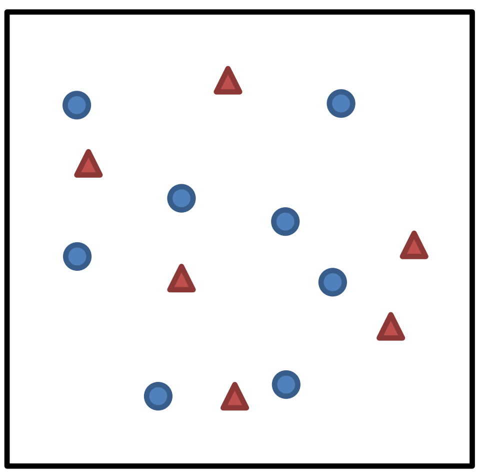
Neural Networks and Deep Learning

Autoencoders

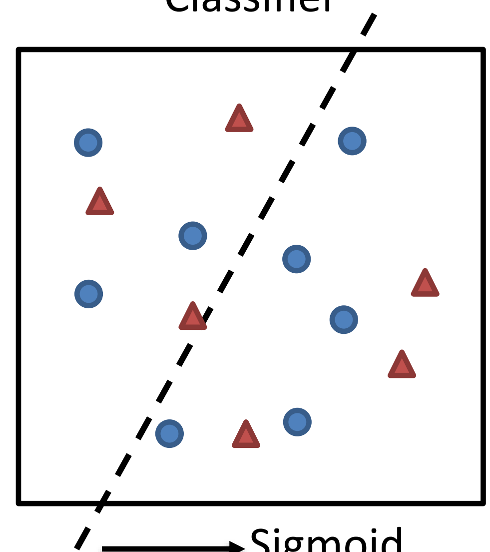
MOTIVATION

Drawing lines in space

Feature space

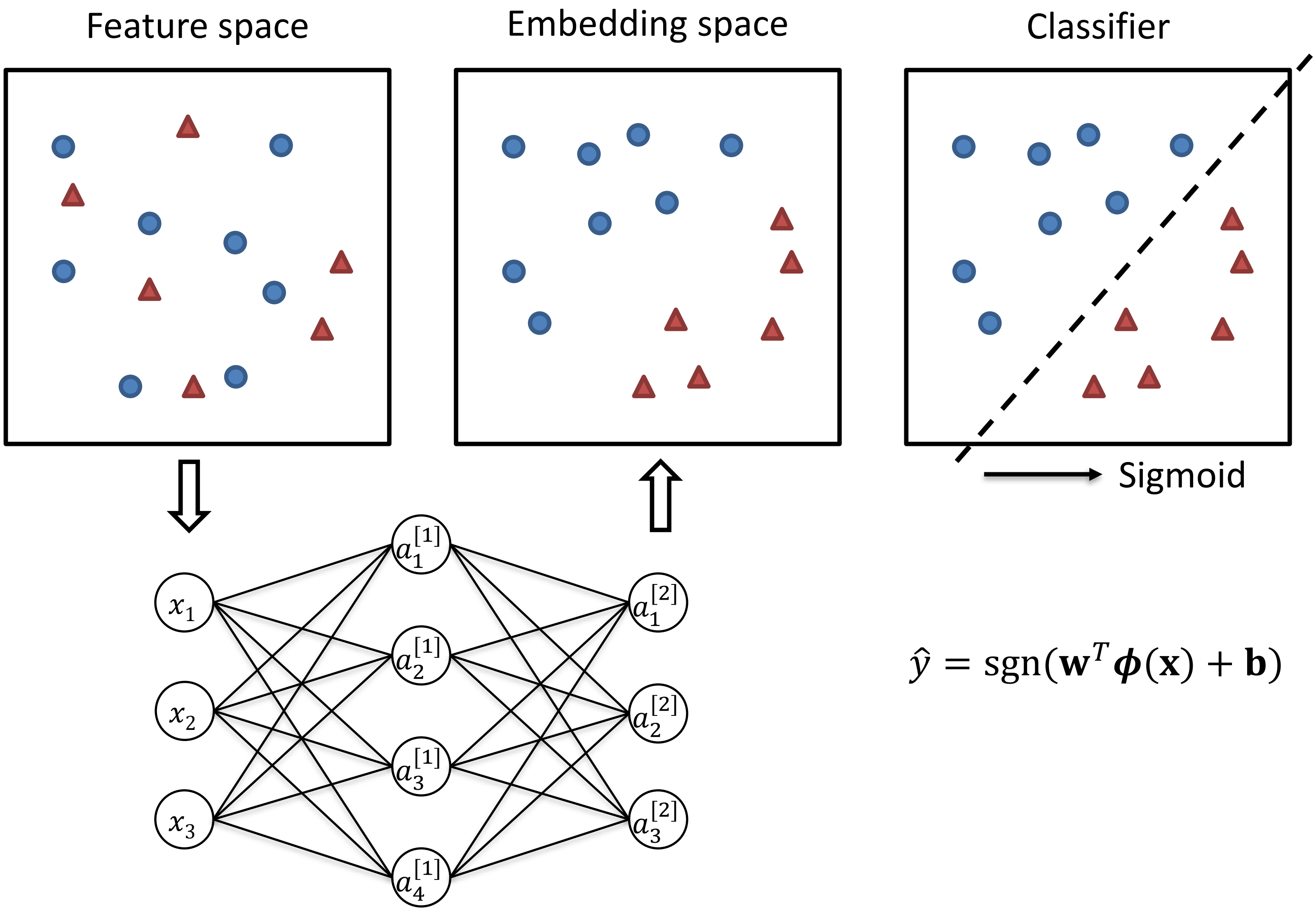


Classifier



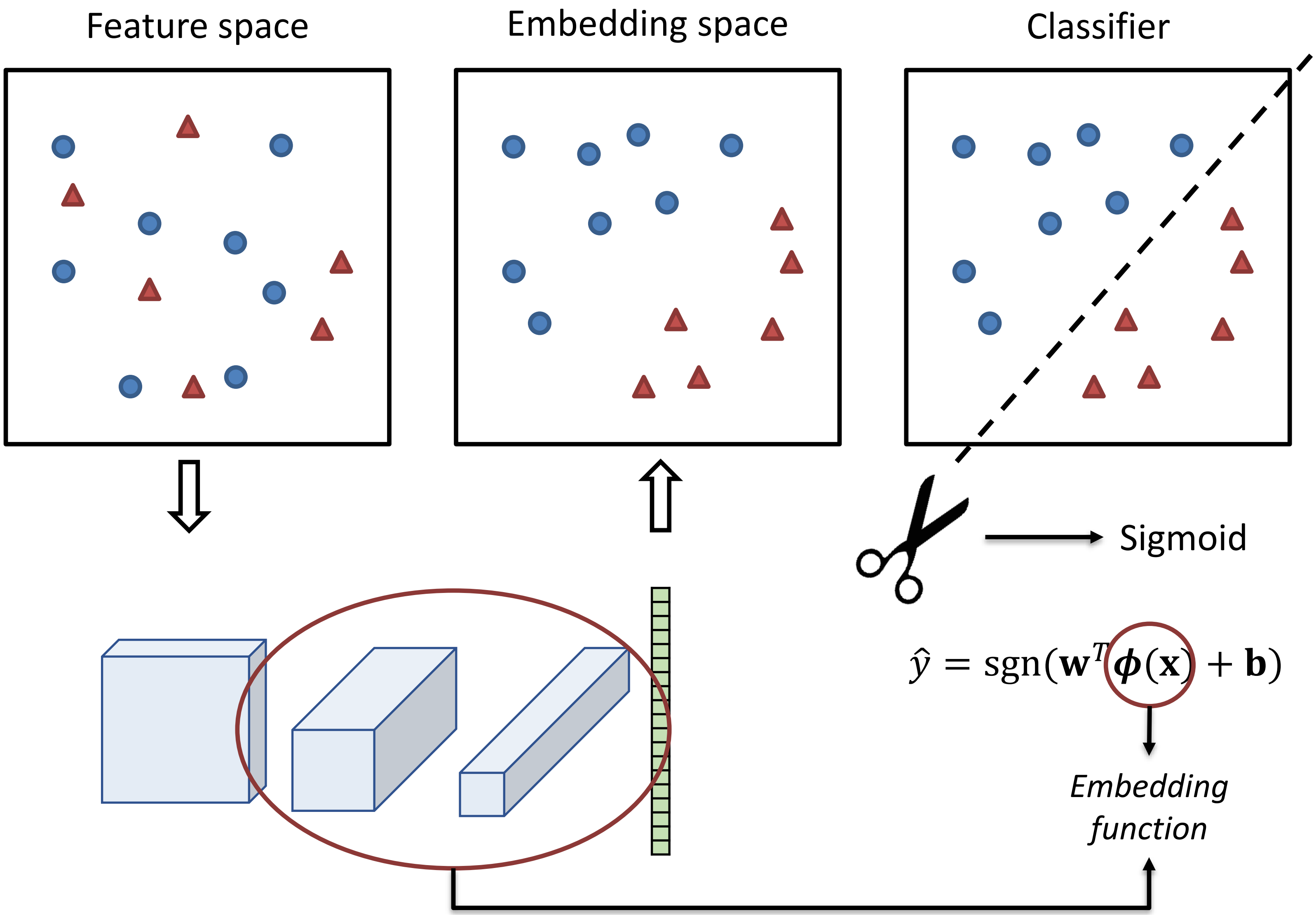
$$\hat{y} = \mathrm{sgn}(\mathbf{w}^T\mathbf{x} + \mathbf{b})$$

Drawing lines in space

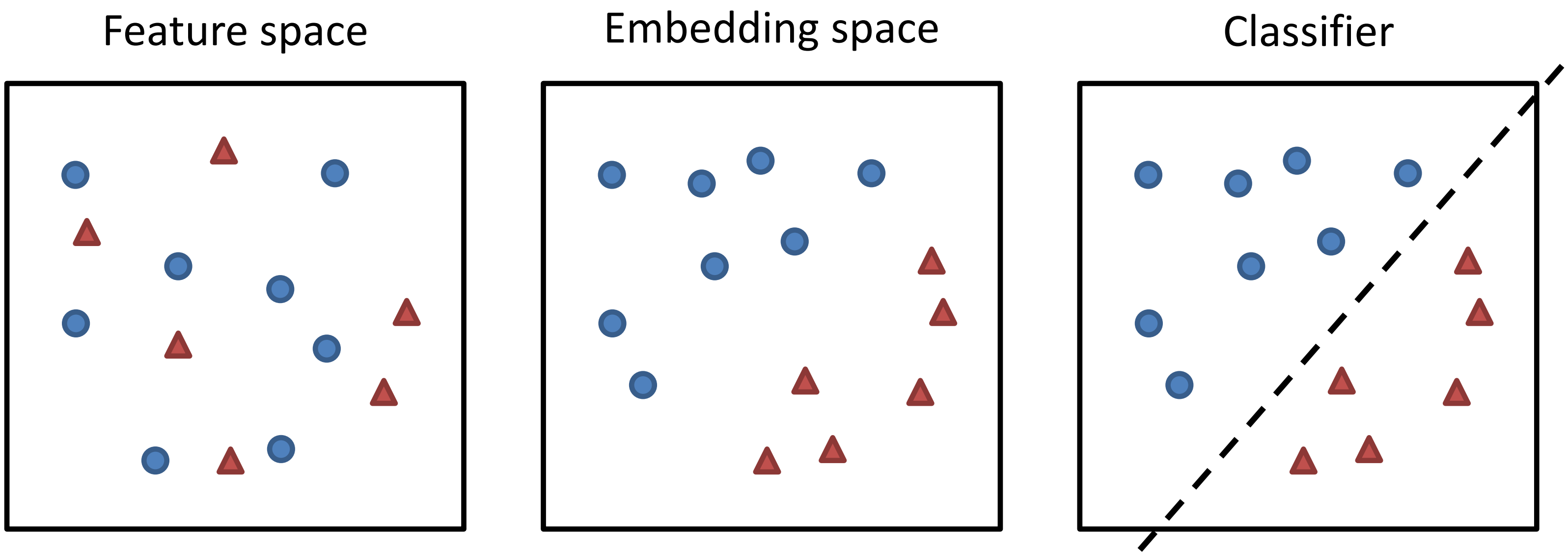


$$\hat{y} = \mathrm{sgn}(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}) + \mathbf{b})$$

Drawing lines in space



Embedding



An **embedding** is a new (intermediate) representation of our original features (data).

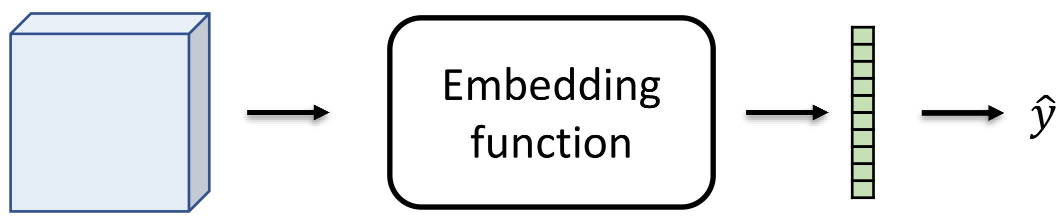
The final **embedding space** is typically a lower-dimensional space than our original features.

The embedding process converts our data in a way that it is **easier to solve the problem at hand**.

Embedding

Embedding
Feature extraction
Projecting data into a new space
Representation learning
Feature design / learning
…

Are all equivalent expressions
(in our scenario)



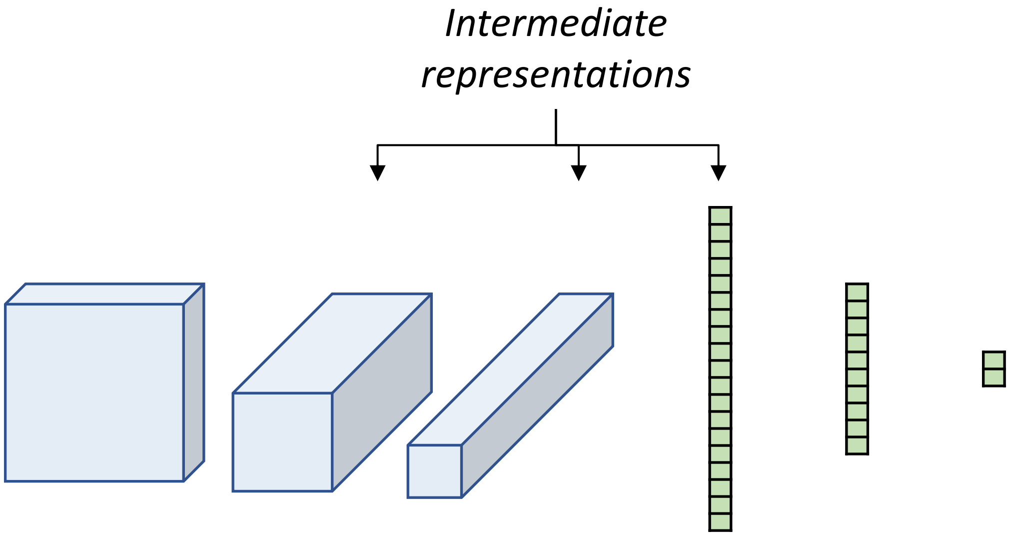
Feature design / learning can be driven by data and/or previous knowledge

Analytical (PCA, LDA, TSNE, …)
Hand-crafted (BoW, SIFT, SURF, …)
Learnt

and reused across models

Embedding

In deep neural networks, we repeatedly embed (project) our data into new spaces

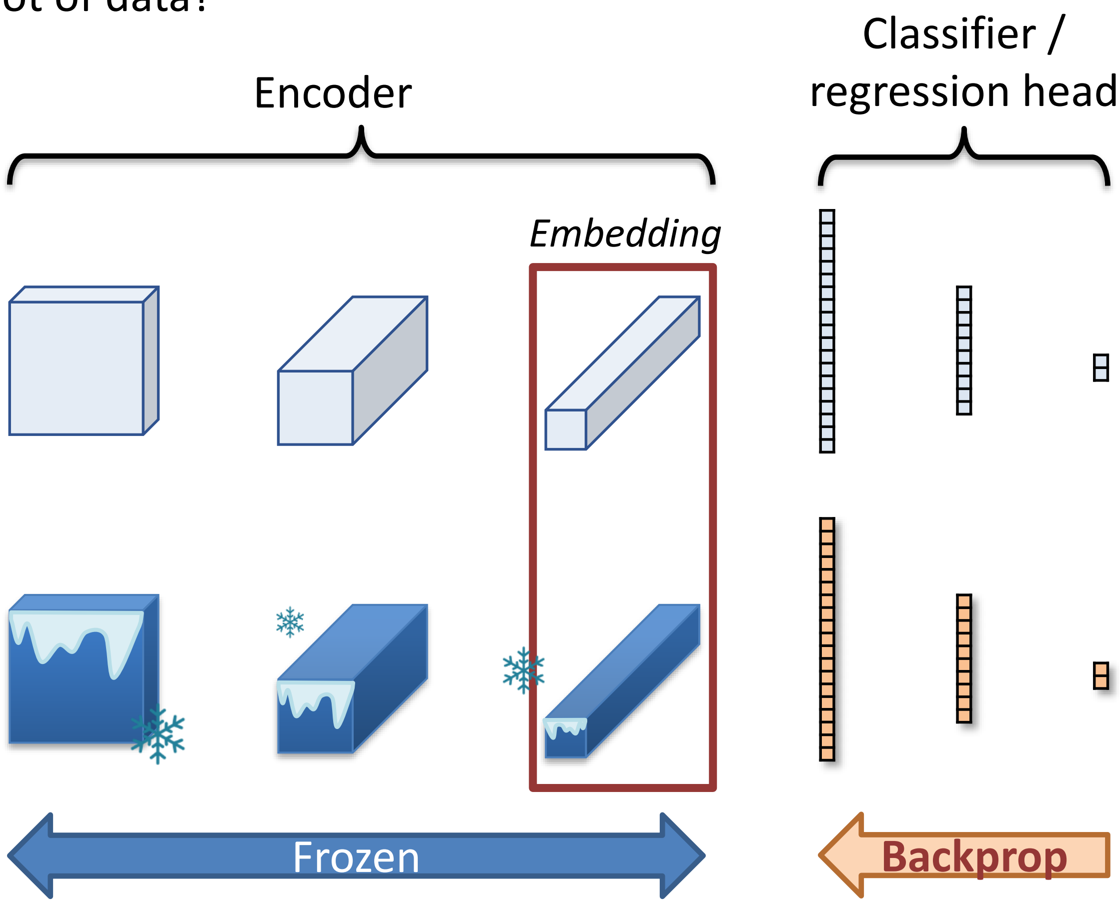


The right embedding to use, depends on the problem we are solving.

In Deep Learning, this process is **end-to-end** (task driven), and **data driven**

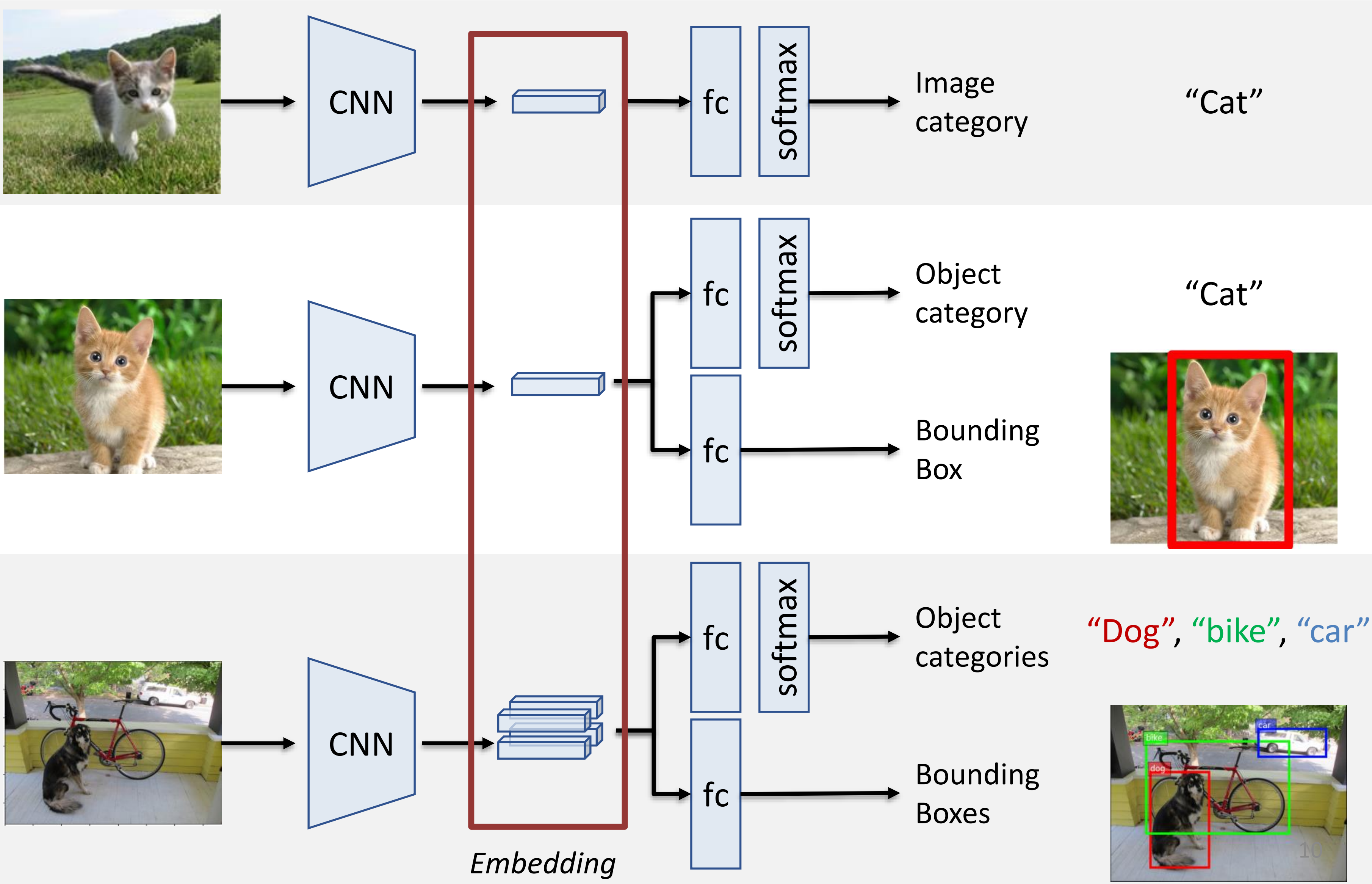
Reusing embeddings

Learning features is data hungry… How can we get an embedding when we lack a lot of data?



Transfer learning: re-use existing embedding learnt from a “well defined” task

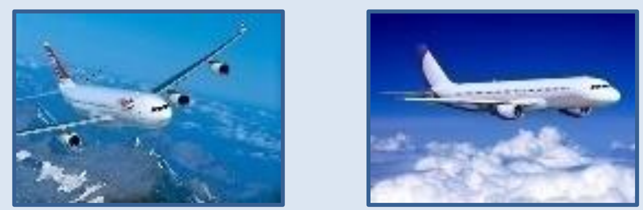
Same pattern...



