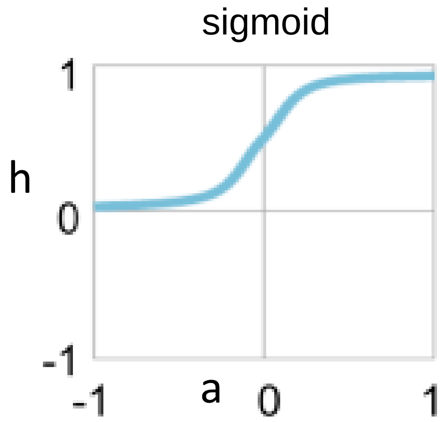
Deep Neural Networks (contd)

CS771: Introduction to Machine Learning

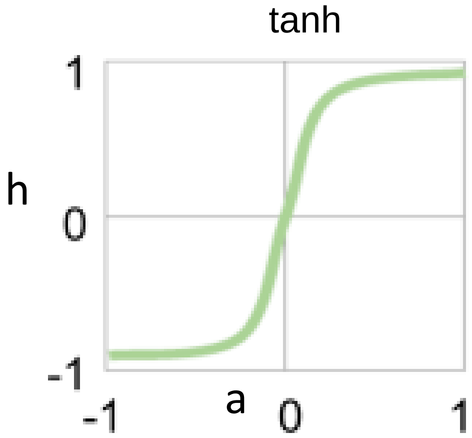
Plan

- Some more illustrations of an MLP's behavior
- Some important aspects of neural net training
- Deep neural networks for "structured" inputs (e.g., images)
  - Convolutional Neural Networks (today)
  - Sequence data models such as recurrent neural nets and transformers (next class)

Activation Functions: Some Common Choices



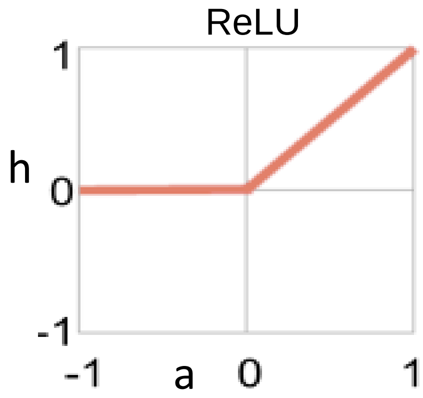
For sigmoid as well as tanh, gradients saturate (become close to zero as the function tends to its extreme values)



Preferred more than sigmoid. Helps keep the mean of the next layer's inputs close to zero (with sigmoid, it is close to 0.5)

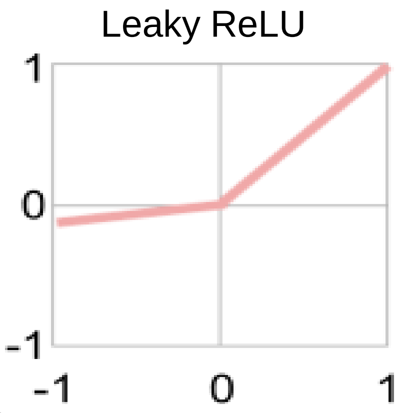
Sigmoid: $h = \sigma(a) = \frac{1}{1+\exp(-a)}$

tanh (tan hyperbolic): $h = \frac{\exp(a)-\exp(-a)}{\exp(a)+\exp(-a)} = 2\sigma(2a) - 1$



ReLU and Leaky ReLU are among the most popular ones (also efficient to compute)

Helps fix the dead neuron problem of ReLU when a is a negative number



$y = \boldsymbol{v}^\top(g(\boldsymbol{W}^\top \boldsymbol{x}))$
$= \boldsymbol{v}^\top \boldsymbol{W}^\top \boldsymbol{x}$

Still linear

Imp: Without nonlinear activation, a deep neural network is equivalent to a linear model no matter how many layers we use

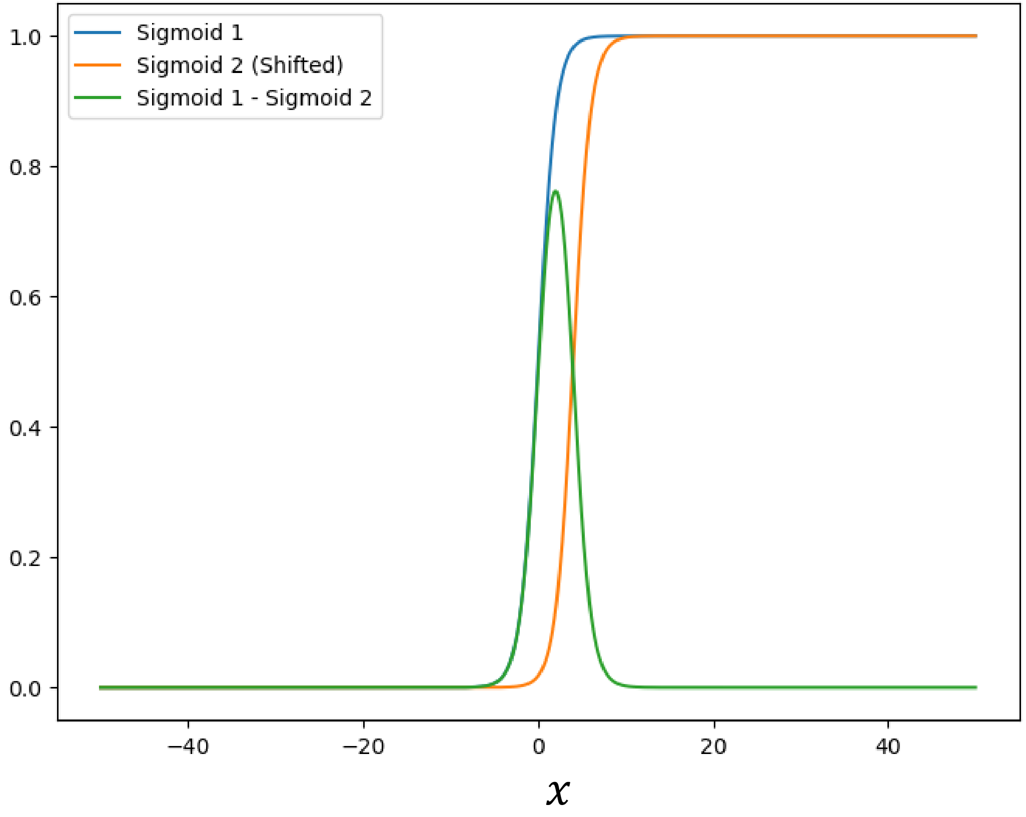
Most activation functions are monotonic but there exist some non-monotonic activation functions as well (e.g., Swish: $a \times \sigma(\beta a)$)

