

# OEAW AI SUMMER SCHOOL

## DEEP LEARNING IV

### (Variational) Autoencoders, GANs

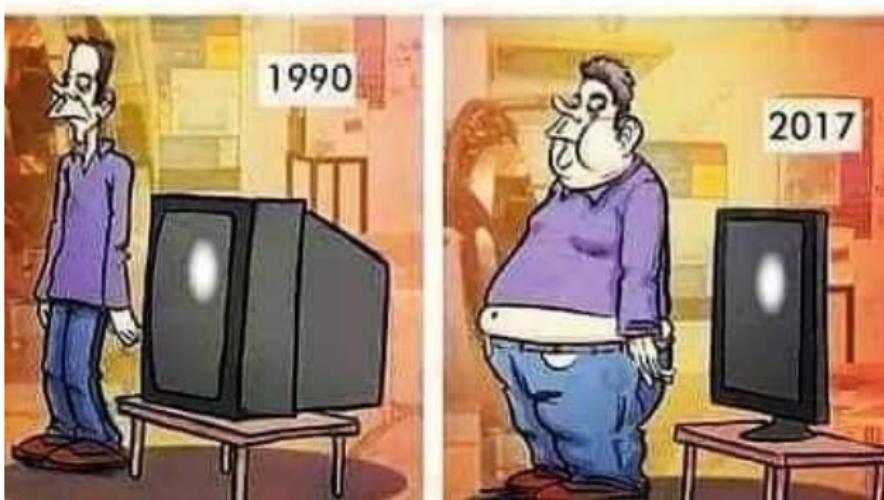


Johannes Brandstetter, Michael Widrich

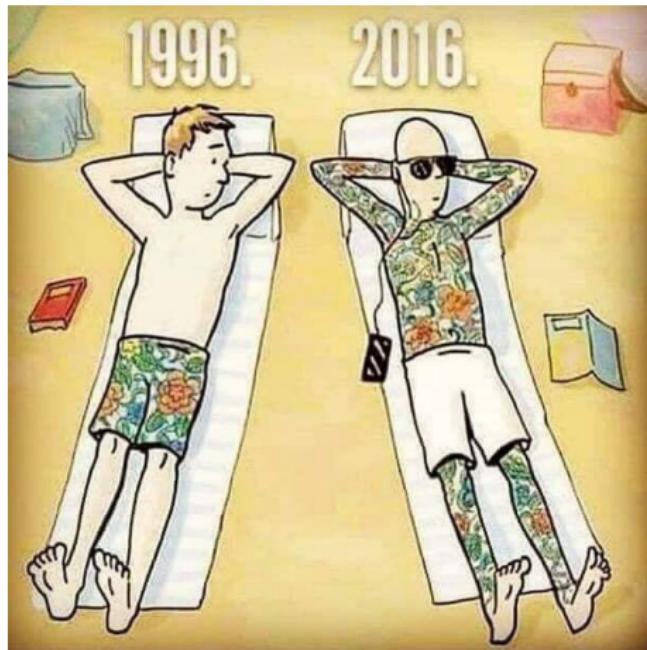
Institute for Machine Learning

## Then and now

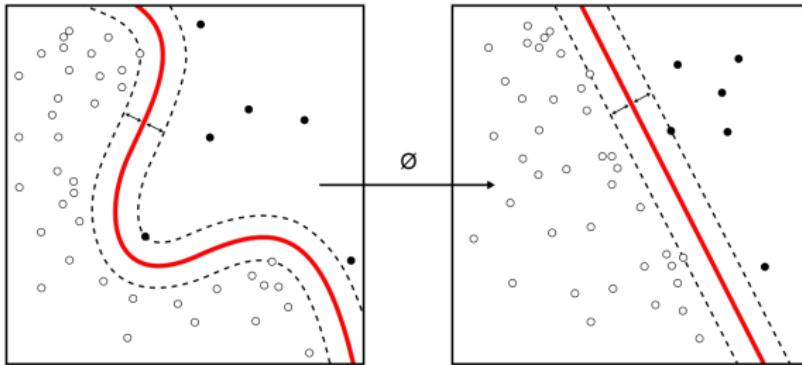
1990 - 2017



## Then and now



# Then and now



# Timeline of Machine Learning

- First serious neural network efforts started in the 1980s.
- 1984 Leo Breiman: **Classification and regression trees**
- 1992-1995: profound mathematical foundation of **Support Vector Machines**:
  - kernel trick
  - soft margin classifier
- 2006-2008: **Restricted Boltzman Machines** started the Deep Learning hype
- 2012: **AlexNet**
- 2013: Speech recognition with **LSTMs**
- 2013 onwards:
  - **Tricks of the Trade**
  - **VAE**
  - **GAN**
  - ...

# We are Deep Learning

- (Deep) Machine Learning has the potential to revolutionize your field of science ...
- ... BUT (Deep) Machine Learning is no black box magic which always works:
  - Not every problem is a ML problem:
    - Sometime “simple” statistics is all you need
  - Not every ML problem is a Deep Learning problem:
    - You have to know your data
    - You have to know the statistics
    - You have to know what algorithm to use
    - You have to know how to control the beast  
(especially in Deep Learning)
- Doesn't matter what you are doing, you can do really cool things with (Deep) Learning!

# Lecture IV: (Variational) Autoencoders, GANs



# Autoencoders

## Hidden representations

- Each hidden layer of the neural net calculates a new representation of the data.
- Each hidden layer sees the data only via the representation of the layer below.

# Hidden representations

- Each hidden layer of the neural net calculates a new representation of the data.
- Each hidden layer sees the data only via the representation of the layer below.
- Imagine a neural net for classification with 1 hidden layer:
  - Neural network gives better performance than logistic regression ...
  - ... but the **output layer is actually a logistic regression** that uses the representation of the hidden layer.
  - ⇒ **Learning a hidden representation normally helps.**

# Autoencoders

**Goal:** Can we use this “representation learning” property of neural nets without needing a label to predict (i.e. in an unsupervised fashion)?

