CSCE 496/896 Lecture 5: Autoencoders

Stephen Scott

(Adapted from Eleanor Quint and Ian Goodfellow)

sscott@cse.unl.edu

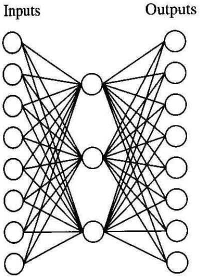
Introduction

- **Autoencoding** is training a network to replicate its input to its output
- Applications:
  - Unlabeled pre-training for semi-supervised learning
  - Learning **embeddings** to support information retrieval
  - Generation of new instances similar to those in the training set
  - Data compression

Outline

- Basic idea
- Stacking
- Types of autoencoders
  - Denoising
  - Sparse
  - Contractive
  - Variational
  - Generative adversarial networks

Basic Idea

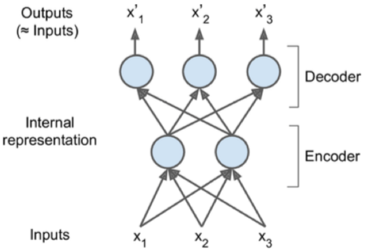


Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

- Sigmoid activation functions, 5000 training epochs, square loss, no regularization
- What's special about the hidden layer outputs?

Basic Idea

- An **autoencoder** is a network trained to learn the **identity function:** output = input



- Subnetwork called **encoder** $f(\cdot)$ maps input to an **embedded representation**
- Subnetwork called **decoder** $g(\cdot)$ maps back to input space

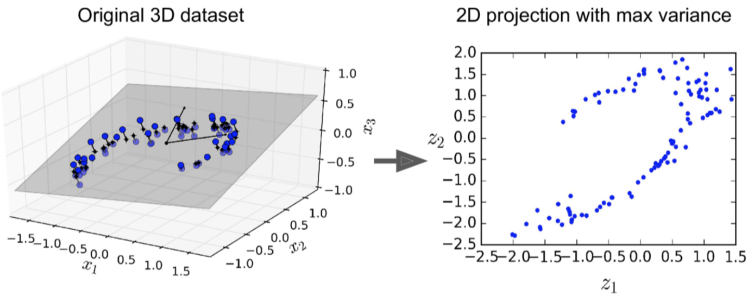
- Can be thought of as **lossy compression** of input
- Need to identify the important attributes of inputs to reproduce faithfully

Basic Idea

- General types of autoencoders based on size of hidden layer
  - **Undercomplete** autoencoders have hidden layer size smaller than input layer size
    - $\Rightarrow$ Dimension of embedded space lower than that of input space
    - $\Rightarrow$ Cannot simply memorize training instances
  - **Overcomplete** autoencoders have much larger hidden layer sizes
    - $\Rightarrow$ Regularize to avoid overfitting, e.g., enforce a **sparsity** constraint

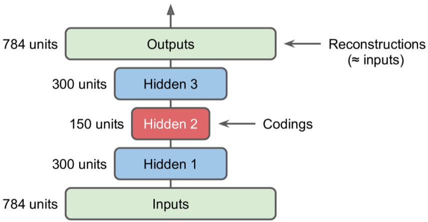
Basic Idea

Example: Principal Component Analysis



- A 3-2-3 autoencoder with linear units and square loss performs **principal component analysis**: Find linear transformation of data to maximize variance

Stacked Autoencoders

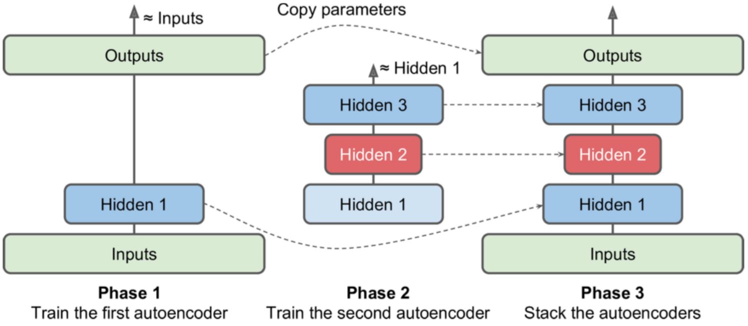


- A **stacked autoencoder** has multiple hidden layers

- Can share parameters to reduce their number by exploiting symmetry: $W_4 = W_1^\top$ and $W_3 = W_2^\top$

```
weights1 = tf.Variable(weights1_init, dtype=tf.float32, name="weights1")
weights2 = tf.Variable(weights2_init, dtype=tf.float32, name="weights2")
weights3 = tf.transpose(weights2, name="weights3")     # shared weights
weights4 = tf.transpose(weights1, name="weights4")     # shared weights
```