Supervised Learning

Semi-Supervised Learning

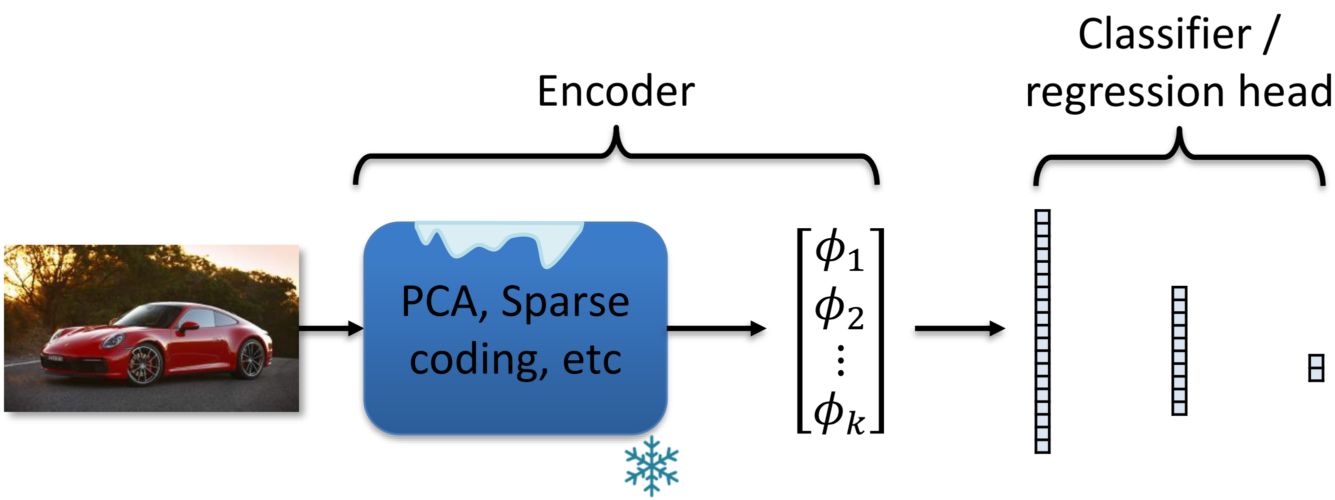
Cars



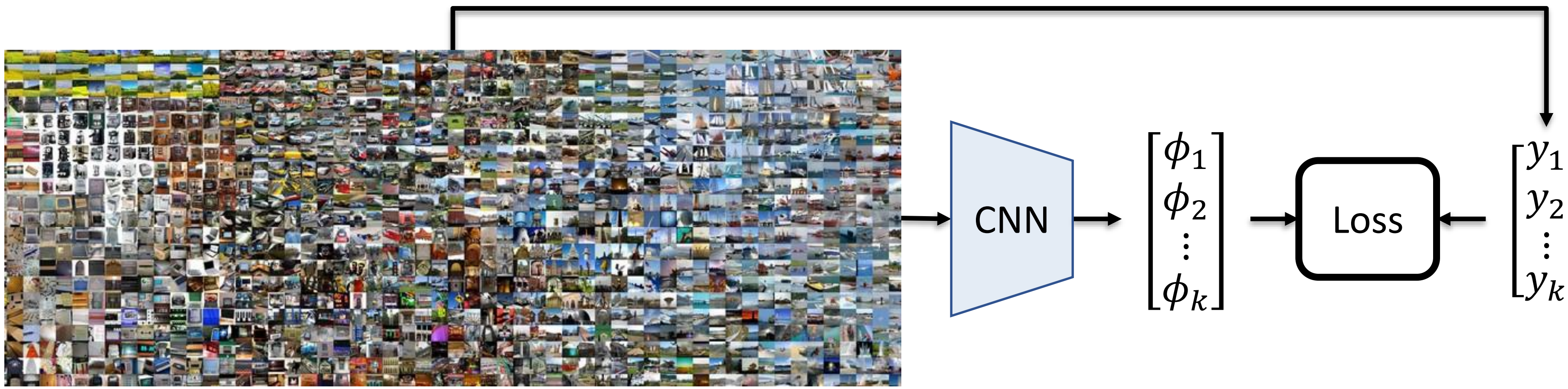
Planes

Unsupervised learning

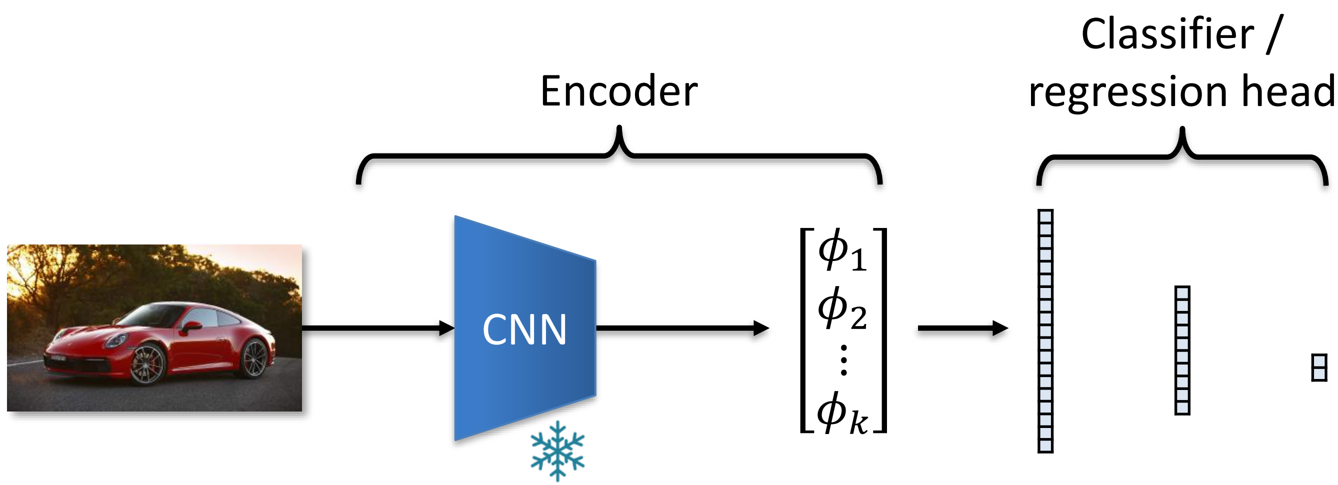
Learning the underlying structure of the data over non-labelled samples, would make it easier to learn a supervised model afterwards



Unsupervised learning



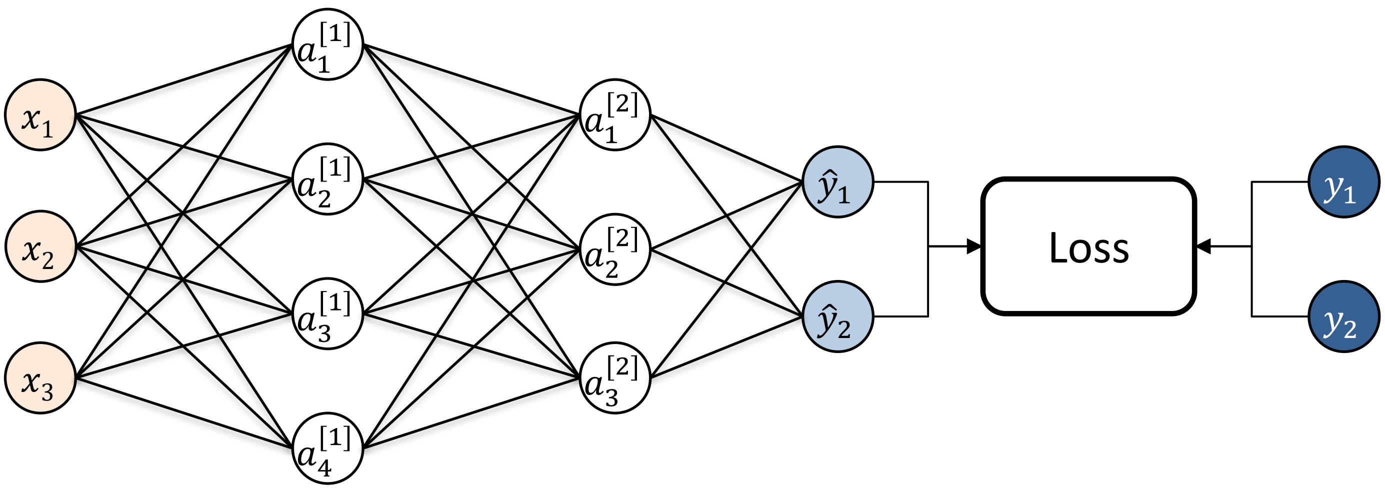
A different idea, is to somehow create a supervisory signal from data that do not have annotations…



How can we create useful embeddings without annotated data?

AUTOENCODERS

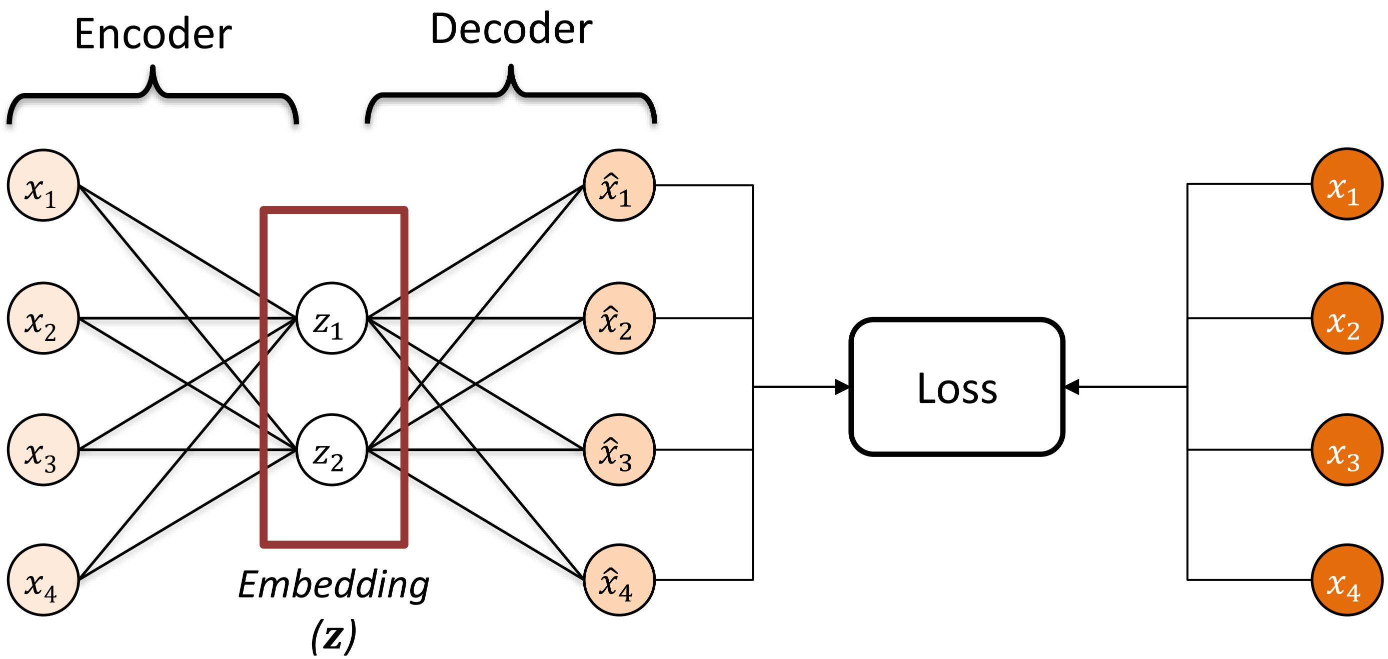
Training a neural network



Given a training set: $(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), (\mathbf{x}^{(2)}, \mathbf{y}^{(2)}), (\mathbf{x}^{(3)}, \mathbf{y}^{(3)}), \ldots$

Adjust parameters $\mathbf{W}$ (of every layer) to make: $\hat{\mathbf{y}}^{(i)} = f_{\mathbf{W}}\left(\mathbf{x}^{(i)}\right) \approx \mathbf{y}^{(i)}$

Autoencoder

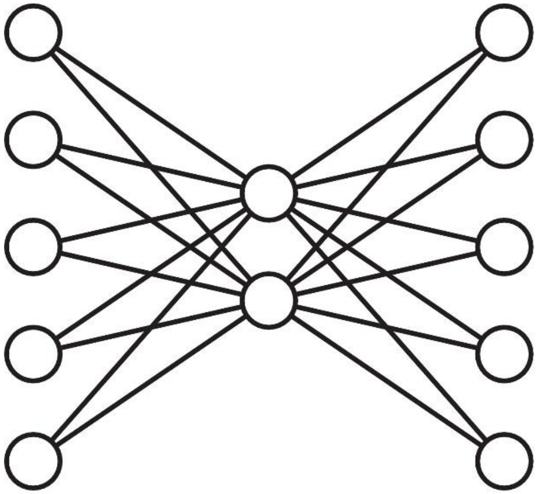


Given a training set without annotations: $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)}, \ldots$

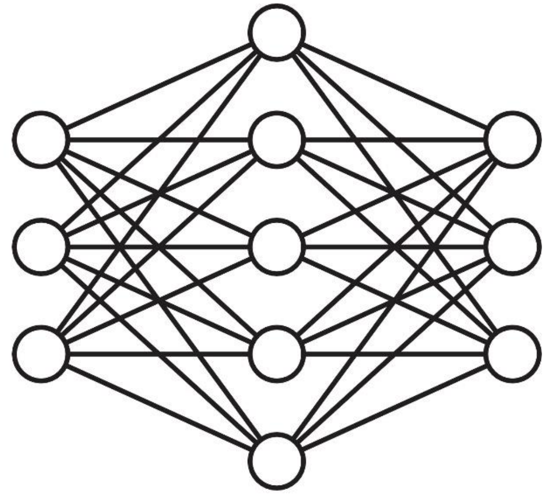
Adjust parameters $\mathbf{W}$ (of every layer) to make: $f_{\mathbf{W}}\left(\mathbf{x}^{(i)}\right) \approx \mathbf{x}^{(i)}$

How can we learn a useful representation ($\mathbf{z}$)?

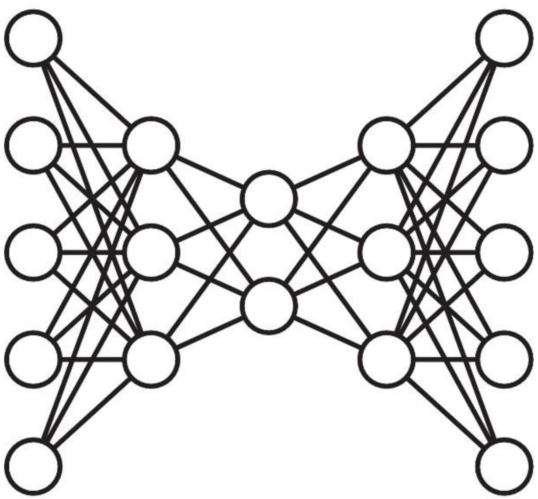
Types of autoencoders



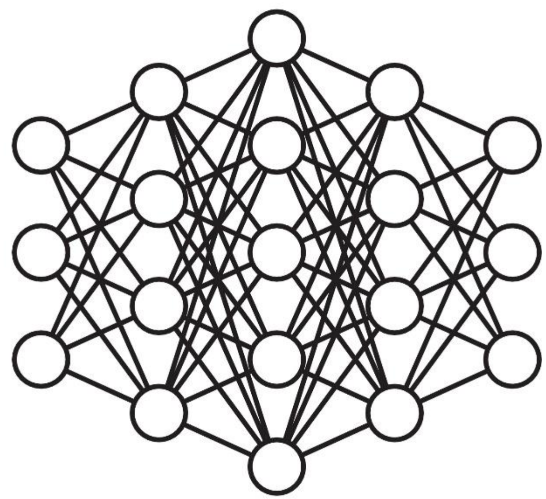
(a) Shallow undercomplete



(b) Shallow overcomplete



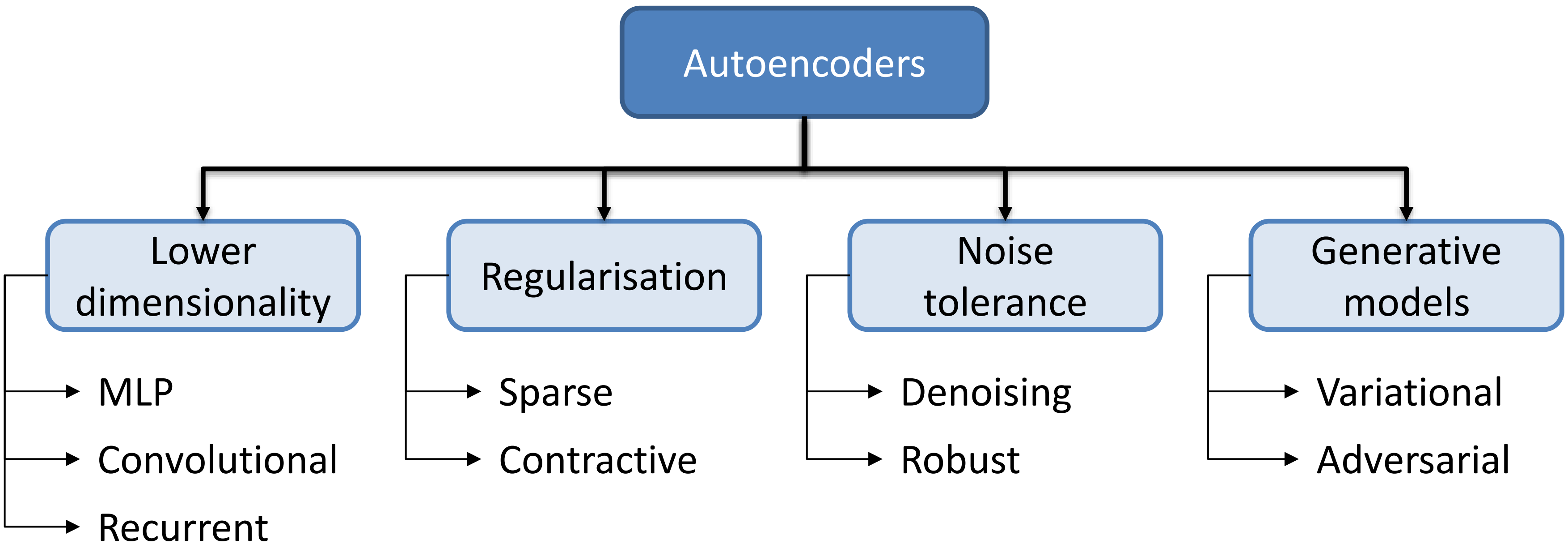
(c) Deep undercomplete



(d) Deep overcomplete

Credit [David Charte]

Autoencoders taxonomy



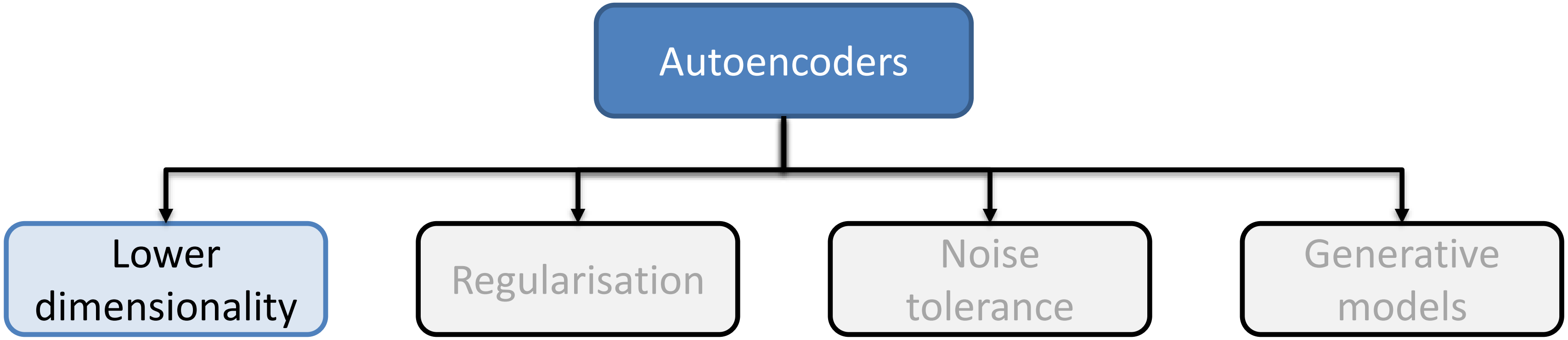
The network is trained to output (reconstruct) the input.

This has a trivial solution (learn the identify function) unless we

- constrain the **number of units** in embedding layer (compressed representation)
- constrain the embedding layer to be **sparse**
- introduce a **small change** in the input and learn to undo it
- force some **particular distribution** for the embeddings

https://towardsdatascience.com/autoencoders-ae-a-smart-way-to-process-your-data-using-unsupervised-neural-networks-9661f93a8509

Adapted from David Charte



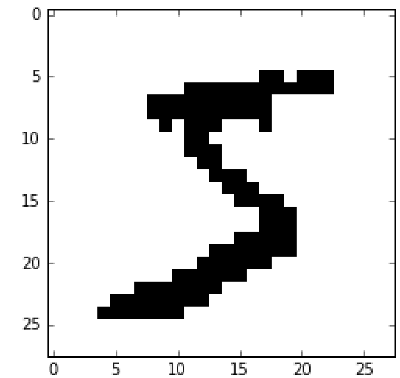
Compression – learning embeddings of lower dimensionality

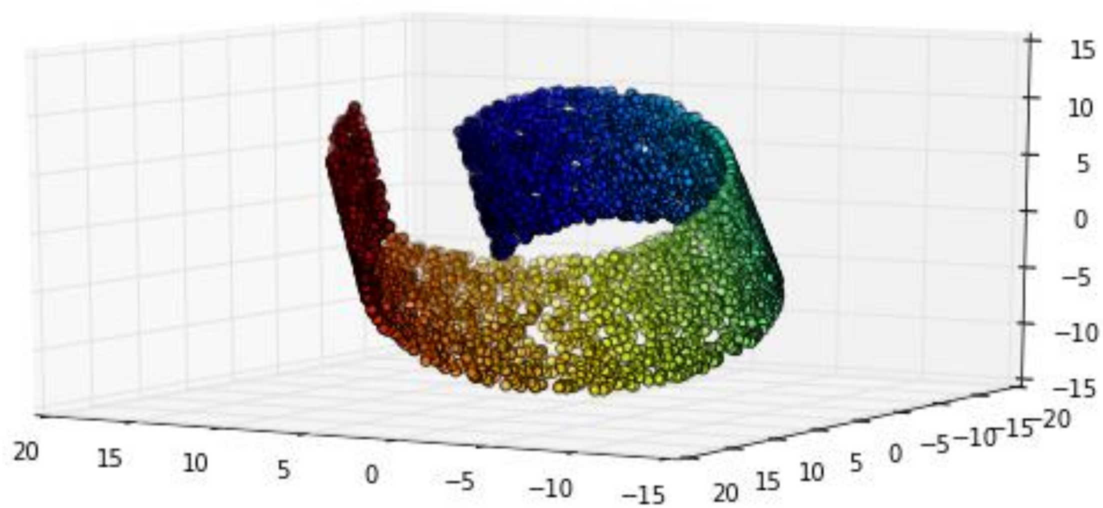
COMPRESSION

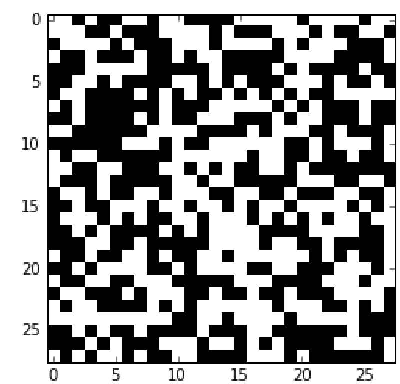
Motivation

Think about the handwritten digits (MNIST) data

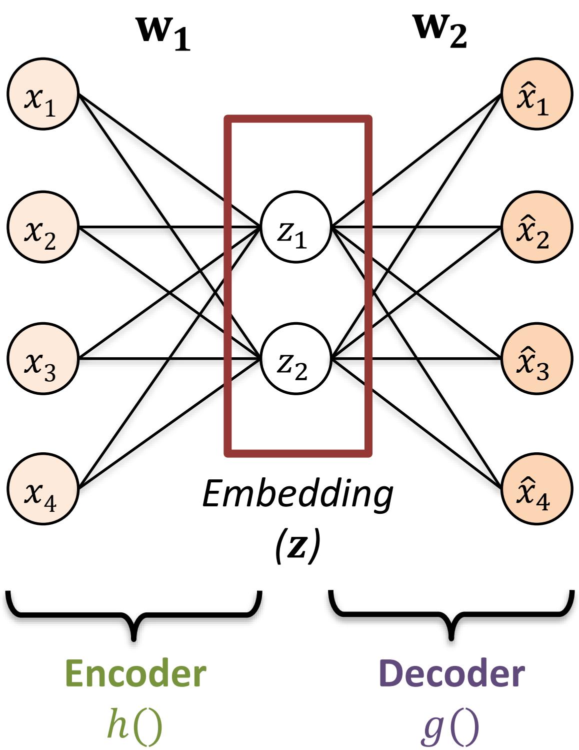
- 28 x 28 bitmaps
- Each pixel can either be black or white: $\{0, 1\}^{784}$ possible events
- We will never see most of the events
- The actual digits are a tiny fraction of the possible events
- It should be possible to describe our data with less features







Autoencoders: compression

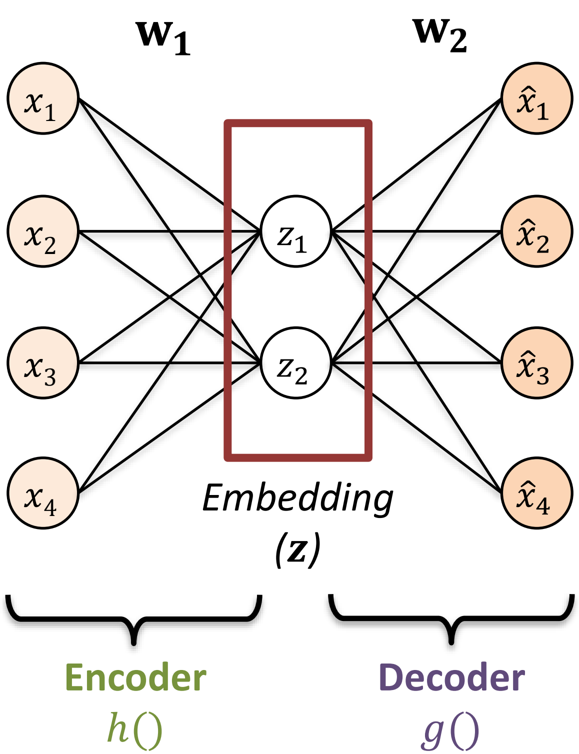


$$J(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m} L^{(i)}(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m} L\big(f_{\mathbf{w}}(\mathbf{x}^{(i)}), \mathbf{y}^{(i)}\big)$$

$$\downarrow$$

$$J(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m} L^{(i)}(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m} L\left(g_{\mathbf{w_2}}\left(h_{w_1}(\mathbf{x}^{(i)})\right), \mathbf{x}^{(i)}\right)$$

Autoencoders: compression



$$J(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m} L^{(i)}(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m} L\left(g_{\mathbf{w_2}}\left(h_{w_1}(\mathbf{x}^{(i)})\right), \mathbf{x}^{(i)}\right)$$

$$L\left(g_{\mathbf{w_2}}\left(h_{w_1}(\mathbf{x}^{(i)})\right), \mathbf{x}^{(i)}\right) = L_{MSE}\left\|\mathbf{x}^{(i)} - \hat{\mathbf{x}}^{(i)}\right\|$$

Or with L2 regularization:

