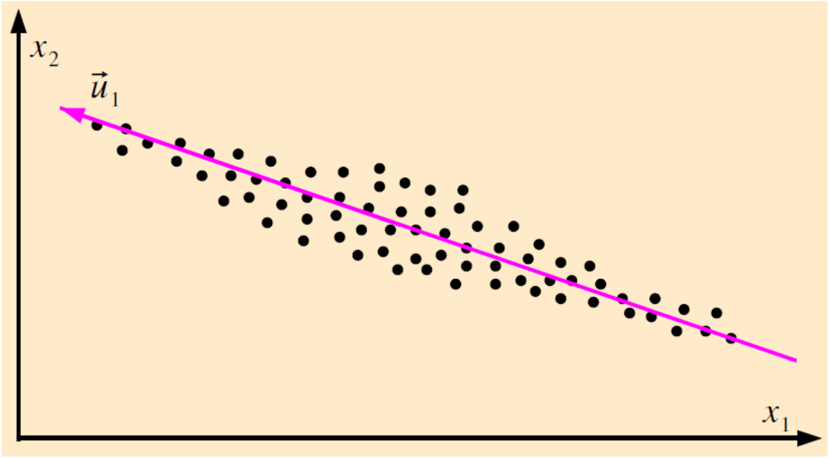
Neural Networks

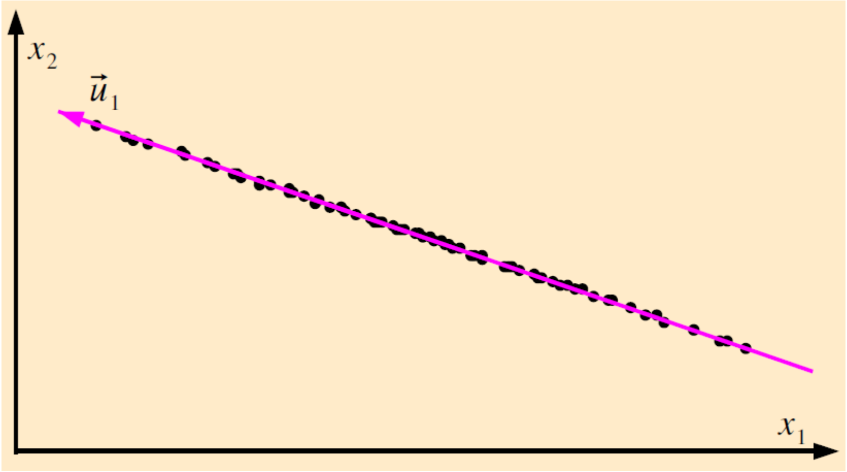
Generative Models: Autoencoders, Generative Adversarial Networks, and Stable Diffusion

CSCI 4850/5850

Unsupervised Learning: PCA

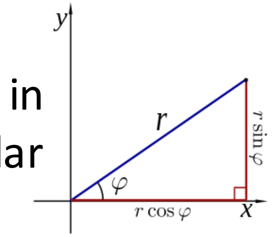
- Unsupervised approaches are common in machine learning. Why?
  - Assume most data lie on a low-dimensional manifold (*latent space*)
  - The full feature space is called the *ambient space*
- PCA
  - Low-dimensional projections
  - Visualization
  - Linear relationships
  - Oja's Rule (1982) – single layer net learns PCs in the weight vector
- Non-linear? Two NN strategies…
  - Autoencoders
  - Generative Adversarial Networks

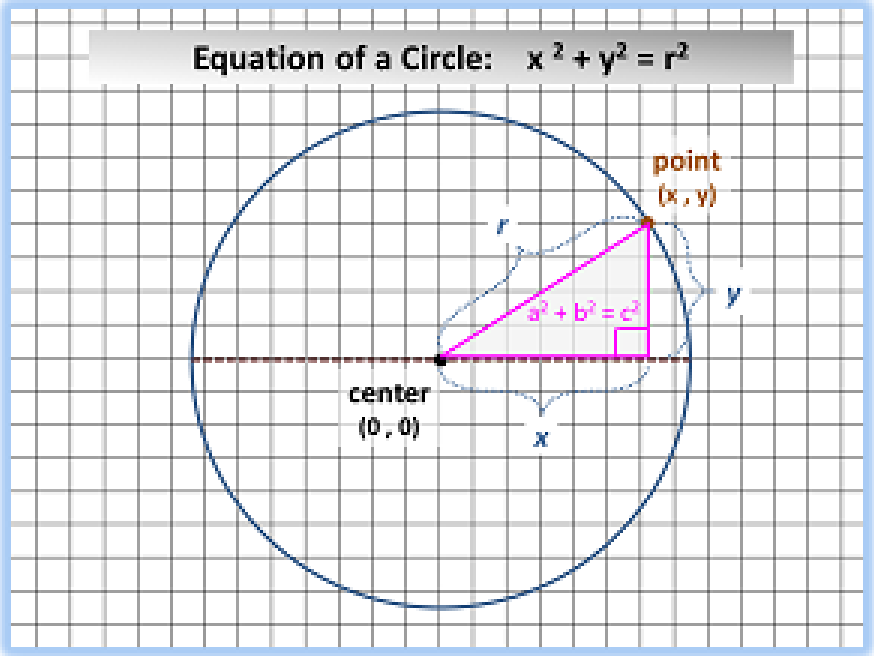




Ambient vs. Latent Space

1D in Polar



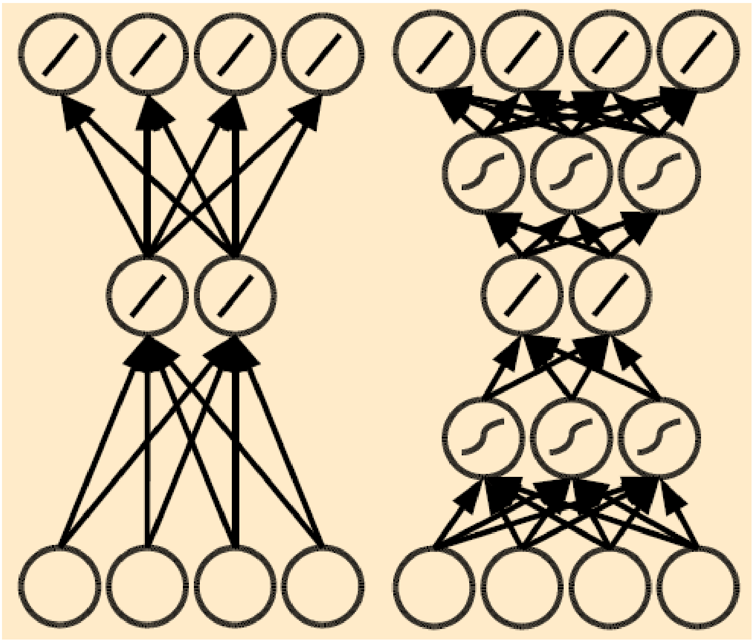


Autoencoders ("Self-encoding")

- Autoencoders are unsupervised neural networks
- Let the output vectors be the same as the input vectors (train using backprop)
- Hidden layer activations now form a learned embedding for the patterns!
- Use *fewer* units in the hidden layer to force the network to form a **low-dimensional representation**

Input=x Hidden=z Output=x

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix} \rightarrow \begin{matrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{matrix} \rightarrow \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$



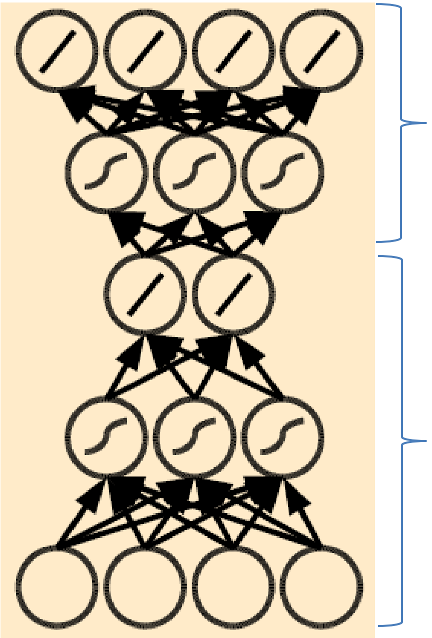
Linear (PCA)

Non-linear (Unsolved in general *Local Minima/Maxima*)

