the input, i.e., it is used once when computing the output of a layer.
- In a convolutional layer, every element of the kernel is used at every position of the input. Instead of learning a separate set of parameters for every position, we learn only one set.



Convolution Stage

- Another property of the convolutional layer is the equivalence property of the convolution operator with respect to translation.
- A function $f(x)$ is equivalent to a function $g(x)$ if:

$$f(g(x)) = g(f(x)). \tag{5}$$



- The convolution operator has equivalence property with respect to translation. Therefore, these two are equivalent:
  - Translate image/input and then apply convolution
  - Apply convolution and then translate image/input.
- Note that convolution is not equivalent with respect to some other transformations such as rotation and changes in scale.

Convolution Stage

- Important parameters in the convolutional stage:
  - **stride**: the step of jump of kernel in convolution
  - **width**: the width and height of kernel
  - number of **channels**: the depth of the kernel. Do not confuse with the number of channels of data/input of the filter kernel!
  - **padding**: we can apply padding to increase the input width and height. Some padding methods are **zero padding**, **mirror padding**, etc.
- Example of convolution with 1 channel of input:
  `https://i.stack.imgur.com/uEoXw.gif`
- Example of convolution with 3 channels of input and 2 channels W_1 and W_2:
  `https://codelabs.developers.google.com/codelabs/keras-flowers-convnets#4`

Detector Stage

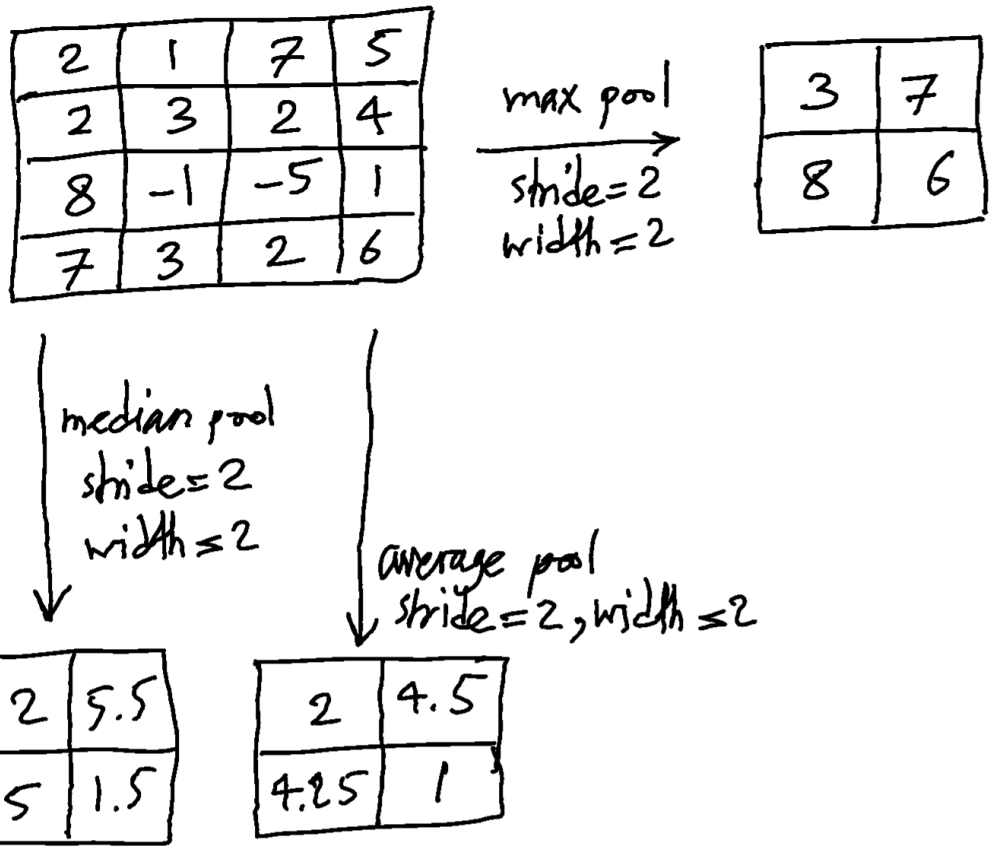
Detector Stage

- Detector stage is another name for the **activation function**. This is because they act like they detect the signal if the signal passes from a threshold.
- We already talked about the activation functions before in this course.