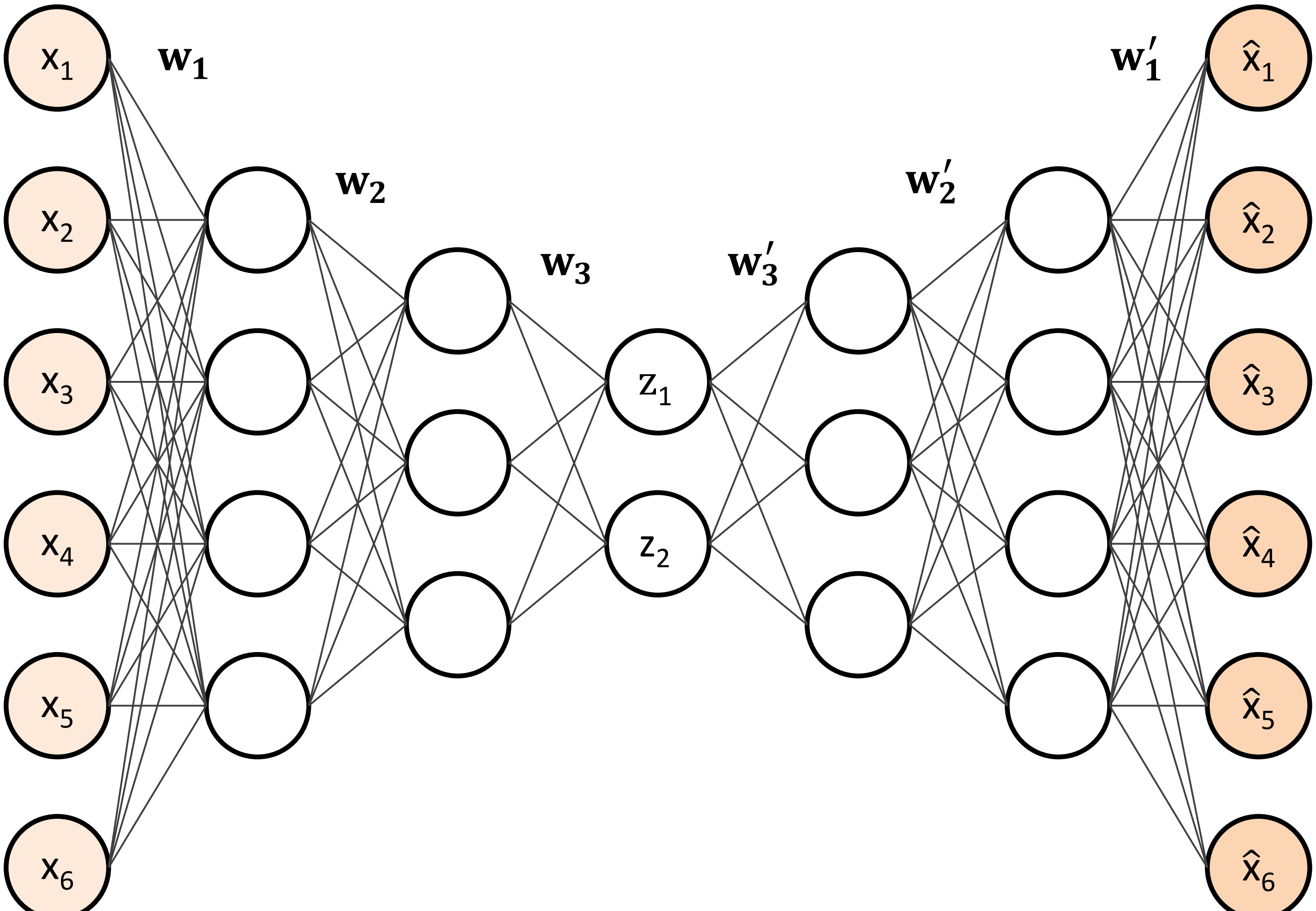
$$L\left(g_{\mathbf{w_2}}\left(h_{w_1}(\mathbf{x}^{(i)})\right), \mathbf{x}^{(i)}\right) = L_{MSE}\left\|\mathbf{x}^{(i)} - \hat{\mathbf{x}}^{(i)}\right\| + \lambda \sum_k \omega_k^2$$

$\mathbf{w_1}, \mathbf{w_2} = \{\omega_k\}$

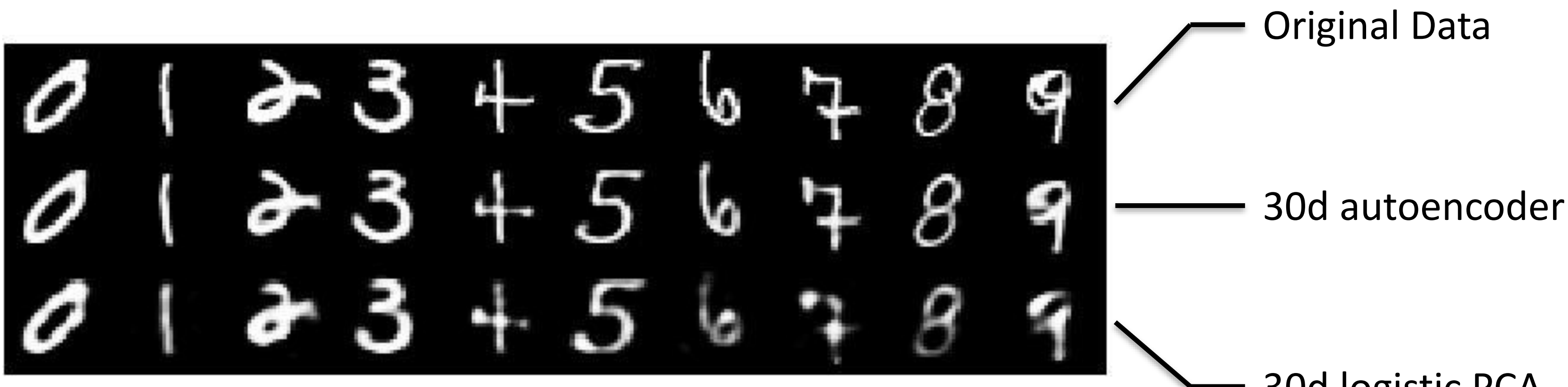
If encoder and decoder are symmetric, we can tie weights (use the same weights) on both sides to decrease number of parameters

$\mathbf{w_1} = \mathbf{w_2}^T$

Going deeper



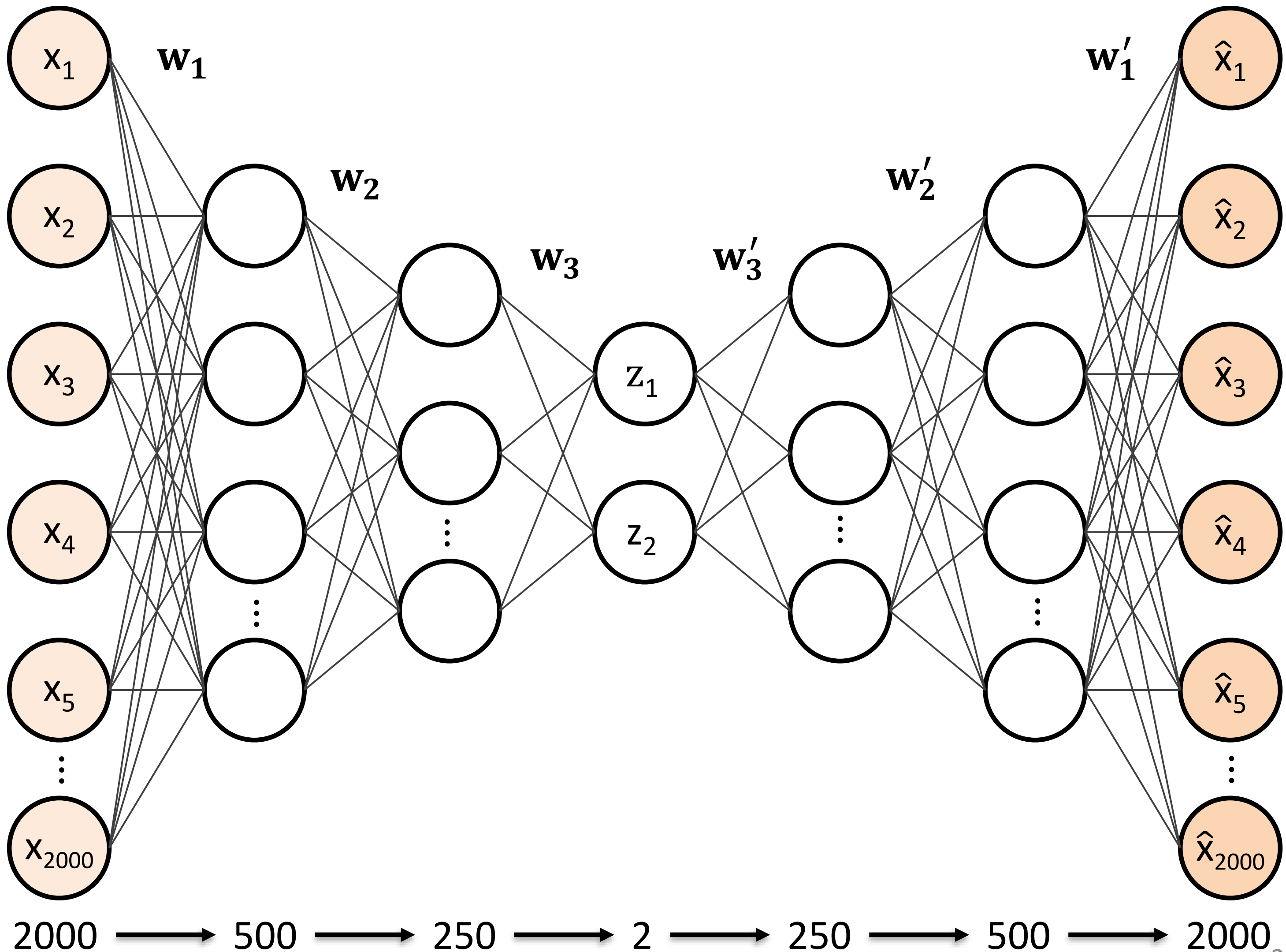
Comparison with PCA



Example from [Hinton]

Example: Document retrieval

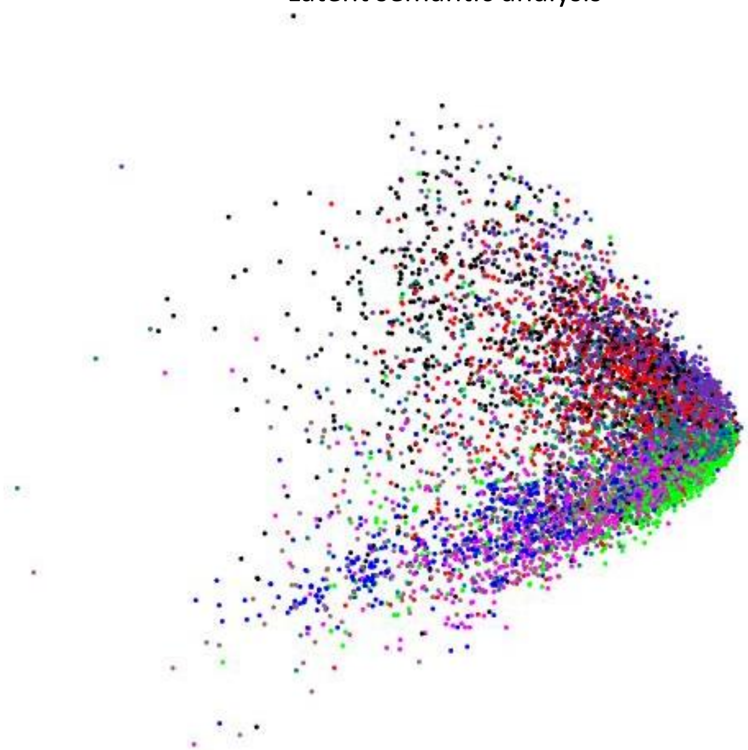
Frequencies of the 2000 commonest words



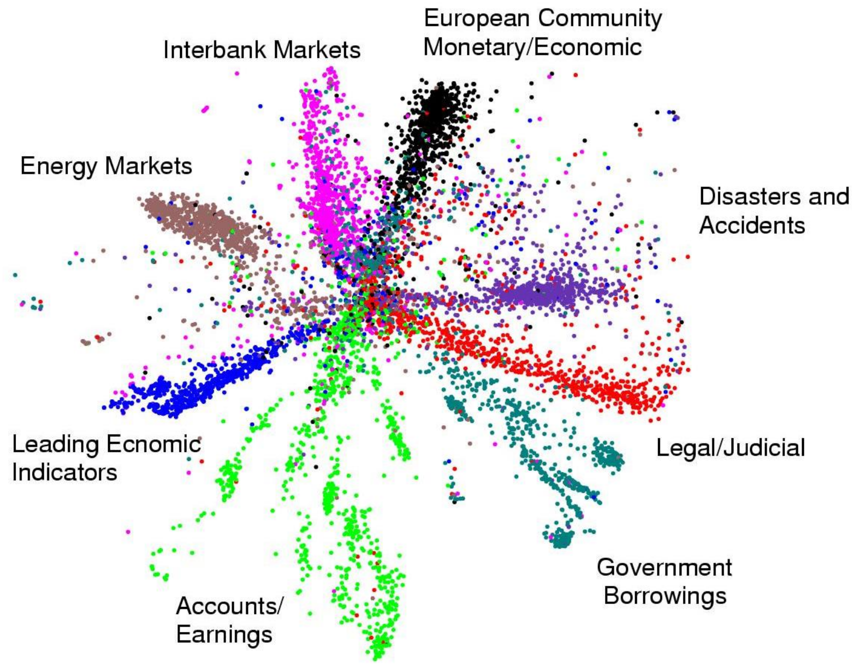
Example from [Hinton]

Example: Final 2D embeddings

Using LSA
Latent semantic analysis



Using Autoencoder



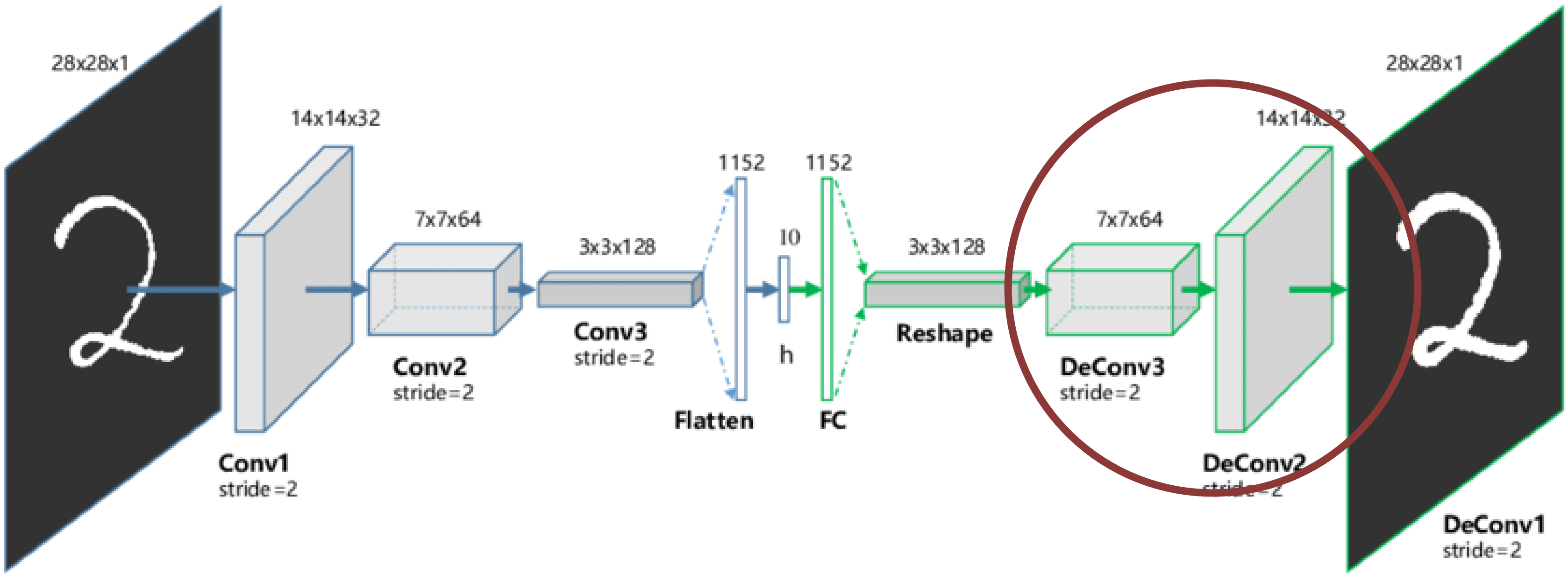
Different colours indicate different document clases – not used during training

Example from [Hinton]

Basic Autoencoder

```
class autoencoder(nn.Module):
    def __init__(self):
        super(autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(28 * 28, 128), nn.ReLU(True),
            nn.Linear(128, 64), nn.ReLU(True),
            nn.Linear(64, 12), nn.ReLU(True),
            nn.Linear(12, 3))

        self.decoder = nn.Sequential(
            nn.Linear(3, 12), nn.ReLU(True),
            nn.Linear(12, 64), nn.ReLU(True),
            nn.Linear(64, 128), nn.ReLU(True),
            nn.Linear(128, 28 * 28),
            nn.Tanh())

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

model = autoencoder().cuda()
criterion = nn.MSELoss()
```

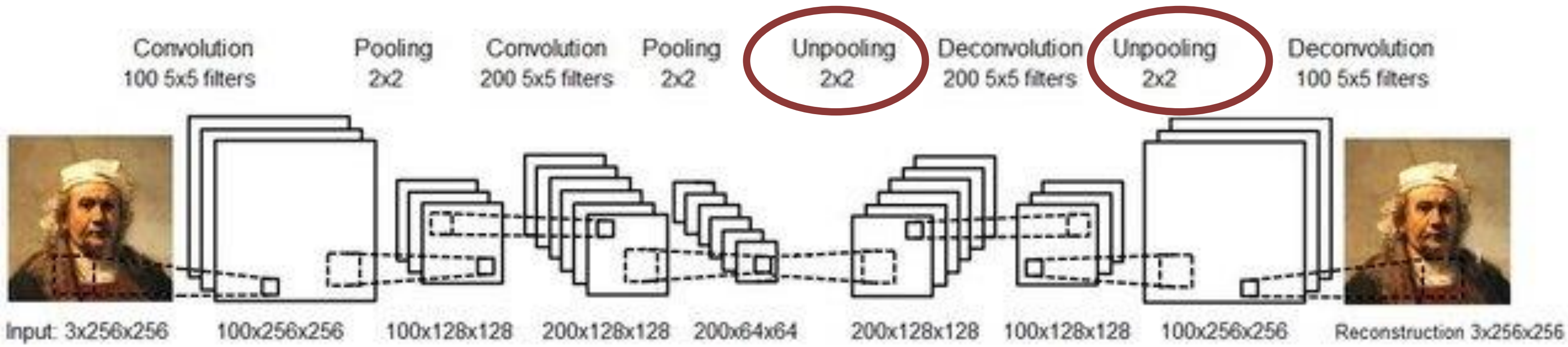
Convolutional Autoencoders



How to deconvolve?

Image credit: Guo et al, "Deep Clustering with Convolutional Autoencoders", 2017

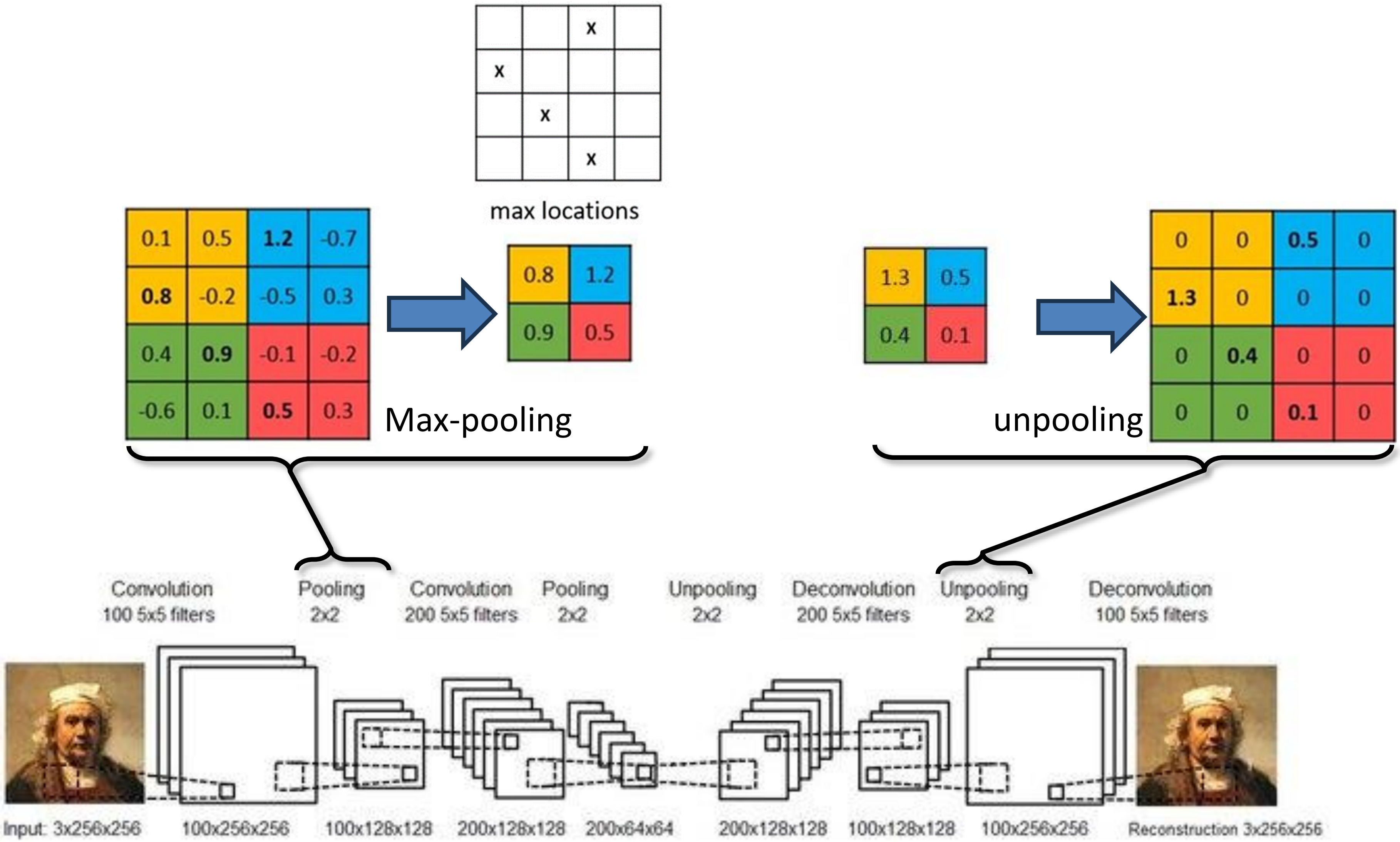
Convolutional Autoencoders



How to unpool?

Image credit: David et al, "[DeepPainter: Painter Classification Using Deep Convolutional Autoencoders](#)", 2016

Unpooling


max locations
0.1 0.5 1.2 -0.7
0.8 -0.2 -0.5 0.3
0.4 0.9 -0.1 -0.2
-0.6 0.1 0.5 0.3
0.8 1.2
0.9 0.5
Max-pooling
1.3 0.5
0.4 0.1
unpooling
0 0 0.5 0
1.3 0 0 0
0 0.4 0 0
0 0 0.1 0
Convolution
100 5x5 filters
Pooling
2x2
Convolution
200 5x5 filters
Pooling
2x2
Unpooling
2x2
Deconvolution
200 5x5 filters
Unpooling
2x2
Deconvolution
100 5x5 filters
Input: 3x256x256
100x256x256
100x128x128
200x128x128
200x64x64
200x128x128
100x128x128
100x256x256
Reconstruction 3x256x256

PyTorch

Unpooling

`MaxUnpool2d()` – see the documentation

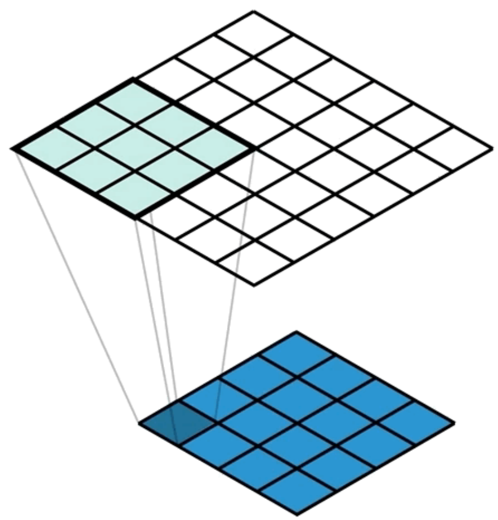
Ask for the indices

```
pool = nn.MaxPool2d(2, stride = 2, return_indices = True)
unpool = nn.MaxUnpool2d(2, stride = 2)
# …
output, indices = pool(X)
unpool(output, indices)
```

Save the indices of the pooling operation and use to unpool

Transposed Convolutions (Deconvolutions)

A **transposed convolution** (also called **fractionally strided convolution** or **deconvolution**) is the reverse process of convolution.



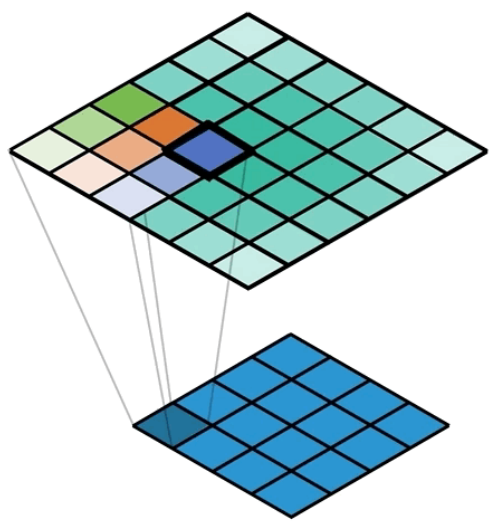
w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

The easiest way of thinking about it is to take each value in your input and distribute it (using the corresponding weights of a kernel) to a local region in the output

Image credit: T. Lane, 2018, https://medium.com/apache-mxnet/transposed-convolutions-explained-with-ms-excel-52d13030c7e8

Transposed Convolutions (Deconvolutions)

A **transposed convolution** (also called **fractionally strided convolution** or **deconvolution**) is the reverse process of convolution.



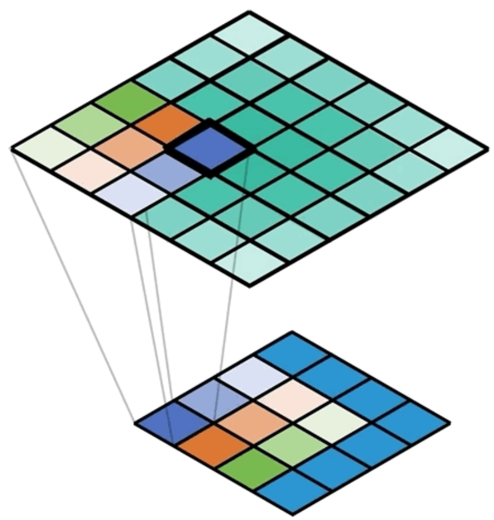
w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

The easiest way of thinking about it is to take each value in your input and distribute it (using the corresponding weights of a kernel) to a local region in the output

Image credit: T. Lane, 2018, https://medium.com/apache-mxnet/transposed-convolutions-explained-with-ms-excel-52d13030c7e8

Transposed Convolutions (Deconvolutions)

A **transposed convolution** (also called **fractionally strided convolution** or **deconvolution**) is the reverse process of convolution.



