Deep Learning (for Computer Vision)

Arjun Jain

Sources

A lot of the material has been shamelessly and gratefully collected from:

- http://cs231n.stanford.edu/

Generative Modeling

- A form of unsupervised learning in which you learn to *generate complex data* like images, speech etc.
- Given training data, generate new samples from the same distribution
- Learn an estimate of the probability distribution of the training data

Training data ~ $p_{data}(x)$

Generated samples ~ $p_{model}(x)$

Want to learn $p_{model}(x)$ similar to $p_{data}(x)$

Taxonomy of Generative Models

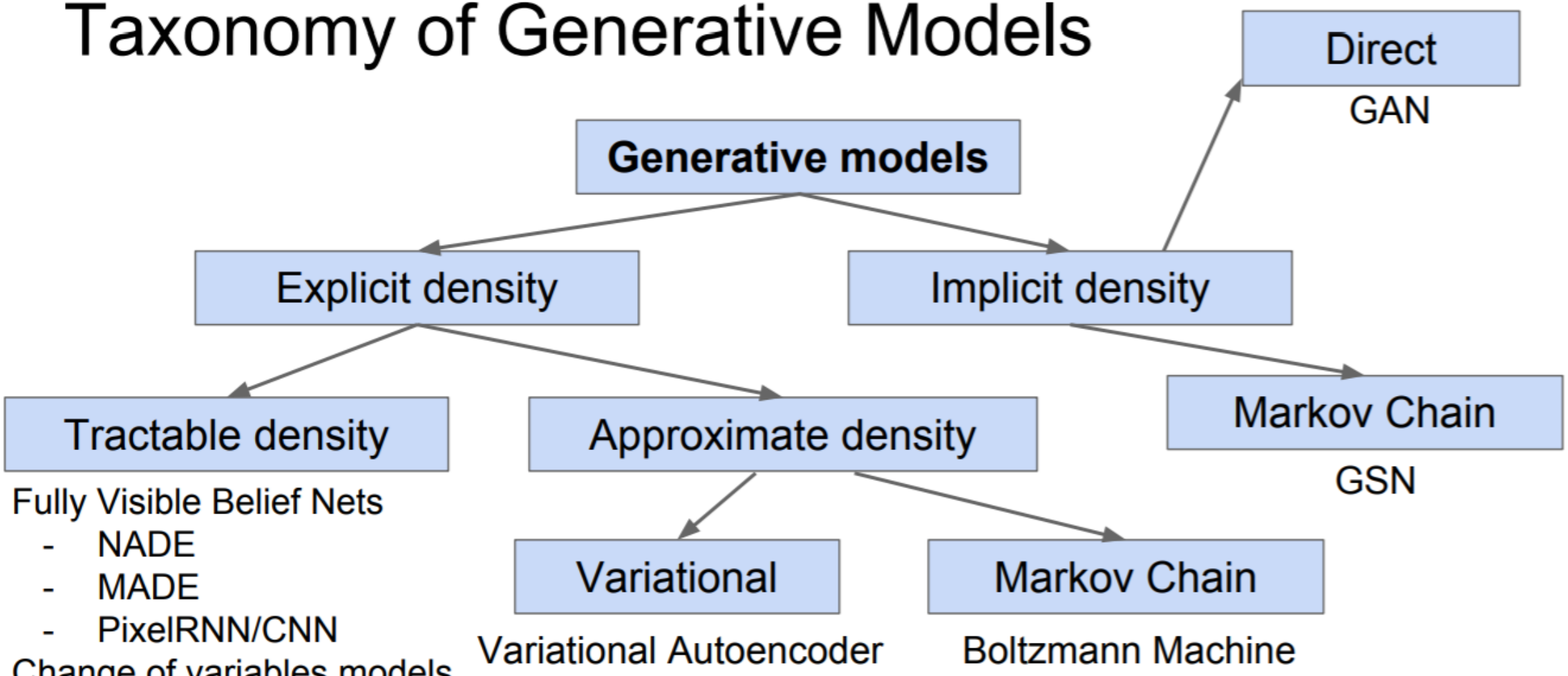


Figure copyright and adapted from Ian Goodfellow, Tutorial on Generative Adversarial Networks, 2017.

Taxonomy of Generative Models

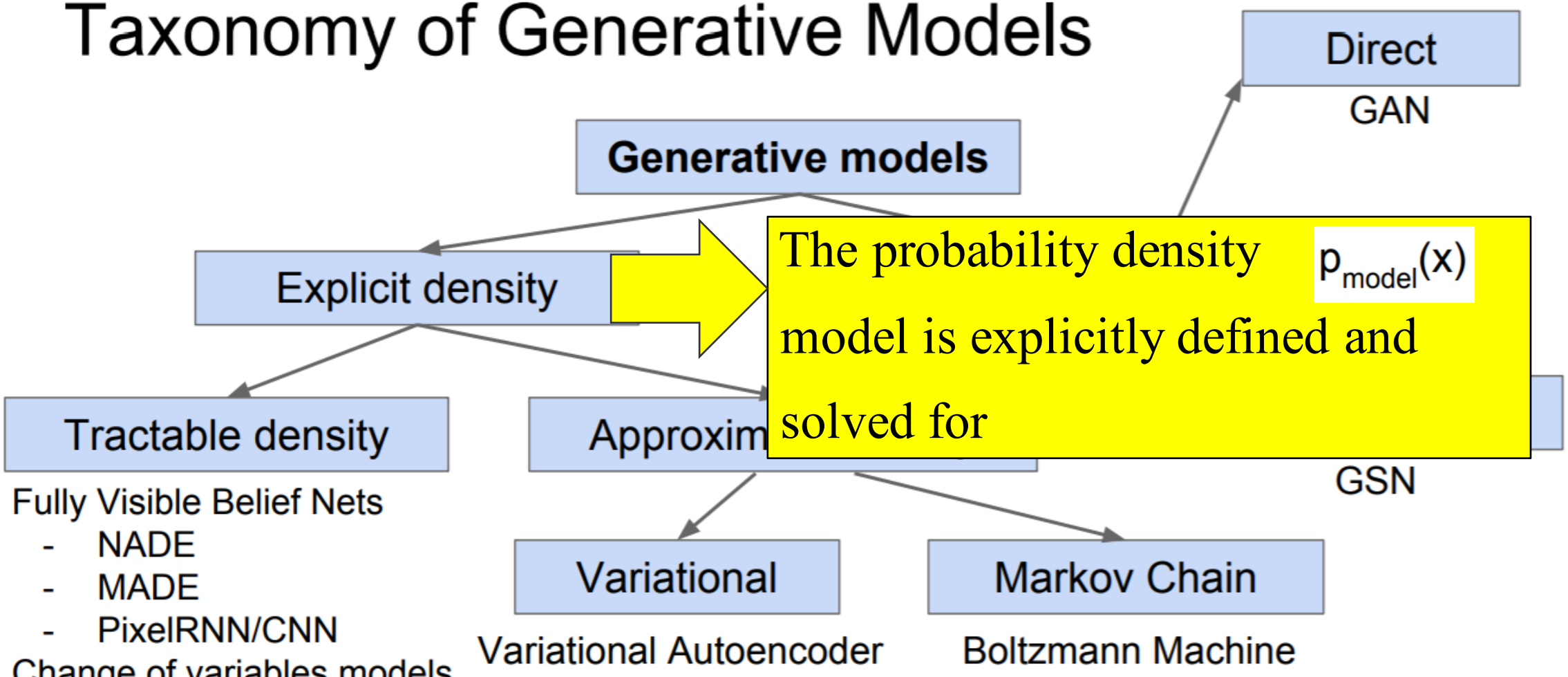
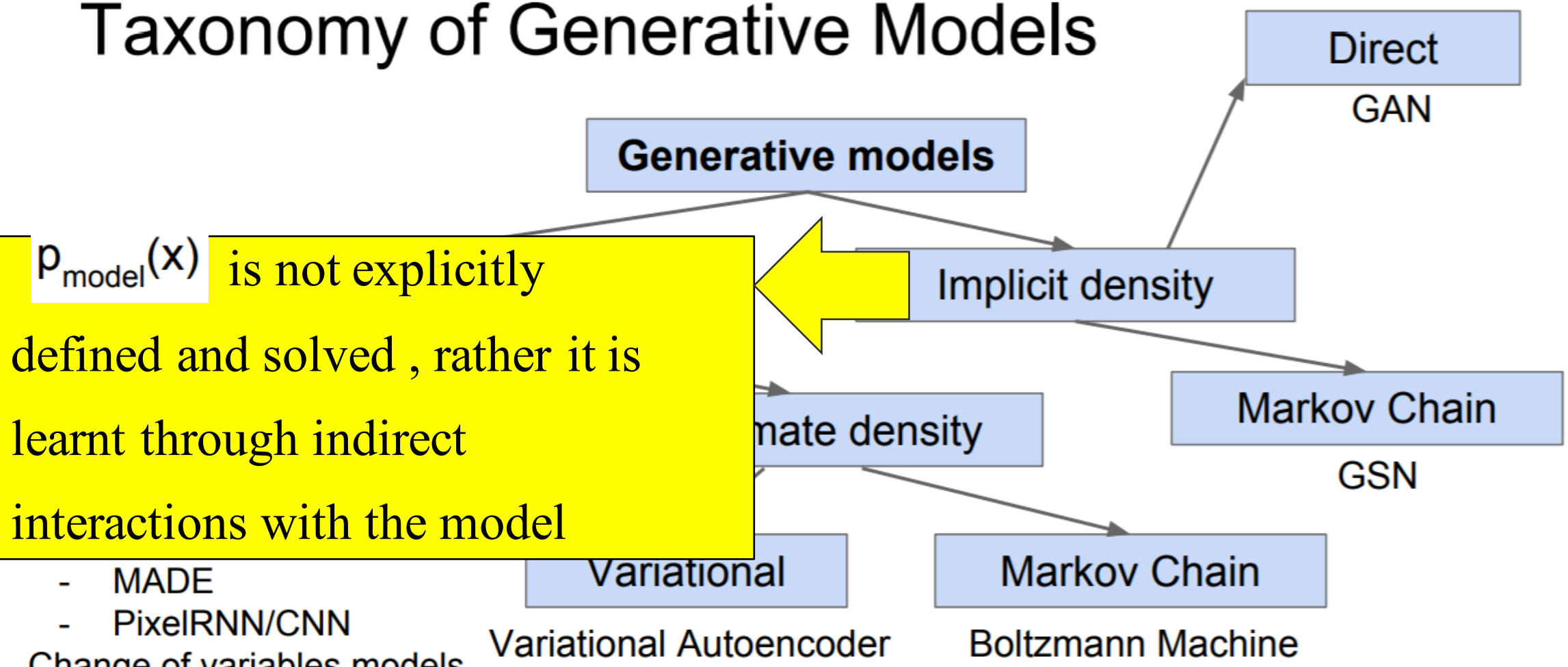


Figure copyright and adapted from Ian Goodfellow, Tutorial on Generative Adversarial Networks, 2017.

Taxonomy of Generative Models



$p_{model}(x)$ is not explicitly defined and solved , rather it is learnt through indirect interactions with the model

- MADE
- PixelRNN/CNN

Change of variables models (nonlinear ICA)

- Wavenet

Figure copyright and adapted from Ian Goodfellow, Tutorial on Generative Adversarial Networks, 2017.

