CENG501 – Deep Learning

Week 13

Fall 2024

Sinan Kalkan

Dept. of Computer Engineering, METU

Previously on CENG501

Taxonomy of Generative Models

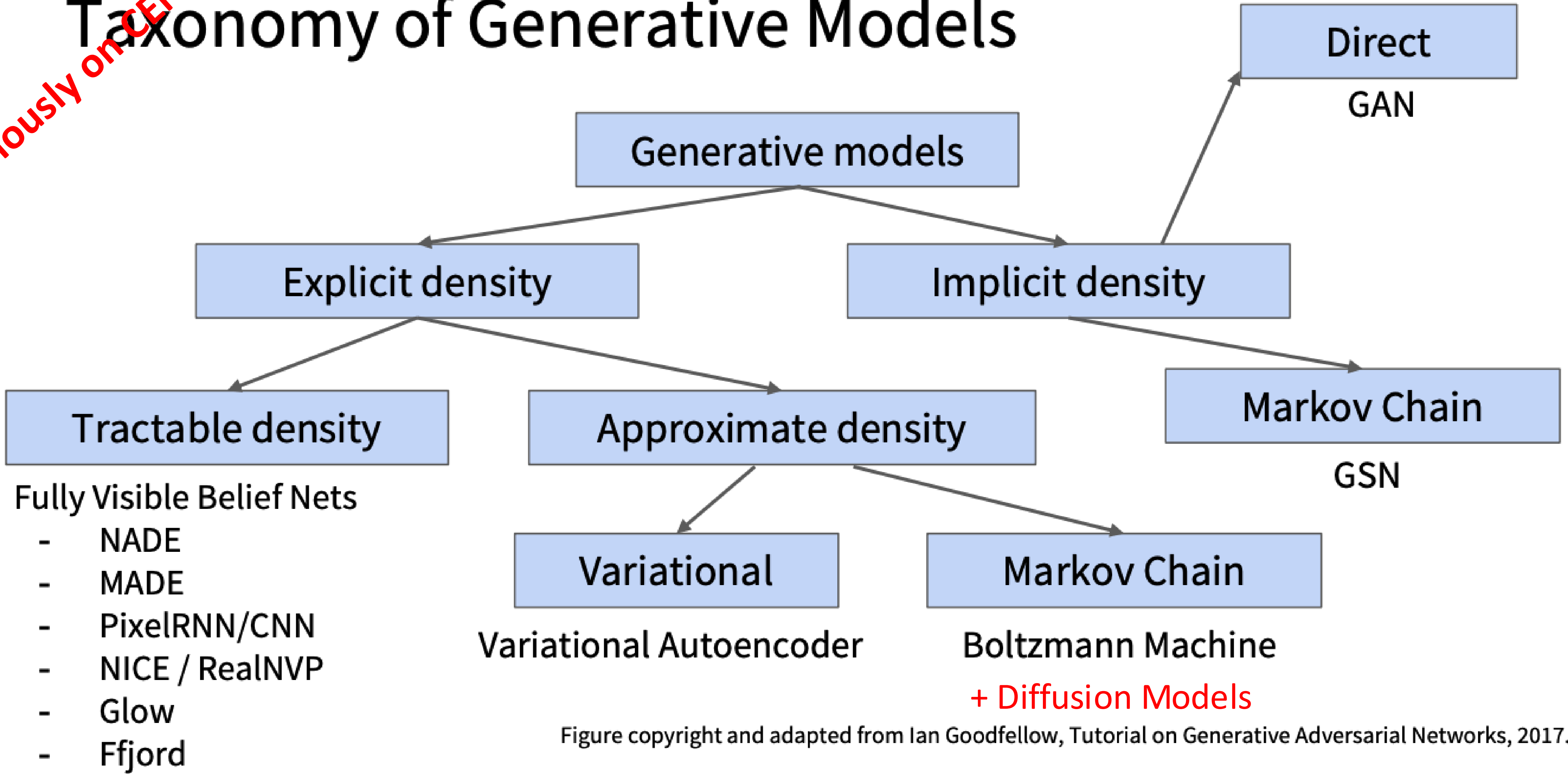


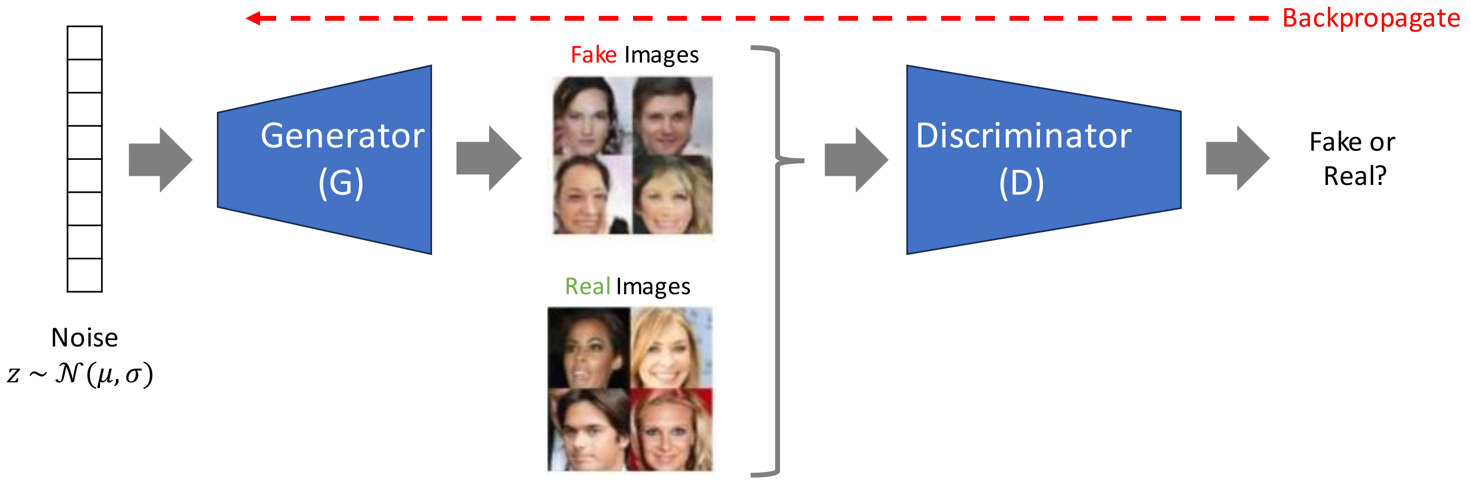
Figure copyright and adapted from Ian Goodfellow, Tutorial on Generative Adversarial Networks, 2017.

2024

Slide: https://cs231n.stanford.edu/slides/2024/lecture_13.pdf

Generative Adversarial Networks (GANs)

Previously on CENG501