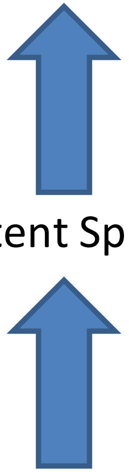
Using the Encoder

- Reduce the dimensionality of a data set (compression)
- Remove *noise* from data
- Fill in *missing data*
- Provide *pre-trained features* to supervised networks

Cottrell & Fleming, 1990

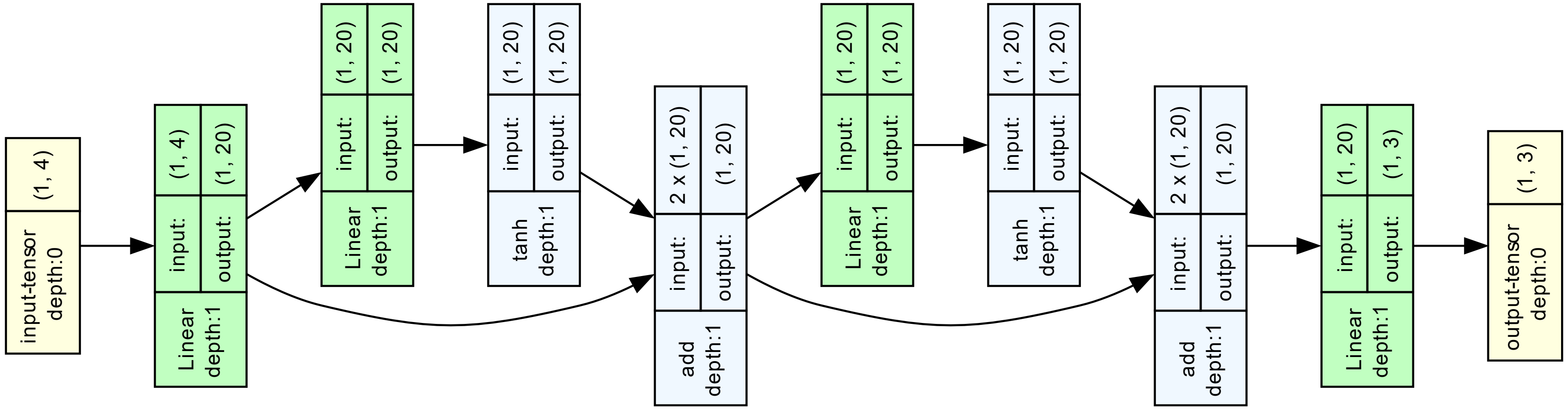


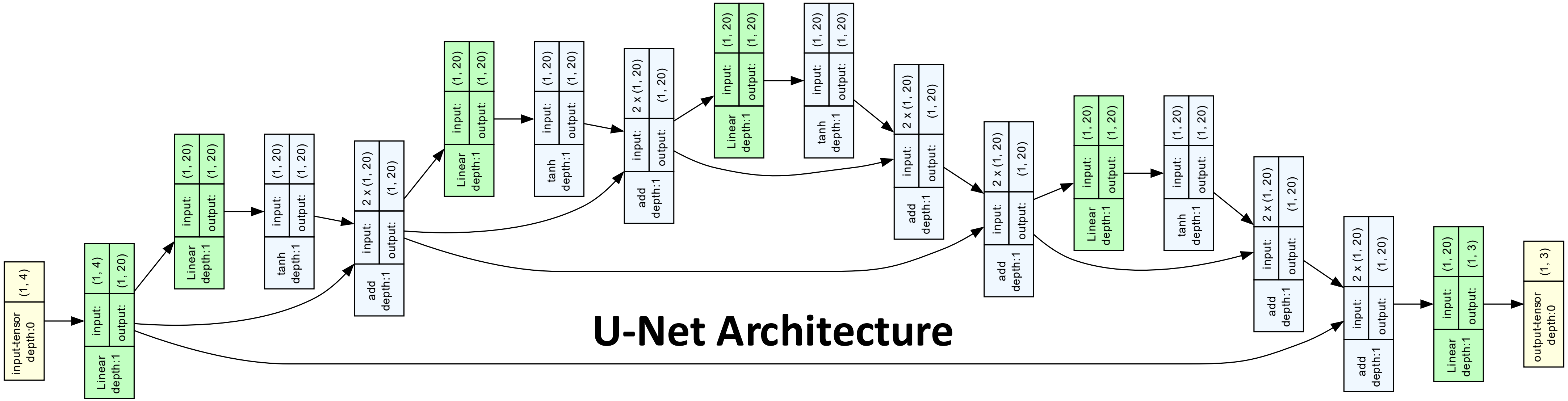
After training, decouple the network into *two parts*



Other Delta-Preserving Changes

Deep Residual Architecture



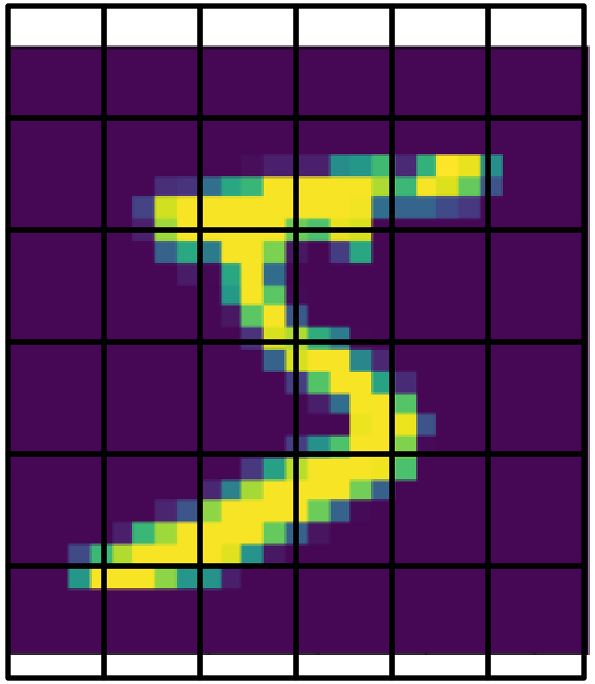


U-Net Architecture

Convolution: Review

- 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!)

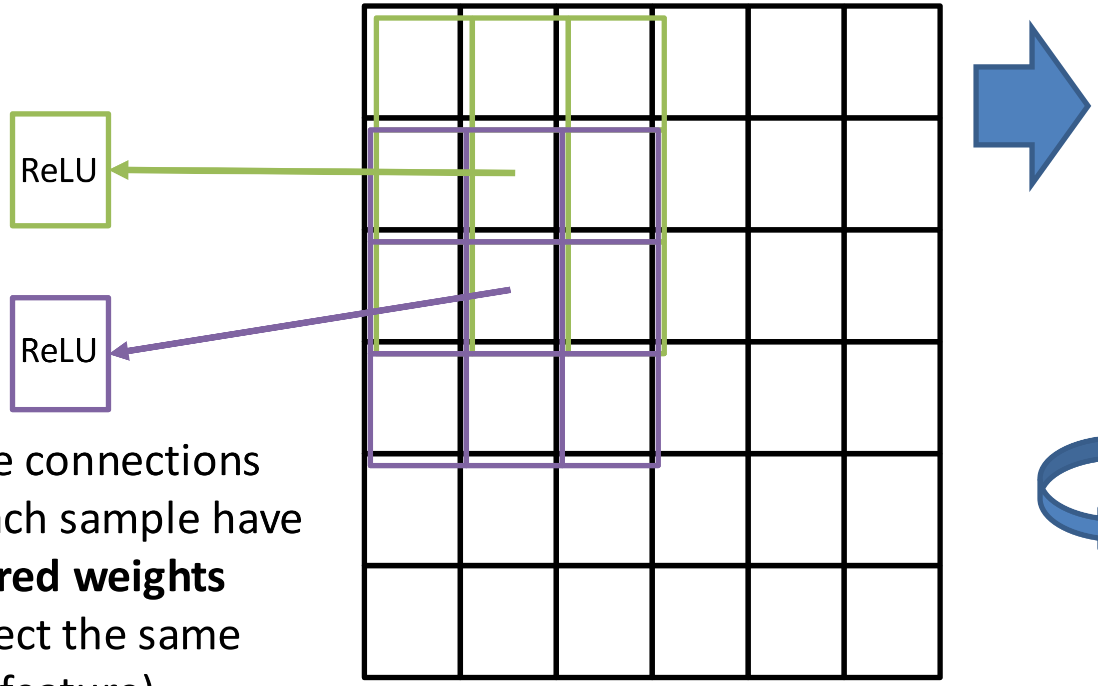
https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html