Taxonomy of Generative Models

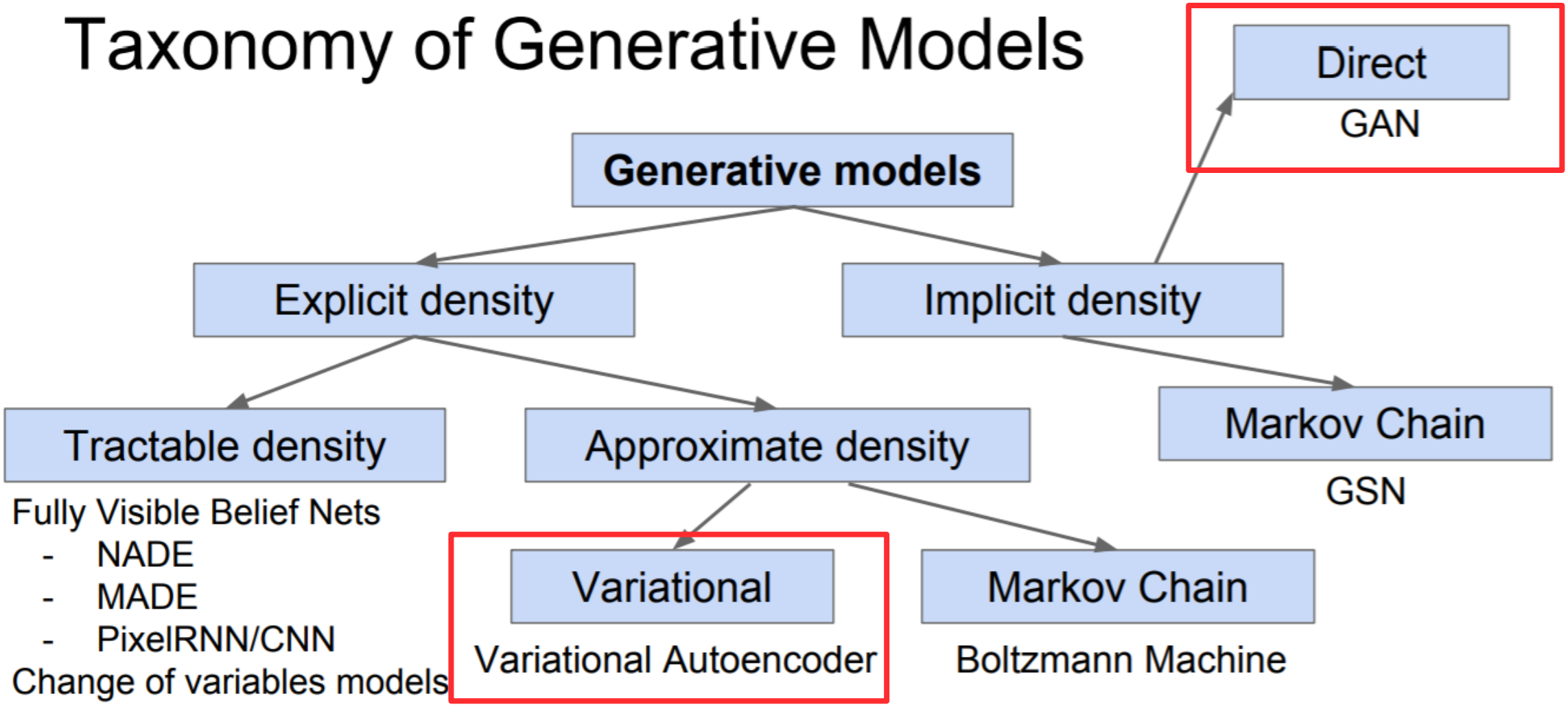
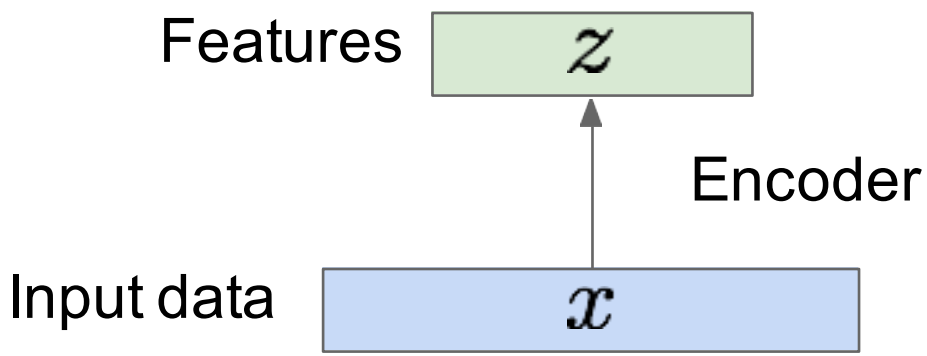
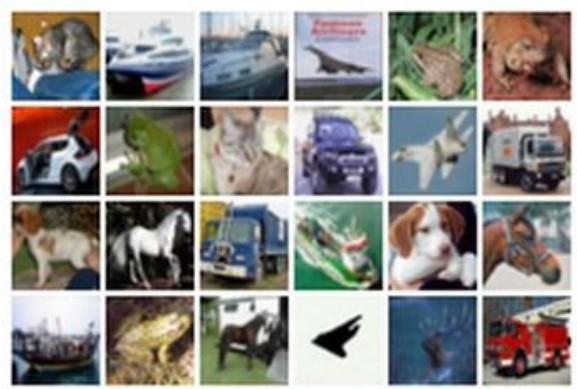


Figure copyright and adapted from Ian Goodfellow, Tutorial on Generative Adversarial Networks, 2017.

Some background first: Autoencoders

Unsupervised approach for learning a lower-dimensional feature representation from unlabeled training data

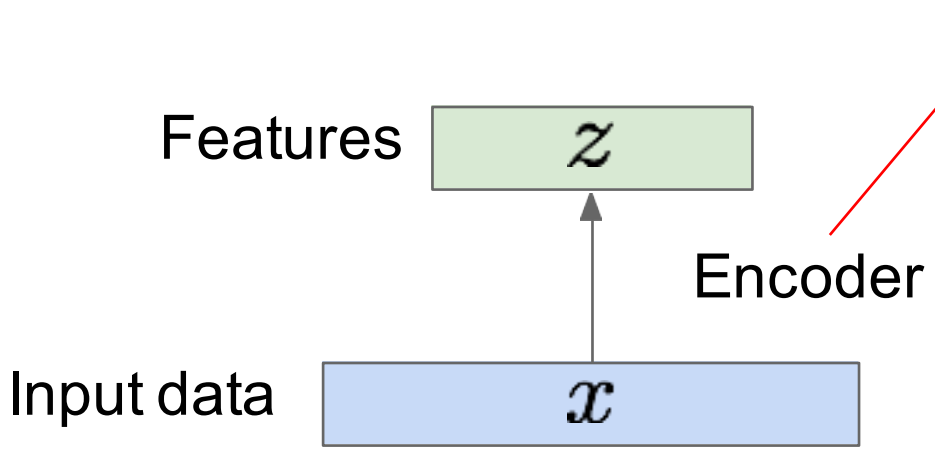


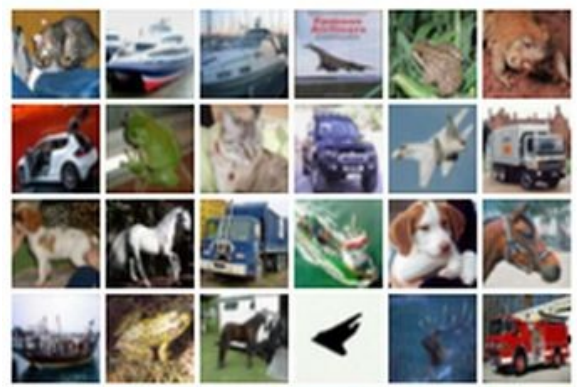


Some background first: Autoencoders

Unsupervised approach for learning a lower-dimensional feature representation from unlabeled training data

Originally: Linear + nonlinearity (sigmoid)
Later: Deep, fully-connected
Later: ReLU CNN





Some background first: Autoencoders

Unsupervised approach for learning a lower-dimensional feature representation from unlabeled training data

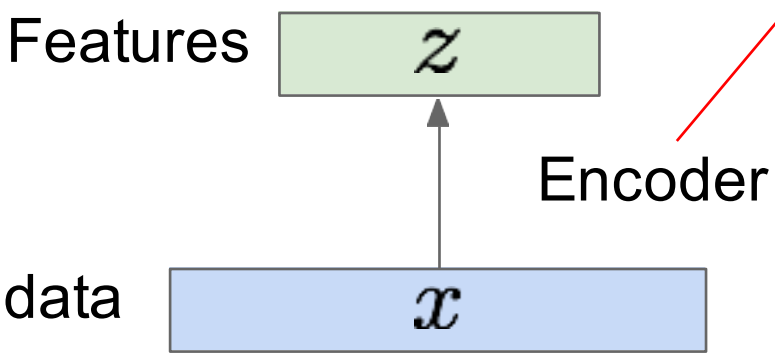
z usually smaller than **x** (dimensionality reduction)

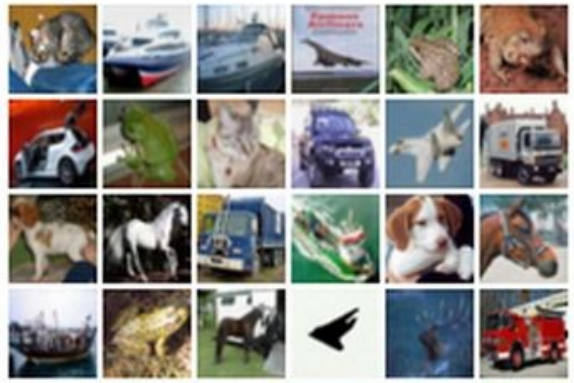
Q: Why dimensionality reduction?

Originally: Linear + nonlinearity (sigmoid)

Later: Deep, fully-connected

Later: ReLU CNN





Some background first: Autoencoders

Unsupervised approach for learning a lower-dimensional feature representation from unlabeled training data

z usually smaller than **x** (dimensionality reduction)

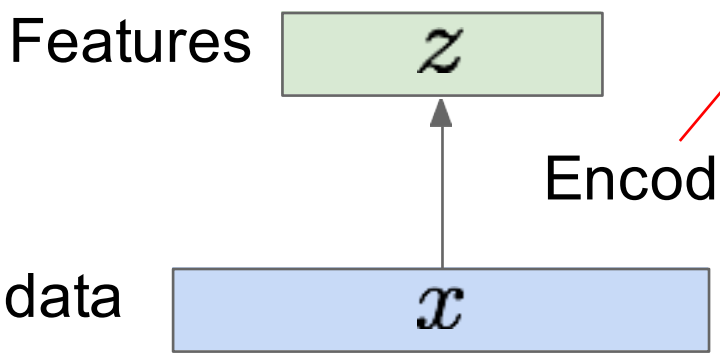
Q: Why dimensionality reduction?

A: Want features to capture meaningful factors of variation in data

Originally: Linear + nonlinearity (sigmoid)

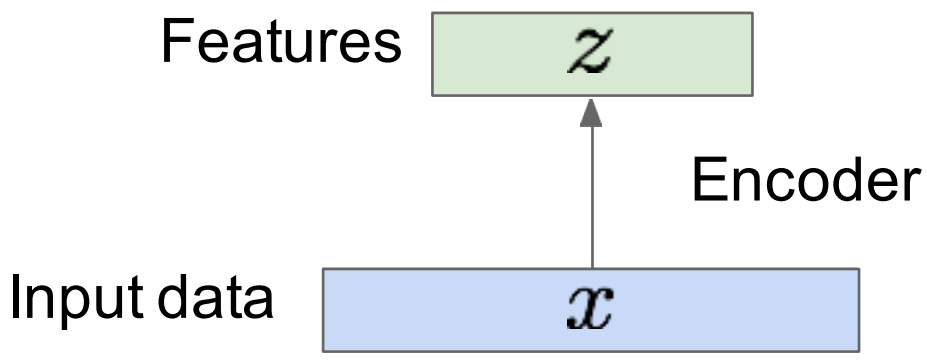
Later: Deep, fully-connected

Later: ReLU CNN



Some background first: Autoencoders

How to learn this feature representation?

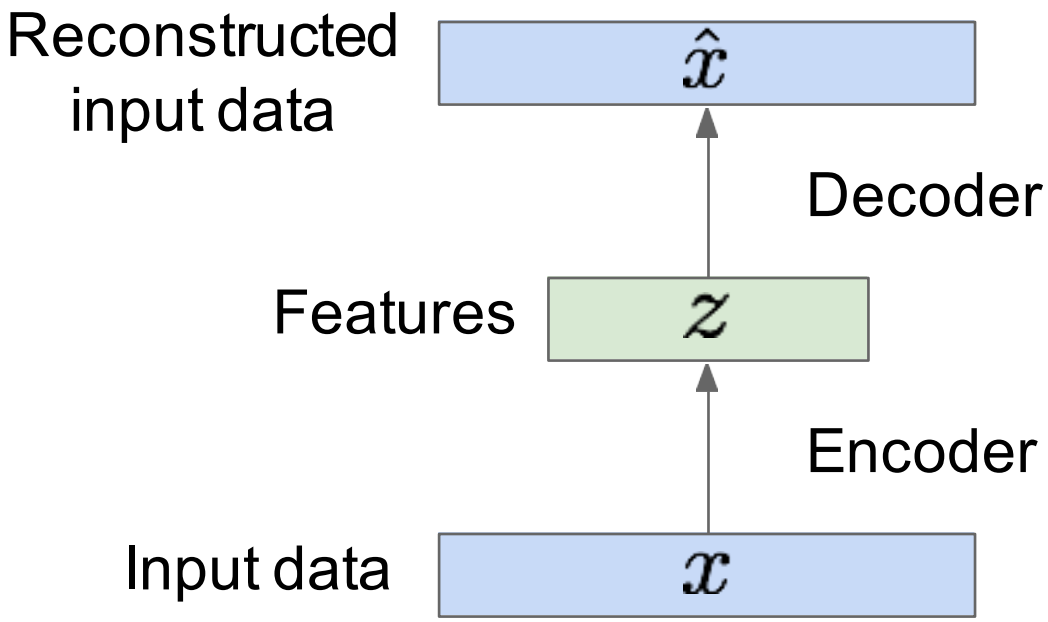


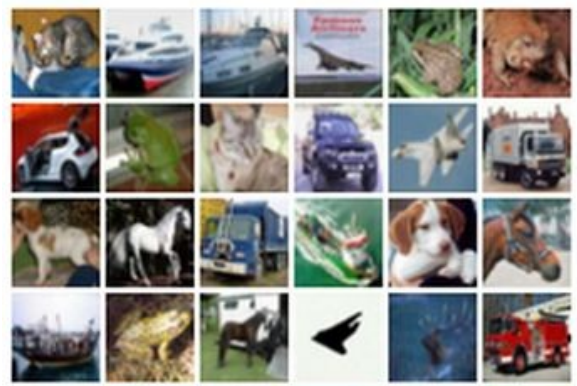
Some background first: Autoencoders

How to learn this feature representation?

