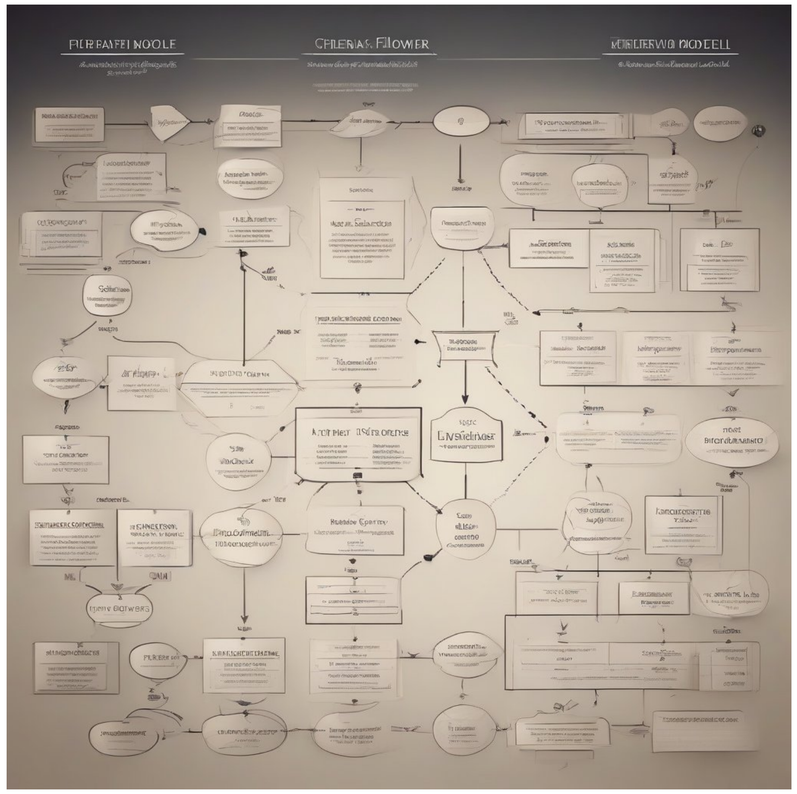


CS545 – Machine Learning for Signal Processing

Deep Learning III: Generative models

Paris Smaragdis
paris@illinois.edu
paris.cs.illinois.edu

Generative Models

- Generative models are good for synthesizing data
  - E.g., speech synthesis, image generation, videos, etc.

- Many ways to go about it
  - AR models
  - Generative Adversarial Networks (GANs)
  - Variational Auto-Encoders (VAEs)
  - Flows
  - Diffusion

High-level idea to keep in mind

- Remember LDS models?

$$\mathbf{x}_{t+1} = \mathbf{A} \cdot \mathbf{x} + \mathcal{N}(0, \mathbf{Q})$$
$$\mathbf{y}_t = \mathbf{C} \cdot \mathbf{x}_t + \mathcal{N}(0, \mathbf{R})$$

- Lots of neural generative models build on this
  - But instead of being “Linear” they are “Neural”
    - i.e. can have more parameters, and have more learning capacity

AutoRegressive models (AR)

- Taking a weighted sum of previous values

$$y_t = \sum_k a_k x_{t-k}$$

- Ubiquitous model in time series processing
  - We already used these to predict missing samples, to classify sequences, etc.

Deepening AR models

- The linear combination of past samples is limited
  - Can only use so many parameters, is only a linear transform

- "Neuralizing" the AR model
  - Replace the transform applied on prior samples with a neural net
  - Key observation: Our model uses convolution, so can use a CNN
    - Note: we only use *causal* convolutions (i.e. using only past samples)

So how does this differ from a CNN?

- In generative AR models we try to predict the future
  - Input is past samples, target output is future samples

$$\hat{\mathbf{x}}_{t+1} = \mathrm{ARModel}(\mathbf{x}_{t,\ldots,t-N})$$

- This allows us to create a signal one sample at a time
  - Start with a seed of past values (can just be noise too)
  - Predict the future output based on it

Some problems

- Neural nets are better when saturating
  - i.e. great with binary outputs that saturate an activation
  - So-so performance for real-valued regression outputs

- This AR model is deterministic
  - You get the same answer for the same prior sample inputs
  - Makes it harder to generate multiple outputs and pick the best

A solution

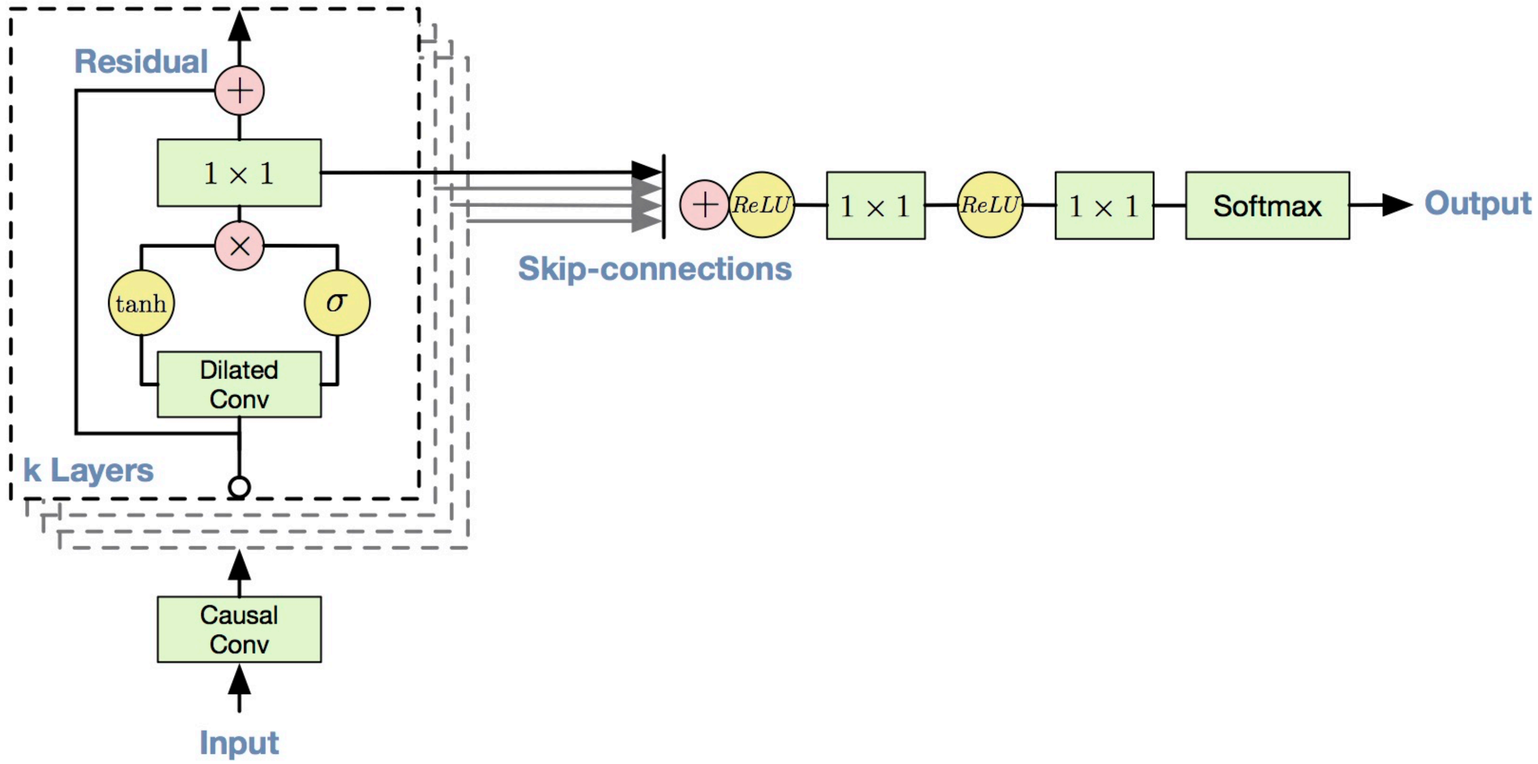
- Predict the distribution of the output, not the output

