

Beyond MLPs: Convolutional Neural Networks

CS771: Introduction to Machine Learning

Piyush Rai

Plan today

- Deep neural networks beyond MLPs (Convolutional Neural Networks)
- Some other considerations in deep neural network training
 - Vanishing/exploding gradients
 - Initialization
 - Batch normalization



Recap

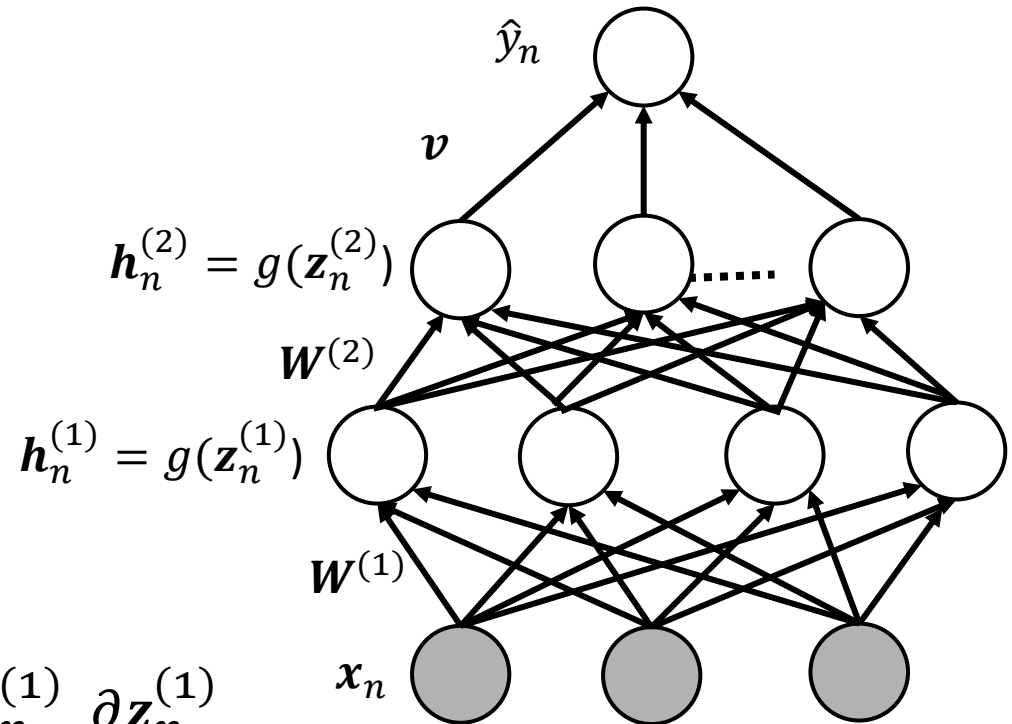
- Multi-layer Perceptrons (MLP)
- Backpropagation algorithm to learn the weights

All these gradients are basically product of Jacobians

$$\frac{\partial \ell_n}{\partial \mathbf{v}} = \ell'(y_n, \hat{y}_n) \mathbf{h}_n^{(2)}$$

$$\frac{\partial \ell_n}{\partial \mathbf{W}^{(2)}} = \ell'(y_n, \hat{y}_n) \frac{\partial \hat{y}_n}{\partial \mathbf{h}_n^{(2)}} \frac{\partial \mathbf{h}_n^{(2)}}{\partial \mathbf{z}_n^{(2)}} \frac{\partial \mathbf{z}_n^{(2)}}{\partial \mathbf{W}^{(2)}}$$

$$\frac{\partial \ell_n}{\partial \mathbf{W}^{(1)}} = \ell'(y_n, \hat{y}_n) \frac{\partial \hat{y}_n}{\partial \mathbf{h}_n^{(2)}} \frac{\partial \mathbf{h}_n^{(2)}}{\partial \mathbf{z}_n^{(2)}} \frac{\partial \mathbf{z}_n^{(2)}}{\partial \mathbf{h}_n^{(1)}} \frac{\partial \mathbf{h}_n^{(1)}}{\partial \mathbf{z}_n^{(1)}} \frac{\partial \mathbf{z}_n^{(1)}}{\partial \mathbf{W}^{(1)}}$$



- Backprop is based on reuse of previous computations to efficiently compute the gradients required for updating the network weights using (stochastic) GD

