# Autoencoders

An autoencoder consists of two parts:

- $\phi : \mathbb{R}^n \to \mathbb{R}^m$ the encoder
- $\psi : \mathbb{R}^m \to \mathbb{R}^n$ the decoder

The goal is to find $\phi$, $\psi$ so that $\psi \circ \phi$ is (almost) identity.

The value $\vec{h} = \phi(\vec{x})$ is called the *latent representation* of $\vec{x}$.

# Autoencoders – training

Assume

$$\mathcal{T} = \{\vec{x}_1, \ldots, \vec{x}_p\}$$

where $\vec{x}_i \in \mathbb{R}^n$ for all $i \in \{1, \ldots, n\}$.

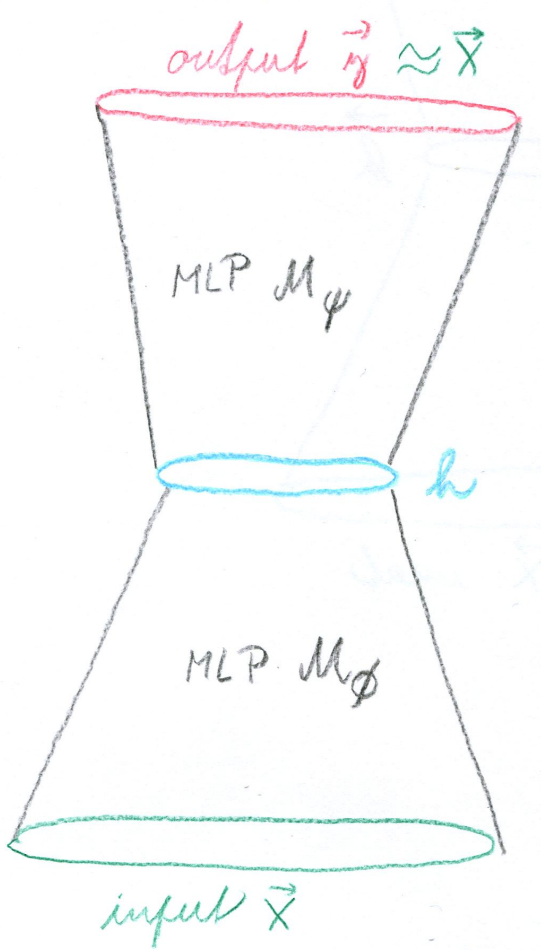
Minimize the **reconstruction** error

$$E = \sum_{i=1}^{p} (\vec{x}_i - \psi(\phi(\vec{x}_i)))^2$$

# Autoencoders – neural networks

Both $\phi$ and $\psi$ can be represented using MLP $\mathcal{M}_\phi$ and $\mathcal{M}_\psi$, respectively.

$\mathcal{M}_\phi$ and $\mathcal{M}_\psi$ can be connected into a single network.



# Autoencoders – Usage

- Compression – from $\vec{x}$ to $\vec{h}$.
- Dimensionality reduction – the latent representation $\vec{h}$ has a smaller dimension.
- Pretraining (next slides)
- Generative versions – (roughly) generate $\vec{h}$ from a known distribution, let $\mathcal{M}_\psi$ generate realistic inputs $\vec{x}$

# Autoencoder – compression – historical implementation

**Architecture:** MLP 64 – 16 – 64

**Activity:** activation function: hyperbolic tangens with limits –1 and 1

# Autoencoder – compression – historical implementation

**Architecture:** MLP 64 – 16 – 64

**Activity:** activation function: hyperbolic tangens with limits $-1$ and 1

**Data:**

- Images $256 \times 256$, 8 bits per pixel.
- Samples: input and output is a frame $8 \times 8$, randomly selected in the image.
- Inputs normalized to $[-1, 1]$.

# Autoencoder – compression – historical implementation

**Architecture:** MLP 64 – 16 – 64

**Activity:** activation function: hyperbolic tangens with limits $-1$ and 1

**Data:**

- Images $256 \times 256$, 8 bits per pixel.
- Samples: input and output is a frame $8 \times 8$, randomly selected in the image.
- Inputs normalized to $[-1, 1]$.

The goal was to compress images to smaller data size.