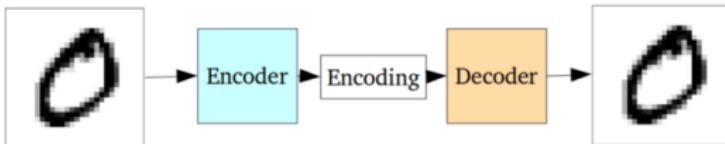
**Idea:** Let the network predict the input data again.  
(The input data is re-used as label.)

**Result:** The hidden layer will learn a representation of the input that encodes all its major features.  
(So it's easy to reconstruct the input again.)

**Use:** This can be used as a general feature construction method, but e.g. also to compress the input data.

This is called an **Auto-Encoder (AE)** or **Auto-Associator**.

## General remarks



Most often, autoencoders have only one hidden layer:

$$h = f_1(\mathbf{W1} \cdot x) \quad \text{"encoding" step}$$

$$o = f_2(\mathbf{W2} \cdot h) \quad \text{"decoding" step}$$

- For binary inputs (and target outputs),  $f_2$  is often a sigmoid.
- For unconstrained or approximately Gaussian inputs,  $f_2$  is often the identity function.
- Sometimes tied weights are used, i.e.  $\mathbf{W2} = \mathbf{W1}^T$  (reduces the number of parameters and often doesn't hurt performance).

## Linear autoencoder

In the simplest case,  $f_1$  is the identity as well:

$$\mathbf{h} = \mathbf{W1} \cdot \mathbf{x}$$

$$\mathbf{o} = \mathbf{W2} \cdot \mathbf{h}$$

- Essentially  $\mathbf{o} = \mathbf{W2} \cdot \mathbf{W1} \cdot \mathbf{x} = \hat{\mathbf{W}} \cdot \mathbf{x}$
- Only interesting if we have less hidden units than inputs, otherwise trivial solution  $\mathbf{W1} = \mathbf{W2} = \mathbf{I}$ .
- Very similar to PCA:
  - Linear AE learns representation that lives in the same linear subspace as the PCA solution.

(Proof: "Neural networks and principal component analysis", Baldi & Hornik, 1989)