Pooling Stage

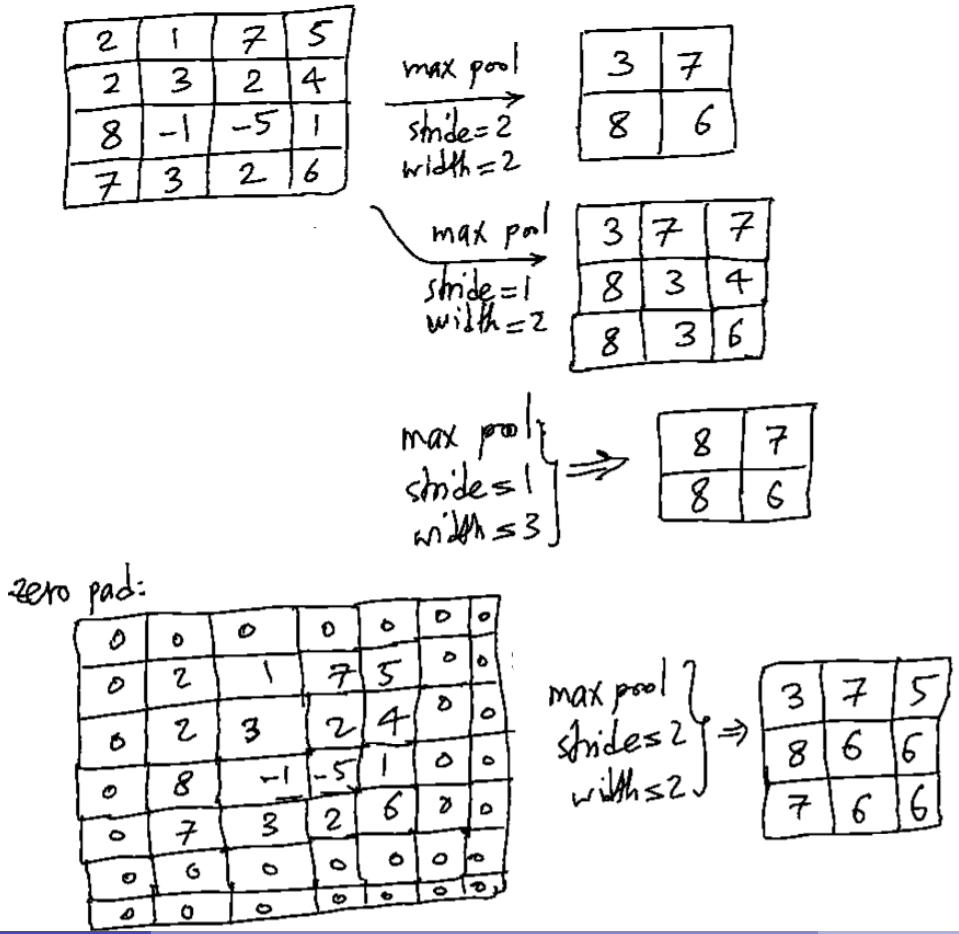
Pooling Stage

- In pooling stage, **pooling** is performed. Pooling provides a **summary statistics** from the signal.
- Some example pooling operators are:
  - Max pooling
  - Average pooling
  - Weighted average pooling
  - Median pooling
  - ℓ_2 norm pooling



Pooling Stage

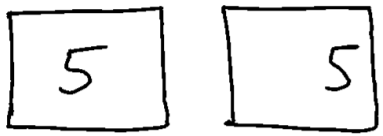
- **stride**: the step of jump for kernel of pooling. Based on stride, pooling may be **overlapping** or **non-overlapping**.
- **width**: width of kernel of pooling
- number of **channels**: the depth of the kernel of pooling which is the same as the number of channels in the input of the pooling stage.
- **padding**: we can apply padding to increase the input width and height. Some padding methods are **zero padding**, **mirror padding**, etc.



