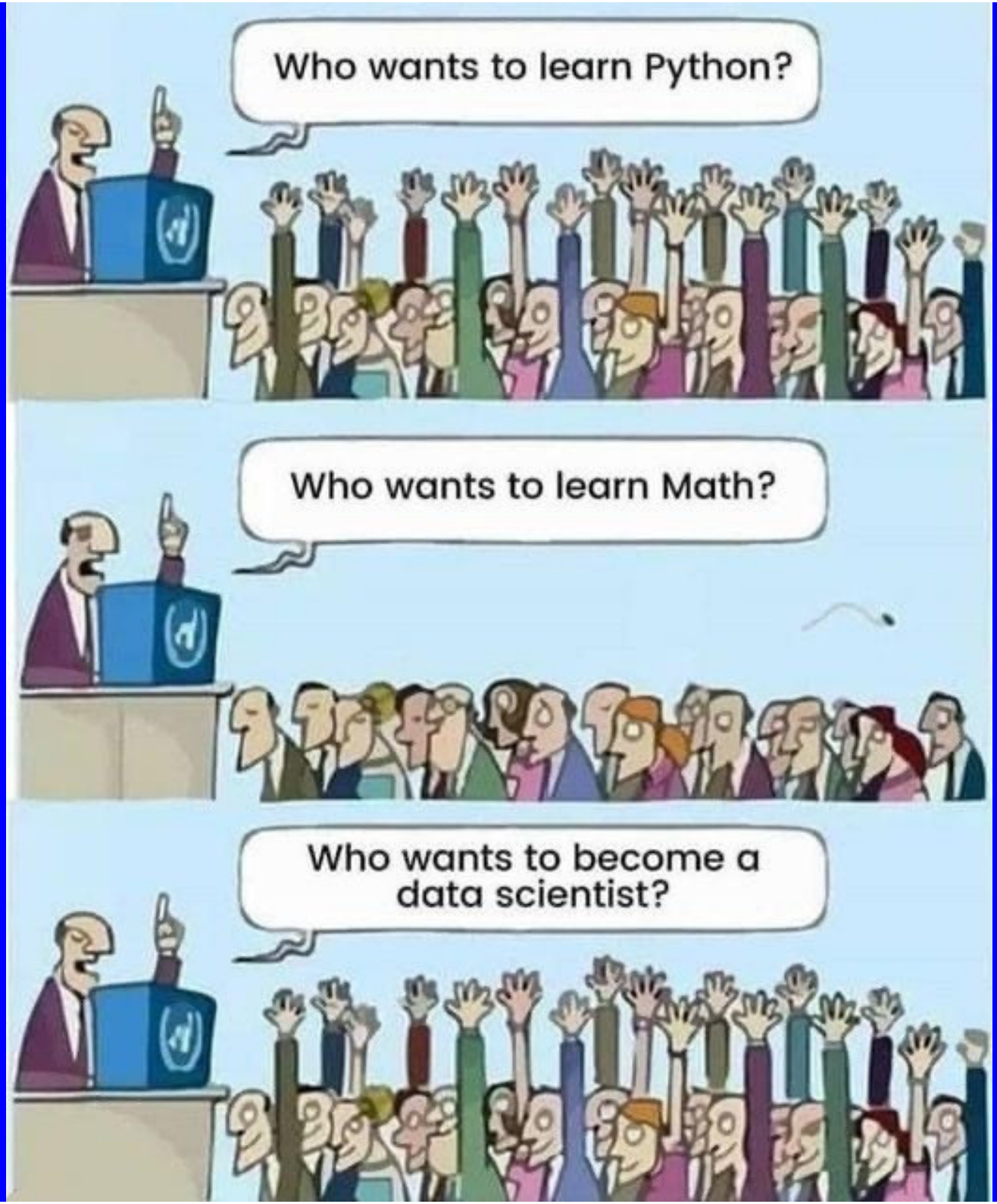
Convolutional Neural Networks
or
ConvNets
or
CNNs

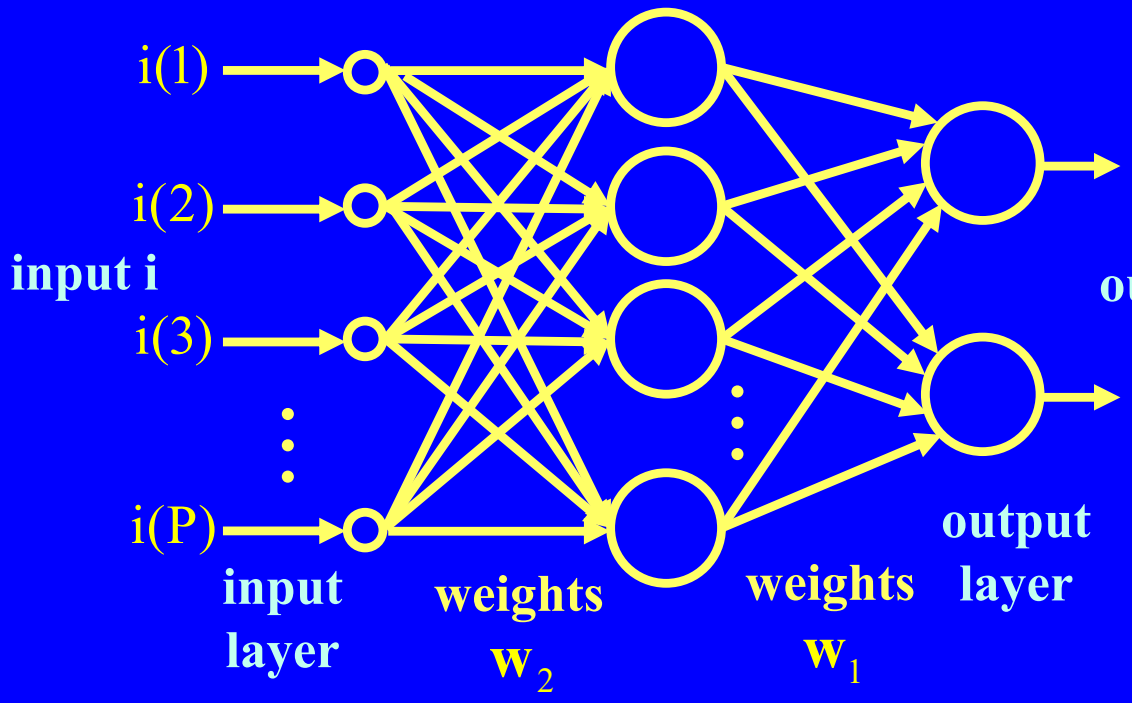

Who wants to learn Python?
Who wants to learn Math?
Who wants to become a data scientist?

ConvNets

- **Actually** an **old idea** from the 1990s generally attributed to **Yann Lecun.**
- The simple idea is to **limit the number of inputs** from **prior layers** that **each neuron receives**.
- This creates layers that are only **partially connected instead** of **fully connected**.

Revisit the Multilayer Perceptron

- Recall the **basic MLP** from earlier with a single hidden layer:



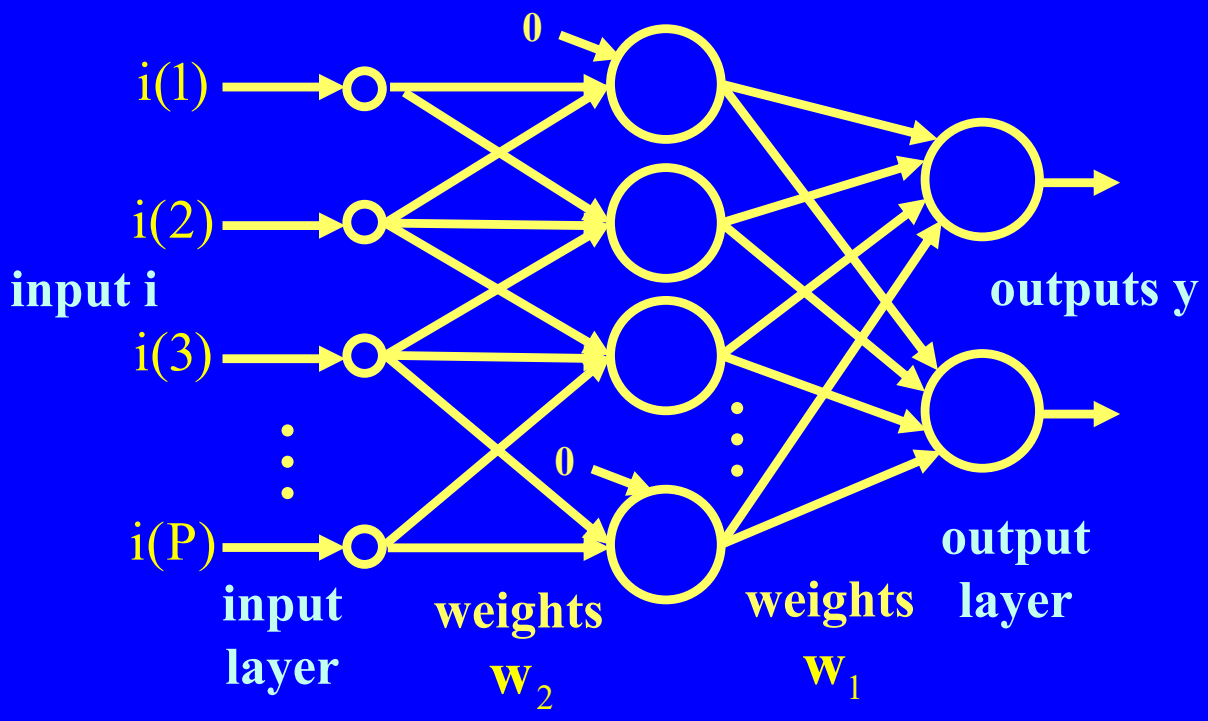
$$\phi\left[\sum_{r=1}^{R} w_q(r) i_q(r)\right]$$

- But let's feed it **image pixels,** instead of "handcrafted" features.

- For a **small** 1024 x 1024 ($P = 2^{20}$ pixels) **image,** the **number of weights** w_2 in the hidden layer is PQ, where Q is the number of nodes in the hidden layer. If P = Q, that's $2^{40} = 1.1 \times 10^{12} \approx$ **1.1 Trillion weights** to optimize via backprop!!!

Convolutional Layer Idea

- Suppose we only feed a **few image pixels** to each node:

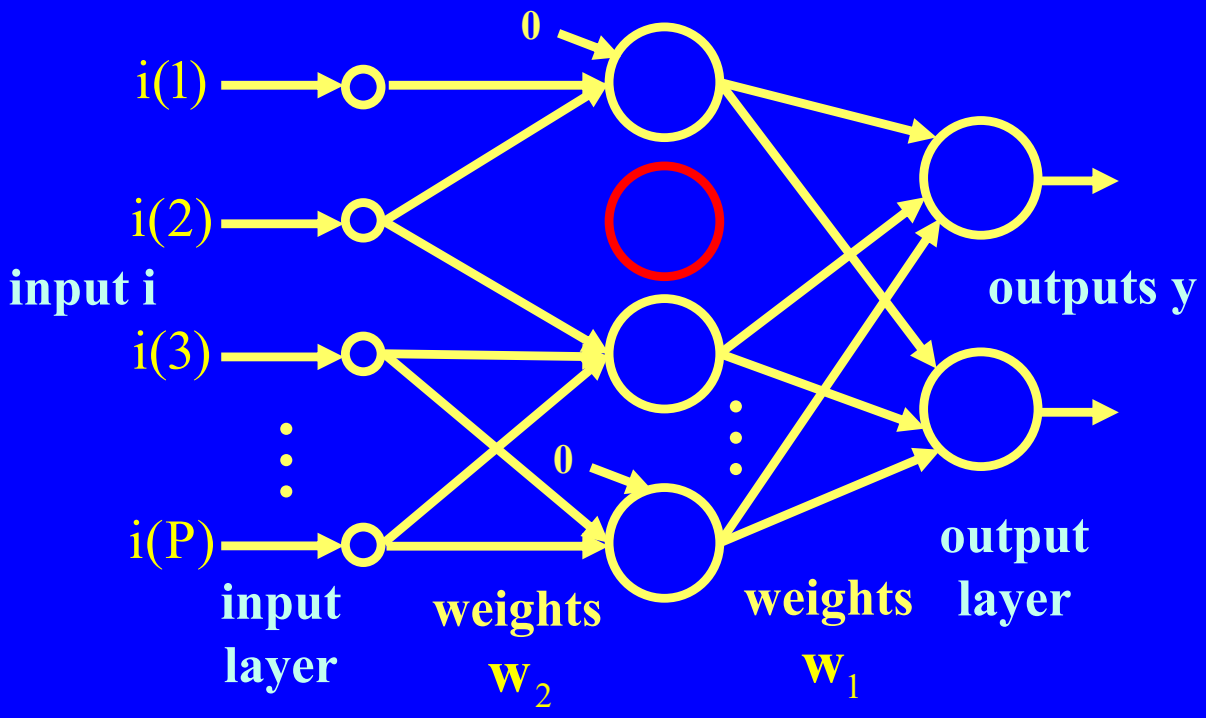


- Notice the **endpoint zero padding.**
- The **nodes** are **identical.**
- **Same number of nodes** as **inputs.**
- The **weight set** used for the hidden layer is called the **filter.**

- In fact suppose we only feed **3 pixel values** to each hidden node. Then if P = Q there are only **3P weights to optimize.**
- For the same image that's 3 x 2^{20} ≈ **3.1 Million** … lots better but still **many!**
- However, in a **convolution layer,** each group of weights are constrained to be identical. So in this example there are only **3 different weights** (+1 bias = 4).