# Pytorch implementation non-linear AE

```
class autoencoder(nn.Module):
    def __init__(self, ae_input_dim):
        super(autoencoder, self).__init__()
        # encoder
        self.encoder_linear1 = nn.Linear(ae_input_dim, 256)
        self.encoder_linear2 = nn.Linear(256, 10)
        # decoder
        self.decoder_linear1 = nn.Linear(10, 256)
        self.decoder_linear2 = nn.Linear(256, ae_input_dim)

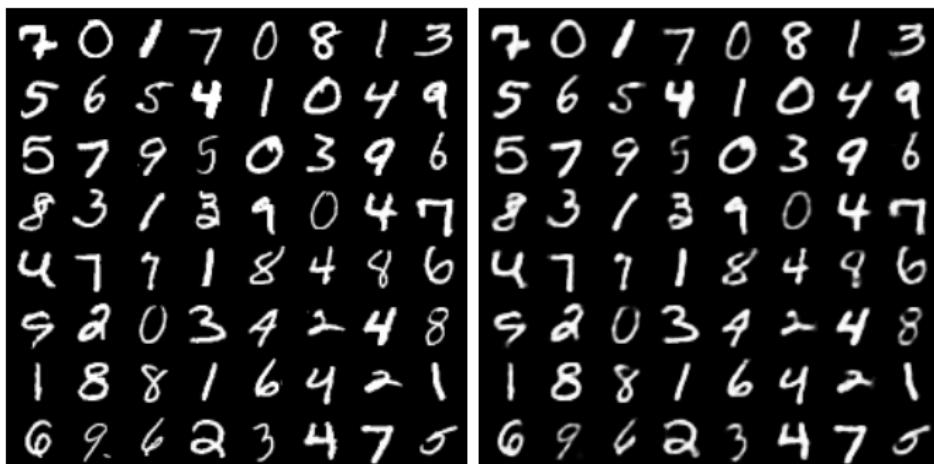
    def encoder(self, x):
        x = self.encoder_linear1(x)
        x = F.relu(x)
        x = self.encoder_linear2(x)
        x = F.relu(x)
        return x

    def decoder(self, x):
        x = self.decoder_linear1(x)
        x = F.relu(x)
        x = self.decoder_linear2(x)
        x = torch.sigmoid(x)
        return x

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

## “Traditional” non-linear autoencoder

- Autoencoder with  $28 \times 28$  dimensional input vector of MNIST examples (left), 10 hidden states, and  $28 \times 28$  dimensional output vector (right).
- Reconstruction error  $\sum_i (\mathbf{o}_i - \mathbf{x}_i)^2$  is almost perfect.

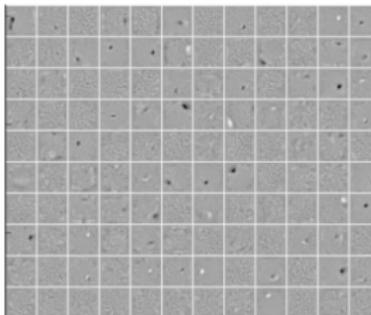


# Denoising autoencoders

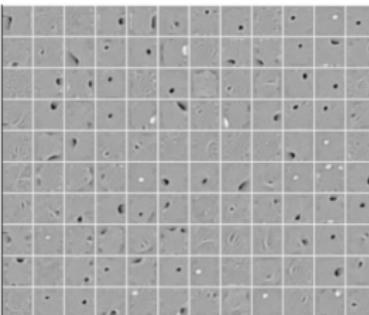
- Add noise to input, force autoencoder to reconstruct a noise-free version:
  - AE must learn to recognize structures.
  - AE must learn not to focus on noise.
  - Usually "drop-out" noise: simply set some inputs to zero  
(This works best on binary data.)

# Denoising autoencoder on MNIST

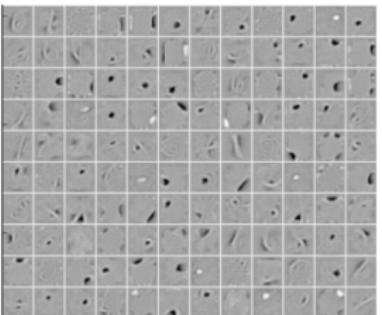
- Patches correspond to rows of learnt weight matrices.
- With increasing level of corruption, an increasing number of filters resembles sensible feature detectors.



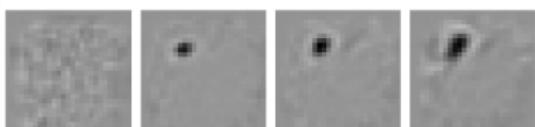
(a) No destroyed inputs



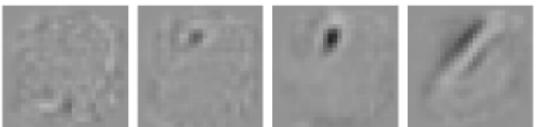
(b) 25% destruction



(c) 50% destruction



(d) Neuron A (0%, 10%, 20%, 50% destruction)



(e) Neuron B (0%, 10%, 20%, 50% destruction)

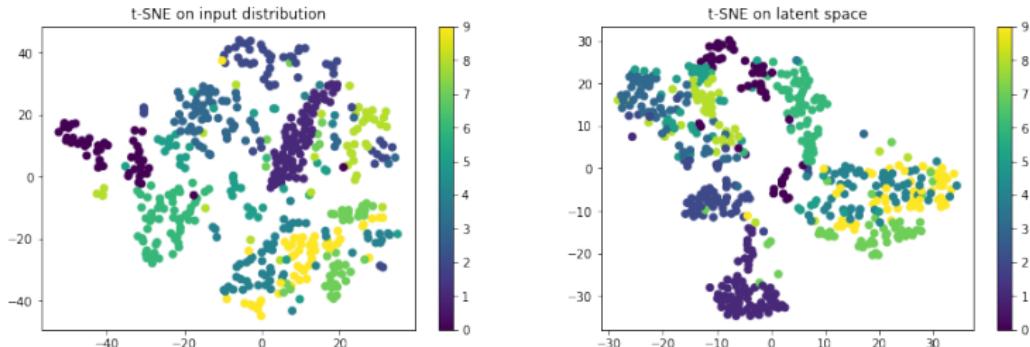
## Autoencoders summarized

- Linear AEs are easy to analyze, but doing PCA isn't very exciting.
- Non-linear activation functions learn more interesting patterns.
- Many different AEs exist:
  - "Traditional" autoencoders
  - Denoising autoencoders
  - **Sparse autoencoders**: forcing hidden units to be inactive most of the time; instead of many large features, we will learn many small ones.
  - ...
- **However**: Autoencoders are limited in their applications: we will discuss why!
- ⇒ **Variational autoencoders (VAE)**

# Variational autoencoders

# Limits of autoencoders

- The latent space of autoencoders is not continuous, or does not allow easy interpolations.
- Continuous latent space is what you need when **building generative models**:
  - Generative models: unsupervised learning
  - You don't want to replicate the input, our classify the input.  
**You want to generate new samples.**

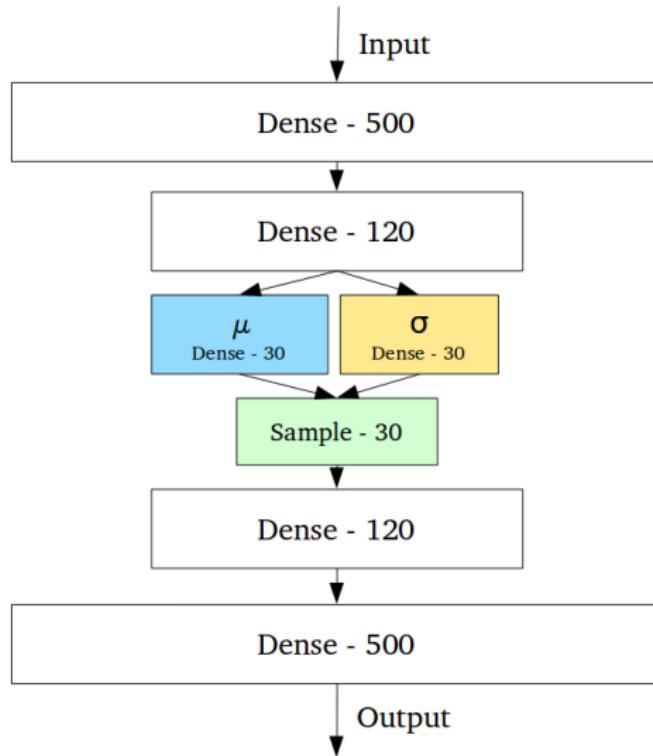


T-SNE visualization of the encodings from the above MNIST examples show distinct clusters.

# Variational autoencoders

- By sampling from discontinuous latent space (gaps between clusters), decoder generates unrealistic outputs.
- Latent space of VAEs is **by design continuous**.
  - Encoder outputs two vectors:
    - vector of means of distribution
    - vector of standard deviations of distribution
  - Force this distribution to be **standard normal distribution**.
  - **Sample from this distribution** and pass the obtained sampling onward to the decoder.
- Through stochastic generation, the encoding will vary on every single pass simply due to sampling.
- The decoder is not only able to decode single encodings in the latent space, but ones that slightly vary too.

# Variational autoencoders



# Pytorch implementation VAE

