APS360: Applied Fundamentals of Deep Learning

Week 6: Unsupervised Learning

Motivation

Challenges with **Supervised Learning**

- Requires large amounts of labeled data
- Obtaining labeled data is expensive
    - Medical tests are expensive → require a specialist to review them
    - Chemical data collection → wet-lab tests are time consuming
- Often there is a lot more unlabeled data than labeled
- Not what we see in biology

Motivation

How do humans learn?

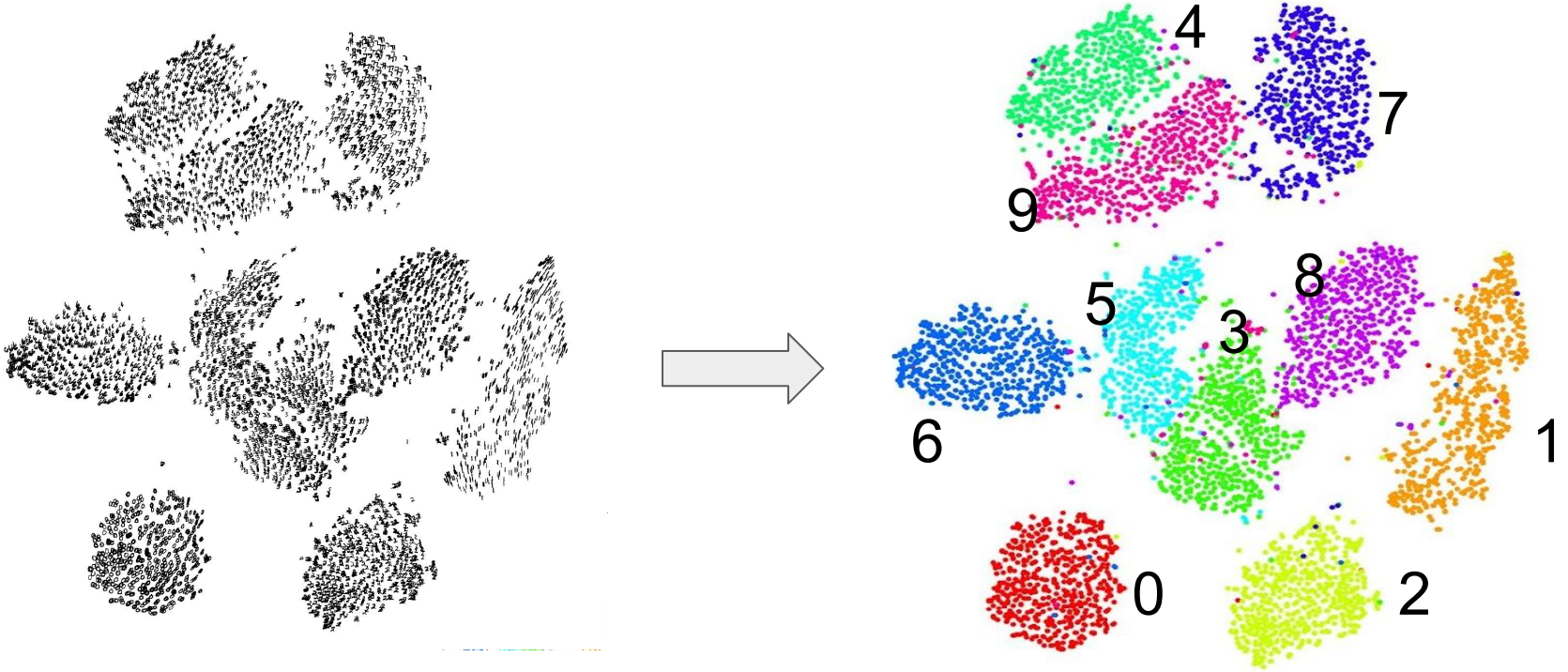
Recognize patterns within observations **without explicit supervisory signals**

Unsupervised Learning

Our brains are constantly observing the world around us for patterns, or some structure to relate objects.

Patterns or clusters of similar features can tell us a great deal about the data before we even have a label.

Feature Clustering



Autoencoders

Autoencoders

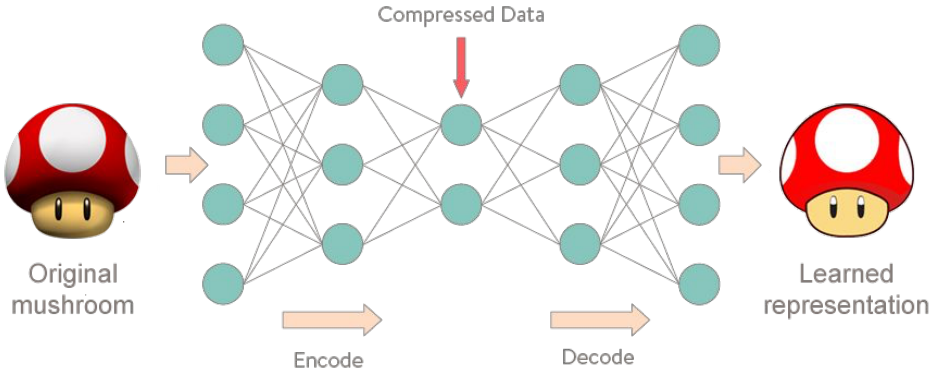
Find efficient representations of input data that could be used to reconstruct the original input using two components:

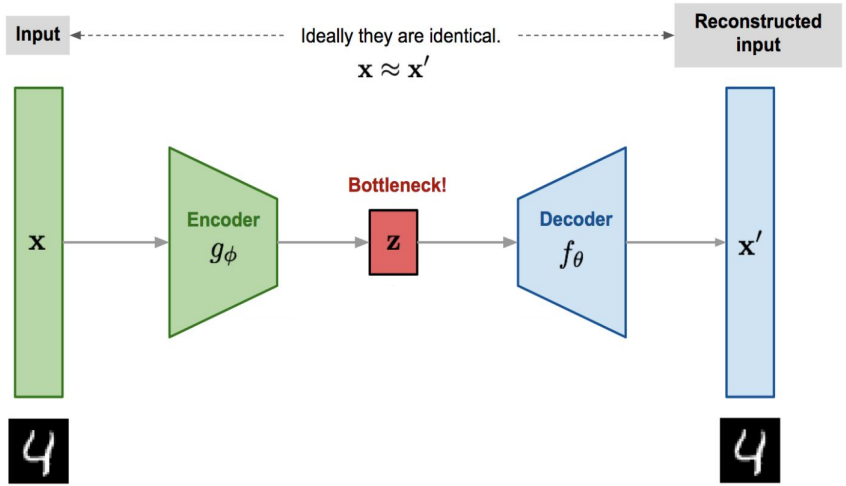
- **Encoder**
  - Converts the inputs to an internal representation
  - Dimensionality reduction
- **Decoder**
  - Converts the internal representation to the outputs
  - Generative network

Autoencoders

Hourglass shape creating a bottleneck layer, lower dimensional representation

It is forced to learn the **most important features** in the input data and drop the unimportant ones



