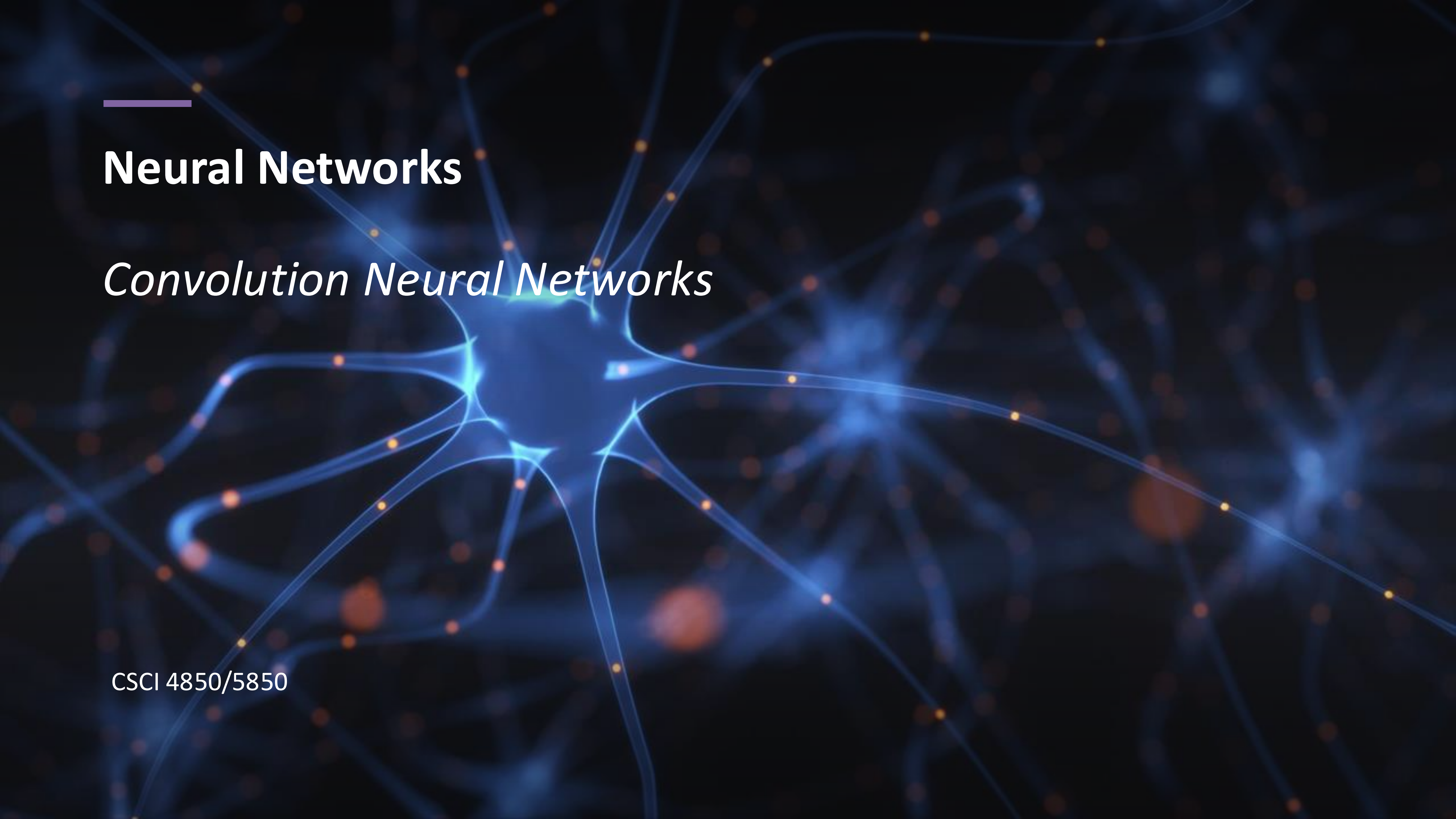


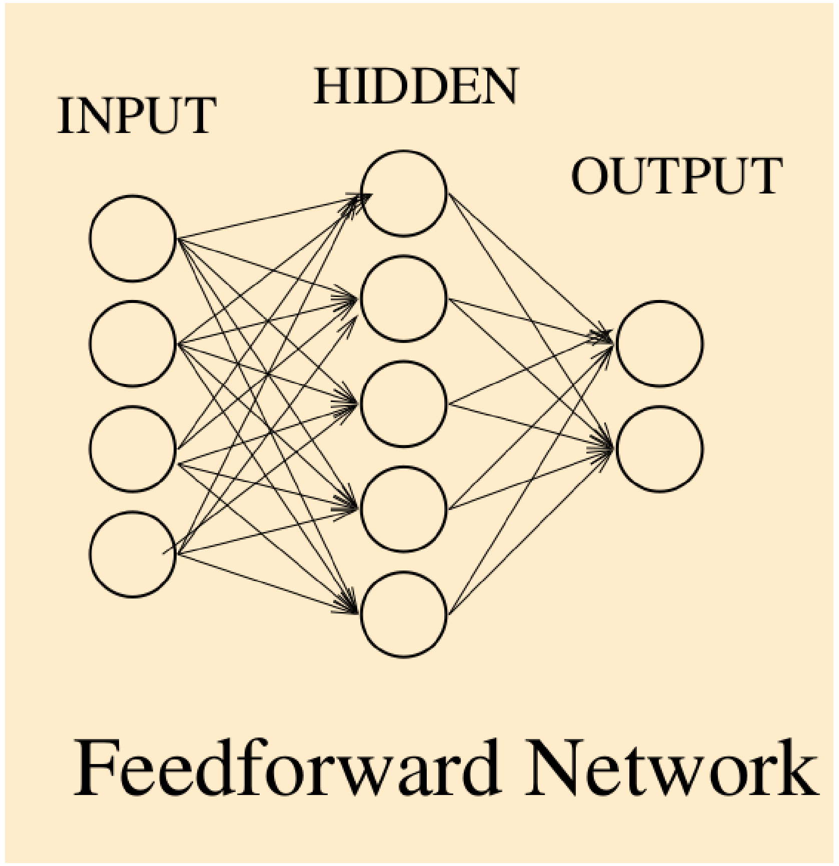
Neural Networks

Convolution Neural Networks

CSCI 4850/5850

Universal Approximation

- It has been proven that given *enough* units and using only a single layer of hidden units, essentially *any* function can be approximated to an arbitrary degree of precision
- Hornik, Stinchcombe, and White, 1989
- Can **every** kind of function be *learned* in this way?
  - We aim for good generalization, not necessarily precision…
  - The loss function (which we are optimizing) pushes us toward better precision, but not necessarily good generalization (which we are **not** optimizing)
  - How do we help make generalization happen?
  - **Regularization** – topic for another day
  - But what else can we do first?



Encodings Matter

- Let's reencode the XOR problem vectors…
  - Let each value (0 or 1) be replaced with a corresponding vector
    - zero=[0 1]
    - one=[1 0]
  - For each pattern:
    - compute the *outer product* of the value vector for column 1 and the value vector for column 2
    - Flatten into a 4 element vector