# Autoencoder – compression – historical implementation

(A)

(B)

(C)

(D)

A frame $8 \times 8$ passes through the image $256 \times 256$ (no overlap)

- **(A)** original
- **(B)** compression
- **(C)** compression + rounding to 6 bits (1.5 bit per pixel)
- **(D)** compression + rounding to 4 bits (1 bit per pixel)

# Dimensionality reduction – compression

(A)

(B)

(C)

(D)

New image (trained on the previous one):

- **(A)** original
- **(B)** compression
- **(C)** compression + rounding to 6 bits (1.5 bit per pixel)
- **(D)** compression + rounding to 4 bits (1 bit per pixel)

# Application – dimensionality reduction

- Dimensionality reduction: A mapping $R$ from $\mathbb{R}^n$ to $\mathbb{R}^m$ where
  - $m < n$,
  - for every example $\vec{x}$ we have that $\vec{x}$ can be "reconstructed" from $R(\vec{x})$.

# Application – dimensionality reduction

- Dimensionality reduction: A mapping $R$ from $\mathbb{R}^n$ to $\mathbb{R}^m$ where
  - $m < n$,
  - for every example $\vec{x}$ we have that $\vec{x}$ can be "reconstructed" from $R(\vec{x})$.
- Standard method: PCA (there are many linear as well as non-linear variants)

# Reconstruction – PCA


1024 pixels compressed to 100 dimensions (i.e. 100 numbers).