ReLU (Rectified Linear Unit): $h = \max(0, a)$

Leaky ReLU: $h = \max(\beta a, a)$
where β is a small postive number

Superposition of two linear models = Nonlinear model

$p(y = \text{red}|x)$



Two sigmoids (blue and orange) can be combined via a shift and a subtraction operation to result in a nonlinear separation boundary

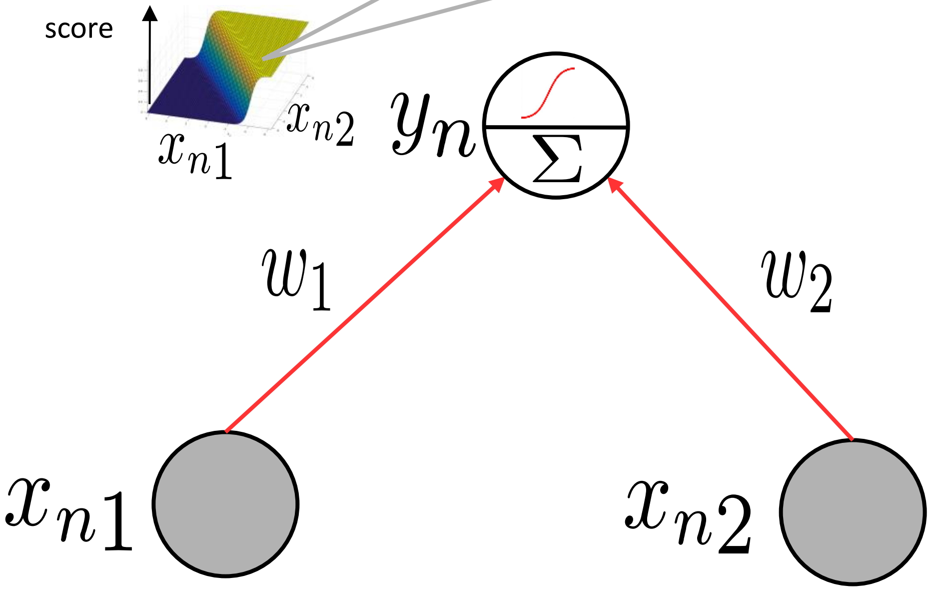
Likewise, more than two sigmoids can be combined to learn even more sophisticated separation boundaries

Nonlinear separation boundary

MLP Can Learn Any Nonlinear Function

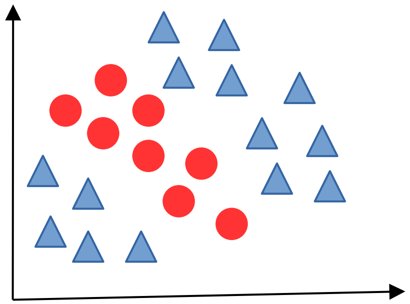
- An MLP can be seen as a composition of multiple linear models combined nonlinearly

Score monotonically increases. One-sided increase (not ideal for learning nonlinear decision boundaries). Just a linear model



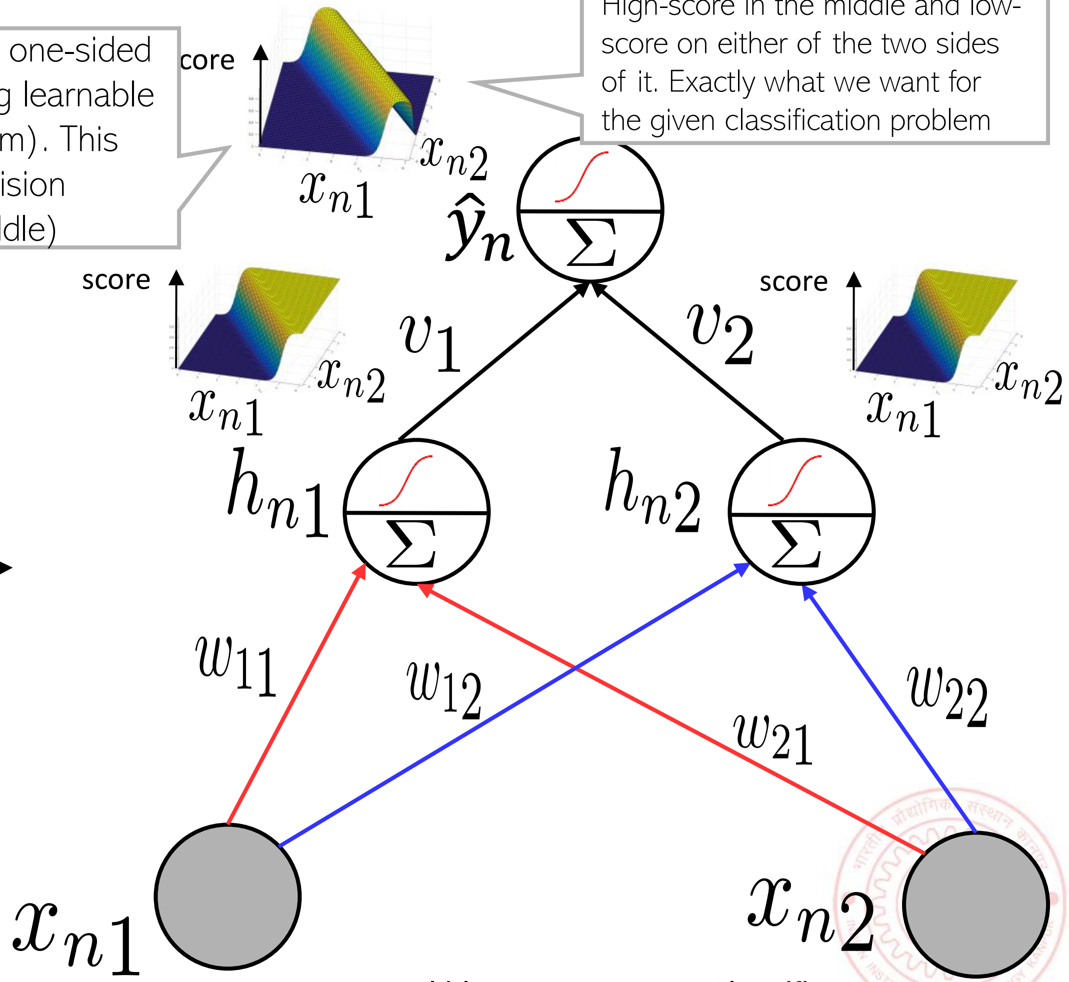
Standard Single “Perceptron” Classifier (no hidden units)