  - Combine vectors into new input pattern matrix…
- Even Hebbian learning can solve this!

$$\begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

First pattern is [0 0], so use two vectors: [0 1] and [0 1]

$$\begin{array}{c|cc} \times & 0 & 1 \\ \hline 0 & 0 & 0 \\ 1 & 0 & 1 \end{array}$$

Flatten into a four element vector…

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

Do the same for all four:

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

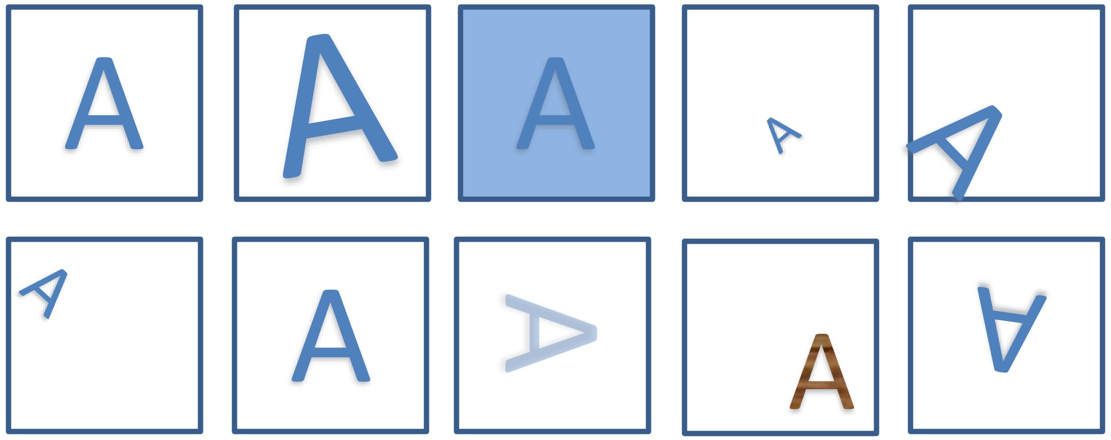
High-dimensional Projections

The previous example is not feasible for many problems since the *encoded* vector lengths will grow **exponentially** in the number of original input features (**the curse of dimensionality**)

However, projecting data into higher dimensional spaces can *expose* boundaries and relationships **hidden** in the low-dimensional encoding (**the blessing of dimensionality**)

Invariance

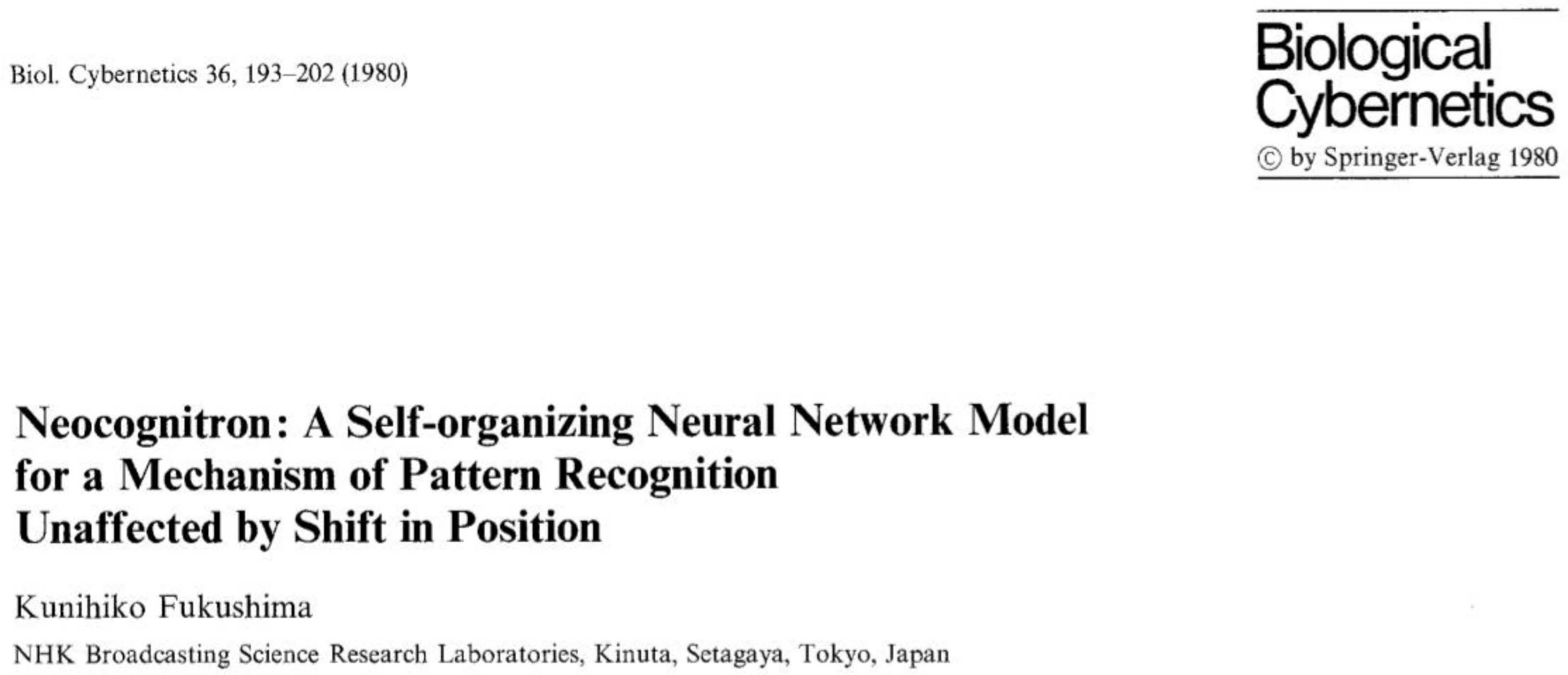
- Natural neural nets (i.e. brains) are very good at recognizing patterns with a wide ***variation*** in how those patterns are presented…