Train such that features can be used to reconstruct original data

"Autoencoding" - encoding itself



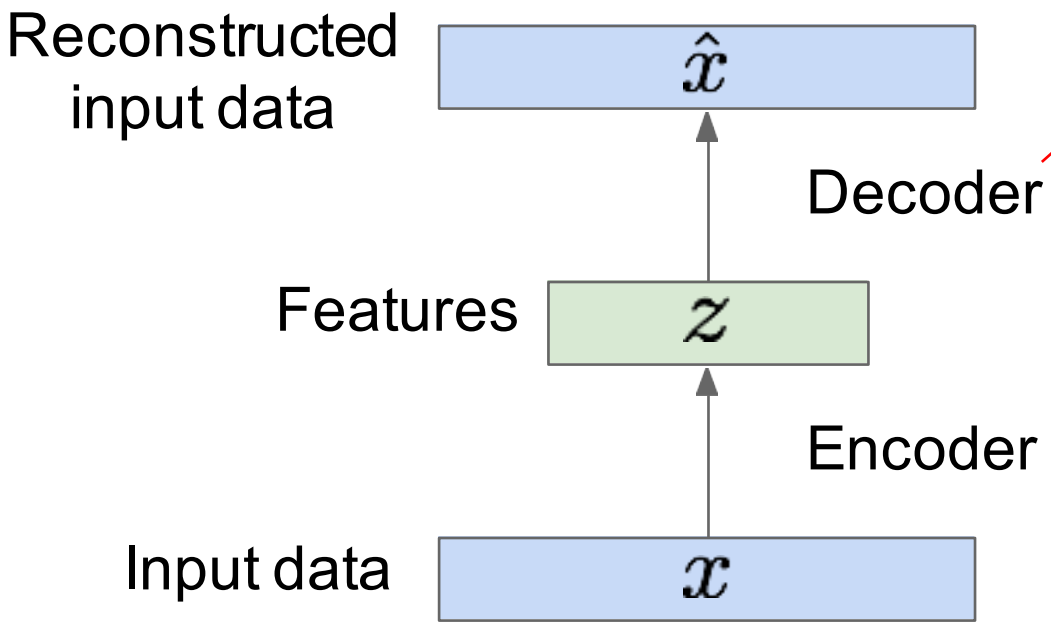


Some background first: Autoencoders

How to learn this feature representation?

Train such that features can be used to reconstruct original data

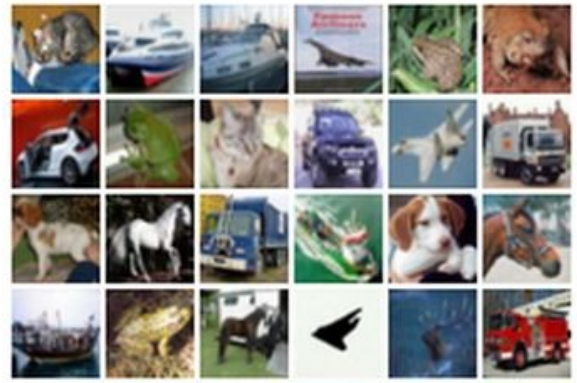
"Autoencoding" - encoding itself



Originally: Linear + nonlinearity (sigmoid)

Later: Deep, fully-connected

Later: ReLU CNN (upconv)

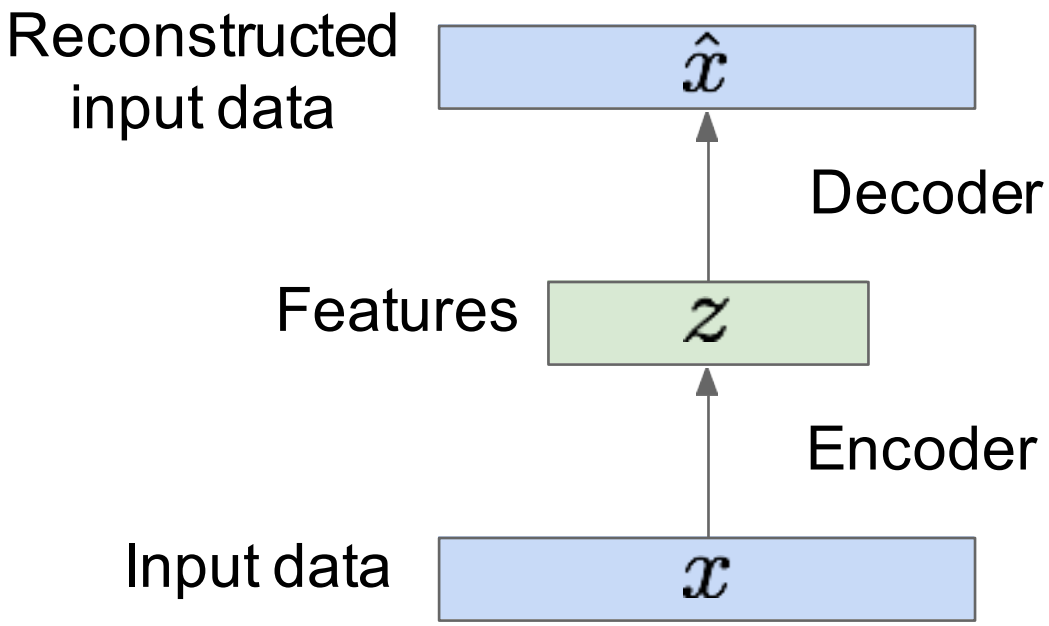


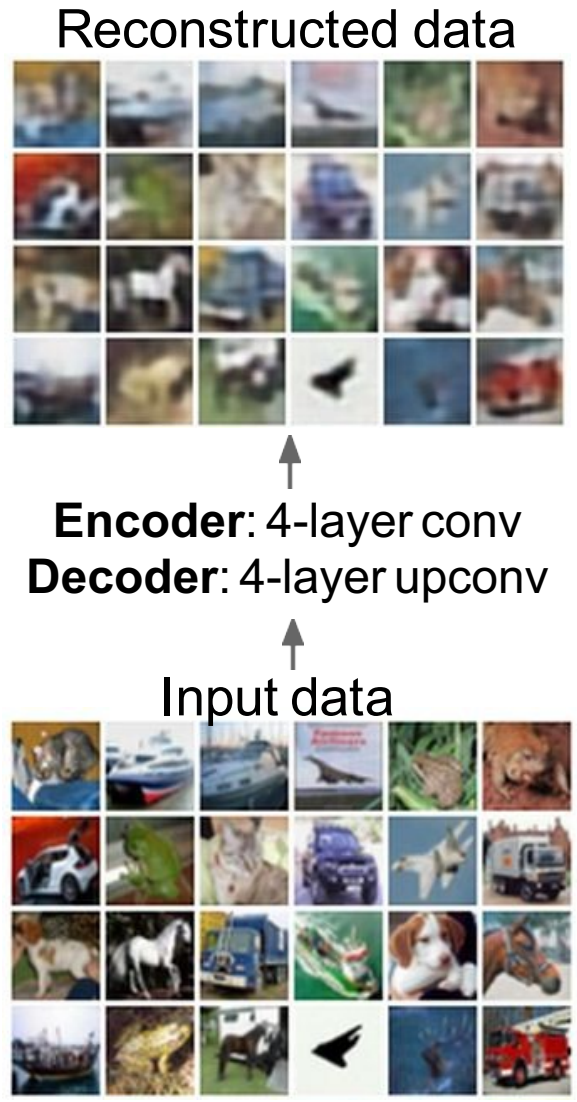
Some background first: Autoencoders

How to learn this feature representation?