Obtained by composing the two one-sided increasing score functions (using learnable weights $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ to “add“ them). This can now learn the nonlinear decision boundary (high score in the middle)



A nonlinear classification problem

High-score in the middle and low-score on either of the two sides of it. Exactly what we want for the given classification problem



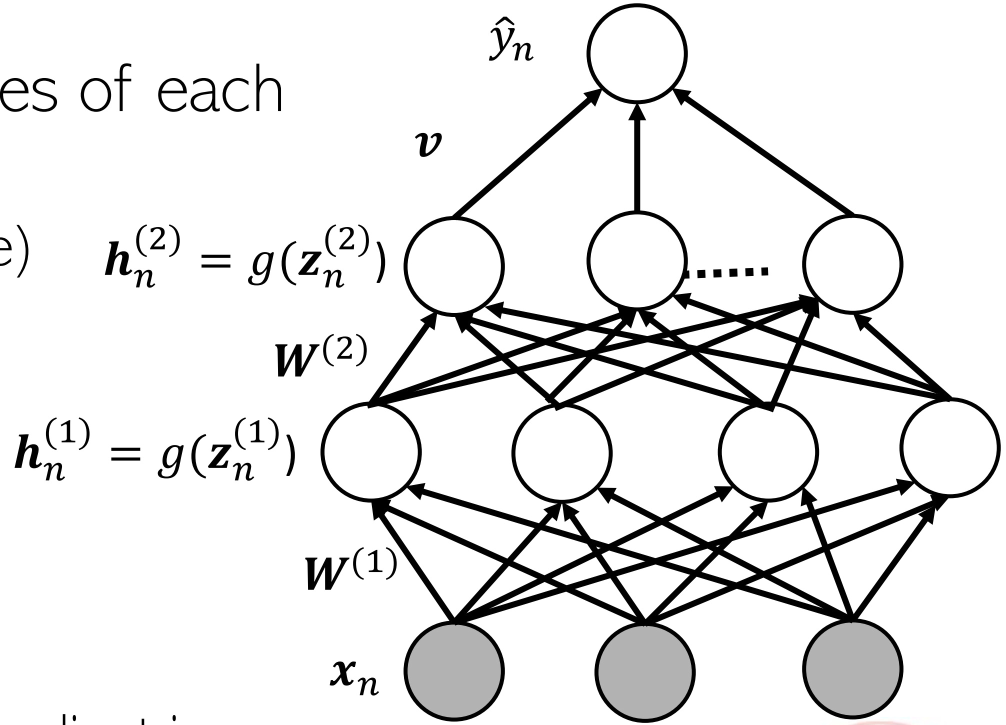
A Multi-layer Perceptron Classifier
(one hidden layer with 2 units)
Capable of learning nonlinear boundaries

A single hidden layer MLP with sufficiently large number of hidden units can approximate any function (Hornik, 1991)

Some Aspects of Neural Net Training

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 $\frac{\partial \boldsymbol{h}_n^{(i)}}{\partial \boldsymbol{z}_n^{(i)}} = \text{diag}\left(g'\left(z_{n1}^{(i)}\right), \ldots, g'\left(z_{nK_i}^{(i)}\right)\right)$, so the derivative $\boldsymbol{g'}$ 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 $\boldsymbol{w_{ij}}$, 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)