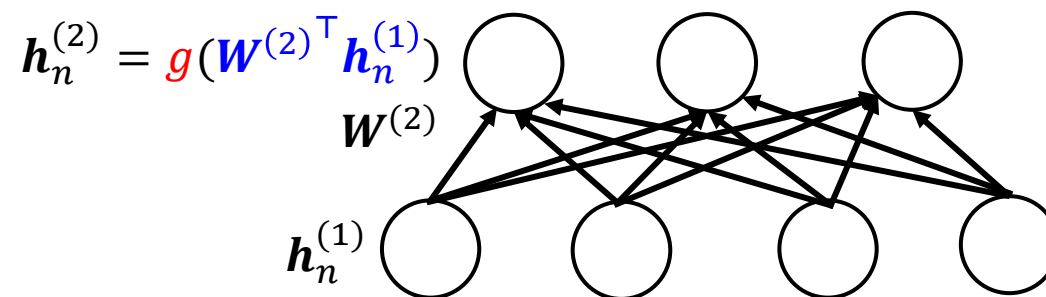
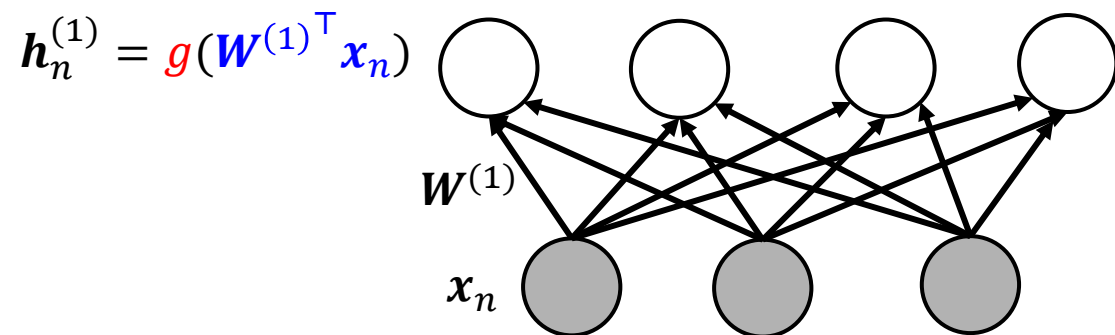


Limitations/Shortcomings of MLP

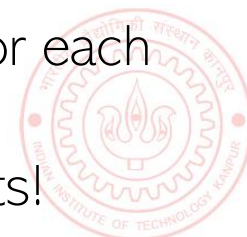
4

Projection using a “linear layer” +
element-wise nonlinearity applied
on these linear projections

- MLP uses fully connected layers defined by matrix multiplications + nonlinearity

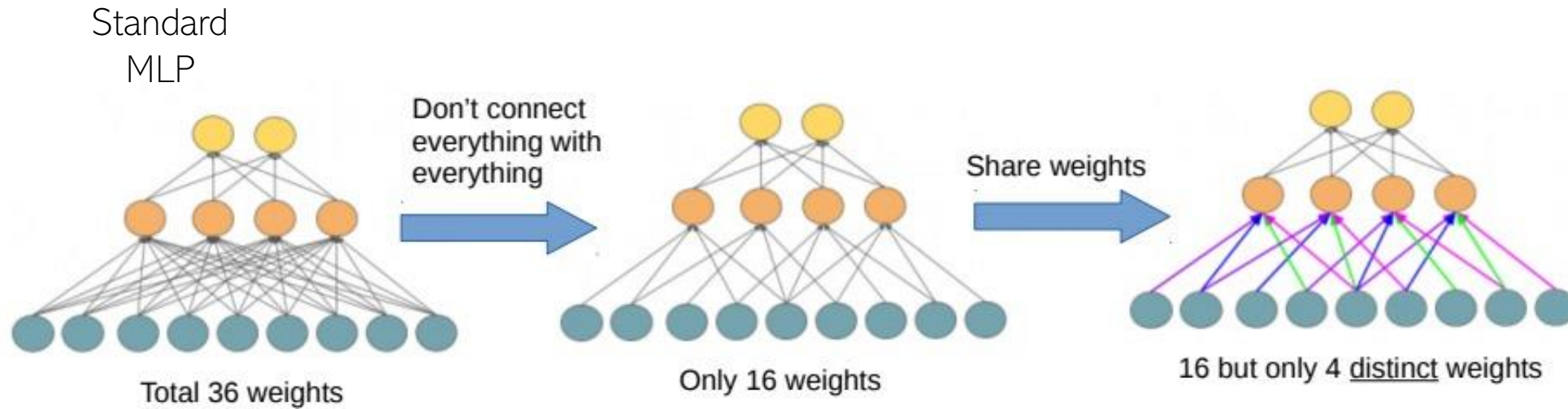


- MLP **ignores structure** (e.g., spatial/sequential) in the inputs
 - Not ideal for data such as images, text, etc. which are flattened as vectors when used with MLP
- Fully connected nature of MLP requires massive number of weights
 - Even a “smallish” 200x200x3 (3 channels – R,G,B) image will need 120,000 weights for each neuron in the first hidden layer (for K neurons, we will need 120,000 x K weights).
 - Recall that each layer is fully connected so each layer needs a massive number of weights!