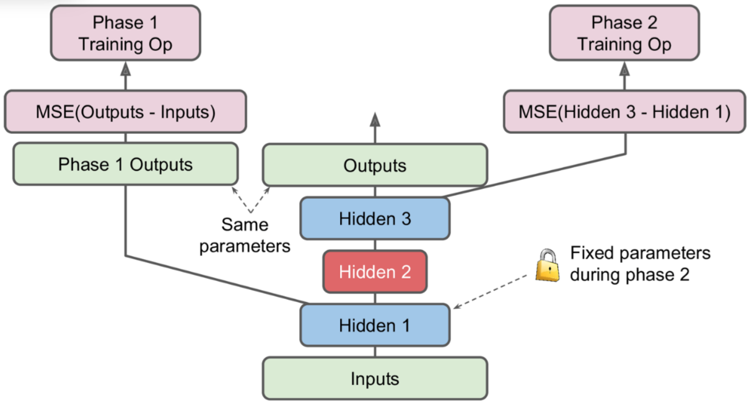
Stacked Autoencoders
Incremental Training



- Can simplify training by starting with single hidden layer H_1
- Then, train a second AE to mimic the output of H_1
- Insert this into first network
- Can build by using H_1's output as training set for Phase 2

Stacked Autoencoders

Incremental Training (Single TF Graph)

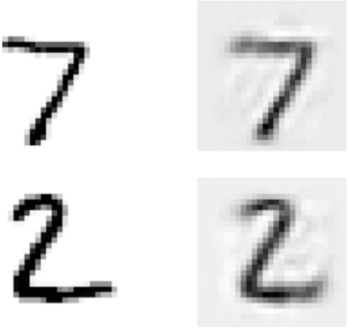


- Previous approach requires multiple TensorFlow graphs
- Can instead train both phases in a single graph: First left side, then right

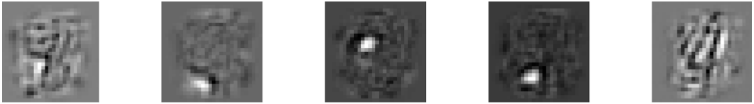
Stacked Autoencoders

Visualization

Input MNIST Digit | Network Output

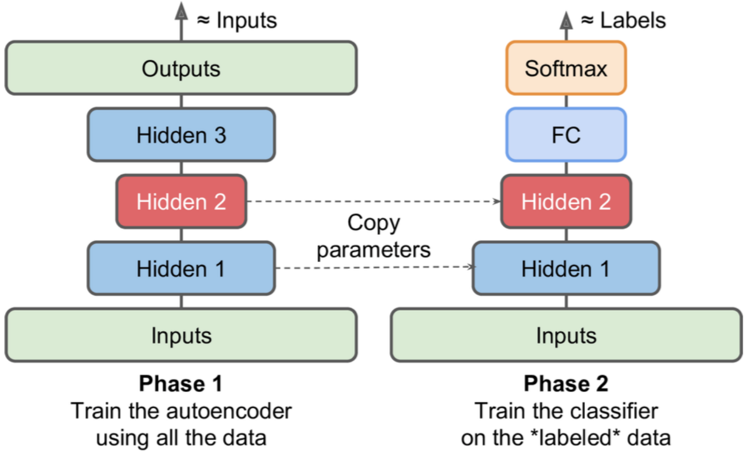


Weights (features selected) for five nodes from H_1:



Stacked Autoencoders

Semi-Supervised Learning

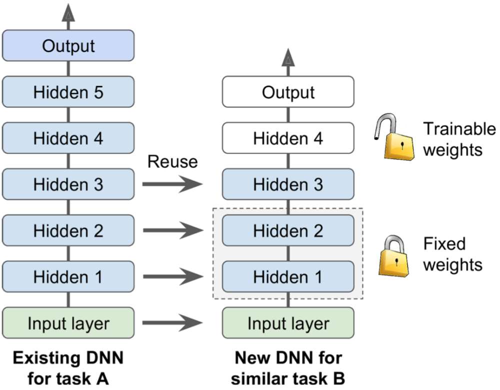


- Can **pre-train** network with unlabeled data

$\Rightarrow$ learn useful features and then train "logic" of dense layer with labeled data