$$P(\hat{\mathbf{x}}_{t+1}) = \mathrm{ARModel}(\mathbf{x}_{t,\ldots,t-N})$$

- In order to generate $\mathbf{x}_{t+1}$ we just sample it
  - Downside: we cannot model a continuous distribution, so we have to estimate a discrete distribution, i.e. a quantized $\mathbf{x}_{t+1}$
    - I.e. $\begin{bmatrix} P(\hat{\mathbf{x}}_{t+1}) = v_1 \\ P(\hat{\mathbf{x}}_{t+1}) = v_2 \\ \ldots \\ P(\hat{\mathbf{x}}_{t+1}) = v_N \end{bmatrix} = \mathrm{ARModel}(\mathbf{x}_{t,\ldots,t-N})$

Example application: WaveNet

- Popular autoregressive neural model for sound
  - Works on 8/16-audio samples
  - Uses all the usual deep learning tricks to work!



Concatenative synthesis

Wavenet

Wavenet without text

Wavenet on music

Pros and cons of AR models

- Pros:
  - Great at modeling time sequences
  - Use of convolutions makes for fast/parallel training
  - Filters can often be interpretable

- Cons:
  - Slow generation! You need to do a forward pass for each generated sample
    - Convolutions are not parallelizable at inference time as they are during training
  - Large memory requirements
  - Need lots of training data
  - Not great for non-sequential data

Latent Variable Models

- Use a hidden state to generate data
  - We've already done this with PCA!
- Transform a (usually) Gaussian RV to a data RV
  - Use neural nets which are more powerful than linear transforms
- Better known version: The VAE

Re-examing PCA

- PCA as a map to a standard Gaussian and back
  - Eigendecomposition transforms data to a standard Gaussian
  - Inverse transform projects points from Gaussian to data space

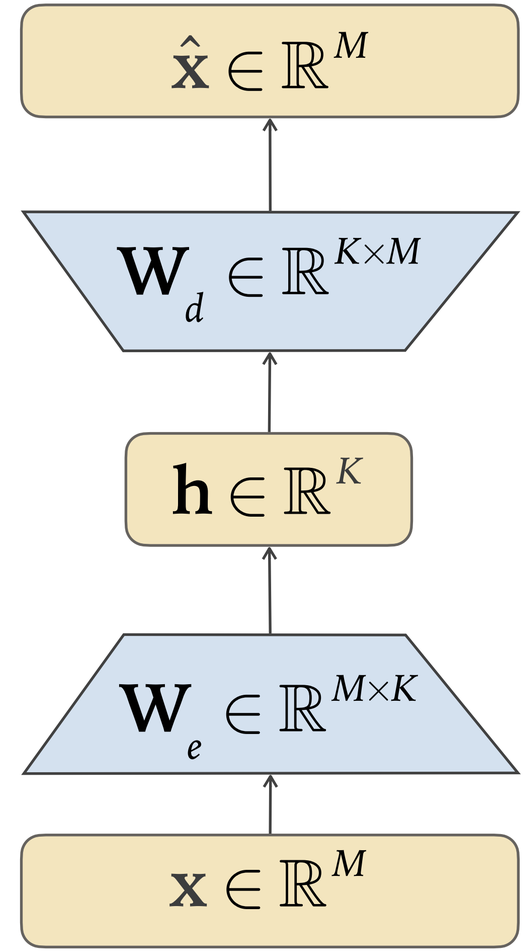
$$\mathbf{h} = \mathbf{W} \cdot \mathbf{x}, \qquad \hat{\mathbf{x}} = \mathbf{W}^\top \cdot \mathbf{h}$$

- We can generate new data by keeping the latter part!
  - Sample a Gaussian point, and put through the inverse transform

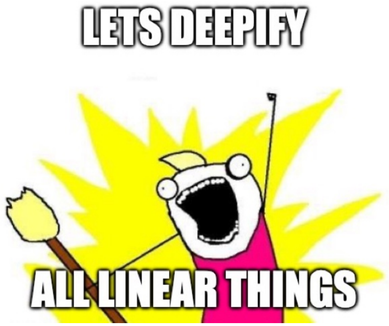
$$\mathbf{x}_{new} = \mathbf{W}^\top \cdot \mathbf{h}_{new}, \qquad \mathbf{h}_{new} \sim \mathcal{N}(0, \mathbf{I})$$

PCA as a shallow auto encoder

- PCA conceptualized as a neural net
  - Layer 1 (encoder) projects data to latent space
    - With added constraint on orthogonality
  - Layer 2 (decoder) projects back to data space