Apply Convolution Filtering

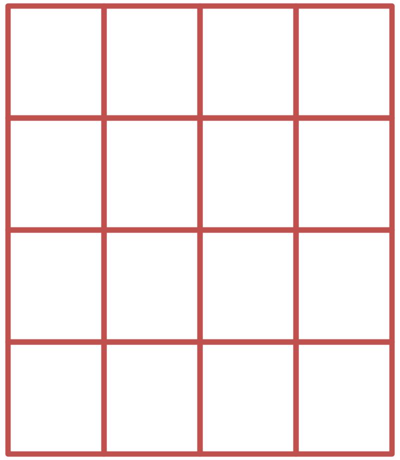
Initial shape:
[1,1,6,6] =
[1,36] (flattened)

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

3x3 Filters
Stride=1 Sampling

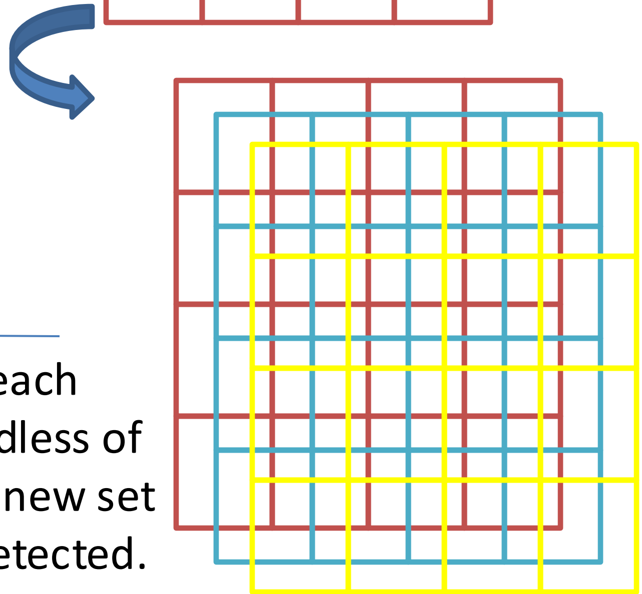


Forms a new activation layer of ReLU units (4x4)



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!!

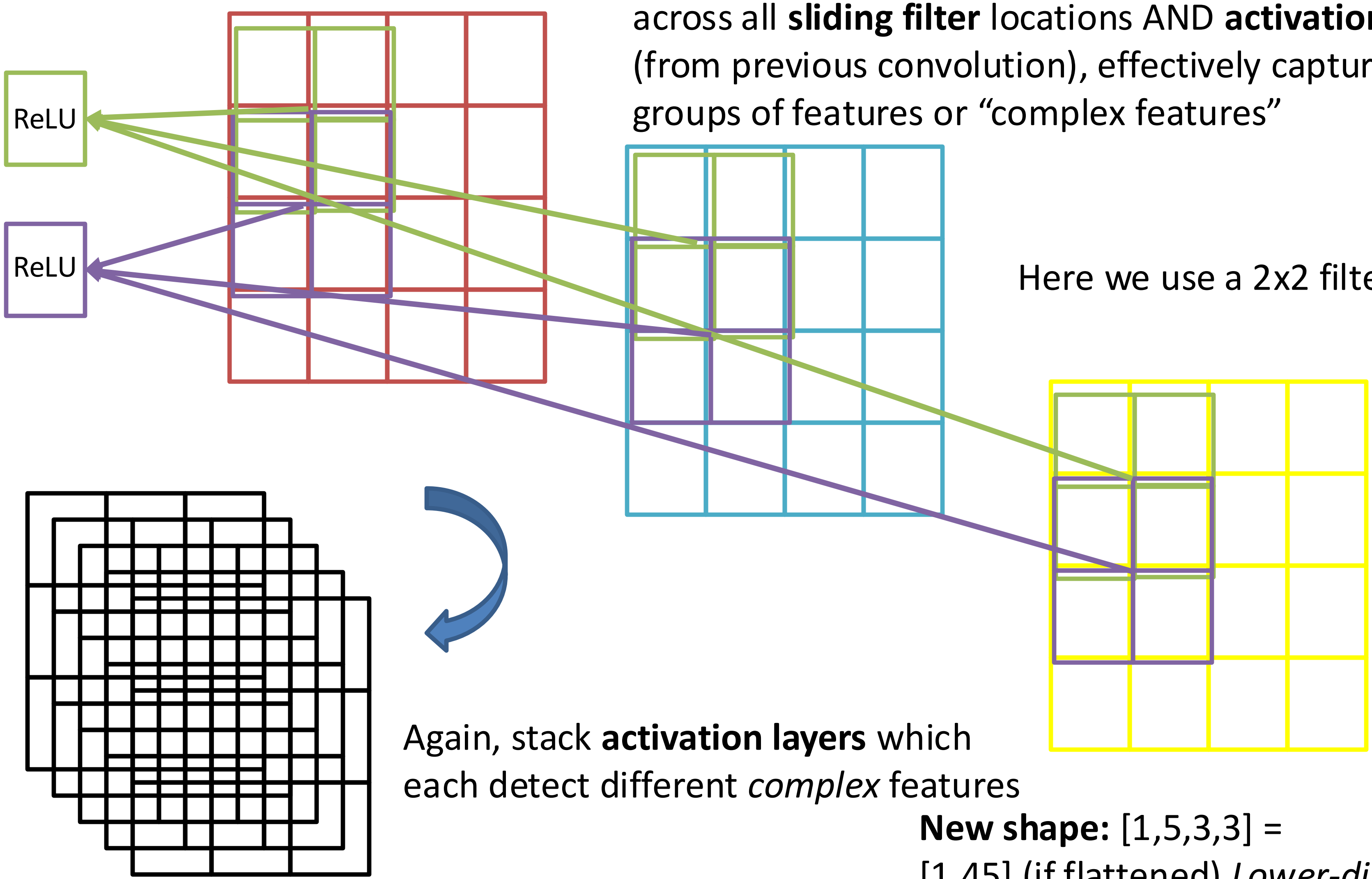
Since each activation layer **shares weights**, you can think of each layer as providing a feature detector (detects a feature regardless of its 2D location). **Additional activation layers** (really only one new set of weights for per layer) allows for **multiple features** to be detected.

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…)

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

2x2 Max Pooling Filter

1	-1	4	0
5	3	1	0
1	0	0	2
0	0	3	1

5	4
1	3

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

Take maximum from each filtering window

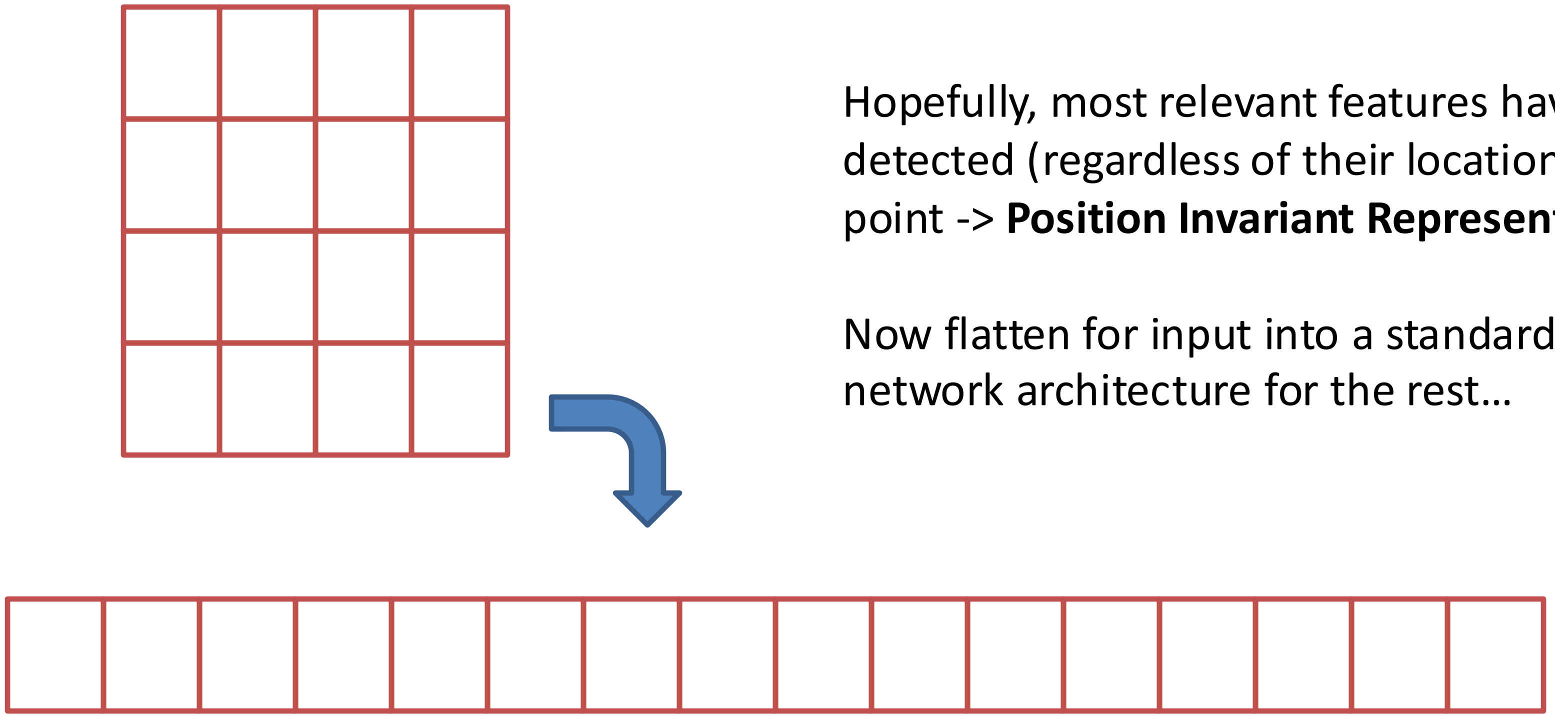
Rinse and repeat the three steps: *maybe several times…*