Stride Length

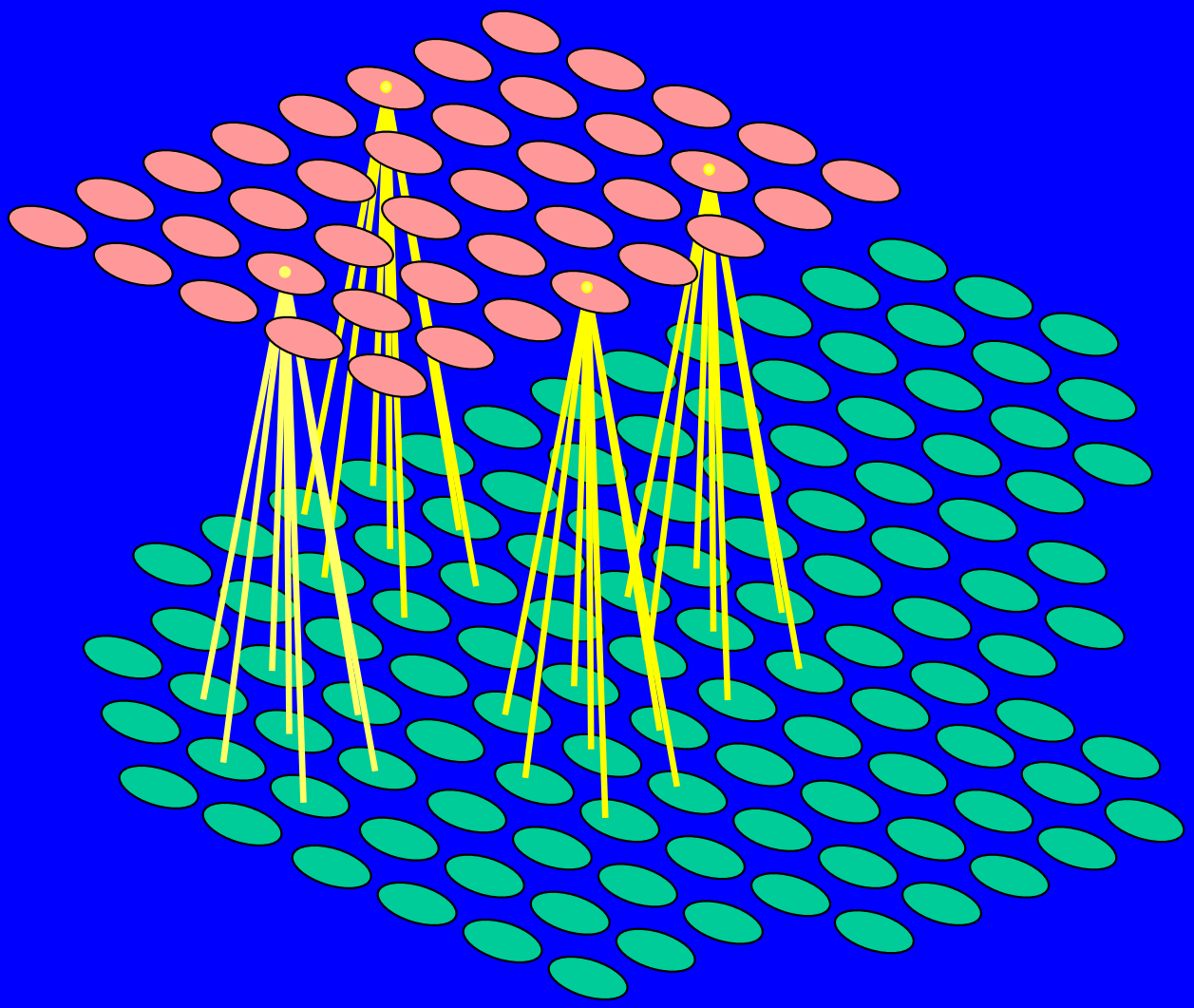
- Don't have to have as many filters as inputs. Can skip by a **fixed stride:**



- Effectively **subsampling** the full output. **Without subsampling**, **stride = 1.** In the above, **stride = 2.** Stride can be larger to **reduce computation.**

- As before **each node** includes a **nonlinear activation function.**

2-D Convolutional "Receptive Field"



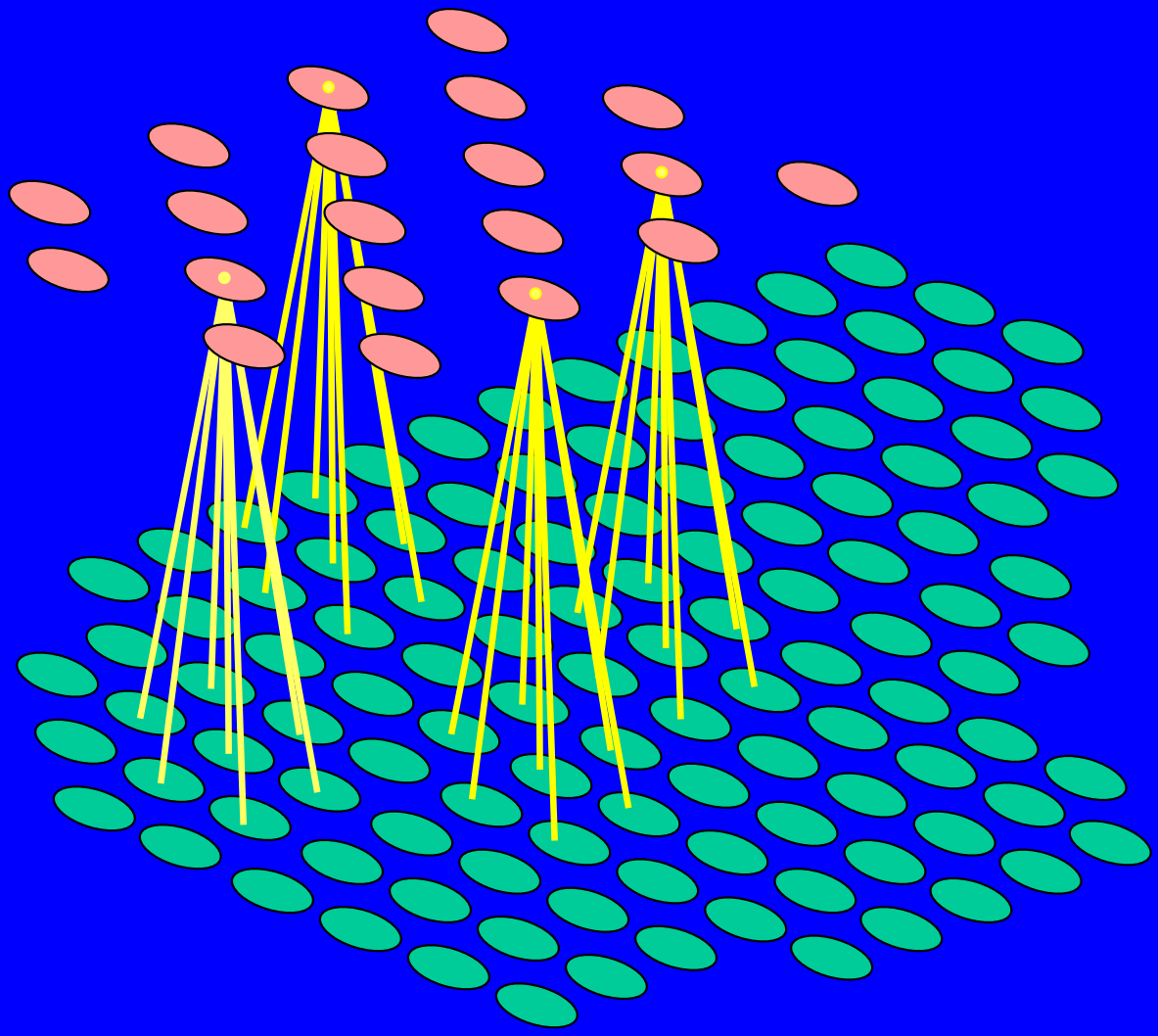
- **Every node** in hidden layer receives inputs from a **local neighborhood**.
- These are **weighted** and **summed**.
- The **weight set (filter) is identical for each node.**
- If stride = 1 **identical to 2D convolution** with the filter.

Input could be
- image pixel values

Similar to **feed-forward** from
- retinal sensors
- neurons in cortex to other neurons.

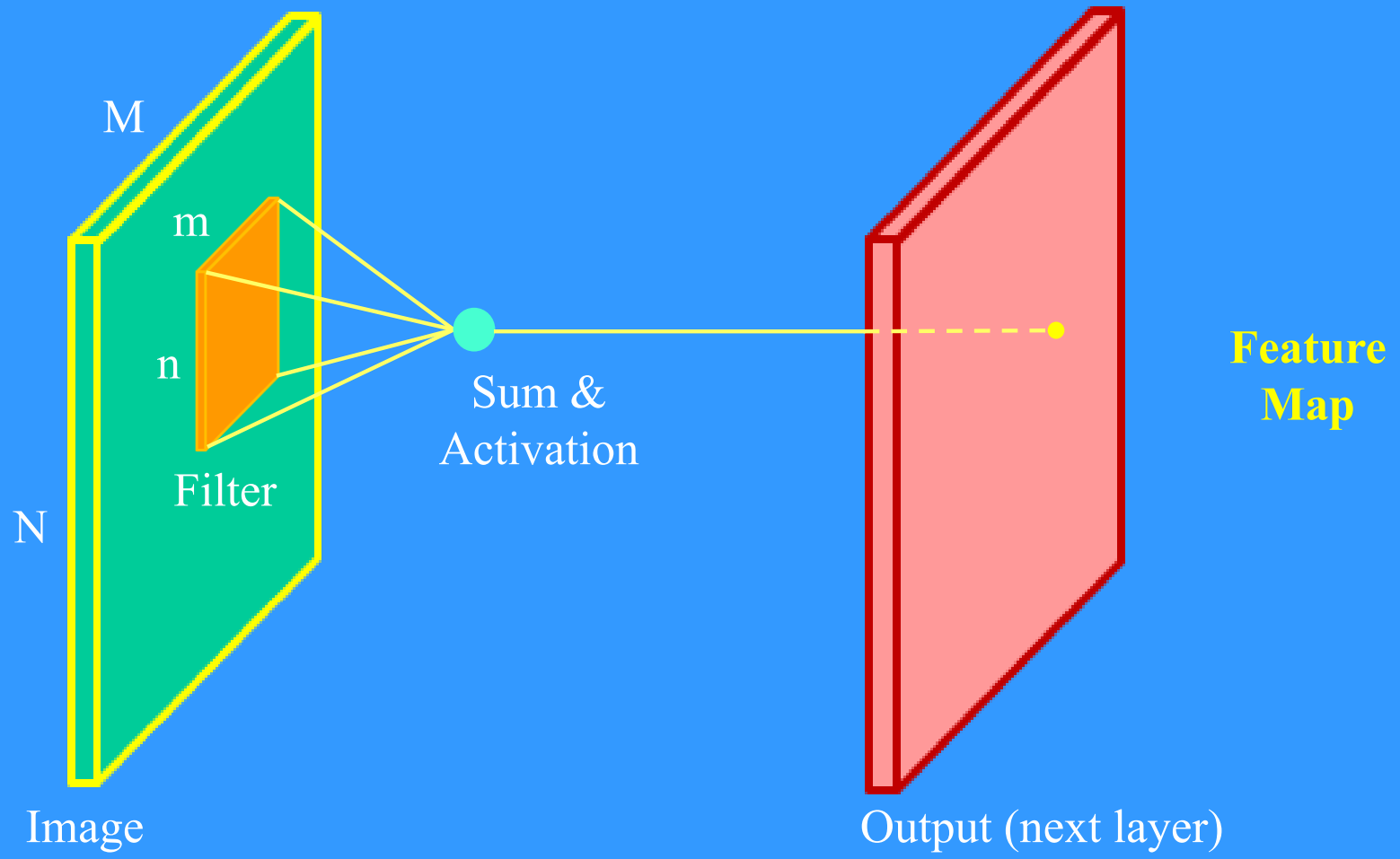
CNNs often called
"biologically inspired"

Stride of 2 (Most Common)



- In a 2-D image, if **stride = S,** then computation is **divided by S^2.**
- Usually best to **retain full resolution in the early layers** (we will soon stack these!) to **extract as much low level feature information** as possible.

Simpler Diagram

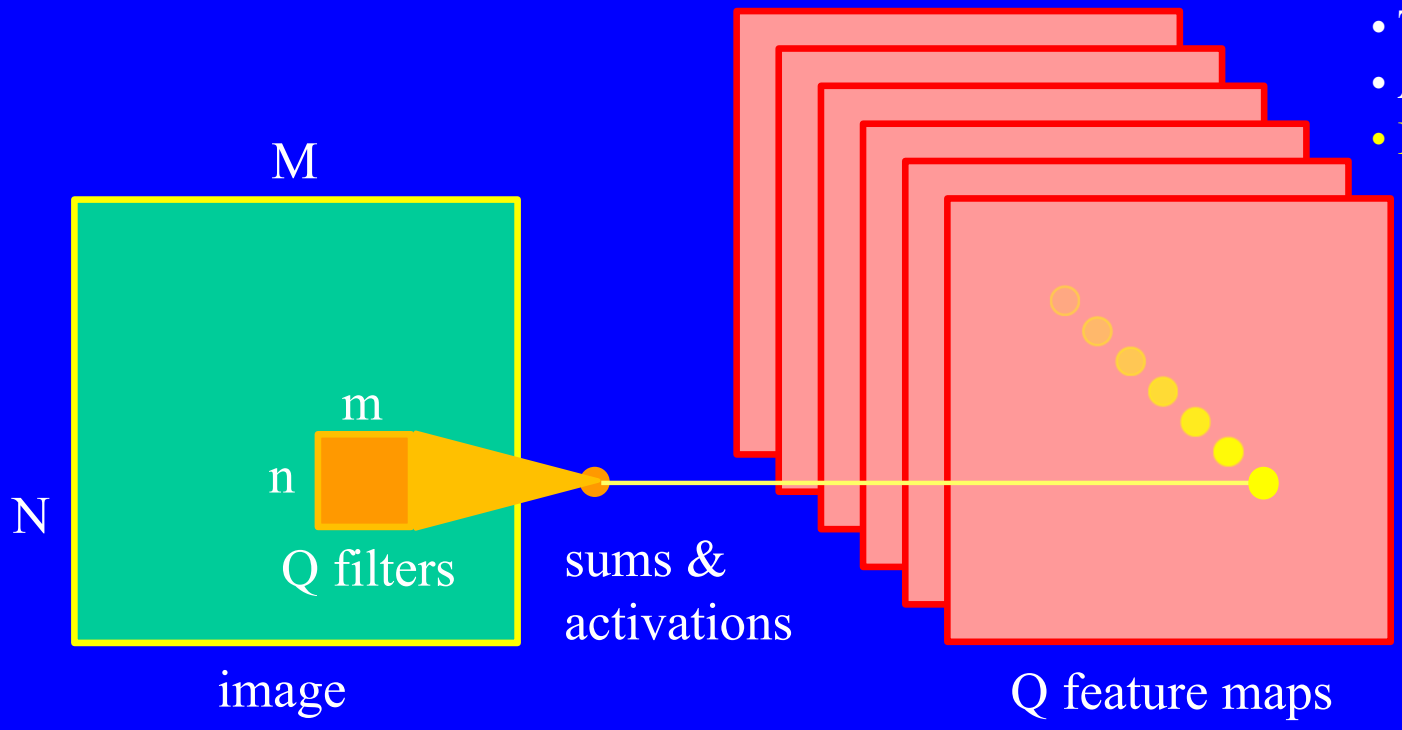


- Output is a **feature map.** Fed to the next layer.
- When **stacked,** early layers extract **low-level features.**
- **Zero padding** the input ensures the **feature map** will also be N x M (stride = 1)