```
class VAE(nn.Module):
    def __init__(self, ae_input_dim):
        super(VAE, self).__init__()
        # encoder
        # 10-dimensional hidden space
        self.encoder_linear1 = nn.Linear(ae_input_dim, 400)
        self.encoder_linear2_1 = nn.Linear(400, 10)
        self.encoder_linear2_2 = nn.Linear(400, 10)
        # decoder
        self.decoder_linear1 = nn.Linear(10, 400)
        self.decoder_linear2 = nn.Linear(400, ae_input_dim)

    def encoder(self, x):
        x = self.encoder_linear1(x)
        x = F.relu(x)
        return self.encoder_linear2_1(x), self.encoder_linear2_2(x)

    def sampling(self, mu, log_var):
        if self.training:
            std = torch.exp(0.5*log_var)
            eps = torch.randn_like(std)
            return eps.mul(std).add_(mu)
        else:
            # During inference, we simply spit out the mean of the
            # learned distribution for the current input.
            return mu

    def decoder(self, x):
        x = self.decoder_linear1(x)
        x = F.relu(x)
        x = self.decoder_linear2(x)
        return torch.sigmoid(x)

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

# The loss function

- $L$  = Cross Entropy Error  $H_y(g(\mathbf{x}; \mathbf{W}))$ 
  - + Kullback-Leibler divergence  $D_{KL}$  between all the components of  $\mu$  and  $\sigma$  and the standard normal
- $H_y(g(\mathbf{x}; \mathbf{W}))$  is calculated for every pixel (output percentage of target value).
- $D_{KL}$ : measure of how two probability distributions differ from each other

```
def loss_function(x_hat, x, mu, log_var):  
    BCE = F.binary_cross_entropy(x_hat, x, reduction='sum')  
    KLD = -0.5 * torch.sum(1 + log_var - mu.pow(2) - log_var.exp())  
    return KLD + BCE
```

## Making the loss differentiable

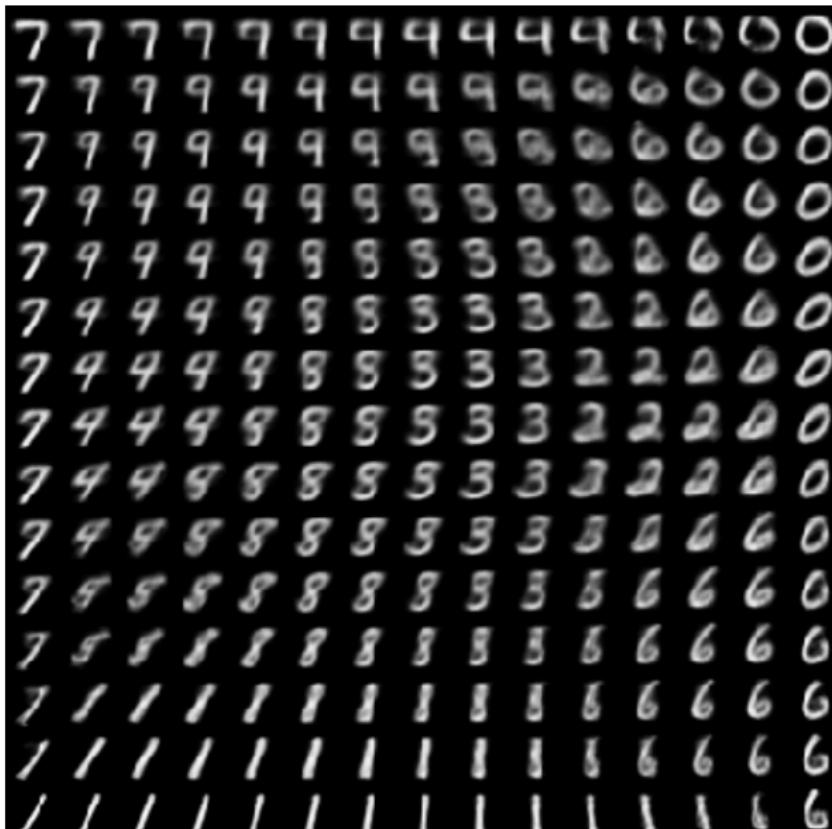
- $x = \text{sample}(\mathcal{N}(\mu, \sigma))$  is not **backpropable** with respect to  $\mu$  and  $\sigma$
- However, it can be rewritten as:
  - $x = \mu + \sigma \cdot \text{sample}(\mathcal{N}(0, 1))$
  - This is backpropable now! :)

## Variational autoencoders on MNIST

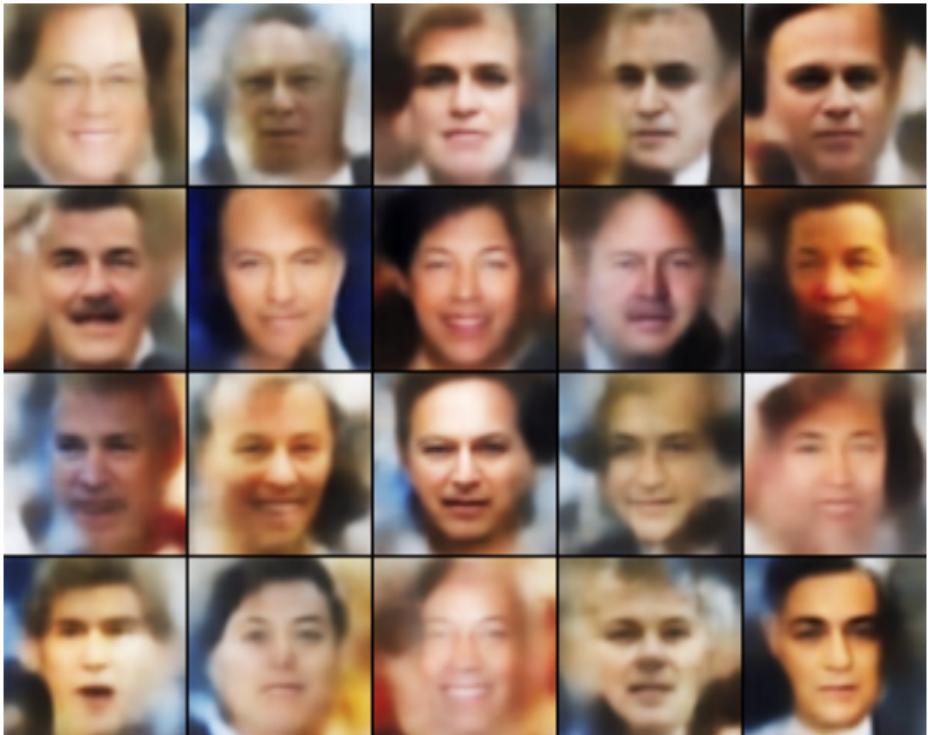
- Variational autoencoder with a  $28 \times 28$  dimensional input vector for every MNIST example, 10 hidden states, and a  $28 \times 28$  dimensional output vector (left)
- 64 random vectors of a 10-dimensional  $\mathcal{N}(0, 1)$  are drawn and handed to the decoder (right)