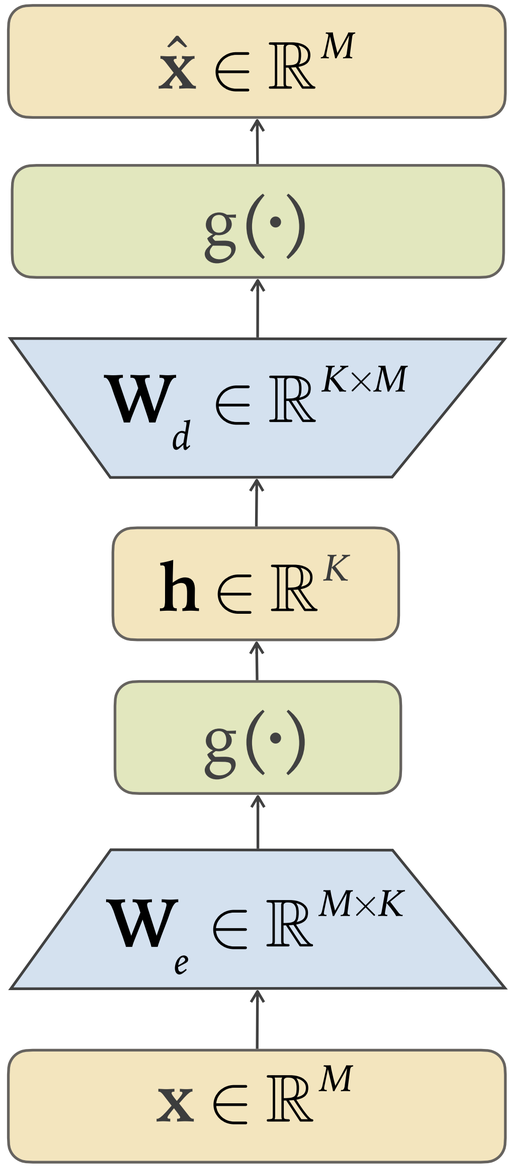


- Limitations
  - Why impose orthogonality?
  - Why use only linear transforms?



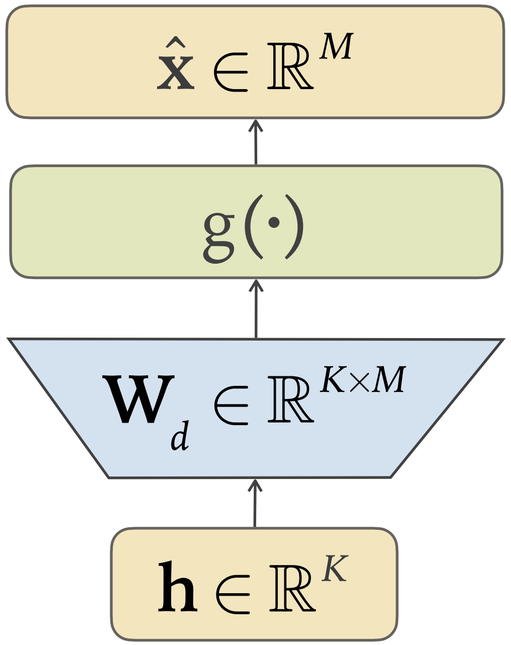
Deep autoencoders

- Consider an autoencoder network
  - High-dimensional input $\mathbf{x}$
  - Transformed to a low-dimensional $\mathbf{h}$
    - Via an *encoder* $\mathrm{g}(\mathbf{W}_e \cdot \mathbf{x})$
  - And back to high-dim approximation of input
    - Via a *decoder* $\mathrm{g}(\mathbf{W}_d \cdot \mathbf{h})$
- $\mathbf{h}$ is a low-dim representation of $\mathbf{x}$
  - Like PCA, but not orthogonal or linear
  - And we can use different types of layers too!



Hallucinating new data

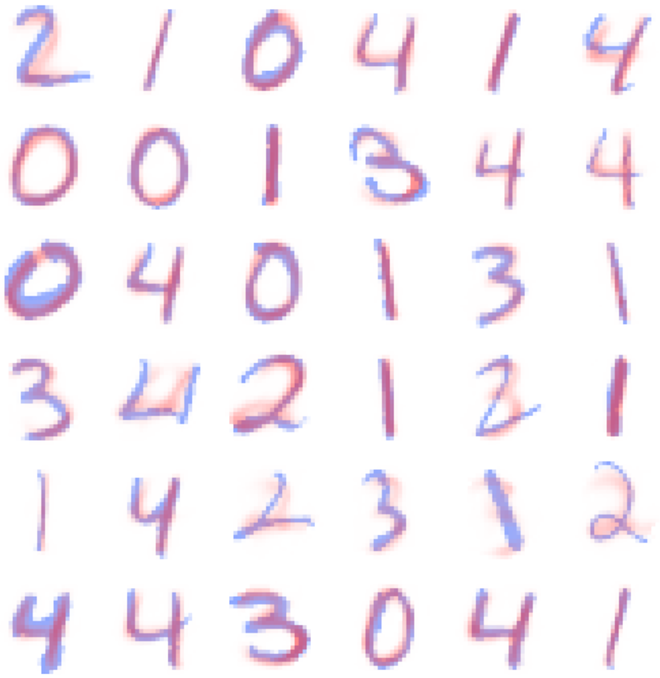
- Trained autoencoder has two parts
  - Encoder: from high-d to low-d
  - Decoder: from low-d to high-d
- The decoder on its own can be used to generate new data
  - Generate a random value for $\mathbf{h}$
  - Pass through decoder to make a new data point



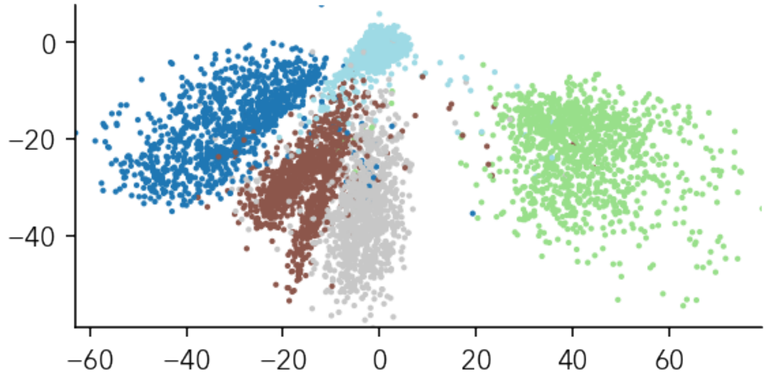
MNIST example

- Train on digits {0,1,2,3,4}, use 2d latent representation
  - Latent dimension is not easy to understand