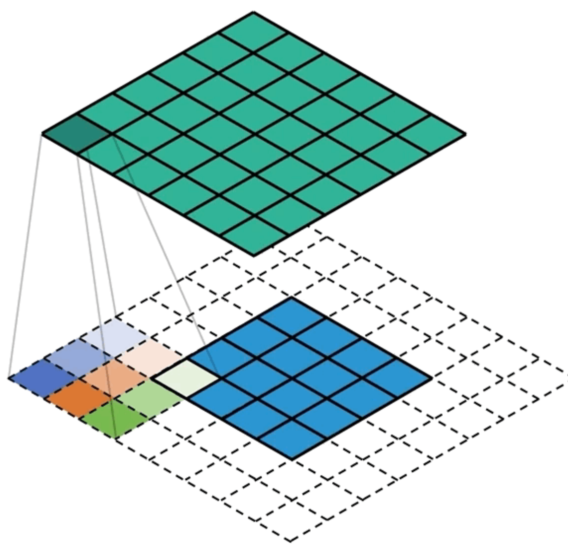
w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

Note how “distributing” the input values ends up being equivalent to applying a normal convolution with the transposed kernel

Image credit: T. Lane, 2018, https://medium.com/apache-mxnet/transposed-convolutions-explained-with-ms-excel-52d13030c7e8

Transponsed Convolutions (Deconvolutions)

A **transposed convolution** (also called **fractionally strided convolution** or **deconvolution**) is the reverse process of convolution.



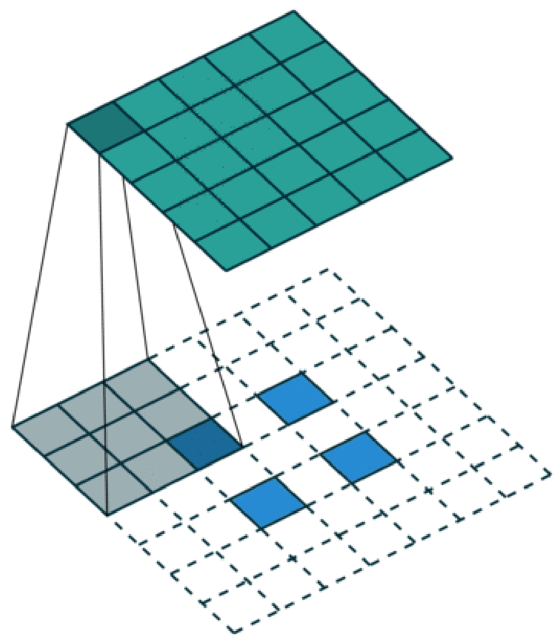
w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

Applying a forward convolution with a transposed kernel, actually results in a much smaller output. What we are really doing here is equivalent to this transposed convolution but with full padding

Image credit: T. Lane, 2018, https://medium.com/apache-mxnet/transposed-convolutions-explained-with-ms-excel-52d13030c7e8

Fractionally strided convolutions

The transpose of a convolution with stride $s > 1$ involves an equivalent convolution with $s < 1$. This is why transposed convolutions are sometimes called *fractionally strided convolutions*.



In the case of transposed convolutions invert the input and output sizes in the formula we defined earlier, and you get: $o = (n - 1)s - 2p + f$

Image credit: V. Dumoulin, F. Visin, “guide to convolution arithmetic for deep learning”, 2018

Transponsed Convolutions (Deconvolutions)

Convolution: $$n_{W/H}^{[l]} = \frac{n_{W/H}^{[l-1]} + 2p - f}{s} + 1$$

Transposed Convolution: $$n_{W/H}^{[l]} = (n_{W/H}^{[l-1]} - 1)s - 2p + f$$

Filter size: f
Padding: p
Stride: s

PyTorch

Transposed Convolutions

`ConvTranspose2d()` – see the documentation

batches *channels* *height* *width*

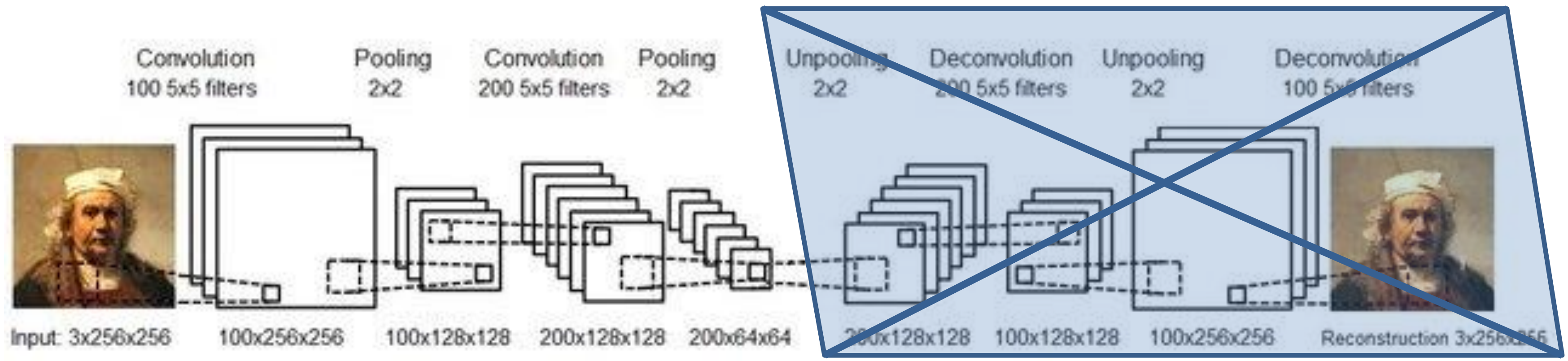
```
input = torch.randn(1, 16, 12, 12)

downsample = nn.Conv2d(16, 16, 3, stride=2, padding=1)
upsample = nn.ConvTranspose2d(16, 16, 3, stride=2, padding=1)

hidden = downsample(input)

output = upsample(h)
```

```
torch.Size([1, 16, 6, 6])
```

```
torch.Size([1, 16, 12, 12])
```

Autoencoders for weight initialisation

Makes sense if labeled training data is scarce, but unlabeled data is easy to get



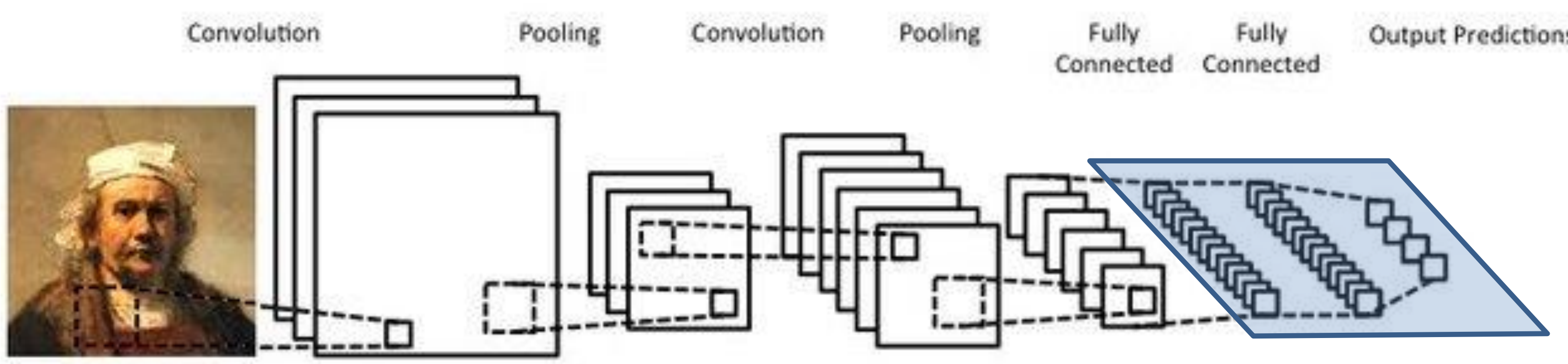
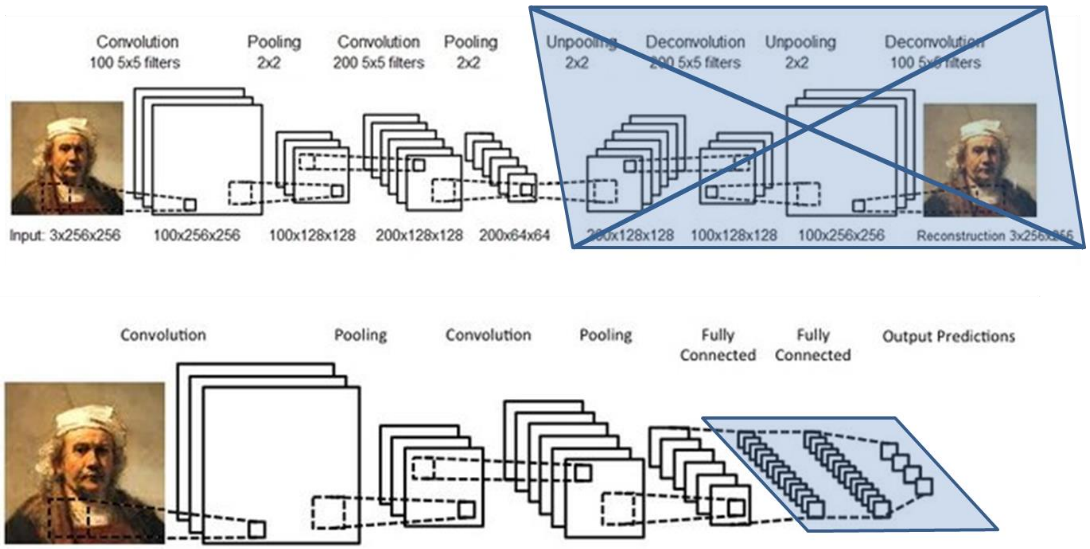


Image credit: David et al, "[DeepPainter: Painter Classification Using Deep Convolutional Autoencoders]", 2016

Autoencoders for weight initialisation

Makes sense if labeled training data is scarce, but unlabeled data is easy to get



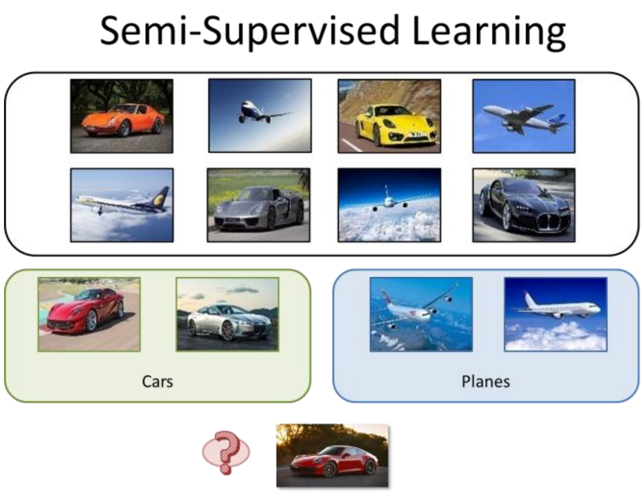


Image credit: David et al, "DeepPainter: Painter Classification Using Deep Convolutional Autoencoders", 2016

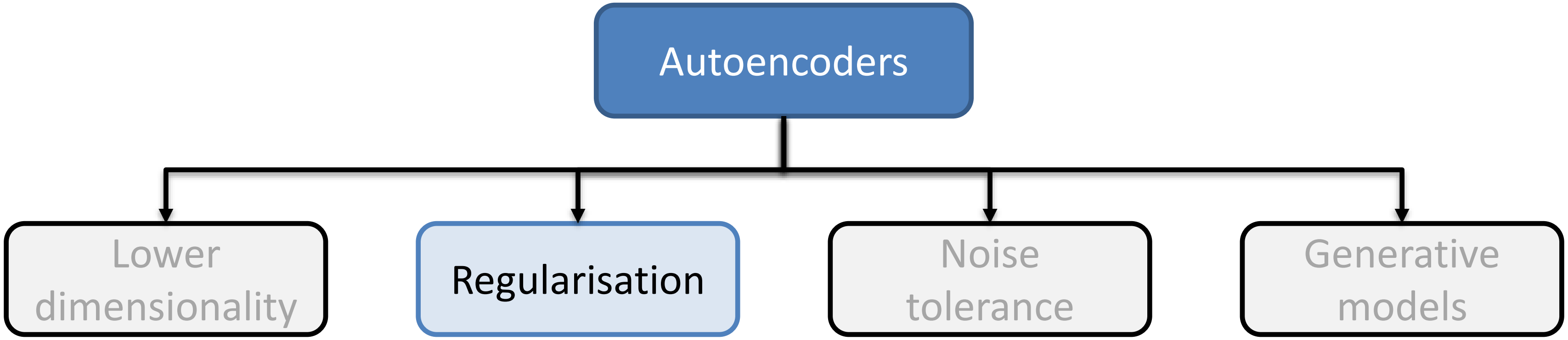
Convolutional AE

```
class autoencoder(nn.Module):                                  #Input: b, 1, 28, 28
    def __init__(self):
        super(autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 16, 3, stride=3, padding=1),  # b, 16, 10, 10
            nn.ReLU(True), nn.MaxPool2d(2, stride=2),  # b, 16, 5, 5
            nn.Conv2d(16, 8, 3, stride=2, padding=1),  # b, 8, 3, 3
            nn.ReLU(True), nn.MaxPool2d(2, stride=1)   # b, 8, 2, 2
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(8, 16, 3, stride=2),            #b,16,5,5
            nn.ReLU(True),
            nn.ConvTranspose2d(16, 8, 5, stride=3, padding=1), #b,8,15,15
            nn.ReLU(True),
            nn.ConvTranspose2d(8, 1, 2, stride=2, padding=1),  #b,1,28,28
            nn.Tanh()
        )

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

model = autoencoder().cuda()
criterion = nn.MSELoss()
```

Note: This is a non symmetric example

Convolution: $n_{W/H}^{[l]} = \frac{n_{W/H}^{[l-1]} + 2p - f}{s} + 1$

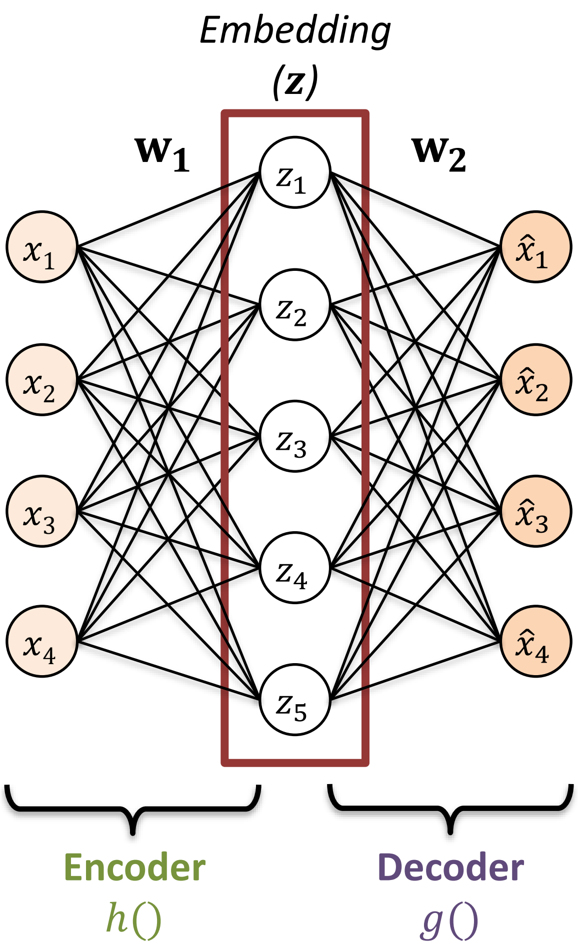
Transposed Convolution: $n_{W/H}^{[l]} = (n_{W/H}^{[l-1]} - 1)s - 2p + f$



Regularisation – learning sparse / smooth embeddings

REGULARISATION

Regularise by enforcing sparsity



Regularisation on weights

$$L\left(g_{\mathbf{w_2}}\left(h_{\mathbf{w_1}}\left(\mathbf{x}^{(i)}\right)\right),\mathbf{x}^{(i)}\right) = L_{MSE}\left\|\mathbf{x}^{(i)} - \hat{\mathbf{x}}^{(i)}\right\| + \lambda \sum_k \omega_k^2$$

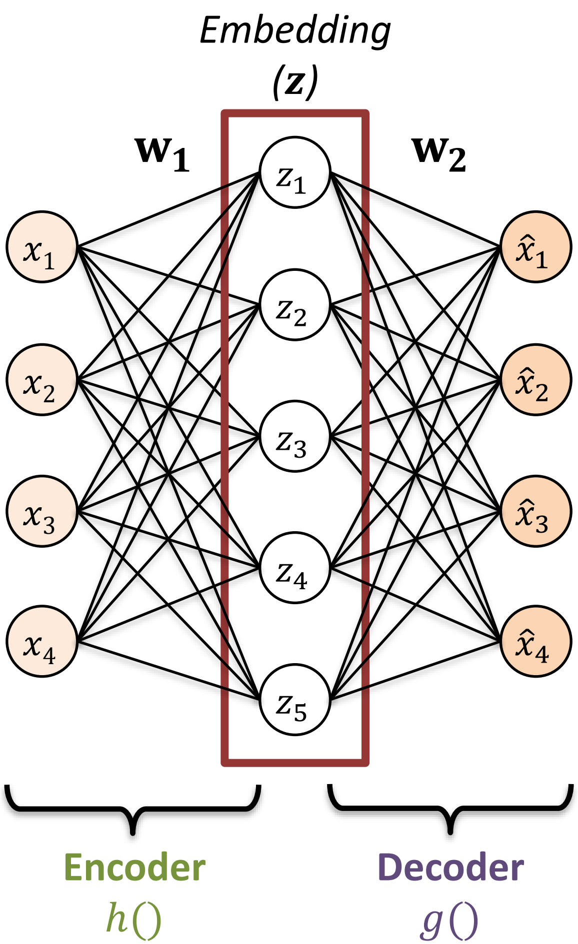
$\mathbf{w_1},\mathbf{w_2} = \{\omega_k\}$

Sparsity ➔ Regularisation on embedding

$$L\left(g_{\mathbf{w_2}}\left(h_{\mathbf{w_1}}\left(\mathbf{x}^{(i)}\right)\right),\mathbf{x}^{(i)}\right) = \underbrace{L_{MSE}\left\|\mathbf{x}^{(i)} - \hat{\mathbf{x}}^{(i)}\right\|}_{\textit{Reconstruction error term}} + \underbrace{\lambda \sum_k \left|z_k^{(i)}\right|}_{\textit{L1 sparsity term}}$$

Note that in this case we can use overcomplete models

Regularise by enforcing smoothness



Intuition: Points close to each other in input space should maintain that property in the embedding space

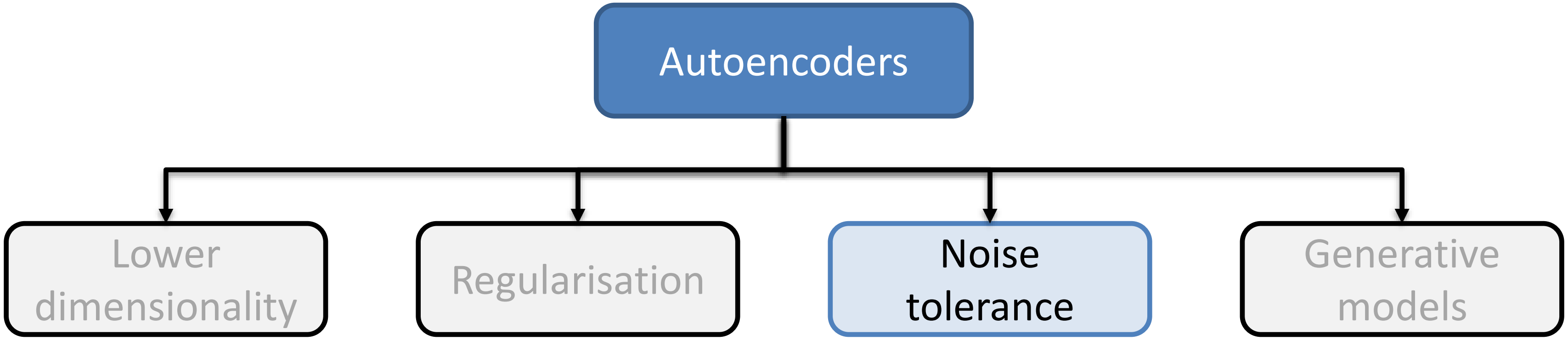
The **"contractive" autoencoder** encourages the derivatives of the embedding with respect to the input to be small, meaning that the representation of the input should be robust to small changes in the input

$$\|J_h(\mathbf{x})\|_F^2 = \sum_{j=1}^{d} \sum_{i=1}^{c} \left(\frac{\partial h_i}{\partial x_j}(\mathbf{x}) \right)^2$$

$$\Omega_{CAE}(\mathbf{w}) = \sum_{x^i} \left\| J_h(\mathbf{x}^{(i)}) \right\|_F^2$$

Smooth means small derivatives (Jacobian)

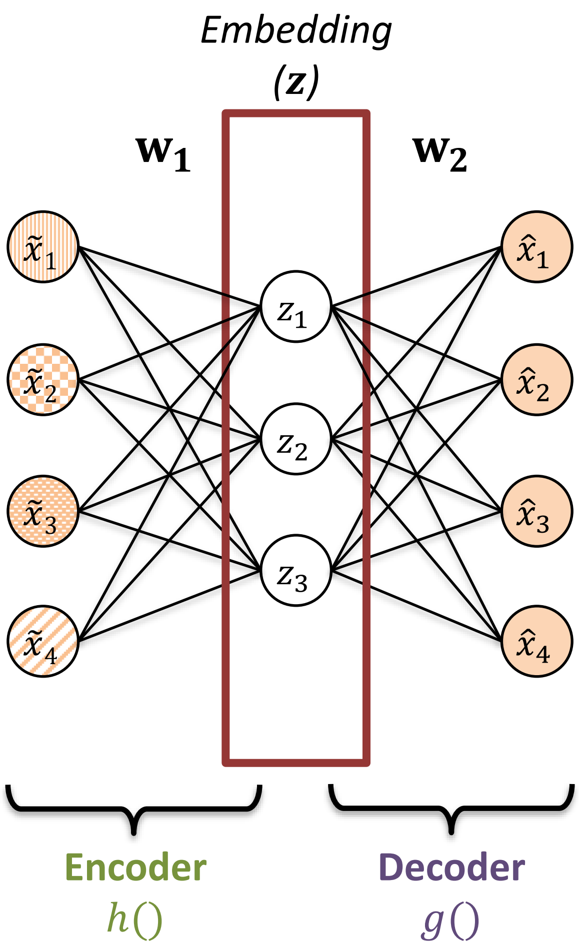
$$L\left(g_{\mathbf{w}_2}\left(h_{\mathbf{w}_1}(\mathbf{x}^{(i)}) \right), \mathbf{x}^{(i)} \right) = \underbrace{L_{MSE} \left\| \mathbf{x}^{(i)} - \hat{\mathbf{x}}^{(i)} \right\|}_{\text{Reconstruction error term}} + \underbrace{\lambda \Omega_{CAE}(\mathbf{w_1})}_{\text{Contractive term}}$$



Introduce a small change in the input and learn to undo it

NOISE TOLERANCE

Denoising Autoencoders



Intuition: learn to generate robust features from inputs by reconstructing partially destroyed samples