## VAE walking around in the latent space



# VAE image generation



Larsen and Sønderby. Blog. Generating Faces with Torch, 2015

# VAEs vs GANs

- Variational autoencoders ...
  - ... are usually easier to train and easier to get working.
  - ... are relatively easy to implement and robust to hyperparameter choices.
  - ... have an explicit inference network. So we can do reconstruction.
- However, variational autoencoders collapse all the dimensions in the latent, which are not needed:
  - This limits the generative power to a certain degree.
  - ⇒ Generative adversarial networks ...

# Generative adversarial networks

# Cool idea



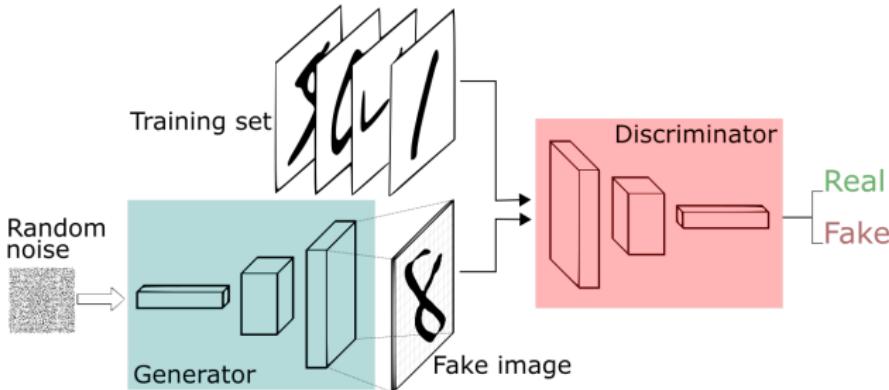
“The coolest idea in machine learning in the last twenty years.”  
(Yann LeCun)

# The guy behind GANs



# Min-max game

- Two networks play against each other.
- Automatically labeled data: generated, real
- Generator: tries to generate realistic images, tries to fool the discriminator
- Discriminator: tries to tell apart real and generated images, tries to outsmart the generator



Goodfellow et al. Generative adversarial nets. In Advances in Neural Information Processing Systems, 2014

# GAN loss discriminator

- Remember cross entropy error:

$$L = - \sum_i y_i \log g(\mathbf{x}_i; \mathbf{W}) + (1 - y_i) \log(1 - g(\mathbf{x}_i; \mathbf{W}))$$

- In the GAN setting:

$$\underbrace{\min_D L(D)}_{D} = -\mathbb{E}_{\mathbf{x} \sim p_{\text{real}}(\mathbf{x})} [\log D(\mathbf{x})] - \mathbb{E}_{\mathbf{z} \sim p_{\text{gen}}(\mathbf{z})} [\log (1 - D(\mathbf{z}))]$$

$$\underbrace{\max_D L(D)}_{D} = \mathbb{E}_{\mathbf{x} \sim p_{\text{real}}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\text{gen}}(\mathbf{z})} [\log (1 - D(\mathbf{z}))]$$

# GAN loss discriminator

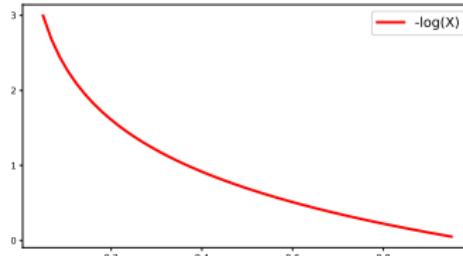
- Remember cross entropy error:

$$L = - \sum_i y_i \log g(\mathbf{x}_i; \mathbf{W}) + (1 - y_i) \log(1 - g(\mathbf{x}_i; \mathbf{W}))$$

- In the GAN setting:

$$\underbrace{\min_D L(D)}_{D} = -\mathbb{E}_{\mathbf{x} \sim p_{\text{real}}(\mathbf{x})} [\log D(\mathbf{x})] - \mathbb{E}_{\mathbf{z} \sim p_{\text{gen}}(\mathbf{z})} [\log (1 - D(\mathbf{z}))]$$

$$\underbrace{\max_D L(D)}_{D} = \mathbb{E}_{\mathbf{x} \sim p_{\text{real}}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\text{gen}}(\mathbf{z})} [\log (1 - D(\mathbf{z}))]$$



□ Probability of D(real)

# GAN loss discriminator

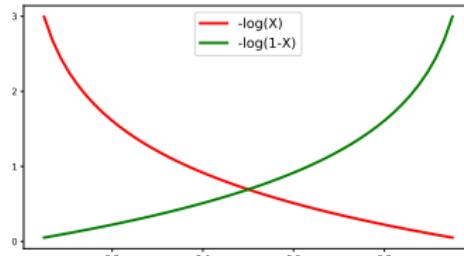
- Remember cross entropy error:

$$L = - \sum_i y_i \log g(\mathbf{x}_i; \mathbf{W}) + (1 - y_i) \log(1 - g(\mathbf{x}_i; \mathbf{W}))$$