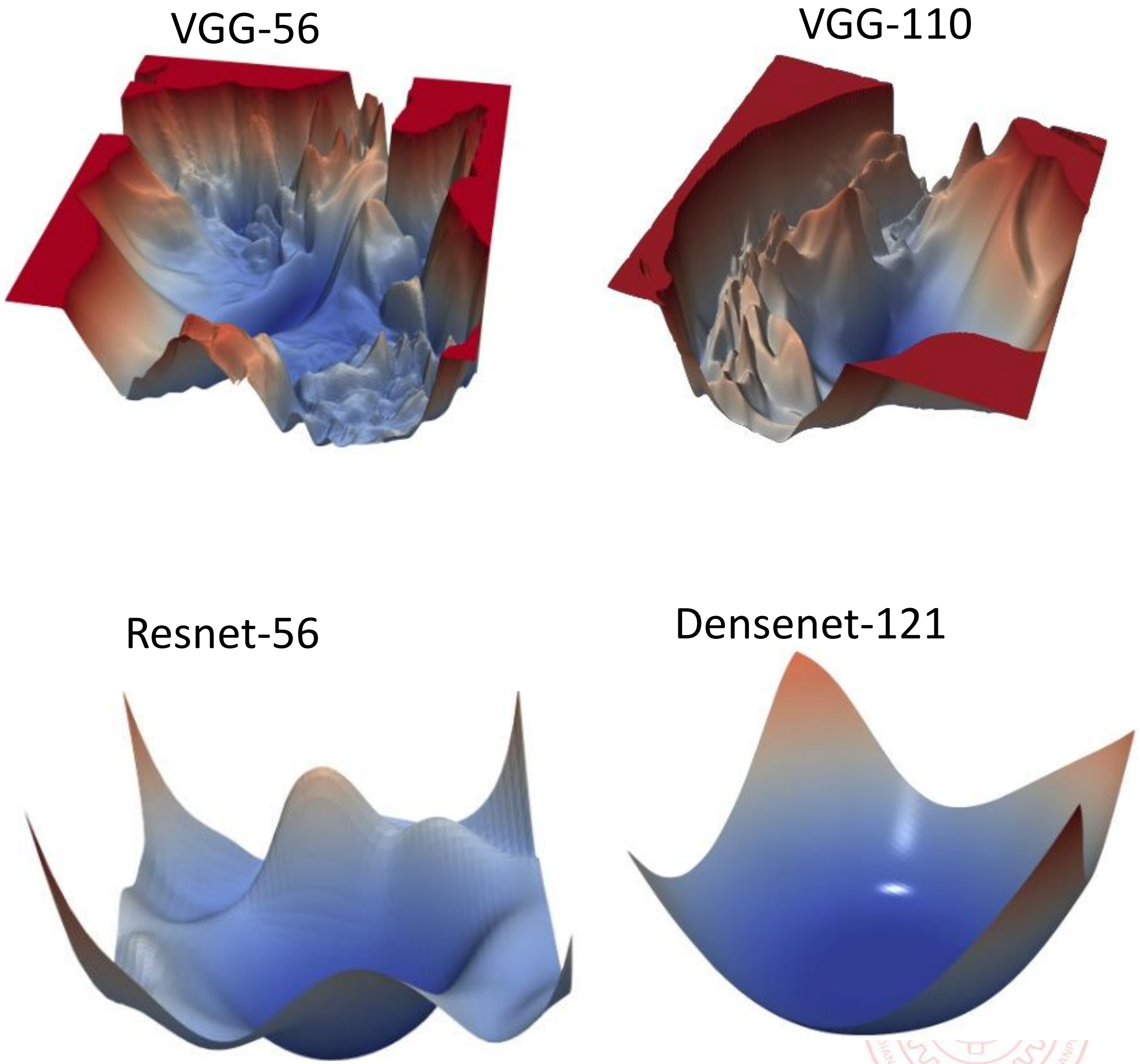


Fig credit: https://www.cs.umd.edu/~tomg/projects/landscapes/

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

- 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 $\boldsymbol{h}_n$ by $\tilde{\boldsymbol{h}}_n$

We compute $\boldsymbol{\mu}_{\mathcal{B}}$ and $\boldsymbol{\sigma}_{\mathcal{B}}^2$ using the data from the current minibatch of examples $\mathcal{B}$ (thus the name "batch norm")

$$\boldsymbol{\mu}_{\mathcal{B}} = \frac{1}{|\mathcal{B}|}\sum_{\boldsymbol{h}\in\mathcal{B}} \boldsymbol{h} \qquad \boldsymbol{\sigma}_{\mathcal{B}}^2 = \frac{1}{|\mathcal{B}|}\sum_{\boldsymbol{h}\in\mathcal{B}} (\boldsymbol{h} - \boldsymbol{\mu}_{\mathcal{B}})^2$$

$$\hat{\boldsymbol{h}}_n = \frac{\boldsymbol{h}_n - \boldsymbol{\mu}_{\mathcal{B}}}{\sqrt{\boldsymbol{\sigma}_{\mathcal{B}}^2 + \epsilon}} \qquad \tilde{\boldsymbol{h}}_n = \boldsymbol{\gamma} \odot \hat{\boldsymbol{h}}_n + \boldsymbol{\beta}$$

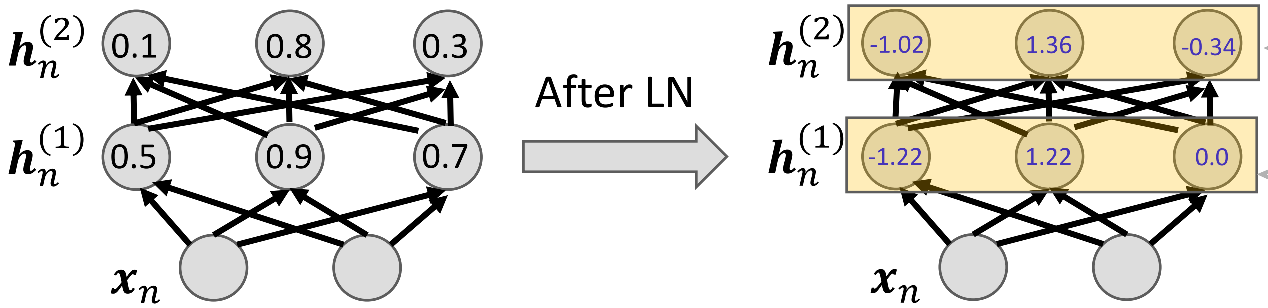
$\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are trainable batch-norm parameters

- After training, we store $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ + the statistics $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$ computed on the whole training data, and use these values to apply batch-norm on each test input

Layer Normalization

- Normalization helps improve training and performance overall
- Unlike batch normalization (BN), which we already saw, layer normalization (LN) normalizes each $\boldsymbol{h}_n$ across its dimensions (not across all minibatch examples)
  - Often used for sequence data models (will see later) where BN is difficult to apply
  - Also useful when batch sizes are small where BN statistics (mean/var) aren't reliable
- For an MLP, the LN operation would look like this

After LN operation, we apply another transformation defined by another set of learnable weights (just like we did in BN using γ and β)



$\boldsymbol{h}_n^{(2)}$ has zero mean and unit std-dev along its dimensions

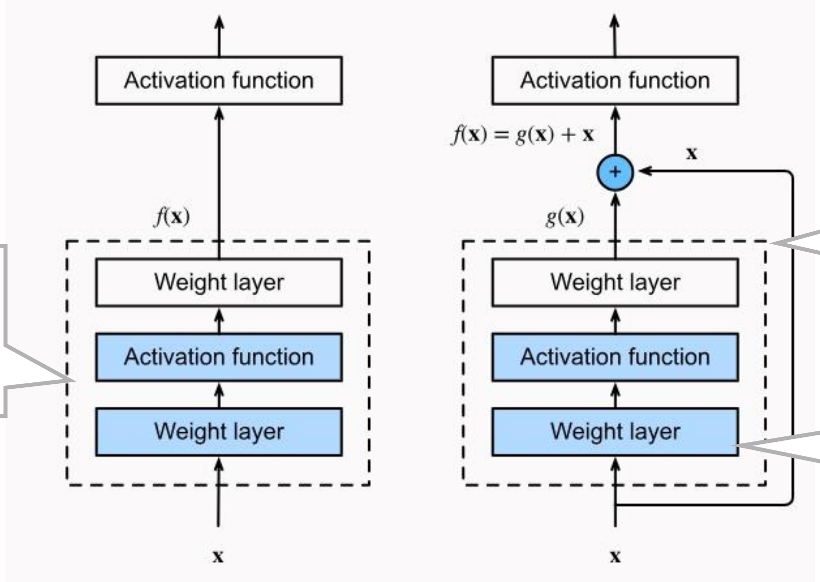
$\boldsymbol{h}_n^{(1)}$ has zero mean and unit std-dev along its dimensions