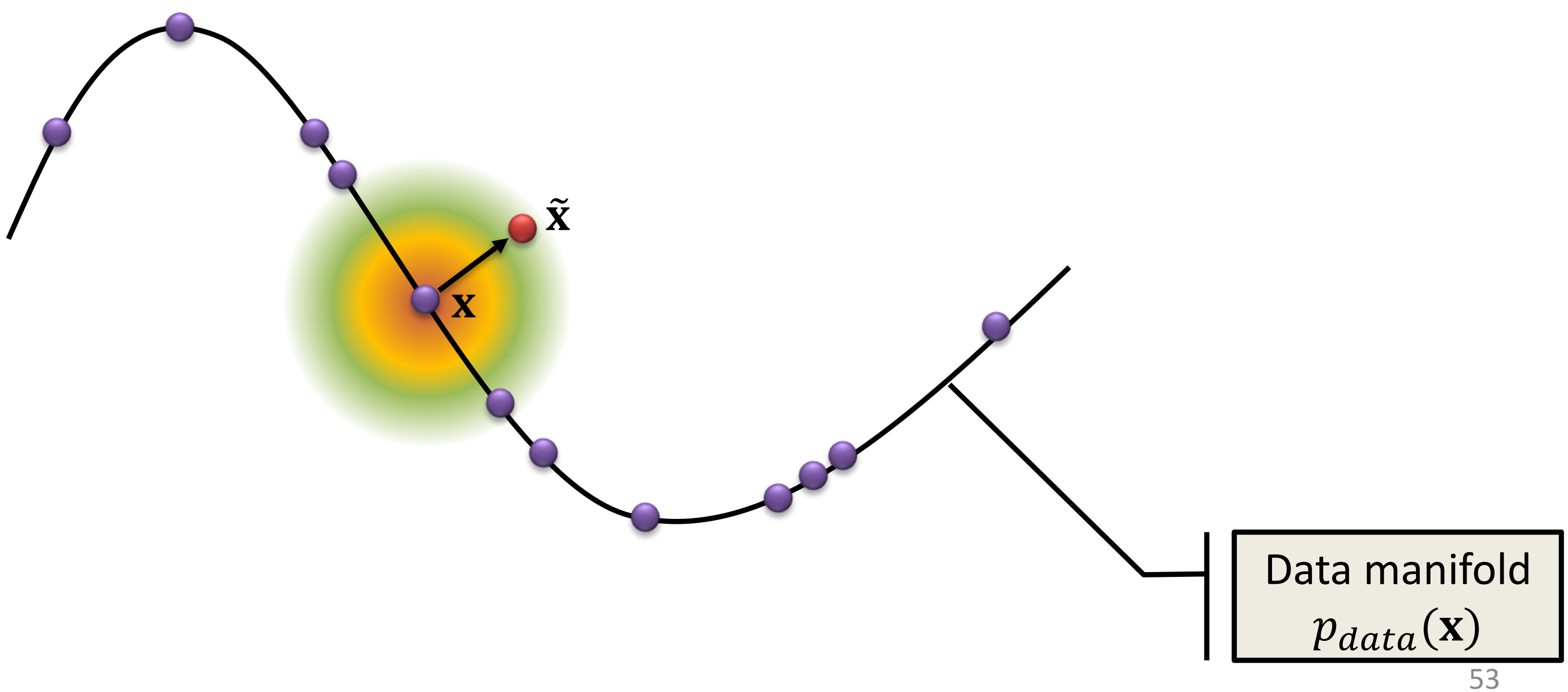
For every input x, we apply a corrupting function $C(\cdot)$ to create noisy version: $\tilde{\mathbf{x}} = C(\mathbf{x})$

$$L\left(g_{\mathbf{w_2}}\left(h_{w_1}\left(\tilde{\mathbf{x}}^{(i)}\right)\right), \mathbf{x}^{(i)}\right) = L_{MSE}\left\|\mathbf{x}^{(i)} - \hat{\mathbf{x}}^{(i)}\right\|$$

$$\boxed{\tilde{\mathbf{x}} = C(\mathbf{x})}$$

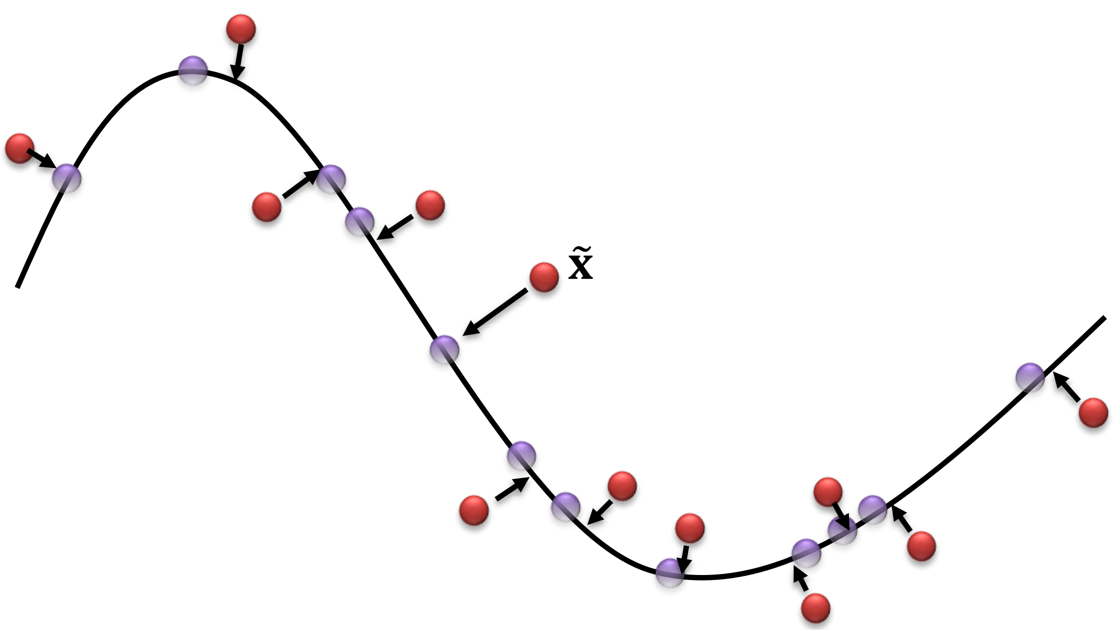
Learning the manifold

The corrupting function $C(\cdot)$ can corrupt in any direction. autoencoder must learn the "location" of data manifold and its distribution $p_{data}(\mathbf{x})$



Learning the manifold

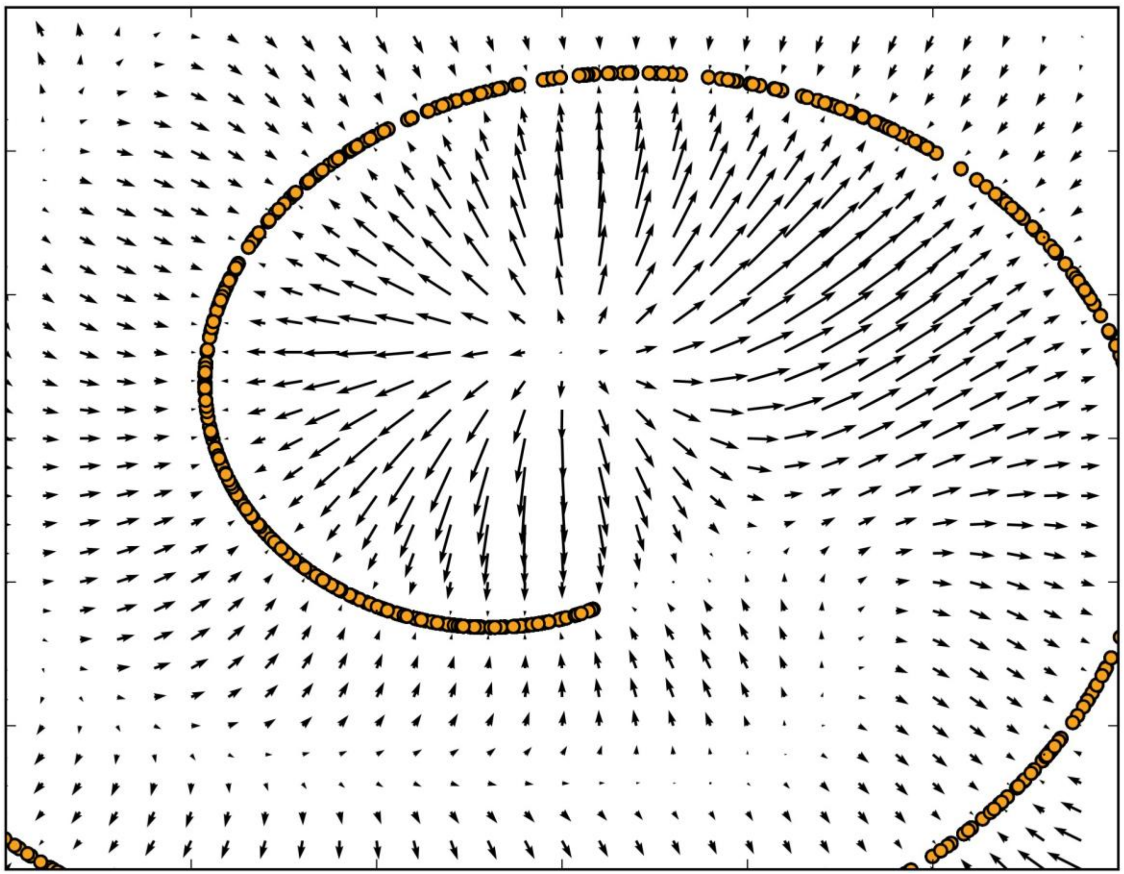
The corrupting function $C(\cdot)$ can corrupt in any direction. autoencoder must learn the "location" of data manifold and its distribution $p_{data}(\mathbf{x})$



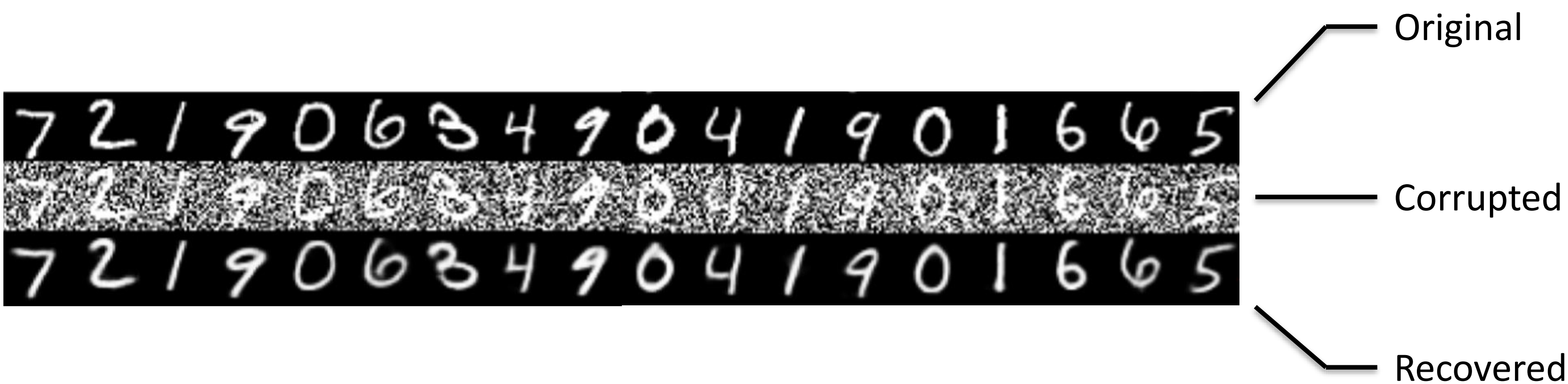
Any corrupted point is locally mapped back to the manifold

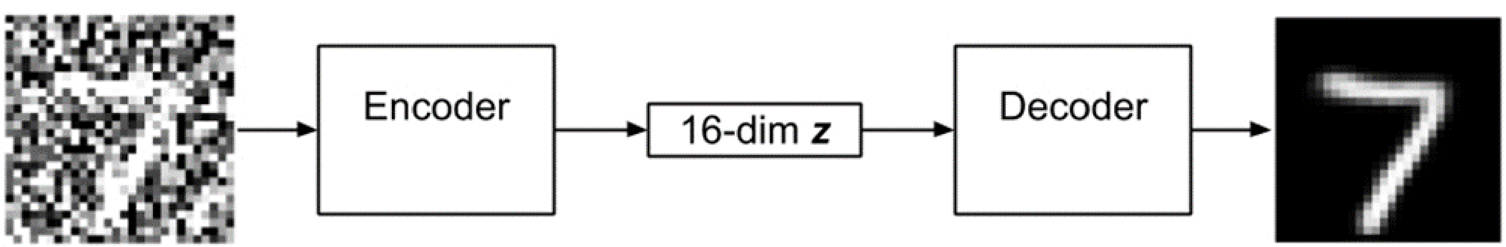
Learning the manifold

2D vector field around a 1D curved manifold where the data concentrates



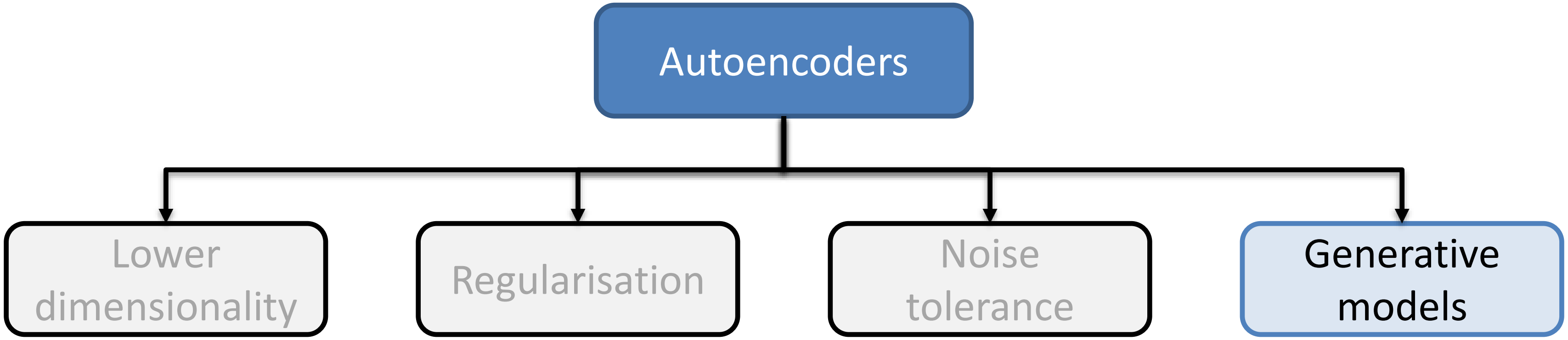
Denoising Example





Denoising examples

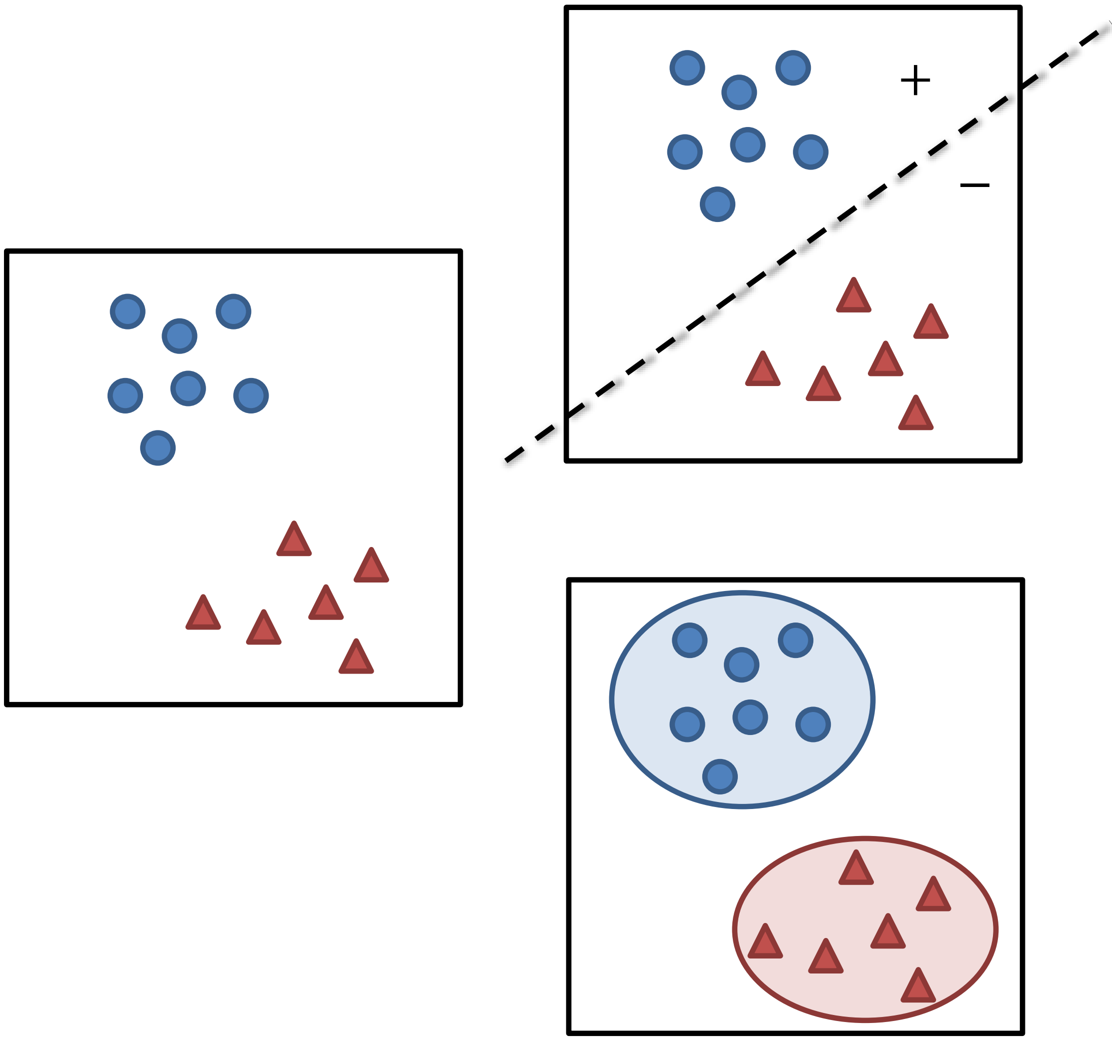
Quan, "[Iterative Application of Autoencoders for Video Inpainting and Fingerprint Denoising](#)", 2019



Generative models

VARIATIONAL AUTOENCODERS

Discriminative and Generative Models

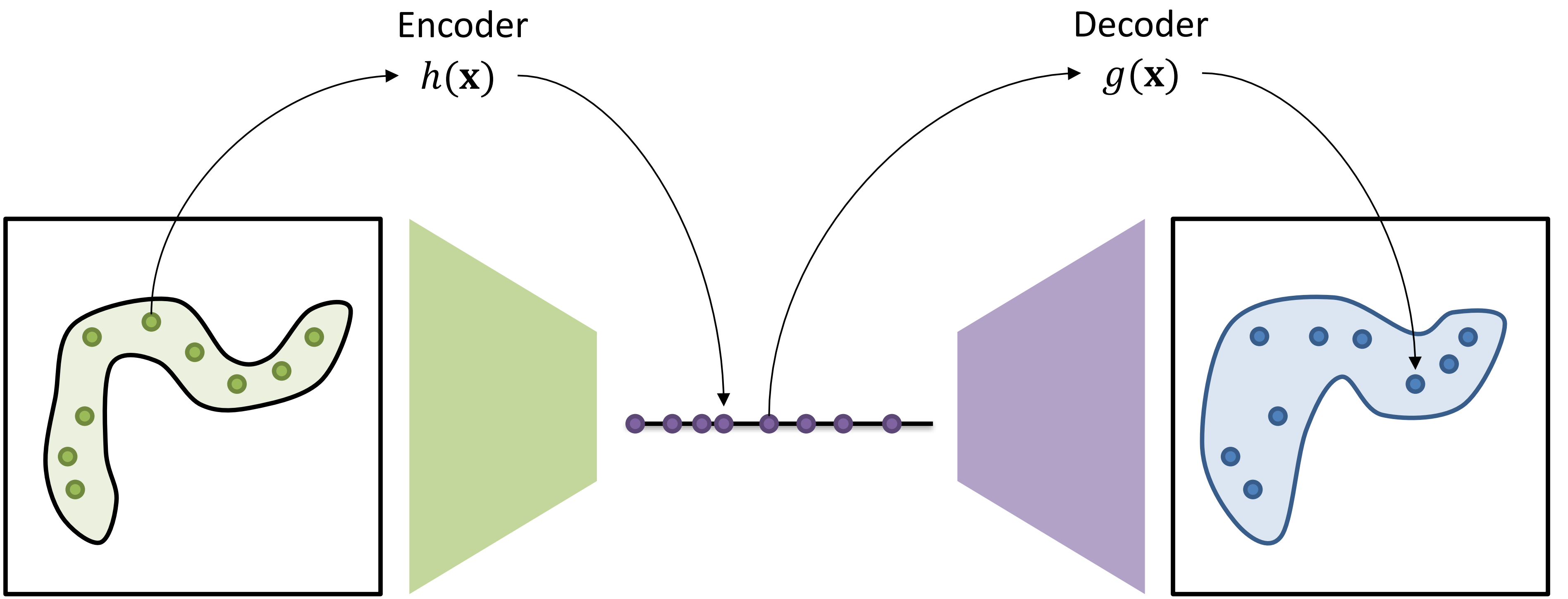


Discriminative models describe the decision surface; we can tell if a new point is on one side or another

Generative models describe the data distribution. We can ask which distribution a new point is most probable to come from

And we can use the learnt distributions to **generate new data**!

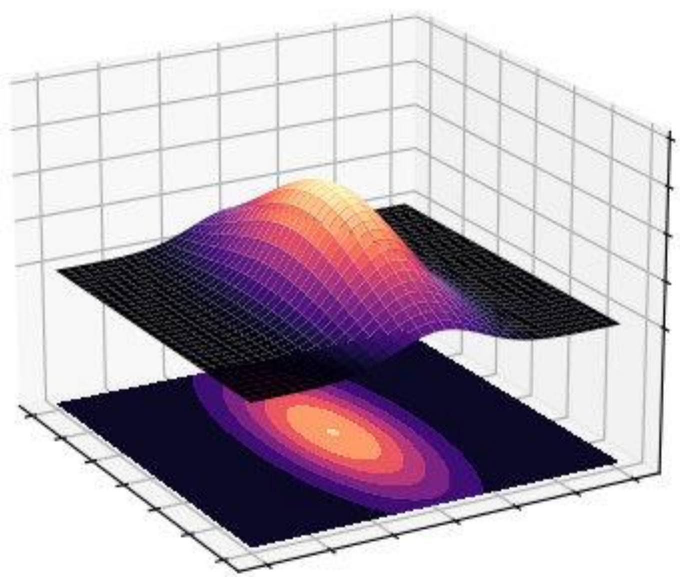
What do autoencoders model?



Let's rethink the underlying idea of autoencoders. Instead of seeing the encoder and decoder as functions that map points between spaces, we can see them as probability distributions

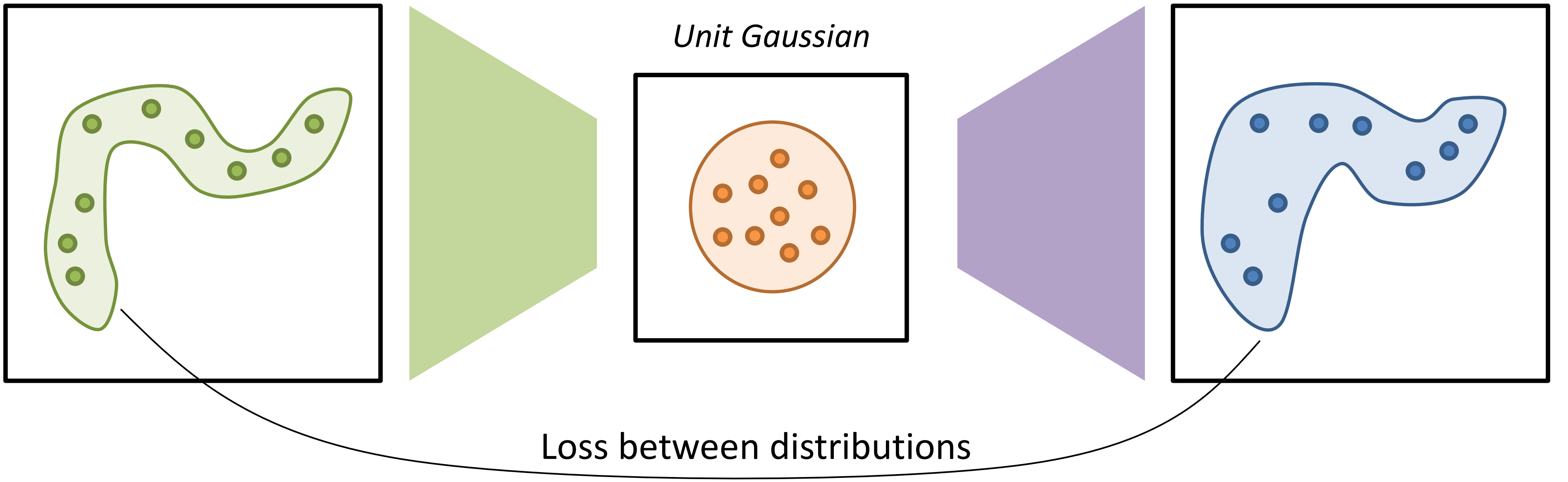
Generating new data

Which is the probability of a pixel to be ‘on’ on an image representing number ‘1’



Generating new data

Since we do not know the data distribution in the original space, we cannot directly generate new data



BUT, if we manage to map whatever distribution we have into a known distribution we know how to sample from, then we could use our generator to generate new data

