Image Classification

Nadia Ahmed

Computer Vision Task: **Image Classification**

"Cat"

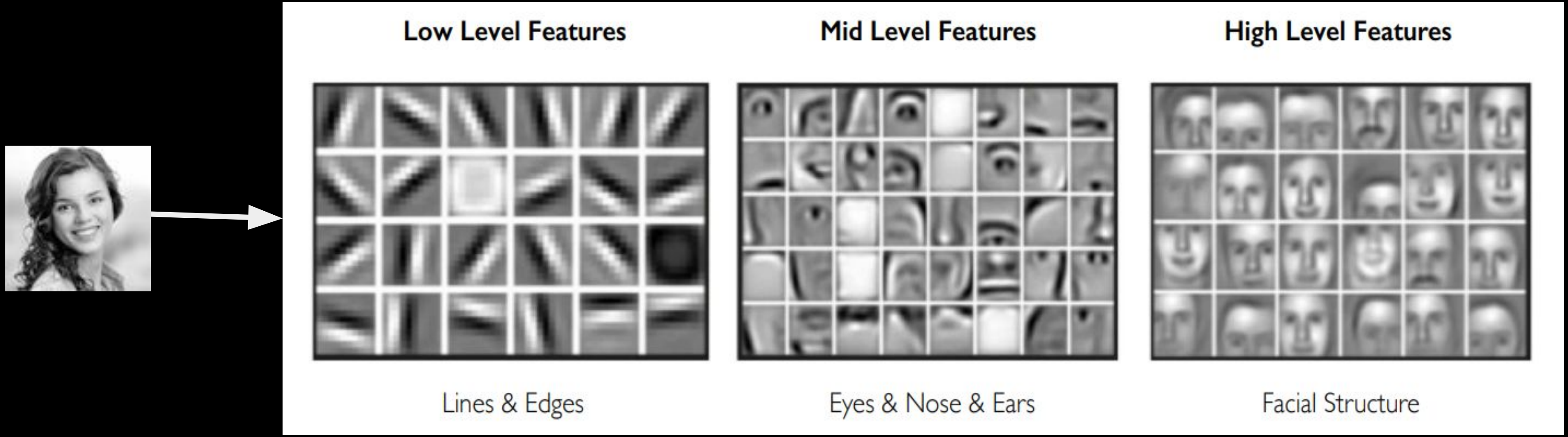
"Dog"

Image Classification:

- **Input:** image
- **Output:** single category for the whole image
- Why is this interesting?
  - Useful for image search and other applications.
  - Simple input/output specification, can resize images to the same dimensions, standard loss functions.
  - Requires reasoning about what is happening in the image-analysis and understanding of the image.
  - Useful high level proxy task to learn good representations which transfer to new tasks and domains.
  - Given a big dataset we can train a representation and take that model and fine tune it on another task and we get a way with fewer data on the target task b/c we have pretrained on a much larger dataset

Feature Learning vs Feature Engineering

- **Edge detector**: early attempts of hand-engineering object detection algorithms failed as object shapes and appearances are too hard to describe
- Need ML and **labeled datasets** to learn relationships



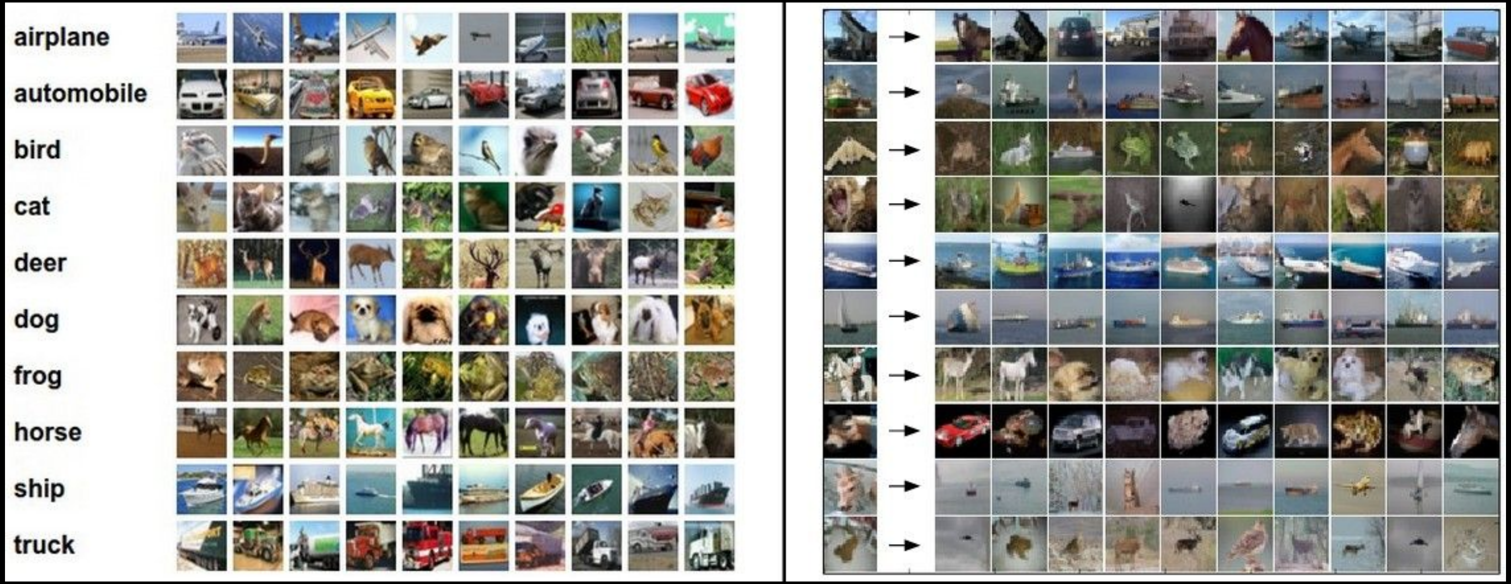
*source: [introtodeeplearning]

The Data Driven Approach

- Popular Classification Single Label Datasets
  - **MNIST**: 60k training w/ labels 10k test 28x28, 10 categories
  - **Caltech101**: 101 categories
  - **ImageNet**: 15,000,000 images w/ 22,000 categories
  - **CIFAR-10**: 60,000 images 10 categories

- An Aside and Preview of Coming Attractions: **Transfer Learning**
  - Models **pre-trained** on large datasets **generalize**
  - Can get CNNs **pre-trained** on **generic tasks** (like ImageNet)
  - Can fine-tune (**retrain**) only the **last layers** on little data of a new task which leads to state of the art performance
  - **Usual** standard approach today-can download model from internet

Simple Models



- Nearest Neighbor Classifier (find the image distance)

$$d(I_1, I_2) = \sum_p |I_1(p) - I_2(p)|_1$$

*source: [Stanford 231n]

Simple Models

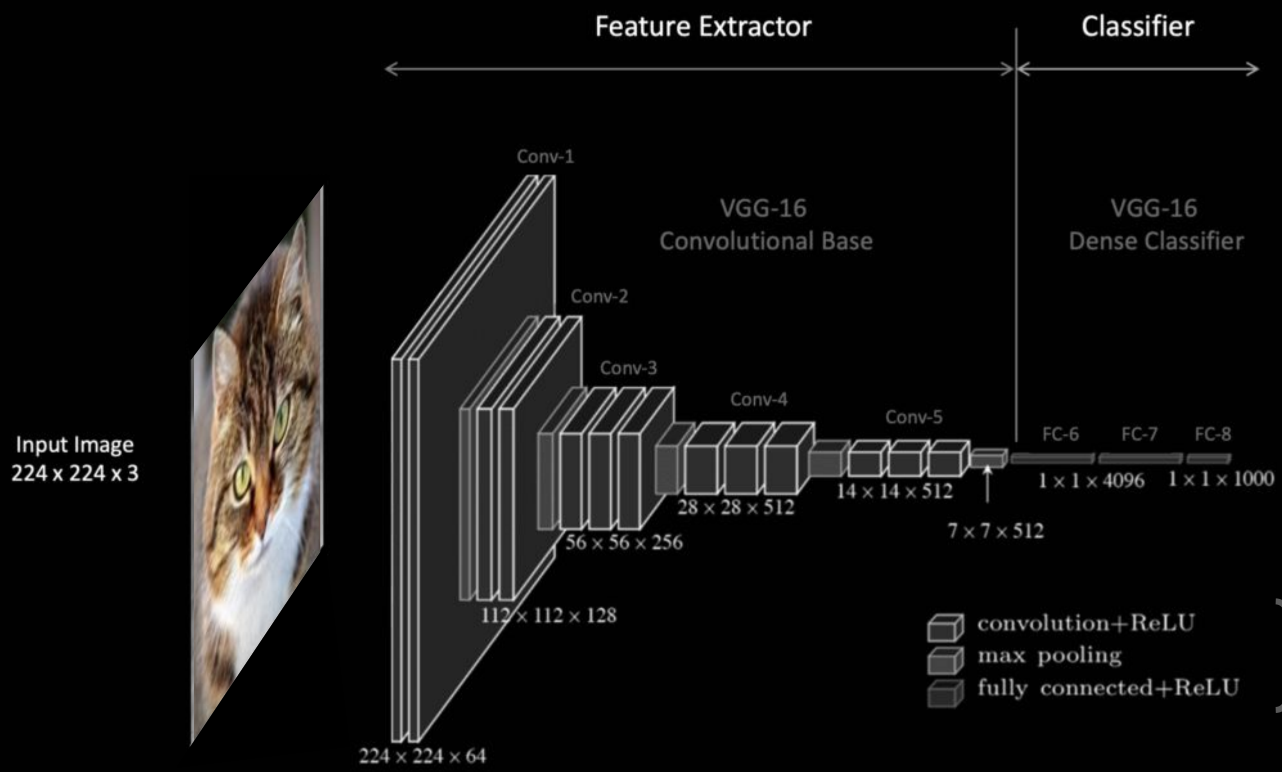
$$
\begin{array}{ccc}
\text{test image} & \text{training image} & \text{pixel-wise absolute value differences} \\
\left| \begin{matrix} 56 & 32 & 10 & 18 \\ 90 & 23 & 128 & 133 \\ 24 & 26 & 178 & 200 \\ 2 & 0 & 255 & 220 \end{matrix} \right. - &
\left. \begin{matrix} 10 & 20 & 24 & 17 \\ 8 & 10 & 89 & 100 \\ 12 & 16 & 178 & 170 \\ 4 & 32 & 233 & 112 \end{matrix} \right| = &
\begin{matrix} 46 & 12 & 14 & 1 \\ 82 & 13 & 39 & 33 \\ 12 & 10 & 0 & 30 \\ 2 & 32 & 22 & 108 \end{matrix} \rightarrow 456
\end{array}
$$