Convolutional Neural Networks (CNN)

- CNNs use connections between layers that are different from MLPs in two key ways

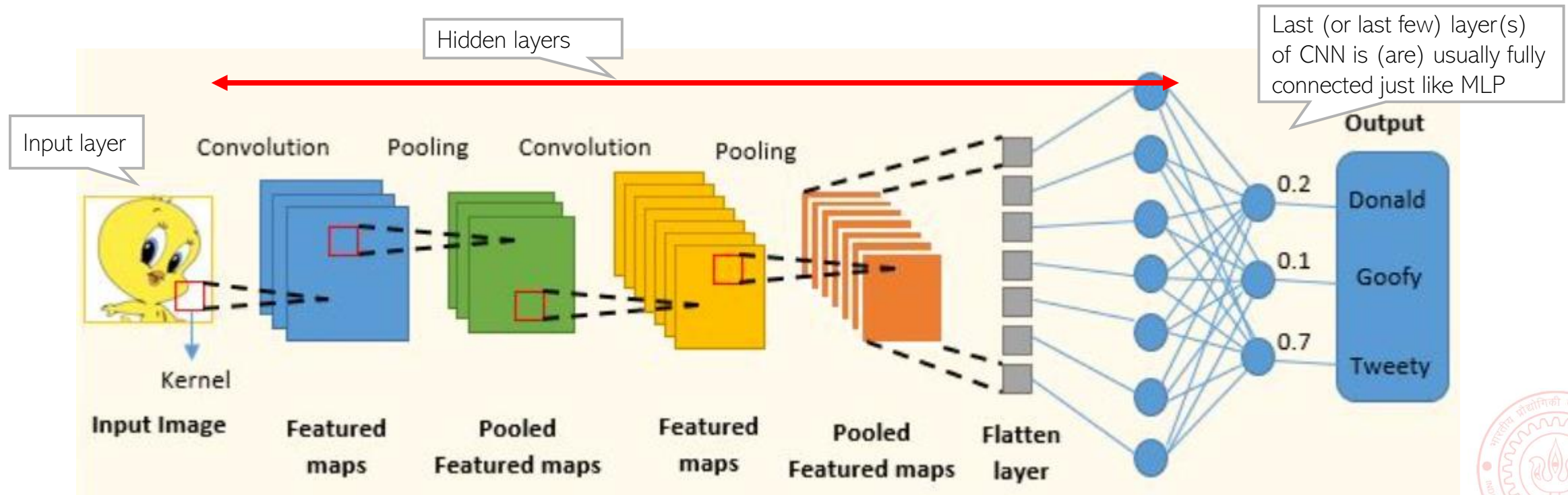


- Change 1: Each hidden layer node is connected only to a local patch in previous layer
- Change 2: Same set of weights used for each local patch (purple, blue, green, pink is one set of weights, and this same set of used for all patches)
- These changes help in
 - Substantial reduction on the number of weights to be learned
 - Learning the local structures within the inputs
 - Capturing local and global structure in the inputs by repeating the same across layers



Convolutional Neural Networks (CNN)

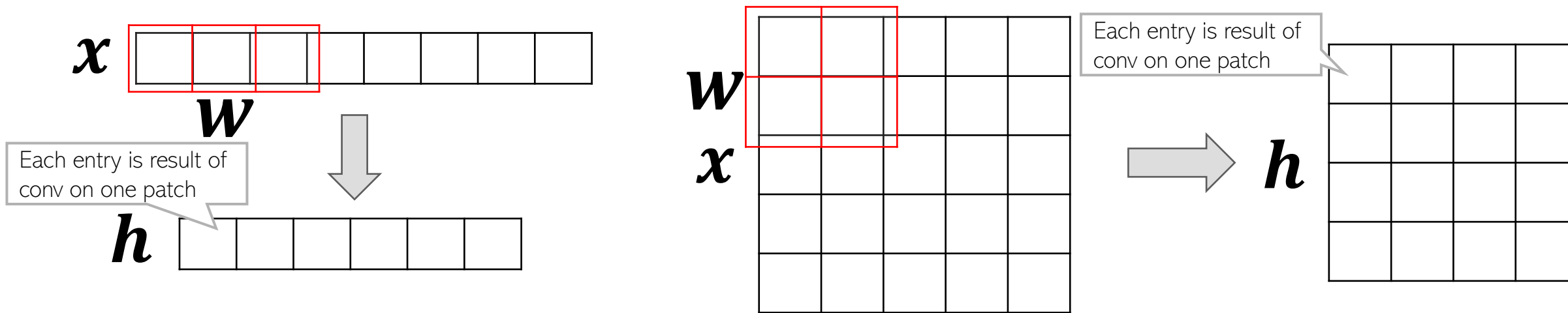
- CNN consists of a sequence of operations to transform an input to output
 - **Convolution** (a linear transformation but more “local” than the one in MLP)
 - Nonlinearity (e.g., sigmoid, ReLU, etc) after the convolution operation
 - **Pooling** (aggregates local features into global features and reduce representation size)



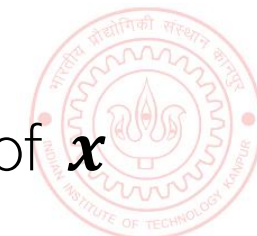
Convolution

Sometimes also called a “kernel”, though not the kernel we have seen in kernel methods ☺