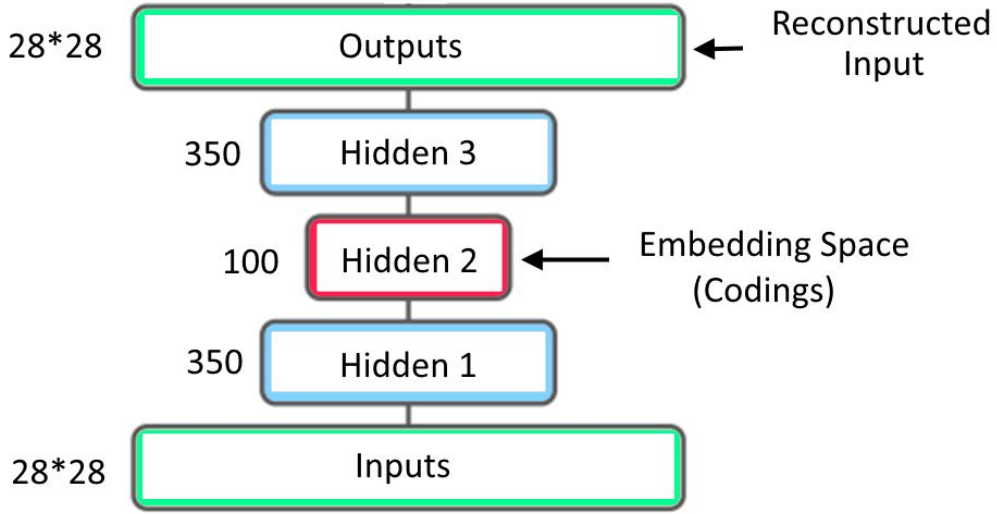


PyTorch Implementation

```
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        encoding_dim = 32
        self.encoder = nn.Linear(28 * 28, encoding_dim)
        self.decoder = nn.Linear(encoding_dim, 28 * 28)

    def forward(self, img):
        flattened = img.view(-1, 28 * 28)
        x = self.encoder(flattened)
        # sigmoid for scaling output from 0 to 1
        x = F.sigmoid(self.decoder(x))
        return x

    criterion = nn.MSELoss()
```

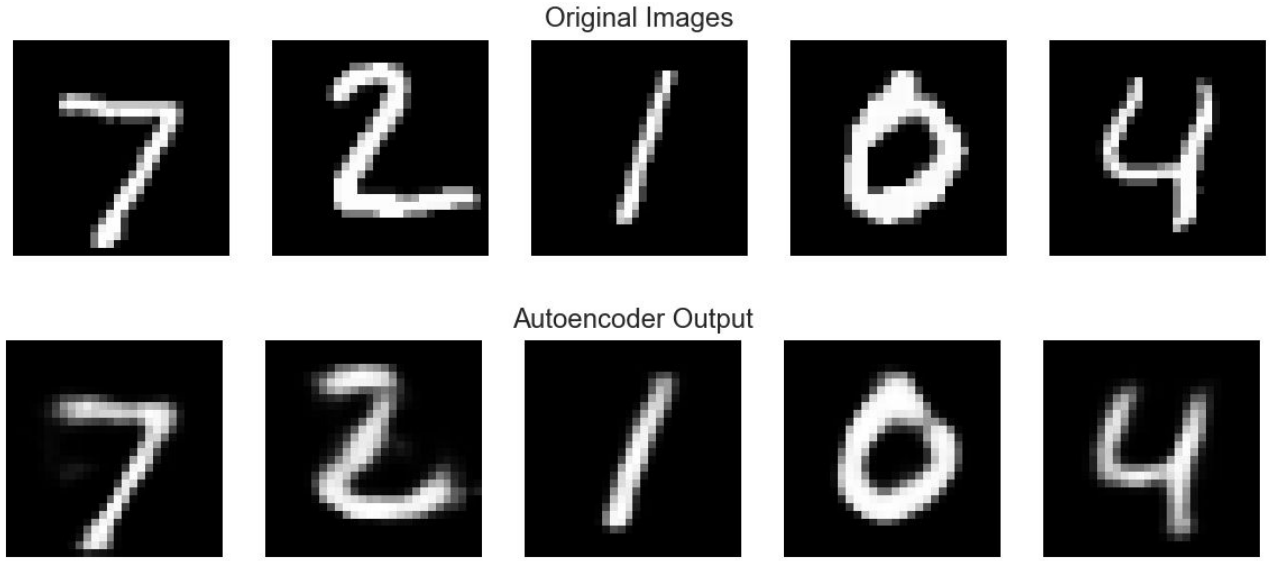
Stacked Autoencoders

- Autoencoders can have multiple hidden layers: stacked (deep) autoencoders
- Typically symmetrical with regards to the central coding layer.



Visualizing Reconstructions

One way to ensure that an autoencoder is properly trained is to compare the inputs and the outputs.



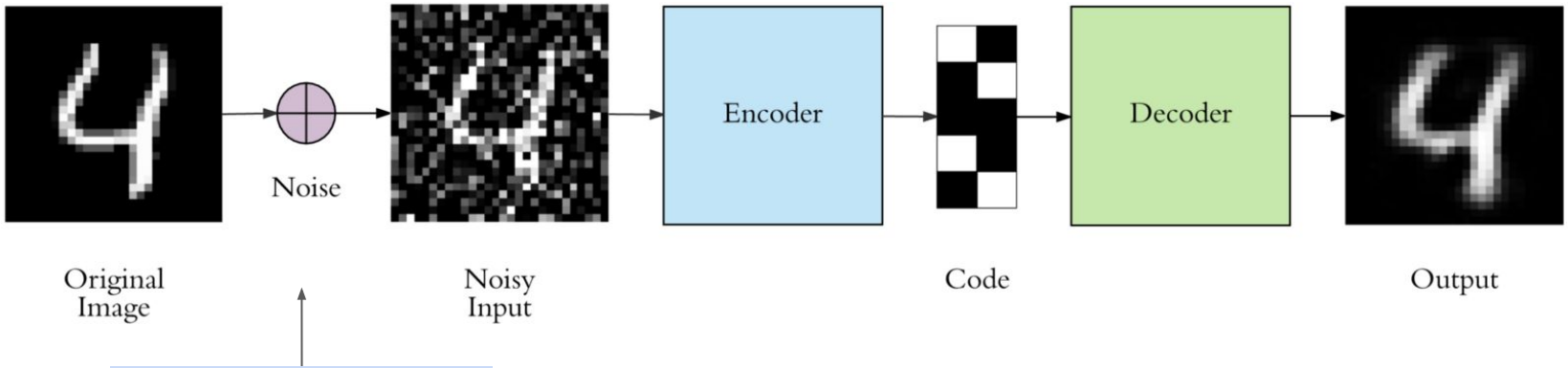
Perfect reconstruction does not rule out overfitting!

Denoising Autoencoders

Noise can be added to the input to force the model to learn useful features

Autoencoder is trained to recover the original, noise-free inputs.