- With two competing networks, we solve the following minimax game:

$$\min_G \max_D V(D,G) = E_{x \sim p_{\text{data}}(x)}[\log D(x)] + E_{z \sim p_z(z)}\left[\log\left(1 - D(G(z))\right)\right]$$

- Discriminator's objective:

$$\max_D V(D,G) = E_{x \sim p_{\text{data}}(x)}[\log D(x)] + E_{z \sim p_z(z)}\left[\log\left(1 - D(G(z))\right)\right]$$

- Generator's objective:

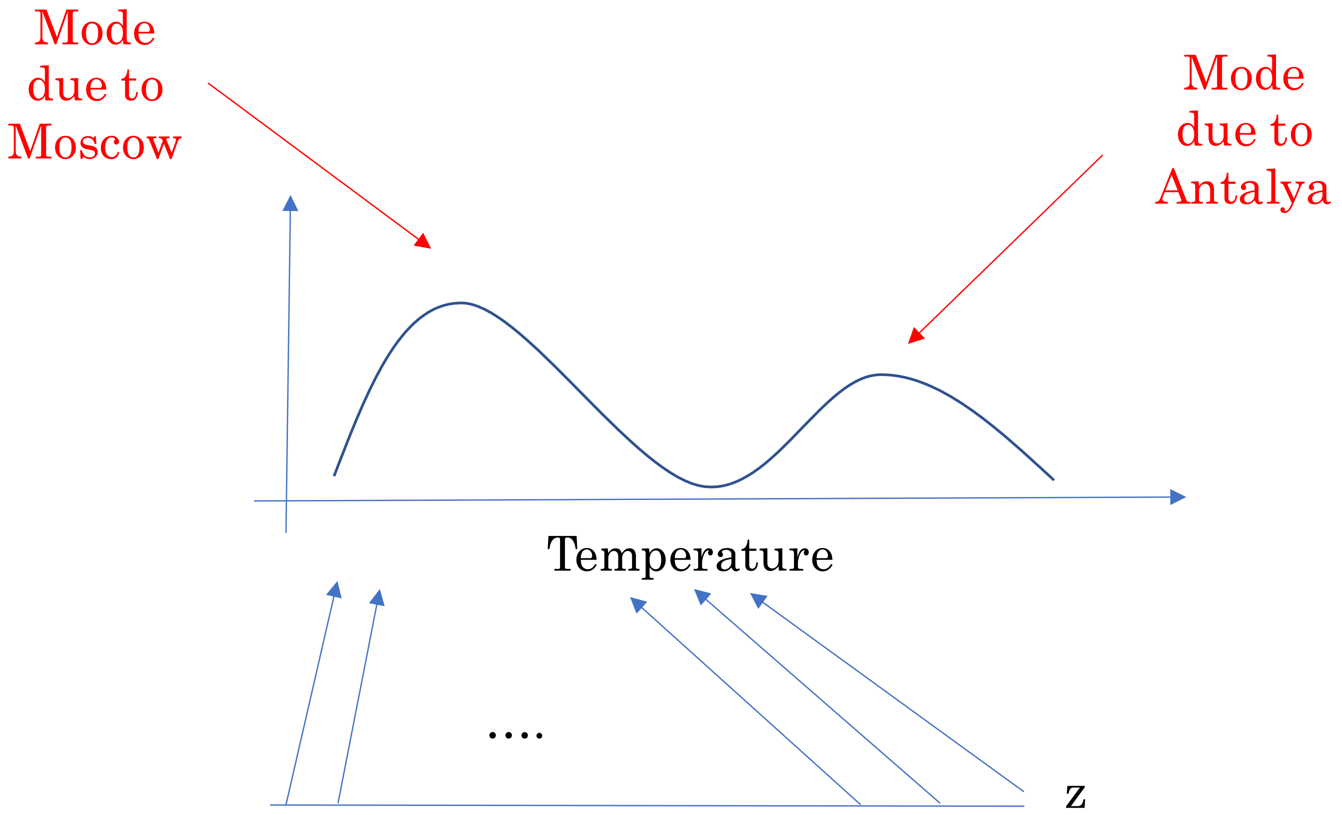
$$\min_G V(D,G) = E_{z \sim p_z(z)}\left[\log\left(1 - D(G(z))\right)\right]$$