Transfer Learning from Trained Classifier

- Can also transfer from a classifier trained on different task, e.g., transfer a GoogleNet architecture to ultrasound classification



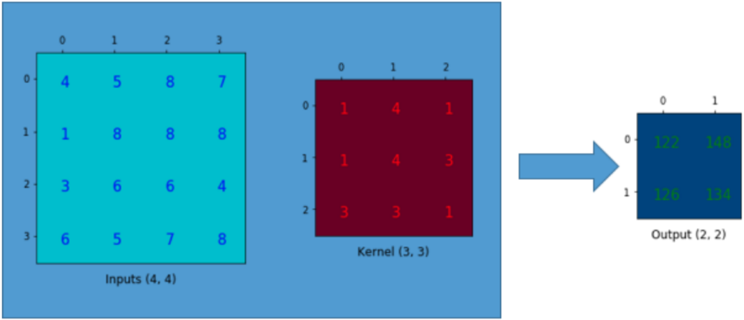
- Often choose existing one from a **model zoo**

Transposed Convolutions

- What if some encoder layers are convolutional? How to upsample to original resolution?
- Can use, e.g., **linear interpolation**, **bilinear interpolation**, etc.
- Or, **transposed convolution**, e.g., `tf.layers.conv2d_transpose`

Transposed Convolutions (2)

Consider this example convolution



Transposed Convolutions (3)

An alternative way of representing the kernel

	0	1	2
0	1	4	1
1	1	4	3
2	3	3	1

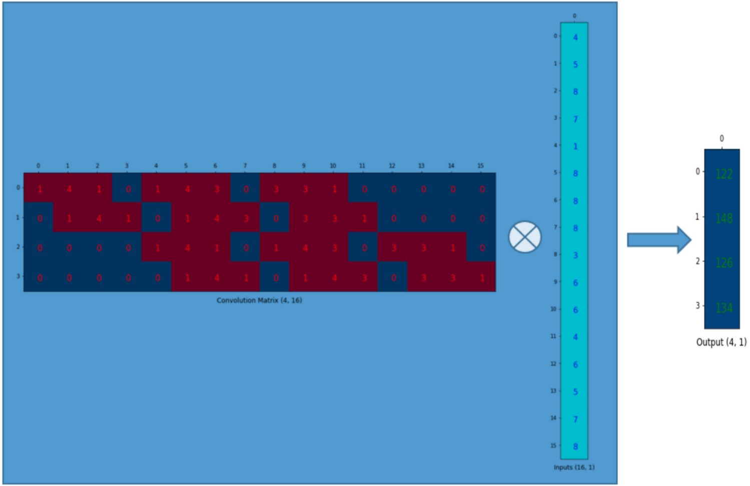
Kernel (3, 3)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	4	1	0	1	4	3	0	3	3	1	0	0	0	0	0
1	0	1	4	1	0	1	4	3	0	3	3	1	0	0	0	0
2	0	0	0	0	1	4	1	0	1	4	3	0	3	3	1	0
3	0	0	0	0	0	1	4	1	0	1	4	3	0	3	3	1

Convolution Matrix (4, 16)

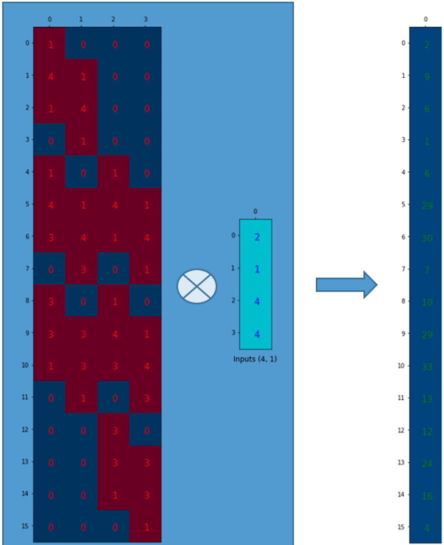
Transposed Convolutions (4)