Train such that features can be used to reconstruct original data

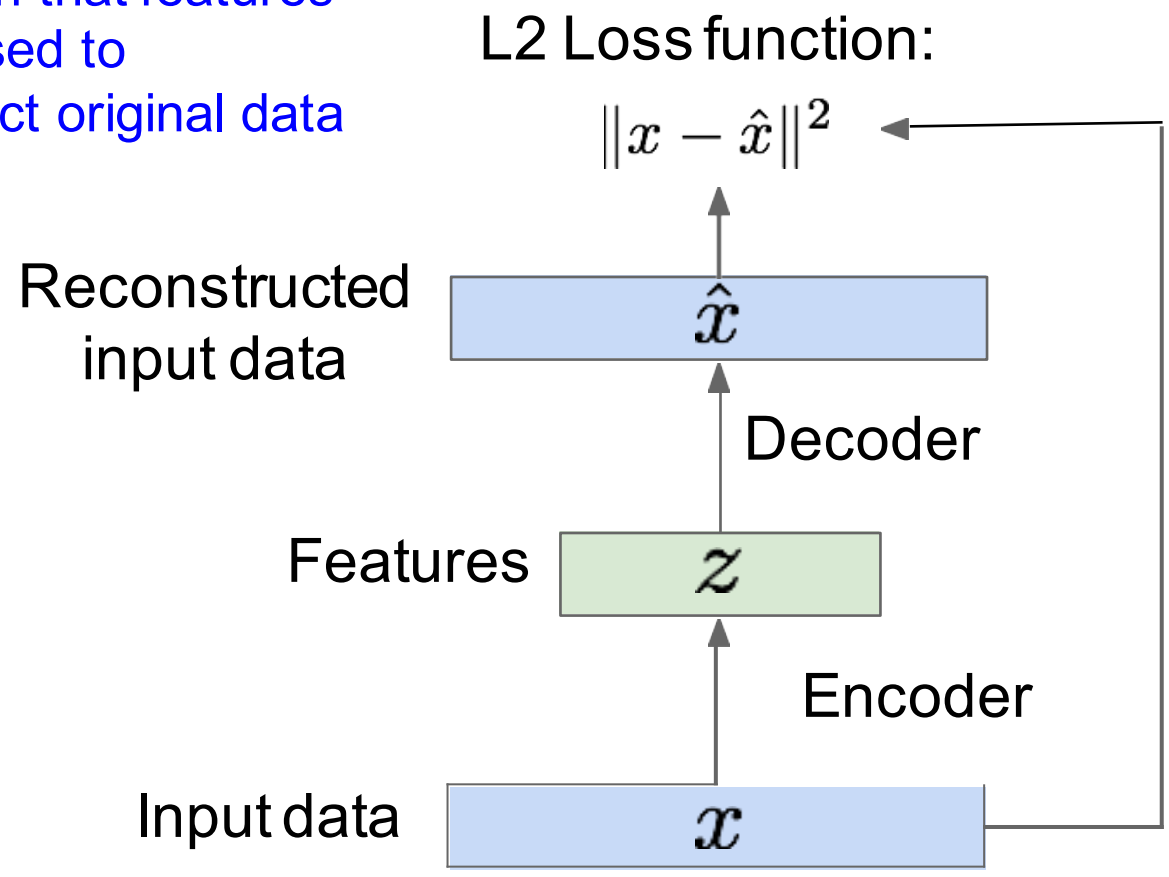
"Autoencoding" - encoding itself

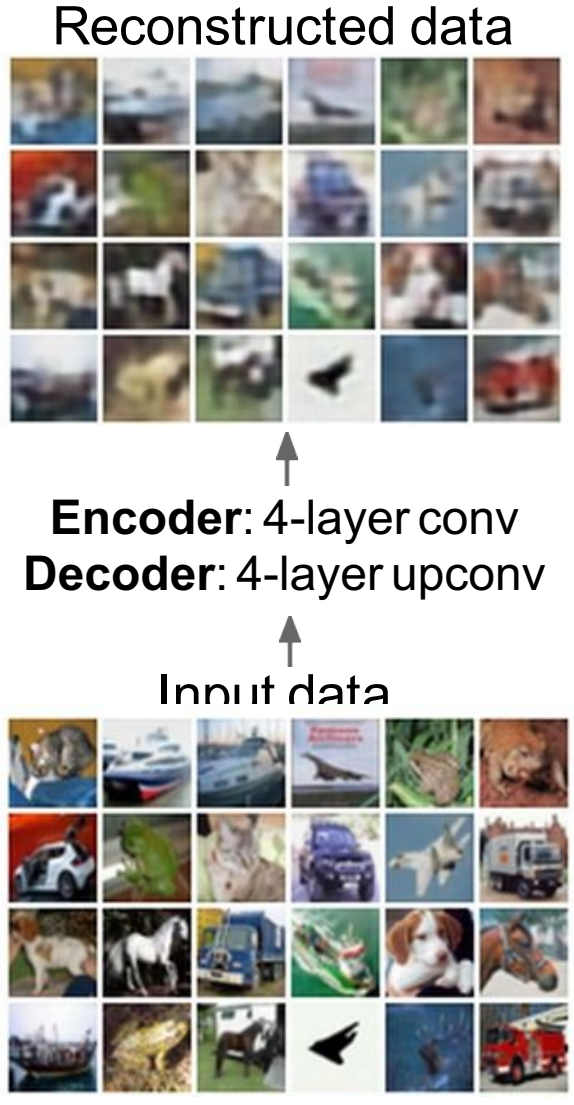




Some background first: Autoencoders

Train such that features can be used to reconstruct original data

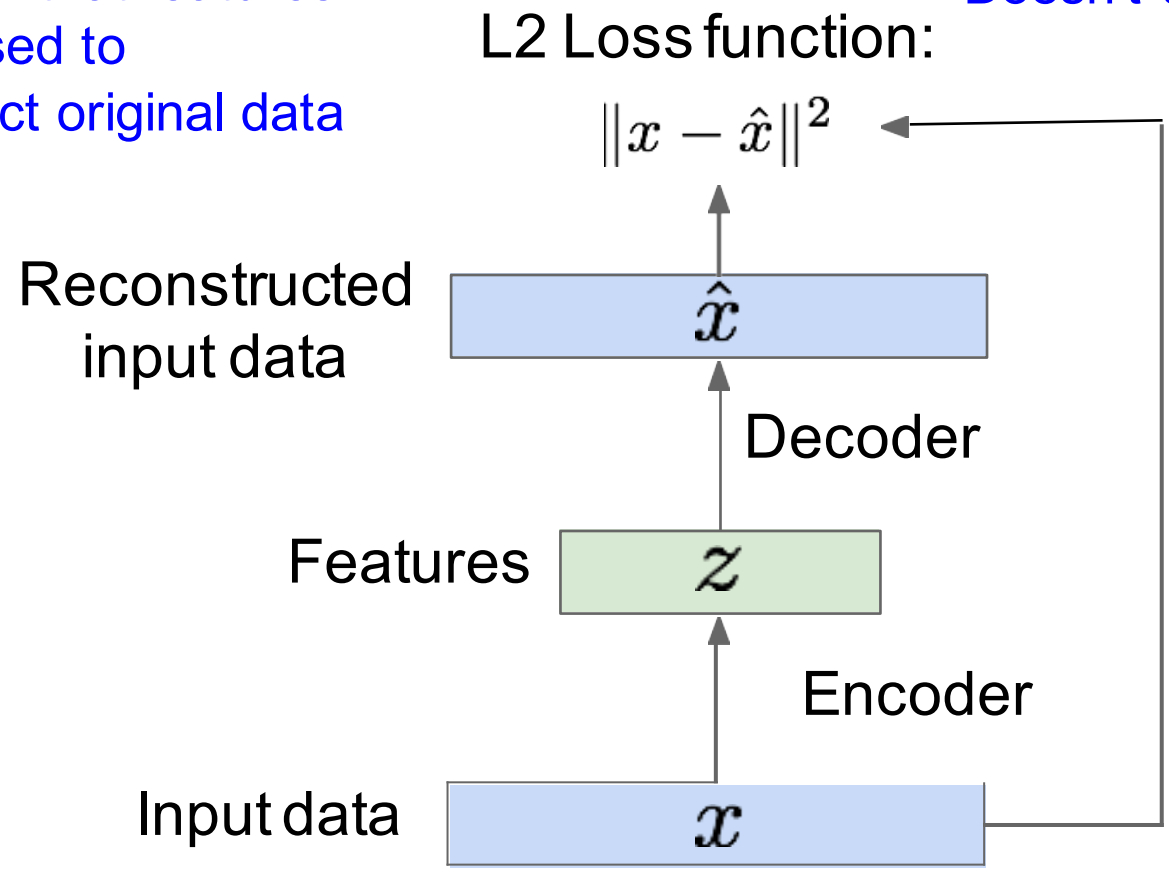


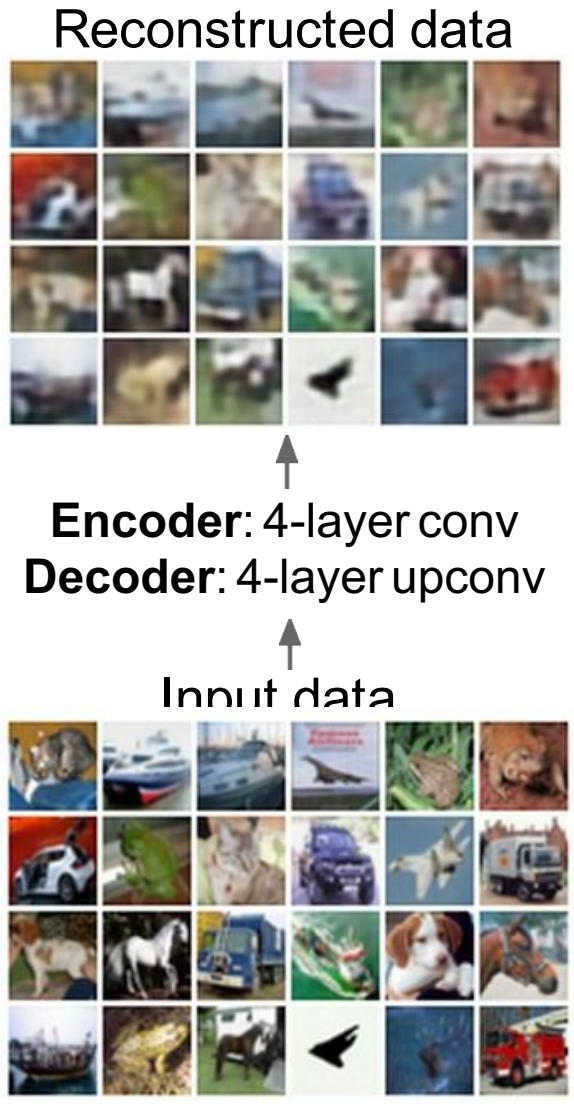


Some background first: Autoencoders

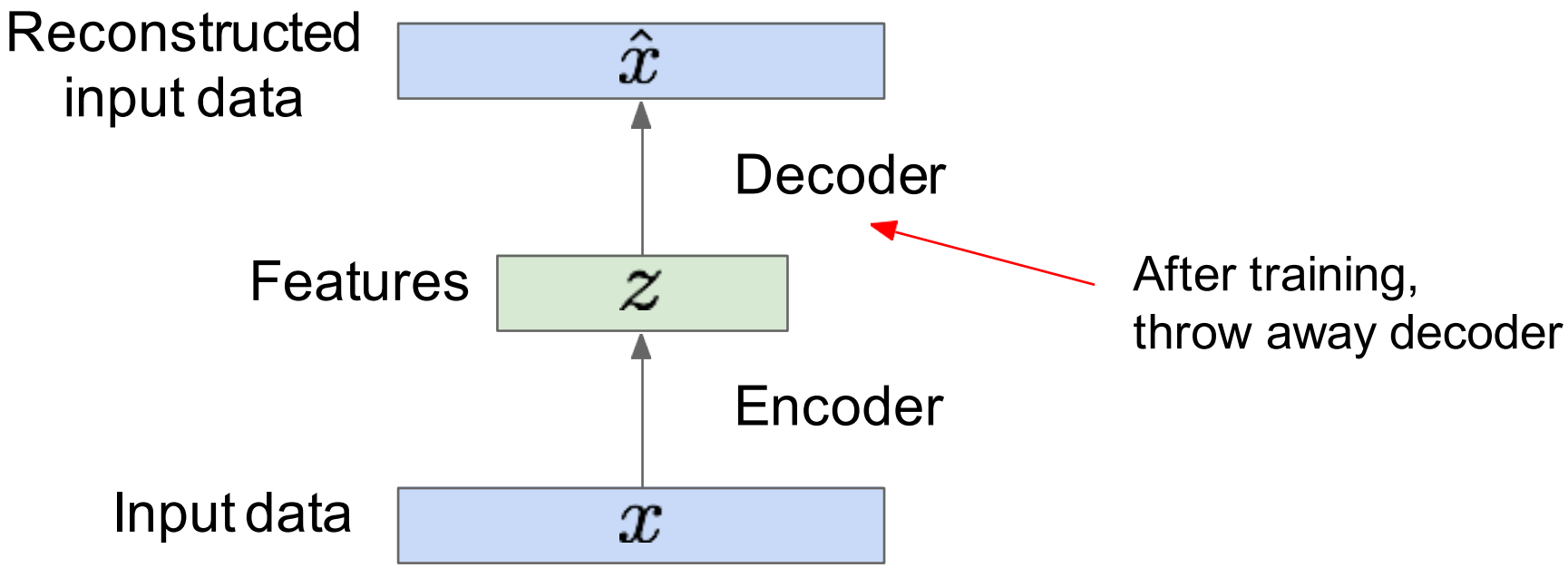
Train such that features can be used to reconstruct original data

Doesn't use labels!

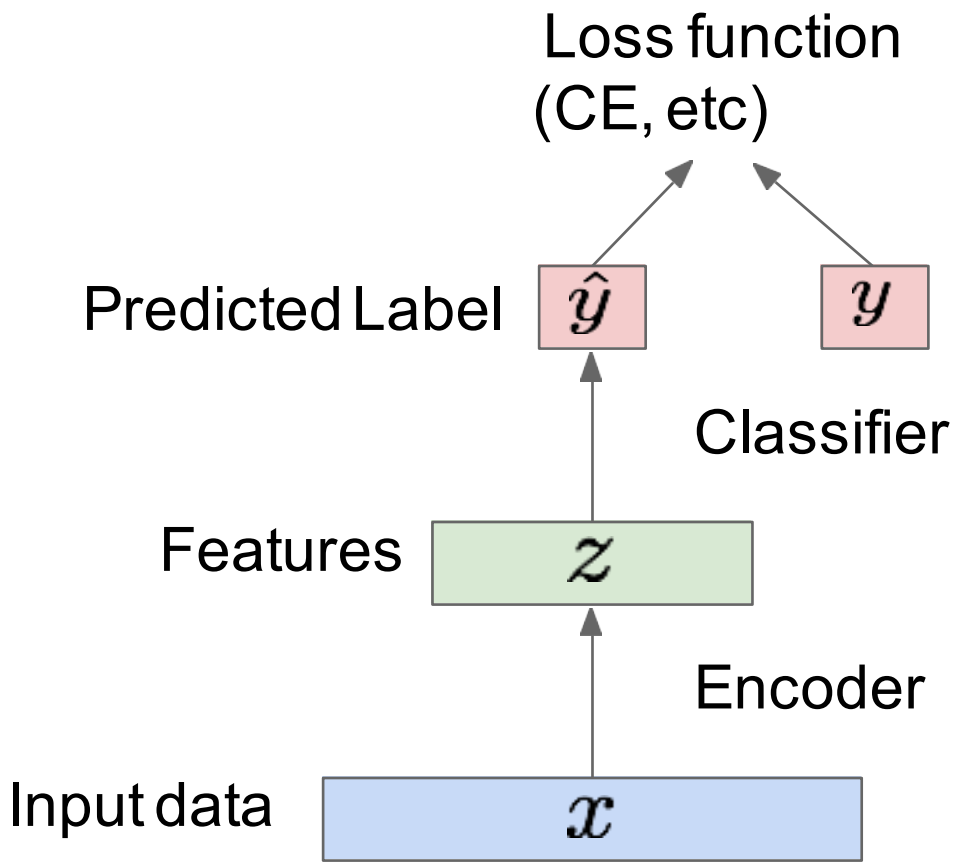




Some background first: Autoencoders



Some background first: Autoencoders



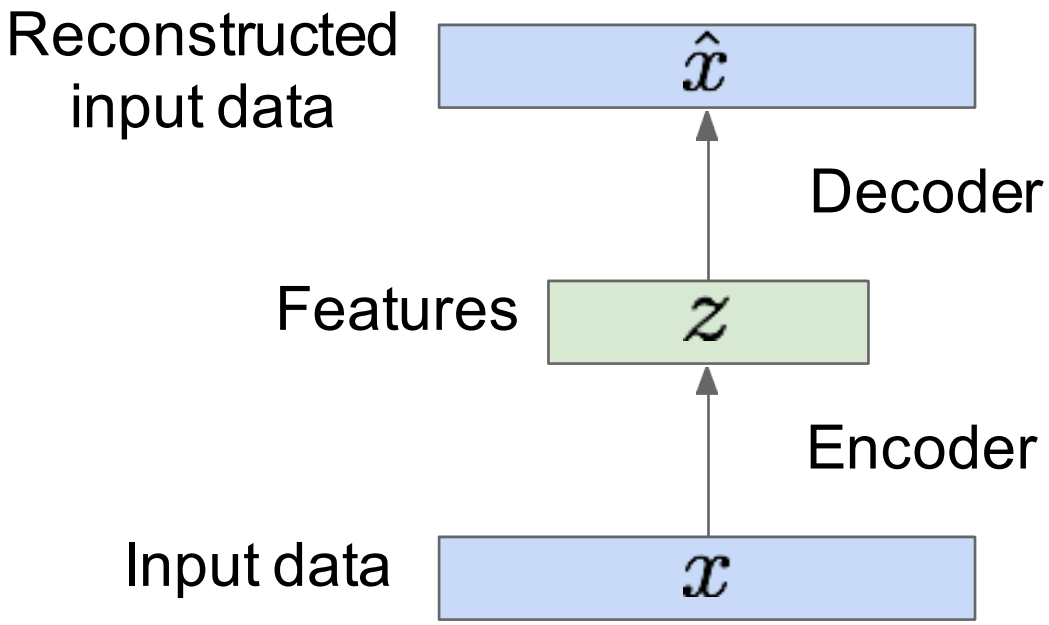
Encoder can be used to initialize a **supervised** model