$D(x)$: Probability that x is real (came from data).

Adapted from: http://guimperarnau.com/blog/2017/03/Fantastic-GANs-and-where-to-find-them

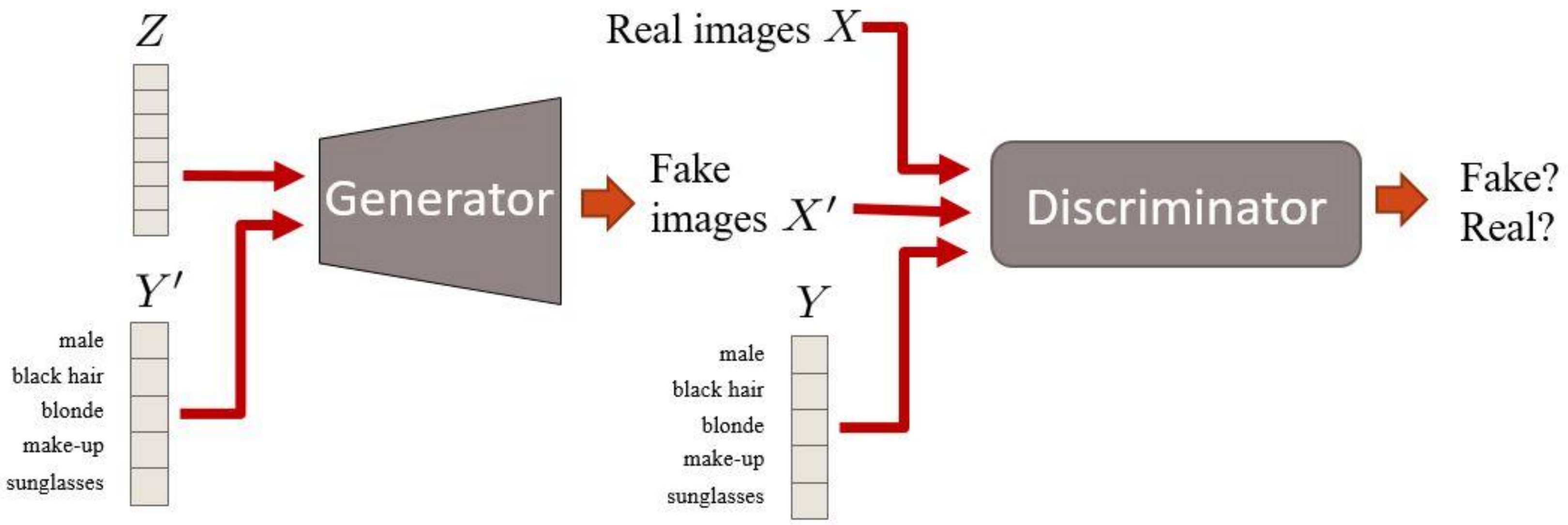
Mode collapse in GANs

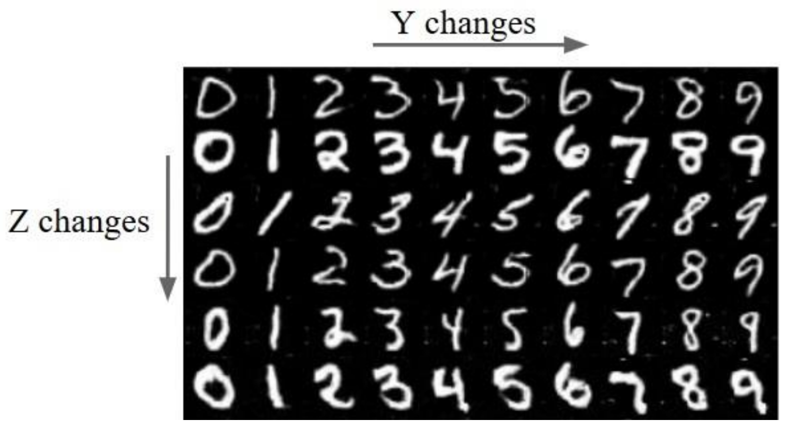
- Problem:
  - The generator network maps the different z (embedding/noise) values into similar images.



Conditional GANs

Previously on CENG501





2024

http://guimperarnau.com/blog/2017/03/Fantastic-GANs-and-where-to-find-them

Cycle GAN

Previously on CENG501

Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks

Jun-Yan Zhu* Taesung Park* Phillip Isola Alexei A. Efros