- Brains somehow compute a vector **representation** (i.e. activation vector from a set of neurons, some form of encoding) from inputs that are invariant to position, scale, orientation, color, texture, intensity, etc.
- Therefore, making your representation vectors **invariant** by *removing* such variance can greatly improve your network's ability to generalize
- However, this is really hard for many problems (chicken-and-egg)…

Back in time…

- Fukushima, 1980 – The Neocognitron
- First serious attempt at invariance removal



Biol. Cybernetics 36, 193–202 (1980)

Biological Cybernetics
© by Springer-Verlag 1980

Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position

Kunihiko Fukushima

NHK Broadcasting Science Research Laboratories, Kinuta, Setagaya, Tokyo, Japan

Neocognitron (Fukushima, 1980)

- **Unsupervised** architecture
- Inspired by the neural architecture of the *retina* and *primary visual cortex* in the brain

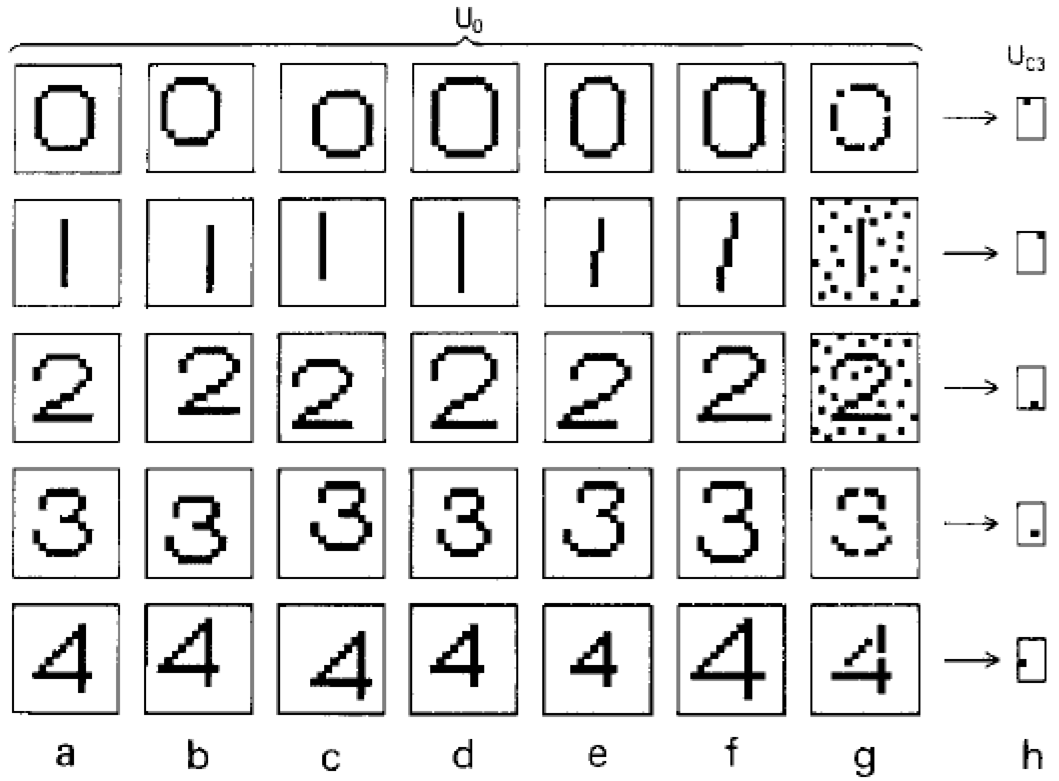
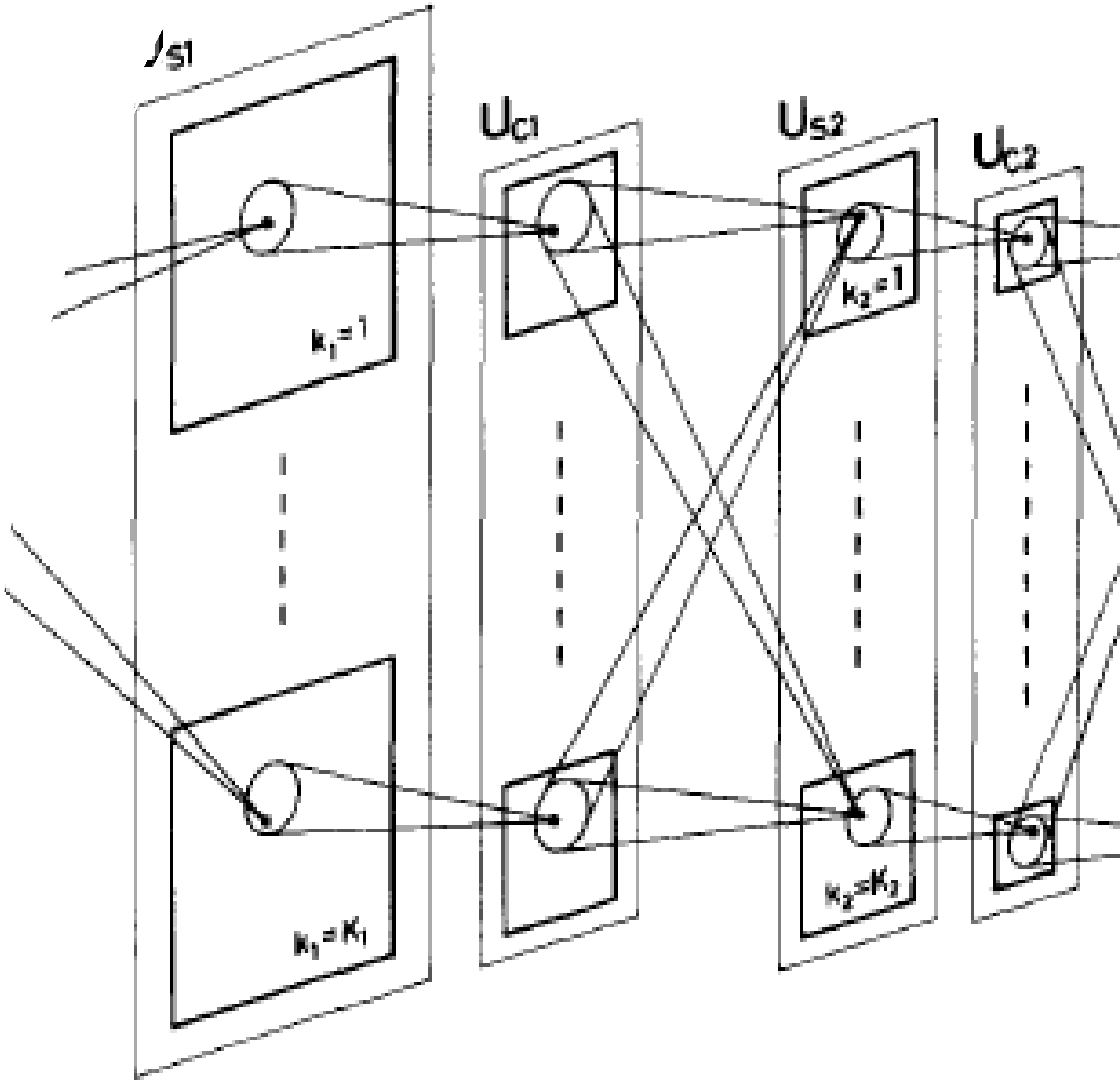