Fine-tune encoder jointly with classifier

Train for final task (sometimes with small data)

bird plane
dog deer truck

Some background first: Autoencoders



Autoencoders can reconstruct data, and can learn features to initialize a supervised model

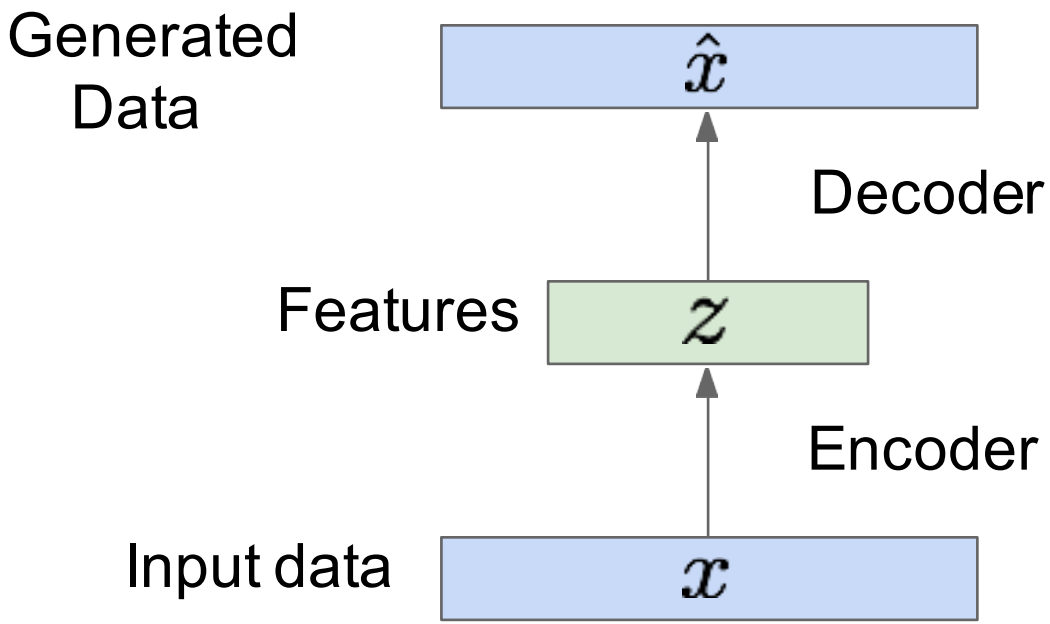
Features capture factors of variation in training data. Can we generate new images from an autoencoder?

Variational Autoencoders

Probabilistic spin on autoencoders - will let us sample from the model to generate data!

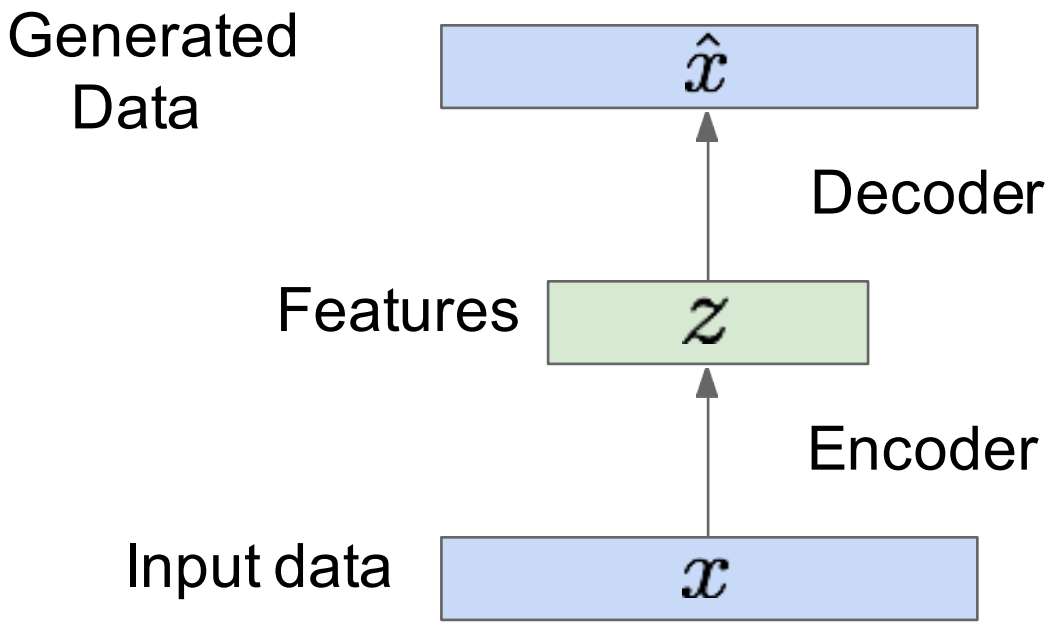
Variational Autoencoders

Probabilistic spin on autoencoders - will let us sample from the model to generate data!



Variational Autoencoders

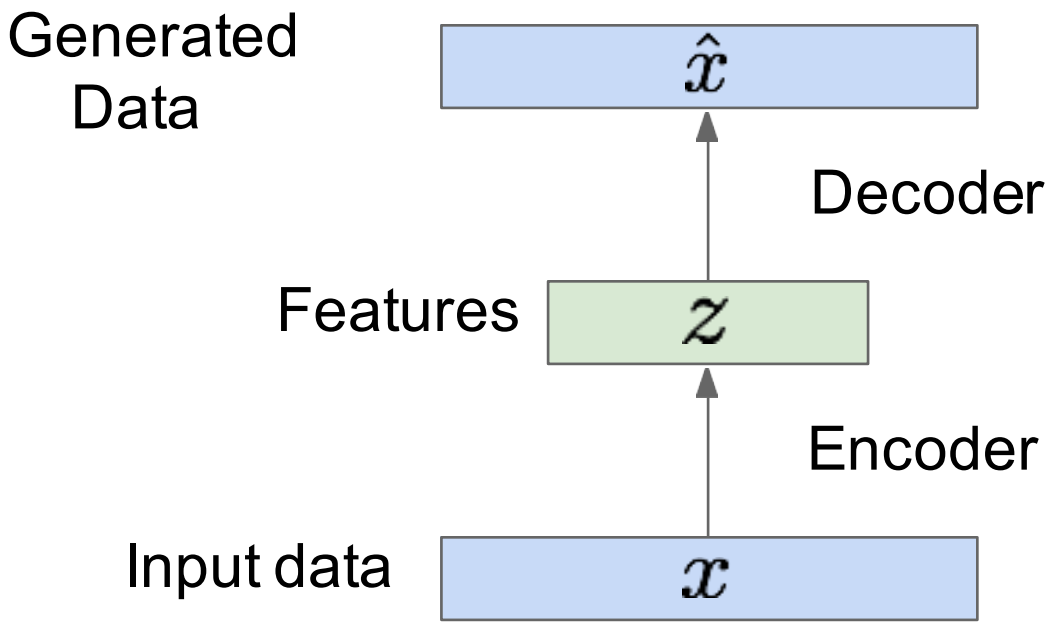
Probabilistic spin on autoencoders - will let us sample from the model to generate data!



Regularize autoencoder to 'fill in holes' by fitting z to a distribution

Variational Autoencoders

Probabilistic spin on autoencoders - will let us sample from the model to generate data!

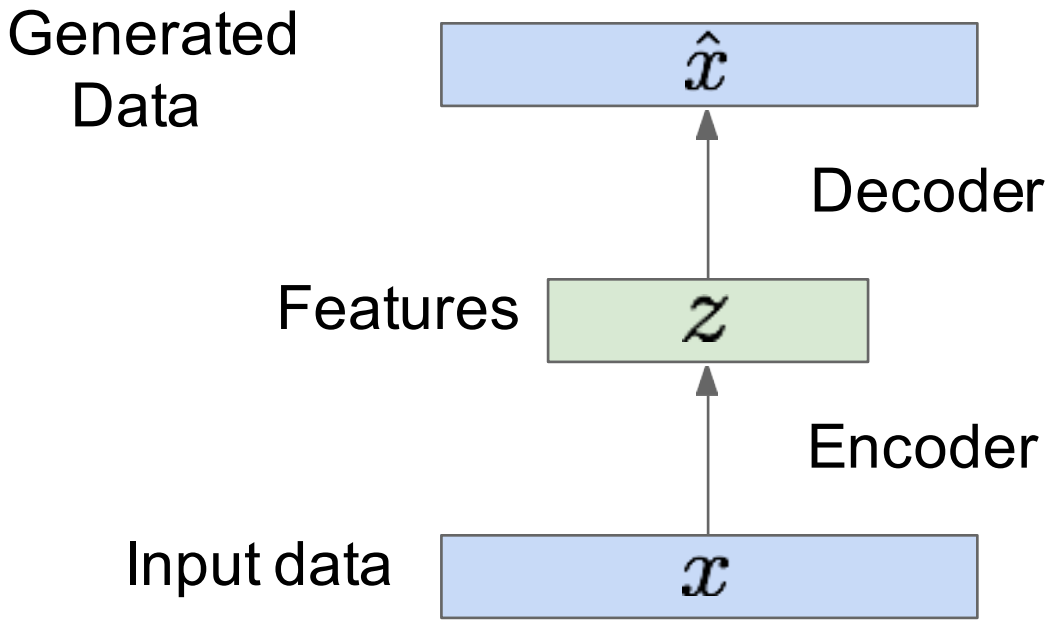


Regularize autoencoder to 'fill in holes' by fitting z to a distribution

Choose prior p(z) to be simple, e.g. Gaussian.
Reasonable for latent attributes, e.g. pose, how much smile.

Variational Autoencoders

Probabilistic spin on autoencoders - will let us sample from the model to generate data!



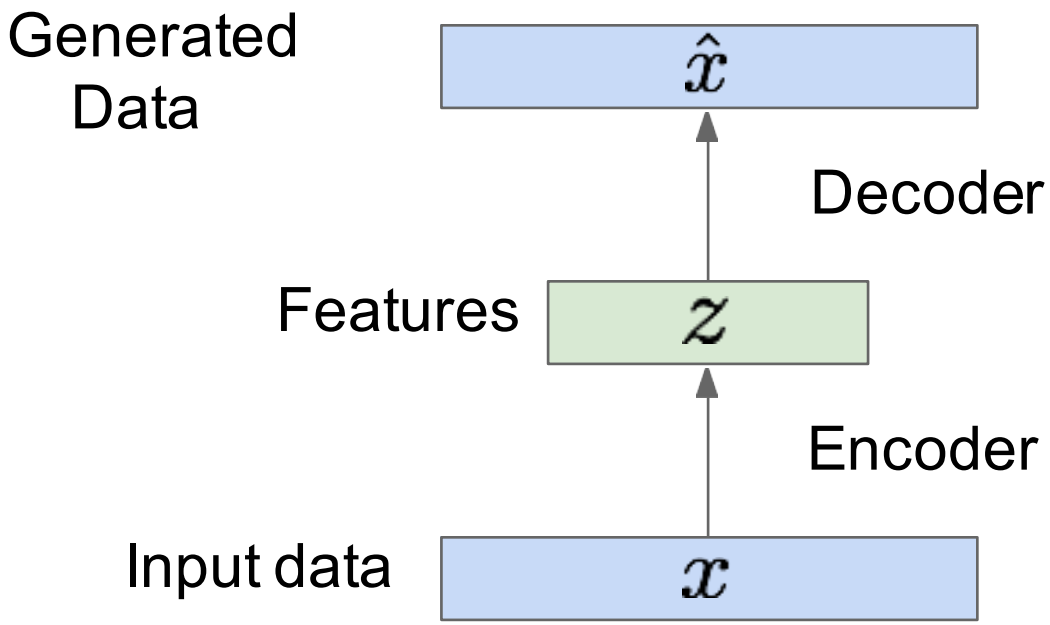
Regularize autoencoder to 'fill in holes' by fitting z to a distribution

Choose prior p(z) to be simple, e.g. Gaussian.
Reasonable for latent attributes, e.g. pose, how much smile.