Berkeley AI Research (BAIR) laboratory, UC Berkeley

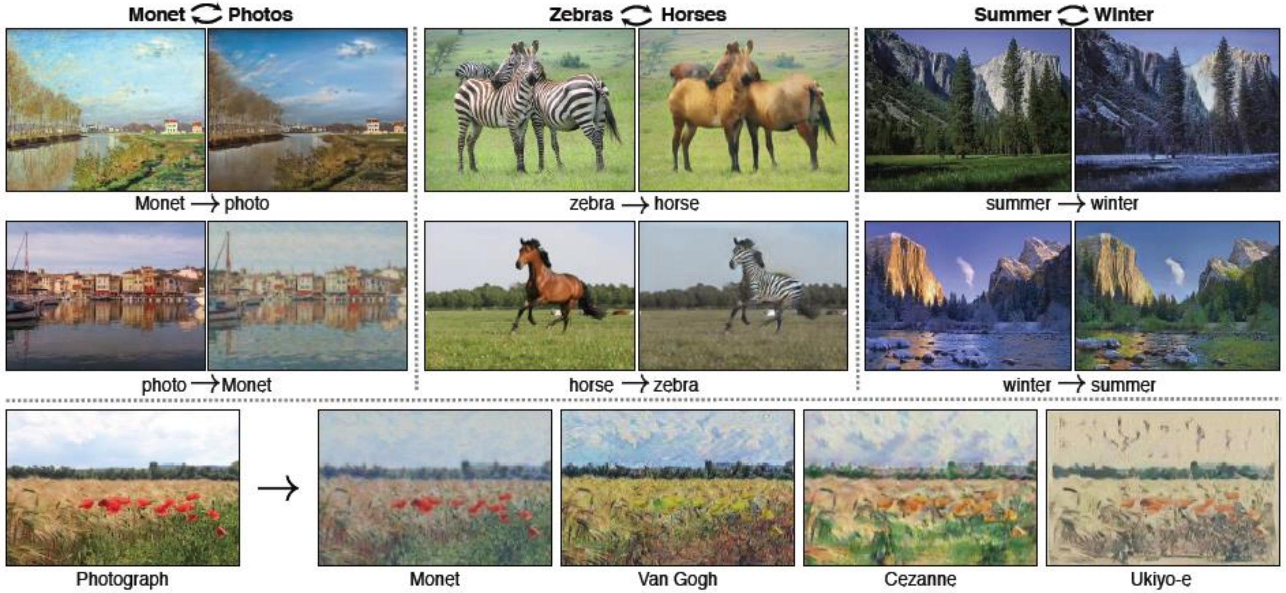
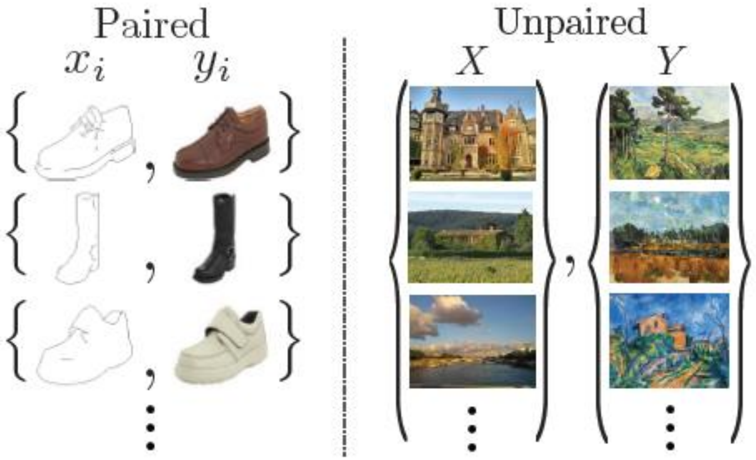


Figure 1: Given any two unordered image collections X and Y, our algorithm learns to automatically "translate" an image from one into the other and vice versa: *(left)* 1074 Monet paintings and 6753 landscape photos from Flickr; *(center)* 1177 zebras and 939 horses from ImageNet; *(right)* 1273 summer and 854 winter Yosemite photos from Flickr. Example application *(bottom)*: using a collection of paintings of a famous artist, learn to render a user's photograph into their style.