- 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

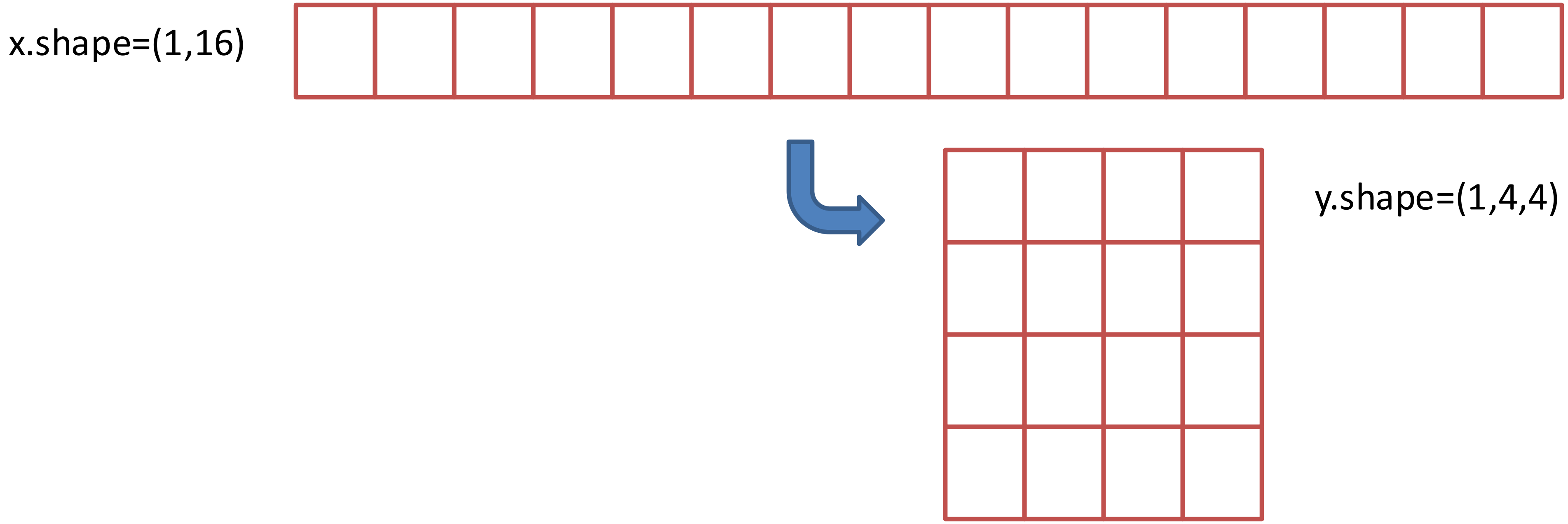
After repeating the (conv, conv, pool) operation several times, you will reach a reasonably compact feature map size (2D information has moved into the Channels dimension).



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

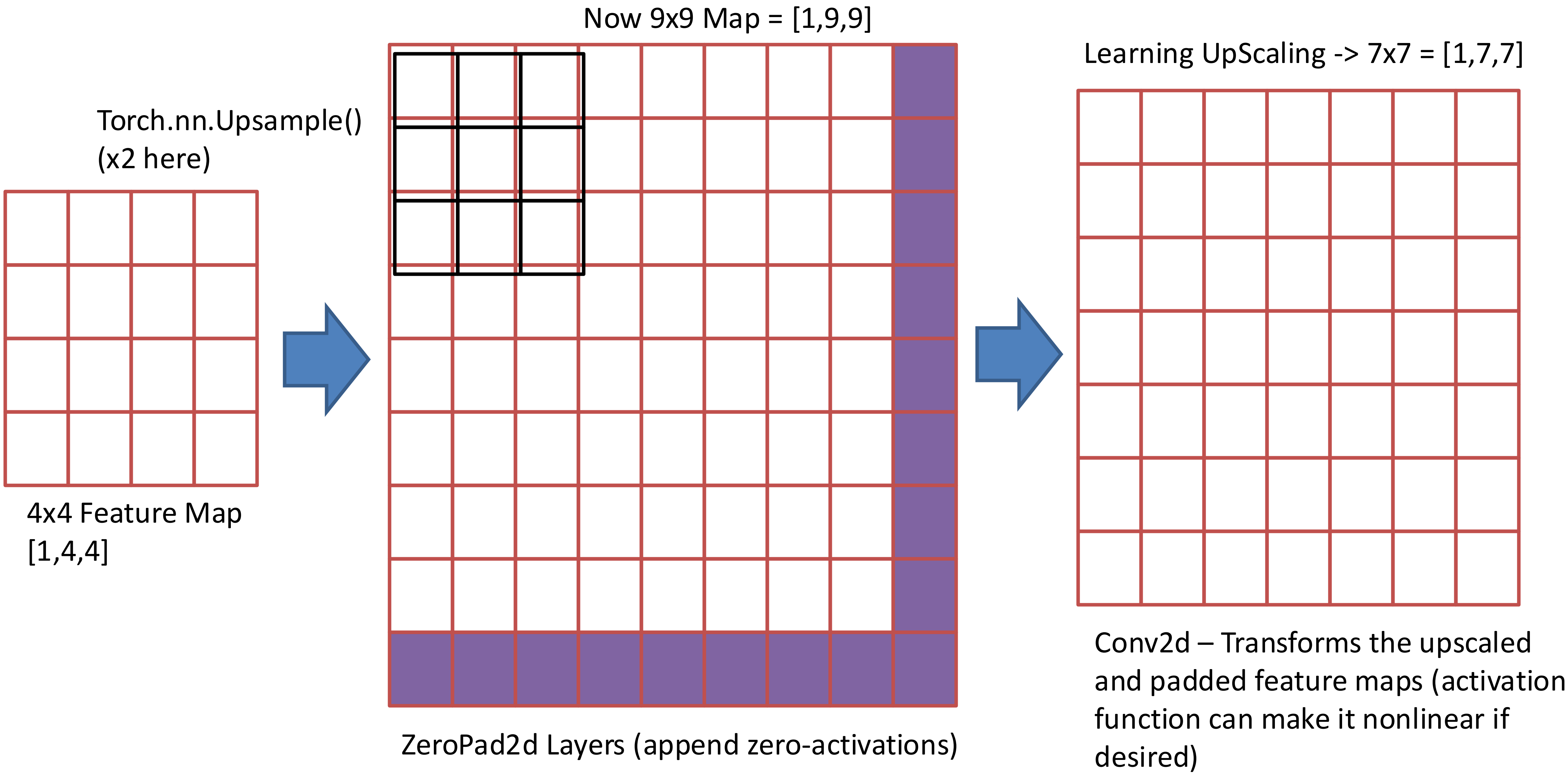
Now flatten for input into a standard deep network architecture for the rest…