- Nearest Neighbor Classifier (find the image distance)

$$d(I_1, I_2) = \sum_p |I_1(p) - I_2(p)|_1$$

*source: [Stanford 231n]

Convolutional Neural Networks



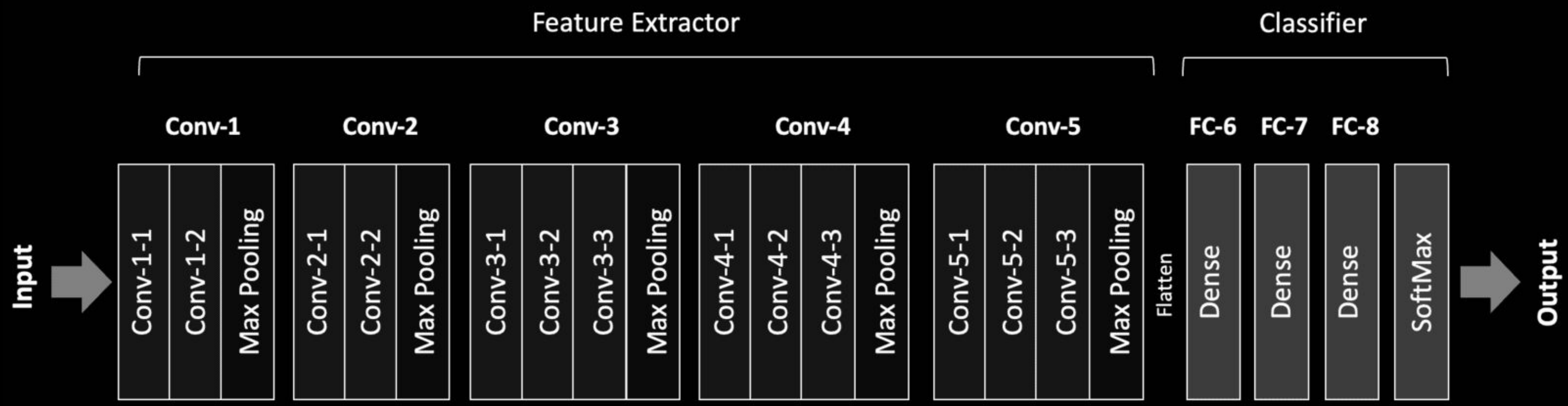
output to softmax

3 types of layers

*source: learnopencv.com

Convolutional Neural Networks

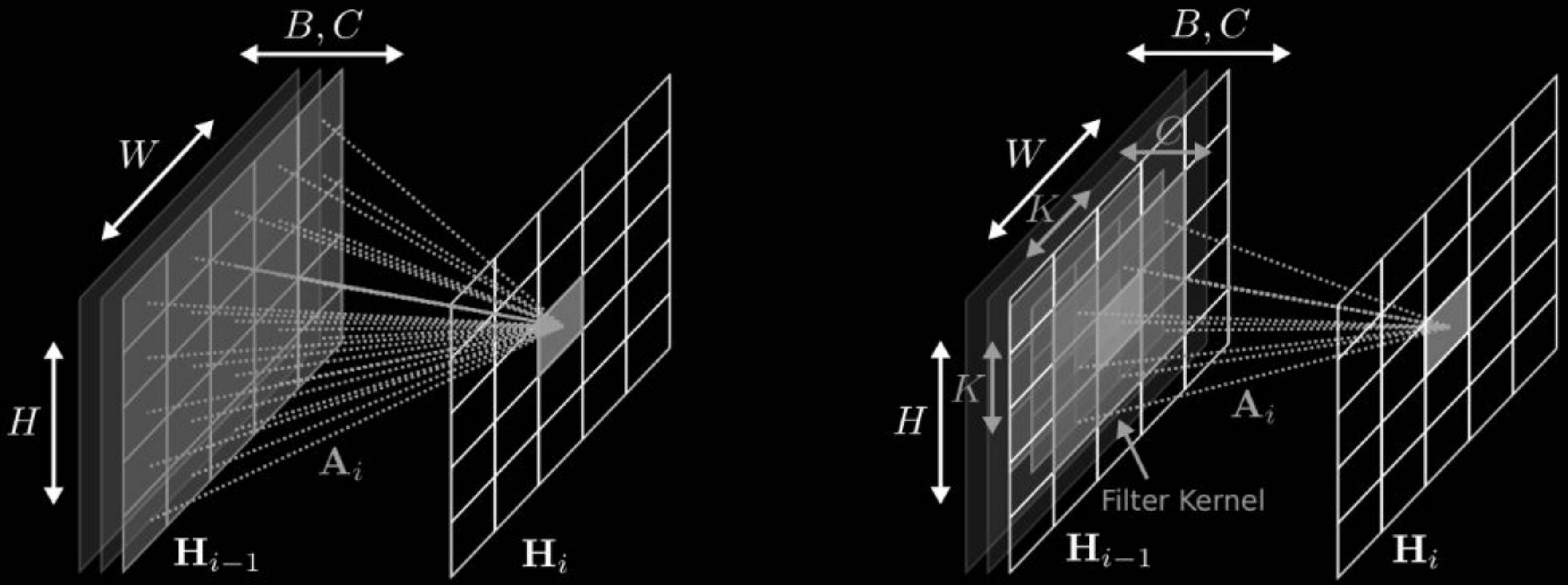
- Convolutional layers
- Downsampling layers
- Fully Connected layers



*source: learnopencv.com

Convolutional Layer

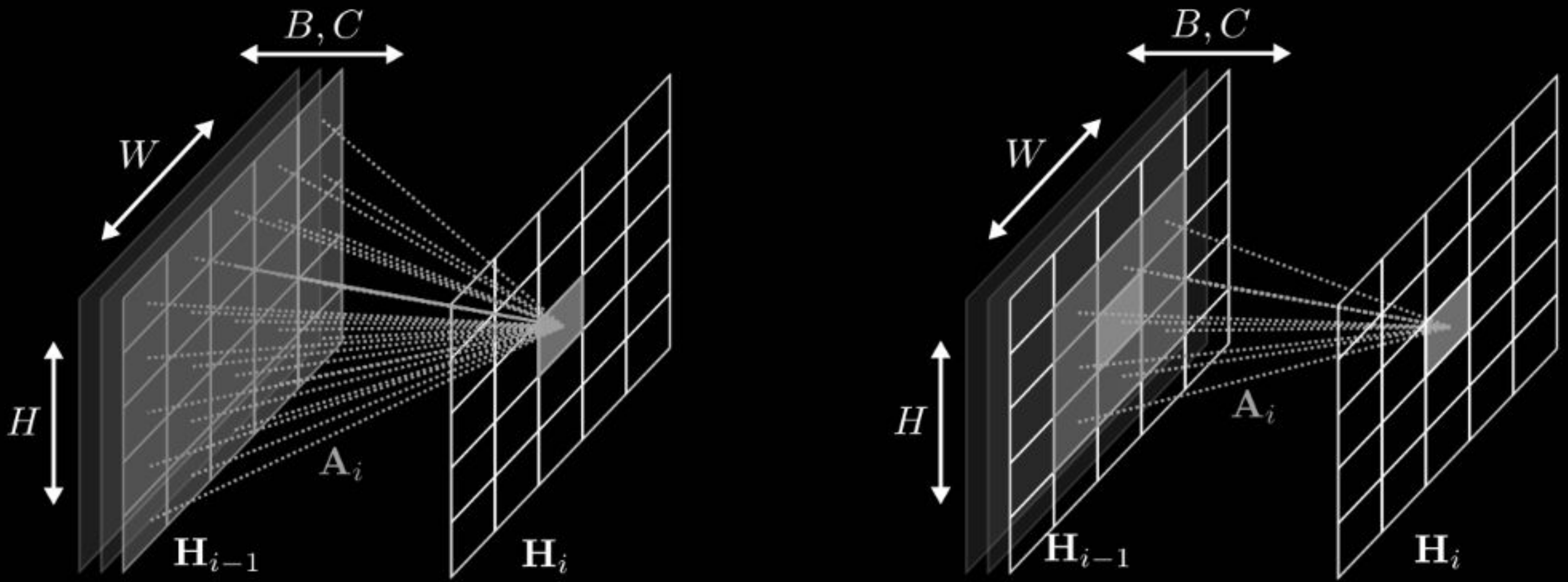
Fully Connected vs. Convolutional Layers



- Compared to fully conn. layers (left), convolutional layers (right) share weights which reduces parameters significantly.
- Convolutional layers are translationally equivariant.