# PCA vs Autoencoders

Real data

30-d deep autoencoder

30-d logistic PCA

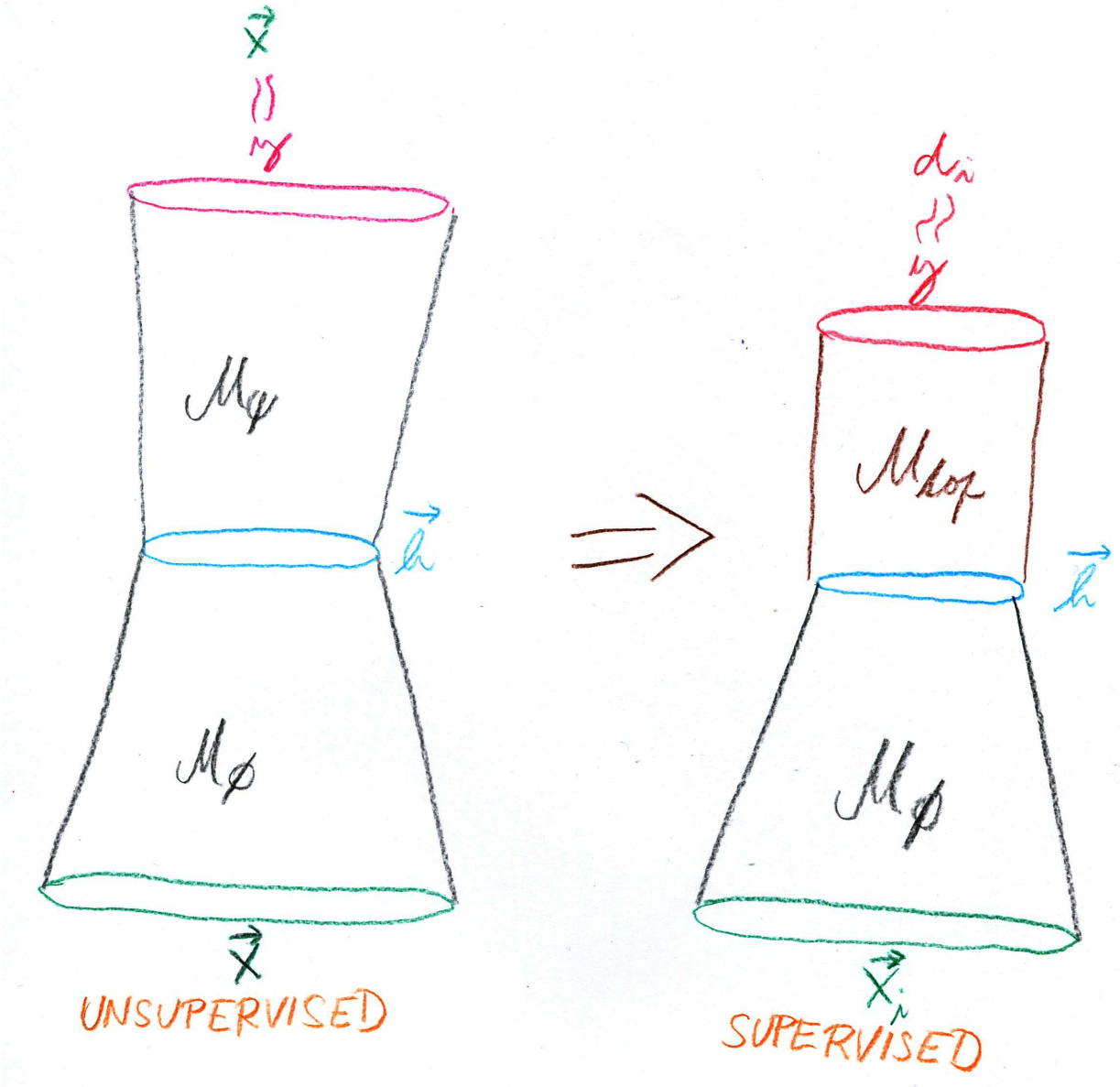
30-d PCA

# Autoencoders – Pretraining

- An autoencoder is (pre)trained on input data $\vec{x}_i$ *without* desired outputs (unsupervised)

  typically much larger datasets of unlabelled data
- the encoder $\mathcal{M}_\phi$ computes a latent representation for every input vector, it is supposed to extract important features (controversial)
- A new part of the model $\mathcal{M}_{top}$ is added on top of $\mathcal{M}_\phi$ (e.g. a MLP taking the output of $\mathcal{M}_\phi$ as an input).
- Subsequently, labels are added and the whole model (composed of $\mathcal{M}_\phi$ and $\mathcal{M}_{top}$) is trained on labelled data.