Fig. 6. Some examples of distorted stimulus patterns which the neocognitron has correctly recognized, and the response of the final layer of the network

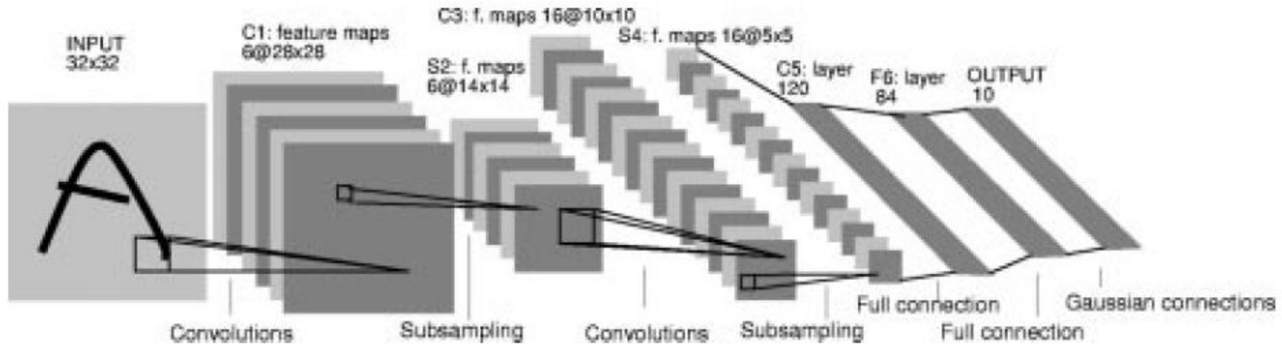
Multi-layer architecture for invariant digit recognition

- Many shifted layers at the beginning
  - high-dimensional projection
- Shrinking layer sizes down the line
  - low-dimensional selection