- Convolution moves the same “filter”/“template” \mathbf{w} over different patches of input \mathbf{x}
 - Filter is like a set of weights (like in MLP) but only operate on local regions of \mathbf{x}
- Convolution = dot product of \mathbf{w} with different patches of the input \mathbf{x}

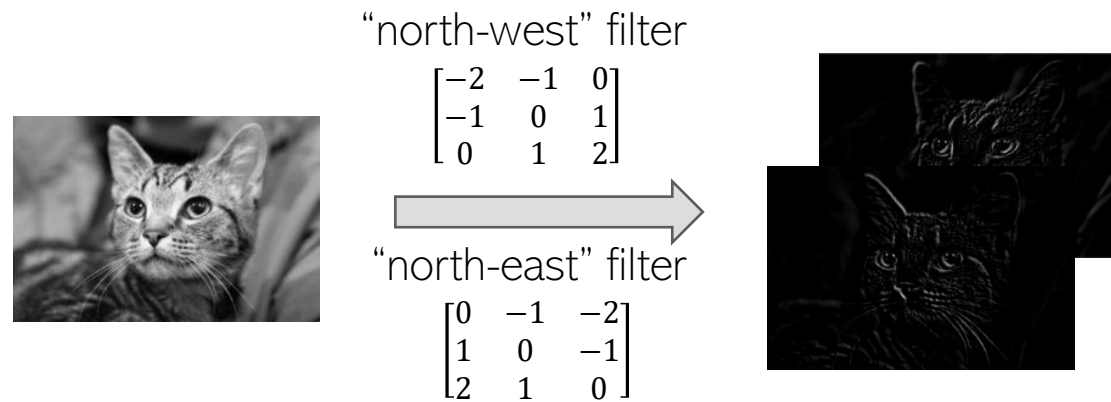


- Output \mathbf{h} of the convolution operation is also called a “feature map”
- If \mathbf{x} is $n_H \times n_W$, \mathbf{w} is $k_H \times k_W$ then \mathbf{h} is $(n_H - k_H + 1) \times (n_W - k_W + 1)$
- If we want \mathbf{h} to have larger size than then we do **zero-padding** at boundaries of \mathbf{x}

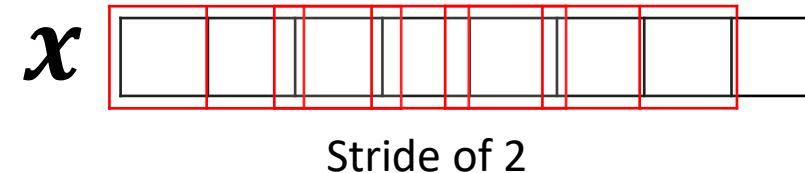
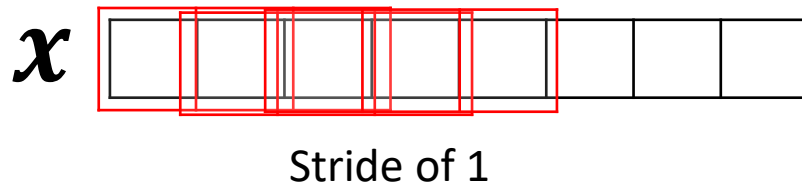


Convolution

- High “match” of a filter/kernel with a patch gives high values in the feature map
- In CNN, these weights/filters are **learnable**. Also, usually **multiple filters** are used
 - Each filter gives us a different feature map (K filters will give K feature maps)
 - Each map can be seen as representing a different type of feature in the inputs