- In the GAN setting:

$$\underbrace{\min_D L(D)}_{D} = -\mathbb{E}_{\mathbf{x} \sim p_{\text{real}}(\mathbf{x})} [\log D(\mathbf{x})] - \mathbb{E}_{\mathbf{z} \sim p_{\text{gen}}(\mathbf{z})} [\log (1 - D(\mathbf{z}))]$$

$$\underbrace{\max_D L(D)}_{D} = \mathbb{E}_{\mathbf{x} \sim p_{\text{real}}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\text{gen}}(\mathbf{z})} [\log (1 - D(G(\mathbf{z})))]$$



- Probability of D(real)
- Probability of D(fake)

# GAN loss generator

- Discriminator loss:

$$\underbrace{\max_D L(D)}_{D} = \mathbb{E}_{\mathbf{x} \sim p_{\text{real}}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\text{gen}}(\mathbf{z})} [\log (1 - D(G(\mathbf{z})))]$$

- Generator loss:

$$\begin{aligned}\underbrace{\min_G L(D, G)}_{G} &= \mathbb{E}_{\mathbf{z} \sim p_{\text{gen}}(\mathbf{z})} [\log (1 - D(G(\mathbf{z})))] \\ &\simeq \underbrace{\max_G \mathbb{E}_{\mathbf{z} \sim p_{\text{gen}}(\mathbf{z})} [\log (D(G(\mathbf{z})))]}_{G}\end{aligned}$$

- Putting it all together:

$$\underbrace{\min_G \max_D L(D)}_{D} = \mathbb{E}_{\mathbf{x} \sim p_{\text{real}}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\text{gen}}(\mathbf{z})} [\log (1 - D(G(\mathbf{z})))]$$

# Pytorch implementation GAN

```
class Generator(nn.Module):
    def __init__(self, g_input_dim, g_output_dim):
        super(Generator, self).__init__()
        self.linear1 = nn.Linear(g_input_dim, 256)
        self.linear2 = nn.Linear(256, 512)
        self.linear3 = nn.Linear(512, 1024)
        self.linear4 = nn.Linear(1024, g_output_dim)

    def forward(self, x):
        x = self.linear1(x)
        x = F.relu(x)
        x = self.linear2(x)
        x = F.relu(x)
        x = self.linear3(x)
        x = F.relu(x)
        x = self.linear4(x)
        return torch.tanh(x)

class Discriminator(nn.Module):
    def __init__(self, d_input_dim):
        super(Discriminator, self).__init__()
        self.linear1 = nn.Linear(d_input_dim, 1024)
        self.linear2 = nn.Linear(1024, 512)
        self.linear3 = nn.Linear(512, 256)
        self.linear4 = nn.Linear(256, 1)

    def forward(self, x):
        x = self.linear1(x)
        x = F.relu(x)
        x = F.dropout(x, 0.3)
        x = self.linear2(x)
        x = F.relu(x)
        x = F.dropout(x, 0.3)
        x = self.linear3(x)
        x = F.relu(x)
        x = F.dropout(x, 0.3)
        x = self.linear4(x)
        return torch.sigmoid(x)
```

# Pytorch implementation generator

```
def train_discriminator(x):
    D.zero_grad()

    # discriminator loss on real
    x_real, y_real = x.view(-1, mnist_dim), torch.ones(x.view(-1, mnist_dim).shape[0], 1)
    x_real, y_real = Variable(x_real.to(device)), Variable(y_real.to(device))

    D_output = D(x_real)
    D_real_loss = criterion(D_output, y_real)
    D_real_score = D_output

    # discriminator loss on fake
    z = Variable(torch.randn(args.batch_size, z_latent).to(device))
    x_fake, y_fake = G(z), Variable(torch.zeros(args.batch_size, 1).to(device))

    D_output = D(x_fake)
    D_fake_loss = criterion(D_output, y_fake)
    D_fake_score = D_output

    # gradient backprop
    D_loss = D_real_loss + D_fake_loss
    D_loss.backward()
    D_optimizer.step()

    return D_loss.data.item()
```

# Pytorch implementation discriminator

```
def train_generator():
    G.zero_grad()

    z = Variable(torch.randn(args.batch_size, z_latent).to(device))
    y = Variable(torch.ones(args.batch_size, 1).to(device))

    G_output = G(z)
    D_output = D(G_output)
    G_loss = criterion(D_output, y)

    # gradient backprop
    G_loss.backward()
    G_optimizer.step()

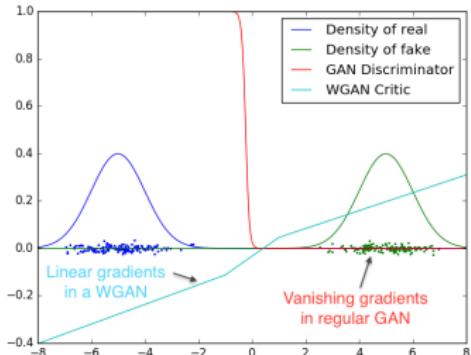
    return G_loss.data.item()
```

# Loss as divergence

- For solving the optimization problem of the generator the loss can be interpreted as Jensen-Shannon divergence  $JSD(p_{real}(\mathbf{x}), p_{gen}(\mathbf{x}))$ , which has to be minimized.
- Jensen-Shannon divergence is the symmetrized Kullback-Leibler divergence.
- Both are measures of the similarity between probability distributions.
- This has put up the questions if there are maybe better similarity measurements?