# Autoencoders – Pretraining



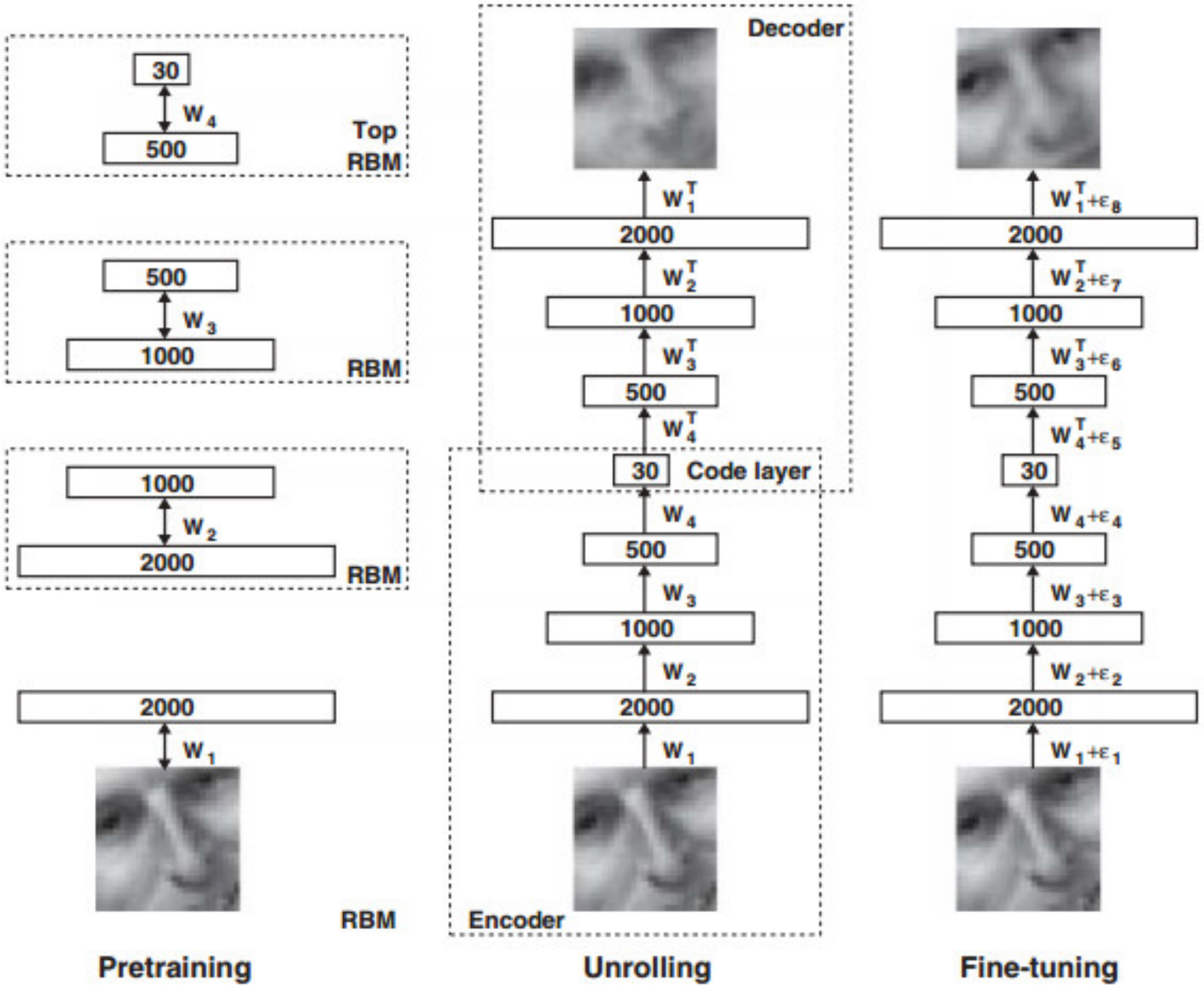
# Deep MLP – dimensionality reduction

Hinton, G. E., Osindero, S. and Teh, Y. (2006)
A fast learning algorithm for deep belief nets.
Neural Computation, 18, pp 1527-1554.

Hinton, G. E. and Salakhutdinov, R. R. (2006)
Reducing the dimensionality of data with neural networks.
Science, Vol. 313. no. 5786, pp. 504 - 507, 28 July 2006.

This basically started all the deep learning craze ...

# Deep MLP – dimensionality reduction



# Images – pretraining

- **Data:** 165 600 black-white images, $25 \times 25$, mean intensity 0, variance 1.

  Images obtained from Olivetti Faces database of images $64 \times 64$ using standard transformations.
- 103 500 training set, 20 700 validation, 41 400 test
- **Network:** 2000-100-500-30, training using layered RBM.

Notes:

Training of the lowest layer (2000 neurons): Values of pixels distorted using Gaussian noise, low learning rate: 0.001, 200 iterations

Training all hidden layers: Values of neurons are binary.

Training of output layer: Values computed directly using the sigmoid activation functions + noise. That is, values of output neurons are from the interval $[0, 1]$.

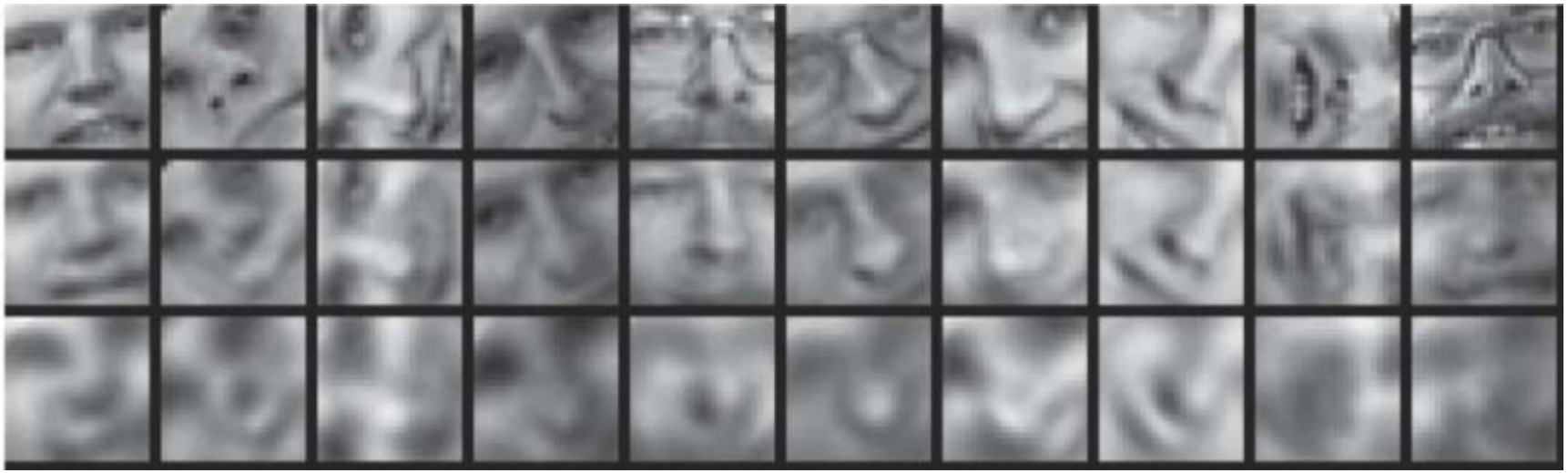
# Images – fine-tuning

- Stochastic activation substituted with deterministic.
  That is the value of hidden neurons is not chosen randomly but directly computed by application of sigmoid on the inner potential (this gives the mean activation).
- Backpropagation.
- Error function: cross-entropy

$$-\sum_i p_i \ln \hat{p}_i - \sum_i (1 - p_i) \ln(1 - \hat{p}_i)$$

here $p_i$ is the intensity of $i$-th pixel of the input and $\hat{p}_i$ of the reconstruction.