Decoder Architectural Details



x. resize((1,4,4)) -> y : offers the ability to restructure tensors

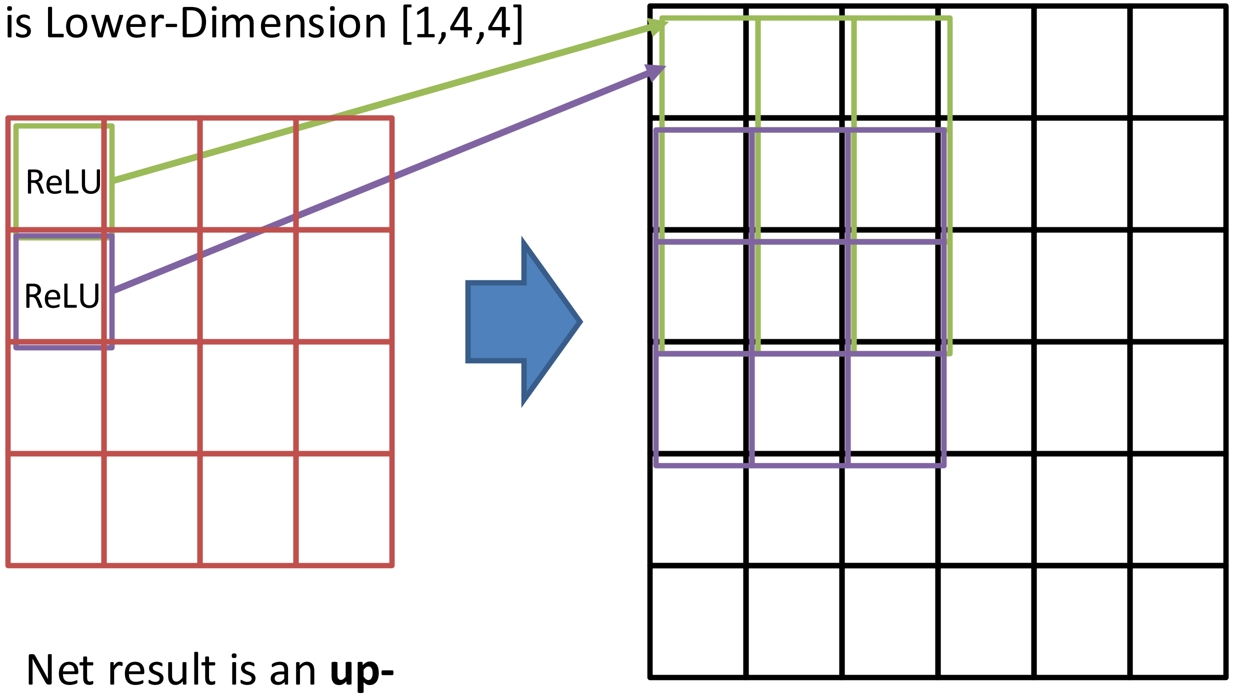
Upsampling and Padding



Transpose Convolution

3x3 Filters
Stride=1 Sampling

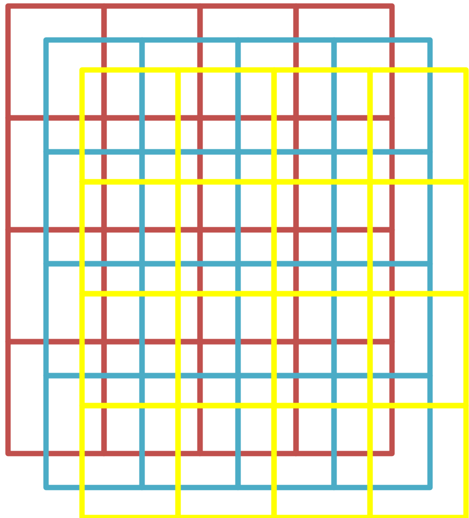
Input is Lower-Dimension [1,4,4]



Net result is an **up-sampling** = higher dimensional transform of the input

torch.nn.ConvTranspose2d Layers
Output is Higher-Dimensional = [1,6,6]

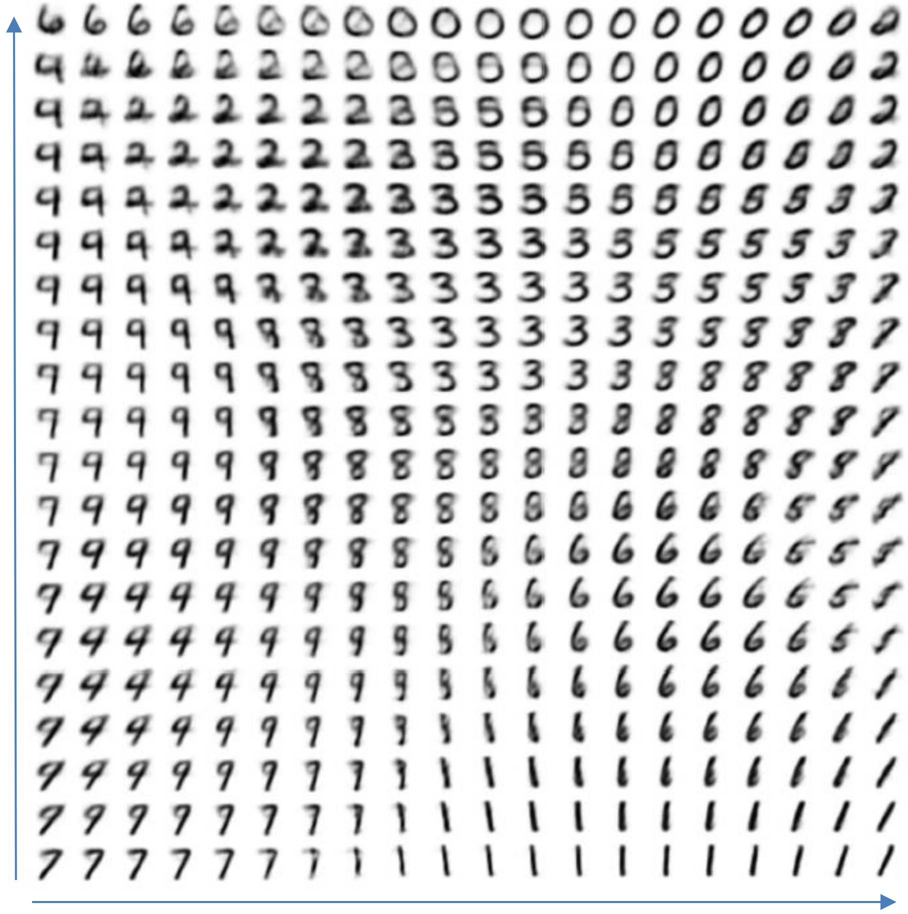
The sum of overlapping regions is accumulated – each unit from the input layer will contribute as well…



[C,4,4] -> [N,7,7]

Variational Autoencoders (Kingma and Welling, 2014)

- If we knew the distribution of hidden activations activations, p(z|x): we could randomly sample z
- Why?
  - Can use the decoder!
  - Provide a random z: generate a pattern, x
- Key insight: Assume the hidden layer (z) represent a Gaussian distribution in the *latent space*
  - Not unreasonable since the hidden layer should be largely removing nonlinearities!
- Network constructed to learn the mean and covariance of the Gaussian function at this layer
  - Lots of details we won't cover here
  - Easy to use for *generative modeling*

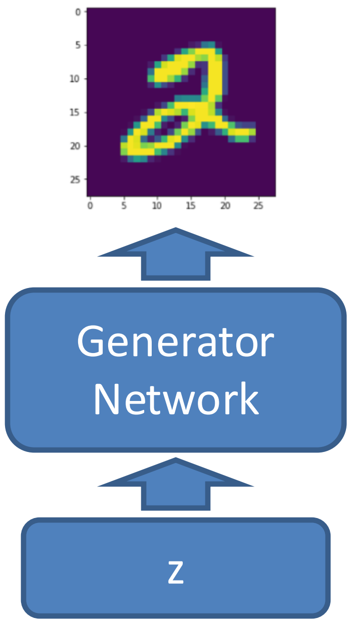


Hidden layer: 2D (2 means and 2x2 covariance matrix)

Sample from 2D Gaussian -> z is just of length 2!