Feature Maps

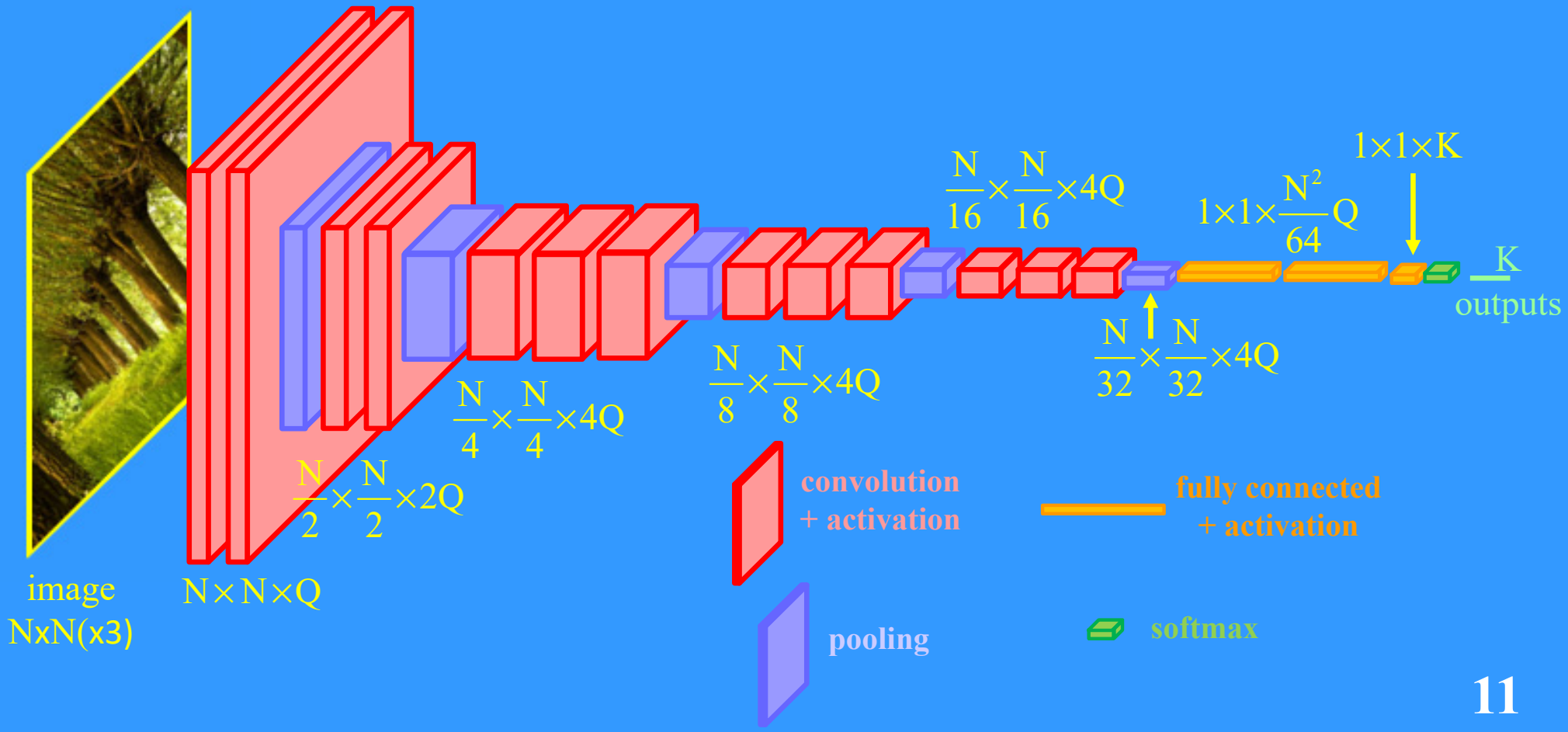
- A **convolutional layer** produced by a **fixed** filter only contains **one feature type** – say oriented in **some direction,** at **some scale.**
- The network may "decide" to learn **a specific 2D bandpass filter** (and in fact, they do!).
- **One feature map not enough**!
- Instead generate **multiple (Q) feature maps!**

- Here **Q = 6 feature maps.**
- Each generated with a **different learned filter.**
- This **multiplies** the **computation.**
- Also **multiplies** the **data volume.**
- **But a lot more info is obtained!**



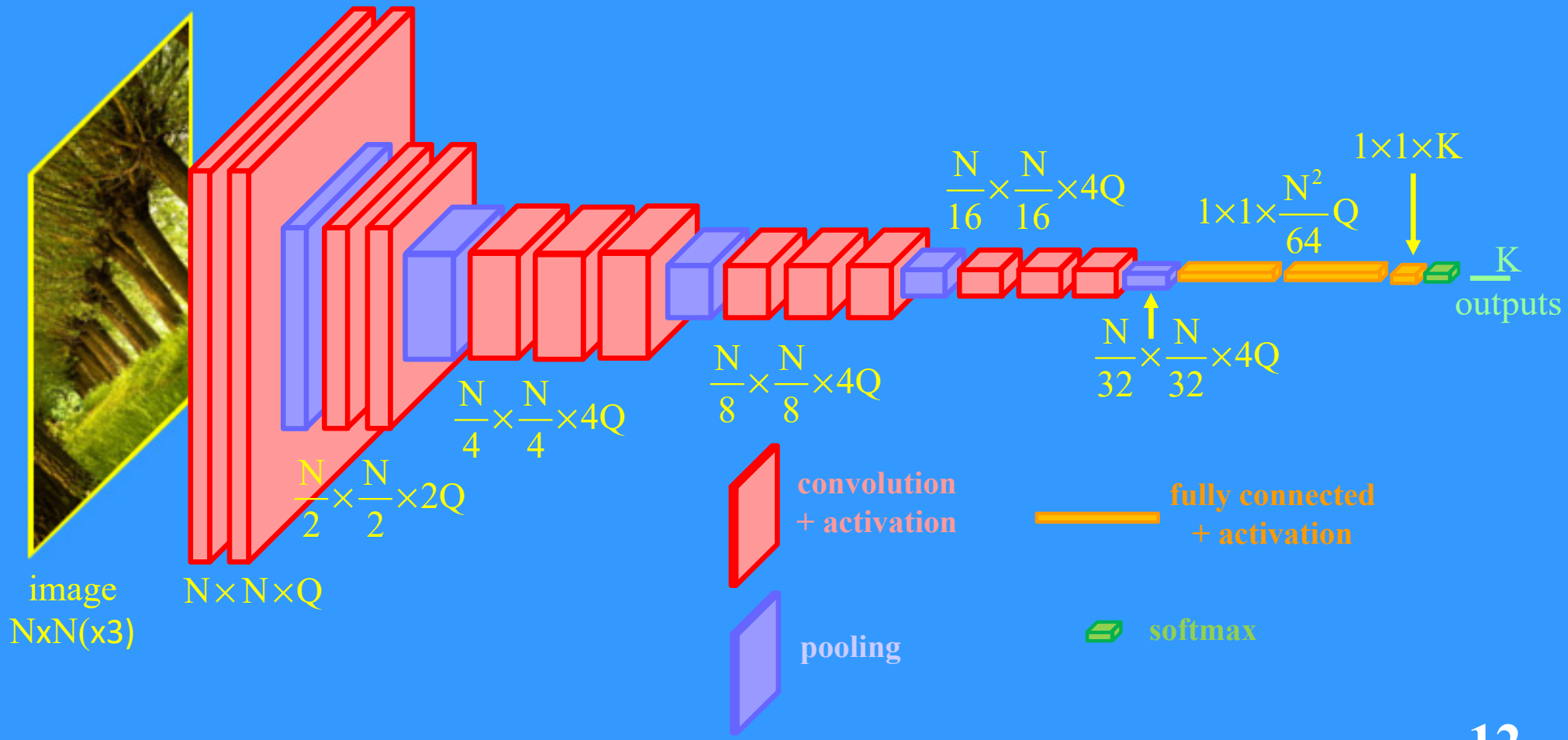
Stacking Layers

- For **multiple layers** simplify the notation, omitting flow lines, filters, and sum/activation nodes.
- **Example Network:** We stacked **multiple convolutional layers** (each with **multiple filters** and **activations**). We have also added some things. **Let's explain.**



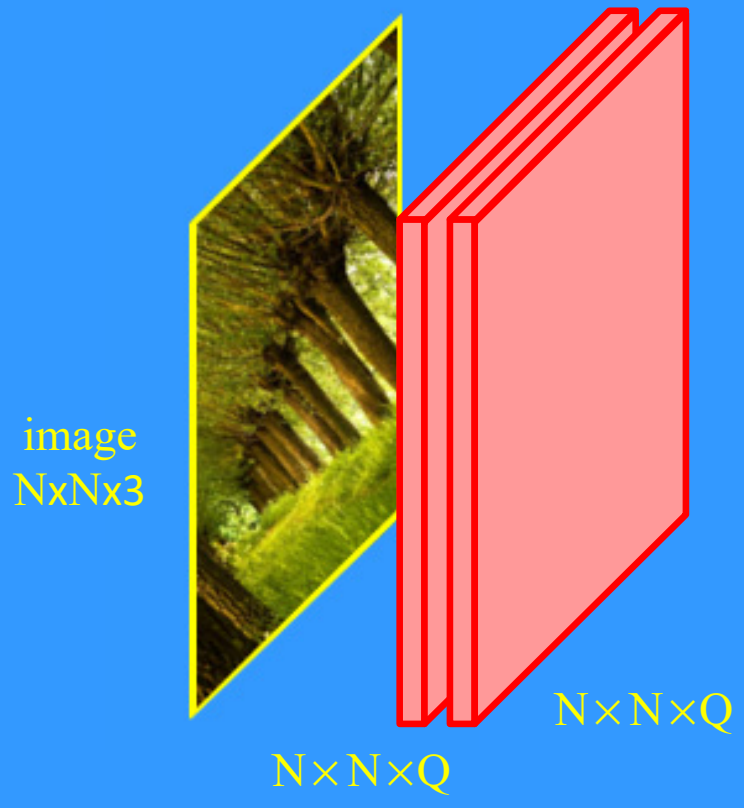
Multiple Channels

- The input is **color,** so has **3 color channels** (such as **RGB**).
- Hence **each filter does also:** if a filter is P x P, then it is really a 3 x P x P filter.
- Diagram is the same, but the **computation is three-fold.**



Stacked Convolutions

- **Stacking convolutions** allows for **small filter kernel** (weights)**.**
- Since **repeated** layers "reach" further to become **less local.**
- Similar to **repeated linear filtering** – except each layer has an **activation function.**



Activation Functions

- **Classic sigmoid functions** can be used – but for **modern Convnets / CNNs,** other activation functions are **more effective.**

- Training **multi-layered ConvNets** requires **backprop,** which uses **gradient descent.**

- **Convergence problems** arise, like the **vanishing gradient problem.** If the **gradient** approaches **zero**, **convergence slows** or **halts!**

Activation Functions

- The **most popular** is **"ReLu,"** or **Rectified Linear Unit:**

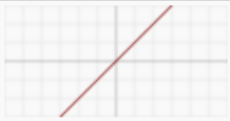
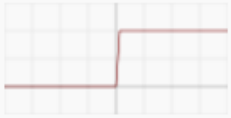
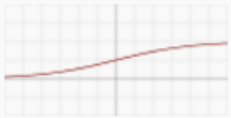
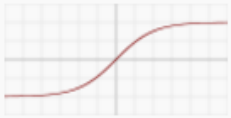
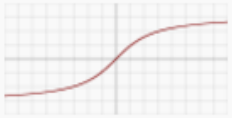
$$\phi(x)=\begin{cases}0\ ;\ x<0\\ x\ ;\ x\geq 0\end{cases}\qquad \phi'(x)=\begin{cases}0\ ;\ x<0\\ 1\ ;\ x\geq 0\end{cases}$$

  - These A.F.'s can speed up training many-fold.
  - Vanishing gradient reduced by not limiting the range (e.g., to [0, 1])

- **Another** popular one is the **Exponential Linear Unit** (**ELU**):

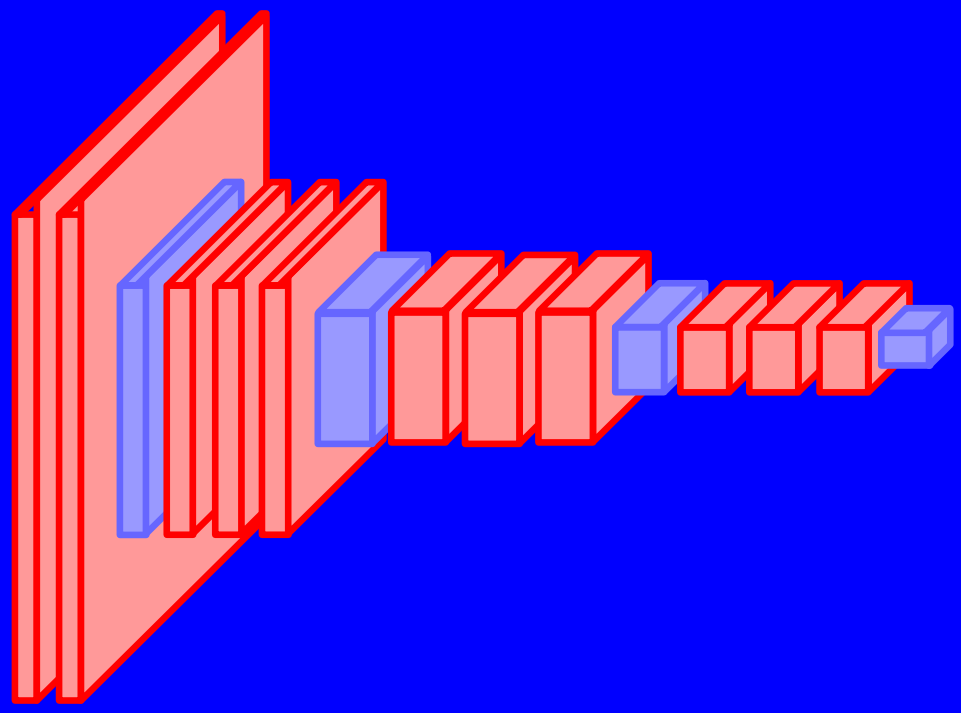
$$\phi(x)=\begin{cases}a(e^{x}-1)\ ;\ x<0\\ x\ ;\ x\geq 0\end{cases}\qquad \phi'(x)=\begin{cases}\phi(x)+a\ ;\ x<0\\ 1\ ;\ x\geq 0\end{cases}$$

More Activation Functions

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1-f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		'Leaky' ReLU $f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1+e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

Pooling Layers

- Another way to **reduce network complexity** that is **often used.**
- Another way to **expand the influence** of each **filter's receptive field** across the image.
- Appropriate **after the earliest layers** when the network focus is no longer on extracting local low-level features.
- Instead, the **network begins to build higher-level abstractions.**



Max and Average Pooling

- **Simple: Partition** the most recent feature map(s) into **non-overlapping** P x P blocks (Stride = P). Usually P = 2.
- **Max-Pool:** Replace each P x P block with a single value **max{block}:**

12	20	30	4
8	12	2	4
34	70	37	14
88	96	25	12

2 x 2 Max-Pool →

20	30
96	37

Tends to propagate sharp, sparse features, like edges and contours.