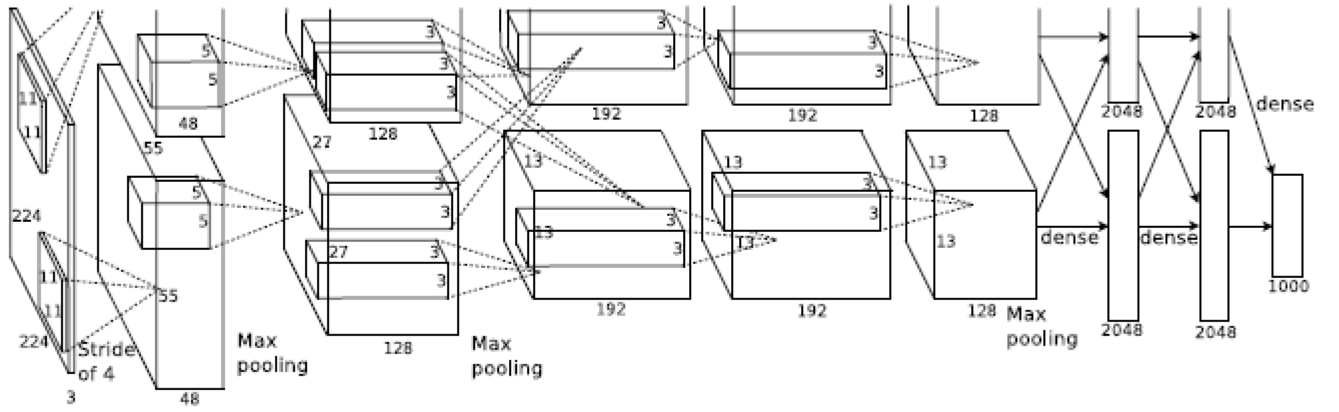


Fukushima paved the way for the modern version: Convolution Nets

- LeCun et al. 1998 – MNIST & LeNet-5 (99.3%)



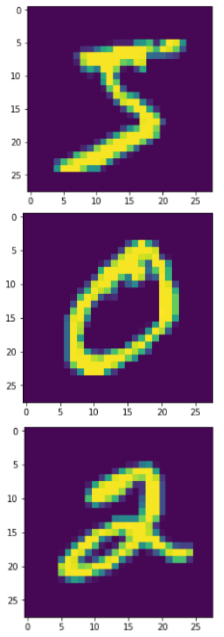
- Krizhevshy et al. 2012 – ImageNet & AlexNet (+10%)



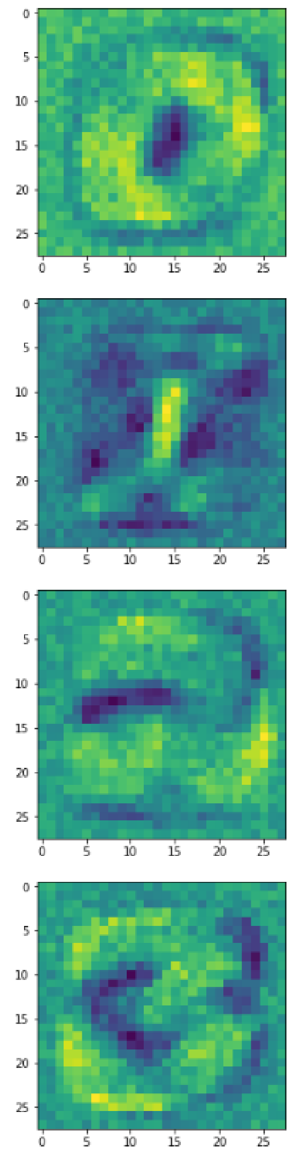
Traditional Approach: Non-Conv

- Simply take each image 28x28 and flatten it to create a 784-element vector
- 10-class softmax layer for output -> train categorical cross-entropy -> 93%
- Add ReLU hidden layers -> 98%
- Not bad, but that's because MNIST isn't the best data set
  - Most variance already removed

MNIST Data Set



Weight-vectors for each output unit in a single-layer network



Convolution Net (Basic) Components

Intuition: try to encode relationships between pixels since we already know that such relationships exist *a priori*

We are changing the *inductive bias* of the network, pushing it to learn what we want – pixels have information only in *context* with their *neighbors*

Trick one: preserve spatial relationships

Rather than flattening the input vectors, we present each pattern as a 2D matrix (more natural presentation)

Trick two: apply local filter units into layers (convolution)

Small groups of nearby pixels should form “features” of objects: mainly edges and boundaries

Make a filter unit (RelU) using only small n x m “patches” of activations from the previous layer (remember, that layer has 2D structure…)

Copy the unit (weights shared) across all possible subpatches (convolution)

That unit could now become a reliable detector of some feature (edge, boundary, curve, tip, etc.)

Multiple local filter layers combine (using weight sharing) features together (additional convolutions)

Trick three: eliminate spatial invariance by pooling

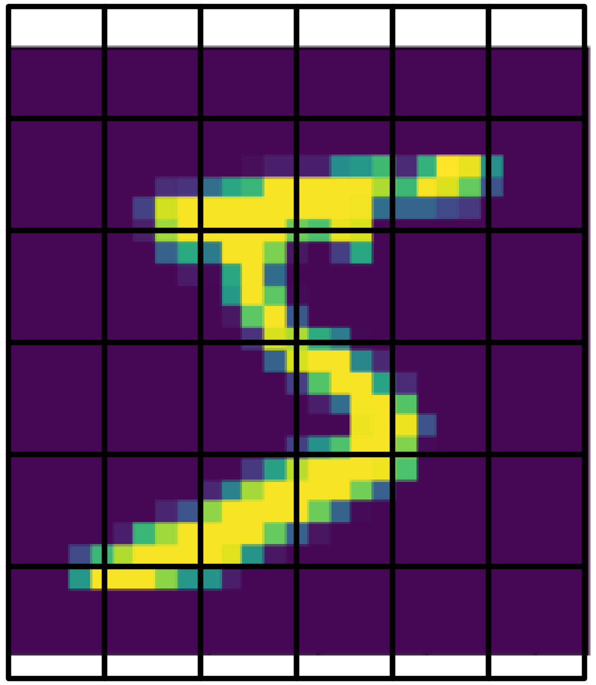
Apply a **non-overlapping** filter to the previous layer of feature (or multifeature) detectors

Analogous to downsampling or “shrinking” an image (removing rows or columns) by keeping the most likely detected feature (or multifeature) at the filter locations

Preserve 2D Relationships

- Imagine a 6x6 image
- Normally, we flatten into a 36 unit layer
  - Each unit is considered independent to a standard network
  - *a priori* relationships with neighbors is lost
- Instead, we make the input a 6x6 matrix
  - More natural form
  - Preserved relationships encoded by pixel proximity