Prevents it from trivially copying its inputs to its outputs, has to find patterns in the data



PyTorch Implementation

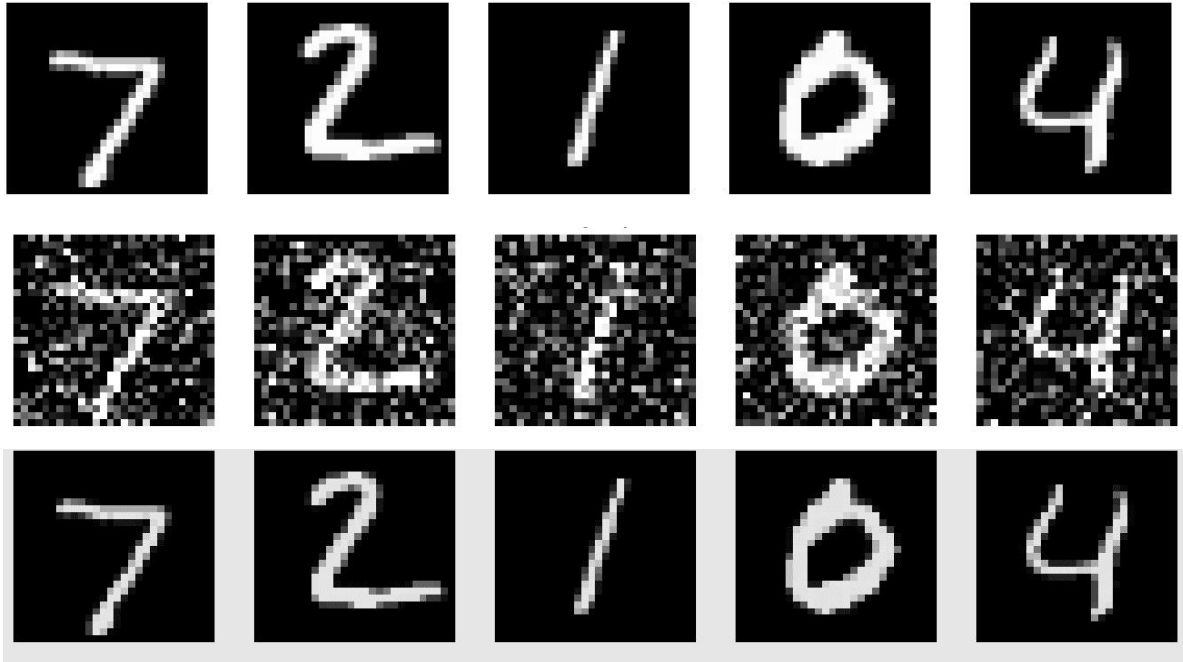
```
# how much noise to add to images
nf = 0.4

# add random noise to the input images
noisy_img = img + nf * torch.randn(*img.shape)

# Clip the images to be between 0 and 1
noisy_img = np.clip(noisy_img, 0., 1.)

# compute predicted outputs using noisy_img
outputs = model(noisy_img)

# the target is the original img
loss = criterion(outputs, img)
```

Denoising Autoencoders

Original image

Noisy image

Reconstructed image



Applications

- Feature Extraction
- Unsupervised Pre-training
- Dimensionality Reduction
- Anomaly detection → Autoencoders are bad at reconstructing outliers
- Generate new data

Generating New Images

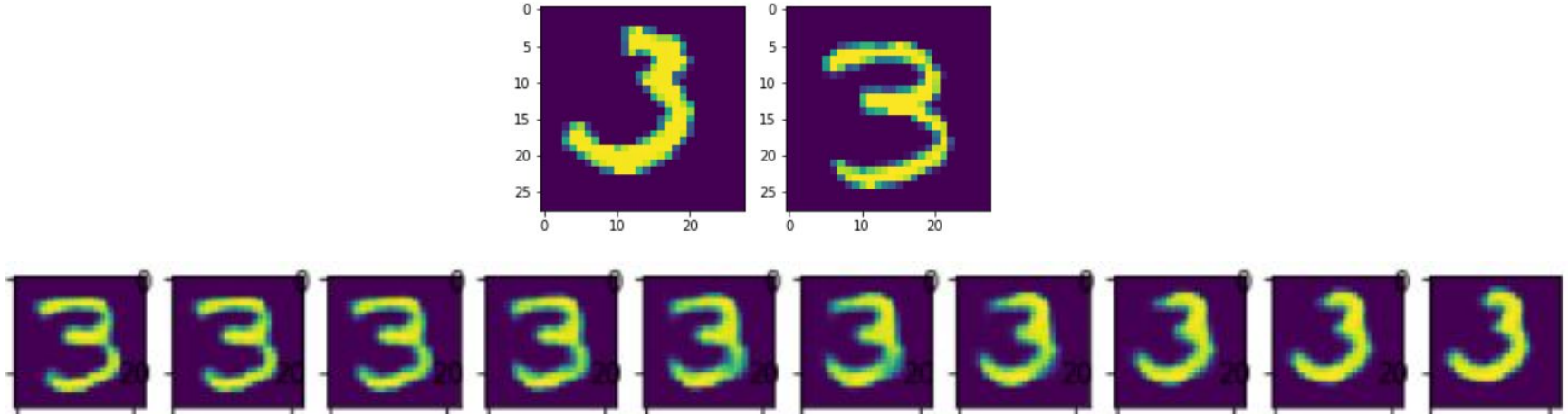
- Since we are drastically reducing the dimensionality of the image, there has to be some kind of structure in the codings (i.e. embedding space).
- That is, the network should be able to **save space by mapping similar images to similar embeddings** .
- Let's see how we can exploit this to allow us to generate new types of images.

New Images with Interpolation

First compute low-dimensional embeddings of two images.