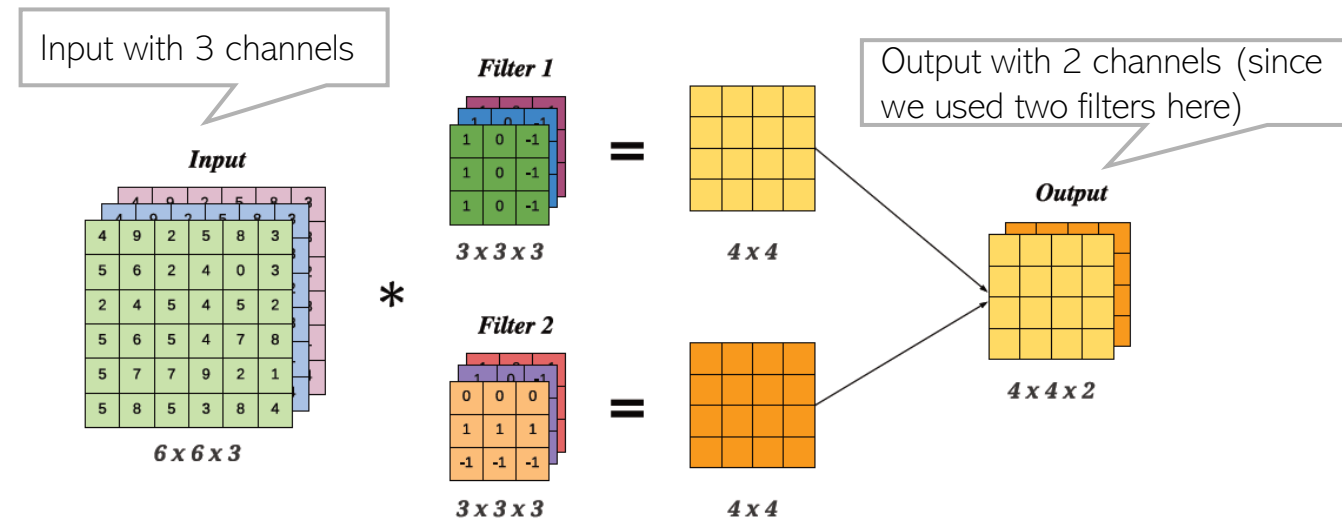
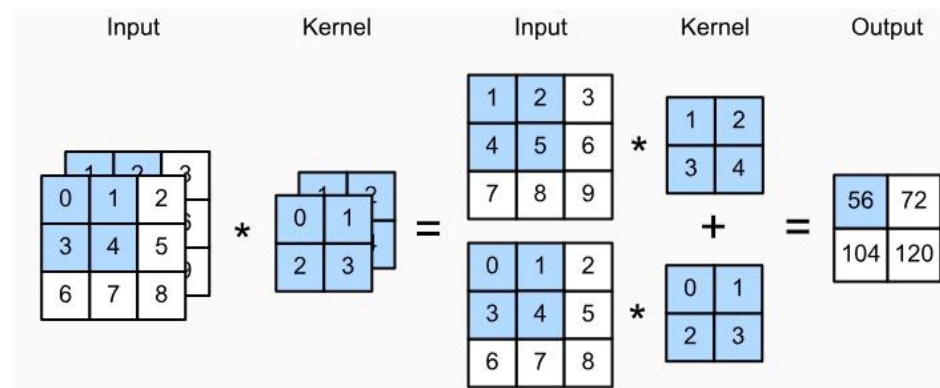


- When “moving” the filter across the input, the **stride size** can be one or more than one
 - Stride means how much the filter moves between successive convolutions



Multiple Input Channels

- If the input has multiple channels (e.g., images with R,G,B channels), then each filter/kernel also needs to have multiple channels, as shown below (left figure)
- We perform per-channel convolution followed by an aggregation (sum across channels)

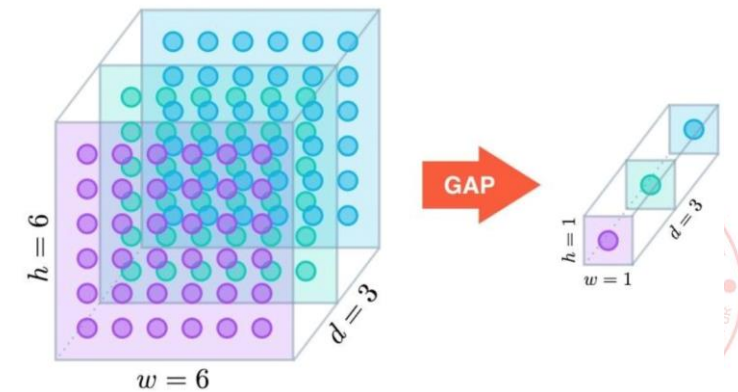
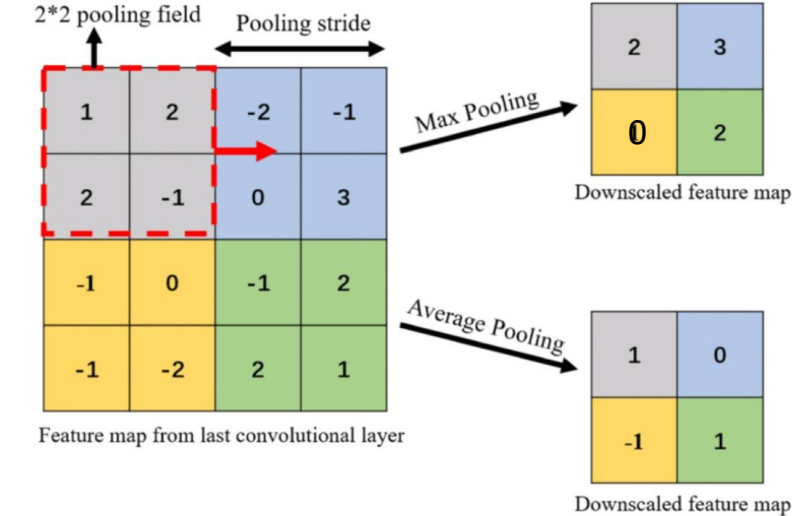


- Note that (right figure above) we typically also have multiple such filters (each with multiple channels) which will give us multiple such feature maps