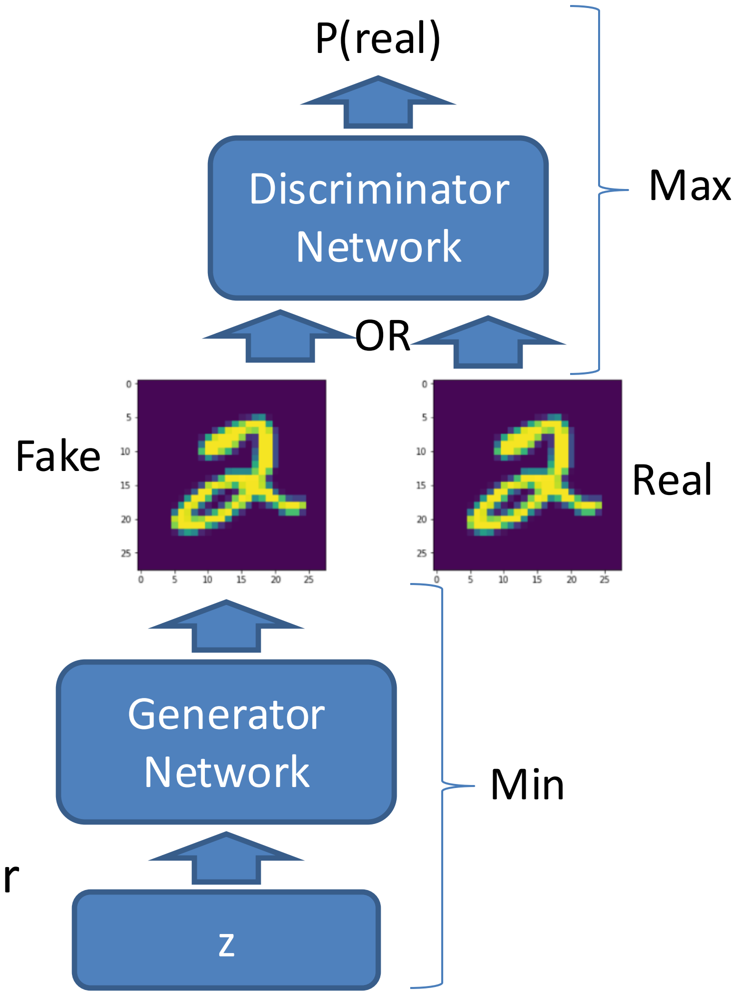
Generative Adversarial Networks (Goodfellow, et al. 2014)

- I don’t want to worry about distributions!
  - Not avoiding the math…
  - Don’t know the best function for representing the transformation…
  - Bottleneck size (regularization, optimizers, etc.) may impede formation of a Gaussian formation…
- Can we just make a decoder that takes random samples and learns to associate it with a pattern?
  - Mapping Gaussian samples to a latent space feature vector would be a ***function***
  - Let the neural network learn how to map from our sampling function (Gaussian) to the correct latent representation
  - Also learns how to decode the latent vectors
  - No need to worry about the input distribution shape!
- How would this network know if it is doing a good job?



Generative Adversarial Networks

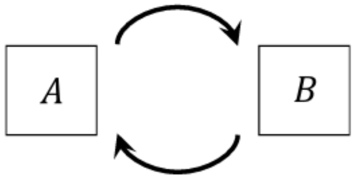
- Game-theoretic approach
  - Generator network learns to map from a random sample to a pattern (anything really!)
  - Discriminator (different) network learns whether and image is
    - real (from pattern data set)
    - fake (created by the generator)
- Pitting the two networks against each other will provide complementary feedback/learning signals
  - As the generator gets better at generating patterns, the discriminator gets more errors (feedback)
  - As the discriminator gets better at telling apart real from fake, the generator gets more errors (feedback)
- The complementary feedback between the two networks allows them to train each other
- Similar to how TD-Gammon and AlphaGo were trained using reinforcement learning



Competing objectives!

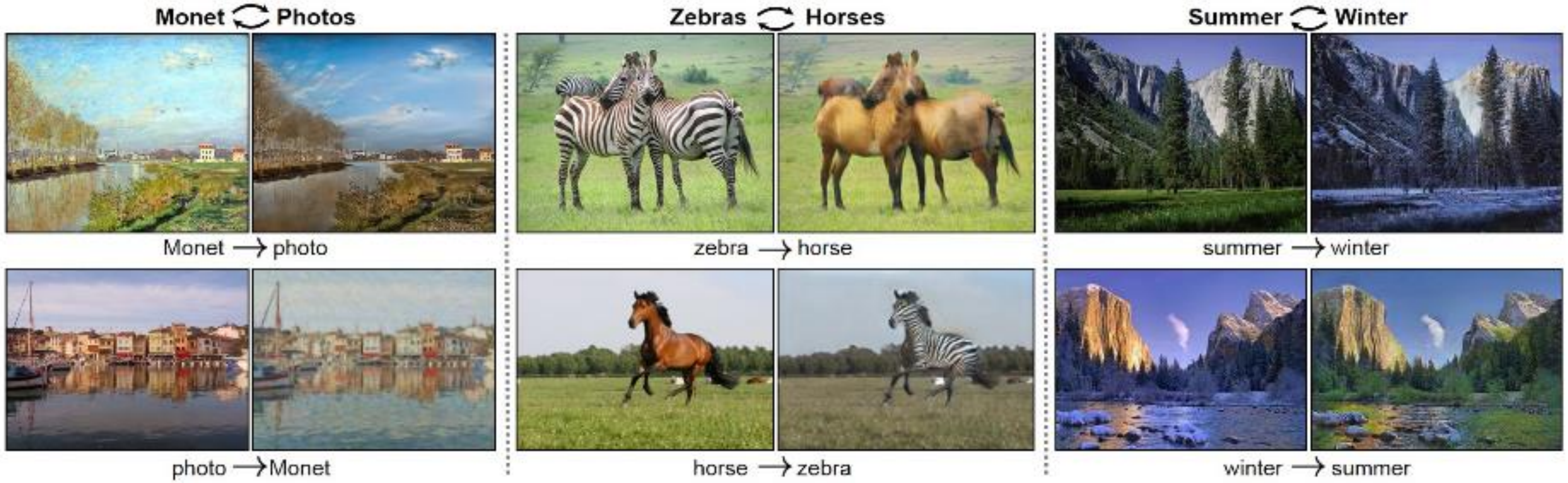
Examples

No smile (z_1) Smile (z_2)



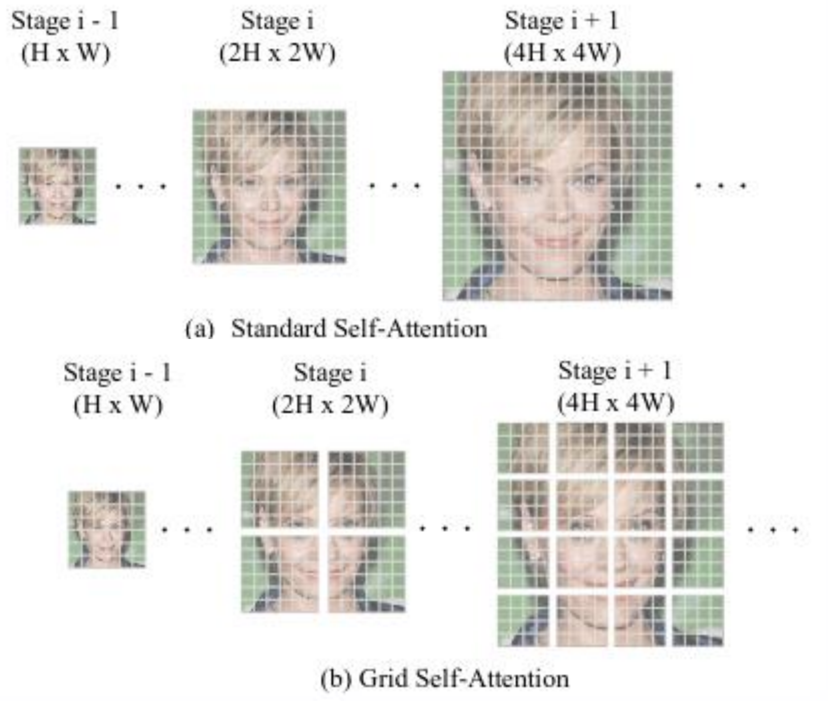
Difference between the two points forms a vector…
Average across many such pairs = smile-vector
Take a sample without a smile and add the smile vector = new smiling sample!