- **Ave-Pool:** Replace each P x P block with a single value **ave{block}:**

12	20	30	4
8	12	2	4
34	70	37	14
88	96	25	12

2 x 2 Ave-Pool →

13	10
72	22

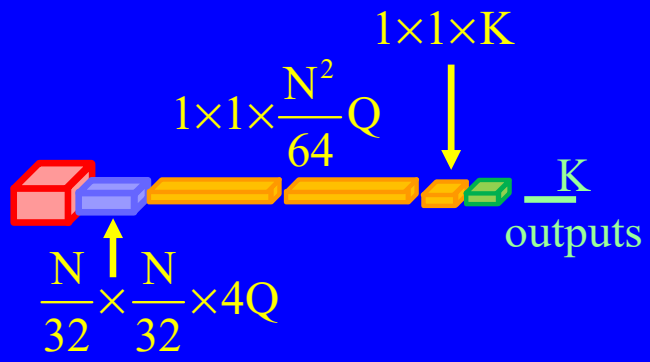
Tends to propagate a similar, smooth version of the prior layer.

- Both **help "over-fitting"** by providing an **abstracted form** of the **representation,** and reducing **positional dependence** of the representation.

Flattening and Fully Connected Layers

- The final stages of a CNN are often **fully connected networks or FCNs** (the same MLP's as earlier).

- Since the data has been **down-sampled** and **abstracted**, it is manageable with today's hardware.



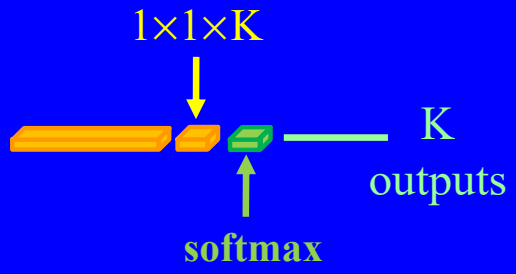
- The output of the **previous layer** is first **flattened** into a 1-D vector, to format it as input to the FCN. Everything is connected to everything, so **nothing** (such as spatiality) **is lost.**

- The image features have been converted into **higher-level abstractions,** on which the FC network **performs** the **final inferencing.**

Softmax

- The **final FC network** generates K outputs $\mathbf{y} = \{y(k); k = 1, \ldots, K\}$, corresponding to K classes in the problem to be solved.

- The **softmax** (or **softargmax***) function converts these into **probabilities** by mapping K non-normalized outputs of the **last FC network** to a **probability distribution** over the K predicted output classes.



- If for image **i**(j) (train or test) the inputs to softmax are $y_k(j)$, k = 1, …, K, the softmax outputs are

$$p_k(j) = \frac{e^{y_k(j)}}{\sum_{m=1}^{M} e^{y_m(j)}} \quad \text{for} \quad k = 1, \ldots, K.$$

- These **sum to one** and may be thought of as **probabilities.**

***Suppose** $y(k_0) = \max\{y(k); k = 1, \ldots, K\}$. Then **arg max** $\{y(k); k = 1, \ldots, K\} = \{\delta(k-k_0); k = 1, \ldots, K\} = \{0, 0, 0, \ldots, 0, \mathbf{1}, 0, \ldots, 0\}$.

Cross Entropy Loss Function

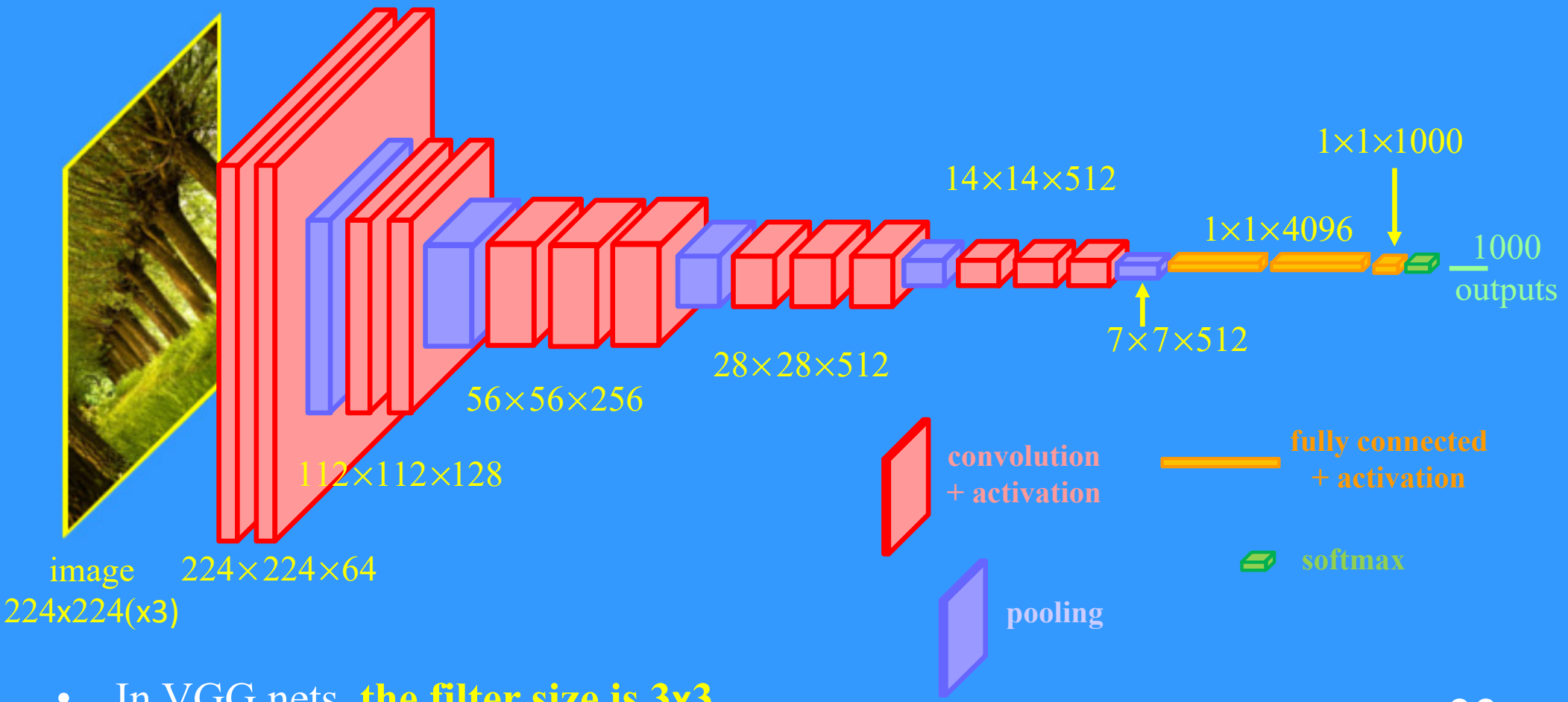
- Just to **complete** the **picture,** a **loss function** for **classification**…
- Assume **multiple classes:** Given **known ground truth** training labels $\mathbf{T} = \{t_j;\ 1 \le j \le J\}$ of training images $\mathbf{I} = \{\mathbf{i}(j);\ 1 \le j \le J\}$. **Each label** takes one of K values $t_j \in \{1, \ldots, K\}$ (multi-class). For example, in **images** these could index {dog, cat, bird …}.
- K = 2 is the **simplest,** e.g., **face / no face.**
- Let the **output (predicted class probabilities)** of the J training images $\mathbf{i}(j)$ be given by $p_k(j)$, k = 1, …, K.
- The **cross-entropy loss** (**log-loss**) b/w **labels** and **predictions** is

$$CE = -\frac{1}{J}\sum_{j=1}^{J}\sum_{k=1}^{K}\delta(t_j, k)\log p_k(j)$$

- A **good** way to **generate** the **probabilities** $p_k(j)$ is **softmax.**

VGG-16

- You have learned **all of the elements** of a very famous **deep learning network** or **DLN** called **VGG-16.**
- Designed for **1000-object classification,** has done well in many image analysis contests and remains **very popular.**
- **Many variations** and **deeper** ones, but these are the original "hyperparameters":



- In VGG nets, **the filter size is 3x3**

Let's Make a 'Dog vs Cat Classifier' Using a VGG-7

VGG-7 Classifier

- Train a VGG-7 from scratch. Not a heavy model (about 2.6 million parameters). Reshape images to 224x224.

- Create a CNN having **7 learnable layers** (2.6M learnable parameters)
  - **First conv layer:** 3 (RGB) input channels, 32 224x224 output channels, 3x3 filters (3 x 32 x 3x3 + 32 = 896), denote as **conv(3, 32, 3x3)**
  - **Second conv layer:** 32 input channels, 64 224x224 output channels, 3x3 filters (32 x 64 x 3x3 + 64 = 18,496), denote as **conv(32, 64, 3x3)**
  - **Third conv layer:** 64 input channels, 128 112x112 output channels, 3x3 filters (64 x 128 x 3x3 + 128 = 73,856), denote as **conv(64, 128, 3x3)**
  - **Fourth conv layer:** 128 input channels, 128 112x112 output channels, 3x3 filters (128 x 128 x 3x3 + 128 = 147,584), denote as **conv(128, 128, 3x3)**
  - **Fifth conv layer:** 128 input channels, 256 56x56 output channels, 3x3 filters (128 x 256 x 3x3 + 128 = 295,168), denote as **conv(128, 256, 3x3)**
  - **First fully connected layer:** 256 4x4 channel inputs, 512 outputs (256 x 4x4 x 512 + 512) = 2,097,664), denote as **linear(256x4x4 → 512)**
  - **Second fully connected layer:** 512 inputs, 1 outputs (512 x 1 + 1 = 513), denote as **linear(512 → 1)**

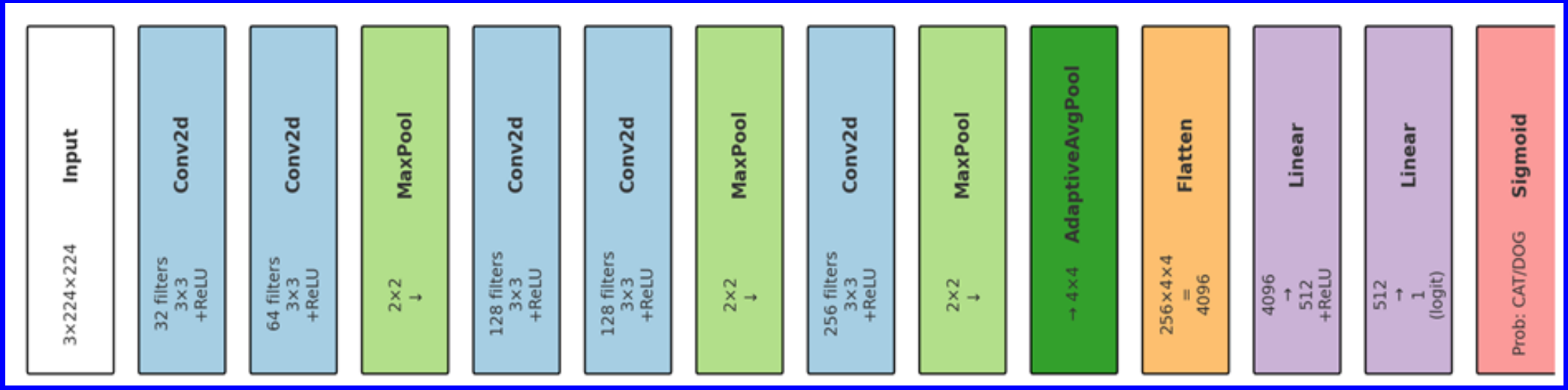
Total parameter count: 896+18,496+73,856+147,584+295,168+2,097,664+513 = **2,634,167**

VGG-7 Classifier

- Used **2x2 Max Pooling** after conv layers 2, 4, and 5, followed by Global Average pooling to 4x4, which feeds FC layers.

- **ReLu activations** used after every conv layer and the first FC layer. **Sigmoid activation** / probability mapping $1/(1-e^{-x})$ yields final output. Positive for 'DOG" = 1, for 'CAT' = 0.

- Binary **cross-entropy loss,** trained over **10 epochs.**



Dog / Cat Datasets

- Trained on the **Oxford Pet Dataset** which is here. The portion we used is just the dog and cat images for this binary classification problem, and these you can find here. This includes 2400 cat images and 4990 dog images. **Possible bias!**
- A set of 100 independent dog and cat images were borrowed from **Google Open Images,** found here.
- Images are .jpg. For those interested, the Oxford Dataset also defines **37 breeds** (25 dog, 12 cat), with **200 images / breed.**

Example Oxford Dog / Cat Images


















The dog images

The cat images

Deep Learning

Deep Learning

- We have already arrived at **Deep Learning**! VGG-16 is a classical Deep Learning neural net.

- Once it was regarded as **"very deep,"** but not so now.

- Notice in the diagram that there are **no "handcrafted" features** extracted to feed the network. It is an **"end-to-end" (possibly "from scratch")** design, meaning just pixels fed at one end, result at the other.

- Most **Deep Learning Networks** (DLNs) are just **ConvNets,** which have been known about since the 1980s. So why the "hubbub"?