Encoder and decoder networks also called "recognition"/"inference" and "generation" networks

Variational Autoencoders

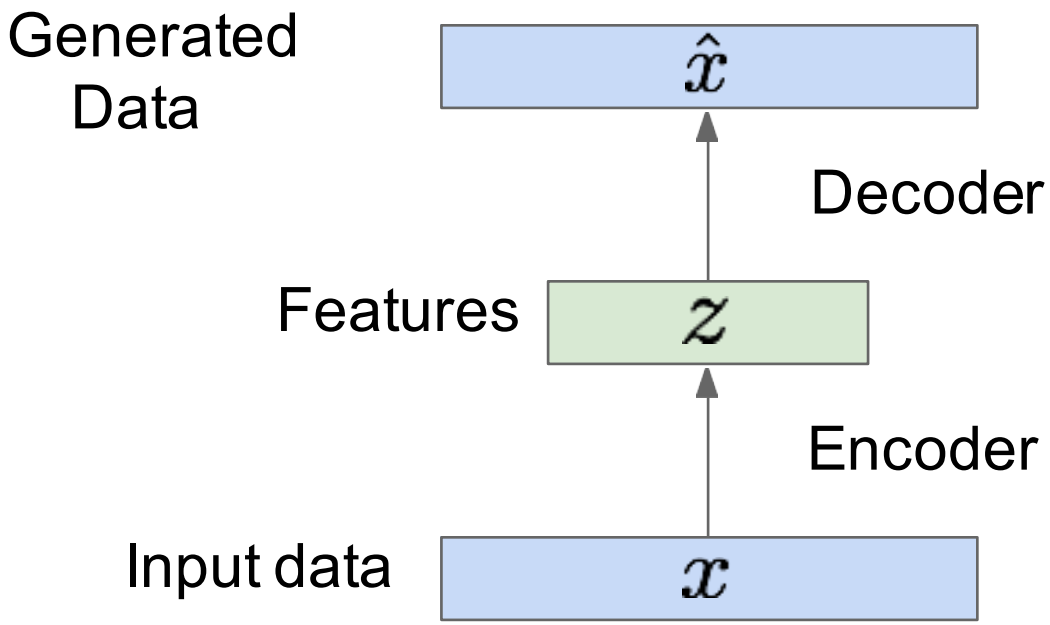
Probabilistic spin on autoencoders - will let us sample from the model to generate data!



Minimize KL-Divergence between z prior and a Gaussian
=> Additional regularizer in the loss function

Variational Autoencoders

Probabilistic spin on autoencoders - will let us sample from the model to generate data!

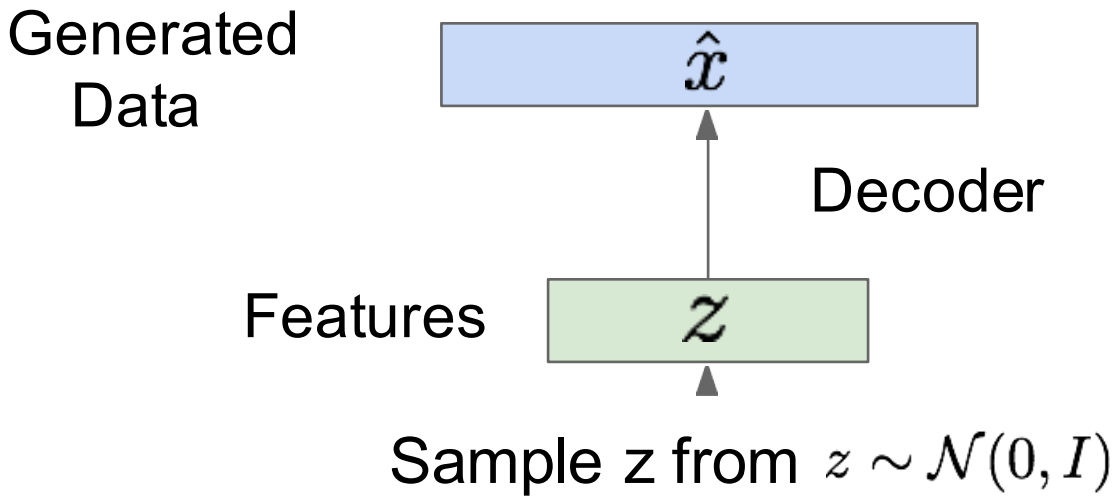


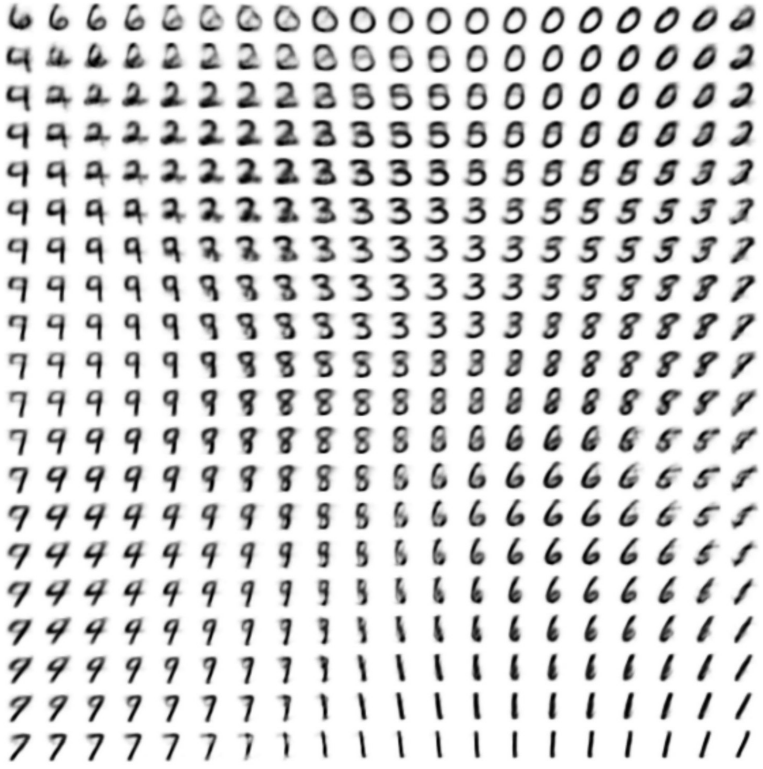
Minimize KL-Divergence between z prior and a Gaussian
=> Additional regularizer in the loss function

KL divergence is a measure of how one probability distribution varies from another

Variational Autoencoders: Generating Data!

Use decoder network. Now sample z from prior!

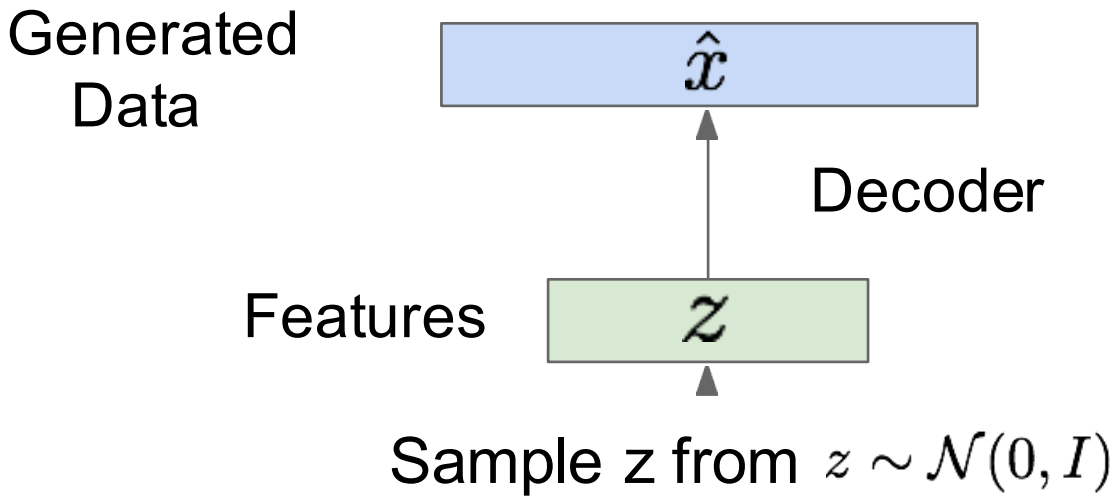




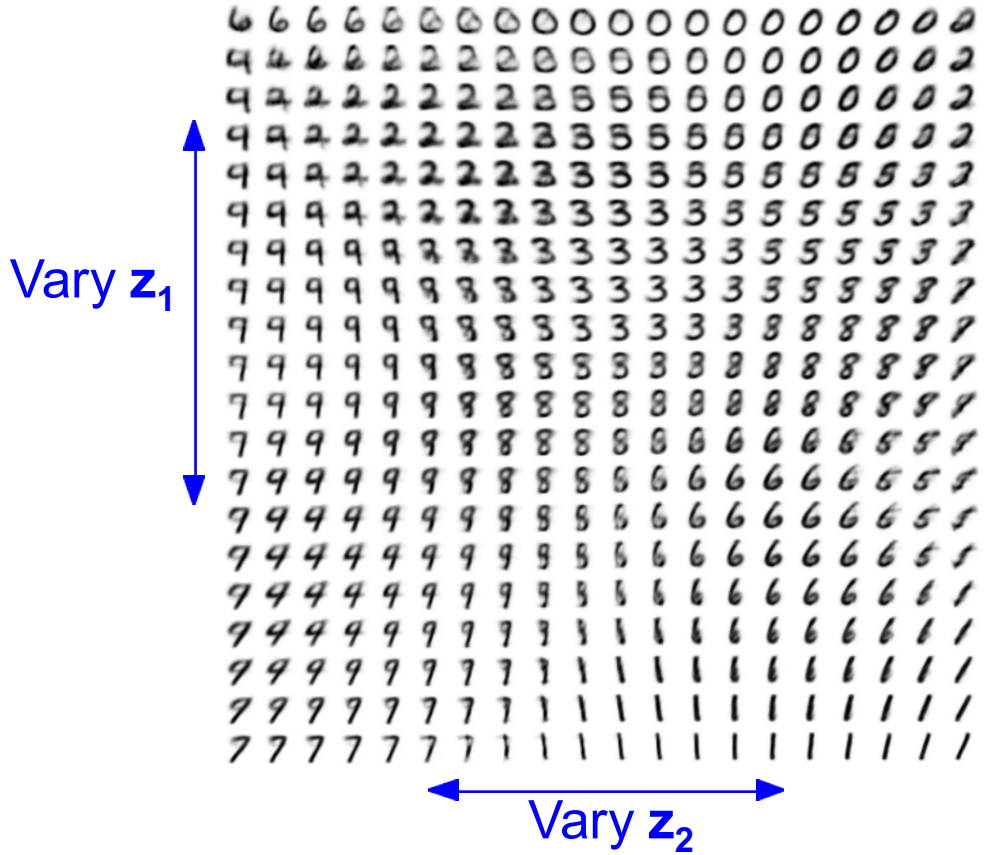
Kingma and Welling, “Auto-Encoding Variational Bayes”, ICLR 2014

Variational Autoencoders: Generating Data!

Use decoder network. Now sample z from prior!