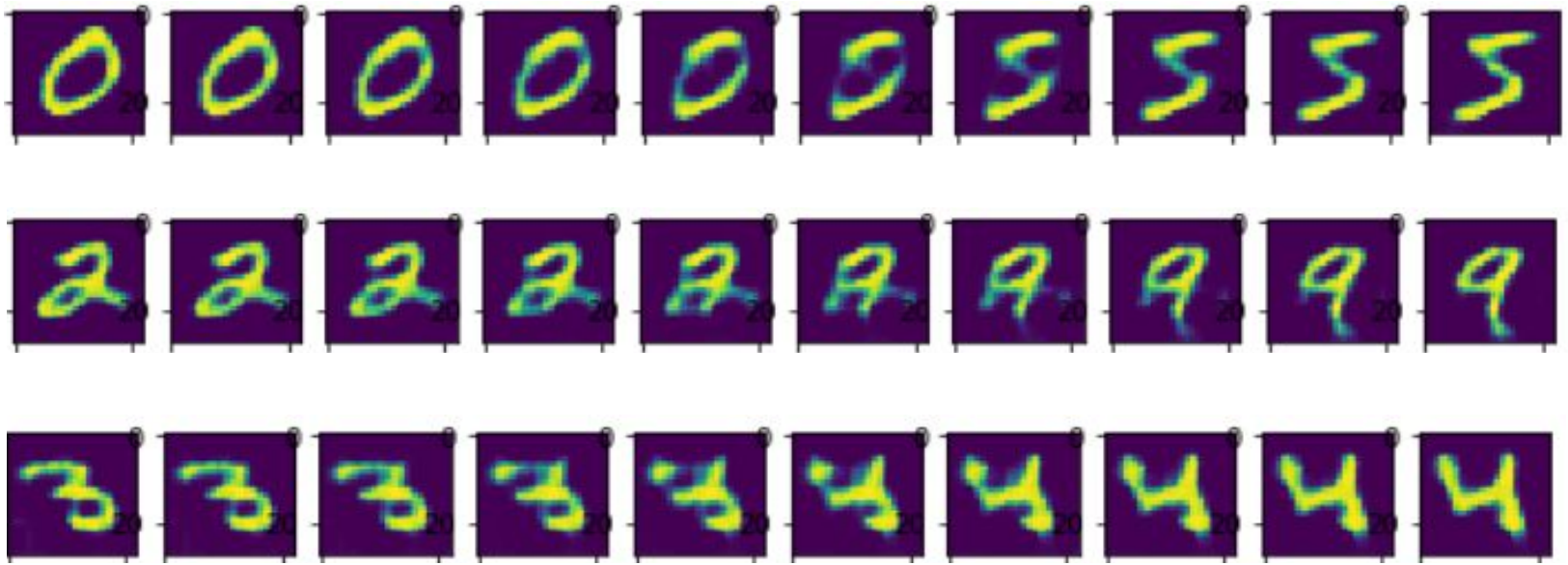
Then **interpolate between the two embeddings** and decode those as well!

Interpolated codings result in new images that are somewhere in between the two starting images.



Plotting Interpolated Codings

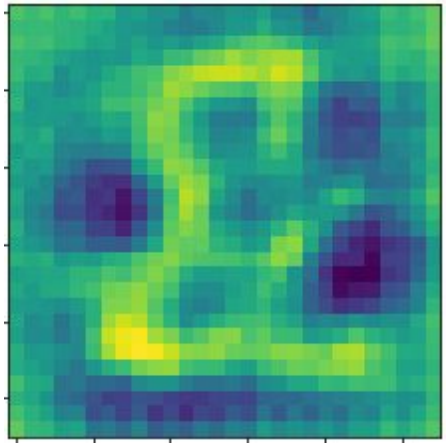
We can do this for other image combinations



Plotting Interpolated Codings

What if we randomly select a coding?

The latent space in autoencoders can become disjoint and non-continues



Variational AutoEncoders (VAE)

VAEs

They are quite different from the autoencoders we have discussed so far:

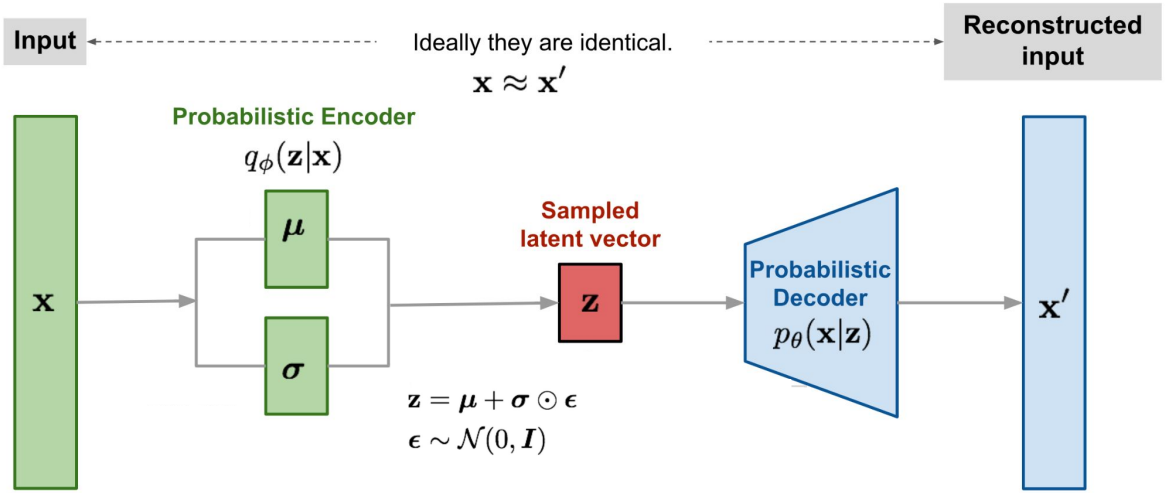
- **Probabilistic** → their outputs are partly determined by chance even after training
- **Generative** → they can generate new instances that look like they were sampled from the training set.

They impose a distribution constraint on the latent space to have a smooth space.

VAEs

Encoder generates a normal distribution with mean **μ** and a standard deviation **σ** instead of a fixed embedding.

An embedding is sampled from the distribution and decoder decodes the sample to reconstruct the input.



VAEs

We want the encoder distribution $q_\phi(z|x) = \mathcal{N}(\mu, \sigma)$ to be close to prior $p(z) = \mathcal{N}(0, I)$

We can use Kullback–Leibler (KL) divergence to measure the difference between two distributions P(X) and Q(X):

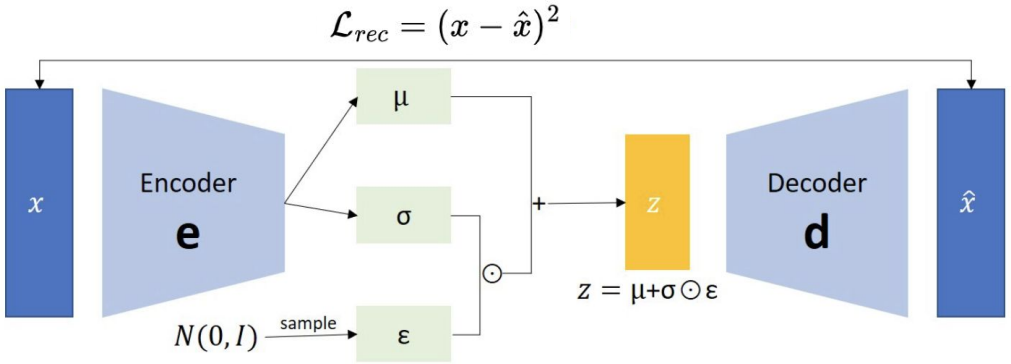
$$D_{KL}(P||Q) = \sum_{x \in X} p(x) \log \left(\frac{p(x)}{q(x)} \right)$$

If we plug-in the encoder distribution and the prior into KL-divergence of two multivariate Gaussians, we get:

$$D_{KL}(p|q) = \frac{1}{2} \sum_{i=1}^{N} \left[\mu_i^2 + \sigma_i^2 - \left(1 + \log(\sigma_i^2) \right) \right]$$