ImageNet Contest

- In the early 2000s, neural nets were **still shallow** (too much computation) and still fed just a **few extracted features.**
- This changed when **Geoffrey Hinton** and his students Alex Krizhevsky and Ilya Sutskever published a paper entitled

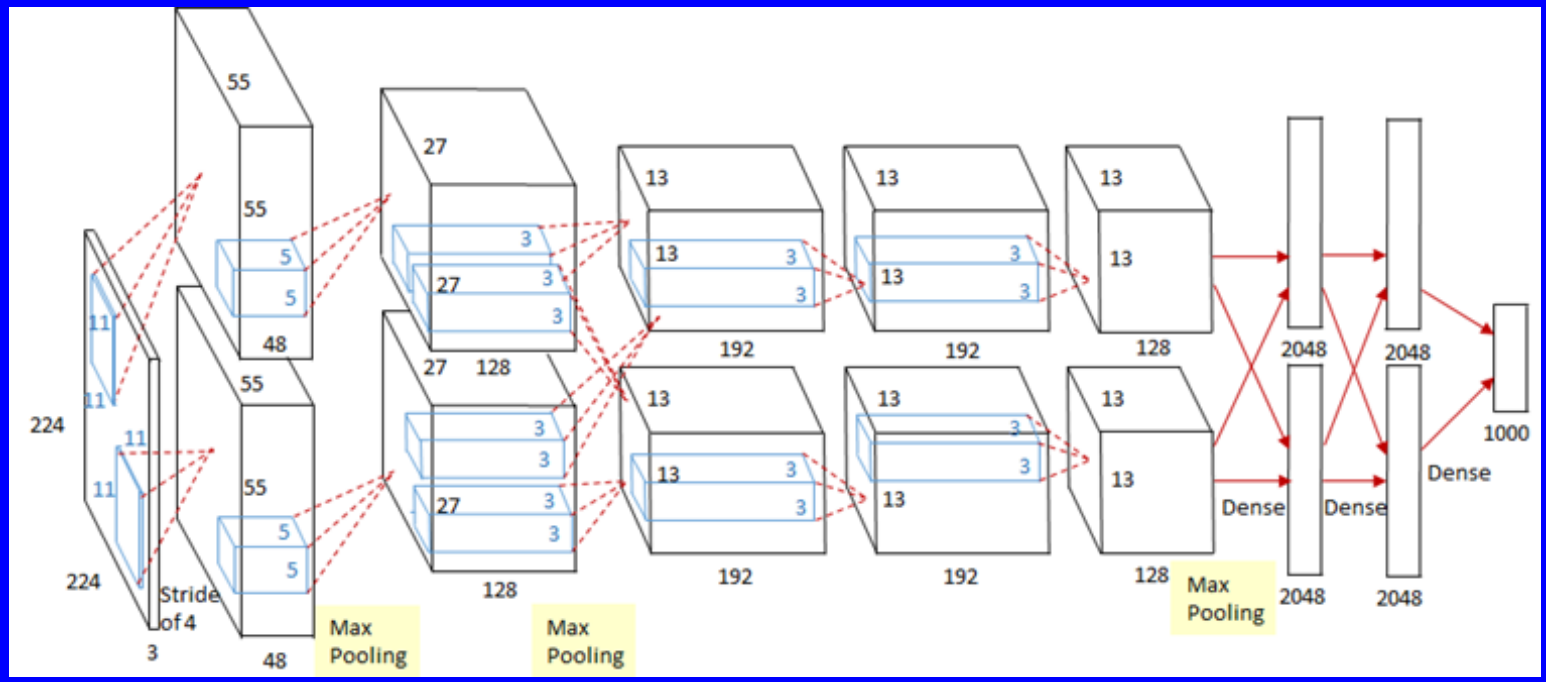
"ImageNet Classification with Deep Convolutional Neural Networks"

- In 2012 they showed that an end-to-end **8-layer convnet** (5 CNN and 3 FC) **significantly outperformed** all **computer vision** image classifiers on the standard ultimate testbed, **"ImageNet."**
- **ImageNet** (at the time) contained 15 million human-labeled* images having >20,000 class labels. They trained and tested on a subset of **1.2 Million images** having **1000 class labels.**
- They accomplished this by adopting highly efficient **Graphical Processing Units (GPUs)** originally designed for graphics/image manipulation.

***Labeled using Amazon Turk crowdsource platform**

AlexNet

- Here is the DLN they used (from their paper):



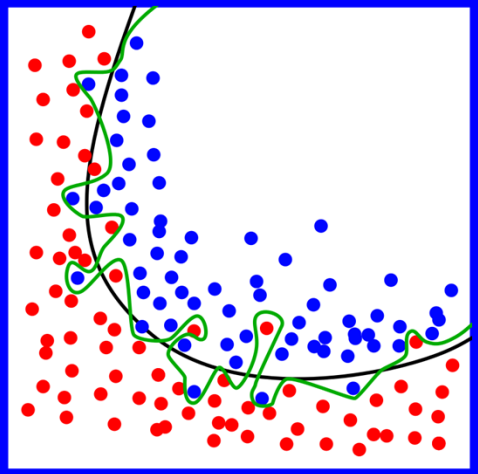
224x224(x3) 27×27×256 13×13×384 4096×1

55×55×96 13×13×384 13×13×256 4096×1

- **Details**
  - 650,000 neurons and **60 million parameters**
  - **ReLu activation** to avoid "overfitting"
  - Stride of 4 at input (input subsampling)
  - **Maxpooling** of layers 1, 2 and 5
  - 1000-way **softmax** at output
  - **Response normalization** to [0, 1] after 1, 2, and 5
  - New method called **"Drop-Out"** to avoid **overfitting**
  - **"Data Augmented"** by also training on **randomly shifted & horizontally flipped images**

Overfitting

- A great problem is the **lack of data** needed to train large networks and **avoiding overfitting.**
- **Overfitting** occurs when a network learns a model that is **too close** to the training data, and therefore **does not generalize** well to new data.
- Usually from **too-little data** (vs #network parameters) or training for **too long.**
- **Some methods** of handling:
  - **Data augmentation**
  - **Early stopping**
  - **Dropout**
  - **Batch normalization**
  - **Regularization**



Blue and Red are training data in two classes. Green curve is overfitted. Black is "regularized" to avoid that.

Handling Overfitting

- There is **no simple rule of thumb** regarding **how much data is needed** to train a network with P parameters (although sometimes say 1/10th #parameters at least).
- **Data Augmentation.** Increasing the amount of data artificially (when enough real data is not available). This involves reusing images by flipping, rotating, scaling, and adding noise to create "new" images. Care is needed but it works!
- **Early stopping.** A lot of theory, but usually empirical / *ad hoc* in application. Simple stop the training when some measure of convergence is observed, typically.
- **Dropout.** Idea: Neighboring neurons rely on each other too much, forming complex co-adaptations that can overfit, especially in parameter-heavy FC layers. **Method:** For each input, randomly fix P% of neurons to zero. In AlexNet, P = 50.

Batch Normalization

- **Batch normalization.** Extends the idea of **input normalization** which we discussed briefly in context of MLPs.
- Suppose the input is $i_0(1), i_0(2), \ldots, i_0(P)$. Then let

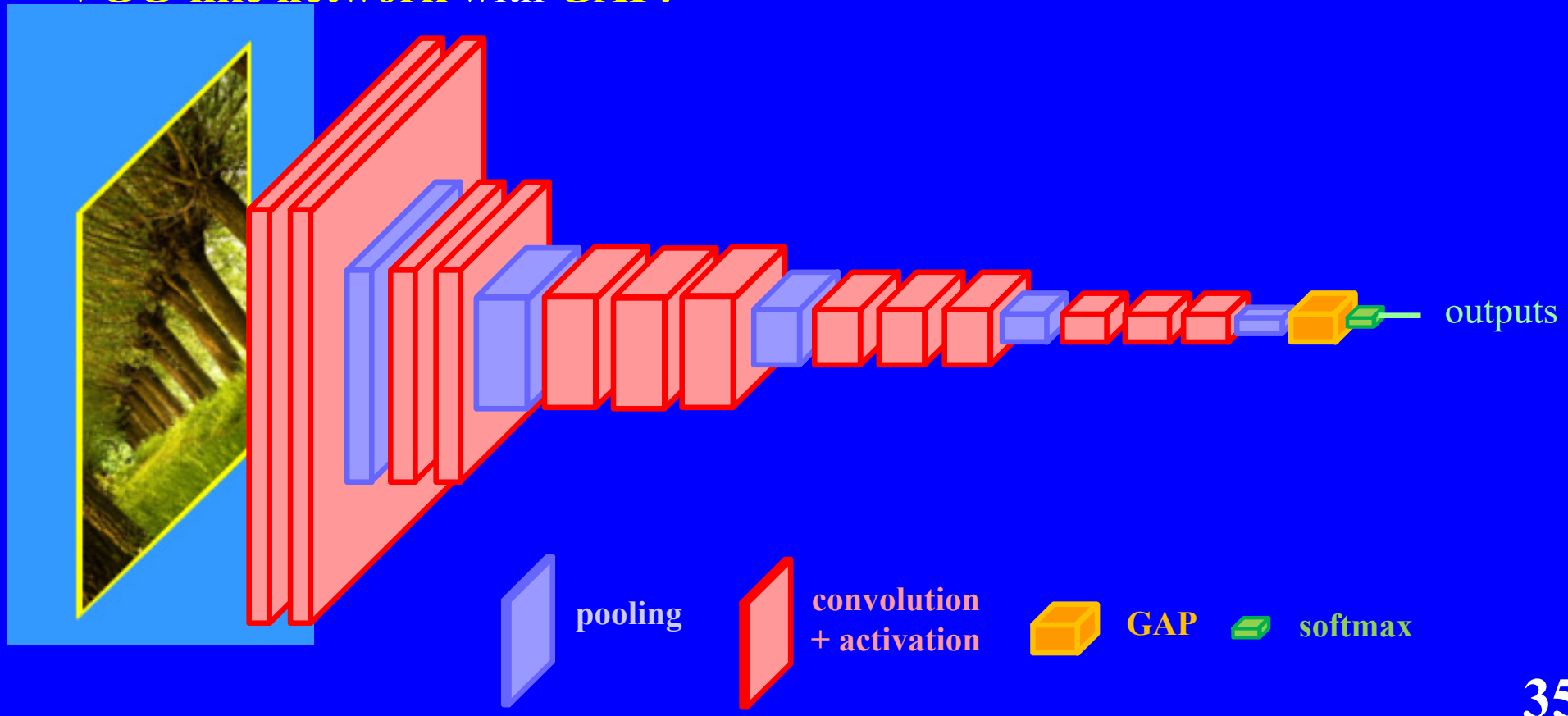
$$\mu_0 = \frac{1}{P}\sum_{p=1}^{P} i_0(p) \qquad \sigma_0^2 = \frac{1}{P}\sum_{p=1}^{P} (i_0(p) - \mu_0)^2$$

- **Input normalization** is training and testing on $\hat{i}_0(p) = \dfrac{i_0(p) - \mu_0}{\sqrt{\sigma_0^2 + \varepsilon}}$

- In **batch normalization,** this is instead applied as follows
  - Breaking each epoch into small "batches" which are optimized over
  - Normalizing all the feature map values in each batch
- **Many benefits:**
  - Improves convergence
  - Reduces overfitting / improves generalization
  - Greatly reduces vanishing gradient problem
  - Often dropout not needed

Global Average Pooling (GAP)

- **Much of overfitting** occurs in the **parameter-heavy FC layers.** GAP replaces them in a simple way by simply **globally averaging each final feature map.**
- Then **input** these responses **to softmax.**
- **Greatly reduces overfitting** while also greatly **reducing model size.**
- **VGG-like network** with **GAP:**



Learning Rate Optimization

- **Stochastic gradient descent** or SGD (re-ordering training data b/w epochs) is a cornerstone of deep learning. But we can try to **optimize SGD** by **modifying the learning rate** over iteration time.
- Often a **fixed learning rate** is used, or it might be altered once or more during training. This is often purely **heuristic guesswork** or observation.
- **Goal: Efficiently minimize loss** during training.
- The Adam Optimizer (**ADA**ptive **M**oment Estimation) uses ideas from **momentum** and **adaptive learning.**
  - Tracks the **averages of past gradients** (momentum) during backprop to help **smooth the updates.**
  - Tracks the **averages of past squared gradients** to adaptively **adjust the learning rate** of each individual network weight / bias (**bigger gradients -> reduced rate**).
  - While some authors dispute its optimality, it is **effective** and is a way to **speed training** in a **principled** and **automatic** way.
  - For a detailed discussion of SGD and Adam, see Wikipedia: here.

Deep Image Regression Networks

Classification vs Regression

- **Computer vision** (big early application of DLNs) models often **classify** (what class of object does this image contain?)
- **Image processing** models often create **image-sized** results like **denoised images, compressed images, quality maps,** and so on.
- Many of these are "**image-to-image" transformations.**
- This generally involves **regression** instead of **classification.**
- In **regression** the goal is to **predict numerical values** (better pixels, distances/ranges, quality etc) from an **image input** (RGB, luminance, YUV, etc).

General Image Regression

- Assume a **large collection** of **"before"** and **"after"** images. Call these **I** = $\{I_1, I_2, \ldots, I_N\}$ and **J** = $\{J_1, J_2, \ldots, J_N\}$. N could be in the **millions.**

- Most often each "after" image J_n is a **changed version** of a "before" image I_n, which are assumed to have a **desirable appearance.**

- The "after" images are the **result of processing** (intentional or otherwise), such as:
  - **Noise added**
  - **Compression by an algorithm**
  - **Blur occurs**
  - **Smoke, fog, or dust appear**
  - **Pieces of the image are lost or cropped**
  - **Image is dark or night falls**
  - **Color is lost or distorted**

- The "after" images can be other things like **distance, quality, style,** but this requires other inputs also (later).

Prediction

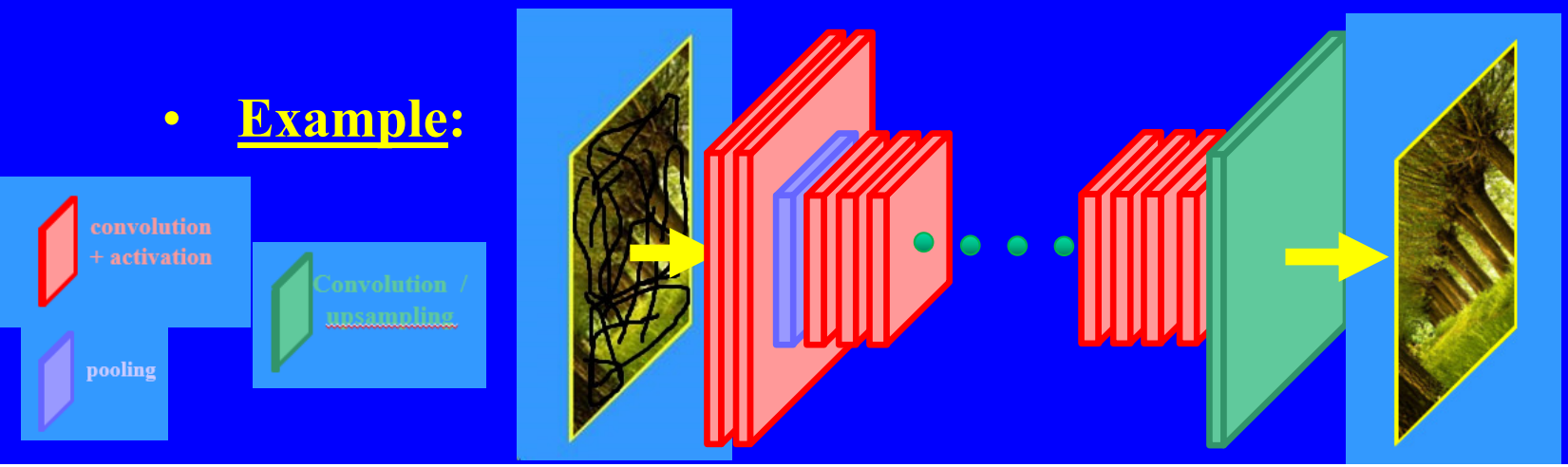
- **Basic idea: Predict the original image** (remove noise, blur, smoke, compression) using a **DLN.**

- **How: Train the DLN** on **many examples** of imperfect images, using **original images as labels.** Simplest: pixelwise labels.