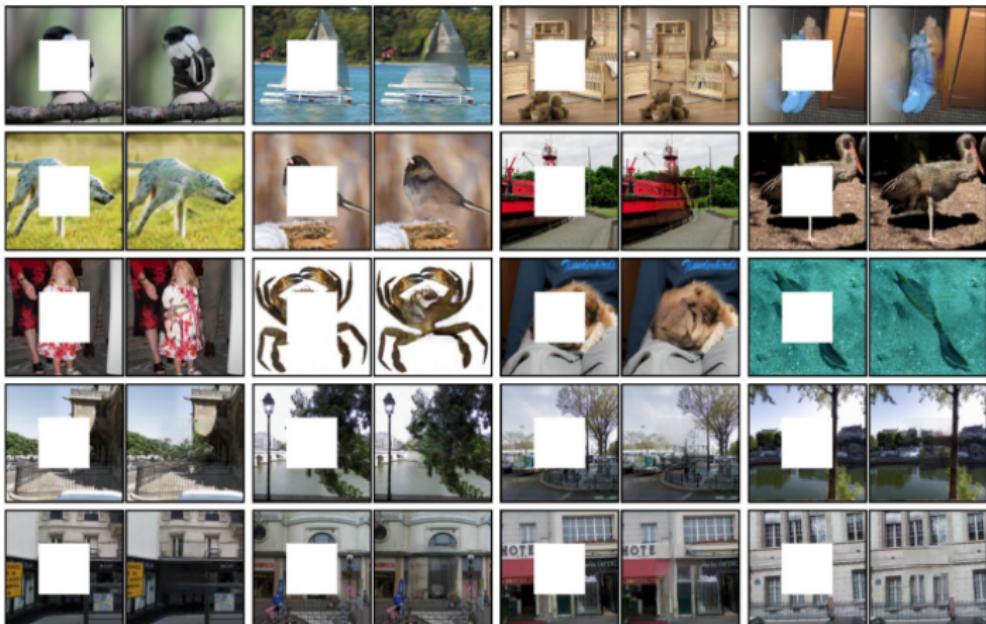
# Wasserstein distant

- Out of thousands the Wasserstein GAN worked best.
- The reason is twofold:
  - Wasserstein distance has gradients even if distributions are **not overlapping**.
  - A penalty term (free when clever implemented) allows to **move distributions towards regions of largest change**:
    - similar to the spectral norm and the largest eigenvalue

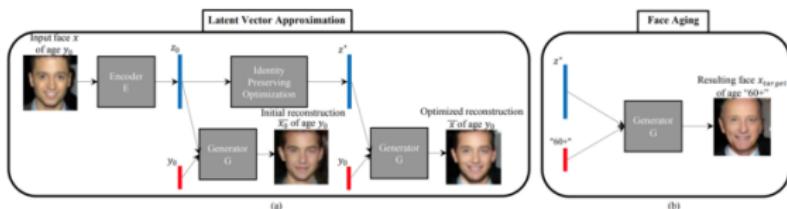
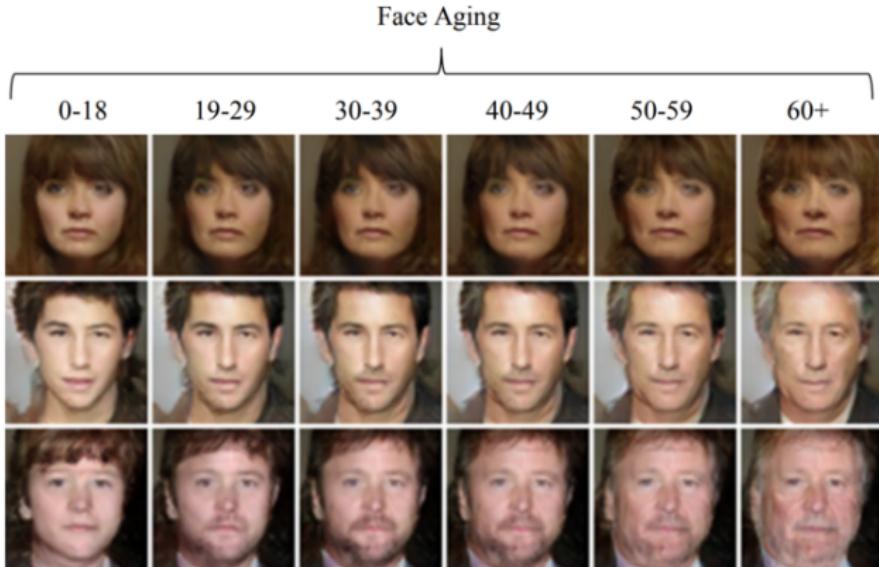


# Image inpainting

- Repairing images has been an important subject decades ago. GANs are ideally suited for filling the missing part with created “content”.



# Face aging



# Music generation

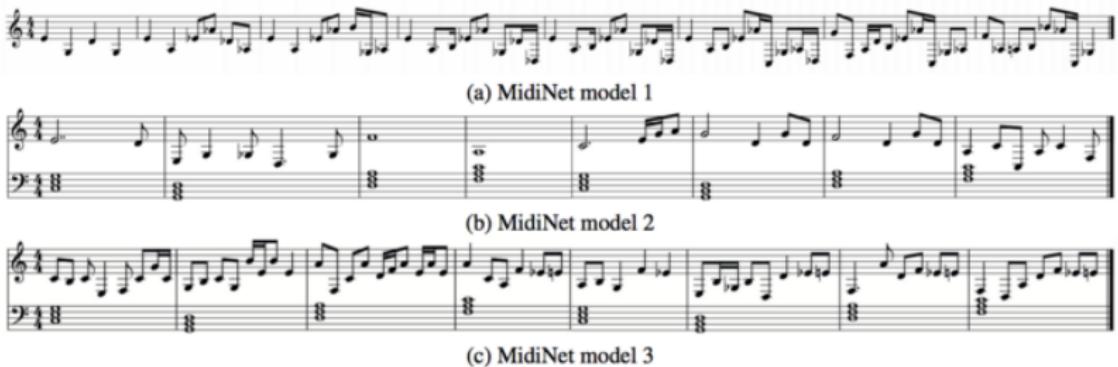
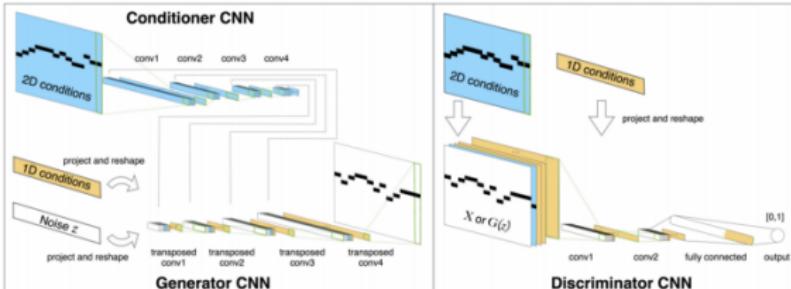


Figure 3. Example result of the melodies (of 8 bars) generated by different implementations of MidiNet.