Let's assume a 6x6 image as an example…
Technically, in [N,C,H,W] form: [1, 1, 6, 6]
CIFAR10 would be [1, 3, 32, 32]



Input shape: [1,1,6,6]

Full tensor would be [N,C,H,W]
N=batch_size, C=num_channels,
H=height, W=width (permuted!)

https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html

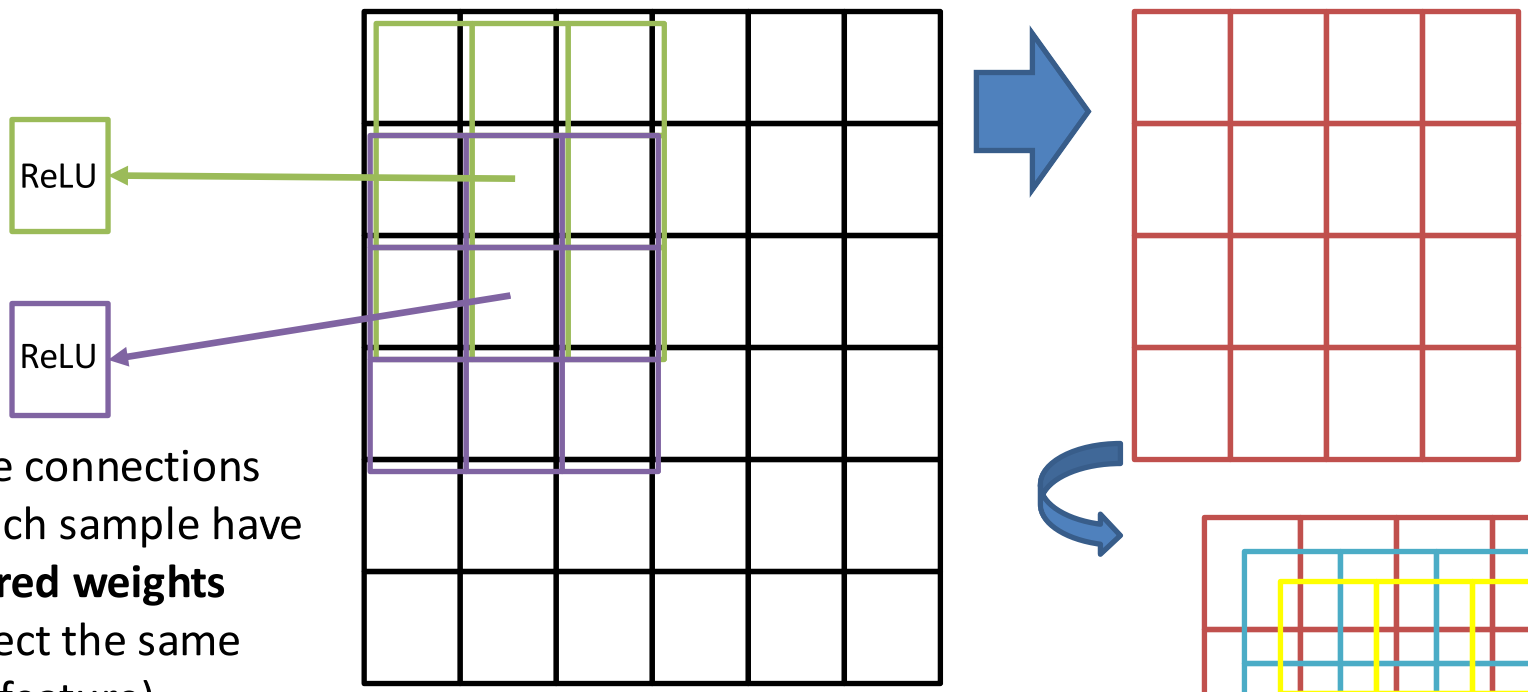
Apply Convolution Filtering

Initial shape:
[1,1,6,6] =
[1,36] (flattened)

(C=1 in this illustration, assumes greyscale intensity image)

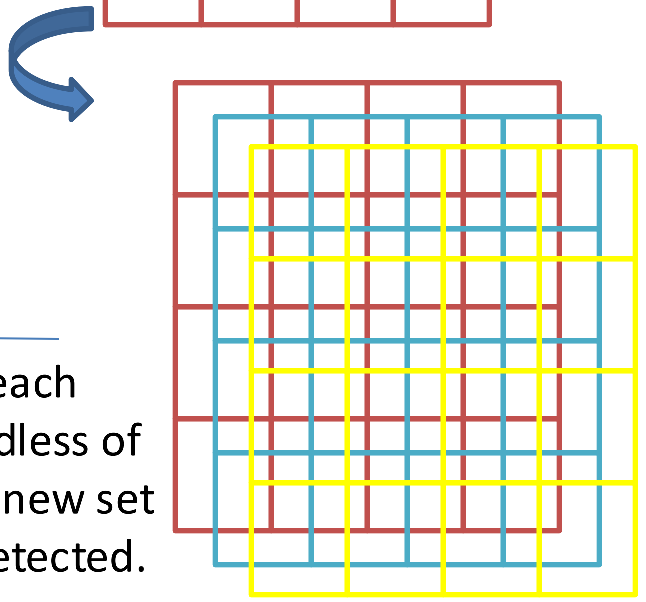
3x3 Filters
Stride=1 Sampling

Forms a new activation layer of ReLU units (4x4)



Net result is a **down-sampling** = lower dimensional transform of the input (in 2D)

Dense connections from each sample have **shared weights** (detect the same feature)



New shape:
[1,3,4,4] =
[1,64] (flattened)

(C=3 in this illustration)
High-dimensional Projection!!