- **Result: DLN** (hopefully) **learns to predict** the labels (the desired changed images). Not much thinking, but also **not so easy!**

- Since the result is **of the same size as** the **original** images, **little** or **no downsampling** used.

- **Example:**



Loss Functions

- Given **"before"** and **"after" (distortion or change)** images $\mathbf{I} = \{I_1, I_2, \ldots, I_P\}$ and $\mathbf{J} = \{J_1, J_2, \ldots, J_P\}$ as before.

- The **training network** produces **predictions** $F(J_n)$ of I_n. Optimizing the network may be approached by minimizing a loss like the MSE (or L2 loss):

$$MSE = \left\| F(J_n) - I_n \right\|_2 = \varepsilon_2 \left[F(J_n), I_n \right] = \sum_{i=1}^{N} \sum_{j=1}^{M} \left\{ F\left[J_n(i,j) \right] - I_n(i,j) \right\}^2$$

or (L1 loss):

$$MAE = \left\| F(J_n) - I_n \right\|_1 = \varepsilon_1 \left[F(J_n), I_n \right] = \sum_{i=1}^{N} \sum_{j=1}^{M} \left| F\left[J_n(i,j) \right] - I_n(i,j) \right|$$

- **Both are common.**
  - L2 is more sensitive to "outliers" (wrong labels) while L1 is robust against them.
  - L2 optimization is more stable and yields unique solutions. L1 less stable, and nonunique.
  - L1 optimization often is sparse, yielding weights or "codes" that are mostly zeros.

- When the **appearance** of the result matters, the **Structural Similarity Index (SSIM),** which is **differentiable,** is often used:

$$SSIM\left[F(J_n), I_n \right]$$

Residual Nets?

- Again assume **"before"** and **"after"** images **I** = $\{I_1, I_2, \ldots, I_N\}$ and **J** = $\{J_1, J_2, \ldots, J_N\}$. A **"residual network"** is a simple and intuitive idea.

- **Suppose: Instead** of learning predictions $F(J_n)$ of I_n, instead **learn to predict the residuals** $\mathbf{R(J_n) = I_n - J_n}$ b/w the "before" and "after" images. The **training residuals** are the **labels.**

- The **loss would be**

$$\text{RES-MSE} = \varepsilon_R\left[R(J_n)\right] = \sum_{i=1}^{N}\sum_{j=1}^{M}\left\{R(J_n) - \left[I_n(i,j) - J_n(i,j)\right]\right\}^2$$

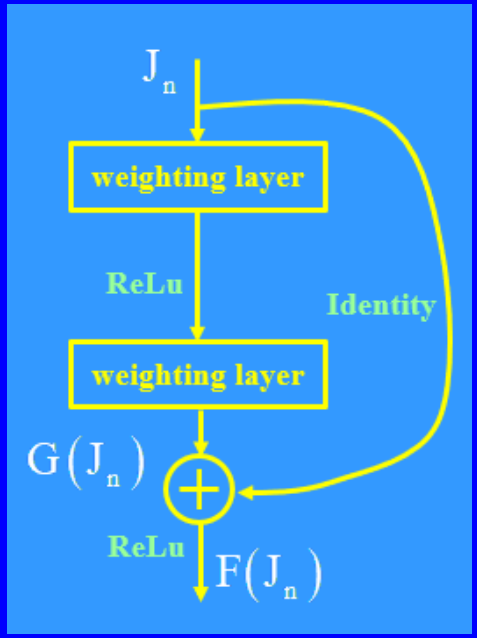
- **Residuals simpler, lower entropy** (clustered around 0), **easier to predict.**

- **Application Phase:** Ooops! The network is trained to **predict residuals,** which **we do not always have labels of (sometimes though)!**

ResNet

- **There is a way** to apply the idea! Let the network be trained on the **input** (J_n) and **output** (I_n) **training data.** The goal is still to produce a prediction $F(J_n) \approx I_n$, using a loss $\|F(J_n) - I_n\|$
- This network learns the **same mapping:**

Figure adapted from the famous paper "Deep Residual Learning for Image Recognition" by He, Zhang, Ren, and Sun



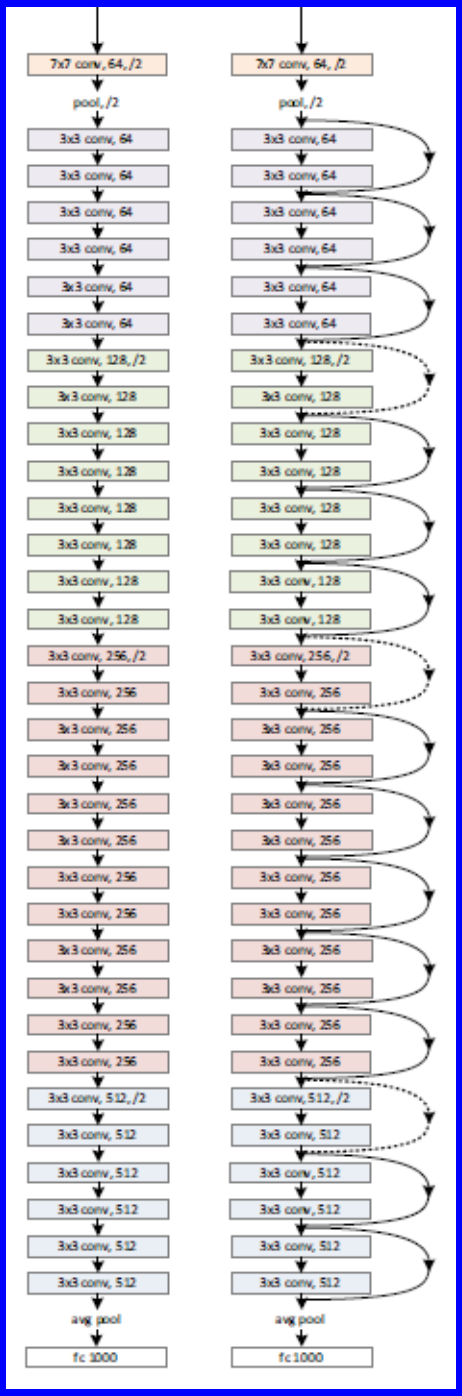
- However, the **weighting layers** instead actually **predict residuals**

$$G(J_n) = F(J_n) - J_n \approx I_n - J_n$$

- **The secret**: A **"skip connection"** whereby J_n **bypasses** one or more layers, then added before activation.

ResNet

- **ResNet** is an **extremely powerful** and **popular idea!**
- One of the **highest-citing papers ever!**
- However, **in practice** it is used **differently** for **even better results.**
- The **residual idea** can be applied **throughout the network** (every few layers) to **powerfully reduce overfitting** and **improve convergence.**
- The **ResNet idea** is used in **other network designs** that use different connectivities, scales, and skip connections, such as **DenseNet**, and **GoogLeNet/Inception.** These **keep evolving.**



34-layer (standard and ResNet)

ResNet

- **ResNet** has made possible **very, very deep networks** (hundreds of layers deep), which was not possible because of the **degradation problem.**
- The performance of **standard networks** decrease with great depth, because of **vanishing gradients,** poor convergence, **worse accuracy.**
- **Developers** now routinely use **"ResNet-50," "ResNet-101,"** and **"ResNet-152,"** for example.
- **ResNet models** have **won** nearly every **image detection, recognition, segmentation, classification, and regression contest!**

Transfer Learning

Transfer Learning

- The **biggest problem** in deep learning is obtaining **enough data.** For very deep networks, even when parameter-efficient, very large datasets are needed which are often not available.

- As we know they can be **trained end-to-end w/o feature computation.** Only pixels need be fed, if the network is **large enough** and the data volume is **large enough.** However, often there is **not nearly enough data.**

- However, a **property of DLNs** trained for image processing is their **remarkable generality.** This is very useful when there is **not enough data.**

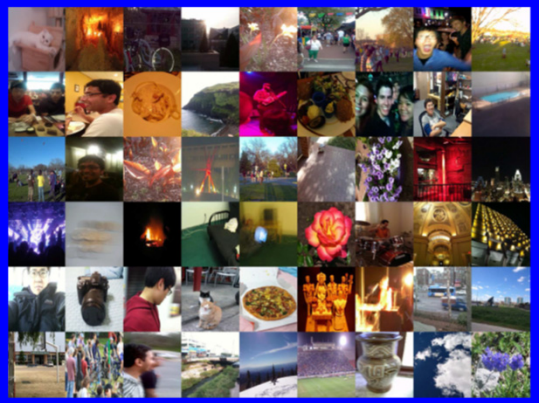
- These features can be **remarkably generic!** Can often use a trained network (with learned feature outputs) to **conduct another visual task.**

- This might involve **"fine tuning"** the **"pre-trained"** network and/or adding **additional layers** (or an SVC/SVR) that are **subsequently trained.**

- Called **TRANSFER LEARNING.**

Transfer Learning (Fine Tuning)

Source Task
(very large data, e.g. ImageNet)



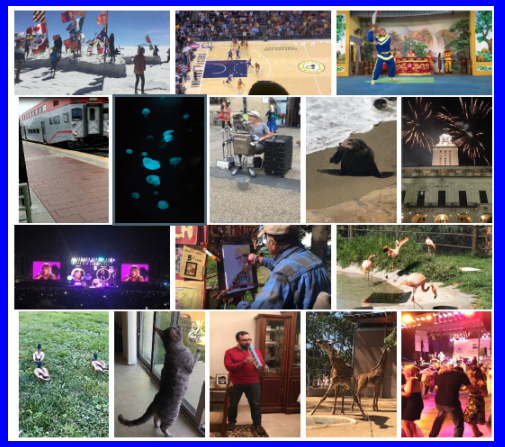
+ labels

pre-train

DLN

(Knowledge)

Target Task
(much less data)



+ new labels

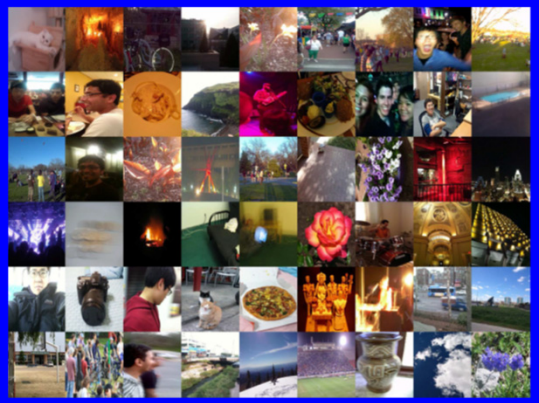
fine-tune

Pretrained DLN

predictions

Transfer Learning (New Layers)

Source Task
(very large data, e.g. ImageNet)



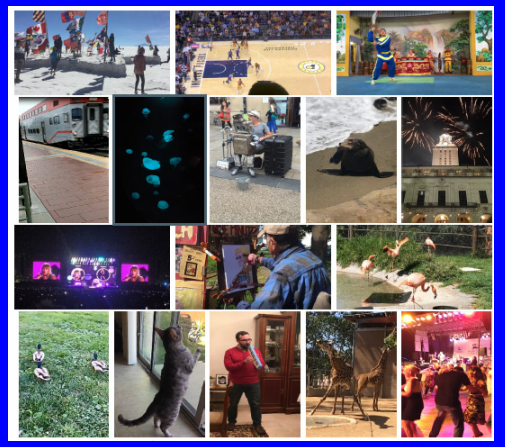
+ labels

pre-train

DLN

(Knowledge)

Target Task
(much less data)



+ new labels

fine-tune

Pretrained DLN

New Layers / SVR

predictions

Transfer Learning

- **Transfer learning** is one of the most common ways to address **most image processing / analysis tasks.**
- Most of which have **much more limited data** than ImageNet.
- **Typical** a network designer will begin with a ResNet-20, -50, 100, 150, Inception/GoogLeNet, Transformer, or **any other large network.**
- Some of these now have **many thousands of layers!**
- In **application examples** we will often use Transfer Learning.

DLN Training Environments

DLN Training Environments

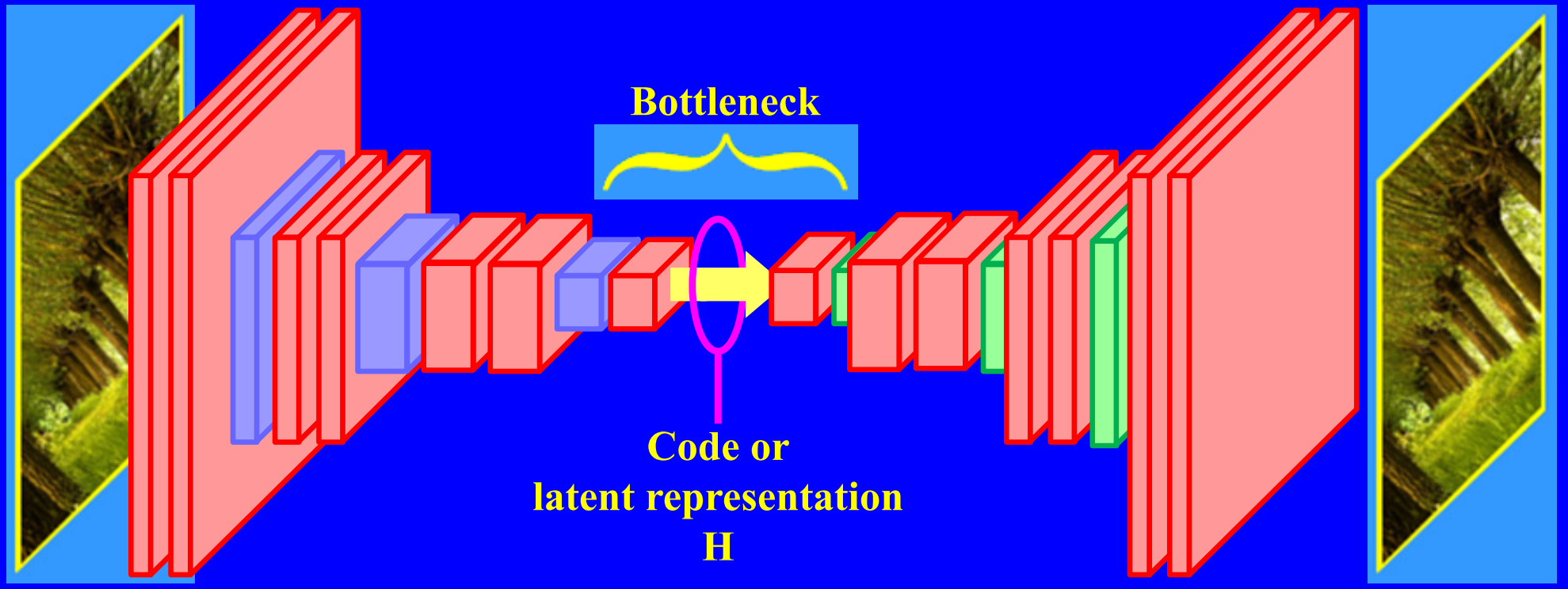
- There are **many programming environments** for training DLNs.
- Some of these have become **quite popular.**
- These include
  - **Tensorflow**
  - **Caffe**
  - **Keras**
  - **Pytorch**
  - **Matlab**
- Currently, Facebook's **Pytorch** is often **favored** amongst DLN R&D engineers.