Data manifold for 2-d **z**

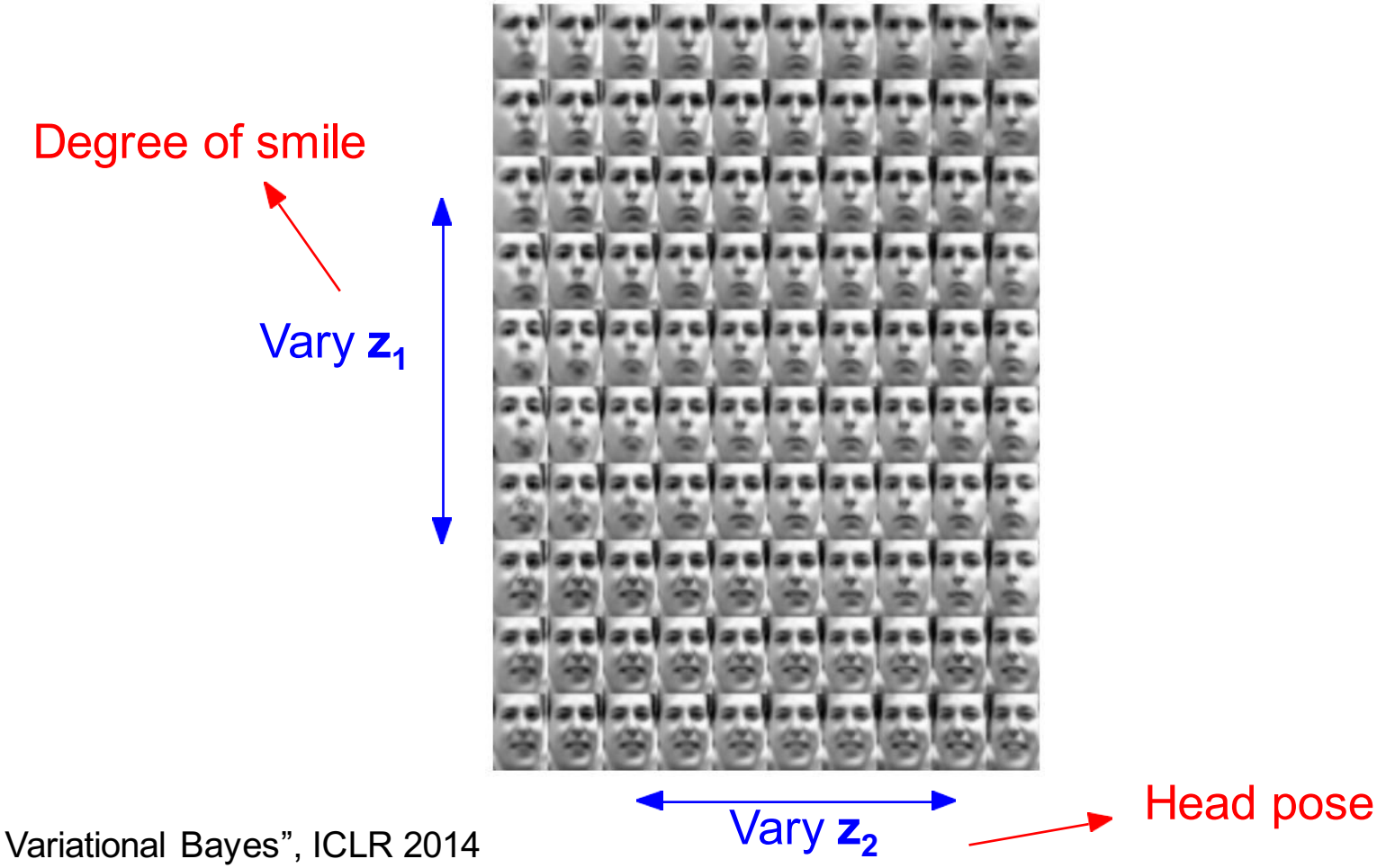


Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

Variational Autoencoders: Generating Data!

Diagonal prior on **z**
=> independent
latent variables

Different
dimensions of **z**
encode
interpretable factors
of variation



Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

Variational Autoencoders: Generating Data!

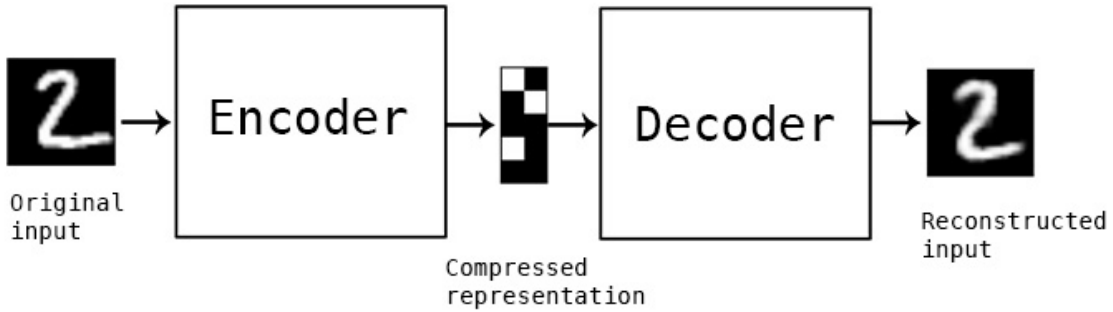
Diagonal prior on **z**
=> independent
latent variables

Different
dimensions of **z**
encode
interpretable factors
of variation

Also good feature representation that can be computed using $q_\phi(z|x)$!

Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

Variational Autoencoders



Instead of Gaussian, can enforce mixture-of-Gaussians or other distributions on the compressed representation

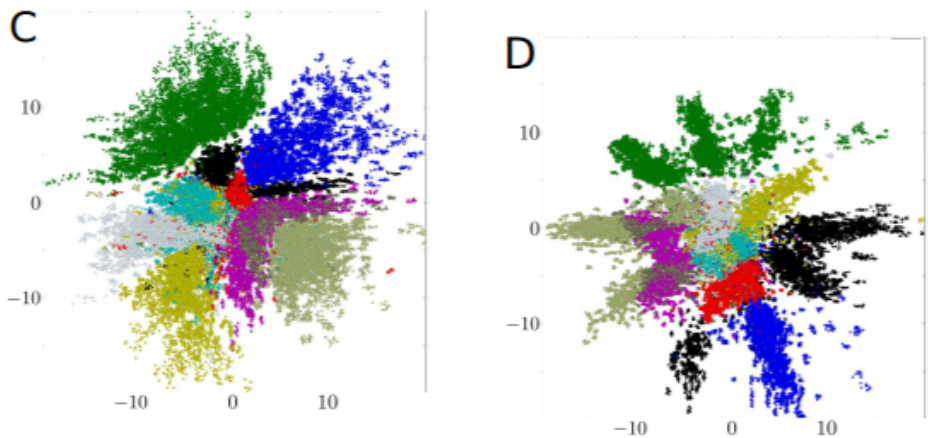
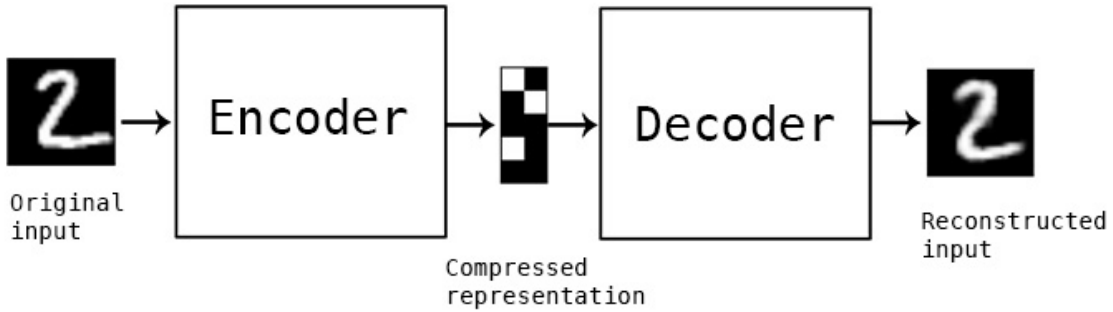


Image taken from Makhzani et. al. Adversarial Autoencoders ICLR 2016

Variational Autoencoders



Instead of Gaussian, can enforce mixture-of-Gaussians or other distributions on the compressed representation

Select based on use case!

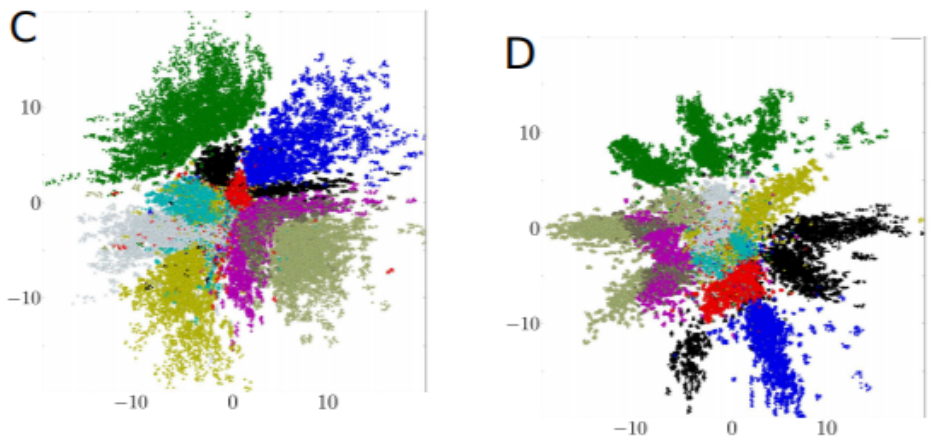
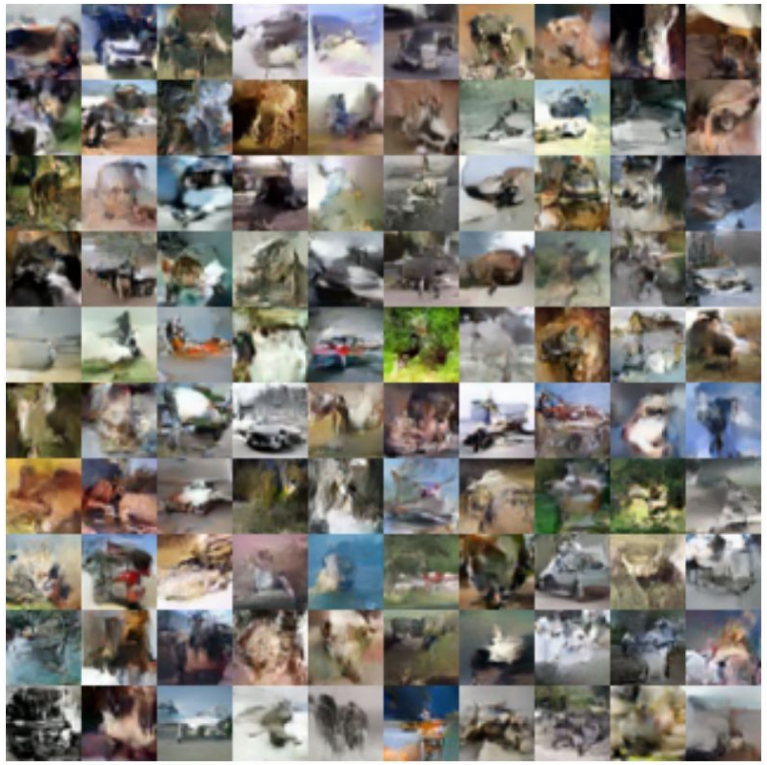


Image taken from Makhzani et. al. Adversarial Autoencoders ICLR 2016

Variational Autoencoders: Generating Data!



32x32 CIFAR-10

Labeled Faces in the Wild

Figures copyright (L) Dirk Kingma et al. 2016; (R) Anders Larsen et al. 2017. Reproduced with permission.

Vanilla GANs

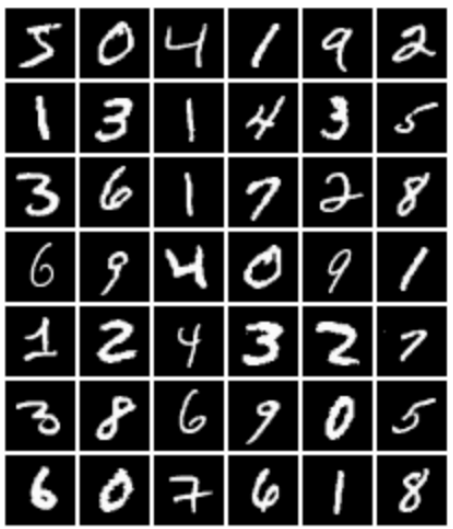
The idea, in a nutshell, is to make two networks - the generator network and the discriminator network compete with other in a minimax game. The generator tries to fool the discriminator creating more and more realistic images, and the discriminator tries not to be fooled by the generator.

Load the MNIST Dataset

```
mnist_x = torch.load('train_32x32.t7', 'ascii').data:double()/255
mnist_x = 2*(mnist_x - .5)
print('Max value: '..mnist_x:max())
print('Min value: '..mnist_x:min())
itorch.image(mnist_x[{{1,42},{},{},{}}])
print('Loaded dataset tensor size:')
print(mnist_x:size())
dataSize = mnist_x:size(1)
dimSize = mnist_x:size(2) * mnist_x:size(3) * mnist_x:size(4)
```

```
Max value: 1

Min value: -1
```



```
Loaded dataset tensor size:
 60000
     1
    32
    32
[torch.LongStorage of size 4]
```

Define Discriminator Network

The Discriminator Network is a binary classifier. It takes in an image of size 32x32 and has two output neurons corresponding to the two classes: *Real* and *Fake*. We again use `nn.LeakyReLU`.

```
-- Discriminator
model_d = nn.Sequential()
model_d:add(nn.Linear(32*32, n_hidden))
model_d:add(nn.LeakyReLU(0.01))
model_d:add(nn.Linear(n_hidden, 2))
```

Define Generator Network

The Generator Network takes in a random vector of size 100 and outputs 32x32 numbers which are reshaped to produce the image. We use `nn.LeakyReLU` as the activation function which are the way to go for GANs as they fecilitate better gradient flow, however, you can experiment with regular `nn.Relu` and see if it matters for our case.

```
n_hidden = 128
-- Generator
model_g = nn.Sequential()
model_g:add(nn.Linear(100, n_hidden))
model_g:add(nn.LeakyReLU(0.01))
model_g:add(nn.Linear(n_hidden, 32*32))
model_g:add(nn.Tanh())
```

Helper unctions to get real and fake (generated) data

```
function get_real_batch(batchData, batchId, shuffle, batchSize)
    for i = 1, batchSize do
        local id = shuffle[ (batchId - 1) * batchSize + i ]
        batchData[i]:copy(mnist_x[{{id},{},{},{}}]:reshape(dimSize))
    end
end

function get_fake_batch(randomData)
    return model_g:forward(randomdata)
end
```

Cost function and other hyperparameters

We will use the cross entropy function `nn.CrossEntropyCriterion()` to train both the Discriminator and the Generator networks. The training of the Discriminator is pretty obvious: the network should output *Fake* when the input is an image produced by the Generator, and *Real*, when it is indeed a real input image from the MNIST dataset.