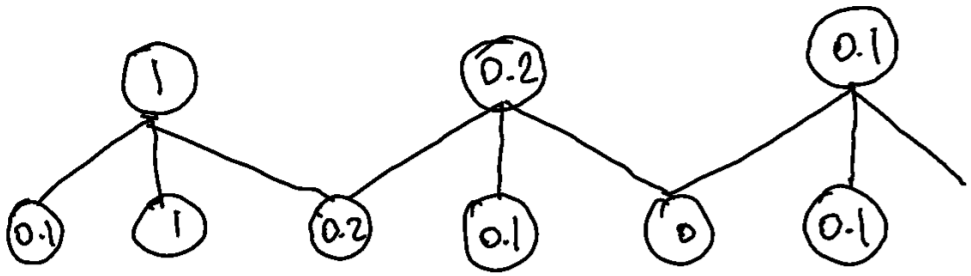
Pooling Stage

- Pooling makes the features almost invariant to **local translations**.
- For example, here, all four features of input
  of pooling stage have changes. However, only two features have changed after max pooling.

- In most cases, it is good to become invariant and robust to local translations but in some applications, it may be destructive. It is usually helpful because we usually
  want to know whether a specific feature exists in data but we do not care where exactly it is.

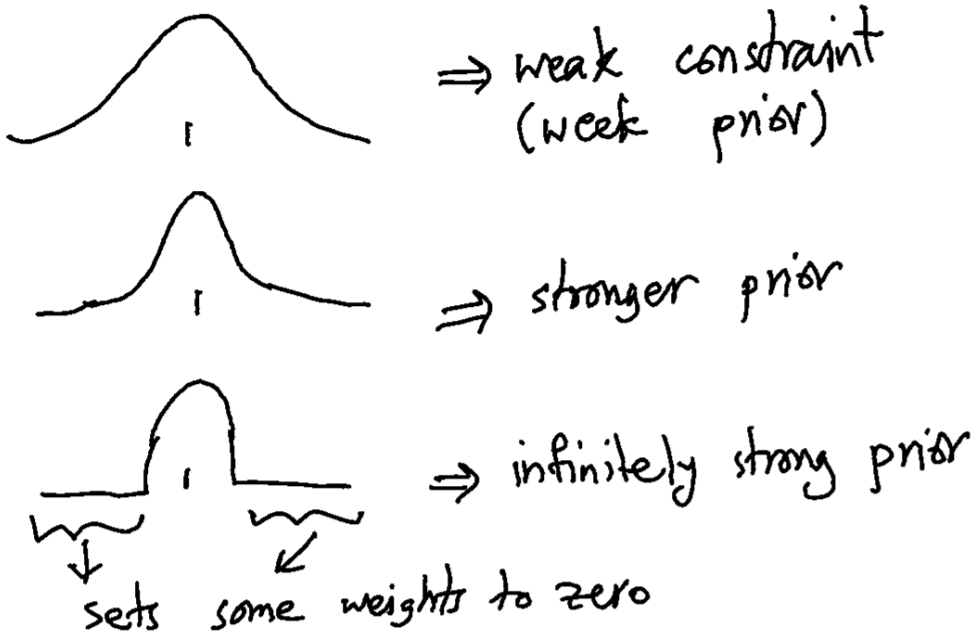


- Pooling also helps in **downsampling** data. This reduces the **representation size** and hence the **computational and the statistical burden** on the next layer.