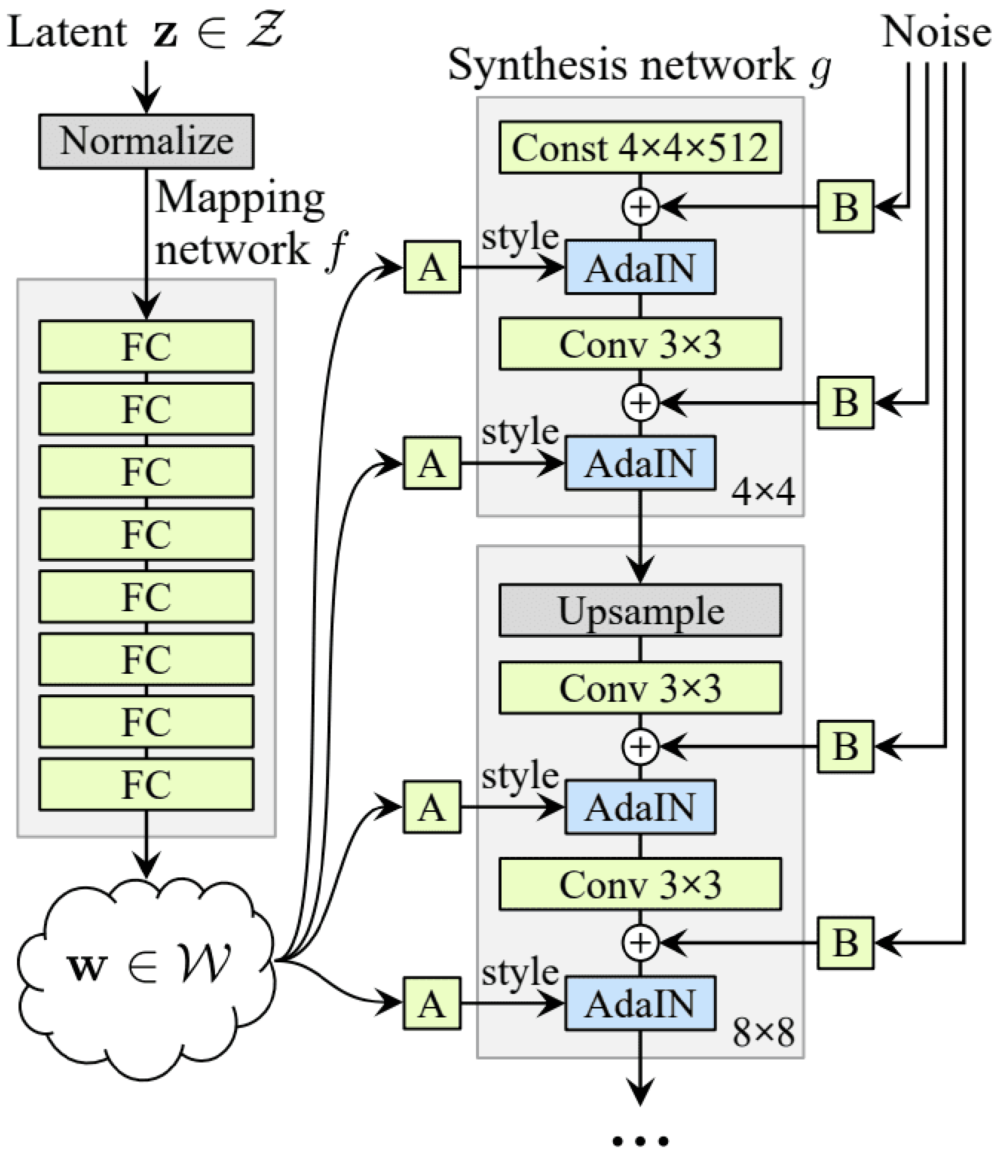
https://junyanz.github.io/CycleGAN/

GAN – state of the art

Previously on CENG501

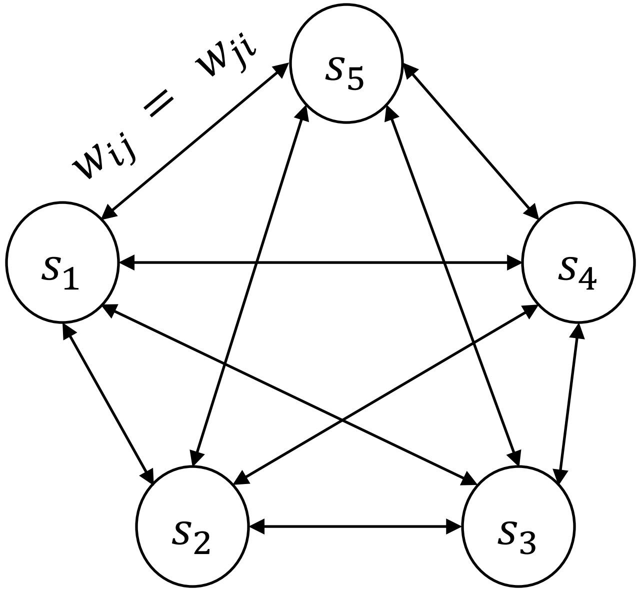
$$\text{AdaIN}(\mathbf{x}_i, \mathbf{y}) = \mathbf{y}_{s,i} \frac{\mathbf{x}_i - \mu(\mathbf{x}_i)}{\sigma(\mathbf{x}_i)} + \mathbf{y}_{b,i},$$

https://github.com/NVlabs/stylegan2



Previously on CENG501

Hopfield Networks



- $s_i = -1 \text{ or } +1$
- Then,

$$s_i \leftarrow \begin{cases} +1, & \sum_j w_{ij} s_j \geq \theta_i \\ -1, & \text{otherwise} \end{cases}$$

- θ_i: threshold of neuron i. Mostly we set this to zero.
- In short:

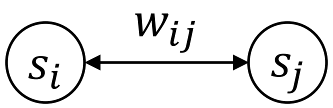
$$s_i = \text{sgn}\left(\left[\sum_j w_{ij} s_j\right] - \theta_i\right)$$

Previously on CENG501

Hopfield Networks: An Energy Perspective

- We can define a scalar for the energy of the state of the network:

$$E = -\sum_i \sum_{j<i} w_{ij} s_i s_j + \sum_i \theta_i s_i$$



- This is called energy since when you update neurons randomly, it either decreases or stays the same.
- Repeatedly updating the network will eventually make the network converge to a local minimum, i.e., a stable state.

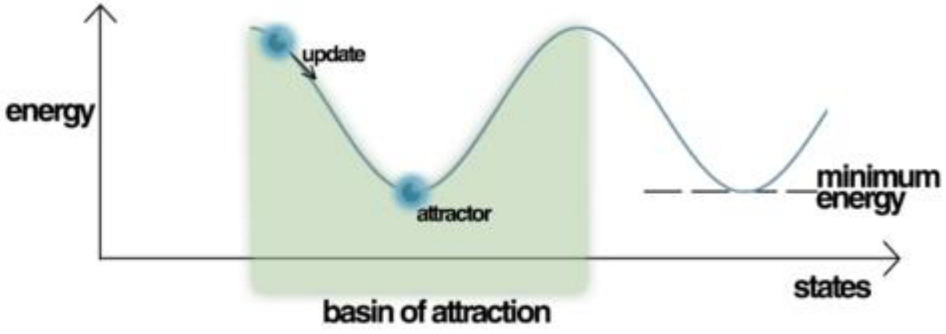
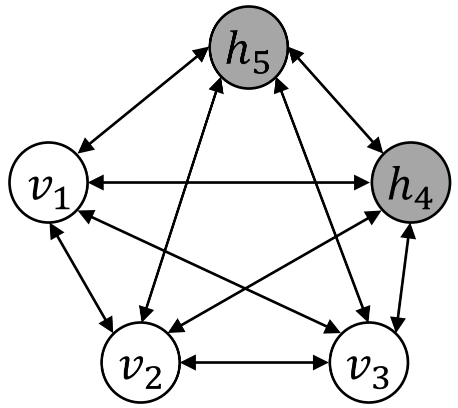


Fig: Wikipedia

Previously on CENG501

Boltzmann Machines vs. Hopefield Networks



- They have the same energy definition ($\mathbf{s} = \{v_m\} \cup \{h_n\}$):

$$E(\mathbf{s}) = -\sum_i \sum_{j<i} w_{ij} s_i s_j + \sum_i \theta_i s_i$$

Differences:

- Updates are stochastic
- We have hidden neurons now
  - Hidden variables ➔ Bigger class of distributions that can be modeled ➔ In principle, we can model distributions of arbitrary complexity