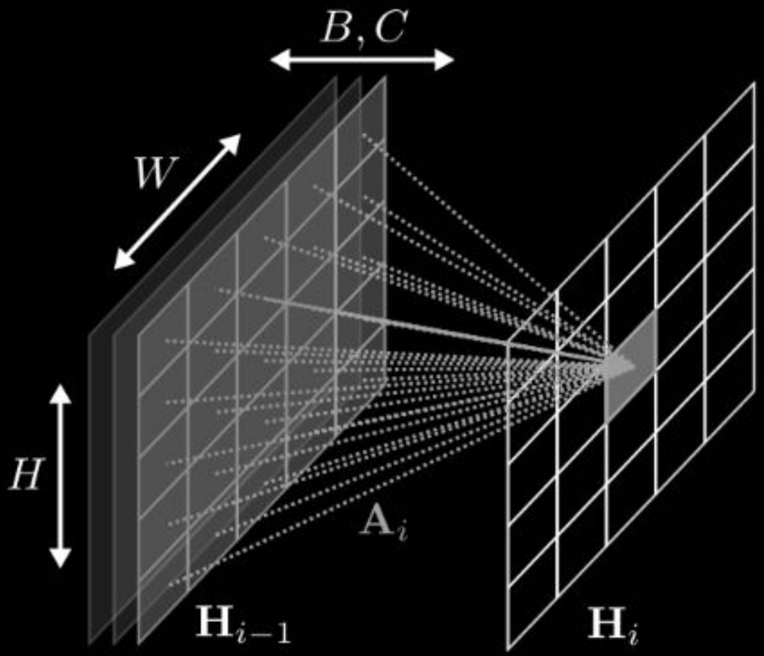
*source: A. Geiger
Wikipedia: Convolution

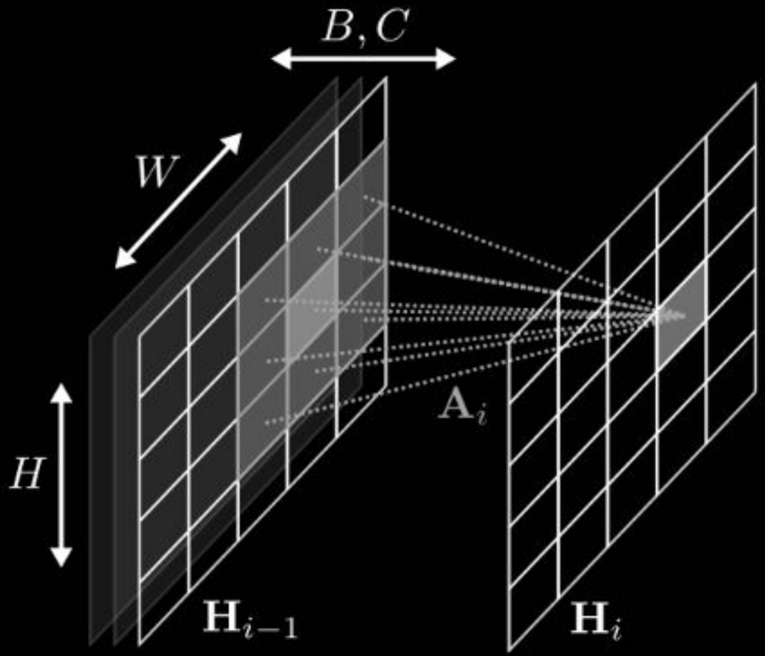
Fully Connected vs. Convolutional Layers



*source: [A. Geiger]

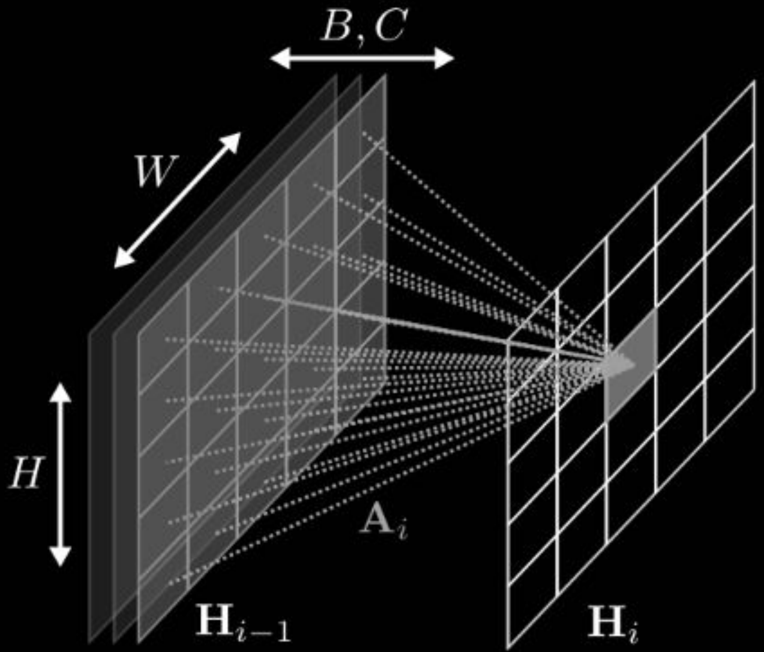
Fully Connected vs. Convolutional Layers





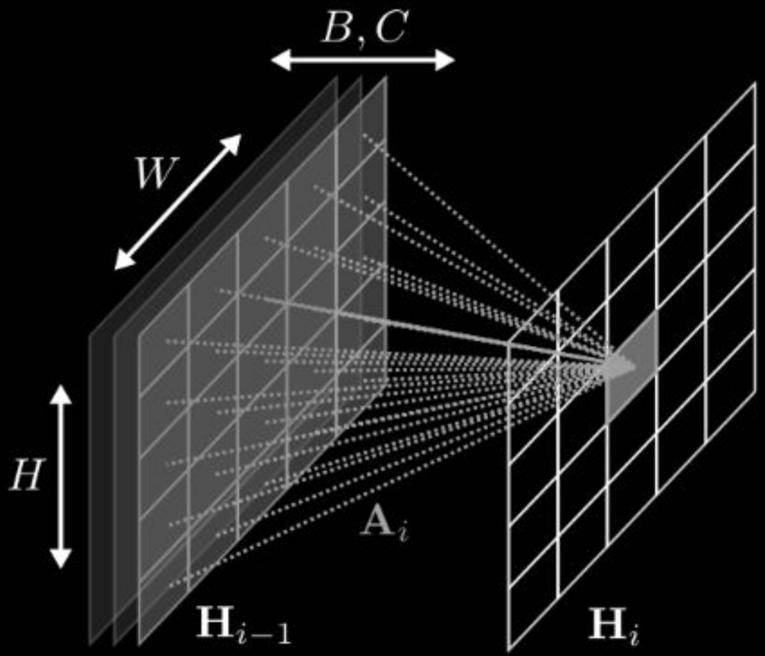
*source: [A. Geiger]

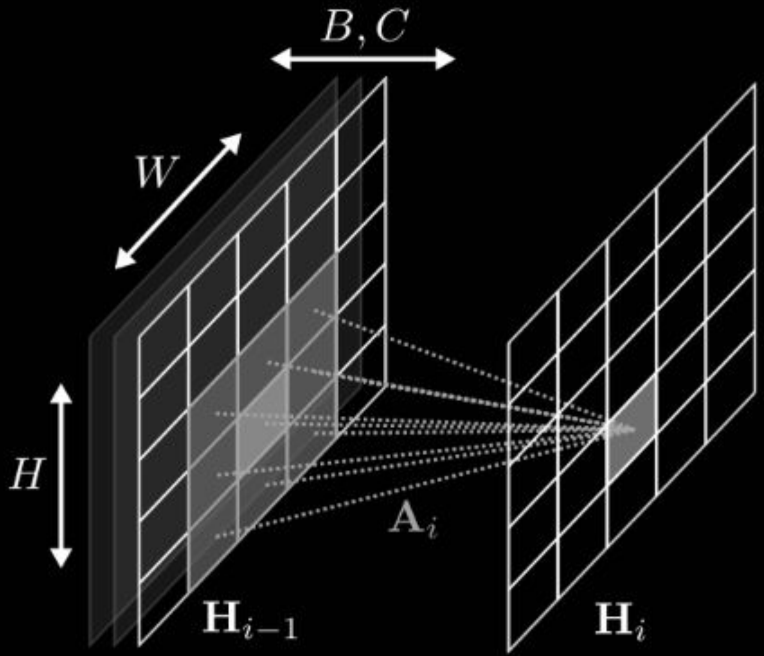
Fully Connected vs. Convolutional Layers



*source: [A. Geiger]

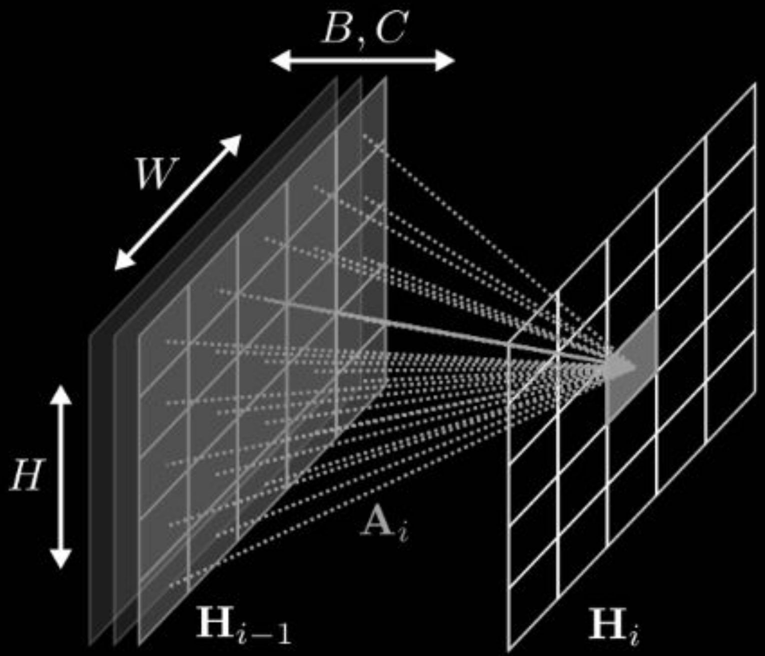
Fully Connected vs. Convolutional Layers