# Results



1. Original
2. Reconstruction using deep networks (reduction to 30-dim)
3. Reconstruction using PCA (reduction to 30-dim)

# Generative adversarial networks

Generative adversarial Nets, Goodfellow et al, NIPS 2014

An unsupervised generative model.

Two networks:

- **Generator**: A network computing a function $G : \mathbb{R}^k \to \mathbb{R}^n$ which takes a *random* input $z$ with a distribution $p_z$ (e.g. multivariate normal distribution) and returns $G(z)$ which should follow the target probability distribution.

  E.g. $G(z)$ could be realistically looking faces.

- **Discriminator**: A network computing a function $D : \mathbb{R}^n \to [0, 1]$ that given $x \in \mathbb{R}^n$ gives a probability $D(x)$ that $x$ is *not* "generated" by $G$.

  E.g. $x$ can be an image, $D(x)$ is a probability that it is a true face of an existing person.

What error function will "motivate" $G$ to generate realistically and $D$ to discriminate appropriately?

# Generative adversarial networks – error function

Let $\mathcal{T} = \{\vec{x}_1, \ldots, \vec{x}_p\}$ be a training multiset (or a minibatch).

**Intuition**: $G$ should produce outputs similar to elements of $\mathcal{T}$.
$D$ should recognize that its input is not from $\mathcal{T}$.

# Generative adversarial networks – error function

Let $\mathcal{T} = \{\vec{x}_1, \ldots, \vec{x}_p\}$ be a training multiset (or a minibatch).

**Intuition**: $G$ should produce outputs similar to elements of $\mathcal{T}$. $D$ should recognize that its input is not from $\mathcal{T}$.

Generate a multiset of noise samples: $\mathcal{F} = \{\vec{z}_1, \ldots, \vec{z}_p\}$ from the distribution $p_z$.

$$E_{\mathcal{T},\mathcal{F}}(G, D) = -\frac{1}{p} \sum_{i=1}^{p} \left( \ln D(\vec{x}_1) + \ln(1 - D(G(\vec{z}_1))) \right)$$

This is just the binary cross entropy error of $D$ which classifies the input as either real, or fake.

The problem can be seen as a game: The discriminator wants to minimize $E$, the generator wants to maximize $E$!

# The learning algorithm

Denote by $W_G$ and $W_D$ the weights of $G$ and $D$, respectively.

In every iteration of the training, modify weights of the discriminator and the generator as follows:

# The learning algorithm

Denote by $W_G$ and $W_D$ the weights of $G$ and $D$, respectively.

In every iteration of the training, modify weights of the discriminator and the generator as follows:

For $k$ steps (here $k$ is a hyperparameter) update the discriminator as follows:

- Sample a minibatch $T = \{\vec{x}_1, \ldots, \vec{x}_m\}$ from the training set $\mathcal{T}$.
- Sample a minibatch $F = \{\vec{z}_1, \ldots, \vec{z}_m\}$ from the distribution $p_z$.
- Update $W_D$ using the *gradient descent* w.r.t. $E$:

$$W_D := W_D - \alpha \cdot \nabla_{W_D} E_{T,F}(G, D)$$

# The learning algorithm

Denote by $W_G$ and $W_D$ the weights of $G$ and $D$, respectively.

In every iteration of the training, modify weights of the discriminator and the generator as follows:

For $k$ steps (here $k$ is a hyperparameter) update the discriminator as follows:

- Sample a minibatch $T = \{\vec{x}_1, \ldots, \vec{x}_m\}$ from the training set $\mathcal{T}$.
- Sample a minibatch $F = \{\vec{z}_1, \ldots, \vec{z}_m\}$ from the distribution $p_z$.
- Update $W_D$ using the *gradient descent* w.r.t. $E$:

$$W_D := W_D - \alpha \cdot \nabla_{W_D} E_{T,F}(G, D)$$

Now update the generator:

- Sample a minibatch $F = \{\vec{z}_1, \ldots, \vec{z}_m\}$ from the distribution $p_z$.
- Update the *generator* by *gradient descent*:

$$W_G := W_G - \alpha \cdot \nabla_{W_G} \left( \frac{1}{p} \sum_{i=1}^{p} \ln(1 - D(G(\vec{z}_1))) \right)$$

(The updates may also use momentum, adaptive learning rate etc.)

# GAN MNIST

# GAN faces

... from the original paper.

# GAN refined

... after some refinements.

... none of these people ever lived.