Since each activation layer **shares weights**, you can think of each layer as providing a feature detector (detects a feature regardless of its 2D location). **Additional activation layers** (really only one new set of weights for per layer) allows for **multiple features** to be detected.

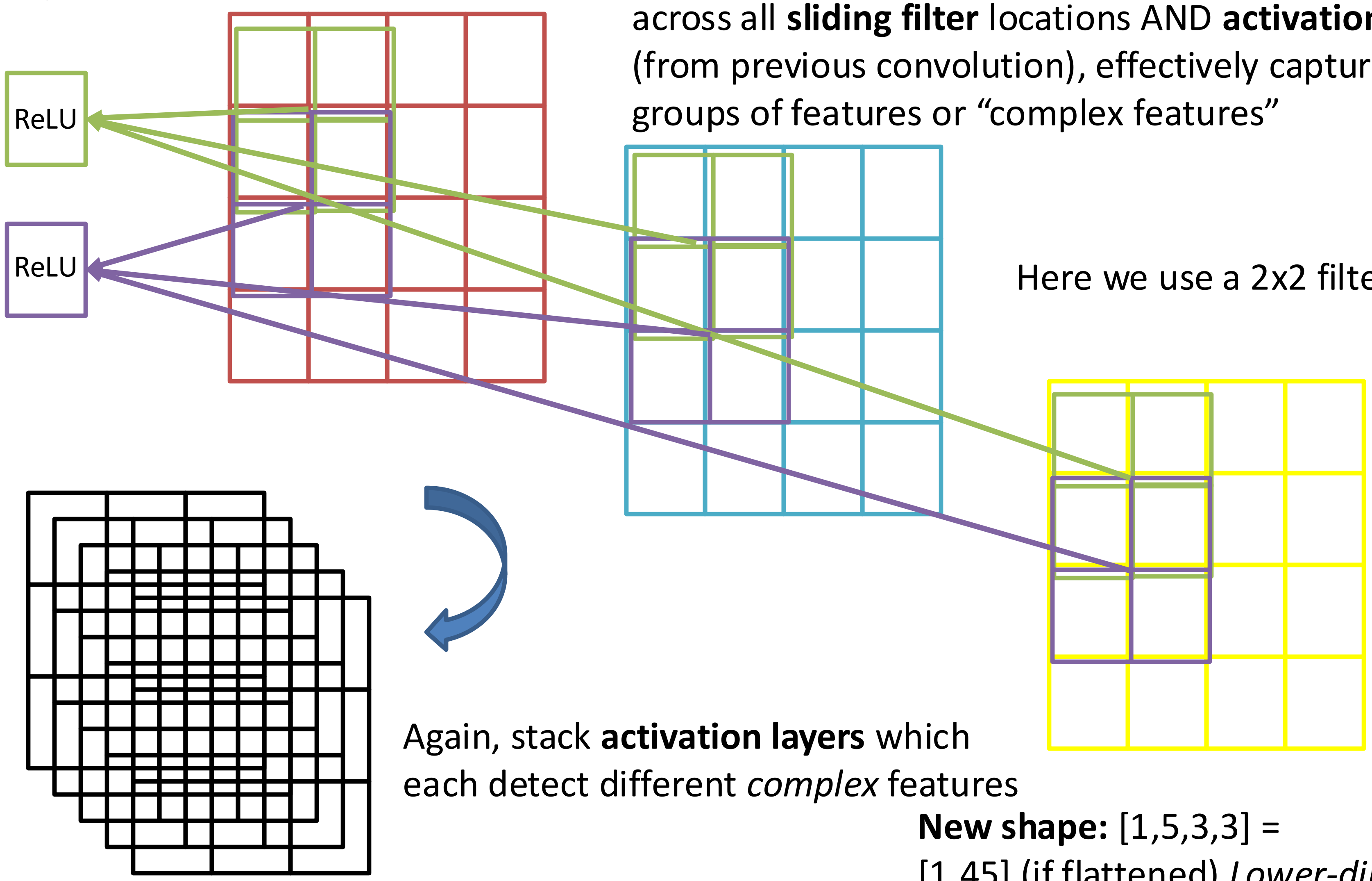
https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html

Usually Convolve Twice

Initial shape: [1,3,4,4] = [1,64] (if flattened)

Second convolution is similar, but **shares weights** across all **sliding filter** locations AND **activation layers** (from previous convolution), effectively capturing groups of features or "complex features"

Here we use a 2x2 filter…



Again, stack **activation layers** which each detect different *complex* features

New shape: [1,5,3,3] = [1,45] (if flattened) *Lower-dimensional Projection!!*

https://pytorch.org/docs/stable/generated/torch.nn.MaxPool2d.html#torch.nn.MaxPool2d

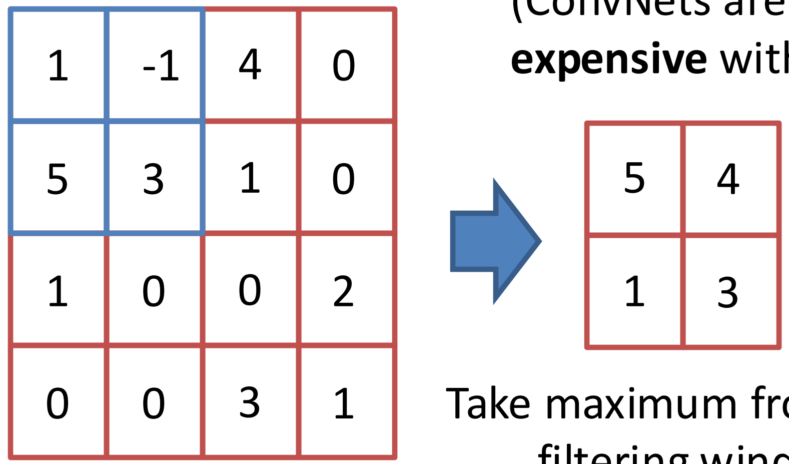
Note that I am not carrying over the tensor from the last slide since [1,5,3,3] would not be compatible with 2x2 pooling... think about why. (Hint: some parts of the tensor would be overlooked)

Pooling: to the Max

Good for reducing network complexity in terms of number of units/weights (ConvNets are **computationally more expensive** with all that layering going on…)

2x2 Max Pooling Filter

Initial shape: [1,3,4,4] = [1,64] (if flattened)



New shape: [1,3,2,2] = [1,12] (flattened)
Lower-dimensional Projection!!