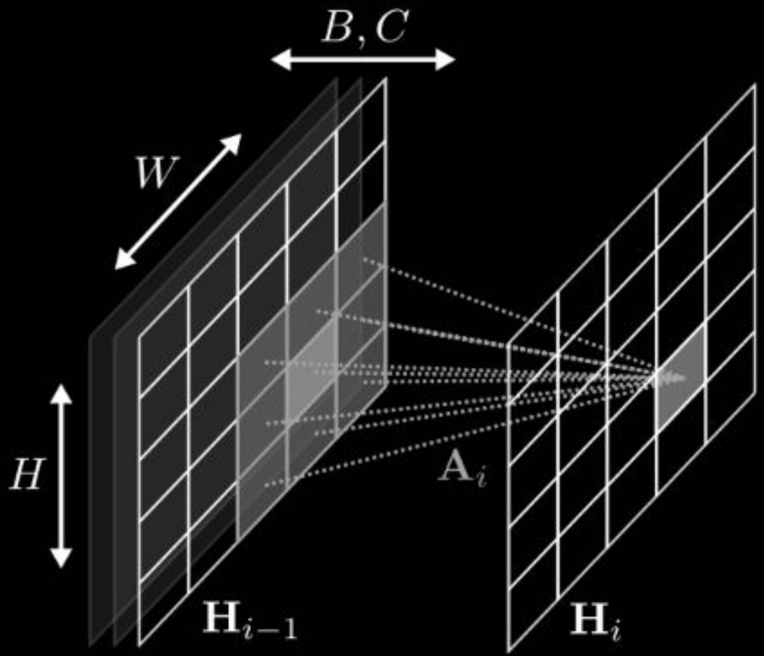




*source: [A. Geiger]

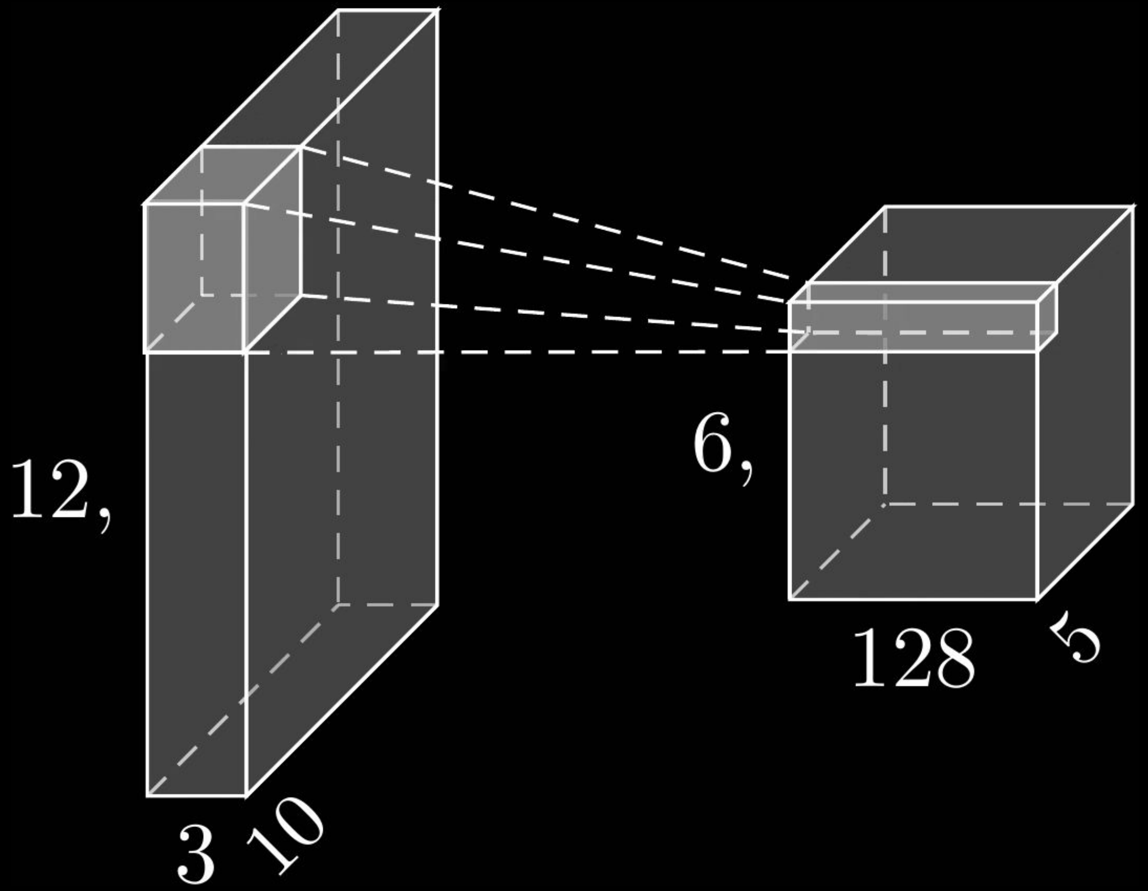
Fully Connected vs. Convolutional Layers





*source: A. Geiger

Convolution is Happening in 3D

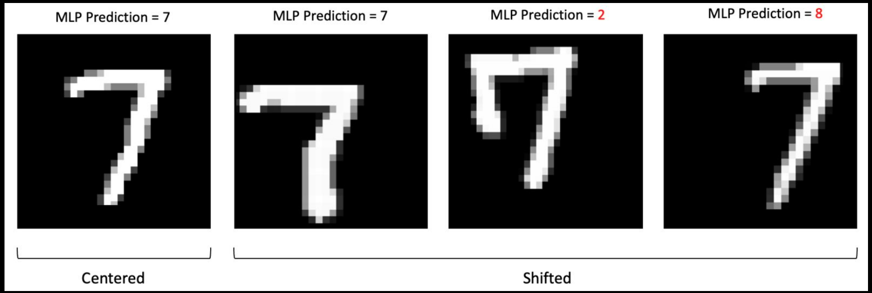


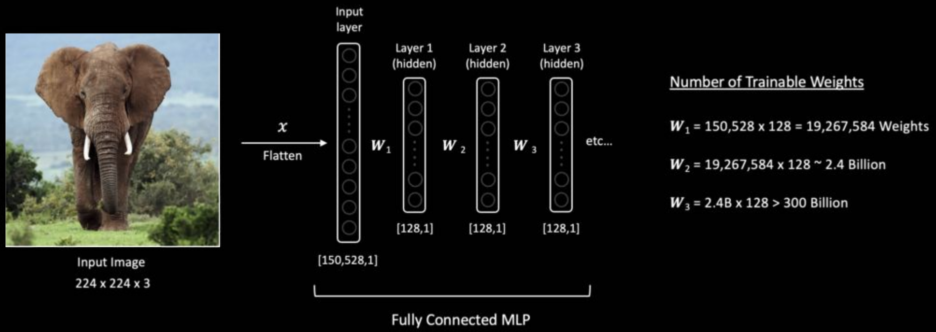
*source: C. White

Fully Connected MultiLayer Perceptron (MLP) vs Convolutional Layers

Multilayer Perceptron:

- Not translational invariant
- Prone to overfitting
  - Since every pixel connects to every neuron the number of weights becomes HUGE which means training takes longer which means the number of parameters to tune becomes larger!