This representation works with matrix multiplication on flattened input:



Transposed Convolutions (5)

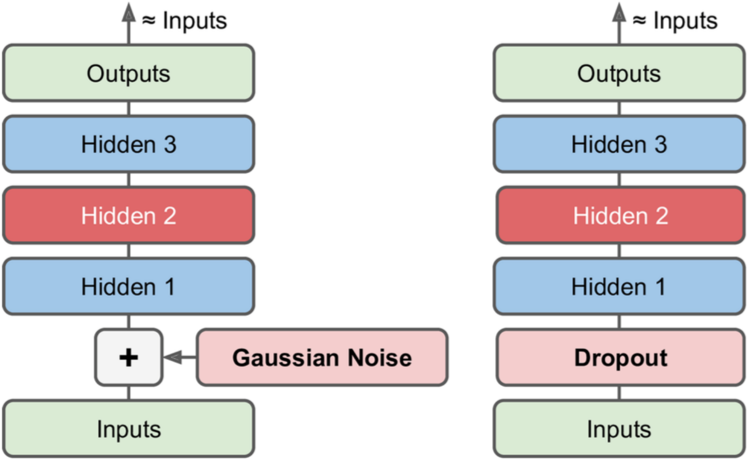
Transpose kernel, multiply by flat 2×2 to get flat 4×4



Denoising Autoencoders
Vincent et al. (2010)

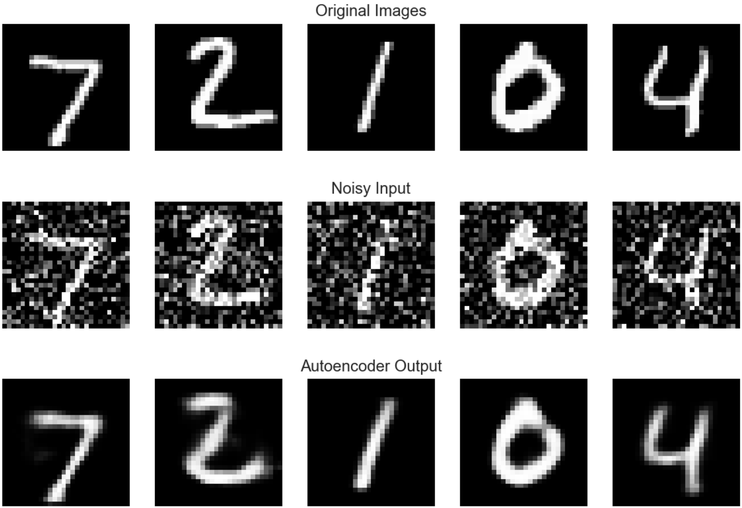
- Can train an autoencoder to learn to **denoise** input by giving input **corrupted** instance $\tilde{\boldsymbol{x}}$ and targeting **uncorrupted** instance $\boldsymbol{x}$
- Example noise models:
  - **Gaussian noise:** $\tilde{\boldsymbol{x}} = \boldsymbol{x} + z$, where $z \sim \mathcal{N}(\mathbf{0}, \sigma^2 I)$
  - **Masking noise:** zero out some fraction ν of components of $\boldsymbol{x}$
  - **Salt-and-pepper noise:** choose some fraction ν of components of $\boldsymbol{x}$ and set each to its min or max value (equally likely)

Denoising Autoencoders



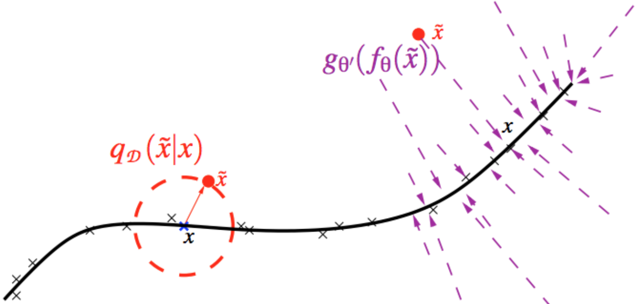
Denoising Autoencoders

Example



Denoising Autoencoders

- How does it work?
- Even though, e.g., MNIST data are in a 784-dimensional space, they lie on a low-dimensional **manifold** that captures their most important features
- Corruption process moves instance x off of manifold
- Encoder f_θ and decoder $g_{\theta'}$ are trained to project $\tilde{x}$ back onto manifold