Take maximum from each filtering window

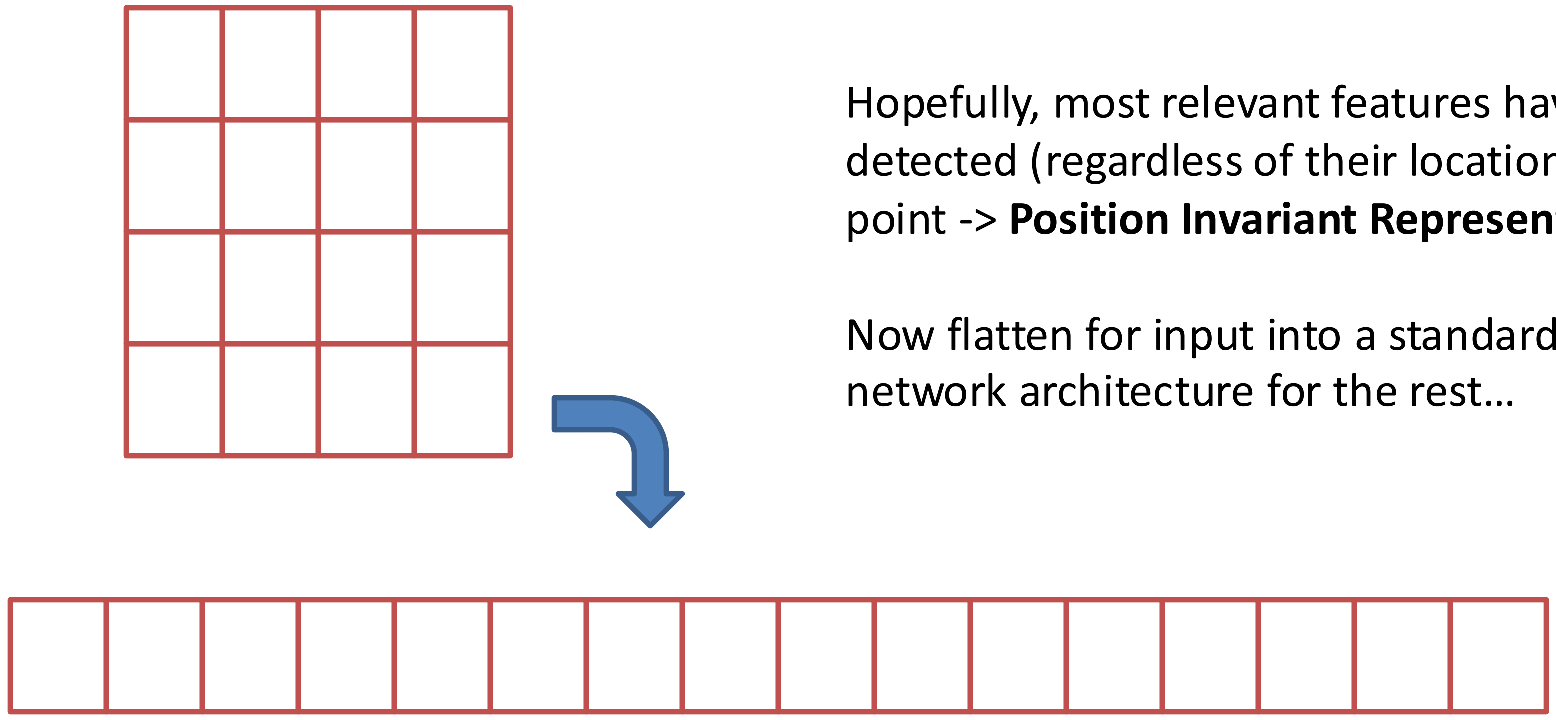
Rinse and repeat the three steps: *maybe several times…*

- Note that other kinds of pooling like taking the **average** might be more appropriate in other domains
- Also, **1D and 3D** convolutional layers are available in PyTorch to work with 1D and 3D data types, respectively

https://pytorch.org/docs/stable/generated/torch.flatten.html#torch.flatten

Final Stage: Flatten

After repeating the (conv, conv, pool) operation several times, you will reach a reasonably compact feature map size (2D information has moved into the Channels dimension).



Hopefully, most relevant features have been detected (regardless of their location) at this point -> **Position Invariant Representation**

Now flatten for input into a standard deep network architecture for the rest…

Nature/Nurture Synergy

- The convolution approach in PyTorch (and almost all other frameworks) is designed to produce **position invariant** representations
  - Inductive bias – given by “nature”
  - Convolving across color channels (red, green, blue) is also common (need Conv3D) to create **color invariant** representations (not necessarily what we would want though, right?)
  - Many (naïve practitioners) still assume invariant representations are induced *by the architecture* even when they are not really being created at all…
- To **make** the representations **invariant in other ways**, we can either again use “nature” OR we can “nurture” the network to produce these representations
  - Ensure that the **training** environment is **rich in variation** (include rotated images, blurry/noisy images, zoom in/out)
  - Such training data creates a **greater challenge** for the network, so **training time may take longer**
  - Any compensation needed to perform well on the task **must be learned**, which can be **slower than tuning the inductive bias**
  - Trade-off in *network complexity vs. training time*
  - Can *potentially* make incorrect assumptions with the inductive bias which may inhibit learning