Residual/Skip Connections

- Many modern deep nets contain a very large number of layers
- In general, just stacking lots of layer doesn't necessarily help a deep learning model
  - Vanishing/exploding gradient may make learning difficult
- Skip connections or "residual connections" help if we want very deep networks
  - This idea was popularized by "Residual Networks"* (ResNets) which can have hundreds of layers
- Basic idea: Don't force a layer to learn everything about a mapping

May need to perform an additional projection/adjustment to that the sizes of x and $g(x)$ match

These layers trying to learn some function $f(x)$



Added a "residual branch" or "short-cut" connection to connect x to the residual output $g(x)$ of these layers

Reducing their burden by just asking them to learn the "residual" $g(x) = f(x) - x$

*Deep Residual Learning for Image Recognition (He et al, 2015)

Pic source: https://www.d2l.ai/index.html

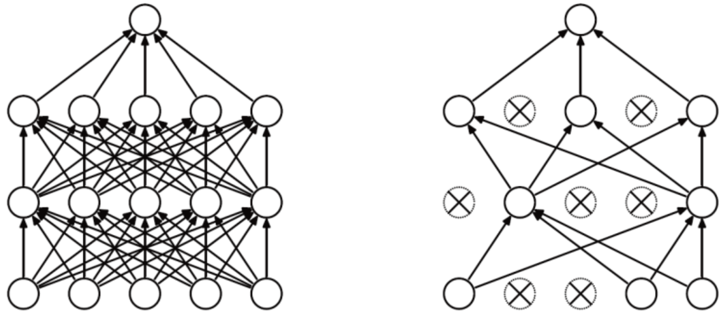
Dropout Layer

- Deep neural networks can overfit when trained on small datasets

- Dropout is a method to regularize without using an explicit regularizer

- In every update of the network, drop neuron $\boldsymbol{i}$ in layer $\boldsymbol{\ell}$ with probability $\boldsymbol{p}$

$$\epsilon_i^{(\ell)} \sim \mathrm{Bernoulli}(1-p)$$



- If $\epsilon_i^{(\ell)} = \mathbf{0}$, set all outgoing weights $\boldsymbol{w}_{ij}^{(\ell)}$ from neuron $\boldsymbol{i}$ to 0

- Each update of weights will change a different subset of weights
  - In doing so, we are making individual neurons more self-reliant and less dependent on others

- At test time, no dropout is used. After training is complete, we multiply each weight by the keep probability $\mathbf{1}-\boldsymbol{p}$ and use these weights for predictions

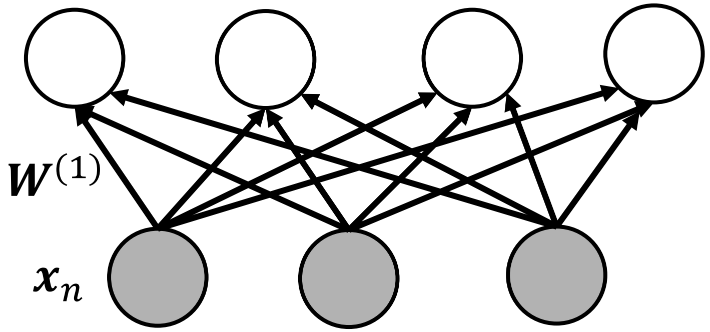
Neural Networks beyond MLPs

Limitations/Shortcomings of MLP

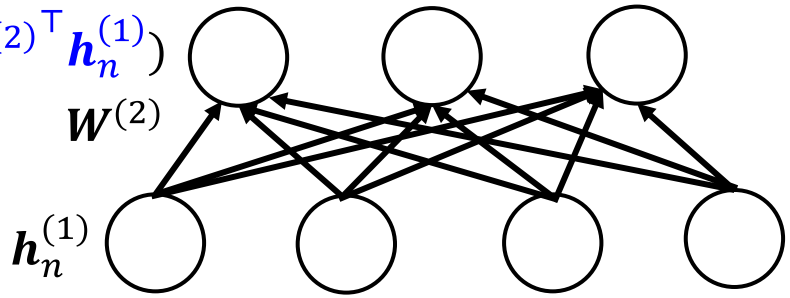
Projection using a "linear layer" + element-wise nonlinearity applied on these linear projections