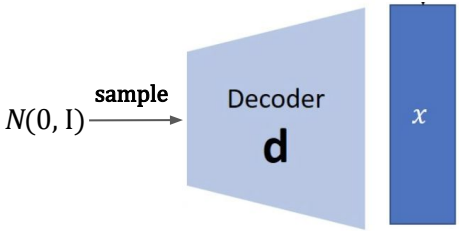
VAEs

Training

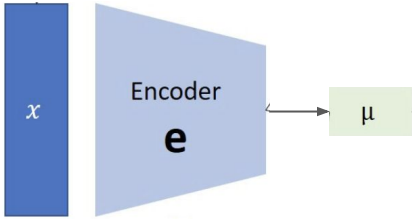


$$\mathcal{L}_{kl} = \frac{1}{2}\sum_{i=1}^{N}\left[\sigma_i^2 + \mu_i^2 - (1 + \log(\sigma_i^2))\right]$$

Generating



Embedding



Generating Data

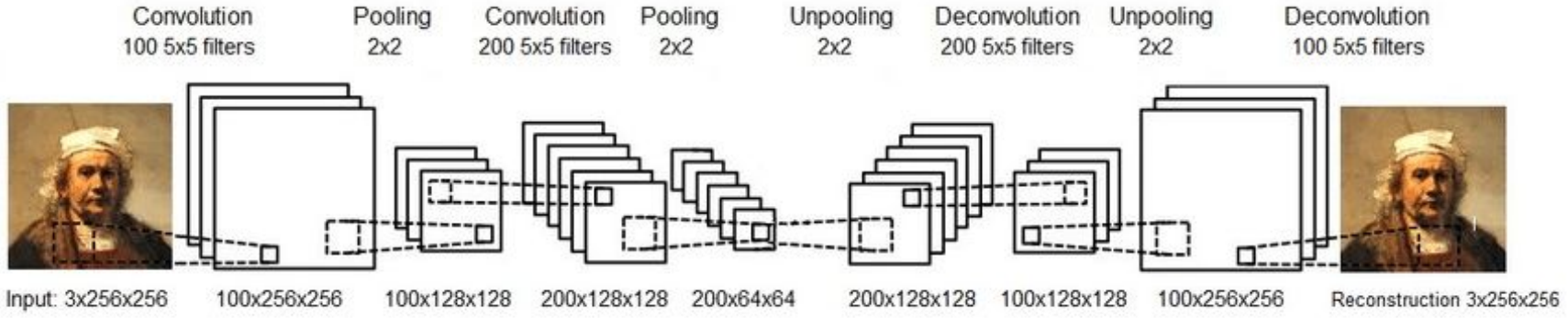
Generate images that look like handwritten digits by training a variational autoencoder.

Convolutional Autoencoders

Convolutional Autoencoder

Convolutional autoencoders take advantage of spatial information.

- **Encoder** → Learns visual embedding using convolutional layers
- **Decoder** → Up-samples the learned visual embedding to match the original size of the image.



Transposed Convolution

The opposite of the convolution is the transposed convolution (different from an inverse convolution).

They work with filters, kernels, padding, strides just as the convolution layers.

Instead of mapping KxK pixels to 1, they can map from 1 pixel to KxK pixels.

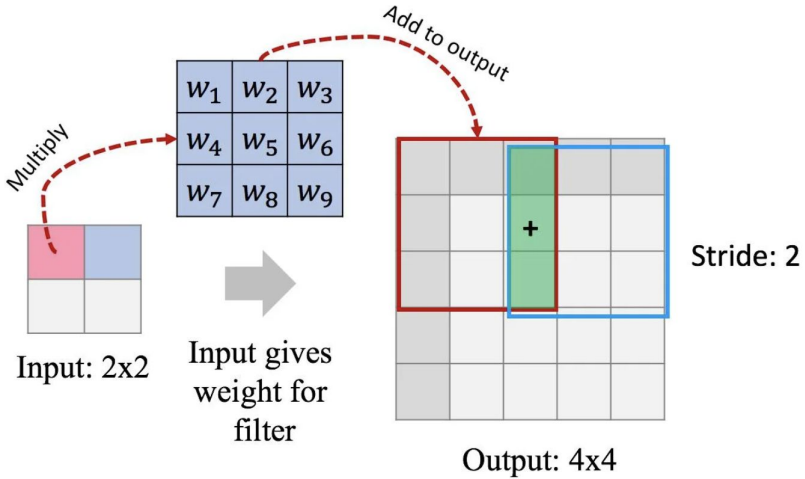
The kernels are learned just like normal convolutional kernels.

$$o = (i - 1) \times s + (k - 1) - 2p + op + 1$$

output dimension | input dimension | stride | kernel size | padding | output padding

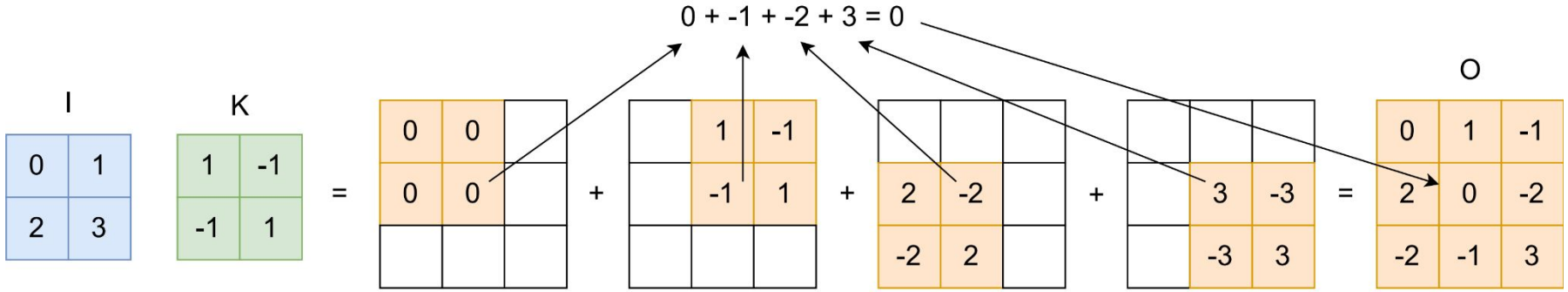
Transposed Convolution

1. Take each pixel of your input image
2. Multiply each value of your kernel with the input pixel to get a weighted kernel
3. Insert it in the output to create an image
4. Where the outputs overlap sum them