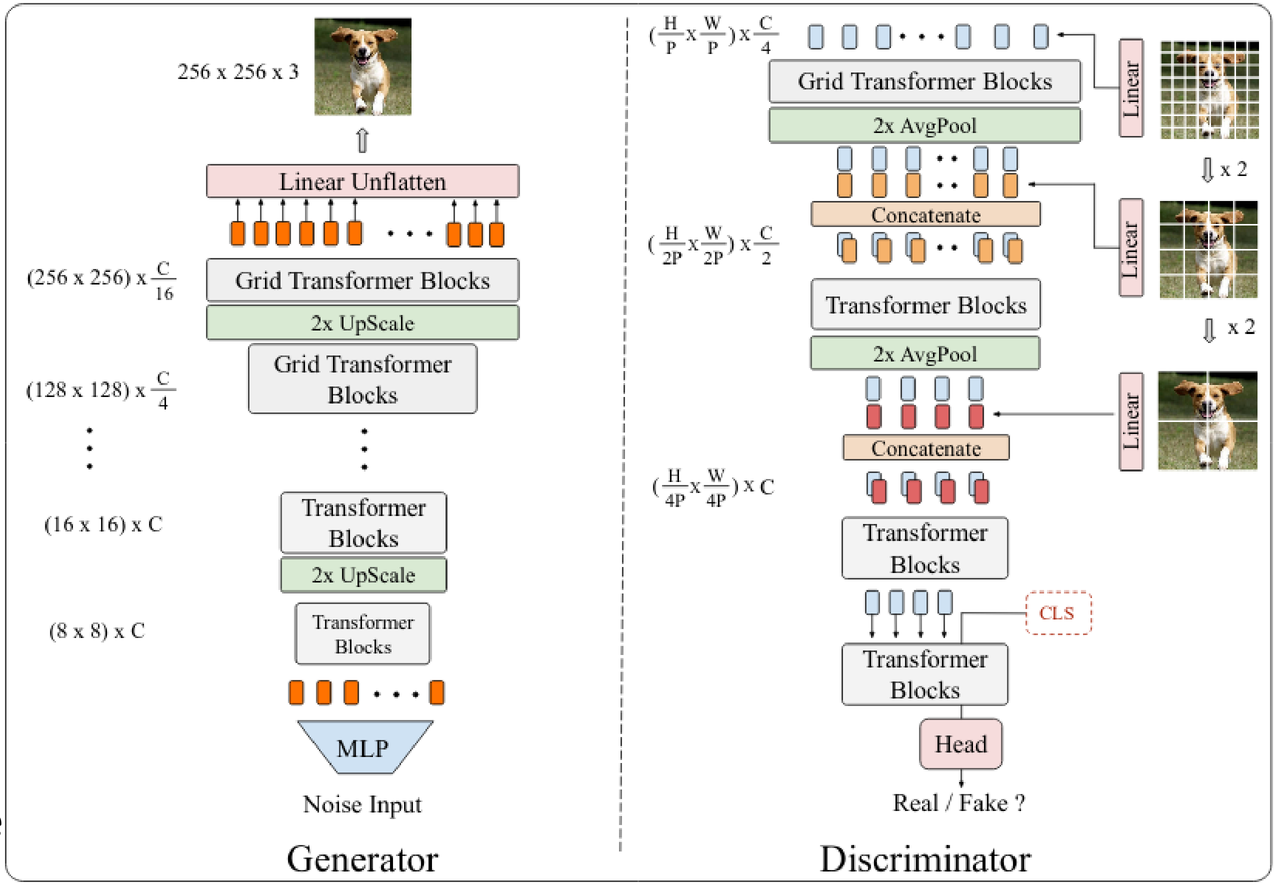
Dumoulin et al. 2017



CycleGAN - Zhu et al. 2017

Generative Models



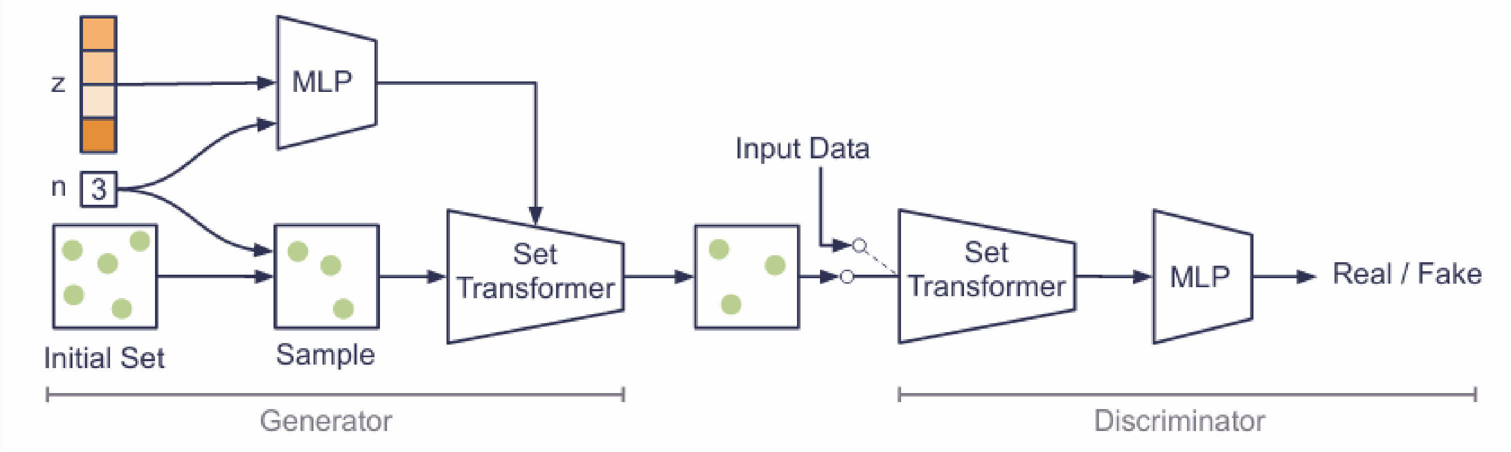


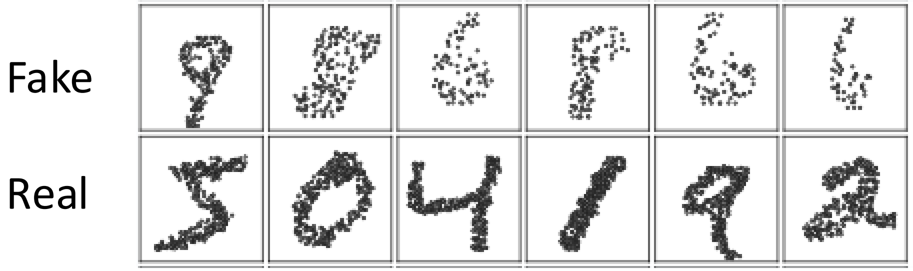
- Expanded the idea of patch-based structure to **multiscale patches**
- GANs (both **generator and discriminator**) constructed purely from residual transformer blocks

Jiang, Chang, and Wang. “TransGAN: Two Pure Transformers Can Make One Strong GAN, and That Can Scale Up” (2021) https://arxiv.org/abs/2102.07074

Generative Models (for Sets)

- Generate sets of arbitrary cardinatily
- No need to expensive loss function
- General embeddings for set-structured data



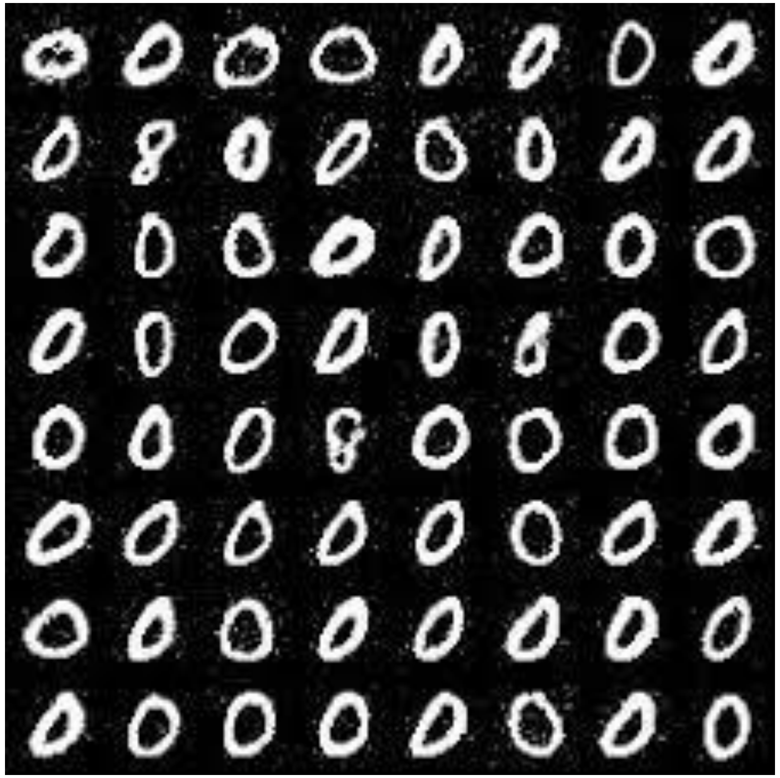


Stelzner, Kersting, and Kosiorek. “Generative Adversarial Set Transformers” ICML (2020). https://www.ml.informatik.tu-darmstadt.de/papers/stelzner2020ood_gast.pdf