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

$\boldsymbol{h}_n^{(1)} = g(\boldsymbol{W}^{(1)^\top} \boldsymbol{x}_n)$



$\boldsymbol{h}_n^{(2)} = g(\boldsymbol{W}^{(2)^\top} \boldsymbol{h}_n^{(1)})$

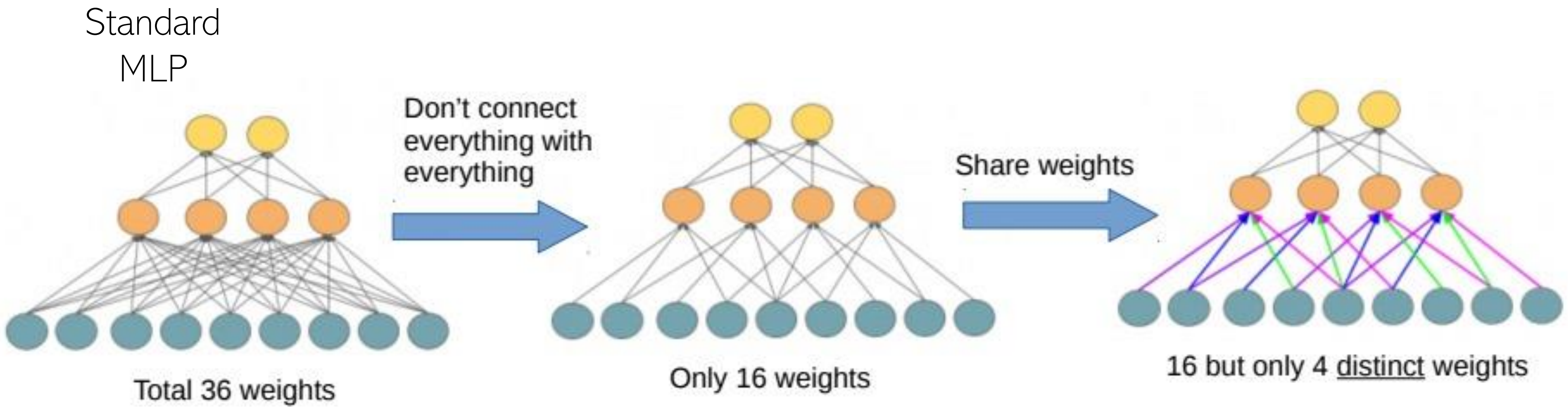


- MLP ignores structure (e.g., spatial/sequential) in the inputs
  - Not ideal for data such as images, text, etc. which are flattened as vectors when used with MLP

- 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)

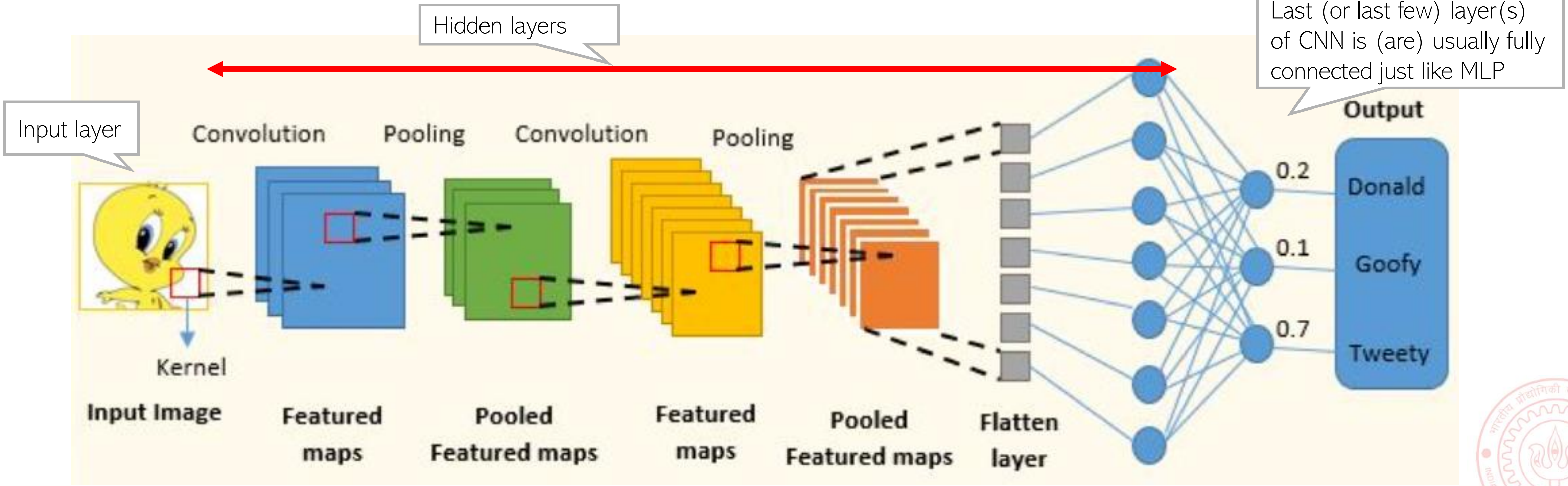
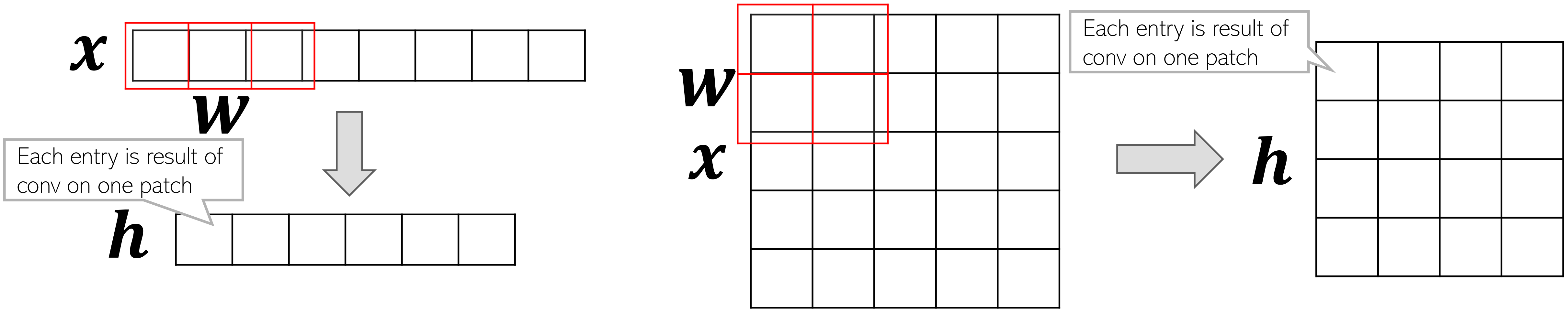


Figure credit: https://www.analyticsvidhya.com/blog/2022/01/convolutional-neural-network-an-overview/

Convolution

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

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



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

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

“north-west” filter

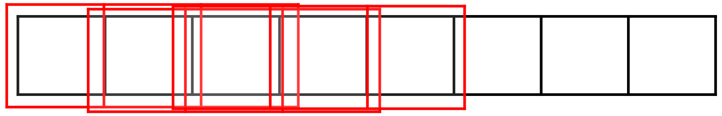
$$\begin{bmatrix} -2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix}$$

“north-east” filter

$$\begin{bmatrix} 0 & -1 & -2 \\ 1 & 0 & -1 \\ 2 & 1 & 0 \end{bmatrix}$$

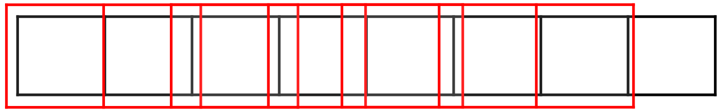
- When “moving” the filter across the input, the stride size can be one or more than one
  - Stride means how much the filter moves between successive convolutions