Boltzmann Machines: Probability of a Neuron's State

- Turning on a neuron i (i.e., s_i is changed to 1 from 0) causes change ΔE_i in energy:

$$\Delta E_i = E_{i=0} - E_{i=1}$$
$$= -kT\ln(Z\,p_{i=0}) - (-kT\ln(Z\,p_{i=1}))$$
$$= -kT\ln\left(\frac{Z\,p_{i=0}}{Z\,p_{i=1}}\right) = -kT\ln\left(\frac{p_{i=0}}{p_{i=1}}\right)$$
$$= -kT\ln\left(\frac{1-p_{i=1}}{p_{i=1}}\right)$$

Using:
$$p_i = \frac{e^{-E_i/kT}}{Z}$$

- This yields the famous logistic / sigmoid function:

$$p_{i=1} = \frac{1}{1+\exp\left(-\frac{\Delta E_i}{\mathrm{T}}\right)}$$

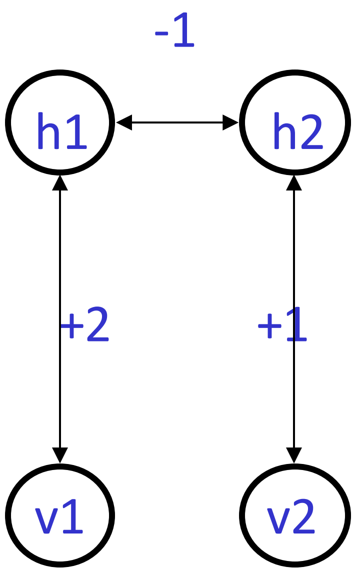
- $\Delta E_i > 0$ => Energy is reduced => High $p_{i=1}$
- $\Delta E_i < 0$ => Energy is increased => Low $p_{i=1}$

Previously on CENG501

Boltzmann Machines: An Example

v	h	$-E$	e^{-E}	$p(\mathbf{v},\mathbf{h})$	$p(\mathbf{v})$
1 1	1 1	2	7.39	.186	0.466
1 1	1 0	2	7.39	.186	
1 1	0 1	1	2.72	.069	
1 1	0 0	0	1	.025	
1 0	1 1	1	2.72	.069	0.305
1 0	1 0	2	7.39	.186	
1 0	0 1	0	1	.025	
1 0	0 0	0	1	.025	
0 1	1 1	0	1	.025	0.144
0 1	1 0	0	1	.025	
0 1	0 1	1	2.72	.069	
0 1	0 0	0	1	.025	
0 0	1 1	-1	0.37	.009	0.084
0 0	1 0	0	1	.025	
0 0	0 1	0	1	.025	
0 0	0 0	0	1	.025	

total = 39.70



Adapted from G. Hinton

Today

- (Deep) Generative Models
  - Diffusion Models
- Self-Supervised Learning
- Deep Reinforcement Learning

CENG796 DEEP GENERATIVE MODELS

Course Code:	5710796
METU Credit (Theoretical-Laboratory hours/week):	3(3-0)
ECTS Credit:	8.0
Department:	Computer Engineering
Language of Instruction:	English
Level of Study:	Graduate
Course Coordinator:	Assoc.Prof.Dr. RAMAZAN GÖKBERK CİNBİŞ
Offered Semester:	Fall Semesters.

Course Objectives

At the end of the course, the students will be expected to:

- Comprehend a variety of deep generative models.
- Apply deep generative models to several problems.
- Know the open issues in learning deep generative models, and have a grasp of the current research directions.

Course Content

Deep generative modeling with Autoregressive models; Energy-based models; Adversarial models; Variational models.

Administrative Notes

- No quiz this week

- Time plan for the projects
  1. Milestone (November 24, midnight):
     - Github repo will be ready
     - Read & understand the paper
     - Download the datasets
     - Prepare the Readme file excluding the results & conclusion
  2. Milestone (December 8, midnight)
     - The results of the first experiment
  3. Milestone (January ~~5~~ 12, midnight)
     - Final report (Readme file)
     - Repo with all code & trained models

Diffusion-based Generative Models

2024

ELBO Recap

Why use ELBO?

Directly maximizing $p(x)$ is very difficult:

- it involves either marginalizing over the entire latent space $\boldsymbol{Z}$ (intractable for complex models) OR
- It involves having access to the ground truth latent encoder $p(z|x)$

ELBO:

$$\log(p(x)) \geq \mathbb{E}_{q_\phi(z \mid x)}\left[\log \frac{p(x,z)}{q_\phi(z|x)}\right]$$

Question: Why does the $\geq$ show up here? → With the derivation in the appendix, we see a $D_{KL}(q_\phi(z|x) \,||\, p(z|x))$ term show up which is always ≥ 0.