Convolutional Autoencoders

Autoencoders

- An **autoencoder** is a special variety of neural network (or DLN).
- It is generally **unsupervised!**
- **Same architecture** as MLPs or DLNs, but they have **two parts.**
- The **first part** is usually **contractive:** multiple layers of **down sampling** to achieve **efficient representation** requiring less data / dimensionality.
- The **second part** is then **expansive:** multiple layers of **up sampling** – generally **mirroring** the front stage of layers.
- The **point of transition** where efficiency is achieved is the **"bottleneck."**

Convolutional Autoencoder



convolution + activation

pooling

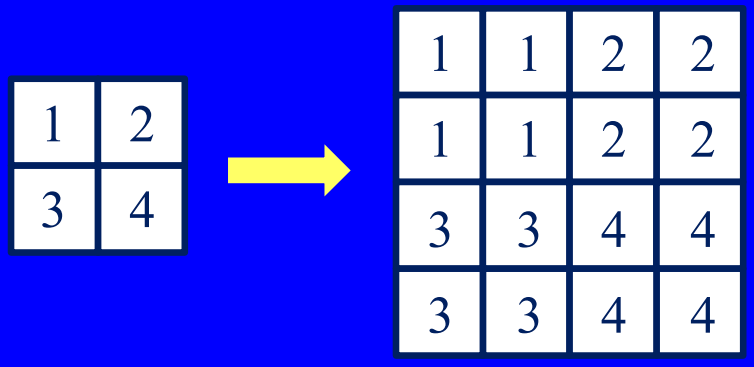
upsampling

Convolutional Autoencoders

- **Basic idea:** learn **efficient, compact representations** of images, from which they can be **"reconstructed."**
- **Down/up sampling** can be **limited** or **omitted** – it's not what defines an autoencoder.
- **Autoencoders** can be
  - **Overcomplete:** the number of features in the code **exceeds** the input data size
  - **Complete:** the number of features in the code is **the same as** the input data size
  - **Undercomplete:** the number of features in the code is **less than** the input data size
- **All three** are useful for **different tasks!** However, without further constraints the first two might learn the identity function (not so useful).
- All can be useful with **additional constraints** on the code.
- For efficiency, **undercomplete autoencoders** are of greater interest.

Up Sampling

- Methods of **up sampling** include
  - Nearest neighbor (pixel replication)
  - Max unpooling (requires recalling where corresp. max pool value came from)
  - Interpolation (bilinear, bicubic, **learned**)



Nearest Neighbor Unpooling

Regularized Autoencoders

- **Regularization**: **constraining the code** at the **bottleneck,** for various purposes.
- This includes **avoiding identity** function, making the represent efficient / **small code,** creating **high-information features,** etc.
- Given **a network loss function**

$$\varepsilon\left[F(J_n), I_n\right] = \sum_{i=1}^{N}\sum_{j=1}^{M}\left\{F[J_n(i,j)] - I_n(i,j)\right\}^2$$

where I_n are **input images** and $F(J_n)$ are corresponding **output images (predictions),** then modify the loss by appending a penalty on the bottleneck code (activations) H_n:

$$\varepsilon\left[F(J_n), I_n\right] + \lambda \cdot R(H_n).$$

Sparse Autoencoders

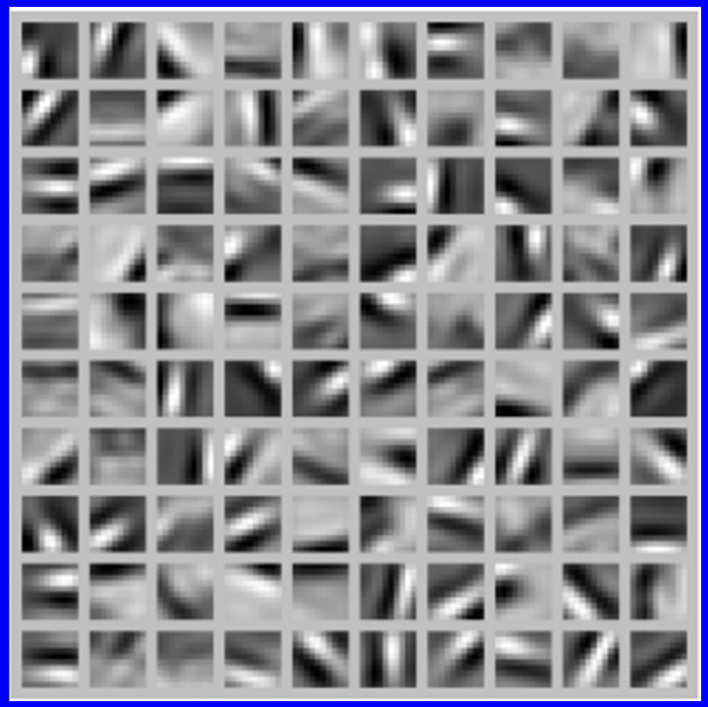
- **Sparse regularization** at the bottleneck is commonly used:

$$R(H_n) = \sum_{k=1}^{K} |H_n(k)|$$

  which is just the L1 norm of the **bottleneck activations.**

- **L1-optimization** creates **sparse codes** where **many zero activations are created, i.e., a sparse code** of **few non-zero** activations.
- Closely related to the **"lasso"** in statistics (L1-L2 optimization).
- Actually, this sparsity constraint can be applied **at any layer** of any DLN to **enforce sparse representations** (and they are).
- An obvious application of these ideas is **image compression.**

Typical Sparse Weights Visualized



- These look a lot like the **Gabor functions** and **derivatives of gaussians** which are good approximations to **the receptive fields** of **human cortical neurons.**

Case Study: Denoising Autoencoder (DAE)

Denoising Autoencoder (DAE)

- A **simple task** is to adapt an autoencoder to perform **denoising.**

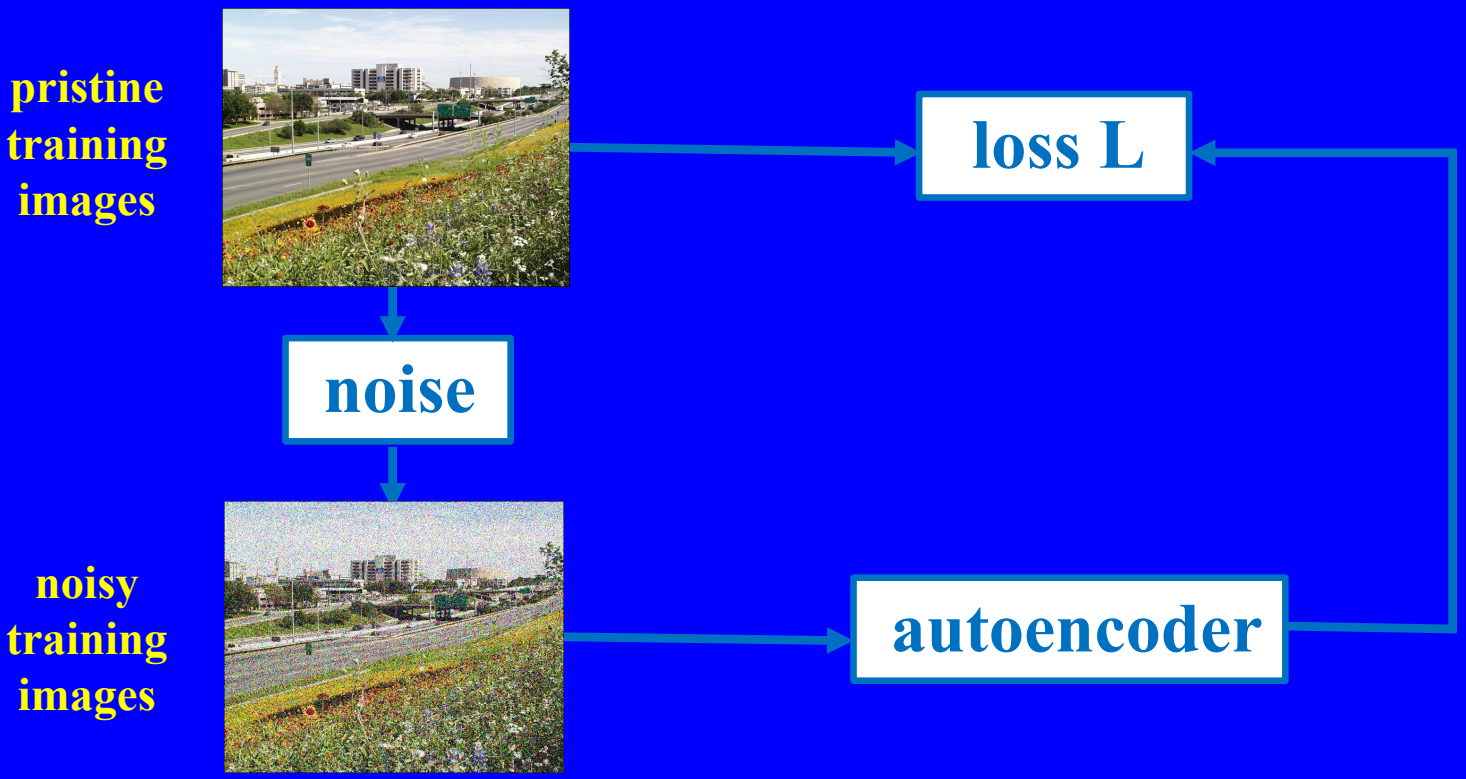
- **Instead of** optimizing output to be equal or similar to input, optimize it to equal a **noise-free version** of it.

- It's about **training:** Given **many noise-free** images $\{I_n\}$, make noisy ones, e.g., add gaussian noise $\hat{I}_n = I_n + N_n$ or other kind of noise, multiplicative, signal-dependent, simulated, etc.

- The autoencoder can be **sparse,** often producing **better results.**

Denoising Autoencoder

- **Training Phase:**



- The network **doesn't have to be** an **autoencoder,** of course. But if so, it is a **denoising autoencoder.**

Denoising Autoencoder

- **Application Phase:**

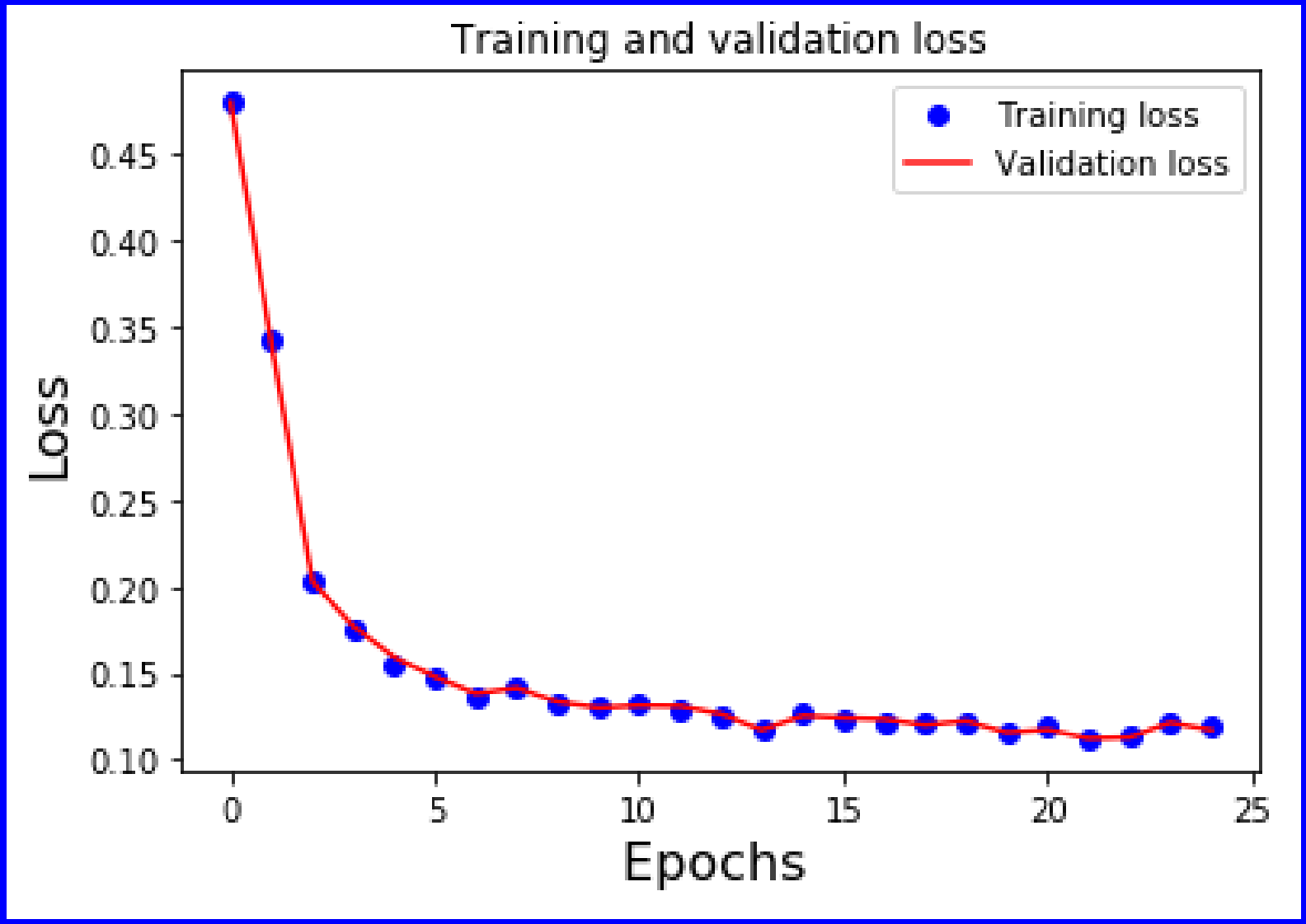
noisy images

"denoised" images (simulated idea)

- **Overcomplete autoencoders generally** do a better job of **denoising** than **undercomplete** ones.