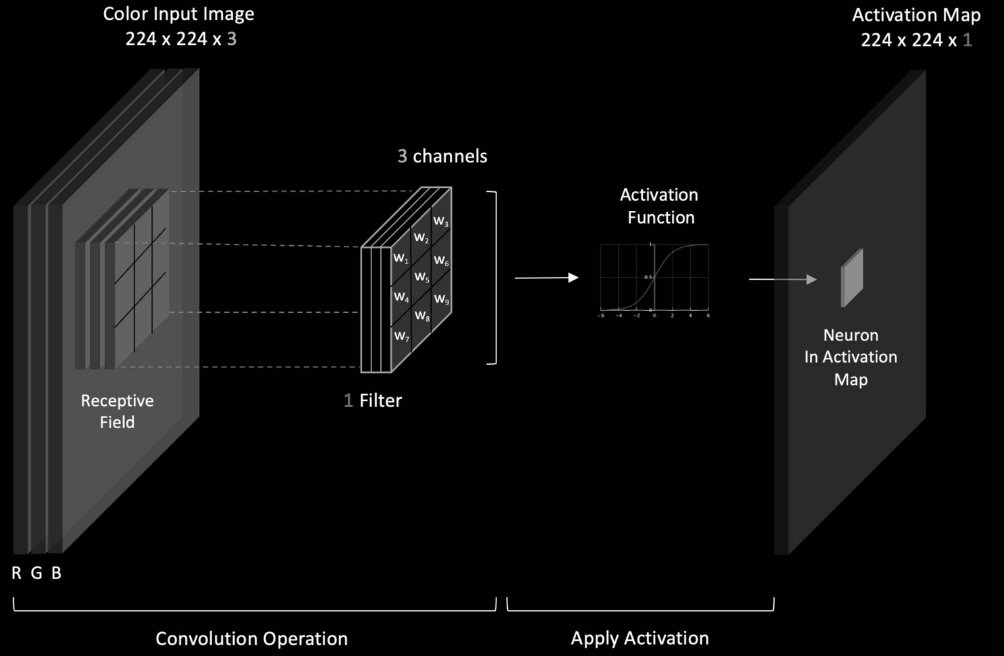


*source: learnopencv.com

Fully Connected MultiLayer Perceptron (MLP) vs Convolutional Layers

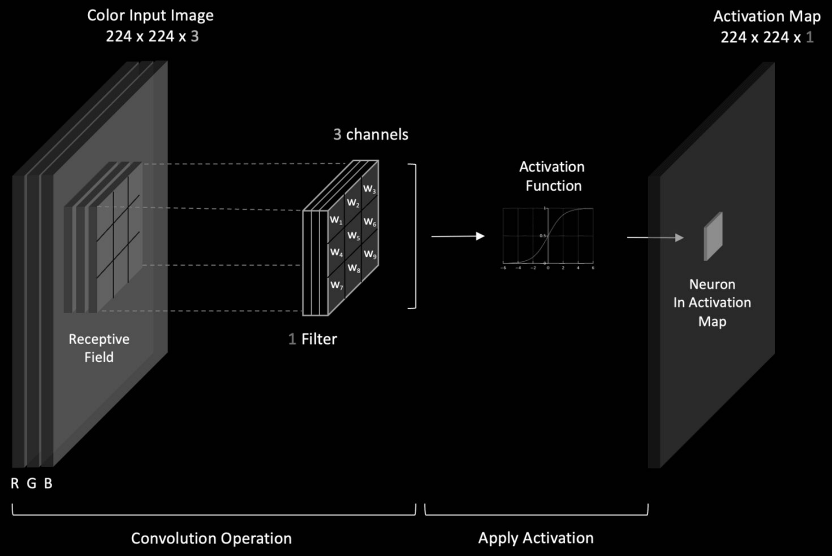
ConvNet:

- Translational equivariance (invariance for classification)
  - parameter sharing: same weights are used to process different parts of the input image
- Eyes of Network



*source: learnopencv.com

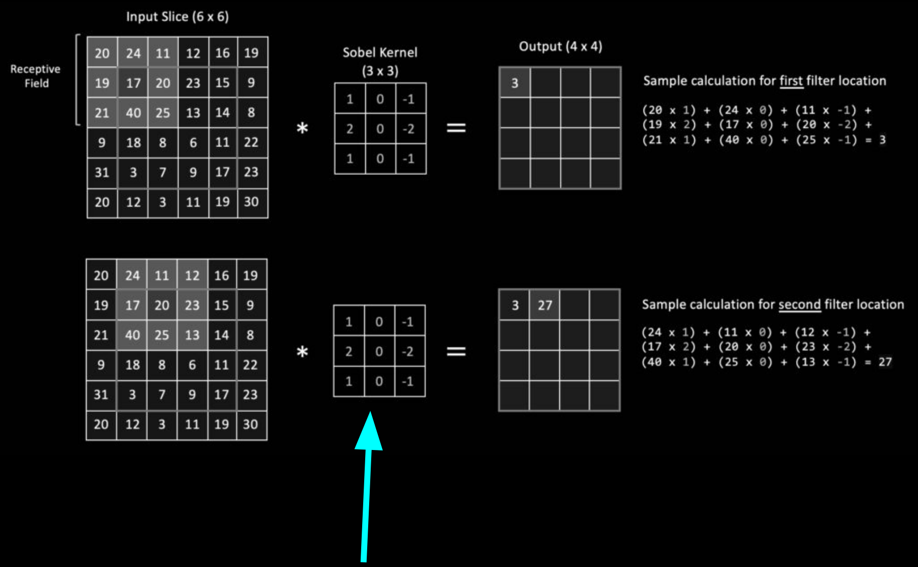
Convolutional Layers



- **filter/kernel** moves across the input, and at each filter location, a convolution operation is performed, which produces a single number
- This value is then passed through an **activation function**, and the output from the activation function populates the corresponding entry in the output, also known as an **activation map**
- the **activation map** is a summary of features from the input via the convolution process
- elements in the **kernel** are weights that are learned by the network during training
- Sliding the filter one pixel at a time corresponds to a **stride** of one
- The region of the filter performs the operation on is called the **receptive field**

*source: learnopencv.com

Convolutional Layers



** Sobel Filter for math demonstration purposes only. Actual elements in the kernel are weights that are learned by the network during training

- **filter/kernel** moves across the input, and at each filter location, a convolution operation is performed, which produces a single number
- This value is then passed through an **activation function**, and the output from the activation function populates the corresponding entry in the output, also known as an **activation map**
- the **activation map** is a summary of features from the input via the convolution process
- elements in the **kernel** are weights that are learned by the network during training
- Sliding the filter one pixel at a time corresponds to a **stride** of one
- The region of the filter performs the operation on is called the **receptive field**

*source: learnopencv.com

Convolutional Layer Properties

- The **depth of a filter** (channels) must **match** the depth of the input data (i.e., the number of channels in the input).
- The **spatial size** of the filter is a **design choice** but 3x3 is very common (or sometimes 5x5).
- The number of **filters** is also a **design choice** and dictates the number of activation maps produced in the output.
- Multiple **activations maps** are sometimes collectively referred to as “an activation map containing multiple channels.” But we often refer to each channel as an activation map.
- Each channel in a filter is referred to as a **kernel**, so you can think of a filter as a container for kernels.
- A single-channel filter has just one kernel. Therefore, in this case, the filter and the kernel are one and the same.
- The **weights in a filter/kernel** are initialized to **small random values** and are **learned** by the network during training.
- The **number of trainable parameters** in a convolutional layer depends on the **number of filters, the filter size, and the number of channels** in the input.

*source: learnopencv.com

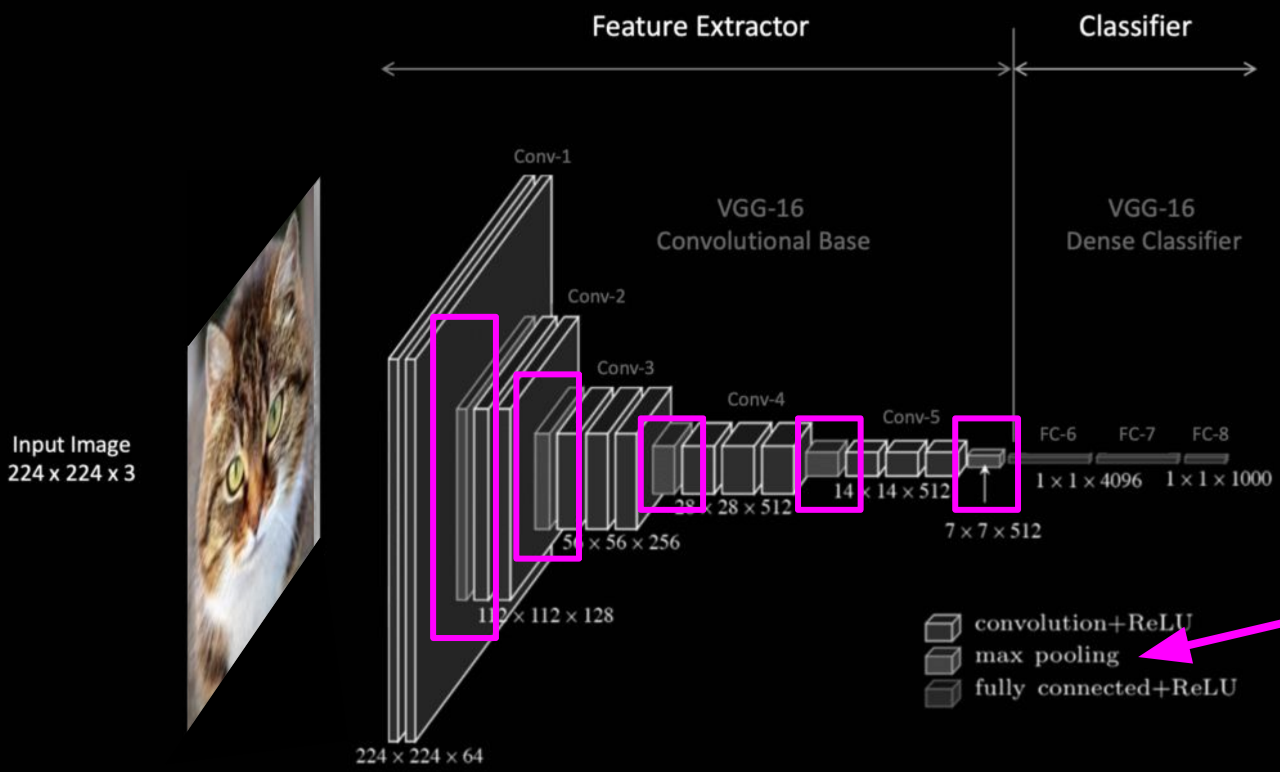
TLDR: Convolutional Layer

- **Input:** volume of size $W_1 \times H_1 \times D_1$
- Requires 4 hyperparameters
  - Number of filters **K**
  - The kernel size/spatial extent **F**
  - The stride **S**
  - The amount of zero padding **P**
- **Output:** volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$
- With parameter sharing, it introduces $F*F*D_1$ weights per filter, for a total of $(F*F*D_1)-K$ weights and **K** biases.
- In the output volume, the dth depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the dth filter over the input volume with a stride of **S**, and then offset by dth bias

*source: [Stanford 231n]