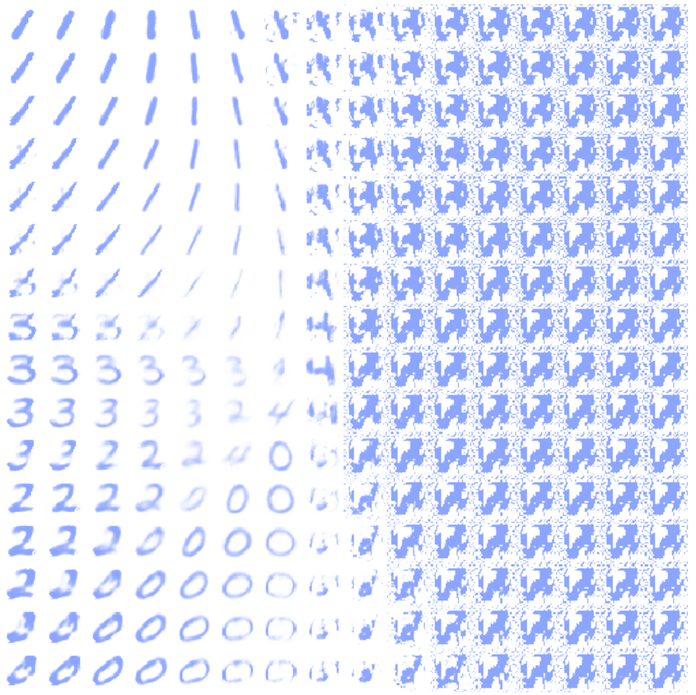
Input/approximations



Latent projection



Latent space

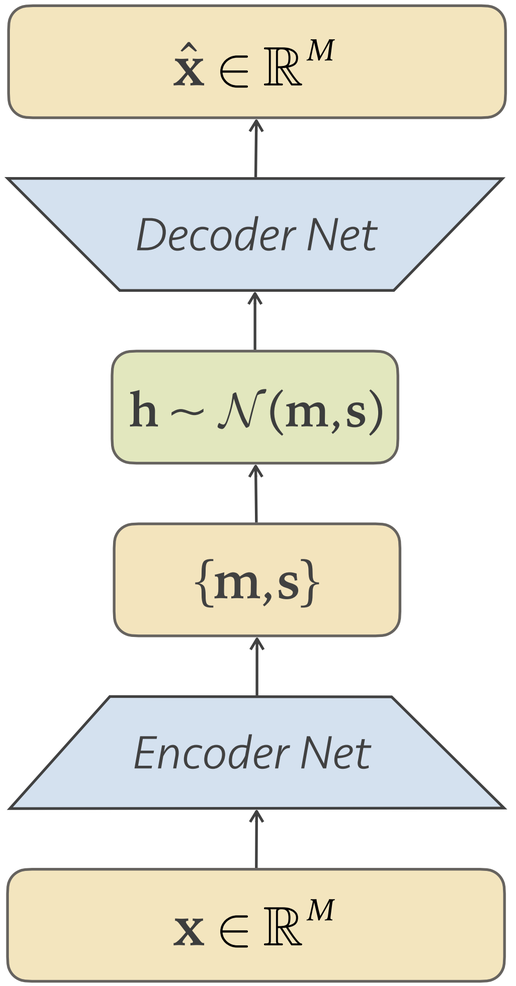


Some problems

- How can we sample plausible $\mathbf{h}$ values?
  - There are gaps which generate garbage data
- How is $\mathbf{h}$ distributed?
  - What range do I sample from?
- What if multiple $\mathbf{x}$'s collapse to a single $\mathbf{h}$?
  - Can I guarantee that the encoder is smooth?

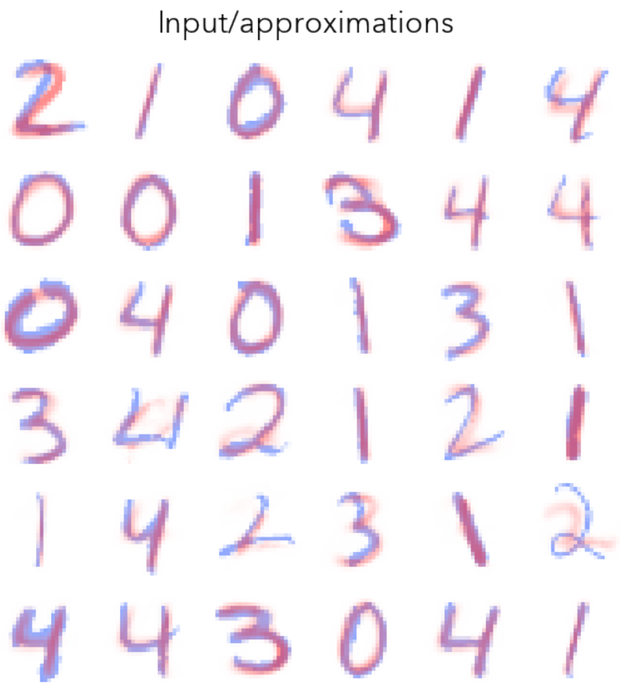
Variational Auto-Encoder (VAE)

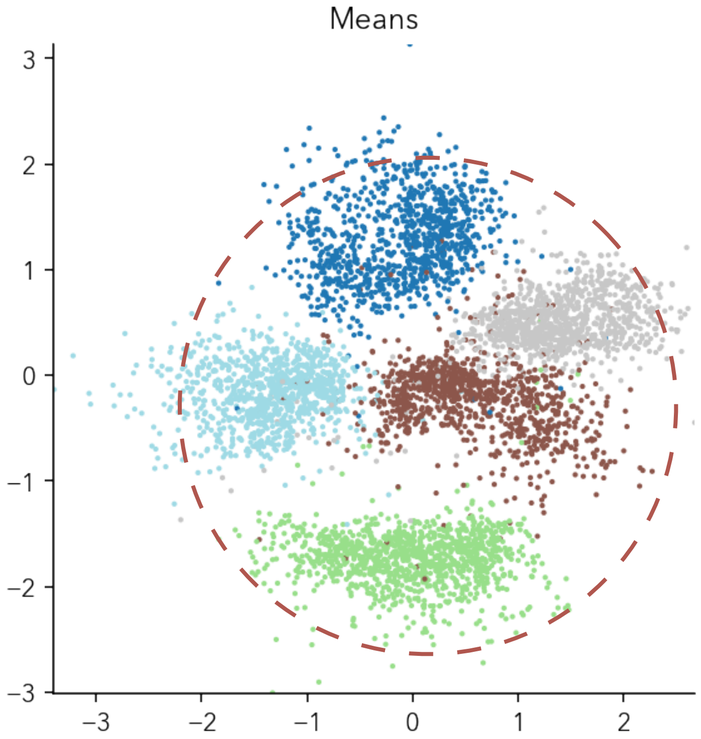
- Force $\mathbf{h}$ to be Gaussian distributed
  - Helps us know the latent distribution
  - Creates a more efficient encoding (more later)
- How to do this?
  - Encoder maps each input $\mathbf{x}$ to a mean and variance
  - We sample a Gaussian with these parameters
  - We present that sample to the decoder
- We additionally minimize: $\mathrm{KL}\left(\mathcal{N}(\mathbf{m},\mathbf{s}) \,\|\, \mathcal{N}(\mathbf{0},\mathbf{I})\right)$
  - i.e. tend to generate $\mathbf{m} = 0$, $\mathbf{s} = \mathbf{I}$