Pooling Stage

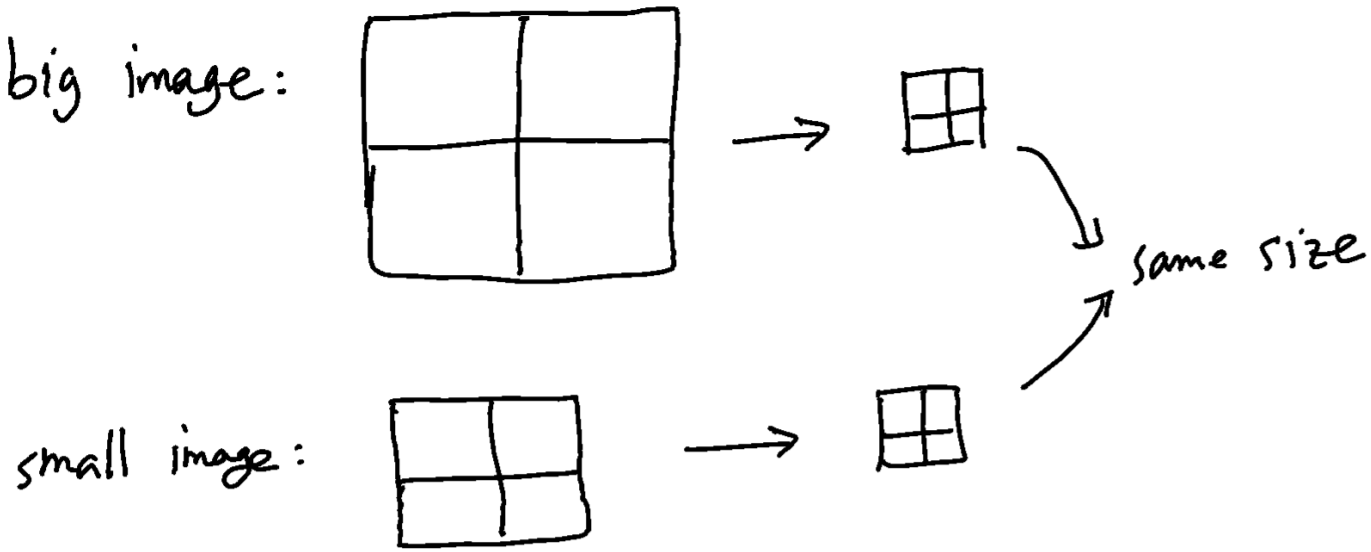
- We can see both convolution and pooling stages as **infinitely strong prior**.



- Again, this is beneficial for "betting on sparisity principal" [6, 7], the Occam's razor [8].

Pooling Stage

- We can dynamically pool the features together, rather than pooling the neighbor features together only. For example, by running a clustering algorithm (such as K-means) on locations of the interesting features, we can pool the features in the same cluster (2011) [10]. In this way, different sets of pooling regions will be used for different data inputs.
- We can rather learn a single pooling structure and then we apply that learned pooling structure to all data inputs (2012) [11].
- We can use pooling to handle inputs of varying sizes.



Batch Normalization

Batch Normalization

- **Batch normalization** was proposed in 2015 [12].
- It can be used in various network structures including fully connected and convolutional layers.
- It helps prevent **covariate shift** and hence helps generalization and avoiding overfitting.
- It also helps training to be faster.
- Assume the input of a layer is $\boldsymbol{x} = [x_1, x_2, \ldots, x_d]^\top$. So, the layer has d neurons.
- For a mini-batch, let the input of the layer be the set $\{\boldsymbol{x}_i\}_{i=1}^b$ where b is the batch size. Let x_{ij} denote the j-th dimension of $\boldsymbol{x}_i$, i.e., $\boldsymbol{x}_i = [x_{i1}, \ldots, x_{id}]^\top$.
- Batch normalization **standardizes** or applies **Z-score normalization** on every feature (every neuron's input) over the mini-batch:

$$\mathbb{R} \ni \mu_j := \frac{1}{b}\sum_{i=1}^{b} x_{ij}, \tag{6}$$

$$\mathbb{R} \ni \sigma_j^2 := \frac{1}{b}\sum_{i=1}^{b} (x_{ij} - \mu_j)^2, \tag{7}$$

$$x_{ij} \leftarrow \frac{x_{ij} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}, \tag{8}$$

where $\varepsilon > 0$ is a small positive number for stability and prevention of possible division by zero.

Initialization of Kernel (Layer) Weights

Initialization of Kernel (Layer) Weights

- Backpropagation is gradient descent and chain rule in derivatives. So, we need to initialize the weights, i.e., the optimization variables, for backpropagation. Two ways for initialization:
  - learn the wights by an unsupervised method. Example: using Principal Component Analysis (PCA) [13].
  - initialization by random features. It is related to random projection and Johnson-Lindenstrauss lemma [14].

Initialization of Kernel (Layer) Weights

- Method 1: learn the wights by an unsupervised method.
- Divide the input of the layer into patches. The patch size s can be the stride of the kernel for convolution.
- Shift the patches a little bit by some features or pixels.
- Collect all the patches and reshape them to become column vectors. Apply Principal Component Analysis (PCA) [13] on the patches. It gives you $s \times s$ eigenvector matrices. Make a matrix with the size of the input, where the $s \times s$ eigenvector matrices of PCA replace the patches in this big matrix. This matrix can be the initial kernel of convolution of that layer.
- Do this for all layers to find the initial kernels for every layer.