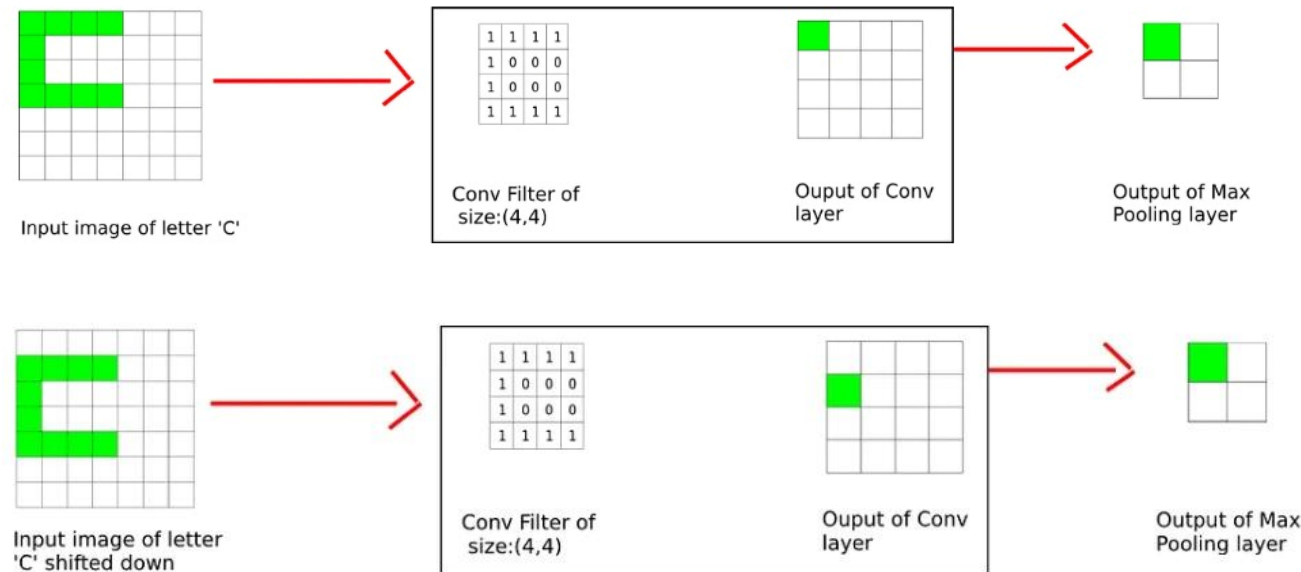
Pooling

- CNNs also consist of a **pooling operation** after each conv layer
- Pooling plays two important roles
 - Reducing the size of the feature maps
 - Combining local features to make global features
- Need to specify the size of group to pool, and pooling stride
- **Max pooling** and **average pooling** are popular pooling methods
- “**Global average pooling**” (GAP) is another option
 - Given feature map of size $h \times w \times d$ (e.g, if there are d channels), it averages all $h \times w$ locations to give a $1 \times d$ feature map
 - Reduces the number of features significantly and also allows handling feature maps of different heights and widths



CNNs have Translation Invariance!

- Even if the object of interest has shifted/translated, CNN don't face a problem (it will be detected regardless of its location in the image)
- The simple example below shows how (max) pooling helps with this