Unit Gaussian Embedding of MNIST data

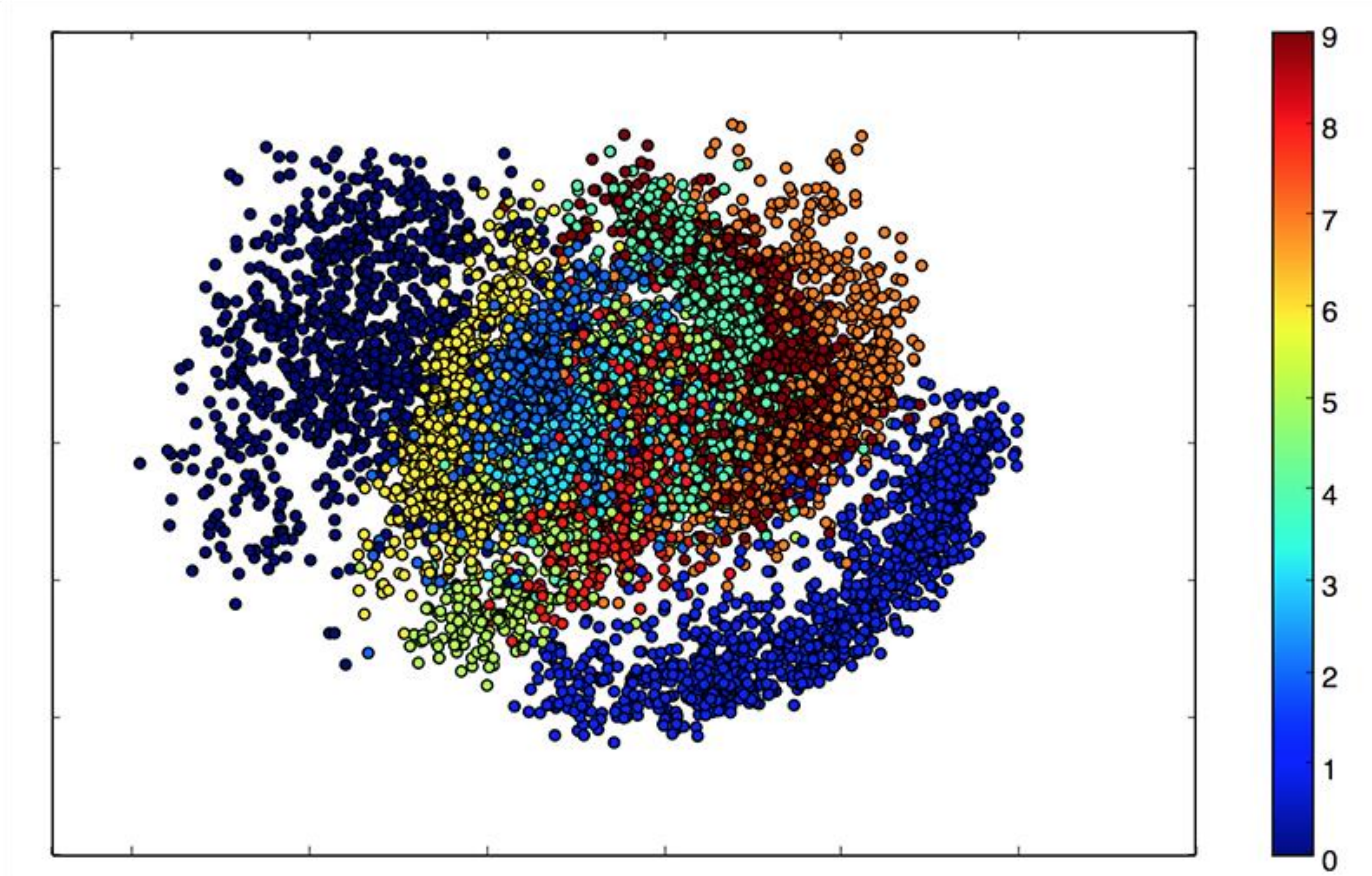
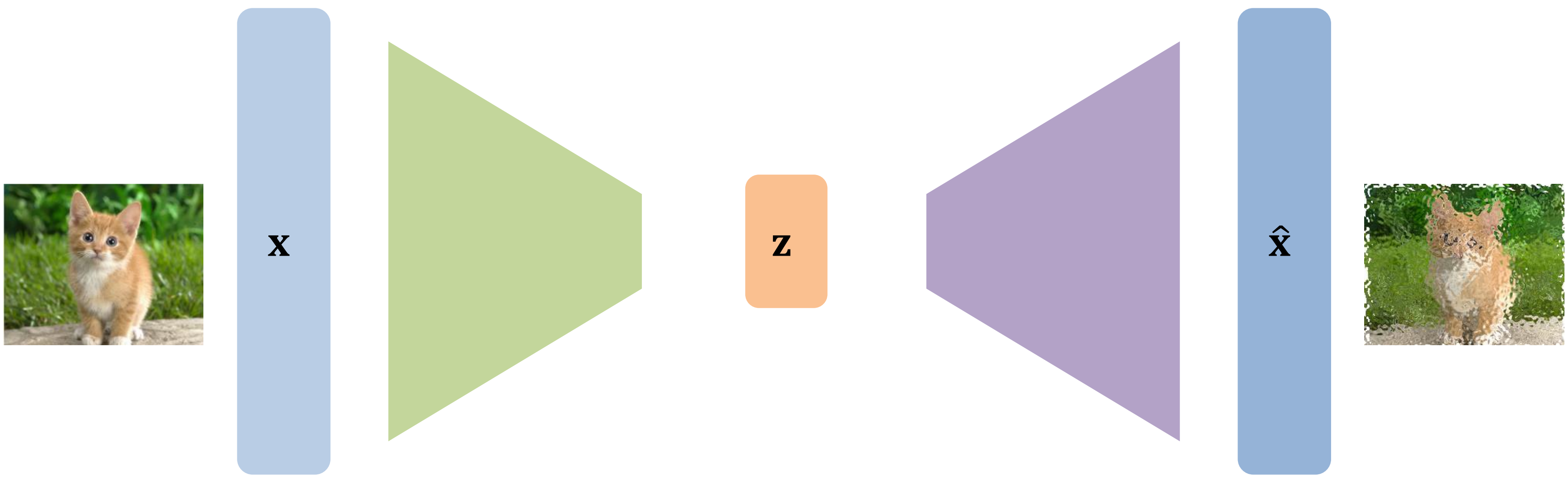


Image reproduced from https://blog.keras.io/building-autoencoders-in-keras.html

Traditional Autoencoder

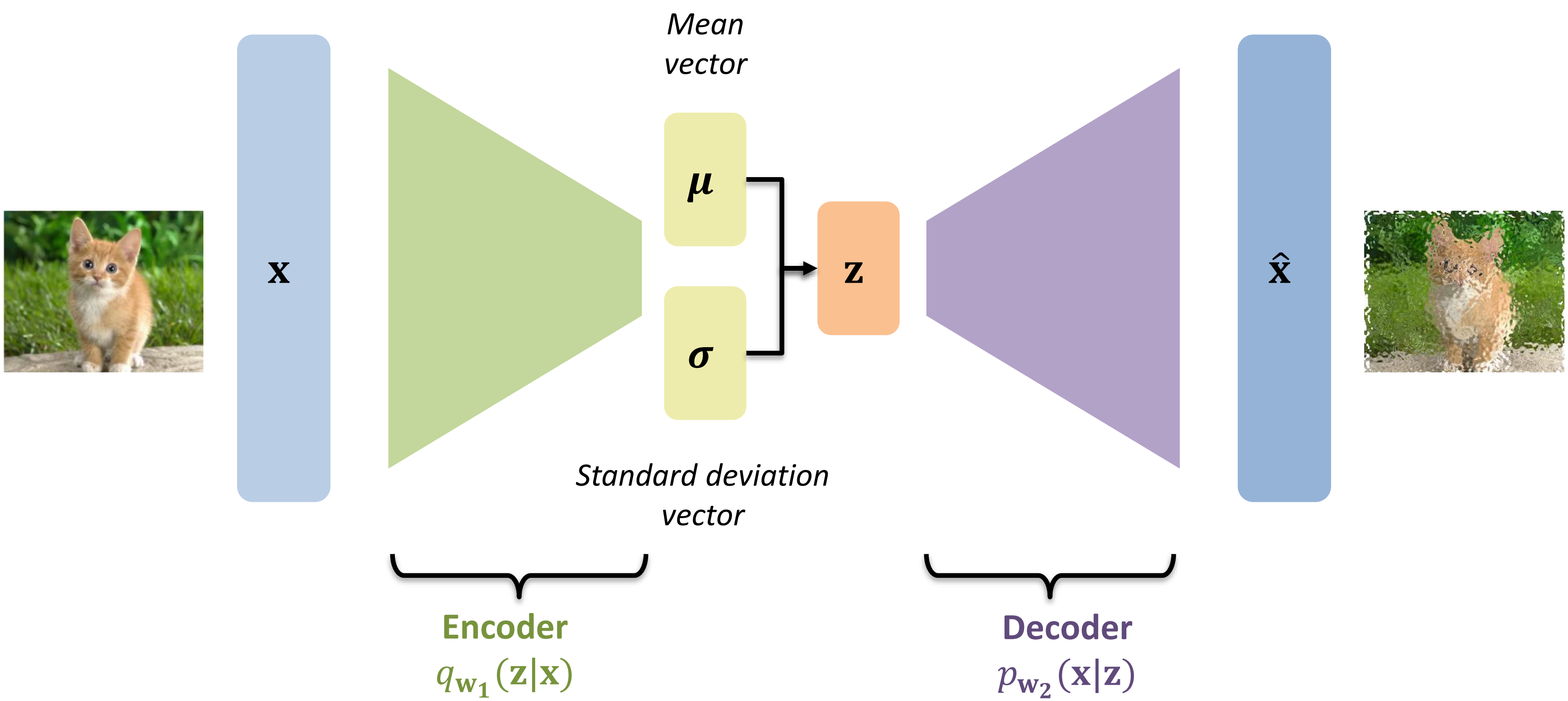


Generative Autoencoders

- Generative models learn a **distribution** in order to be able to **draw new samples** from, different from those observed
- AEs can generally reconstruct encoded data, but are **not** necessarily **able to build meaningful outputs** from arbitrary **encodings**
- Variational and adversarial AEs learn a model of the data from which new instances can be generated

Variational Autoencoders (VAE)

Input → encode to statistics vectors → sample a latent vector → decode for reconstruction

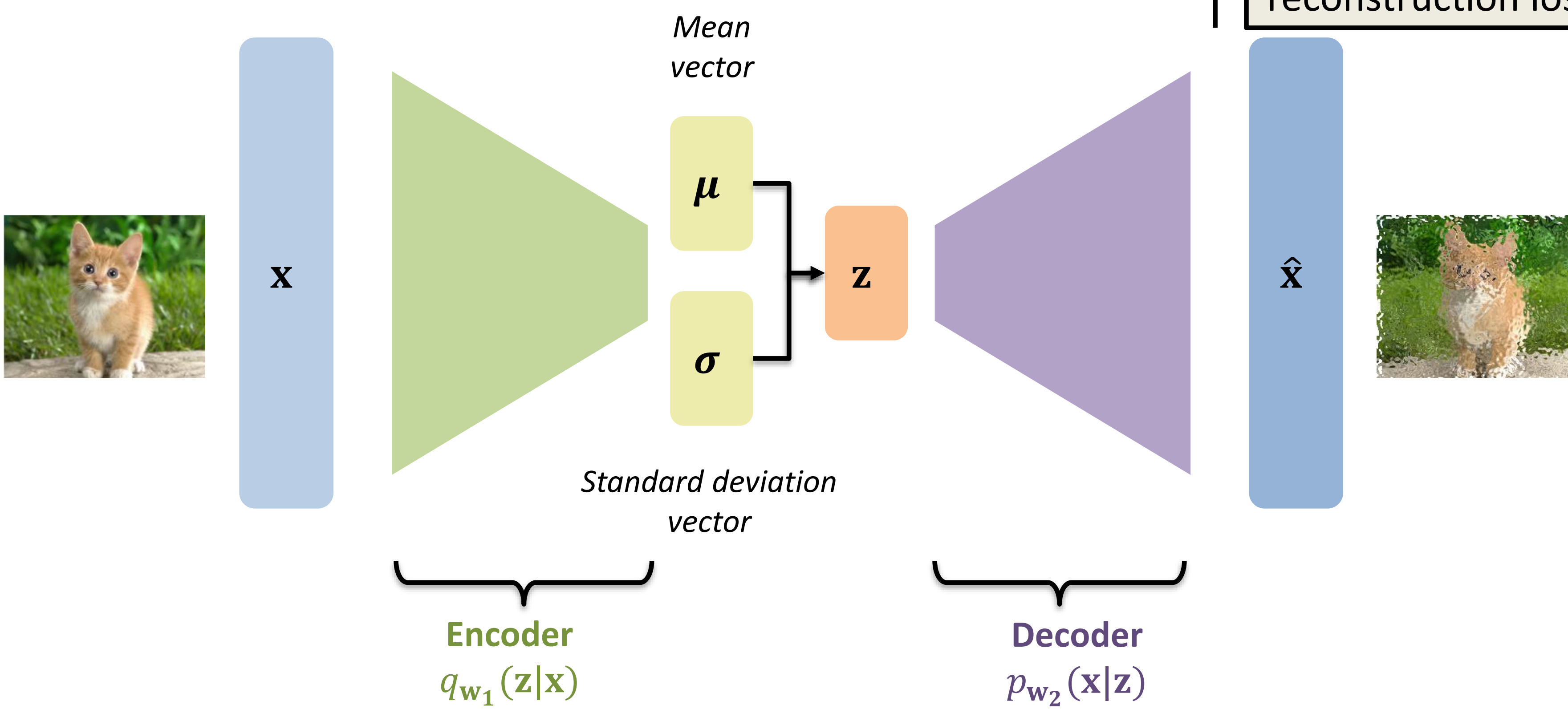


$$L(w_1, w_2, x) = (reconstruction\ loss) + (regularization\ term)$$

Variational Autoencoders (VAE)

$$L_{rec}(x) = -\mathbb{E}_{z \sim q_{w_1}(z|x)}[\log p_{w_2}(x|z)]$$

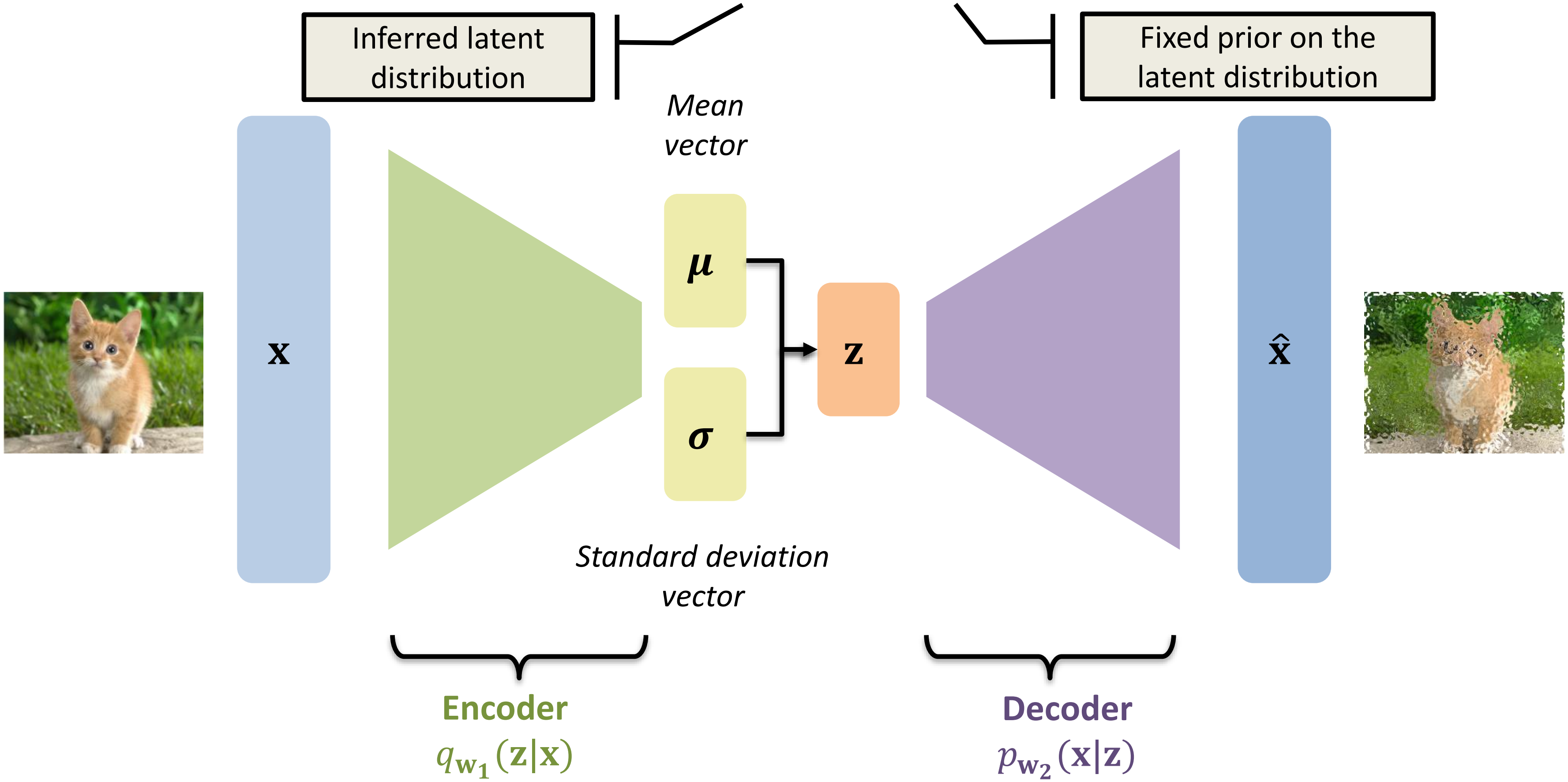
This is actually a reconstruction loss



$$L(w_1, w_2, x) = (reconstruction\ loss) + (regularization\ term)$$

Variational Autoencoders (VAE)

$$L_{div}(x) = \mathbb{KL}(q_{w_1}(z|x) \parallel p(z))$$



$$L(w_1, w_2, x) = (reconstruction\ loss) + (regularization\ term)$$

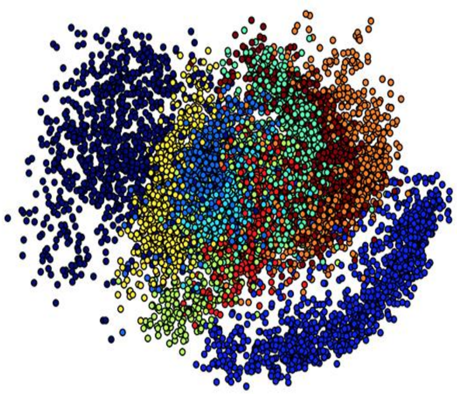
Prior on the latent distribution

Inferred latent distribution

Fixed prior on the latent distribution

$$L_{div}(x) = \mathbb{KL}(q_{w_1}(z|x) \parallel p(z))$$

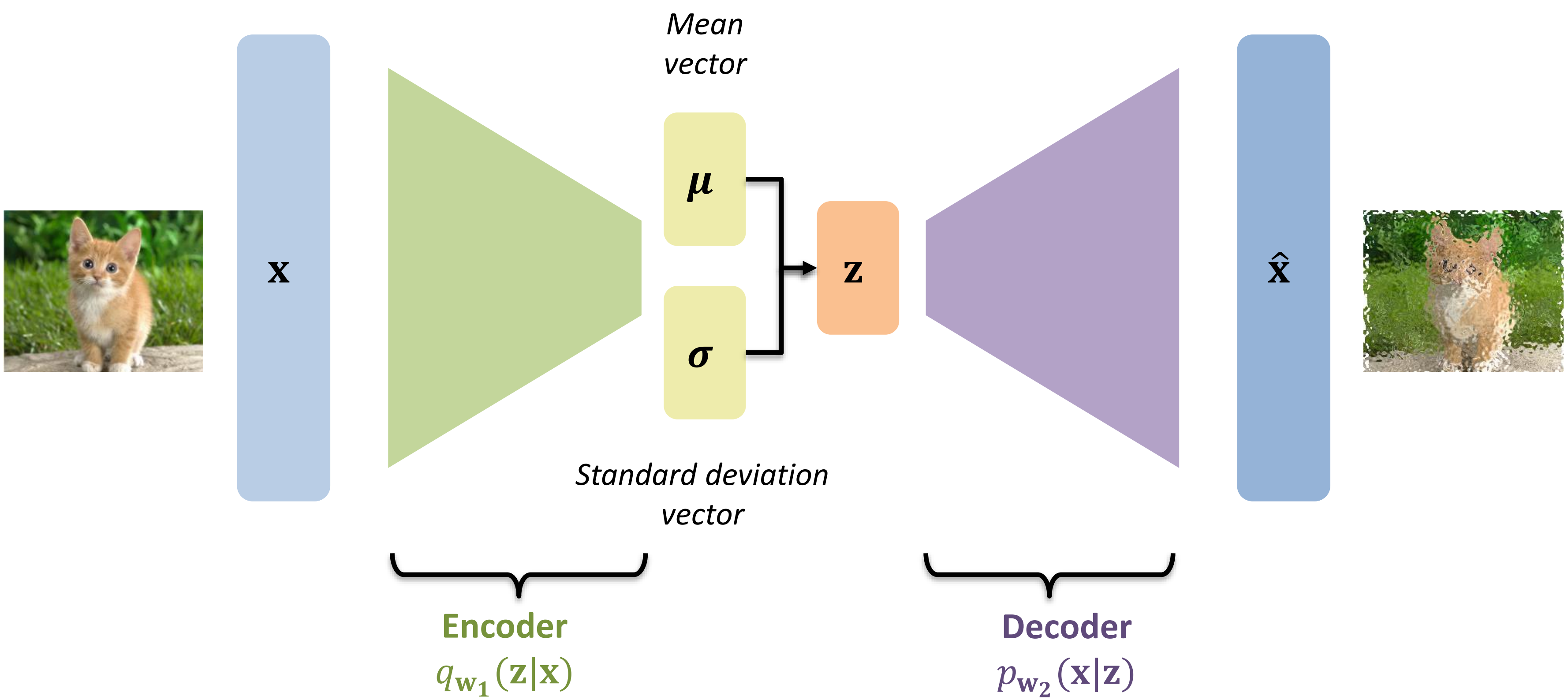
$$L_{div}(x) = -\frac{1}{2}\sum_{j=0}^{k-1}\left(\sigma_j + \mu_j^2 - 1 - \log \sigma_j\right)$$



Common choice of prior – Normal Gaussian:
$p(z) = N(\mu = 0, \sigma^2 = 1)$

Backpropagating

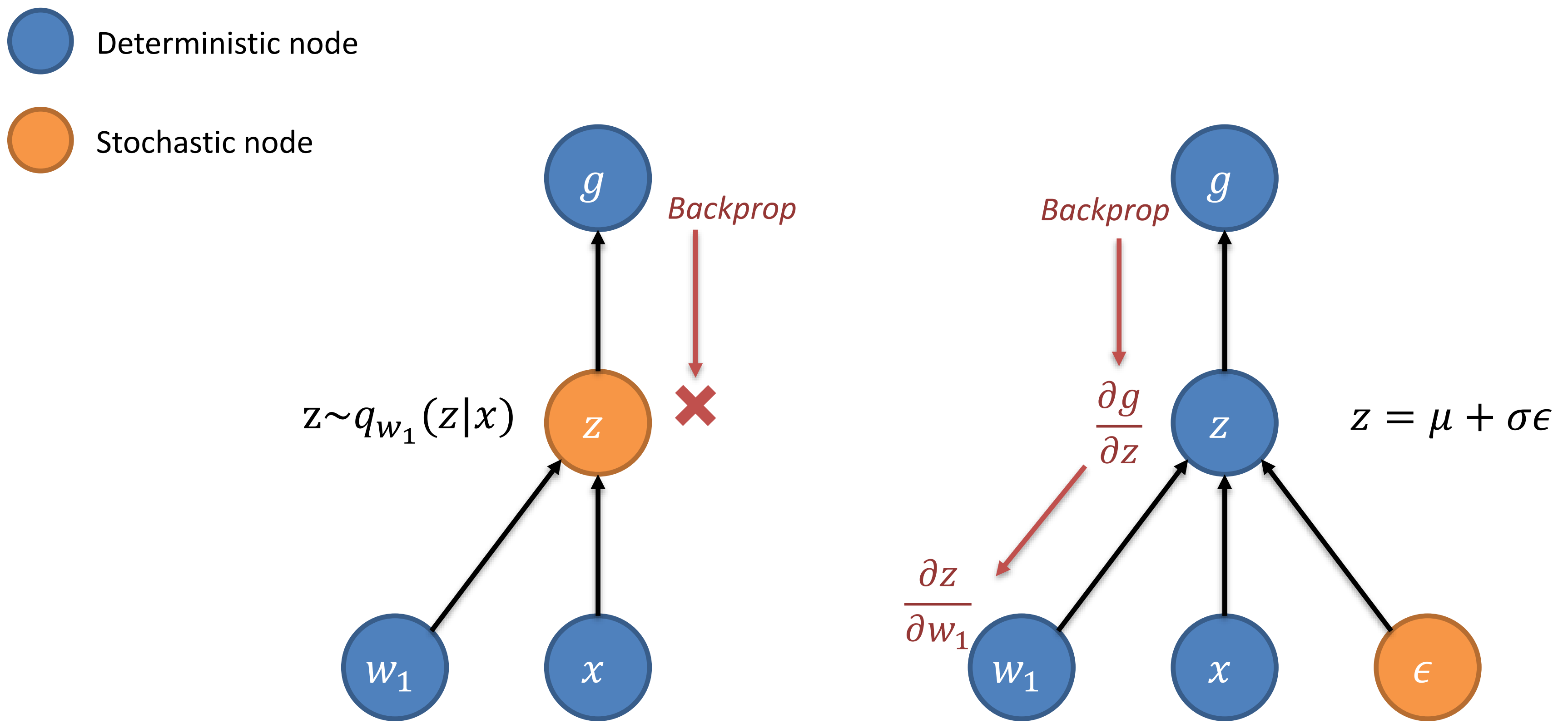
How can we backpropagate through the sampling process?