Sparse Autoencoders

- An overcomplete architecture
- Regularize outputs of hidden layer to enforce **sparsity**:

$$\tilde{\mathcal{J}}(\boldsymbol{x}) = \mathcal{J}(\boldsymbol{x}, g(f(\boldsymbol{x}))) + \alpha\,\Omega(\boldsymbol{h}) \ ,$$

  where $\mathcal{J}$ is loss function, f is encoder, g is decoder, $\boldsymbol{h} = f(\boldsymbol{x})$, and Ω penalizes non-sparsity of $\boldsymbol{h}$
- E.g., can use $\Omega(\boldsymbol{h}) = \sum_i |h_i|$ and ReLU activation to force many zero outputs in hidden layer
- Can also measure average activation of h_i across mini-batch and compare it to user-specified **target sparsity** value p (e.g., 0.1) via square error or **Kullback-Leibler divergence**:

$$p \log \frac{p}{q} + (1-p) \log \frac{1-p}{1-q} \ ,$$

  where q is average activation of h_i over mini-batch

Contractive Autoencoders

- Similar to sparse autoencoder, but use

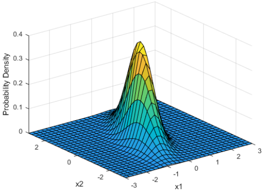
$$\Omega(\boldsymbol{h}) = \sum_{j=1}^{m} \sum_{i=1}^{n} \left(\frac{\partial h_i}{\partial x_j} \right)^2$$

- I.e., penalize large partial derivatives of encoder outputs wrt input values
- This **contracts** the output space by mapping input points in a neighborhood near $\boldsymbol{x}$ to a smaller output neighborhood near $f(\boldsymbol{x})$

  $\Rightarrow$ Resists perturbations of input $\boldsymbol{x}$
- If $\boldsymbol{h}$ has sigmoid activation, encoding near binary and a CE pushes embeddings to corners of a hypercube

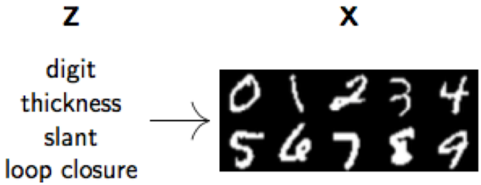
Variational Autoencoders

- VAE is an autoencoder that is also **generative model**
  - $\Rightarrow$ Can generate new instances according to a probability distribution
  - E.g., hidden Markov models, Bayesian networks
  - Contrast with **discriminative models**, which predict classifications
- Encoder f outputs $[\boldsymbol{\mu}, \boldsymbol{\sigma}]^\top$
  - Pair (μ_i, σ_i) parameterizes Gaussian distribution for dimension $i = 1, \ldots, n$
  - Draw $z_i \sim \mathcal{N}(\mu_i, \sigma_i)$
  - Decode this **latent variable** z to get $g(z)$

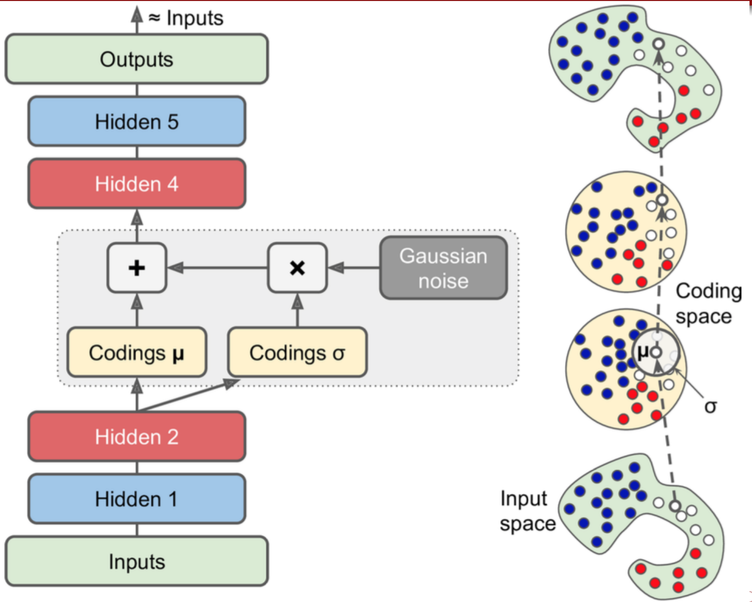


Variational Autoencoders
Latent Variables

- Independence of z dimensions makes it easy to generate instances wrt complex distributions via decoder g
- Latent variables can be thought of as values of attributes describing inputs
  - E.g., for MNIST, latent variables might represent "thickness", "slant", "loop closure"