$\boldsymbol{x}$



Stride of 1

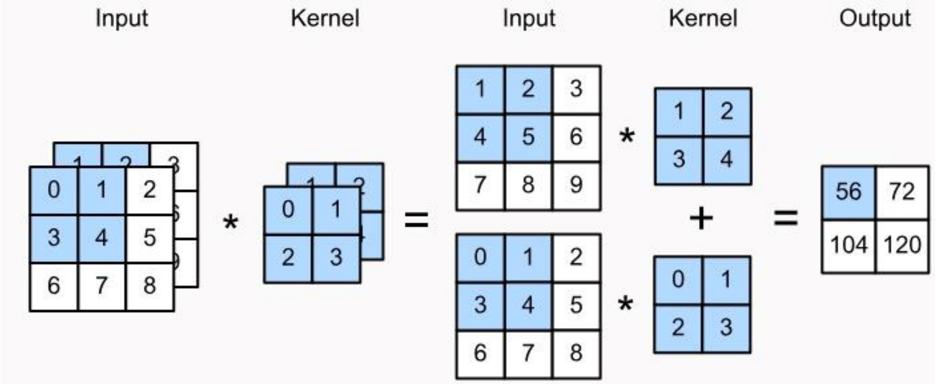
$\boldsymbol{x}$

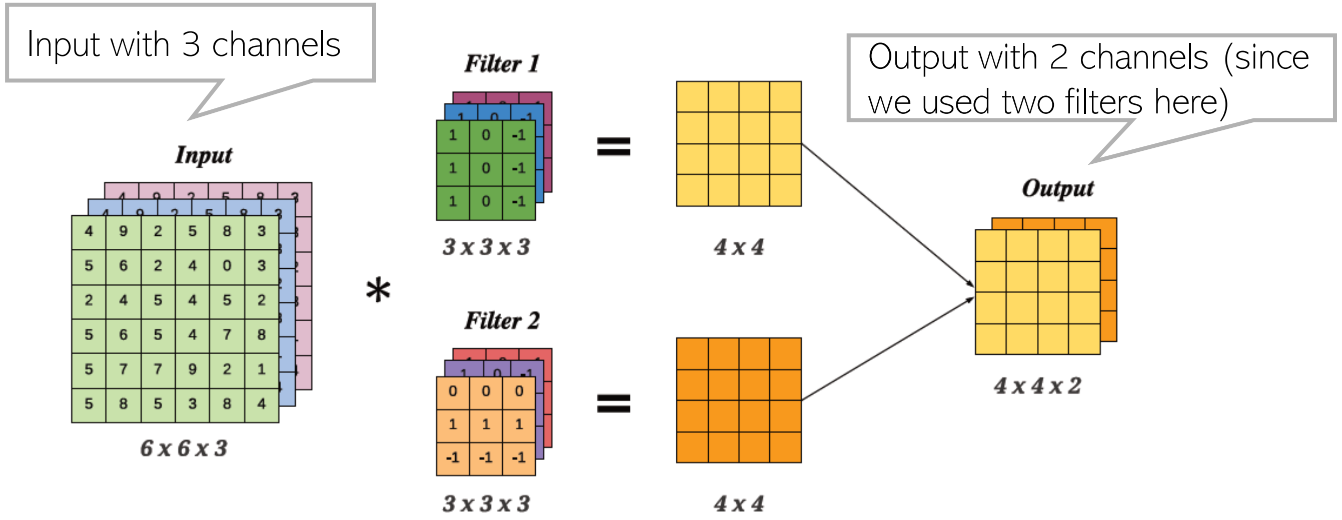


Stride of 2

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)

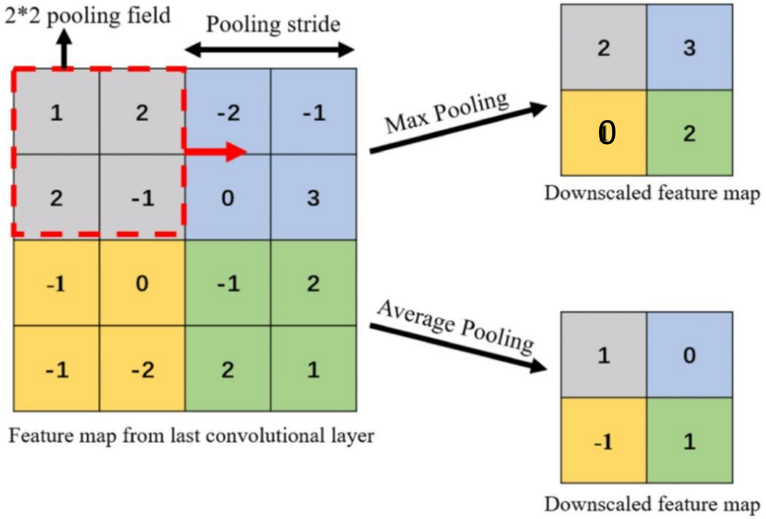




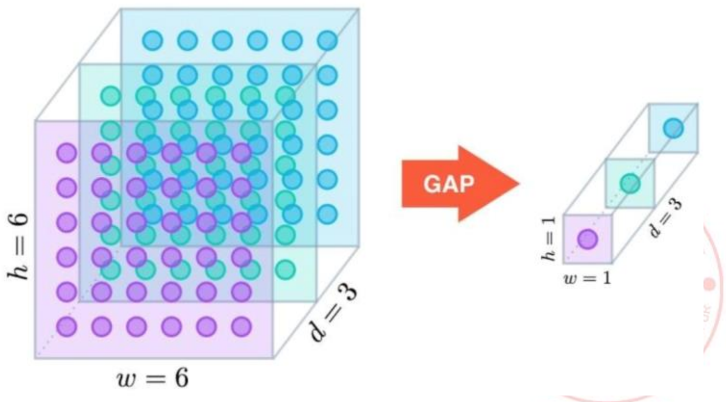
- Note that (right figure above) we typically also have multiple such filters (each with multiple channels) which will give us multiple such feature maps