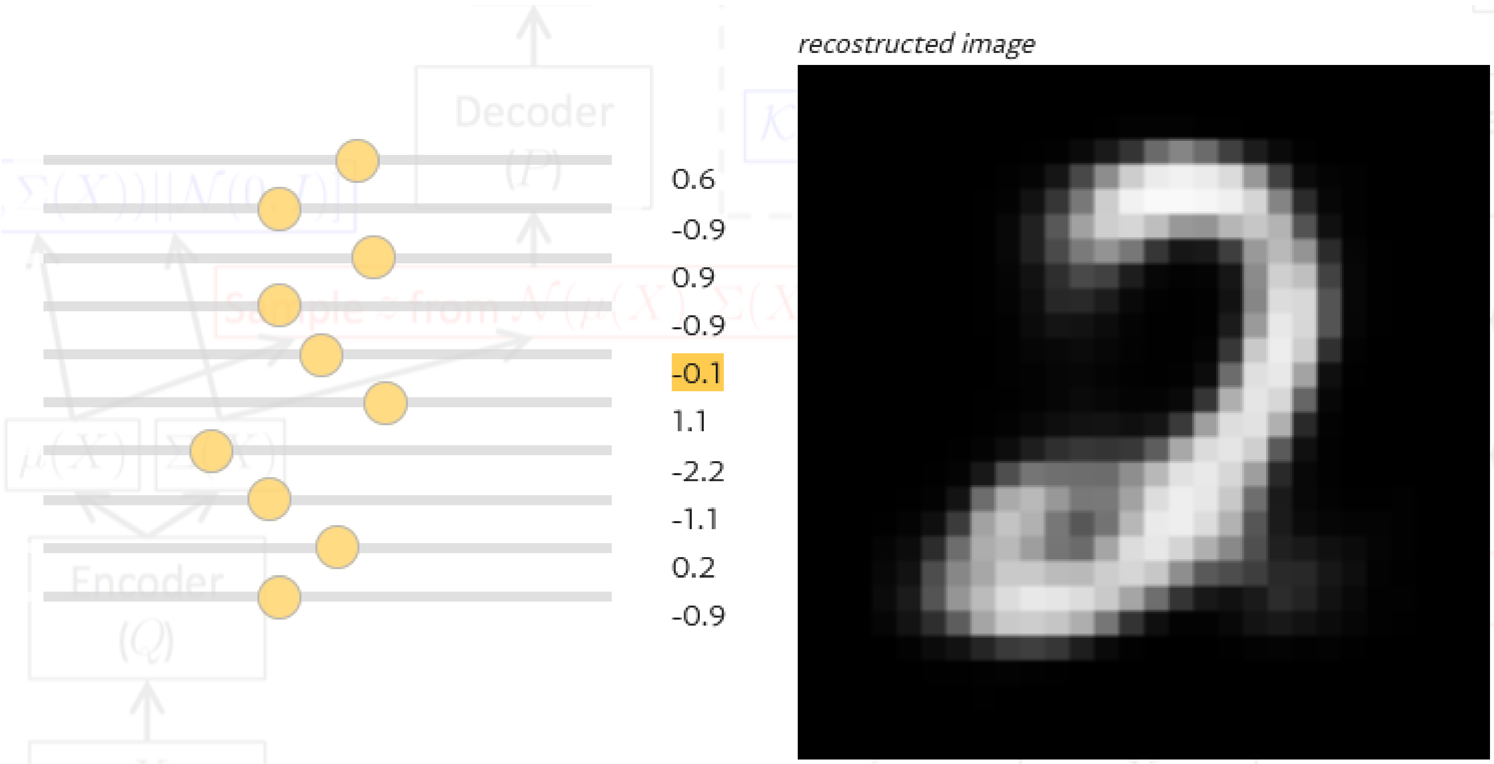
$$L(w_1, w_2, x) = (reconstruction\ loss) + (regularization\ term)$$

Reparametrisation trick

Instead of thinking about z as a sample from a gaussian $z \sim N(\mu, \sigma)$ think of it as the sum of the μ and a fixed vector σ scaled by a random constant ϵ



Variational Autoencoder



https://www.siarez.com/projects/variational-autoencoder

Autoencoders: standard

```
class StandardAE(nn.Module):
   def __init__(self):
       super(VAE, self).__init__()

       self.fc1 = nn.Linear(784, 400)
       self.fc21 = nn.Linear(400, 20)

       self.fc3 = nn.Linear(20, 400)
       self.fc4 = nn.Linear(400, 784)

   def encode(self, x):
       h1 = F.relu(self.fc1(x))
       return self.fc21(h1)







   def decode(self, z):
       h3 = F.relu(self.fc3(z))
       return F.sigmoid(self.fc4(h3))

   def forward(self, x):
       z          = self.encode(x)

       return self.decode(z)
```

Autoencoders: code – VAE

```
class VAE(nn.Module):
   def __init__(self):
       super(VAE, self).__init__()

       self.fc1 = nn.Linear(784, 400)
       self.fc21 = nn.Linear(400, 20)
       self.fc22 = nn.Linear(400, 20)
       self.fc3 = nn.Linear(20, 400)
       self.fc4 = nn.Linear(400, 784)

   def encode(self, x):
       h1 = F.relu(self.fc1(x))
       return self.fc21(h1), self.fc22(h1)









   def decode(self, z):
       h3 = F.relu(self.fc3(z))
       return F.sigmoid(self.fc4(h3))

   def forward(self, x):
       mu, logvar = self.encode(x)

       return self.decode(z), mu, logvar
```

Autoencoders: code – VAE

```
class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()

        self.fc1 = nn.Linear(784, 400)
        self.fc21 = nn.Linear(400, 20)
        self.fc22 = nn.Linear(400, 20)
        self.fc3 = nn.Linear(20, 400)
        self.fc4 = nn.Linear(400, 784)

    def encode(self, x):
        h1 = F.relu(self.fc1(x))
        return self.fc21(h1), self.fc22(h1)

    def reparametrize(self, mu, logvar):
        std = logvar.mul(0.5).exp_()
        eps = torch.cuda.FloatTensor(std.size()).normal_()
        eps = Variable(eps)
        return eps.mul(std).add_(mu)

    def decode(self, z):
        h3 = F.relu(self.fc3(z))
        return F.sigmoid(self.fc4(h3))

    def forward(self, x):
        mu, logvar = self.encode(x)
        z = self.reparametrize(mu, logvar)
        return self.decode(z), mu, logvar
```

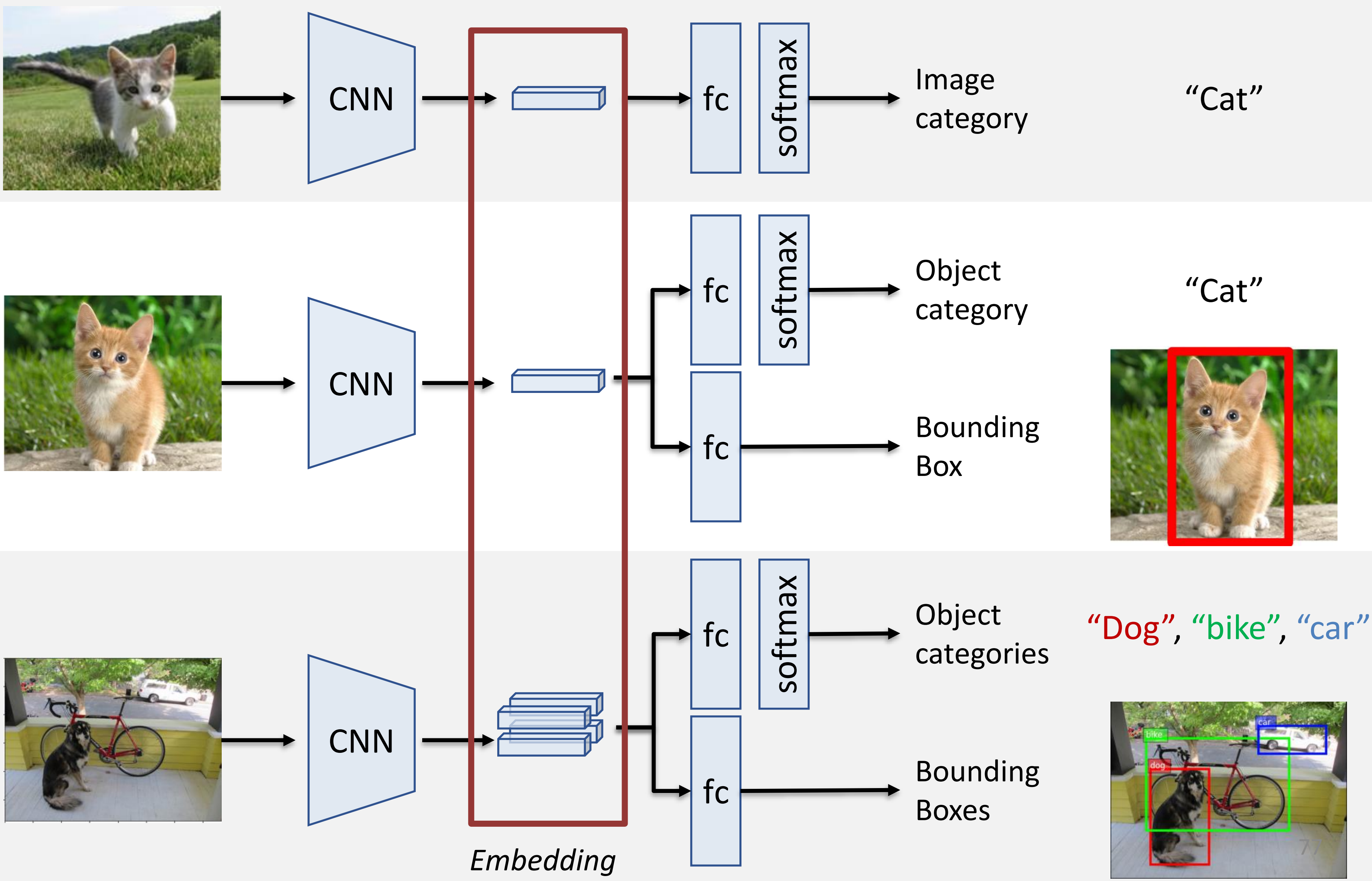
$$z = \mu + \sigma\epsilon$$

```
def loss_function(recon_x, x, mu, logvar):
    BCE = F.binary_cross_entropy(recon_x,
                                 x.view(-1, 784), reduction='sum')

    # see Appendix B from VAE paper:
    # Kingma and Welling. Auto-Encoding Variational Bayes
    # https://arxiv.org/abs/1312.6114
    # 0.5 * sum(1 + log(sigma^2) - mu^2 - sigma^2)
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2)
                           - logvar.exp())

    return BCE + KLD
```

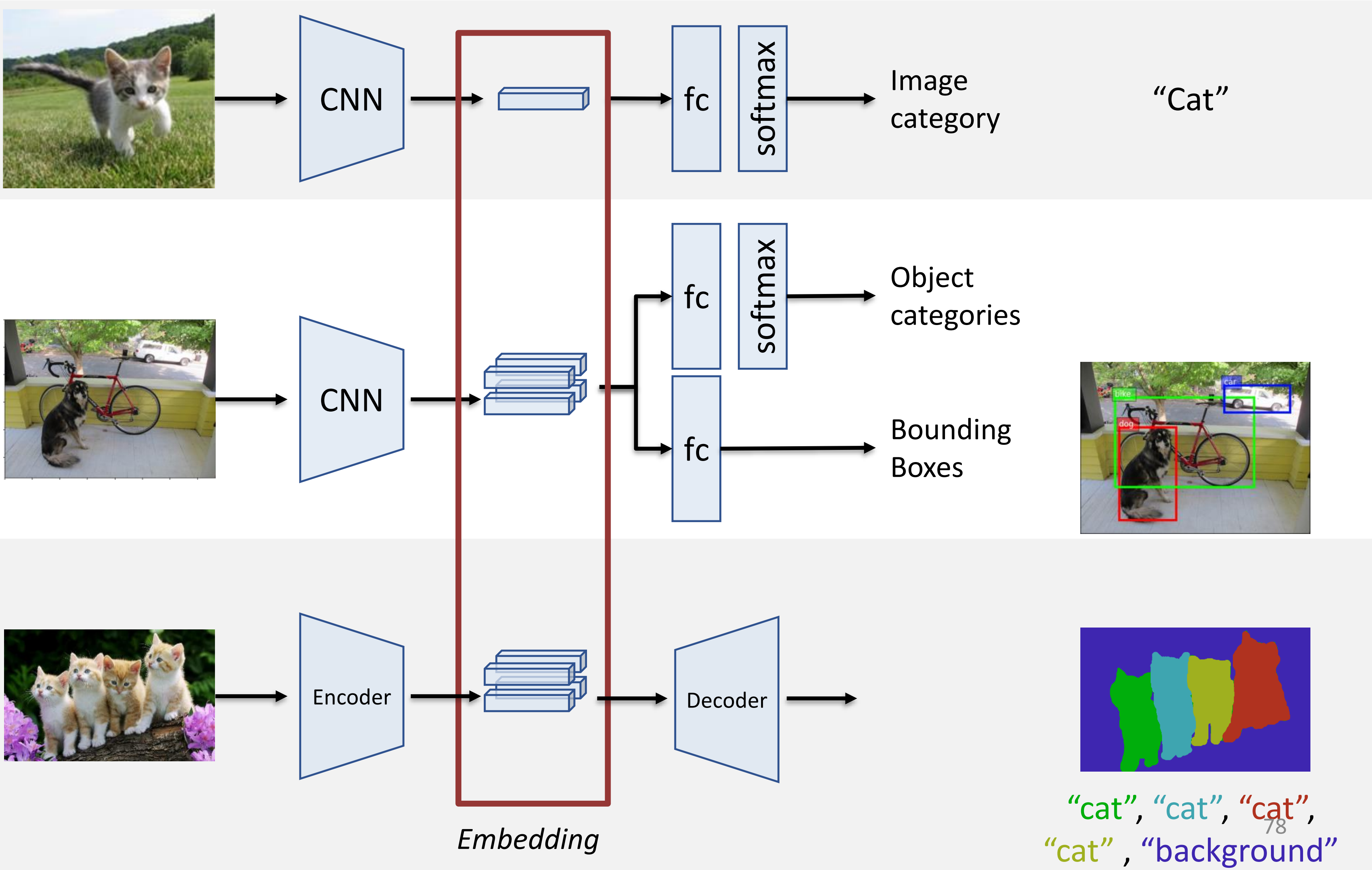
Encoder-Decoder Architectures

EXTENDING THE CONCEPT

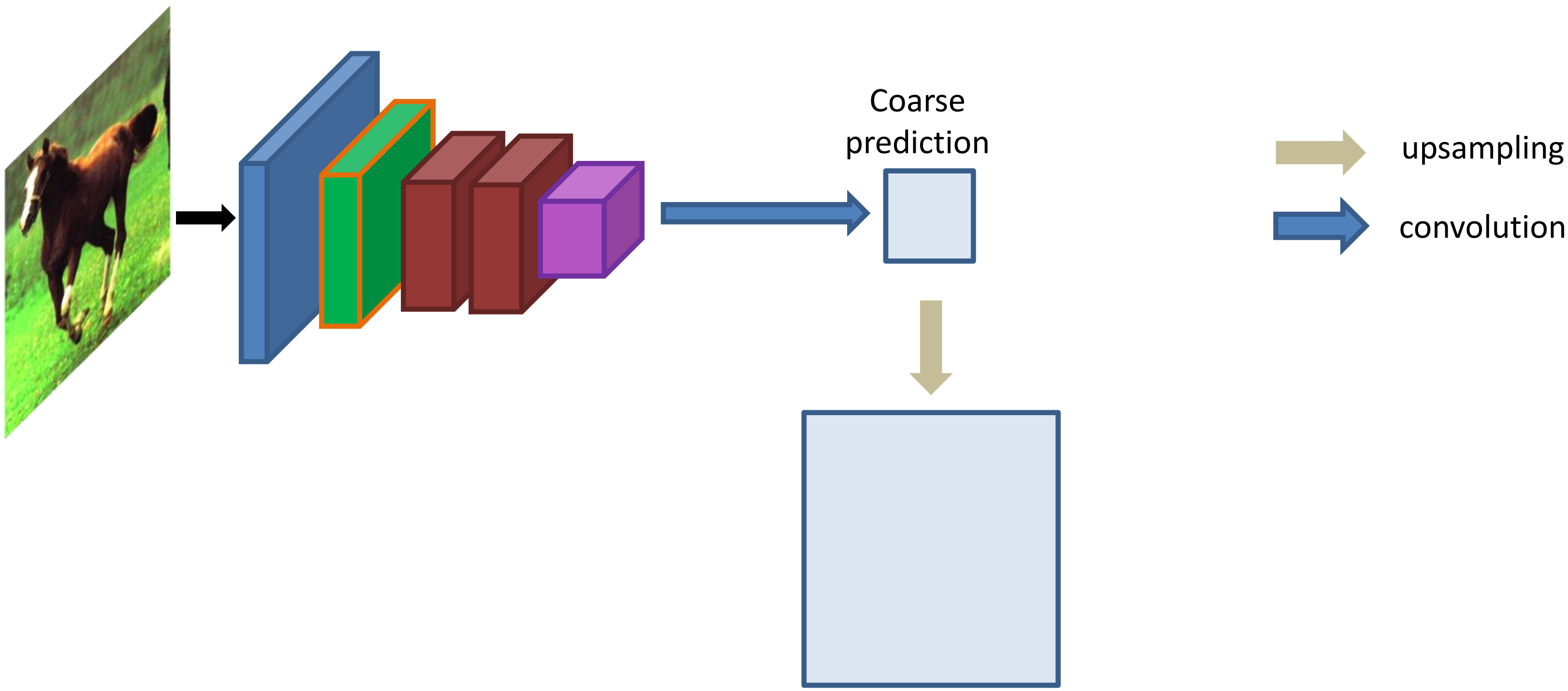
Computer Vision Tasks

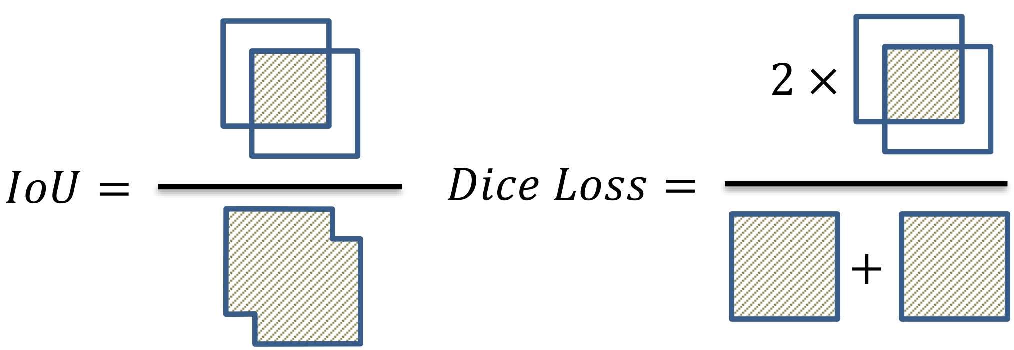


Computer Vision Tasks



Autoencoder for image segmentation





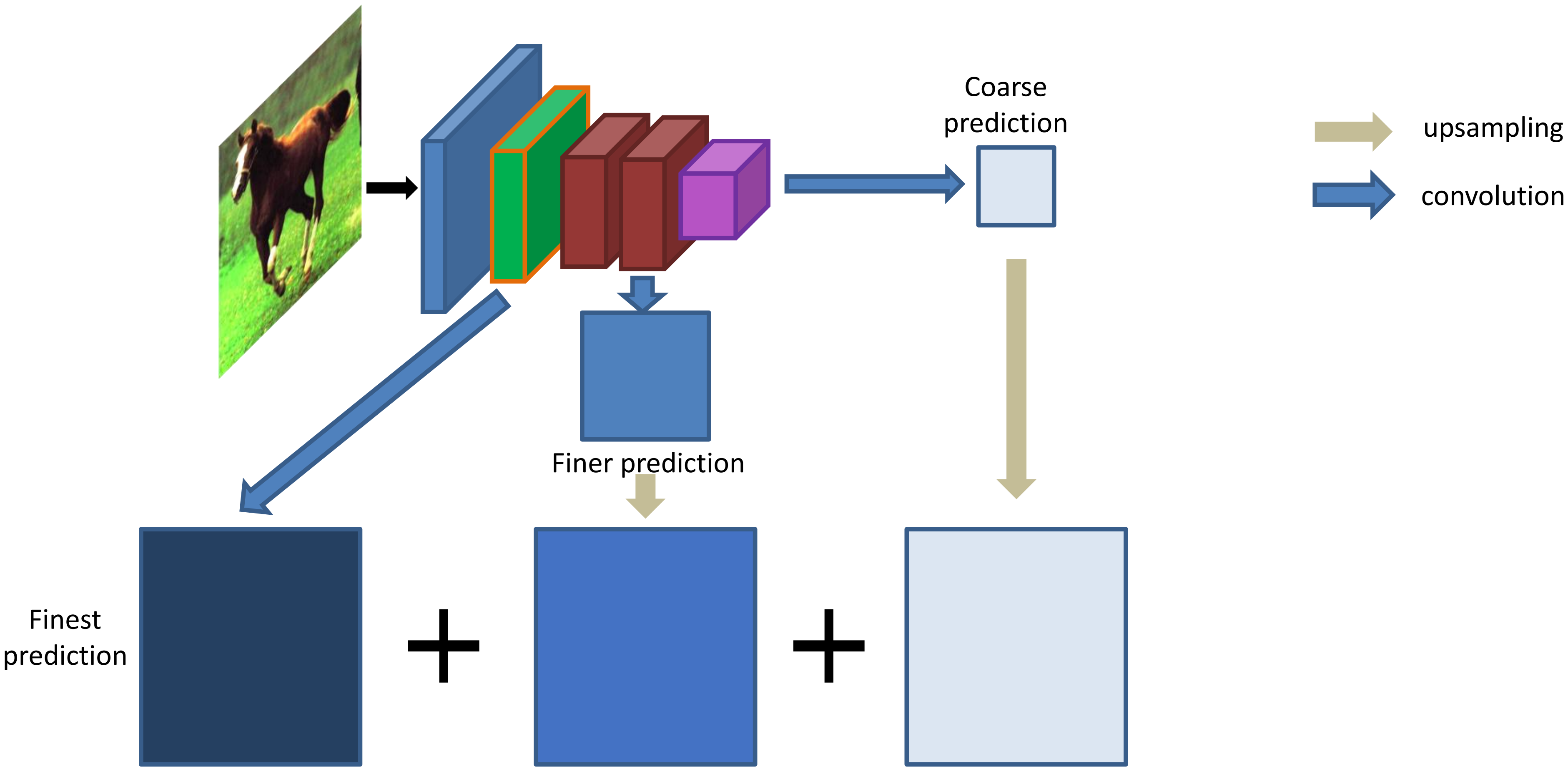
$$L_{DICE}(y, \hat{y}) = \frac{2 \sum_i^N y_i \hat{y}_i}{\sum_i^N y_i + \sum_i^N \hat{y}}$$

[A survey of loss functions for semantic segmentation - arXiv](#)

Semantic segmentation using convolutional networks

Very coarse segmentation

Solution: Skip connections

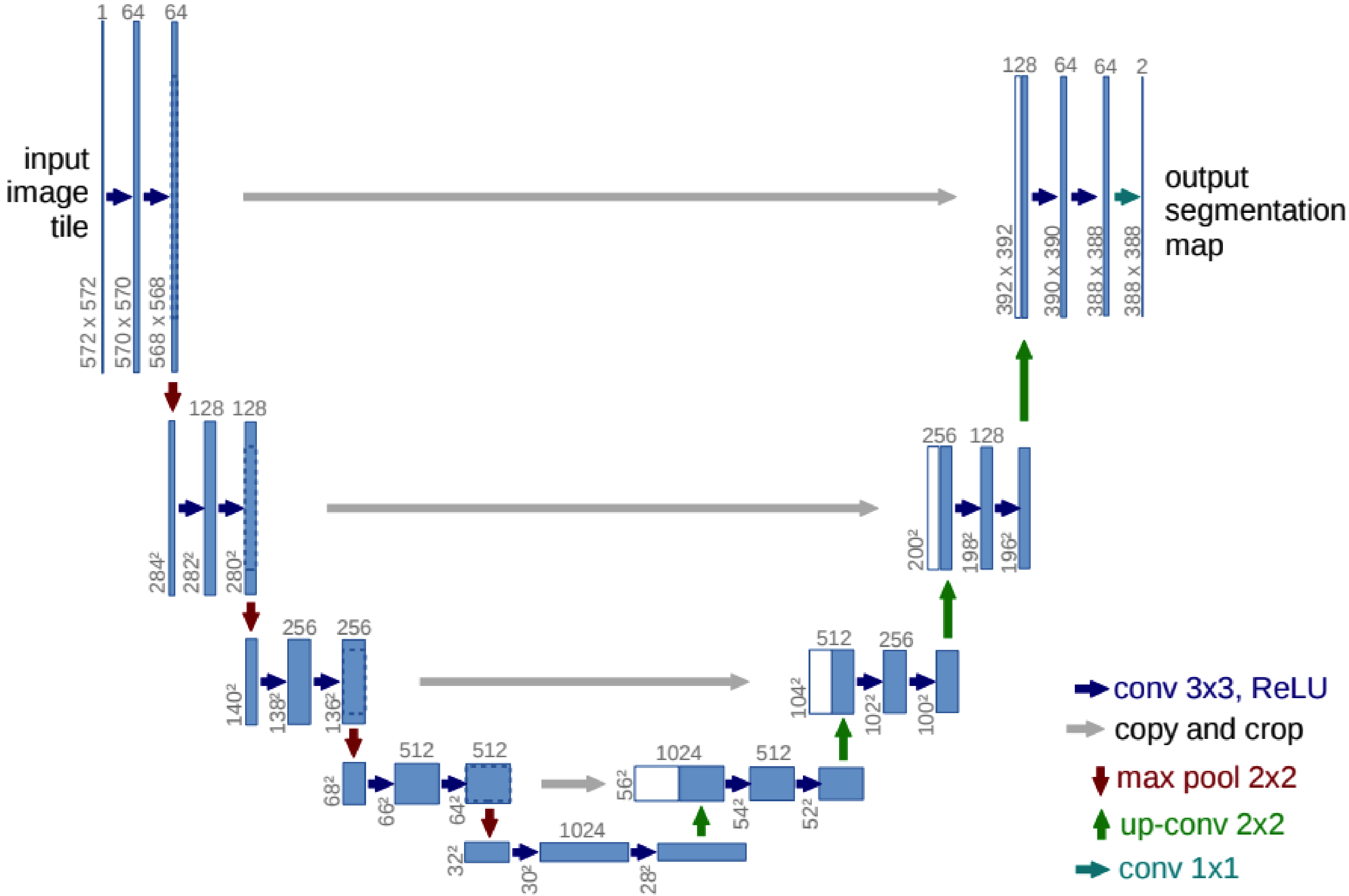


Skip connections

Details are still a problem

Fully convolutional networks for semantic segmentation. Evan Shelhamer, Jon Long, Trevor Darrell. In *CVPR* 2015

U-Net



Ronneberger et al, "U-net: Convolutional networks for biomedical image segmentation." MICAI 2015