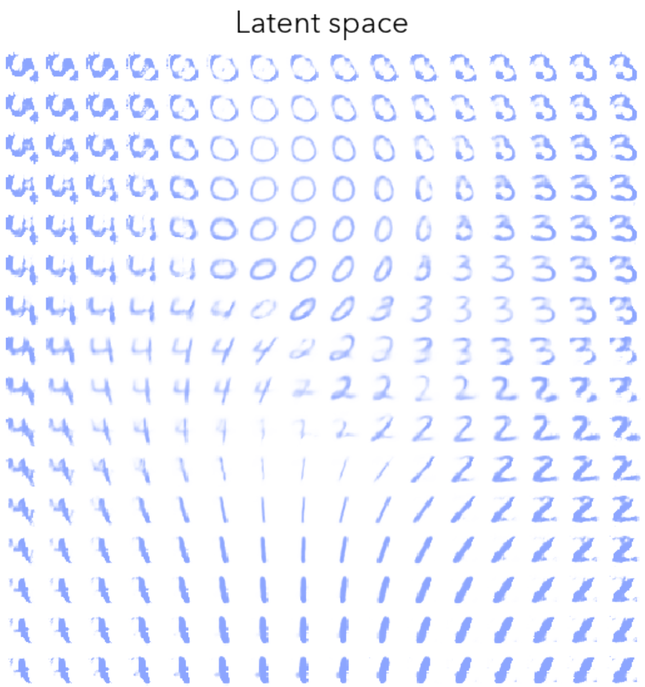


VAE MNIST example

- More convenient results
  - New latent representation is more compact and predictable
  - Sampled latent space generates continuous outputs







Key points to make

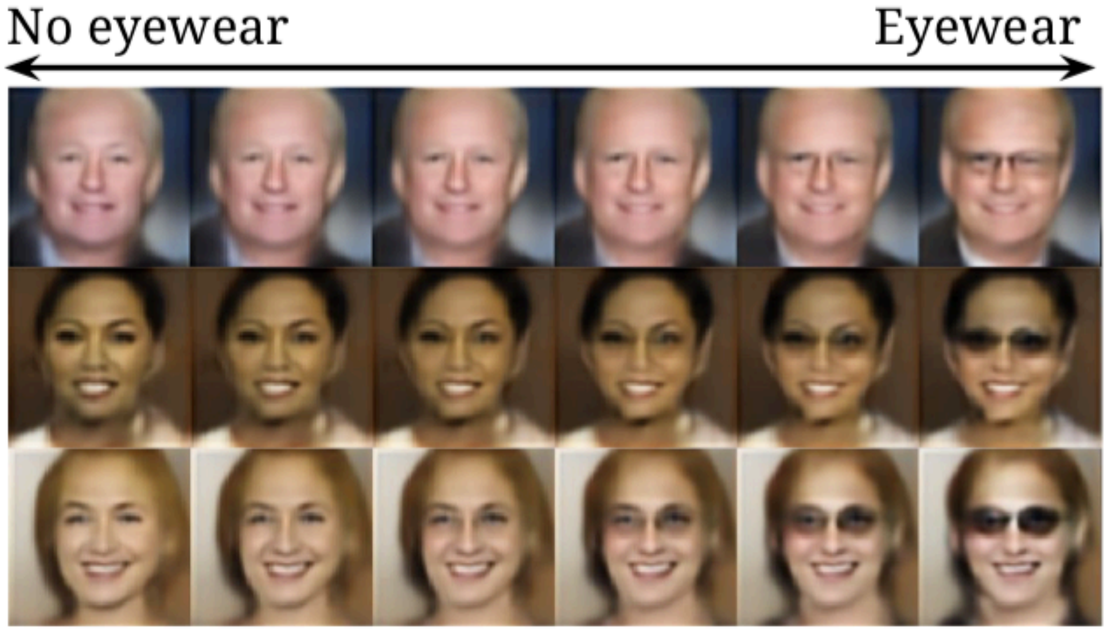
- We now have a known and compact distribution for $\mathbf{h}$
  - We know where to sample from
  - We additionally minimize gaps in the latent space

- The sampling process creates a more efficient encoder
  - Noise in the process forces the same data point to map to an entire region, and not a single point (we avoid overfitting)

Conditional VAEs

- If we also have labels for our data we can use them
  - Provide label as additional input to the encoder/decoder
  - This allows us to specify elements of data to generate

- E.g. Attribute2Image model
  - Learn faces with attributes
  - Sample and impose constraints



VAEs in action

- Popular for generating images
  - Generate new images from old
  - Generate new images with custom attributes (e.g. age, gender, …)
- Also used for making music
  - Interpolate between melodies
  - Generate new sequences from old
    - Can use RNNs in VAE model too!

Pros/cons of VAEs

- Pros:
  - Very powerful and relatively easy to work with
  - A “deep PCA” alternative to move data to a low-dim space
  - Uses a compact and manageable latent space
- Cons:
  - They tend to produce “smoothed” outputs
    - e.g. blurry images
  - You need to carefully balance reconstruction vs. KL in your loss

An alternative approach

- Use a neural network to teach a neural network
  - Avoid selecting a specific cost function
  - Instead of comparing inputs/outputs, use a *critic*

- Why is this good?
  - We don’t have to use a simplistic measure (e.g. L2 reconstruction)
  - We can have a constantly evolving model

Generative Adversarial Networks (GANs)

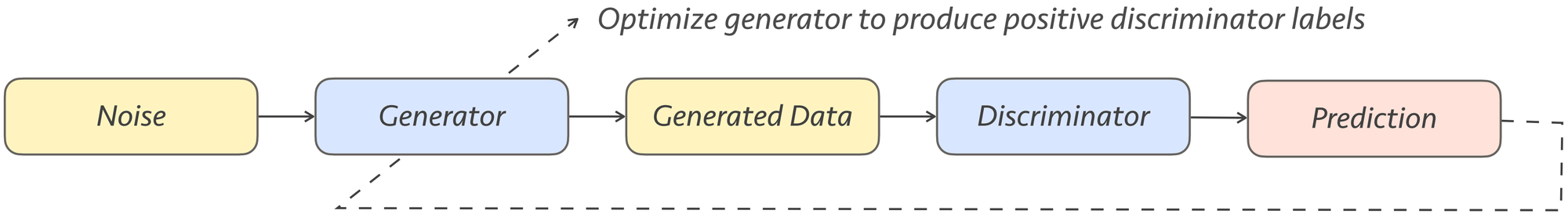
- GANs are composed out of two networks
  - The *Generator*, creates new data points from noise input
  - The *Discriminator*, rates data points as generated or real

$$\hat{\mathbf{x}} = \mathrm{G}(\mathbf{z} \sim \mathcal{N}), \quad \mathrm{P}(\mathbf{x} \text{ is real}) = \mathrm{D}(\mathbf{x})$$