Variational Autoencoders
Architecture



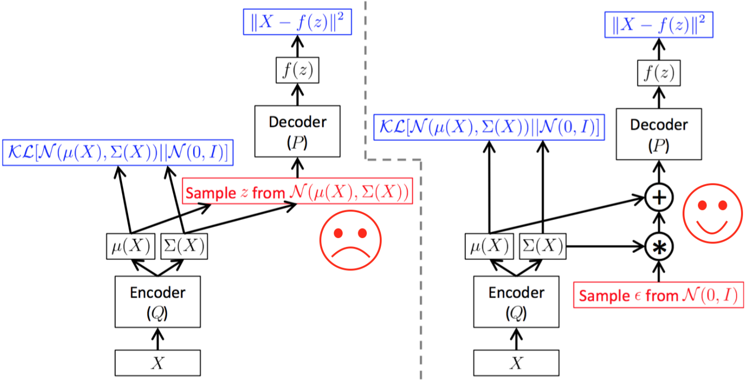
Variational Autoencoders

Optimization

- **Maximum likelihood** (ML) approach for training generative models: find a model ($\boldsymbol{\theta}$) with maximum probability of generating the training set $\mathcal{X}$
- Achieve this by minimizing the sum of:
  - End-to-end AE loss (e.g., square, cross-entropy)
  - Regularizer measuring distance (K-L divergence) from latent distribution $q(z \mid \boldsymbol{x})$ and $\mathcal{N}(\mathbf{0}, I)$ (= standard multivariate Gaussian)
- $\mathcal{N}(\mathbf{0}, I)$ also considered the **prior distribution** over z (= distribution when no $\boldsymbol{x}$ is known)

```
eps = 1e-10
latent_loss = 0.5 * tf.reduce_sum(
        tf.square(hidden3_sigma) + tf.square(hidden3_mean)
        - 1 - tf.log(eps + tf.square(hidden3_sigma)))
```

- Cannot backprop error signal through random samples
- **Reparameterization trick** emulates $z \sim \mathcal{N}(\mu, \sigma)$ with $\epsilon \sim \mathcal{N}(0, 1)$, $z = \epsilon\sigma + \mu$



Variational Autoencoders

Example Generated Images: Random

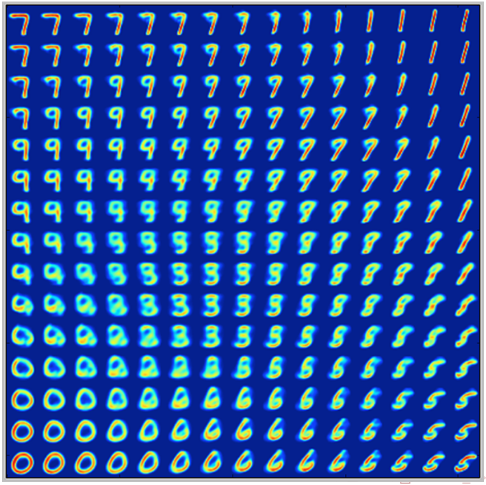
- Draw $z \sim \mathcal{N}(\mathbf{0}, I)$ and display $g(z)$