Initialization of Kernel (Layer) Weights

- Method 2: initialization by random features.
- It is related to random projection and Johnson-Lindenstrauss lemma [14].
- Set the initial values weights randomly by some stochastic distribution such as Gaussian or uniform distribution.
- The values should not be too large not to have gradient explosion. They all should not be too small not to have gradient vanishing.

Random Projection

- **Linear random projection** is projection of data points onto the column space of a projection matrix where the elements of projection matrix are i.i.d. random variables sampled from a distribution with zero mean and (possibly scaled) unit variance.
- In other words, random projection is a function $f : \mathbb{R}^d \to \mathbb{R}^p$, $f : \boldsymbol{x} \mapsto \boldsymbol{U}^\top \boldsymbol{x}$:

$$\mathbb{R}^p \ni f(\boldsymbol{x}) := \boldsymbol{U}^\top \boldsymbol{x} = \sum_{t=1}^{p} \boldsymbol{u}_t^\top \boldsymbol{x} = \sum_{j=1}^{d} \sum_{t=1}^{p} u_{jt}\, x_j, \tag{9}$$

  where $\boldsymbol{U} = [\boldsymbol{u}_1, \dots, \boldsymbol{u}_p] \in \mathbb{R}^{d \times p}$ is the random projection matrix, u_{jt} is the (j, t)-th element of $\boldsymbol{U}$, and x_j is the j-th element of $\boldsymbol{x} \in \mathbb{R}^d$.
- The elements of $\boldsymbol{U}$ are sampled from a distribution with zero mean and (possibly scaled) unit variance. For example, we can use the Gaussian distribution $u_{j,t} \sim \mathcal{N}(0, 1), \forall j, t$.
- Some example distributions to use for random projection are Gaussian [15] and Cauchy [16, 17] distributions.
- It is noteworthy that, in some papers, the projection is normalized:

$$f : \boldsymbol{x} \mapsto \frac{1}{\sqrt{p}} \boldsymbol{U}^\top \boldsymbol{x}. \tag{10}$$

- If we have a dataset of d-dimensional points with sample size n, we can stack the points in $\boldsymbol{X} := [\boldsymbol{x}_1, \dots, \boldsymbol{x}_n] \in \mathbb{R}^{d \times n}$. Eq. (9) or (10) is stated as:

$$\mathbb{R}^{p \times n} \ni f(\boldsymbol{X}) := \boldsymbol{U}^\top \boldsymbol{X} \quad \text{or} \quad \frac{1}{\sqrt{p}} \boldsymbol{U}^\top \boldsymbol{X}. \tag{11}$$

Random Projection

- In contrast to other linear dimensionality reduction methods which learn the projection matrix using training dataset for better data representation of class discrimination, random projection **does not learn the projection matrix** but **randomly samples it independent of data**.
- Surprisingly, projection of data onto the column space of this random matrix works very well although the projection matrix is completely random and independent of data.
- The Johnson-Lindenstrauss (JL) lemma [18] justifies why random projection works. For proof, see our tutorial paper "Johnson-Lindenstrauss lemma, linear and nonlinear random projections, random Fourier features, and random kitchen sinks: Tutorial and survey" [14].

Johnson-Lindenstrauss Lemma

Johnson-Lindenstrauss Lemma (1984) [18]:
For any set $\mathcal{X} := \{\boldsymbol{x}_i \in \mathbb{R}^d\}_{i=1}^n$, any integer n as the sample size, and any $0 < \epsilon < 1$ as error tolerance, let p be a positive integer satisfying:

$$p \geq \Omega(\frac{\ln(n)}{\epsilon^2 - \epsilon^3}), \tag{12}$$

where $\ln(.)$ is the natural logarithm and $\Omega(.)$ is the lower bound complexity [n.b. some works state Eq. (12) as:

$$p \geq \Omega(\epsilon^{-2} \ln(n)). \tag{13}$$

by ignoring ϵ^3 against ϵ^2 in the denominator because $\epsilon \in (0, 1)$]. There exists a linear map $f : \mathbb{R}^d \to \mathbb{R}^p$, $f : \boldsymbol{x} \mapsto \boldsymbol{U}^\top \boldsymbol{x}$, with projection matrix $\boldsymbol{U} = [\boldsymbol{u}_1, \ldots, \boldsymbol{u}_p] \in \mathbb{R}^{d \times p}$, such that we have:

$$(1 - \epsilon)\|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2^2 \leq \|f(\boldsymbol{x}_i) - f(\boldsymbol{x}_j)\|_2^2 \leq (1 + \epsilon)\|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2^2, \tag{14}$$

for all $\boldsymbol{x}_i, \boldsymbol{x}_j \in \mathcal{X}$, with probability of success as:

$$\mathbb{P}\Big((1 - \epsilon)\|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2^2 \leq \|f(\boldsymbol{x}_i) - f(\boldsymbol{x}_j)\|_2^2 \leq (1 + \epsilon)\|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2^2\Big) \geq 1 - \delta, \tag{15}$$

where $\delta := 2e^{-(\epsilon^2 - \epsilon^3)(p/4)}$ and the elements of the projection matrix are i.i.d. random variables with mean zero and (scaled) unit variance. An example is $u_{ij} \sim \mathcal{N}(0, 1/p) = (1/\sqrt{p})\mathcal{N}(0, 1)$ where u_{ij} denotes the (i, j)-th element of $\boldsymbol{U}$.

Acknowledgment

- Some slides of this slide deck are inspired by teachings of Prof. Ali Ghodsi at University of Waterloo, Department of Statistics.

References

[1] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological cybernetics*, vol. 36, no. 4, pp. 193–202, 1980.

[2] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[3] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[4] J. M. Prewitt *et al.*, "Object enhancement and extraction," *Picture processing and Psychopictorics*, vol. 10, no. 1, pp. 15–19, 1970.

[5] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[6] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning: Data Mining, Inference, and Prediction*, vol. 2.
Springer series in statistics, New York, NY, USA, 2009.

[7] R. Tibshirani, M. Wainwright, and T. Hastie, *Statistical learning with sparsity: the lasso and generalizations*.
Chapman and Hall/CRC, 2015.

References (cont.)

[8] P. Domingos, "The role of Occam's razor in knowledge discovery," *Data mining and knowledge discovery*, vol. 3, no. 4, pp. 409–425, 1999.

[9] B. Kulis, M. Sustik, and I. Dhillon, "Learning low-rank kernel matrices," in *Proceedings of the 23rd international conference on Machine learning*, pp. 505–512, 2006.

[10] Y.-L. Boureau, N. Le Roux, F. Bach, J. Ponce, and Y. LeCun, "Ask the locals: multi-way local pooling for image recognition," in *2011 International Conference on Computer Vision*, pp. 2651–2658, IEEE, 2011.

[11] Y. Jia, C. Huang, and T. Darrell, "Beyond spatial pyramids: Receptive field learning for pooled image features," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3370–3377, IEEE, 2012.

[12] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*, pp. 448–456, pmlr, 2015.

[13] B. Ghojogh and M. Crowley, "Unsupervised and supervised principal component analysis: Tutorial," *arXiv preprint arXiv:1906.03148*, 2019.

[14] B. Ghojogh, A. Ghodsi, F. Karray, and M. Crowley, "Johnson-lindenstrauss lemma, linear and nonlinear random projections, random fourier features, and random kitchen sinks: Tutorial and survey," *arXiv preprint arXiv:2108.04172*, 2021.

References (cont.)

[15] R. Giryes, G. Sapiro, and A. M. Bronstein, "Deep neural networks with random Gaussian weights: A universal classification strategy?," *IEEE Transactions on Signal Processing*, vol. 64, no. 13, pp. 3444–3457, 2016.

[16] P. Li, T. J. Hastie, and K. W. Church, "Nonlinear estimators and tail bounds for dimension reduction in ℓ_1 using Cauchy random projections," *Journal of Machine Learning Research*, vol. 8, no. Oct, pp. 2497–2532, 2007.

[17] A. B. Ramirez, G. R. Arce, D. Otero, J.-L. Paredes, and B. M. Sadler, "Reconstruction of sparse signals from ℓ_1 dimensionality-reduced Cauchy random projections," *IEEE Transactions on Signal Processing*, vol. 60, no. 11, pp. 5725–5737, 2012.

[18] W. B. Johnson and J. Lindenstrauss, "Extensions of Lipschitz mappings into a Hilbert space," *Contemporary mathematics*, vol. 26, 1984.