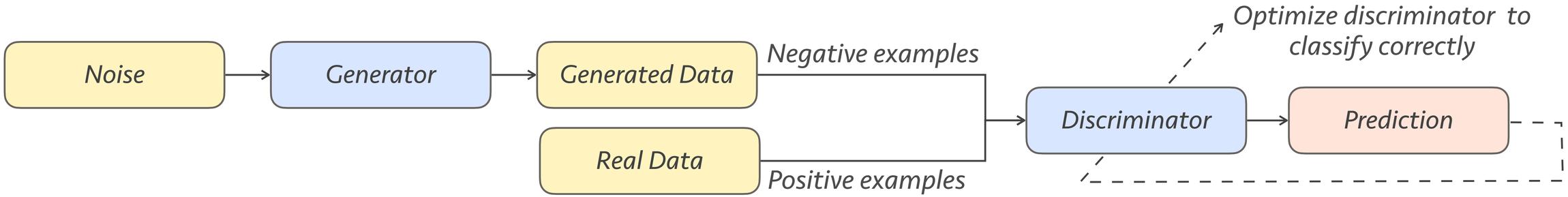
- Adversarial training process
  - Train the generator such that it fools the discriminator
  - Train the discriminator to detect generator's data
    - Through this process they both constantly improve each other

Training a GAN

- Train the generator to fool the discriminator
  - Improves the quality of generated data



- Train the discriminator to detect generated data
  - Makes the discriminator pickier, which will improve generator



How to set this up

- Original formulation, minimax setup

$$\text{for D maximize } \mathrm{E}_{\mathbf{z}}\left\{\log\left[1-\mathrm{D}\left(\mathrm{G}(\mathbf{z})\right)\right]\right\}+\mathrm{E}_{\mathbf{x}}\left\{\log\left[\mathrm{D}(\mathbf{x})\right]\right\}$$

$$\text{for G minimize } \mathrm{E}_{\mathbf{z}}\left\{\log\left[1-\mathrm{D}\left(\mathrm{G}(\mathbf{z})\right)\right]\right\}$$

- However this is very unstable!!
  - Discriminator gradients will saturate when it does well
    - This will effectively freeze the training process

Improving the gradients

- Simplify the costs
  - We can replace $\mathrm{E}_{\mathbf{x}}\{\log[1-\mathrm{D}(\mathbf{x})]\}$ with $-\mathrm{E}_{\mathbf{x}}\{\log[\mathrm{D}(\mathbf{x})]\}$
    - A little better numerically, but still discriminator can saturate
- Wasserstein GANs
  - Replace discriminator cost with $\mathrm{E}_{\mathbf{x}}\{\pm\mathrm{D}(\mathbf{x})\}$ and remove sigmoids on output
    - This creates an unbounded output which won't saturate
- Least-Squares (LS) GANs
  - Replace loss with $\mathrm{E}_{\mathbf{x}}\{(\mathrm{D}(\mathbf{x})-t)^2\}$, $t \in [0,1]$
    - Will not diverge to large values as WGANs tend to

Some GAN problems

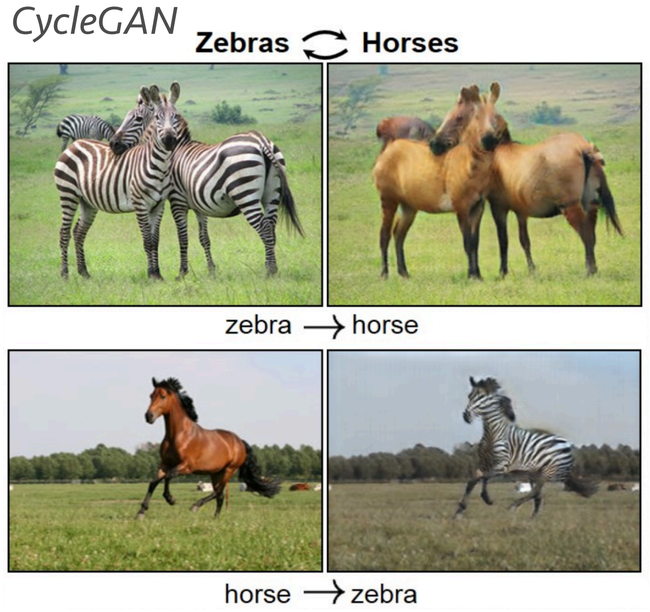
- Generated data are not guaranteed to be right
  - They can fool the discriminator, but it doesn't mean they are good
- The dreaded *mode collapse*
  - The generator can always produce just one passable data point
    - There is no guarantee of having variance in the generated data
- We do not get a latent space we can project to
- Training is an absolute mess to get right!

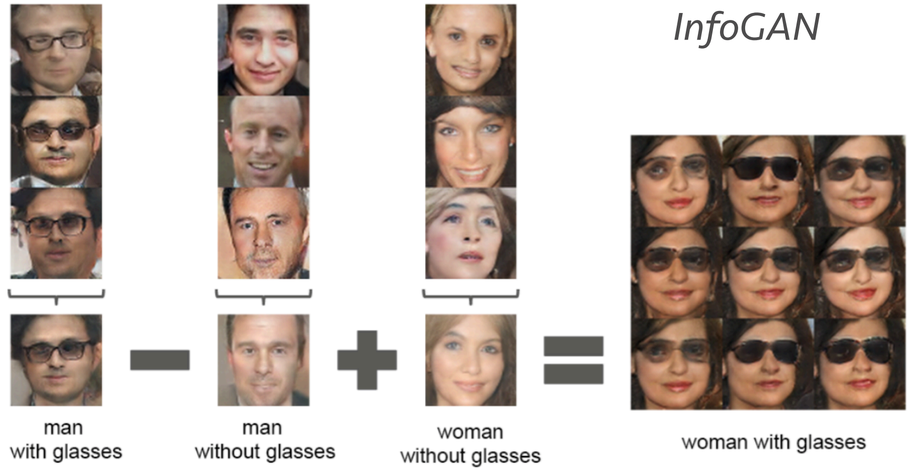
GANs in action

- Very popular in vision/graphics
  - E.g. fake celebrity face generation
- Success in other domains too
  - Very useful when we cannot easily describe the data we want to make