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

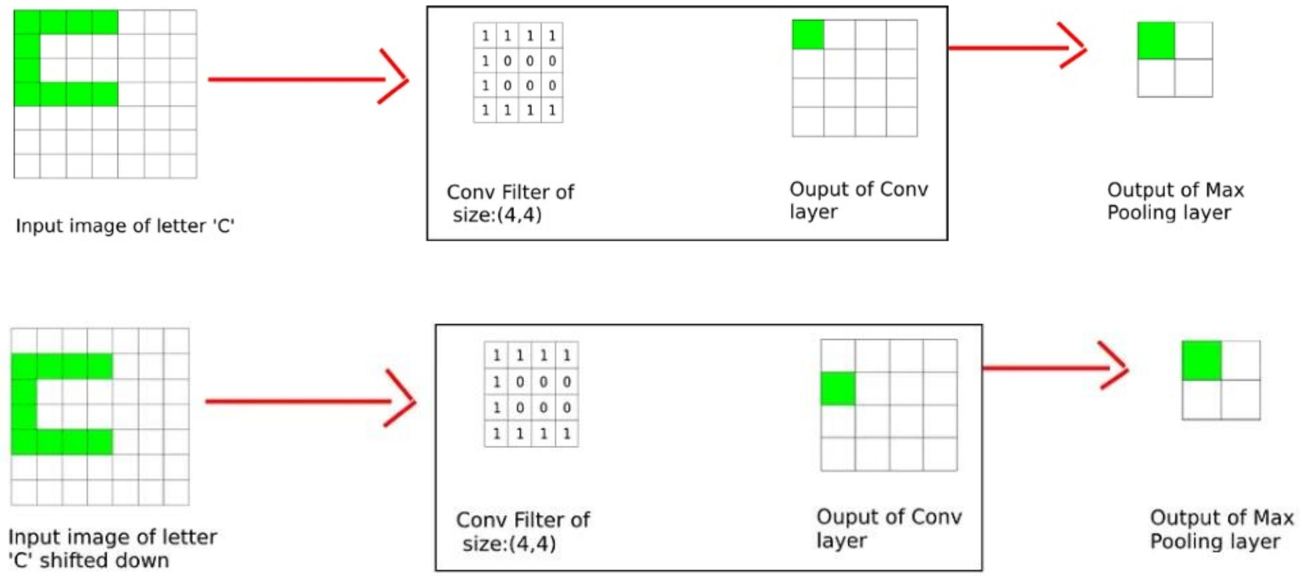


- Max pooling and average pooling are popular pooling methods
- "Global average pooling" (GAP) is another option
  - Given feature map of size $h \times w \times d$ (e.g, if there are d channels), it averages all $h \times w$ locations to give a $1 \times d$ feature map
  - Reduces the number of features 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

Pic credit: https://divsoni2012.medium.com/translation-invariance-in-convolutional-neural-networks-61d9b6fa03df

CNN: Summary of the overall architecture

- The overall structure of a CNN looks something like this

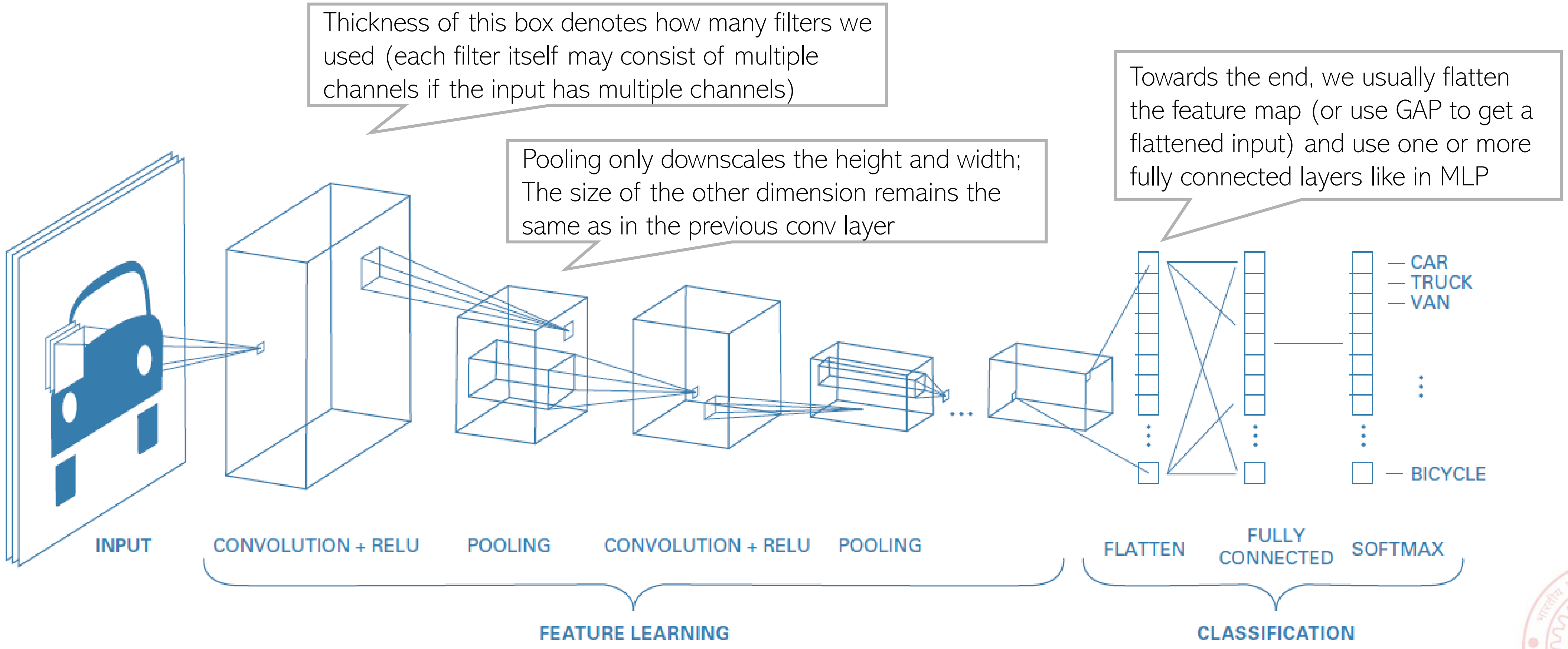


Figure credit: PML-1 (Murphy, 2022),