CSCE 496/896 Lecture 5: Autoencoders

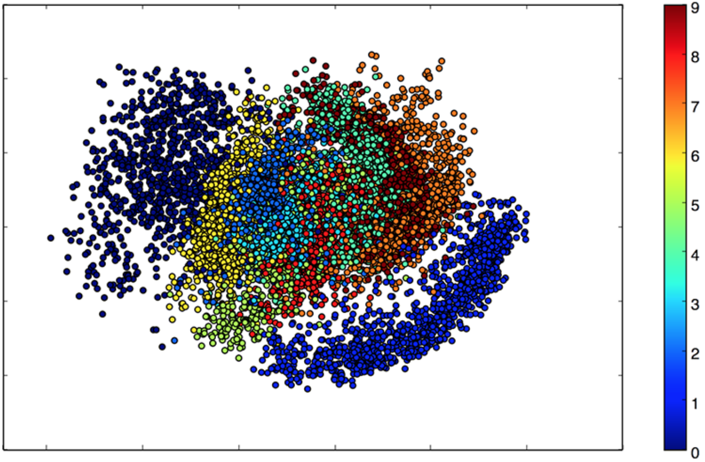
Stephen Scott

Introduction
Basic Idea
Stacked AE
Transposed Convolutions
Denoising AE
Sparse AE
Contractive AE
Variational AE
t-SNE
GAN

- Uniformly sample points in (2-dimensional) z space and decode



- Cluster analysis by digit (2D latent space)



Aside: Visualizing with t-SNE

van der Maaten and Hinton (2008)

- Visualize high-dimensional data, e.g., embedded representations
- Want low-dimensional representation to have similar neighborhoods as high-dimensional one
- Map each high-dimensional $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$ to low-dimensional $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_N$ via matching **pairwise distributions** based on distance
  - $\Rightarrow$ Probability p_{ij} pair $(\boldsymbol{x}_i, \boldsymbol{x}_j)$ chosen similar to probability q_{ij} pair $(\boldsymbol{y}_i, \boldsymbol{y}_j)$ chosen
- Set $p_{ij} = (p_{j|i} + p_{i|j})/(2N)$ where

$$p_{j|i} = \frac{\exp\left(-\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2/(2\sigma_i^2)\right)}{\sum_{k \neq i} \exp\left(-\|\boldsymbol{x}_i - \boldsymbol{x}_k\|^2/(2\sigma_i^2)\right)}$$

  and σ_i chosen to control density of the distribution
- I.e., $p_{j|i}$ is probability of $\boldsymbol{x}_i$ choosing $\boldsymbol{x}_j$ as its neighbor if chosen in proportion of Gaussian density centered at $\boldsymbol{x}_i$

Aside: Visualizing with t-SNE (2)

van der Maaten and Hinton (2008)

- Also, define q via student's t distribution:

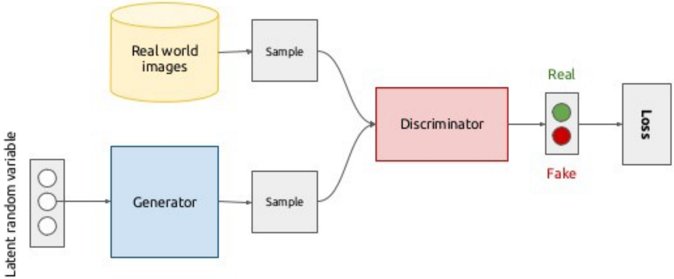
$$q_{ij} = \frac{\left(1 + \|\boldsymbol{y}_i - \boldsymbol{y}_j\|^2\right)^{-1}}{\sum_{k \neq \ell} \left(1 + \|\boldsymbol{y}_k - \boldsymbol{y}_\ell\|^2\right)^{-1}}$$

- Using student's t instead of Gaussian helps address **crowding problem** where distant clusters in $\boldsymbol{x}$ space squeeze together in $\boldsymbol{y}$ space
- Now choose $\boldsymbol{y}$ values to match distributions p and q via **Kullback-Leibler divergence**:

$$\sum_{i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

Generative Adversarial Network

- GANs are also generative models, like VAEs
- Models a game between two players
  - **Generator** creates samples intended to come from training distribution
  - **Discriminator** attempts to discern the "real" (original training) samples from the "fake" (generated) ones
- Discriminator trains as a binary classifier, generator trains to fool the discriminator