Many elaborations

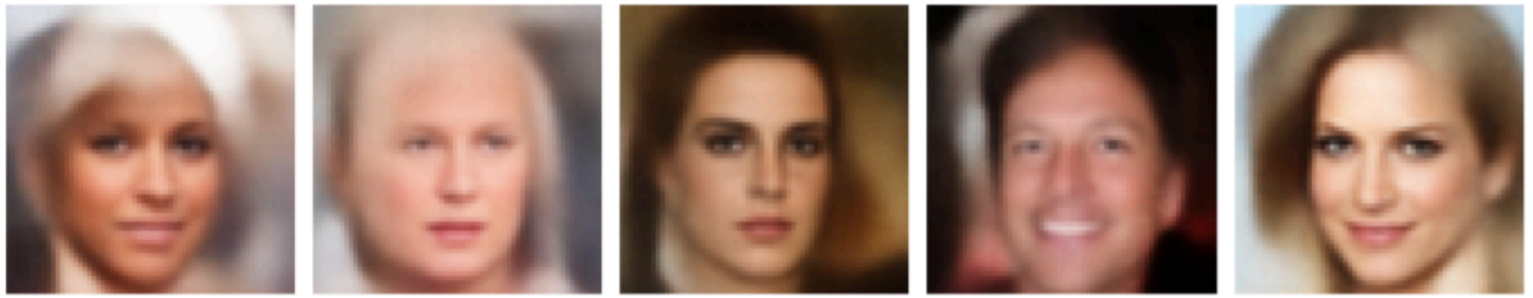
- CycleGAN
  - Make an autoencoder, provide additional output to a discriminator of other style
  - Then run it backwards
- InfoGAN
  - Force part of representation to use a latent code that adds information
- Many more …





VAEs or GANs? *(Baseline comparison)*

- VAEs are known to produce “fuzzier” outputs



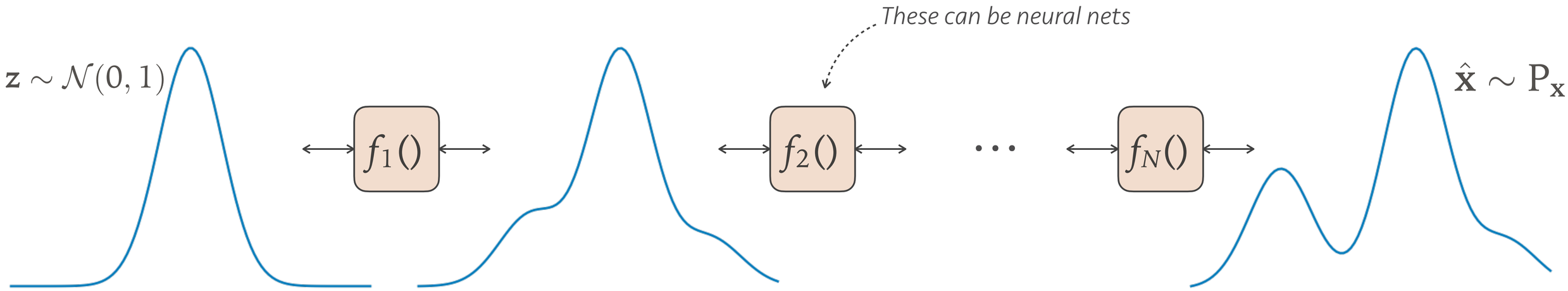
- GANs are known for sharper (but weirder) outputs

Normalizing Flows

- An alternative approach similar to the VAE
  - We still have a simply distributed latent state
    - How we go back and forth from data space is different though
- Core ideas:
  - Use a series of *bijective* transformations as the encoder/decoder
  - Ensure that the transformations are “nice” enough to be "useful”
    - “Nice” = mathematically tractable
    - “Useful” = we can deduce something out of them

Basic flow setup

- A sequence of invertible transforms
  - Start with a known (easy) distribution, move to desired one
  - Make sure we can go the opposite direction as well!
- Effectively we learn a bidirectional encoder/decoder



But this is more than just multiple layers

- Transforming a random variable:

$$\text{if } \mathbf{z} = f(\mathbf{x}) \text{ then } P(\mathbf{z}) = P(\mathbf{x}) \left| \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} \right| \text{ and } P(\mathbf{x}) = P(\mathbf{z}) \left| \frac{\partial f^{-1}(\mathbf{x})}{\partial \mathbf{x}} \right|$$

- As long as $f()$ is bijective, we can use it in a flow
  - Nice perk: we can transform any distribution to another
    - e.g. generate in a simple space (Gaussian) and move to a more complex one (speech, images, etc.)

Putting it all together

- For a sequence of transforms $f_i()$ we will have:

$$P(\mathbf{x}) = P(\mathbf{z}) \prod_{i}^{K} \left| \mathbf{J}_{f_i}(\mathbf{z}_{i-1}) \right|^{-1}$$

We use $\mathbf{J}$ *for the Jacobian matrix:* $\mathbf{J}_{ij} = \partial f(x_i) / \partial x_j$
The product accounts for the successive transforms.
Instead of inverting $f()$ *or* $\mathbf{J}$*, we invert the determinant*

- And the likelihood of $\mathbf{x}$ for $\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$

$$\log P(\mathbf{x}) = \log \mathcal{N}(\mathbf{z}|0, \mathbf{I}) - \sum_{i}^{K} \log \left| \mathbf{J}_{f_i}(\mathbf{z}_{i-1}) \right|$$

That's just an MSE between 0 *and* $f^{-1}(\mathbf{x})$

RealNVP flows

- Good baseline flow demonstrating suitable $f_i()$'s
  - Split each sample into two parts (e.g. top and bottom) $\mathbf{x} = \begin{bmatrix} \mathbf{x}_a \\ \mathbf{x}_b \end{bmatrix}$
  - Apply a *coupling* layer:

$$\mathbf{y} = \begin{bmatrix} \mathbf{x}_a \\ \mathbf{x}_b \cdot S(\mathbf{x}_a) + T(\mathbf{x}_a) \end{bmatrix}$$

    - Where S and T are *scaling* and *translation* neural nets
  - Then apply a *permutation* layer
    - Just swap the top and bottom half of the input
    - Ensures that we don't pass any data without any processing at all

Why is this a good idea?

- Resulting transformation is invertible and “easy”
  - Jacobian determinant of permutation layers is 1
  - Jacobian determinant of coupling layers is simple:

$$\mathbf{J} = \sum_{i}^{D/2} S(\mathbf{x}_a)_i$$

- And overall transform is still very flexible
  - Can map complex distributed data to, e.g., a Gaussian

Pros and cons of Flows?