# Emojis from pictures

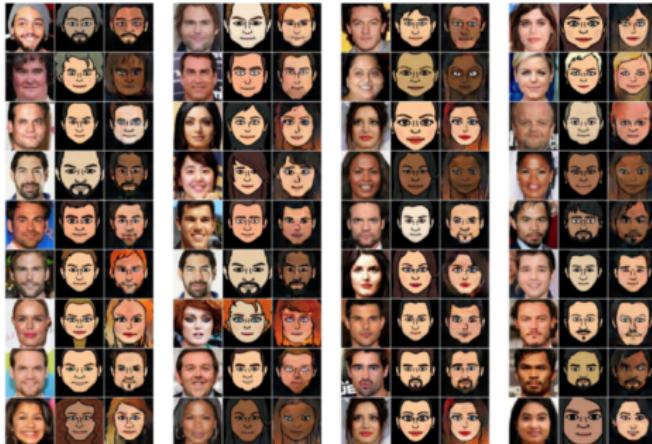


Figure 4: Shown, side by side are sample images from the CelebA dataset, the emoji images created manually using a web interface (for validation only), and the result of the unsupervised DTN. See Tab. 4 for retrieval performance.

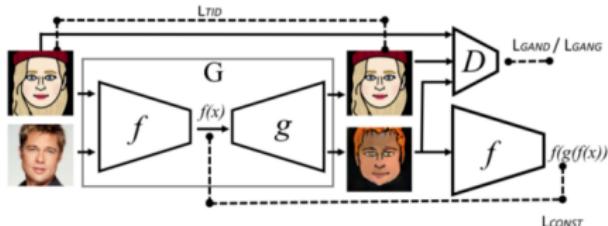


Figure 1: The Domain Transfer Network. Losses are drawn with dashed lines, input/output with solid lines. After training, the forward model G is used for the sample transfer.

# Text to image

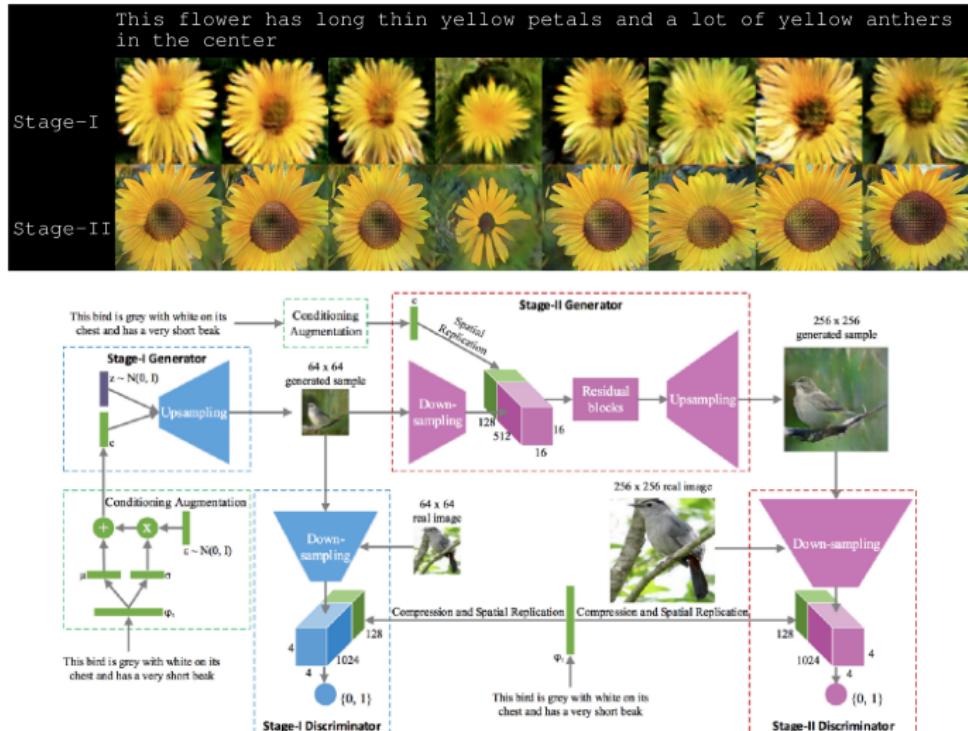


Figure 2. The architecture of the proposed StackGAN. The Stage-I generator draws a low resolution image by sketching rough shape and basic colors of the object from the given text and painting the background from a random noise vector. The Stage-II generator generates a high resolution image with photo-realistic details by conditioning on both the Stage-I result and the text again.

# CycleGAN



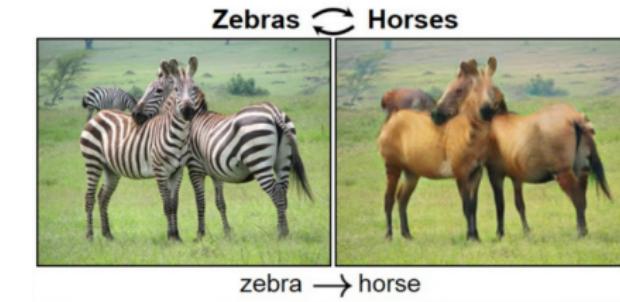
Photograph

Monet

Van Gogh

Cezanne

Ukiyo-e



zebra → horse



horse → zebra

# DiscoGAN



(b) Handbag images (input) & Generated shoe images (output)

# High-resolution image synthesis