Transposed Convolution

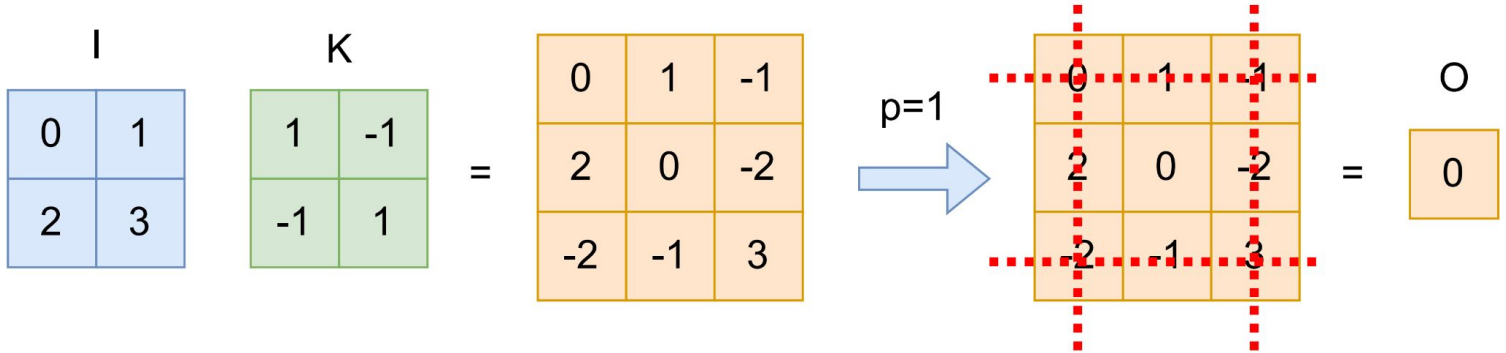
1. Take each pixel of your input image
2. Multiply each value of your kernel with the input pixel to get a weighted kernel
3. Insert it in the output to create an image
4. Where the outputs overlap sum them



Padding

The effect is the **opposite of what happens with the convolution** layers:

1. Compute the output as normal
2. Remove rows and columns around the perimeter



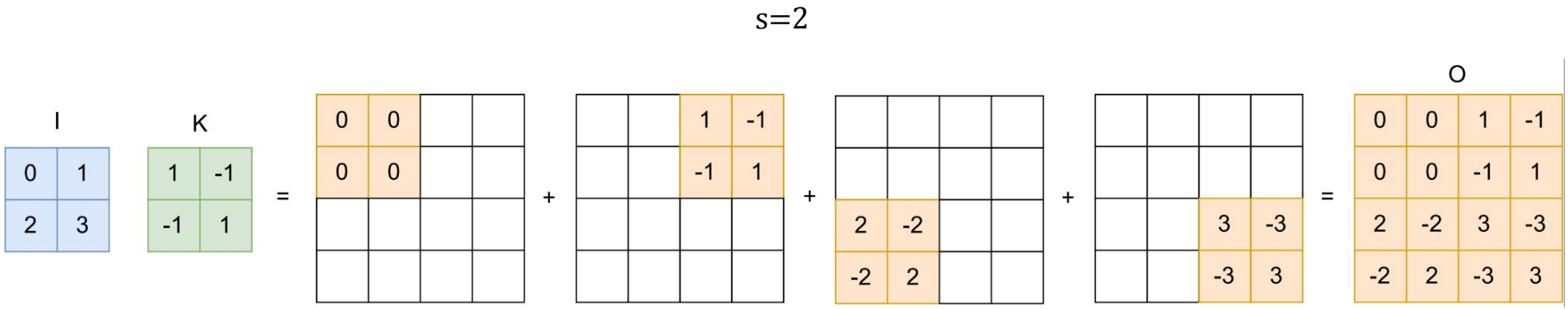
Output padding

- When stride > 1, Conv2d maps multiple input shapes to the same output shape.
- E.g. Inputs of size 7x7 and 8x8 both return an output of 3x3 for a kernel of size 3x3 with stride=2
- When applying the transpose convolution, it is ambiguous that which output shape to return, 7x7 or 8x8 for stride=2 transpose convolution.
- Output padding is provided to resolve this ambiguity by effectively increasing the calculated output shape on one side.
- It is only used to find output shape, but does not actually add zero-padding to output.

Strides

The effect is also the opposite from what happens with the convolution layers

Increasing the stride results in an increase in the upsampling effect.



PyTorch Implementation

A convolution transpose layer with the exact same specifications as the convolution layer would have the **reverse effect on the shape** .

```
conv = nn.Conv2d(in_channels=8,
                 out_channels=8,
                 kernel_size=5)

x = torch.randn(2, 8, 64, 64)
y = conv(x)
y.shape
```

```
torch.Size([2, 8, 60, 60])
```

```
convt = nn.ConvTranspose2d(in_channels=8,
                           out_channels=8,
                           kernel_size=5)

convt(y).shape # should be same as x.shape
```

```
torch.Size([2, 8, 64, 64])
```

PyTorch Implementation

We also have the option of including convolution **transpose padding** :

```
convt = nn.ConvTranspose2d(in_channels=16,
                           out_channels=8,
                           kernel_size=5,
                           padding=2)
x = torch.randn(32, 16, 64, 64)
y = convt(x)
y.shape
```

```
torch.Size([32, 8, 64, 64])
```

$$o = (i - 1) \times s + (k - 1) - 2p + op + 1$$

PyTorch Implementation

We can add a stride to the convolution to increase our resolution!

```
convt = nn.ConvTranspose2d(in_channels=16,
                           out_channels=8,
                           kernel_size=5,
                           stride=2,
                           padding=2)

x = torch.randn(32, 16, 64, 64)
y = convt(x)
y.shape
```

```
torch.Size([32, 8, 127, 127])
```

$$o = (i-1) \times s + (k-1) - 2p + op + 1$$

PyTorch Implementation

Output padding is another type of padding that adds an additional row and column to the output. Easy to mix it up with padding.

```
convt = nn.ConvTranspose2d(in_channels=16,
                           out_channels=8,
                           kernel_size=5,
                           stride=2,
                           padding=2,
                          output_padding=1)

x = torch.randn(32, 16, 64, 64)
y = convt(x)
y.shape
```

```
torch.Size([32, 8, 128, 128])
```

$$o = (i-1) \times s + (k-1) - 2p + op + 1$$

PyTorch Implementation

```
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 16, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(16, 32, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 64, 7)
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(64, 32, 7),
            nn.ReLU(),
            nn.ConvTranspose2d(32, 16, 3, stride=2, padding=1,output_padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(16, 1, 3, stride=2, padding=1, output_padding=1),
            nn.Sigmoid()
        )
```

PyTorch Implementation

```
def forward(self, x):
    x = self.encoder(x)
    x = self.decoder(x)
    return x

def embed(self, x)
    return self.encoder(x)

def decode(self, e):
    return self.decode(e)
```

Pre-training with Autoencoders

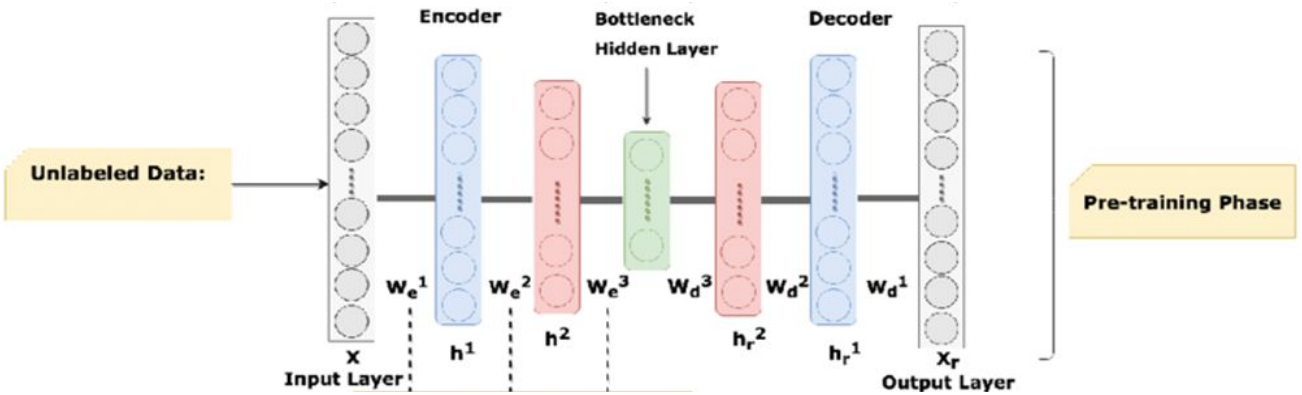
Pre-training with Autoencoders

Previously we discussed how **transfer learning** could use features obtained from ImageNet data to improve classification on other image tasks.