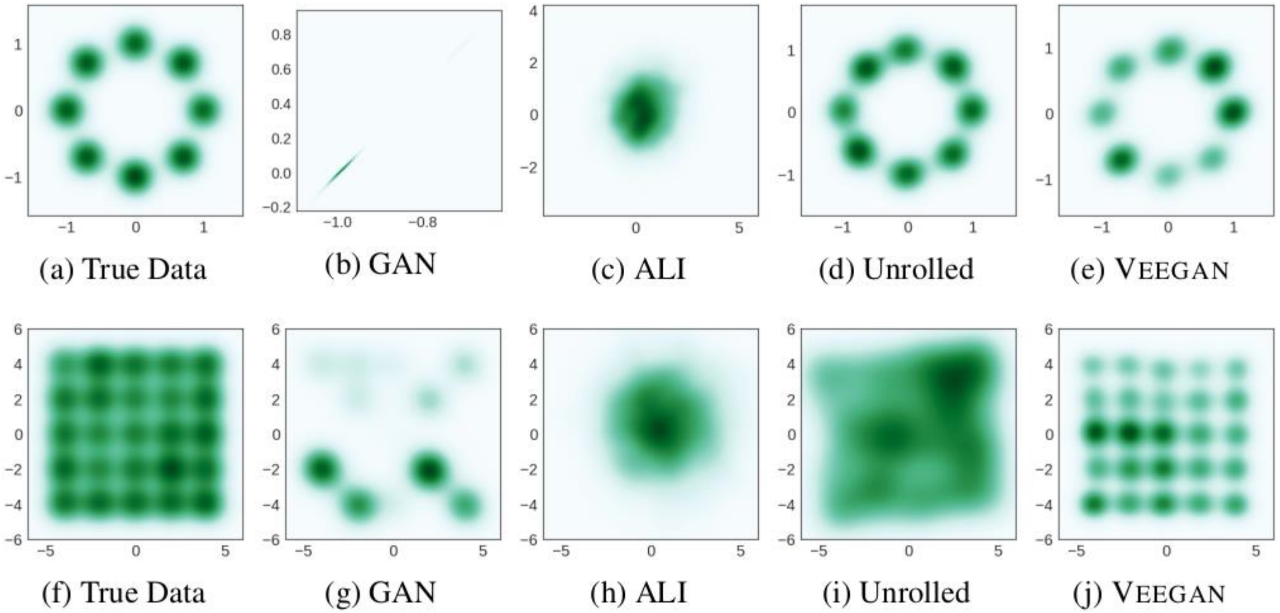
Many-to-Many Mappings

Almahairi, Rajeswar, Sordoni, Philip Bachman, and Aaron Courville (2018)
“**Augmented CycleGAN: Learning Many-to-Many Mappings from Unpaired Data**
https://arxiv.org/abs/1802.10151

Problem: Mode Collapse



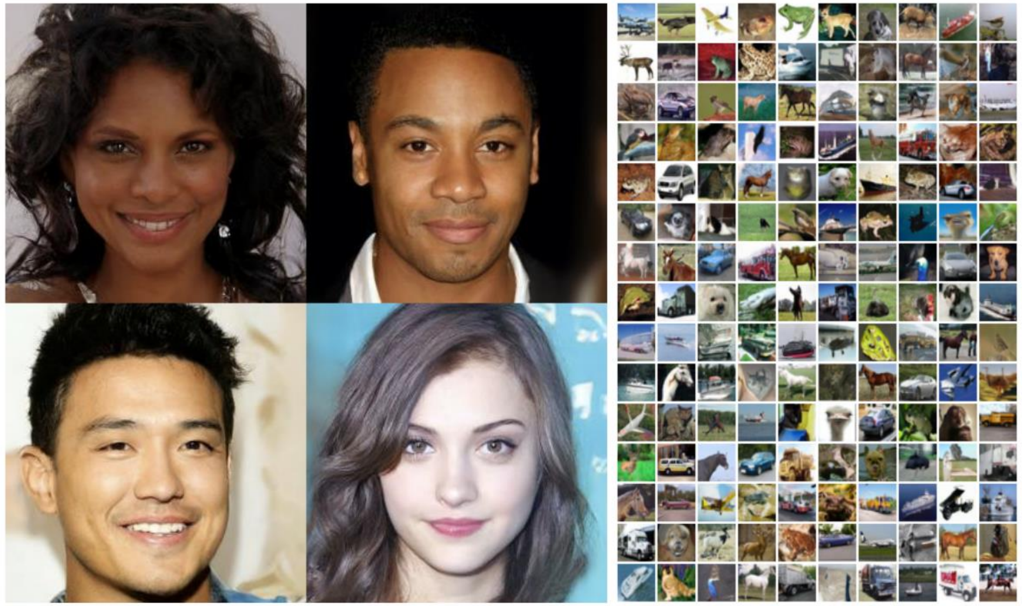
Thanh-Tung and Tran, 2018
https://arxiv.org/abs/1807.04015



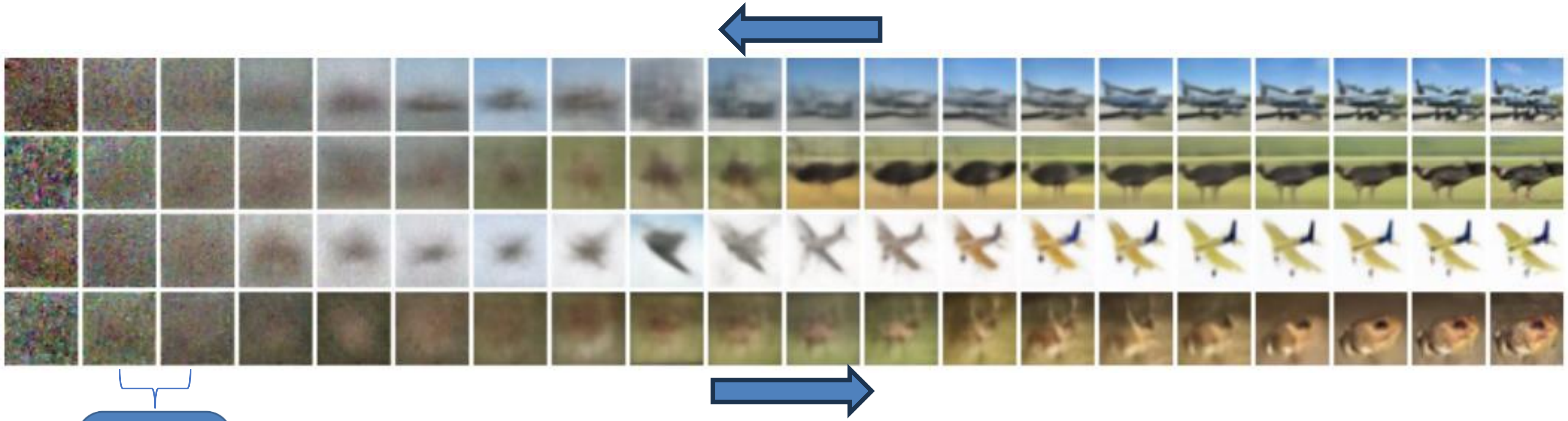
Srivastava et al., 2017
https://arxiv.org/abs/1705.07761

DIFFUSION MODELS

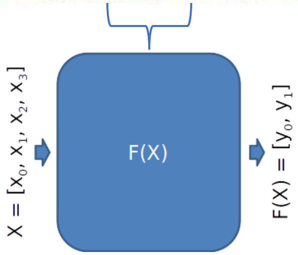
- Ho et al., 2020
- Denoising Diffusion Probabilistic Models (DDPM)
- An elegant solution to the *mode collaspe* issue with generative modeling tasks
- Sometimes called ***Stable Diffusion*** due to the correction of the mode collapse issue



Forward Process



Reverse Process



Separate Noise from the Signal (for each time step)