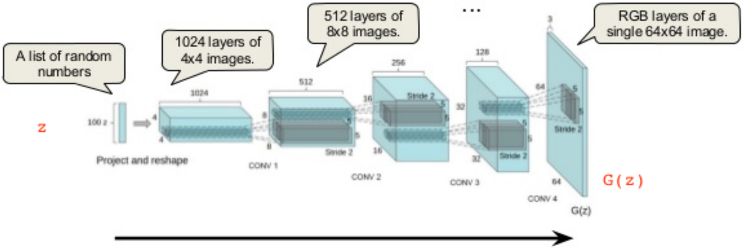
- Let $D(\boldsymbol{x})$ be discriminator parameterized by $\boldsymbol{\theta}^{(D)}$
  - Goal: Find $\boldsymbol{\theta}^{(D)}$ minimizing $J^{(D)}\left(\boldsymbol{\theta}^{(D)}, \boldsymbol{\theta}^{(G)}\right)$
- Let $G(\boldsymbol{z})$ be generator parameterized by $\boldsymbol{\theta}^{(G)}$
  - Goal: Find $\boldsymbol{\theta}^{(G)}$ minimizing $J^{(G)}\left(\boldsymbol{\theta}^{(D)}, \boldsymbol{\theta}^{(G)}\right)$
- A **Nash equilibrium** of this game is $\left(\boldsymbol{\theta}^{(D)}, \boldsymbol{\theta}^{(G)}\right)$ such that each $\boldsymbol{\theta}^{(i)}$, $i \in \{D, G\}$ yields a local minimum of its corresponding J

Generative Adversarial Network
Training

- Each training step:
    - Draw a minibatch of $\boldsymbol{x}$ values from dataset
    - Draw a minibatch of $\boldsymbol{z}$ values from prior (e.g., $\mathcal{N}(\mathbf{0}, I)$)
    - Simultaneously update $\boldsymbol{\theta}^{(G)}$ to reduce $J^{(G)}$ and $\boldsymbol{\theta}^{(D)}$ to reduce $J^{(D)}$, via, e.g., Adam
- For $J^{(D)}$, common to use cross-entropy where label is 1 for real and 0 for fake
- Since generator wants to trick discriminator, can use $J^{(G)} = -J^{(D)}$
    - Others exist that are generally better in practice, e.g., based on ML

Generative Adversarial Network
DCGAN: Radford et al. (2015)

- "Deep, convolution GAN"
- Generator uses **transposed convolutions** (e.g., `tf.layers.conv2d_transpose`) without pooling to upsample images for input to discriminator



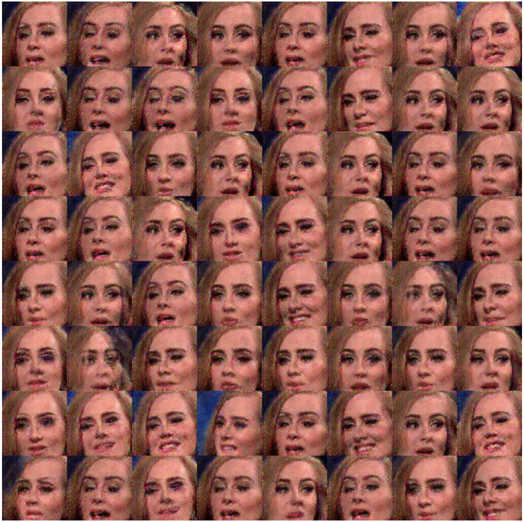
Generative Adversarial Network

DCGAN Generated Images: Bedrooms

Trained from LSUN dataset, sampled z space

Generative Adversarial Network
DCGAN Generated Images: Adele Facial Expressions

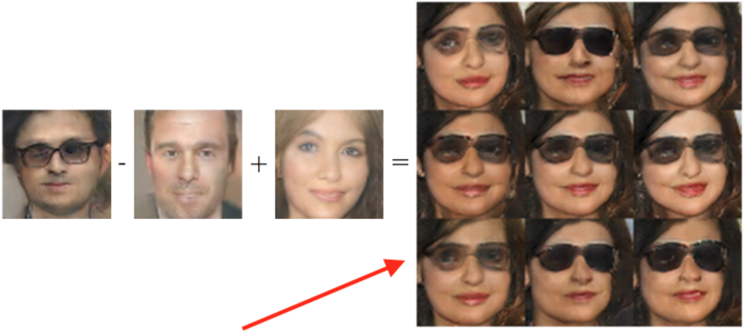
Trained from frame grabs of interview, sampled z space



Generative Adversarial Network

DCGAN Generated Images: Latent Space Arithmetic

Performed semantic arithmetic in z space!



(Non-center images have noise added in z space; center is noise-free)