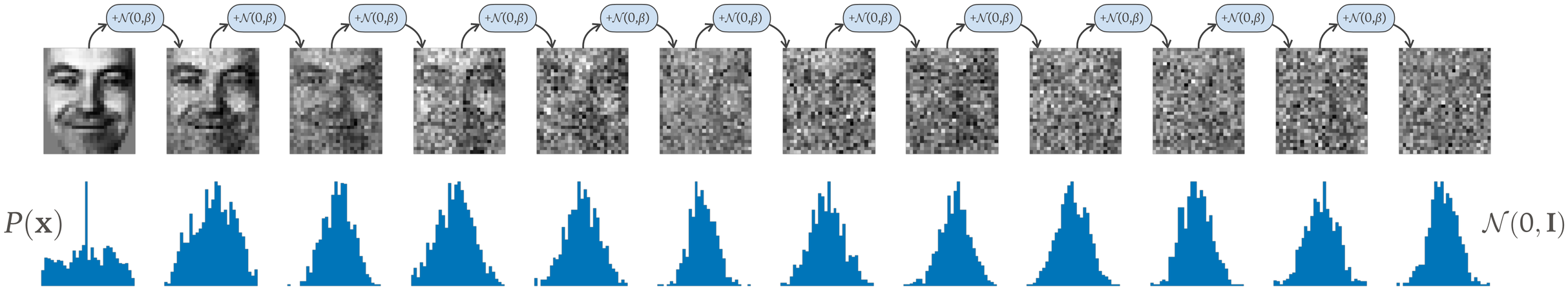
- Pros:
  - You can calculate exact likelihoods through this mapping
    - So you can precisely evaluate new data easily
  - You can move between data and latent space easily
  - They scale well with more data
- Cons:
  - Might need a lot of transforms (more compute/memory)
  - They can employ only limited types of transformations

Diffusion Models

- Today's darling generative model *(for now)*
  - Used widely for image/audio/video generation
- Yet another model that maps noise to data space
- Can be thought of as a variation of previous models
  - As a hierarchical VAE with a deeper pipeline
  - As a flow-like model with multiple repeating steps

Diffusion forward pass

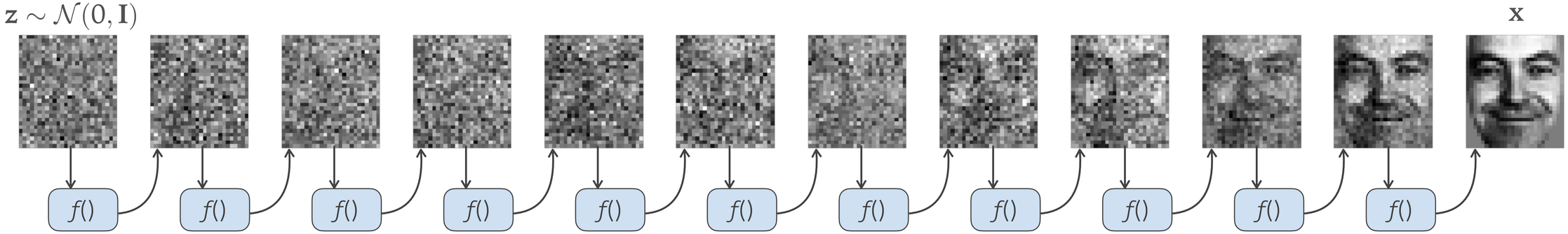
- Sequentially transform data towards noise



  - Do so by progressively adding noise to previous output
    - End state will eventually converge to Gaussian distributed data
    - Which will be our “latent” representation

Diffusion backward pass

- Train neural nets to denoise and reverse forward pass



- This builds a “decoder” of sorts
  - Takes us from a Gaussian sample to a data sample
  - Many variations on this idea
    - Predict cleaner samples, predict the noise, use one or many networks, …

Training it

- Adding Gaussian noise in each step is smart
  - In forward pass we can predict any step directly
    - Adding multiple noises collapses to adding noise once!
  - Involved (Gaussian) likelihoods simplify to MSE losses!

- Training simply uses random pairs from adjacent steps
  - Train a “denoiser”, which is often aware of the step index too
  - The architecture of that net depends on your data structure

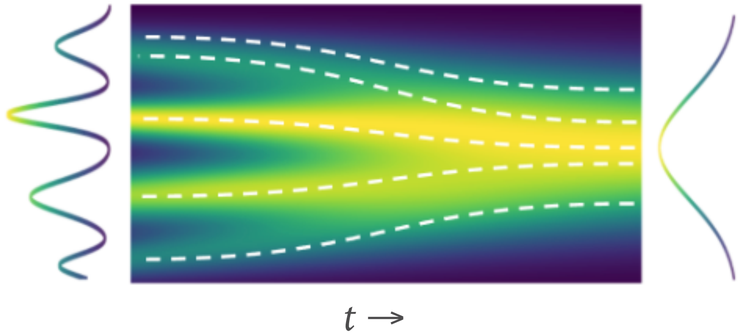
Generating data

- We generate a Gaussian sample and do backward pass
  - This produces an output sample in the data space
- Why is it better than a VAE?
  - Compared to a deeper VAE, it has a “guided” path to the output
- Major downside
  - We need multiple (1000s!) neural net passes to produce a sample!
    - Remember that dimensionality is the same as the data! (Yikes!)

An interesting connection

- Continuous-time Normalizing Flows
  - Using ODEs to model distribution transforms

$$\frac{dy}{dt} = f(y, t, \theta)$$



    - Which creates smooth interpolations between distributions!
- Diffusion steps can be seen a discretization of this!
  - We can use ODE tools to solve it faster/denser

So which model should you use?

- You know the answer, it depends …

	Training	Likelihood	Sampling	Invertible	Low-D latent	Output
AR	😀	–	🐌	–	–	High-quality
GAN	😱	No	😀	No	Yes-ish	High-quality (If you get it to work)
VAE	😀	Approximate	😀	No	Yes	Good
Flows	🙂	Exact	😀	Yes	No	Good
Diffusion	🙂	Approximate	😬	No	No	High-quality

** Based on baseline versions, tweaks with improved characteristics come out weekly*

The generative data revolution

- So what do we need to make a fancy generative model?
  - A low-dimensional latent representation
    - Usually a pre-trained VAE on your data
  - A generative model in the latent space
    - Usually diffusion, which is manageable in low-dim
  - A post-processing model to add the details
    - E.g. image upsampling, phase fill-in, etc.
- Which part makes them work? Not just one …

Important:
The network topologies within these parts will make or break a system!

Recap

- Generative models
  - Auto-Regressive Models
  - Variational Auto-Encoders
  - Adversarial Generative Models
  - Flow Models
  - Diffusion Models

- Nice book on the subject (with code!):
  - https://link.springer.com/book/10.1007/978-3-030-93158-2

Download it for free through our library

Next Lecture

- Graph Signal Processing and Graph Networks
  - What to do when dealing with irregular data formats