Downsampling Layer

Downsampling Layer

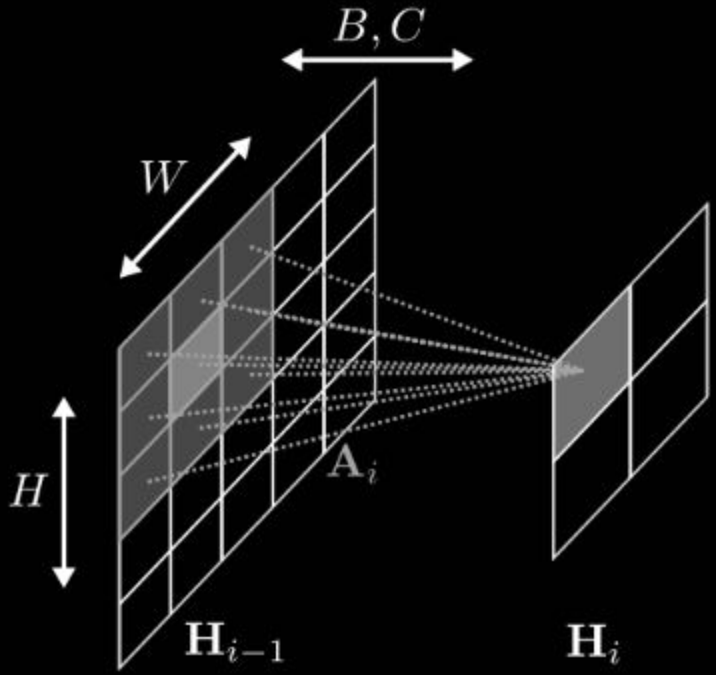


- **reduces** the **spatial resolution**
- increases the **receptive field**

*source: learnopencv.com
[Simonyan et. al 2014]()

Downsampling: Pooling Layer

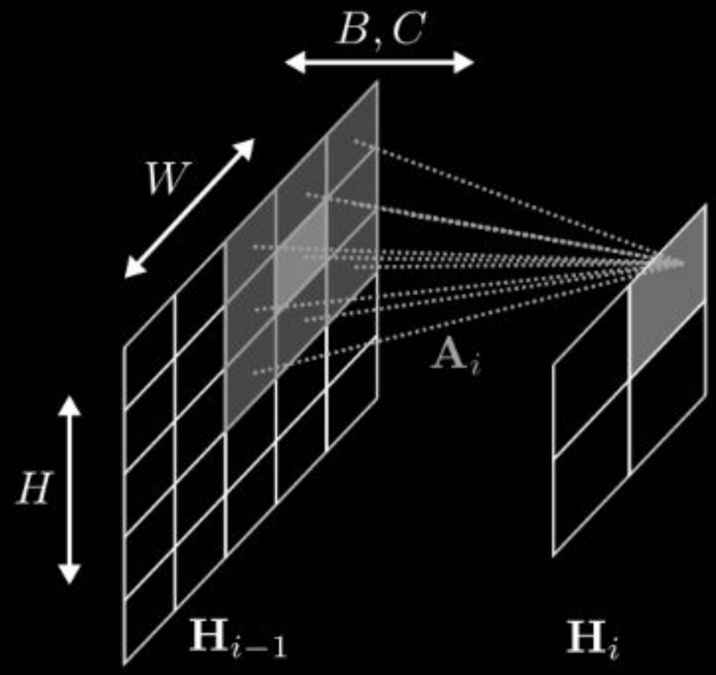
- Stride s=2 and kernel size 2x2 reduces spatial dimension by 2
- Pooling has no parameters
  - max pooling
  - min pooling
  - mean pooling
- Pooling is applied to each channel separately preserving the number of channels



*source: A. Geiger

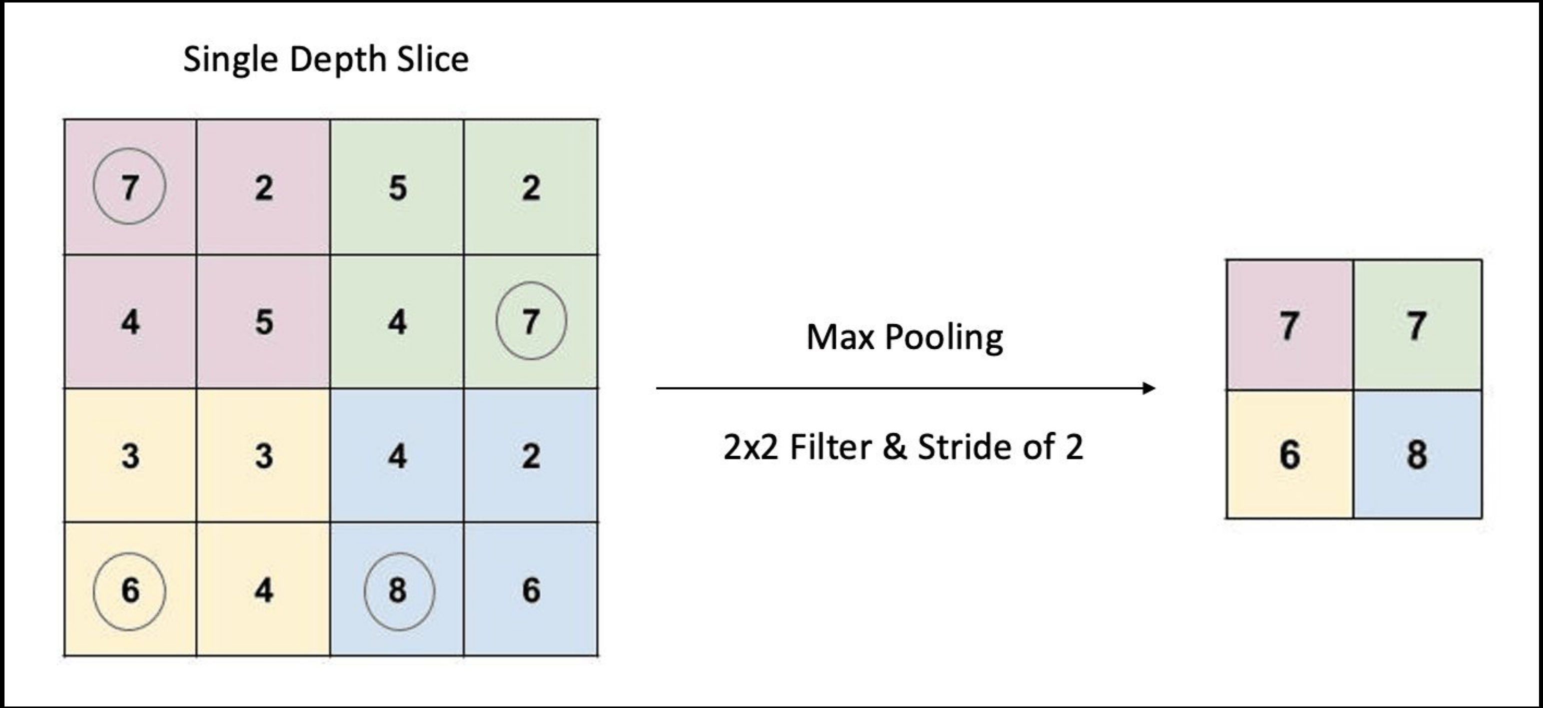
Downsampling: Pooling Layer

- Stride s=2 and kernel size 2x2 reduces spatial dimension by 2
- Pooling has no parameters
  - max pooling
  - min pooling
  - mean pooling
- Pooling is applied to each channel separately preserving the number of channels



*source: [A. Geiger]

Pooling Layer Example: Max Pooling



*source: learnopencv.com

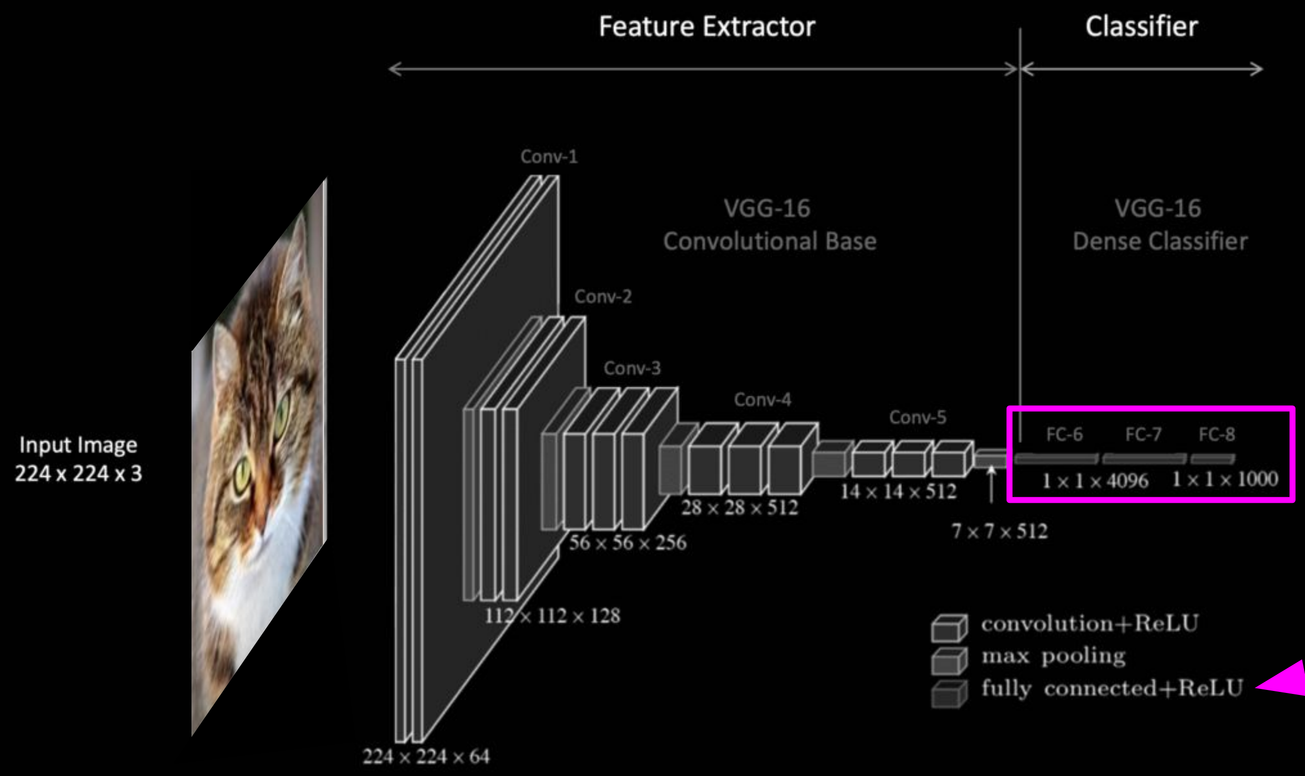
TLDR: Pooling Layer

- **Input:** volume of size $W_1 \times H_1 \times D_1$
- Requires two hyperparameters:
  - The spatial extent/kernel size F
  - The stride S
- Produces a volume of size $W_2 \times H_2 \times D_2$
  - $W_2 = (W_1 - F)/S + 1$
  - $H_2 = (H_1 - F)/S + 1$
  - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- For Pooling layers, it is not common to pad the input using zero-padding