Training the Generator is slightly less intutive: to train the Generator, we want the Discriminator to output *Real* when the input is an image produced by the Generator. We will see how to do this in torch7 in bit.

```
-- Criterion to train both the Generator and the Discriminator
criterion = nn.CrossEntropyCriterion()
-- Hyperparameters
batchSize = 32
batchSizeHalf = batchSize / 2
batchData = torch.Tensor(batchSize, dimSize)
-- Storage for target
target_d = torch.Tensor(batchSize)
target_g = torch.Tensor(batchSizeHalf)
-- Total batches that can be created from the training set
nBatches = math.floor(dataSize / batchSizeHalf)
-- Optimizer to use when training both the networks. Try using optim.sgd (it doesn't work well!)
optimMethod = optim.adagrad
optimState_g = {learningRate = 0.1}
optimState_d = {learningRate = 0.1}
-- Total no. of epoch we want to train it for
maxEpoch = 20
-- Standard practice: we need to pass these to optim
dParams, dGradParams = model_d:getParameters()
gParams, gGradParams = model_g:getParameters()
```

Main training loop

Like any standard torch7 code to train a network, we run a loop from 1 to maxEpoch. In each epoch, we iterate from 1 to nBatches (computed earlier).

Discriminator

The input to the Discriminator is `batchData`, which contains `batchSize=32` images. The first half (16) are real images and the latter half are generated images. Similarly, the first half of the `target_d` is 1 corresponding to *Real* labels and the latter half is 2 (*Fake* labels). The closure `fevald` is standard - `forward` the Discriminator model and get it's output. Compute the loss using the `criterion`, zero out the gradients, backward through the model and return the loss and the gradients computed by the backward for the optimizer to update the weights with.

Generator

The input to the Generator is `randomOutput`, which contains `batchSizeHalf=16` random vectors of size 100. The Generator produces `output_g`, which are 16 32x32 fake images. We pass these through the Detector Network and then compute the loss with the lables flipped (since we now want the Detector to output the label *Real* for these images). We compute the gradients for this loss and backpropagage through the Discriminator Network. Once we reach the input layer of the Discriminator, we then pass the gradients of of the loss with respect to the input of the Discriminator network to the output layer of the Generator Network and continue to backpropagate through the Generator Network.

The training loop

Real

Fake

Real

Train Discriminator

Train Generator

```
for epochId = 1, maxEpoch do
    shuffle = torch.randperm(dataSize)

    for t = 1, nBatches do
        get_real_batch(batchData, t, shuffle, batchSizeHalf)

        randomData = torch.rand(batchSizeHalf, 100) - 0.5
        randomOutput = model_g:forward(randomData)
        batchData[{{batchSizeHalf+1, batchSize},{}}]:copy(randomOutput)

        target_d:narrow(1, 1, batchSizeHalf):fill(1)
        target_d:narrow(1, batchSizeHalf+1, batchSizeHalf):fill(2)
        target_g:fill(1)


        fevald = function(x)
            output_d = model_d:forward(batchData)
            loss_d = criterion:forward(output_d, target_d)
            d_output_d = criterion:backward(output_d, target_d)
            dGradParams:zero()
            model_d:backward(batchData, d_output_d)
            return loss_d, dGradParams / batchSize
        end

        fevalg = function(x)
            output_g = model_d:forward(randomOutput)
            loss_g = criterion:forward(output_g, target_g)
            d_output_g = criterion:backward(output_g, target_g)
            d_randomOutput = model_d:backward(randomOutput, d_output_g)
            gGradParams:zero()
            model_g:backward(randomData, d_randomOutput)
            return loss_g, gGradParams / batchSizeHalf
        end

        optimMethod(fevalg, gParams, optimState_g)
        optimMethod(fevald, dParams, optimState_d)
    end

    print('Epoch ' .. epochId .. ' out of '..maxEpoch..' done !!!')
    randomData = torch.rand(36, 100) - 0.5
    randImgs = model_g:forward(randomData)
    randImgs = randImgs:reshape(36,1,32,32)
    itorch.image(randImgs)
end
```

Conditional GANs

Conditional Generative Models

- We want to control meaningful attributes of the output of our generative model

- Models based on mapping a lower dimensional latent space to a higher dimensional output can be used for making conditional models
    - Latent space z is lower dimensional as compared to the image
    - Latent vectors can be mapped to distributions similar to our factors of variation

- Generate data conditioned on labels, text, drawing etc.

Examples of Conditional Generation of images

Isola et al 2016 (pix2pix)

Image-to-Image translation: pix2pix

Isola et. al. Image-to-Image Translation with Conditional Adversarial Networks, CVPR 2017

- Transform image from one domain to another: drawing->photograph, maps->satellite imagery, semantic labels -> photo
- Requires paired data for training – for eg drawing and image of the same shoe
- Adversarial discriminator learns to discriminate between real and fake {edge, photo} tuples

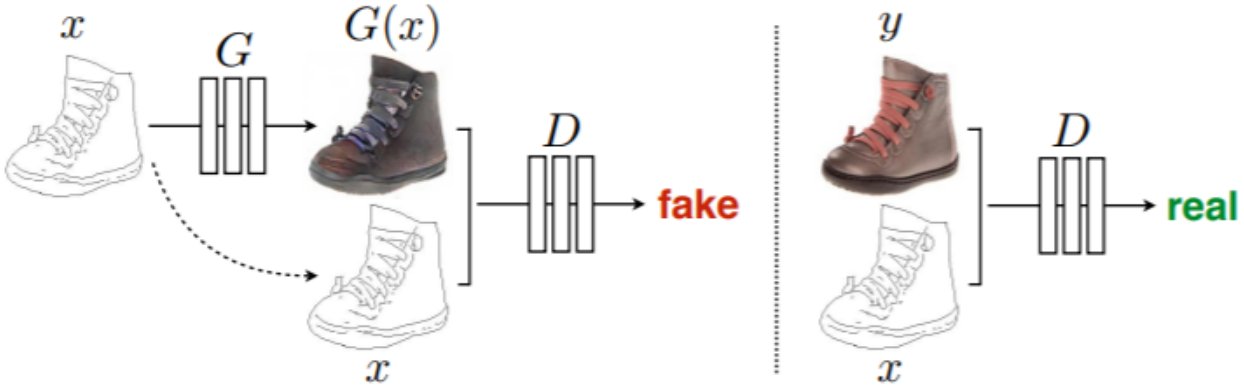


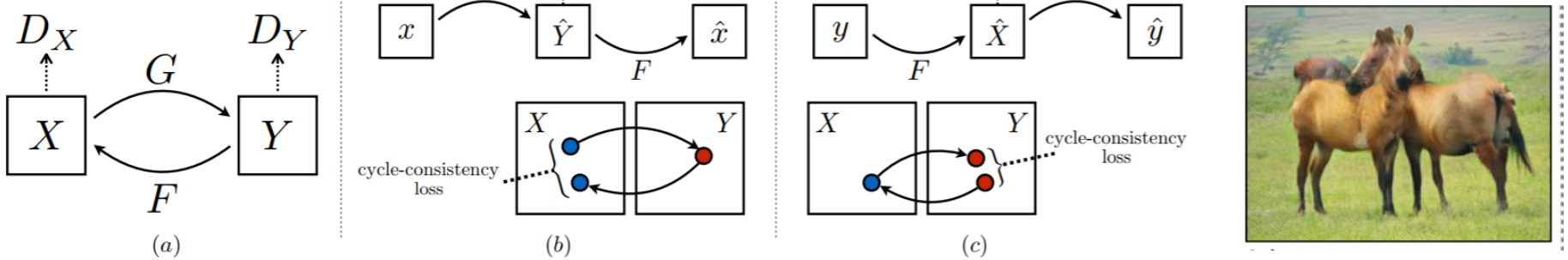
Figure 2: Training a conditional GAN to map edges→photo. The discriminator, D, learns to classify between fake (synthesized by the generator) and real {edge, photo} tuples. The generator, G, learns to fool the discriminator. Unlike an unconditional GAN, both the generator and discriminator observe the input edge map.

Image-to-Image translation: CycleGAN

Zhu et. al. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks, ICCV 2017

- Able to train using **unpaired data – unsupervised learning**
- Learns two functions G and F which transform X->Y and Y->X respectively
- Key idea: cycle consistency – two cycle consistency losses that capture the intuition that if we translate from one domain to another *and back* we should arrive where we started

X: Zebra

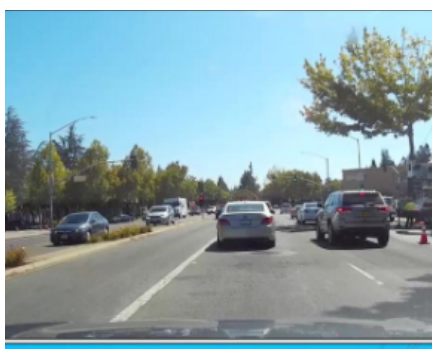


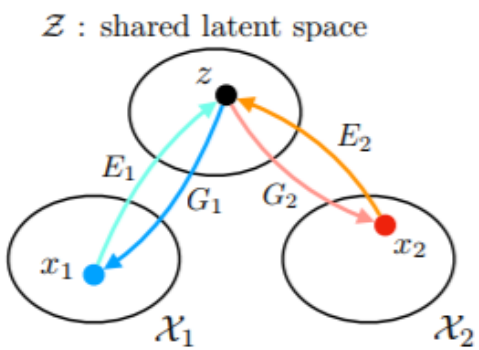
Y: Horse

Image-to-Image translation: UNIT

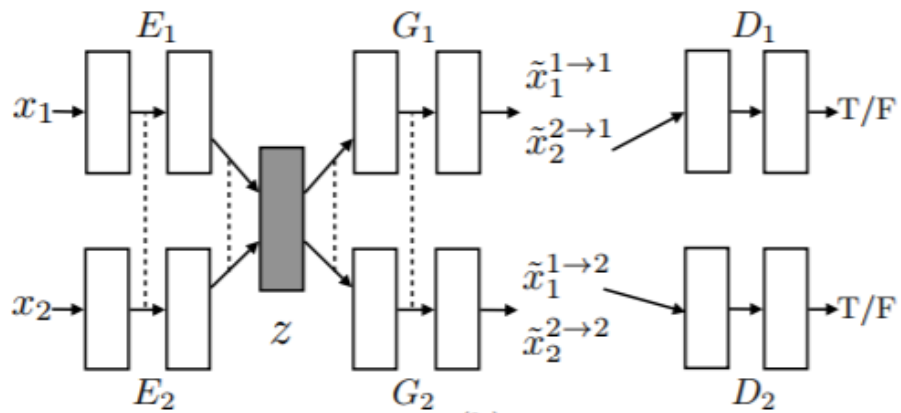
Liu et al. UNIT: UNsupervised Image-to-image Translation Networks, NIPS 2017

- Like CycleGAN, utilizes adversarial and cyclic losses
- Additionally has a strict assumption that a pair of corresponding images *x1* and *x2* from two different domains can be mapped to a shared latent space *Z* - Last few layers of the encoder and the first few layers of the decoder have shared weights
- Four different losses – E1G1 reconstruction, E1G2-E2G1 cycle consistency loss, adversarial loss using D1 and D2, and VAE loss on z

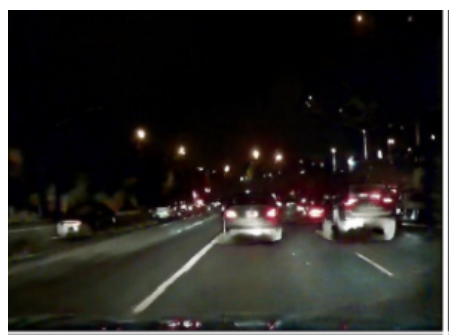




(a)



(b)



Thank you!