Simple Autoencoder Denoising Example

- Simple example of **denoising images** with an **autoencoder.**
- It operates on **gray-scale 28x28 images.**
- **Architecture:**
  - **Encoder:** 2 conv layers, ReLu, 2x2 max pooling, 32 3x3 filters each layer (image: $28^2 = 784$ pixels; codesize: 7x7x32 = 1568)
  - **Decoder:** basically reverses. Same size corresponding filters, 2x2 upsampling)
  - **Training:**
  - Images (28x28x1) of handprinted numerals 0-9, using 55,000 training samples with added noise, 10,000 test samples.
  - Trained over 25 epochs using cross-entropy loss.



Note the **nice convergence** of both the **training study** and the **validation study.** This is **very good** since it means that the DAE is **generalizing** and **not overfitting.**

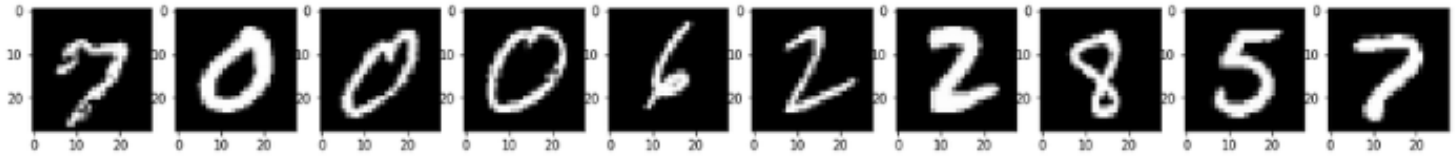


Figure: Original Images

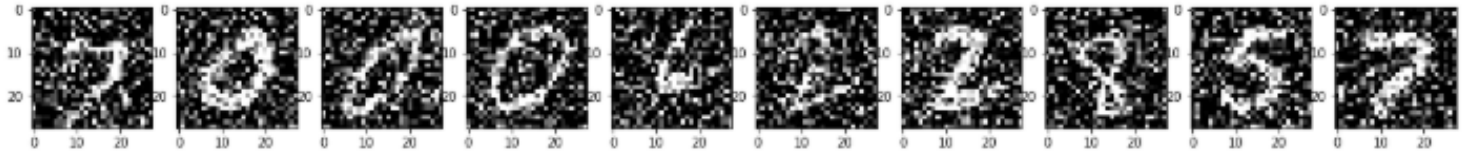


Figure: Images with Noise

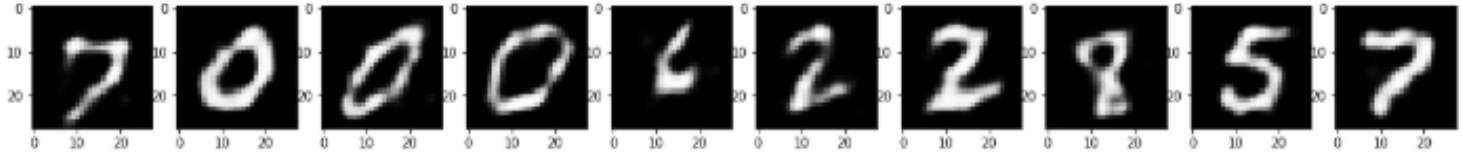


Figure: Reconstruction of Noisy Images

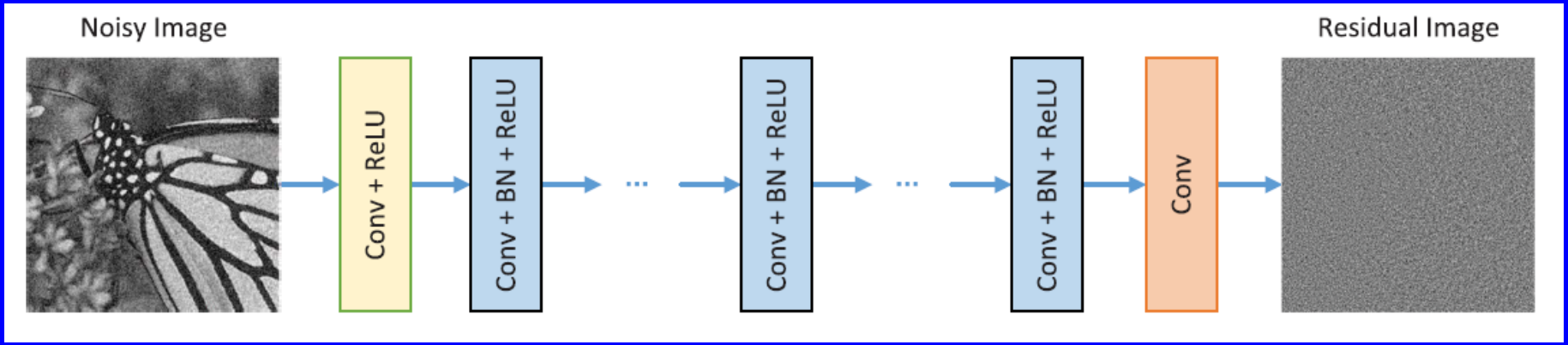
- Pretty **good results** for such a **simple** method, but simple DAEs **struggle on "harder" images.** More advanced DAEs ("stacked" autoencoders, etc) do better and **the field is changing very fast!**
- This simple implementation and two figures are from S. Malik's teaching website here (with **more details**).

Case Study: Deep Residual Denoiser

Deep Residual Denoiser

- Still a **competitive method.**

- **Basic architecture** (Denoising CNN, or DnCNN):



- It uses the **similar concept** as the **ResNet** to learn a **residual image** (the noise).

DnCNN Denoiser

- **Remember** the **RES-MSE** which we could **not use** (no **residuals** available)?

$$\text{RES-MSE} = \varepsilon_R\left[R(J_n)\right] = \sum_{i=1}^{N}\sum_{j=1}^{M}\left\{R(J_n) - \left[I_n(i,j) - J_n(i,j)\right]\right\}^2$$

- This time **we can have residuals!**

- I_n are **clean images,** and J_n are **noisy images.** Train to **estimate the residuals** (noise) $J_n - I_n$.

DnCNN Architecture

- Has **D layers.** In their design the **filter size is 3x3.** Observe that at layer d the effective span is **(2d+1)x(2d+1).**

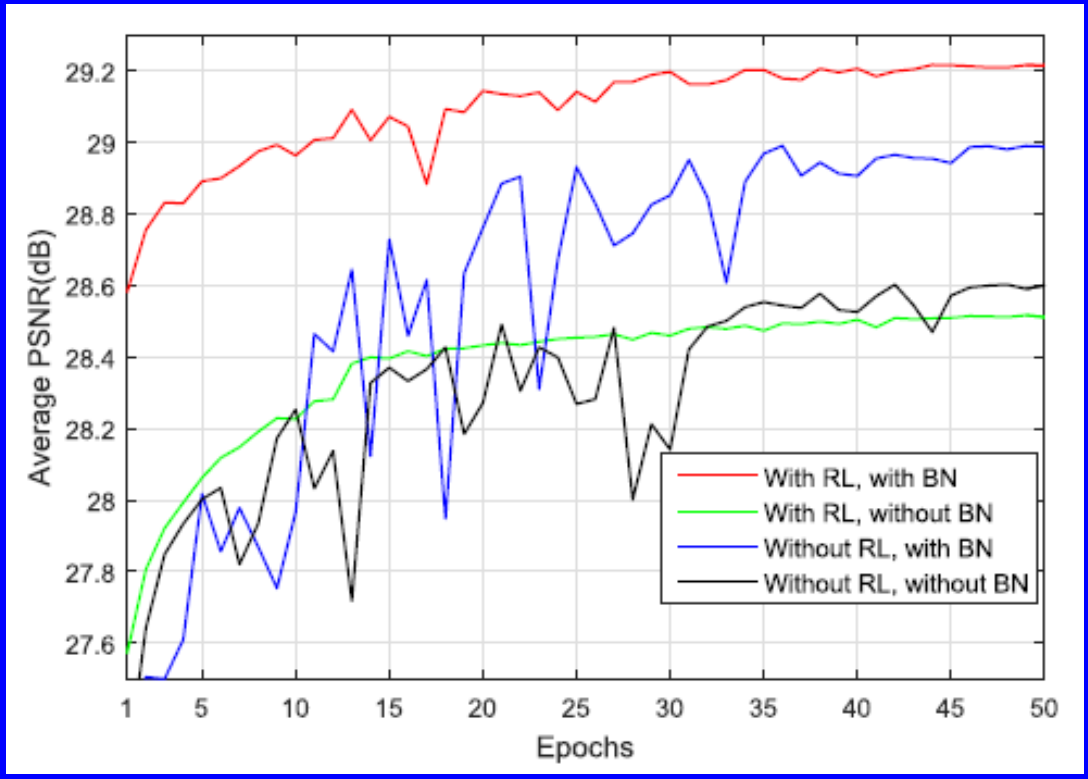
- They use **D = 17** (35x35 effective largest filters) to be comparable with **leading algorithms.**

- They use **ReLu** but **no pooling** / **downsampling.**

- **Filters:**
  - **Layer 1:** 64 filters (3x3xc, c = # colors)
  - **Layers 2 – (D-1):** 64 filters (3x3x64 w/batch normalization)
  - **Layer D:** c filters (3x3x64)

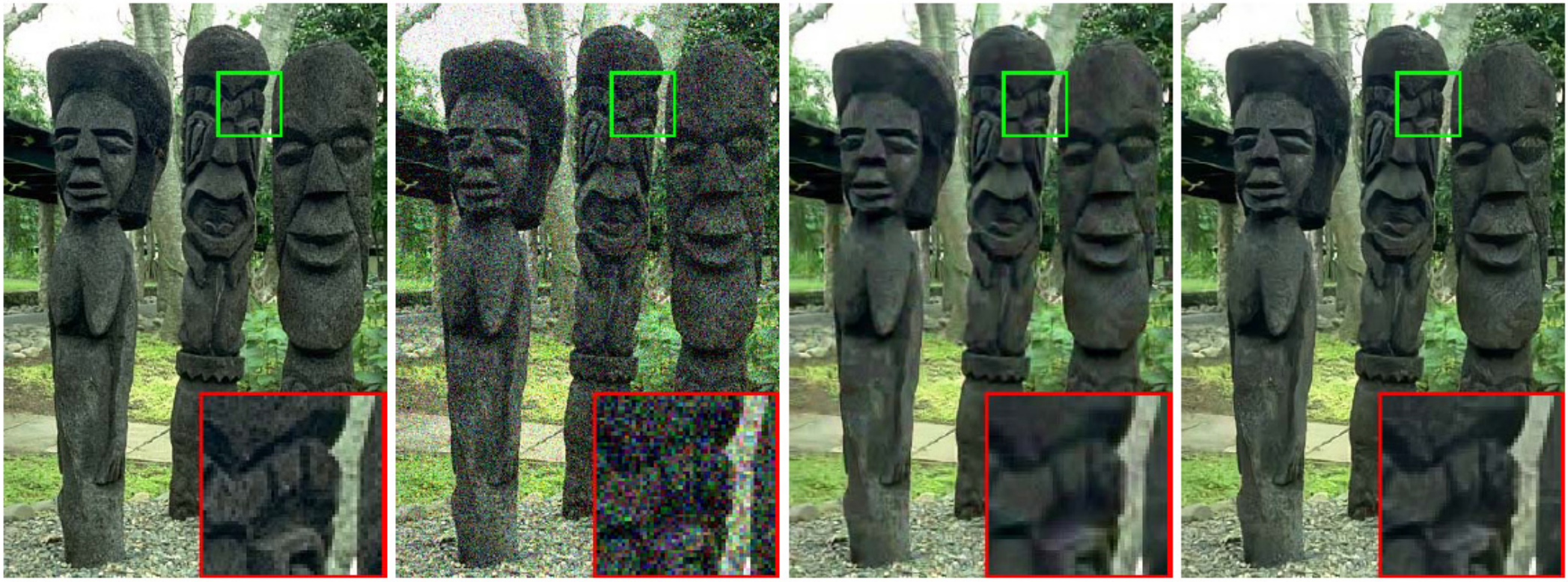
Numerical Predictive Performance

- In terms of **PSNR = 20 log (255/MSE).**
- **Four plots:**
  - With/without **batch normalization** (BN)
  - With/without **residual learning (RL) idea** (versus just estimating I_n)
  - Clearly shows **great benefit** of **both,** especially **combined**



Some Examples

- Trained tested on various datasets. See the paper.*

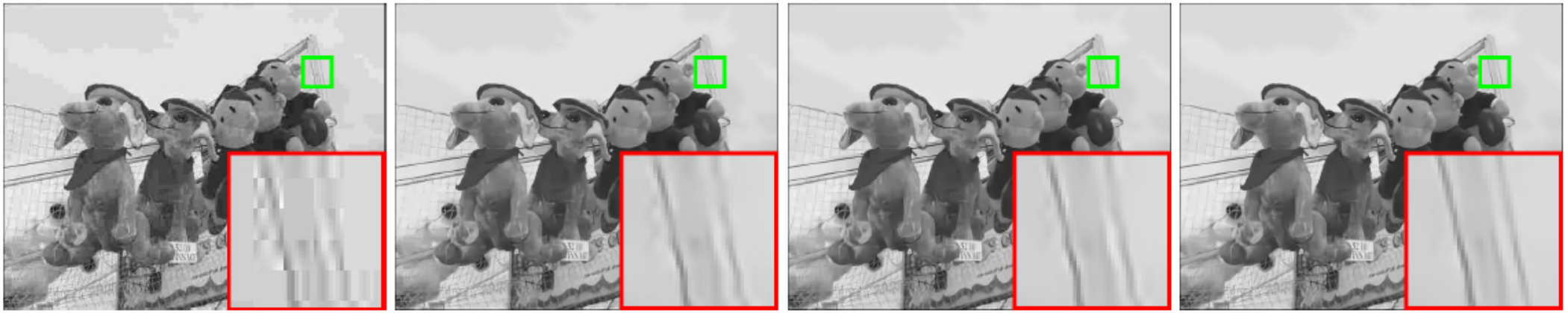


Original **Noisy** **Color BM 3D** **Color DnCNN**

*Zhang et al, "Beyond a gaussian denoiser: Residual learning of deep cnn for image denoising," *IEEE Transactions on Image Processing,* Feb. 2017.

Some Examples

- They also tried it on **JPEG blocking "noise,"** and **compared** against state-of-the-art networks "AR-CNN" and "TNRD" (see the paper):



JPEG AR-CNN TNRD DnCNN

- The topic of **"de-blocking"** is also a **hot research area.** They used this to show the **generality** of their method.