*source: [Stanford 231n]

Fully Connected Layer

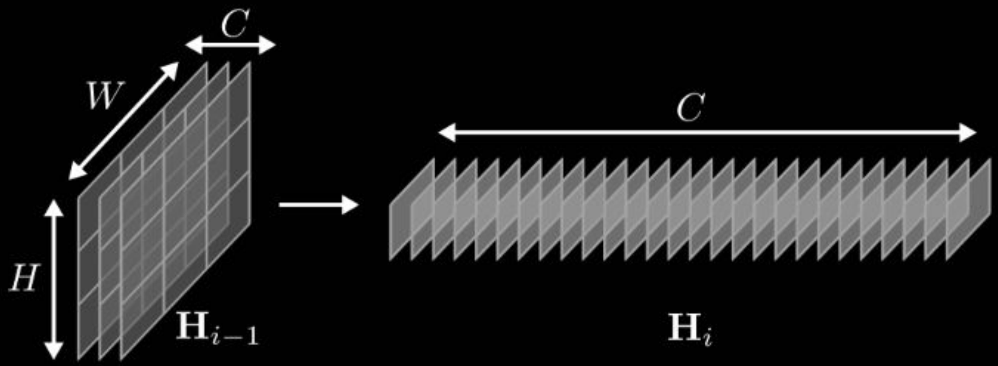
Fully Connected Layer



- The most memory intensive because all pixels contribute

*source: learnopencv.com
[Simonyan et. al 2014]()

Fully Connected Layer

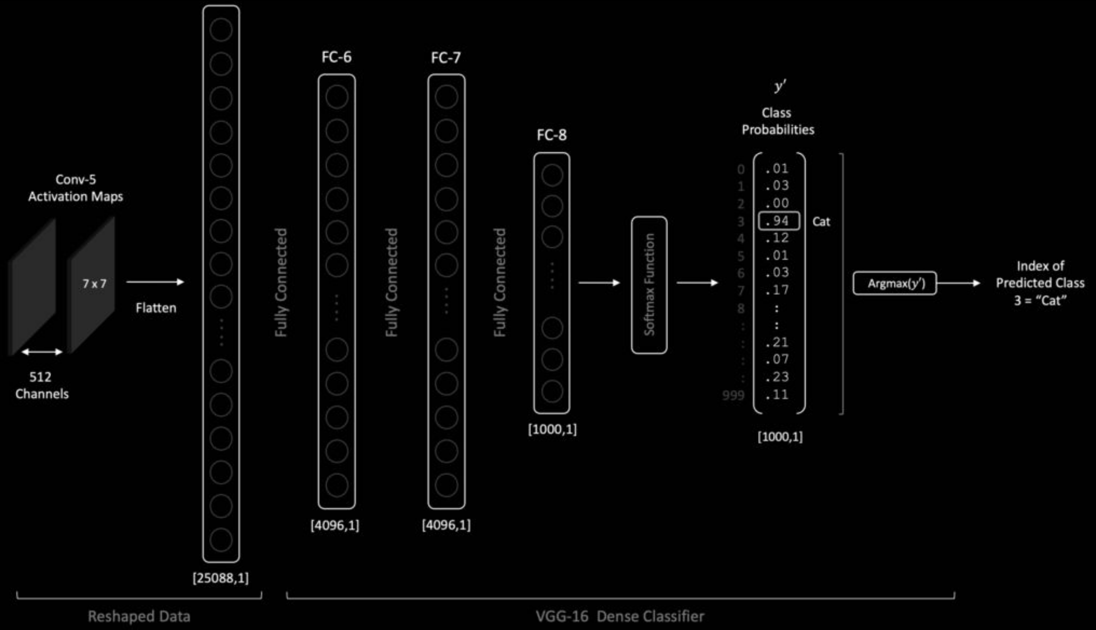


- Reshape $H_{i-1}[B, X, Y, C]$ into $H_i[B, C]$
- Now X and Y are reshaped into the feature channel dimension C

*source: A. Geiger

Fully Connected Layer

- The number of neurons in the output layer of the network is equal to the number of classes
- Each output neuron is associated with a specific class
- Each neuron's value (after the softmax layer) represents the probability that the associated class for that neuron corresponds to the class associated with the input image.



*source: learnopencv.com

The Output

Notice that the output of our fully connected layer becomes the input of our **softmax** function which outputs a one dimensional array that maps probabilities over labels or categories.

Since image classification results in a label or category, we need to convert that label which is a string to a numeric value that the model can understand.

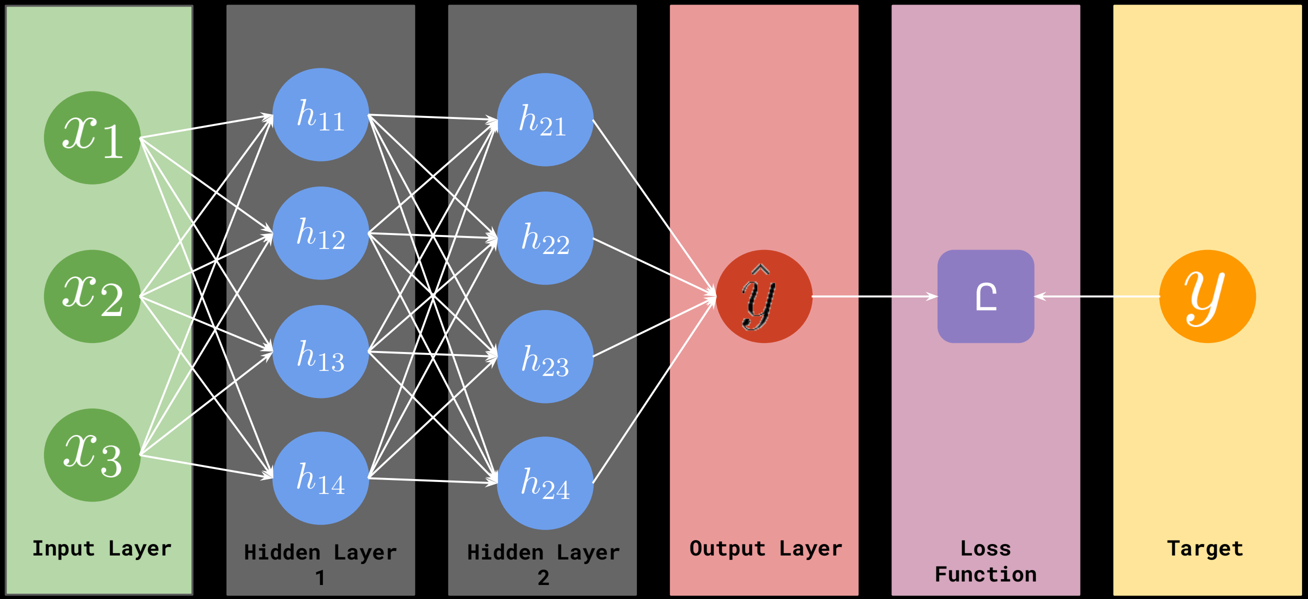
Categorical data can be

- nominal: one-hot encoded (preferred)
- ordinal: label encoded (arbitrary numeric)

*source: Wikipedia

Loss Function

The Output and Loss



- The **output layer** is the last layer which calculates the output
- The **loss function** compares the result of the output layer to the target
- In **image classification** we use a **softmax** output layer and a **cross entropy loss** to get probability values over labels.

*source: [A. Geiger]

Model Output: Categorical Distribution

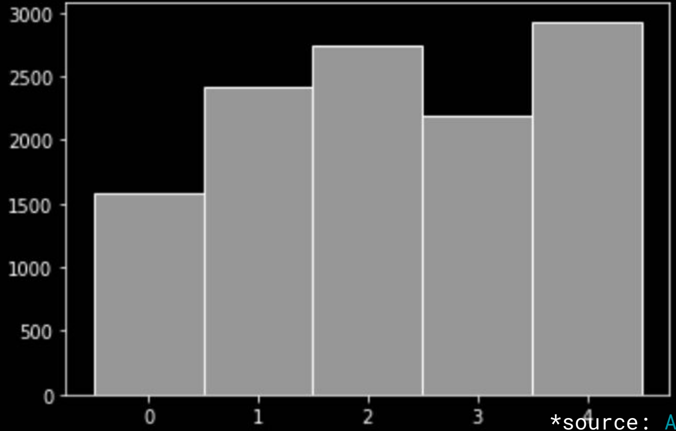
- Obtain a probability distribution over the possible classes that the model can predict
- A vector of scores for each possible class
- The representation is important because it allows us to quantify the model's uncertainty about the predictions and compare it with the ground truth labels.