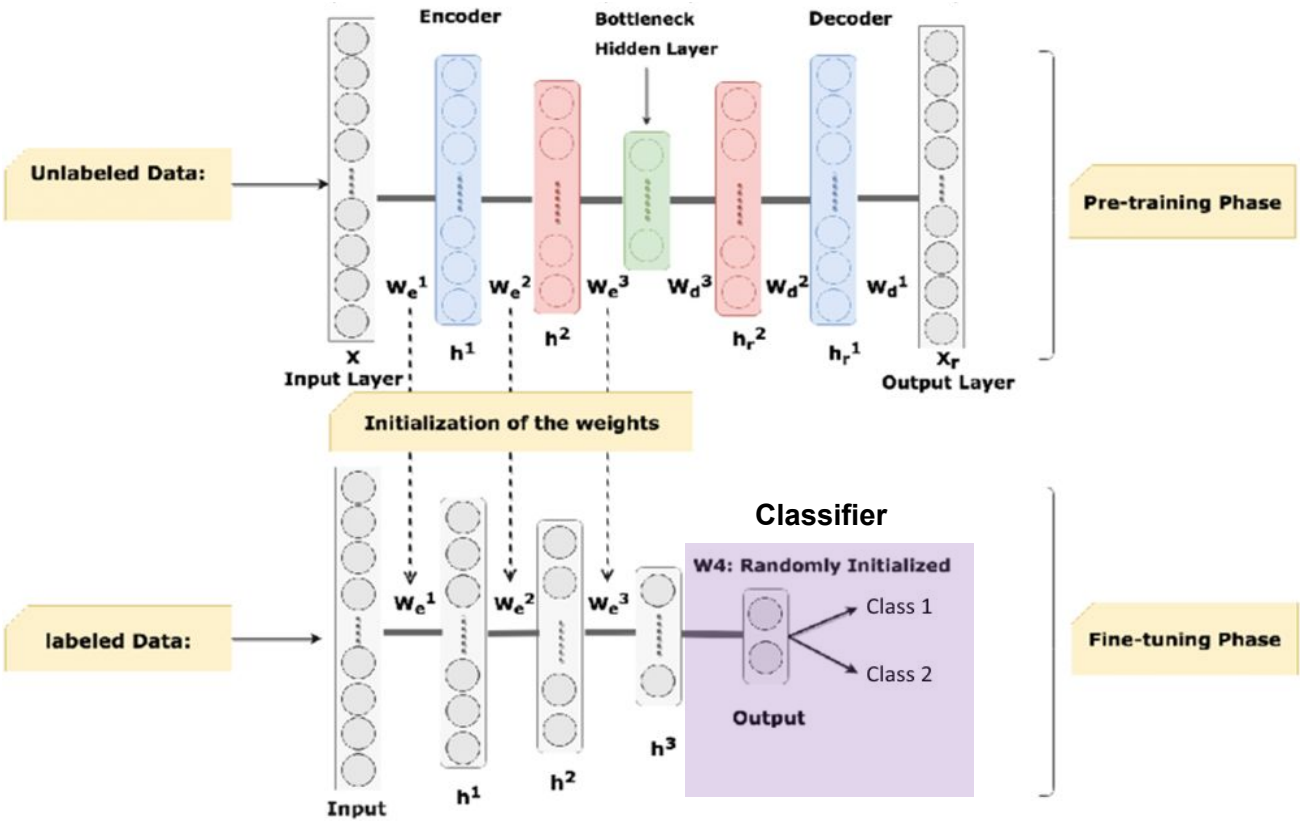
- Assumption that the ImageNet data is similar in the new task.
- If the new task is to detect new objects from similar images, then transfer learning makes sense.

Autoencoders can achieve similar results by pretraining on large set of unlabeled data, same type of data, just missing labels

Pre-training with Autoencoders



Pre-training with Autoencoders



Self-Supervised Learning

Self-supervised learning with pretext tasks

What if we can cast unsupervised learning into supervised setting?

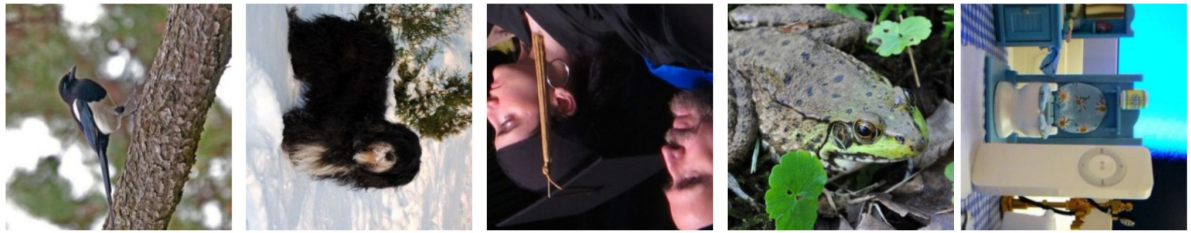
define **proxy supervised tasks** such that:

- The labels are generated automatically for free
- Solving the task, requires the model to “understand” the content

The challenge is devising the tasks such that they enforce the model to learn robust representations.