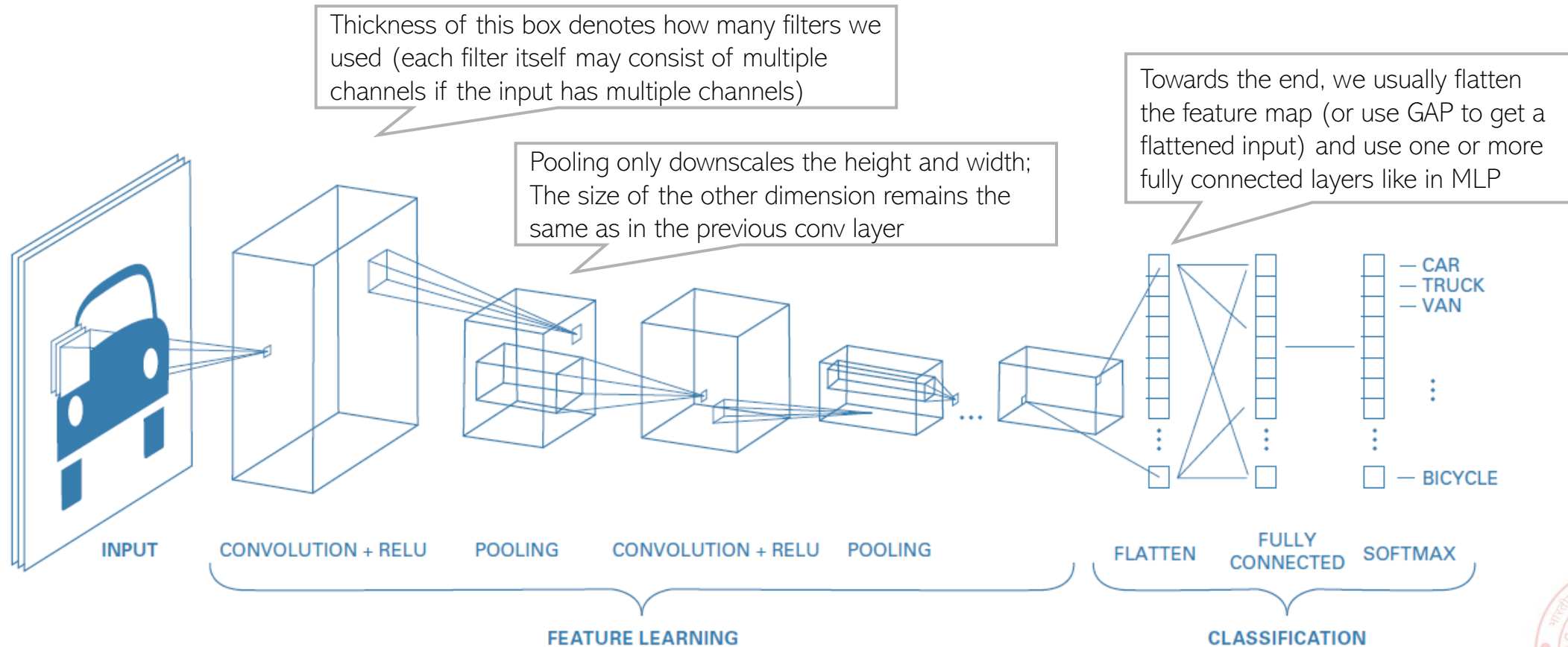


- CNNs use a combination of conv + pooling operations in several hidden layers so CNNs remain invariant to even more significant translations



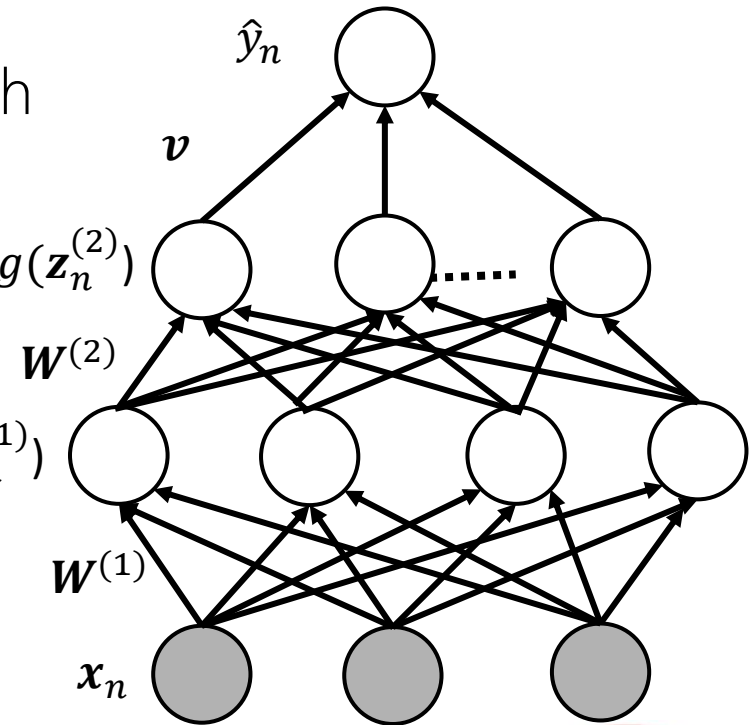
CNN: Summary of the overall architecture

- The overall structure of a CNN looks something like this



Problem of Exploding/Vanishing Gradients

- MLPs/CNNs have many hidden layers and gradients in each layer are a product of several Jacobians
- Result of these products depends on the eigenvalues of each of these Jacobians
 - If they are large (>1), gradients might blow up (explode)
 - If they are small (<1), gradients might vanish
- To prevent blow up, we can use **gradient clipping**
 - Simply cap the magnitude of the gradients!
- To prevent vanishing gradients, several options
 - Use **non-saturating activation functions** (recall that the gradient is a product of terms like $\frac{\partial h_n^{(i)}}{\partial z_n^{(i)}} = \text{diag} \left(g'(z_{n1}^{(i)}), \dots, g'(z_{nK_i}^{(i)}) \right)$, so the derivative g' doesn't become too small
 - Use other architectures such as **skip-connections** (will discuss later)