Categorical Distribution: Index label

$$p(y = c) = \mu_c$$

- μ_c : probability for class c

```
classes = ['filaments', 'flares', 'prominences', 'quiet', 'sunspots']
```



One Hot Encoding Notation:

$$p(y) = \prod_{c=1}^{C} \mu_c^{y_c} \qquad y_c \in 0, 1$$

- y: "one-hot" vector with
- $y = (0, ...0, 1, ..., 0)^\top$ with all zeros except for one (the true class)

*source: A. Geiger
S. Chatterjee

One-Hot Vector Representation

Class	Label	y(indexed)	y(one-hot)
	“Cat”	1	$[1,0]^T$
	“Dog”	2	$[0,1]^T$

- One hot vector y w/ binary elements
- Index c where $y_c = 1$ determines the correct class

*source: A. Geiger

Categorical Distribution & Cross-Entropy Loss

Let $p_{model}(y|x,w) = \prod_{c=1}^{C} f_w^{(c)}(x)^{y_c}$ be a **Categorical distribution**

Maximizing the log-likelihood leads to the **cross-entropy** loss:

$$
\begin{aligned}
\hat{\mathbf{w}}_{ML} &= \underset{\mathbf{w}}{\operatorname{argmax}} \sum_{i=1}^{N} \log p_{model}(\mathbf{y}_i|\mathbf{x}_i, \mathbf{w}) \\
&= \underset{\mathbf{w}}{\operatorname{argmax}} \sum_{i=1}^{N} \log \prod_{c=1}^{C} f_{\mathbf{w}}^{(c)}(\mathbf{x}_i)^{y_{i,c}} \\
&= \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{N} \underbrace{\sum_{c=1}^{C} -y_{i,c} \log f_{\mathbf{w}}^{(c)}(\mathbf{x}_i)}_{\text{CE Loss}}
\end{aligned}
$$

[Math explanation: A. Webb]

The target y = (0,...,0,1,0...,0)$^\top$ is a "one-hot" vector with y_c as it's Cth element.

*source: [A. Geiger]

Softmax

How can we ensure that $f_w^{(c)}(x)^{y_c}$ predicts a valid categorical distribution?

- We must guarantee 2 things:

$$f_w^{(c)}(x) \in [0,1] \qquad\qquad \sum_{c=1}^{C} f_w^{(c)}(x) = 1$$

- Use **softmax** to do this:

$$\text{softmax}(\mathbf{x}) = \left(\frac{\exp(x_1)}{\sum_{k=1}^{C} \exp(x_k)}, \dots, \frac{\exp(x_C)}{\sum_{k=1}^{C} \exp(x_k)} \right)$$

- Let the score vector **s** denote the network output after the last affine layer, then:

$$f_{\mathbf{w}}^{(c)}(\mathbf{x}) = \frac{\exp(s_c)}{\sum_{k=1}^{C} \exp(s_k)} \quad \Rightarrow \quad \log f_{\mathbf{w}}^{(c)}(\mathbf{x}) = s_c - \log \sum_{k=1}^{C} \exp(s_k)$$

*source: [A. Geiger](#) [paperswcode](#)

Log Softmax

Let the score vector **s** denote the network output after the last affine layer.

$$f_{\mathbf{w}}^{(c)}(\mathbf{x}) = \frac{\exp(s_c)}{\sum_{k=1}^{C}\exp(s_k)} \quad \Rightarrow \quad \log f_{\mathbf{w}}^{(c)}(\mathbf{x}) = s_c - \log \sum_{k=1}^{C} \exp(s_k)$$

- If c is the correct class label we would like to maximize the log softmax
  - we would like s_c for the correct class to increase
  - we would like s_k to decrease (negative)
  - The second term can be approximated to $\log \sum_{k=1}^{C} \exp(s_k) \approx \max_k s_k$ since $\exp(s_k)$ insignificant for all $s_k < \max_k s_k$
  - The loss is always strongly penalized for incorrect predictions
  - If the correct class has the largest score, both terms cancel and it will contribute very little to the overall training cost.

*source: A. Geiger

Softmax

- Softmax responds to differences between inputs
- It is invariant to adding the same scalar to all its inputs:

$$softmax(x) = softmax(x + c)$$

- Another variant of this is (taking into account floating point approximations required by computation):

$$softmax(x) = softmax(x - \max_{k=1,\ldots,L} x_k)$$

- It allows for accurate computation even when x is large

*source: A. Geiger

Cross Entropy Loss

The cross entropy loss for a single training sample $(x, y) \in \chi$:

$$\sum_{c=1}^{C} -y_c \log f_w^{(c)}(x)$$

Example: Suppose 4 classes and 4 training samples:

Input **x**		Label **y**	Predicted **s**	softmax(**s**)	CE Loss
	"Cat"	$[1,0,0,0]^T$	$[+3,+1,-1,-1]^T$	$[0.85,0.12,0.02,0.01]^T$	0.16
	"Dog"	$[0,1,0,0]^T$	$[+3,+3,+1,+0]^T$	$[0.46,0.46,0.06,0.02]^T$	0.78
	"Bunny"	$[0,0,1,0]^T$	$[+1,+1,+1,+1]^T$	$[0.25,0.25,0.25,0.25]^T$	1.38
	"Horse"	$[0,0,0,1]^T$	$[+3,+2,+3,-1]^T$	$[0.42,0.15,0.42,0.01]^T$	4.87

← contributes most to loss!

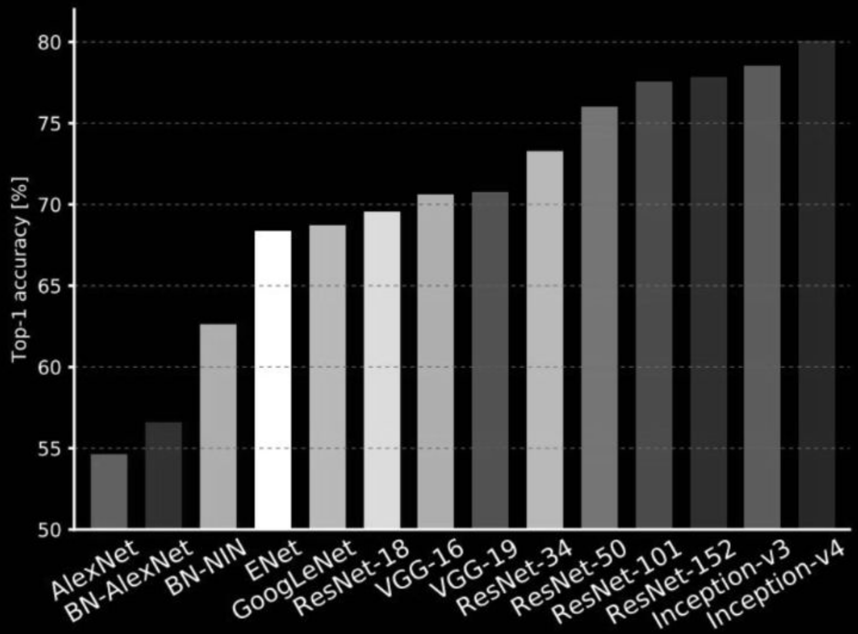
*source: A. Geiger

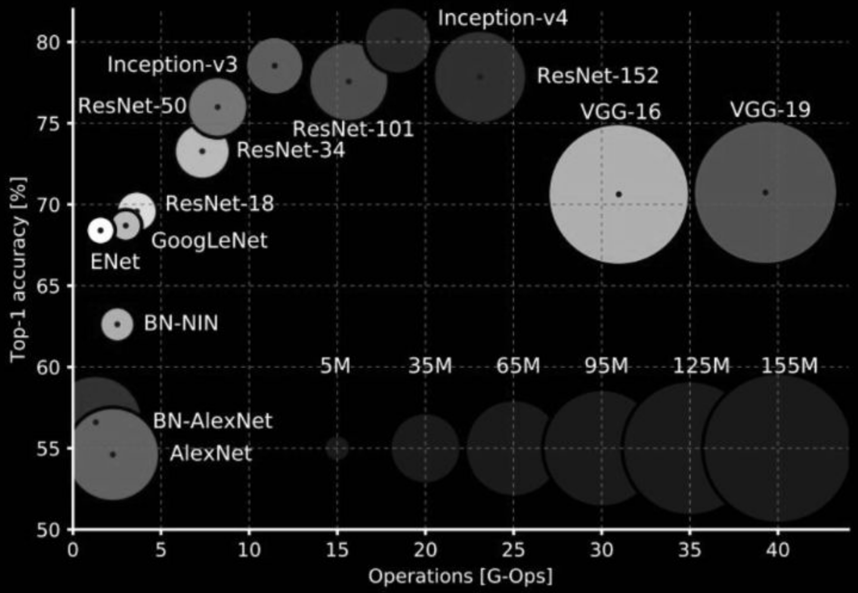
*source: Stanford 231n

Popular Image Classification Architectures

- LeNet-5: 2Conv, 2Pool, 2Fully Connected (MNIST)
- AlexNet: 8 layers, ReLus, dropout, augmentation
- VGG (slowest b/c of number of parameters): 16, 19 layer versions (uses 3x3 receptive field instead of 7x7)
- Inception/GoogLeNet: 22 layers, inception modules
- ResNet: 152 layers use residuals w/ skip connections

Accuracy vs Complexity





*source: A. Geiger