RotNet

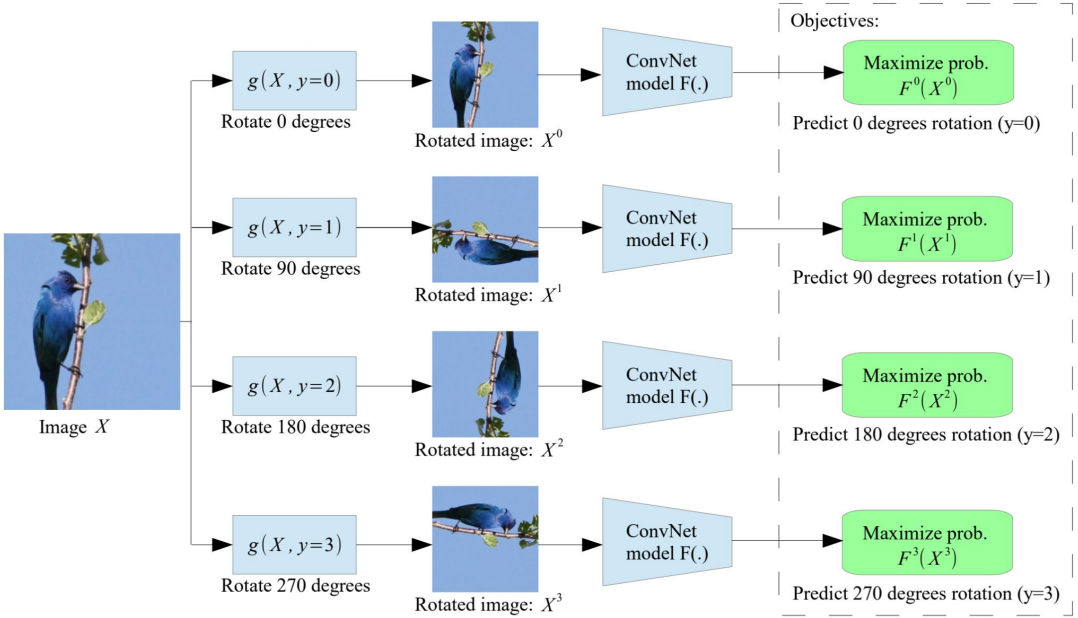
Idea: Rotate images randomly by 0, 90, 180, or 270 degrees and make the model to *predict the rotation angle*



if someone is not aware of the concepts of the objects depicted in the images, they cannot recognize the rotation that was applied to them.

RotNet

The task is multiclass classification with 4 classes (cross-entropy loss) with free labels being generated automatically



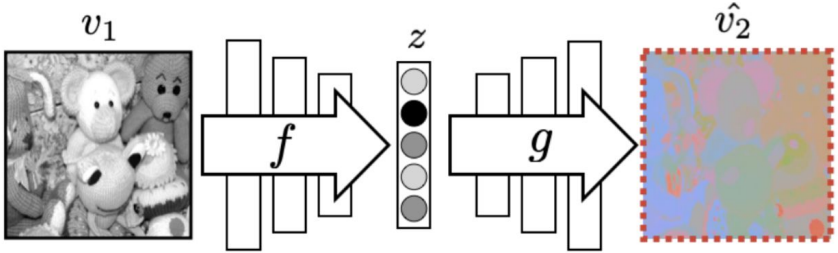
Contrastive Learning

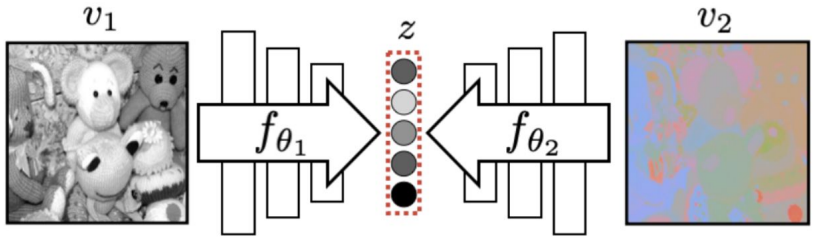
Autoencoding methods :

- Reconstruct input
- Compute the loss in output space
- Compress all the details

Contrastive methods:

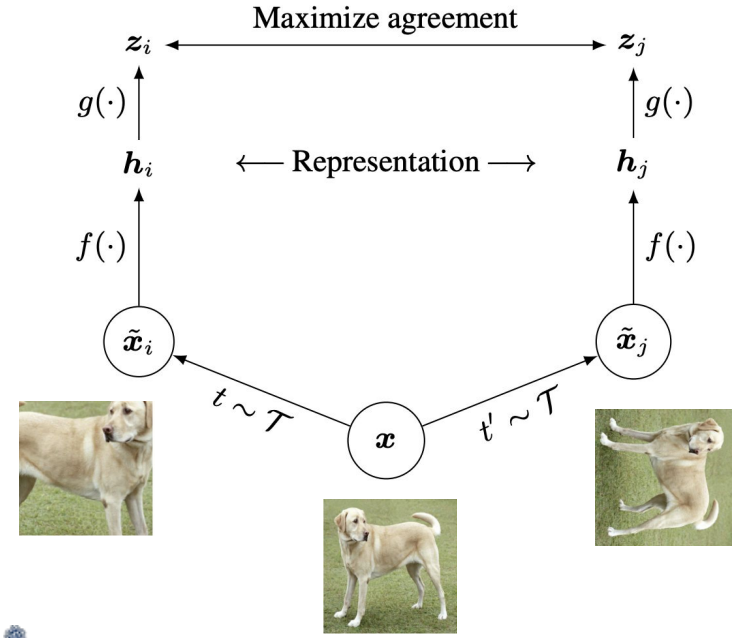
- Contrast pair of positive/negative samples
- Compute the loss in embedding space
- Compress relevant information
- Requires lots of negative examples





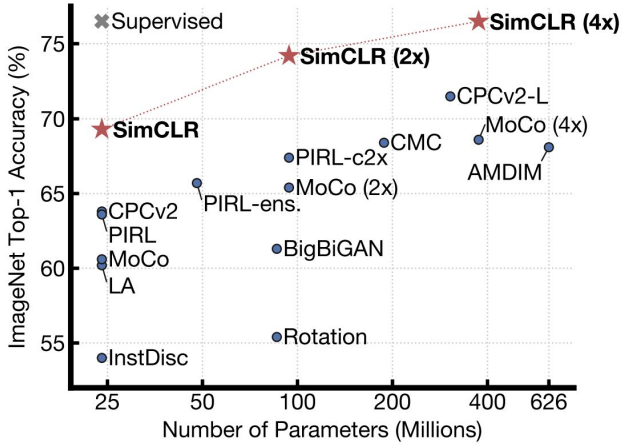
SimCLR

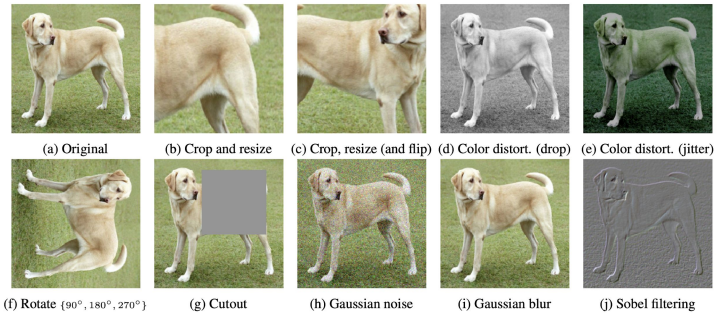
$$\ell_{i,j} = -\log \frac{\exp(\text{sim}(\boldsymbol{z}_i, \boldsymbol{z}_j)/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(\text{sim}(\boldsymbol{z}_i, \boldsymbol{z}_k)/\tau)}$$



MLP

CNN





A Simple Framework for Contrastive Learning of Visual Representations

Ting Chen, Simon Kornblith, Mohammad Norouzi, Geoffrey Hinton

Questions?