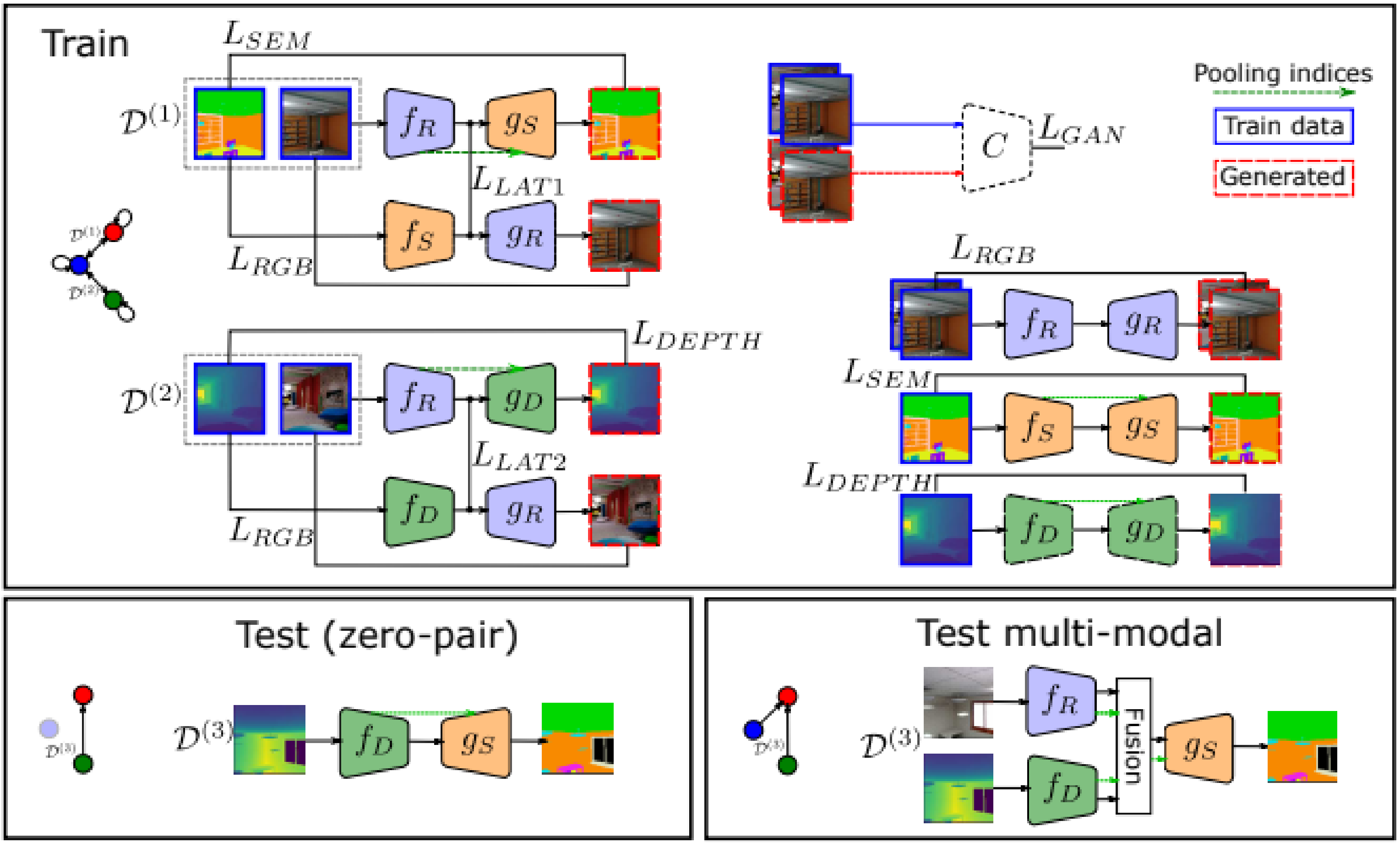
U-Net

Input Ground truth Prediction

FC-DenseNet103 model on UNET

Jegou et al, *“The One Hundred Layers Tiramisu: Fully Convolutional DenseNets for Semantic Segmentation”*, 2017

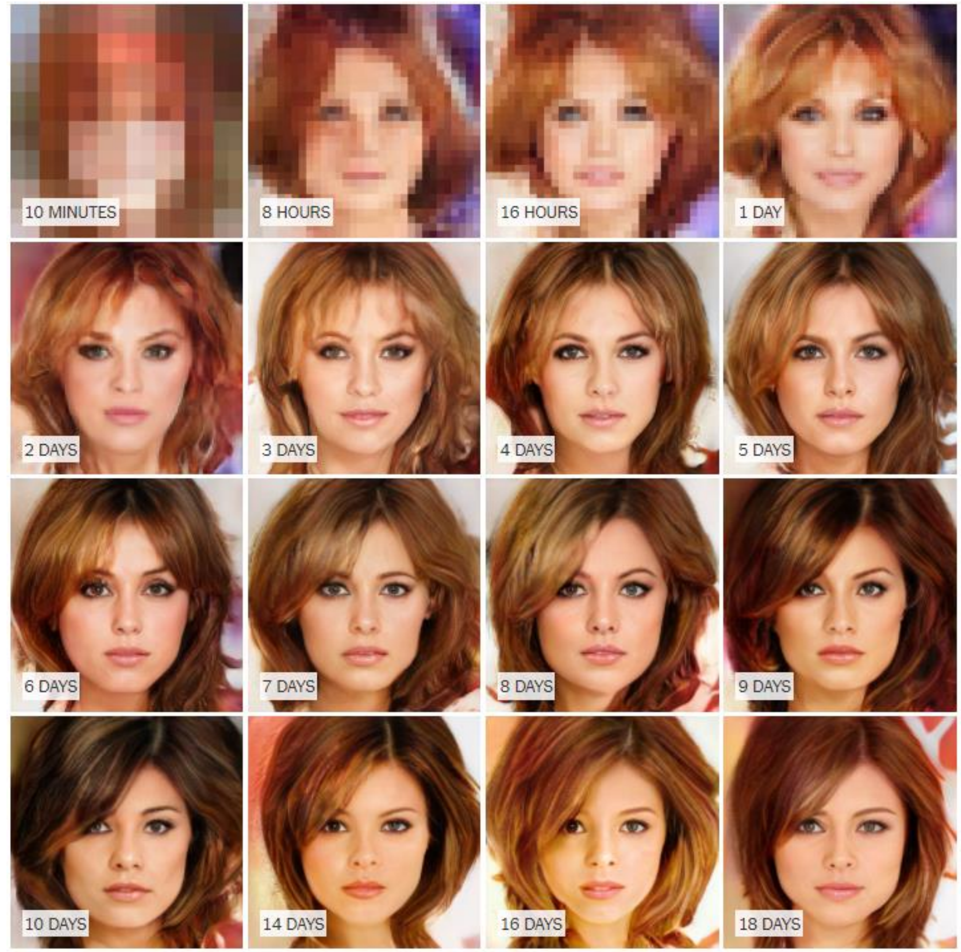
Mixing and Matching encoders and decoders



Wang et al, "*Mix and match networks: encoder-decoder alignment for zero-pair image translation*" CVPR 2018

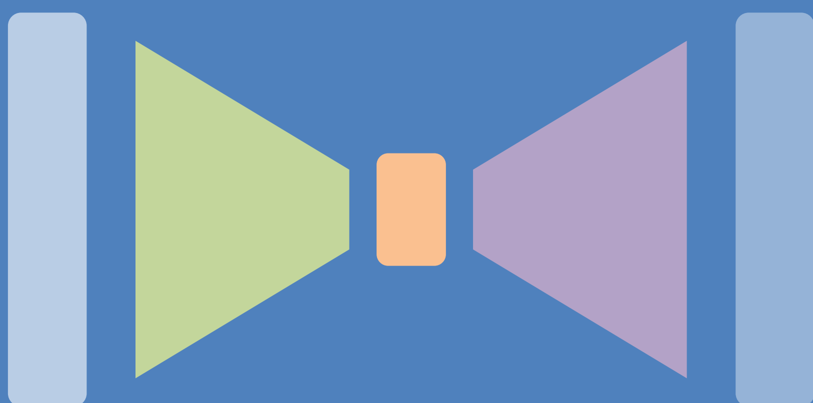
GENERATIVE ADVERSARIAL NETWORKS

Real or Fake?

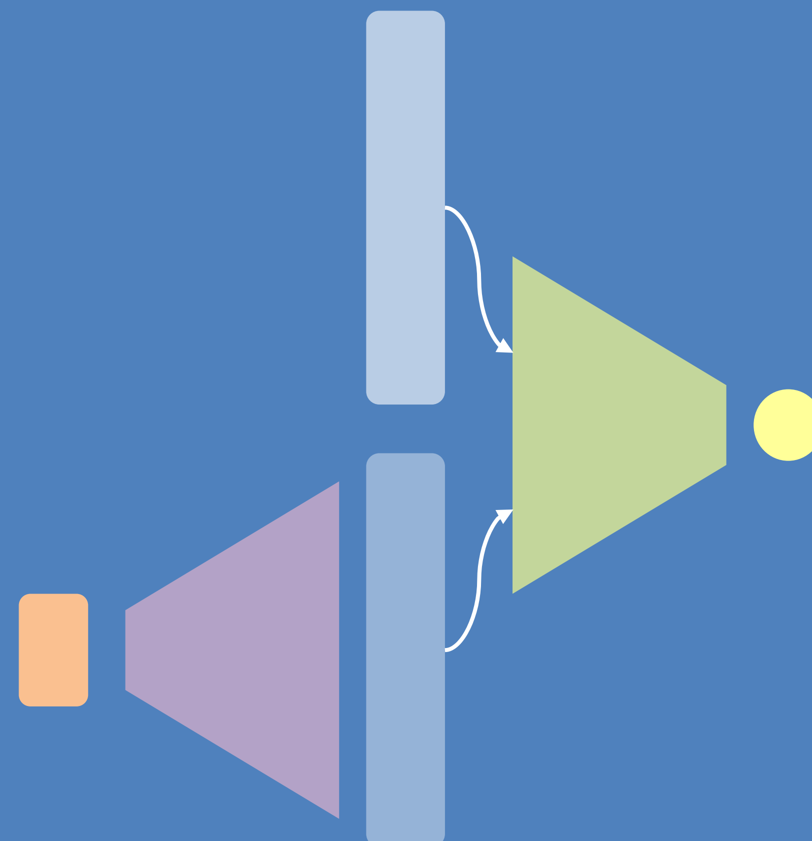


https://www.nytimes.com/interactive/2018/01/02/technology/ai-generated-photos.html

Autoencoders

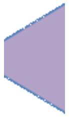


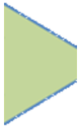
Generative Adversarial Networks (GANs)

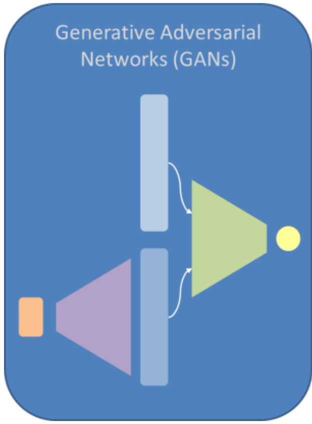


GAN: Generative Adversarial Networks

Two NNs competing against each other

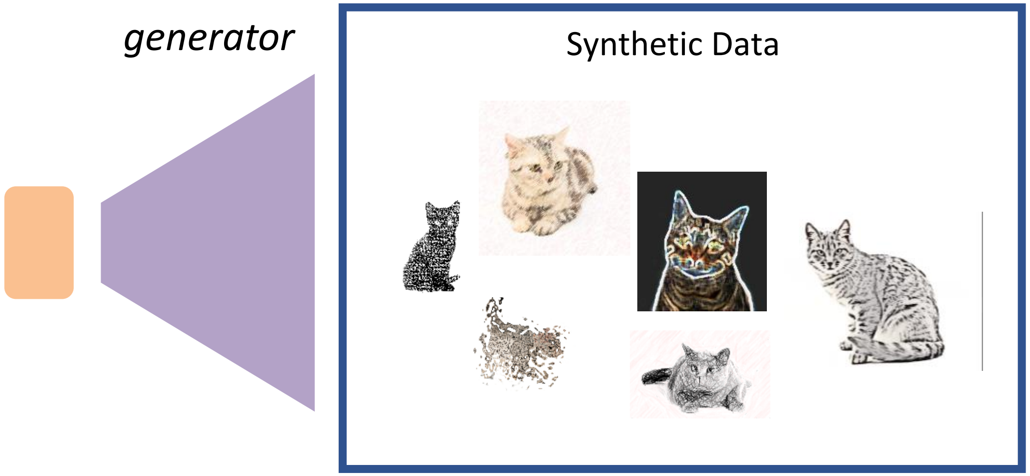
- The **generator** NN learns to generate plausible data. The generated instances become negative training examples for the discriminator. 

- The **discriminator** NN learns to distinguish the generator's fake data from real data. The discriminator penalizes the generator for producing implausible results. 

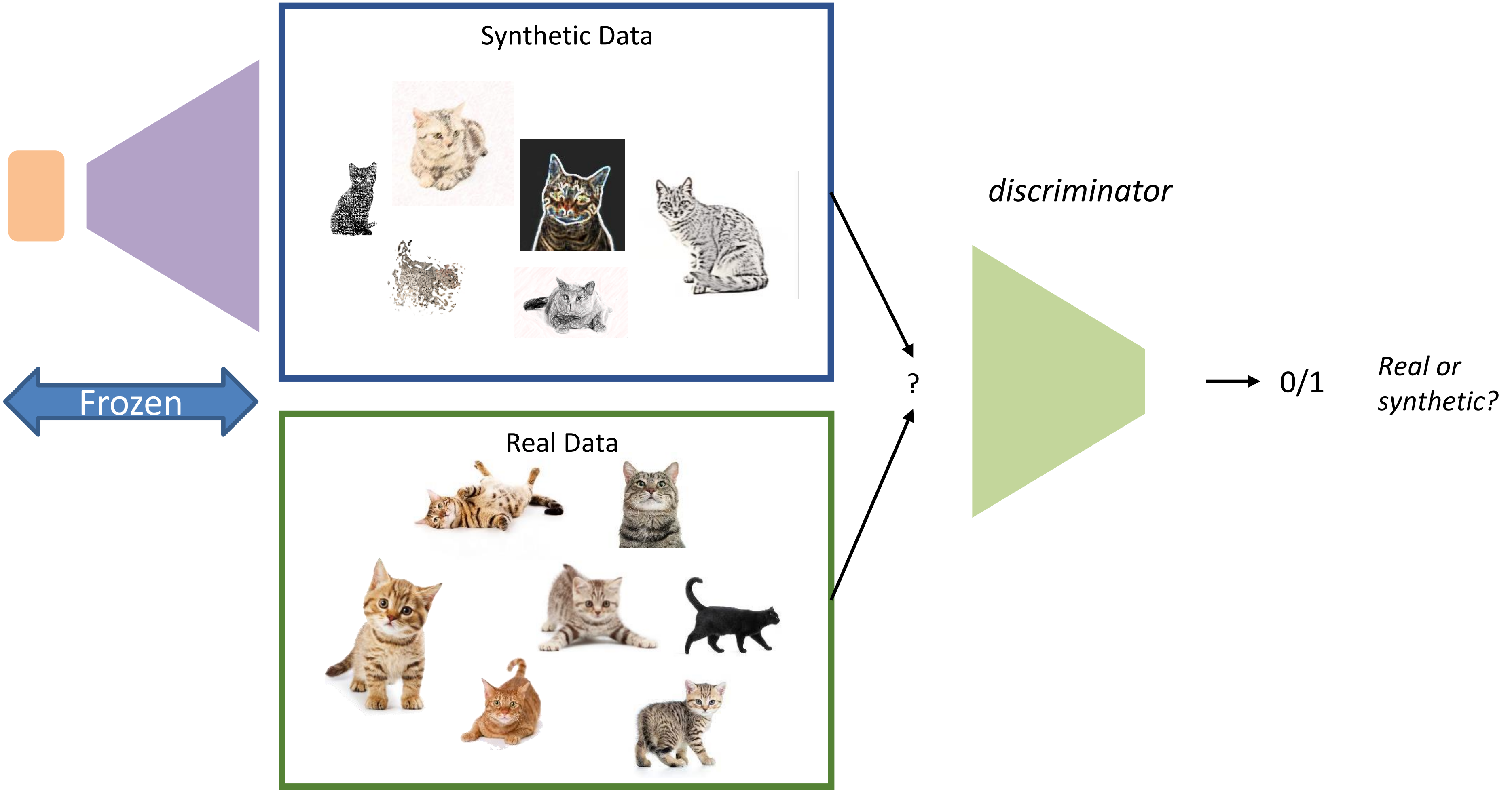


[Google’s Machine Learning course]

GAN Process

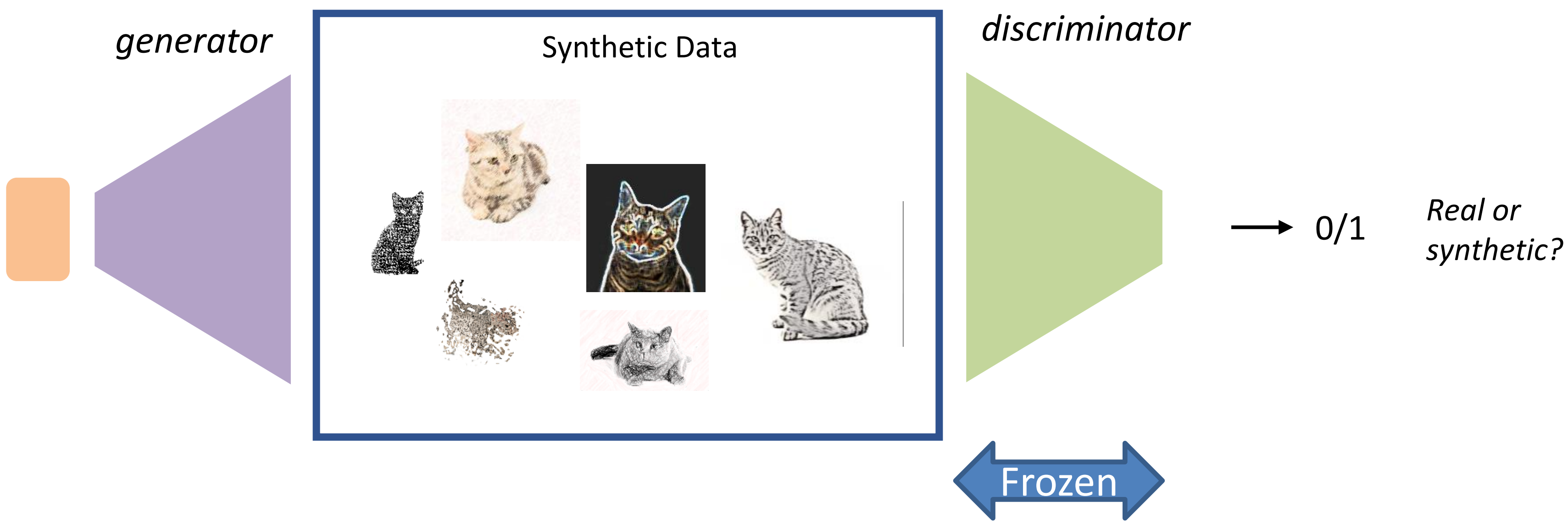


GAN Process



Synthetic Data
Frozen
Real Data
?
discriminator
0/1
Real or synthetic?

GAN Process


generator
Synthetic Data
discriminator
0/1
Real or synthetic?
Frozen

Generative Adversarial Networks

2014

The OG GAN

2015

Deep Convolutional GAN

2016

Coupled GAN

2017

Progressively Growing GAN

2018

Style-based GAN

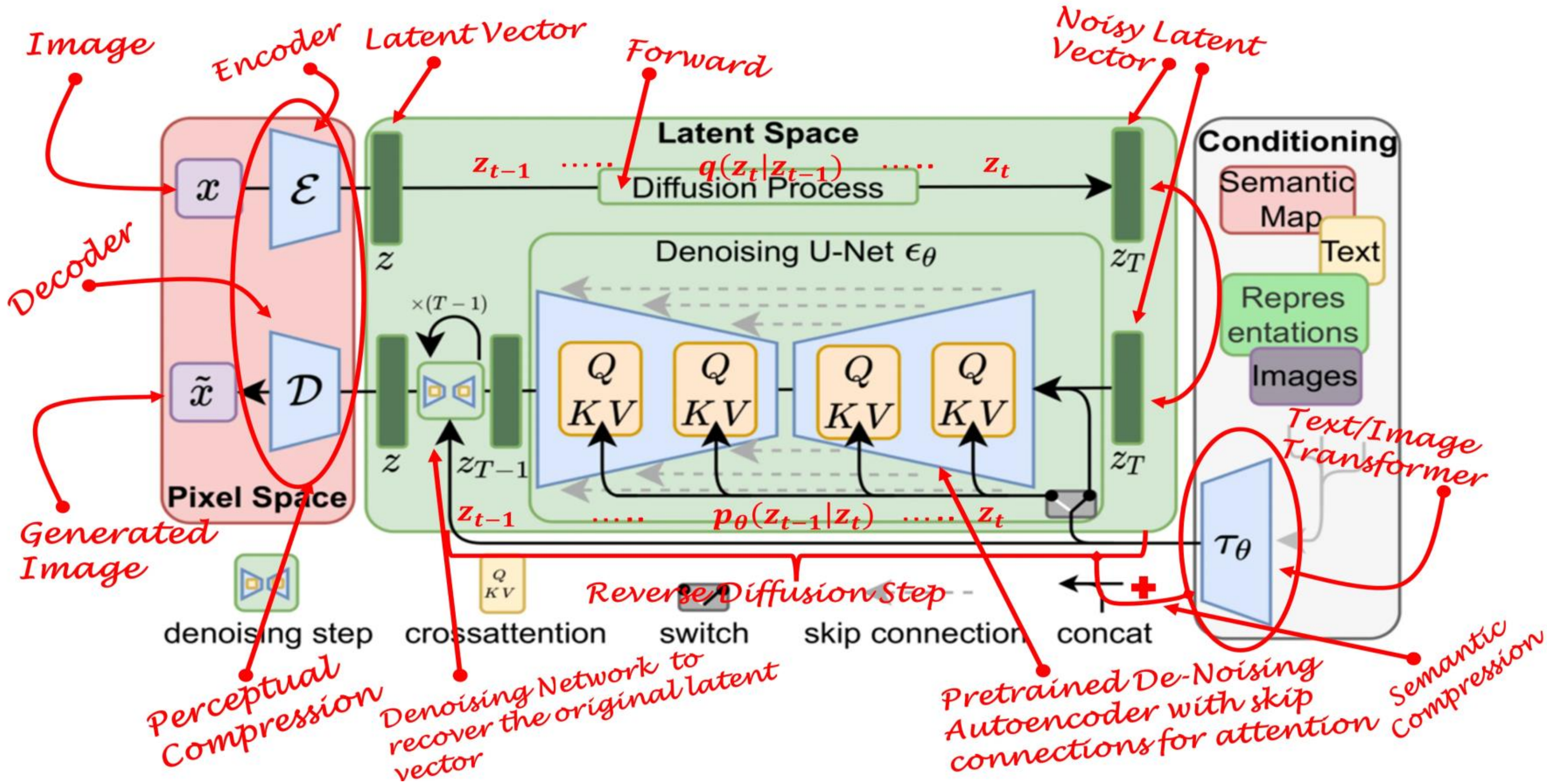
2020

Improved Style-based GAN

The GAN zoo

https://github.com/hindupuravinash/the-gan-zoo

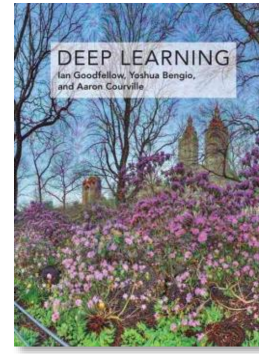
And now? Stable Difussion Models

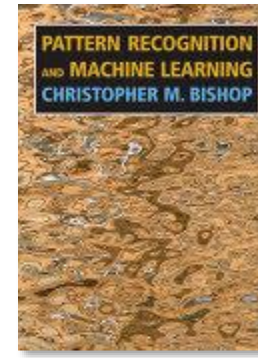


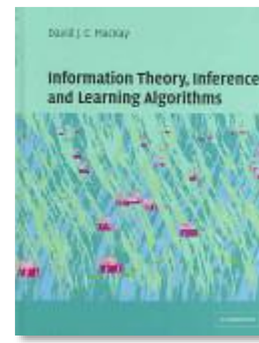
https://towardsdatascience.com/what-are-stable-diffusion-models-and-why-are-they-a-step-forward-for-image-generation-aa1182801d46

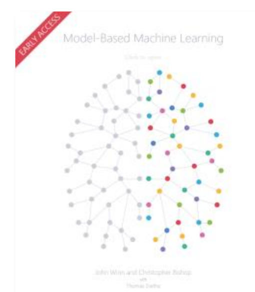
Resources (I)

- *I. Goodfellow, Y. Bengio, A. Courville, "Deep Learning", MIT Press, 2016*
  http://www.deeplearningbook.org/
- *C. Bishop, "Pattern Recognition and Machine Learning", Springer, 2006*
  http://research.microsoft.com/en-us/um/people/cmbishop/prml/index.htm
- *D. MacKay, "Information Theory, Inference and Learning Algorithms", Cambridge University Press, 2003*
  http://www.inference.phy.cam.ac.uk/mackay/
- *R.O. Duda, P.E. Hart, D.G. Stork, "Pattern Classification", Wiley & Sons, 2000*
  http://books.google.com/books/about/Pattern_Classification.html?id=Br33IRC3PkQC
- *J. Winn, C. Bishop, "Model-Based Machine Learning", early access*
  http://mbmlbook.com/









Further Info

- Many of the slides of these lectures have been adapted from various highly recommended online lectures and courses:
  - Andrew Ng's *Machine Learning Course*, Coursera
    https://www.coursera.org/course/ml
  - Andrew Ng's *Deep Learning Specialization,* Coursera
    https://www.coursera.org/specializations/deep-learning
  - Victor Lavrenko's *Machine Learning* Course
    https://www.youtube.com/channel/UCs7alOMRnxhzfKAJ4JjZ7Wg
  - Fei Fei Li and Andrej Karpathy's *Convolutional Neural Networks for Visual Recognition*
    http://cs231n.stanford.edu/
  - Geoff Hinton's *Neural Networks for Machine Learning, (*ex Coursera)
    https://www.youtube.com/playlist?list=PLiPvV5TNogxKKwvKb1RKwkq2hm7ZvpHz0
  - Luis Serrano's introductory videos
    https://www.youtube.com/channel/UCgBncpylJ1kiVaPyP-PZauQ
  - Michael Nielsen's *Neural Networks and Deep Learning*
    http://neuralnetworksanddeeplearning.com/
  - David Charte et al. A practical tutorial on autoencoders for nonlinear feature fusion:Taxonomy, models, software and guidelines
    https://arxiv.org/abs/1801.01586