Training of DNNs: Some Important Aspects

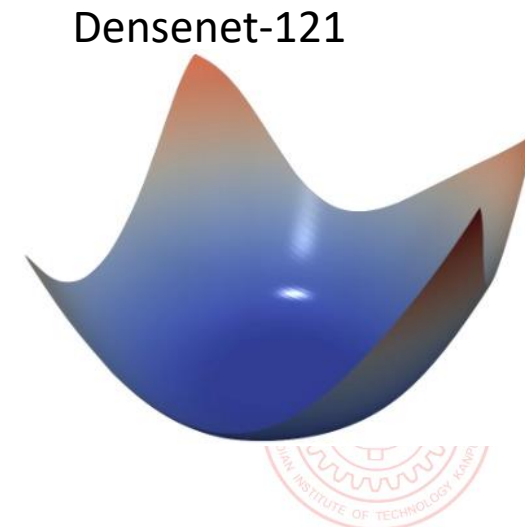
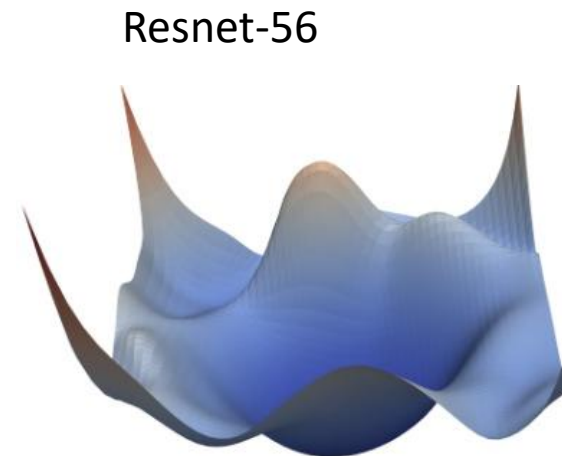
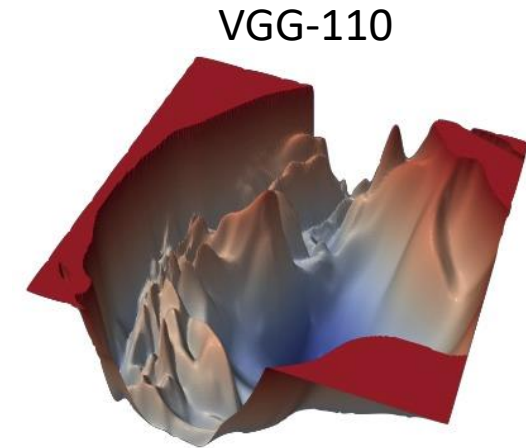
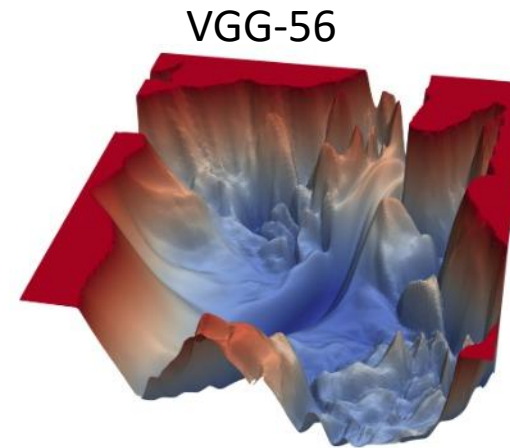
- Deep neural net training can be hard due to non-convex loss functions
- Several ways to address this, e.g.,
 - Good choice of learning rate of (S)GD
 - We have already seen this
 - Good initialization of parameters, e.g., initialize each weight, say w_{ij} , randomly as

$$w_{ij} \sim \mathcal{N}(0, \sigma^2) \quad \text{or} \quad w_{ij} \sim \text{Uniform}(-a, a)$$

Xavier/Gloret initialization,
LeCun init, He init, etc

and set the “spread” of these distribution
as inversely proportional to $n_{\text{in}} + n_{\text{out}}$

- Careful design of the network architecture, e.g.,
 - Networks with “skip connections” (will see later) which lead to less non-convex (more smooth) loss surfaces (figures on the right)
- Vanishing/exploding gradients (already saw)



Normalization Layer

Note: Batch-norm assumes sufficiently large mini-batch \mathcal{B} to work well. There are variants such as “layer normalization” and “instance normalization” that don’t require a mini-batch can be computed using a single training example

Batch normalization is used in MLP, CNN, and various other architectures

15



- Each hidden layer is a nonlinear transformation of the previous layer’s inputs
- To prevent distribution drift in activations’ distribution, we often “standardize” each layer
- Standardize = activation $h_{nk}^{(\ell)}$ should have zero mean and unit variance across all n
- It is achieved by inserting a “batch normalization” layer after each hidden layer
- To do so, during training, (omitting layer number ℓ) we replace each h_n by \tilde{h}_n

γ and β are trainable batch-norm parameters

$$\tilde{h}_n = \gamma \odot \hat{h}_n + \beta$$

$$\hat{h}_n = \frac{h_n - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

We compute $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}^2$ using the data from the current minibatch of examples \mathcal{B} (thus the name “batch norm”)

$$\mu_{\mathcal{B}} = \frac{1}{|\mathcal{B}|} \sum_{h \in \mathcal{B}} h$$

$$\sigma_{\mathcal{B}}^2 = \frac{1}{|\mathcal{B}|} \sum_{h \in \mathcal{B}} (h - \mu_{\mathcal{B}})^2$$

- After training, we store γ and β + the statistics μ and σ^2 computed on the whole training data, and use these values to apply batch-norm on each test input