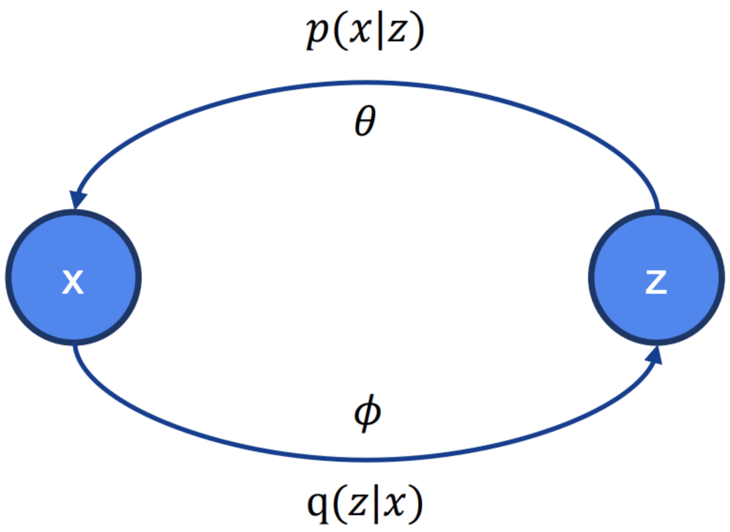
Applying chain-rule of probabilities:

$$ELBO = \mathbb{E}_{q_\phi(z \mid x)}[\log p_\theta(x|z)] - D_{KL}\big(q_\phi(z \mid x)\big|\big|p(z)\big)$$

Reconstruction   Prior matching

Slide: https://deeplearning.cs.cmu.edu/F23/document/slides/lec23.diffusion.updated.pdf

Variational Autoencoder Recap



Latent variable sampling: $z \sim \mathcal{N}(z;\ \mu_\phi(x), \sigma_\phi^2(x))$

Reparameterization trick: $z = \mu_\phi(x) + \sigma_\phi(x) \odot \epsilon,\ \epsilon \sim \mathcal{N}(0, I)$

Training:

- Jointly optimize θ and ϕ
- Maximize **ELBO**

Empirically, we found that two things make VAEs work really well:

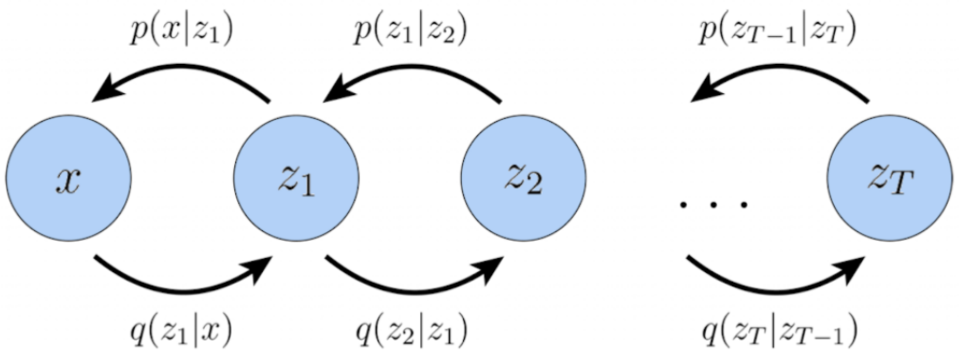
1. Increasing the depth of the networks
2. Introducing a hierarchy of latent variables (latent variables of latent variables)

$x \leftarrow z_1 \leftarrow z_2 \leftarrow \ \dots \leftarrow z_T$, such that each latent is conditioned on all previous latents.

We are particularly interested in such HAVEs that where the process is a **Markovian chain - MHVAE**

Slide: https://deeplearning.cs.cmu.edu/F23/document/slides/lec23.diffusion.updated.pdf

Markovian Hierarchical Variational Autoencoder

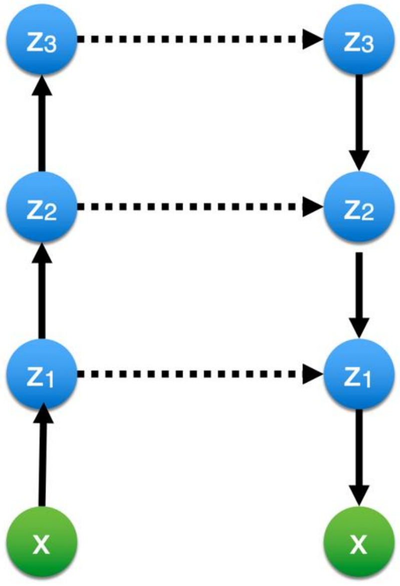


Joint probability: $p(x, z_{1:T}) = p(z_T)p_\theta(x \mid z_1) \prod_{t=2}^{T} p_\theta(z_{t-1}|z_t)$

Posterior probability: $q_\phi(z_{1:T} \mid x) = q_\phi(z_1| x) \prod_{t=2}^{T} q_\phi(z_t|z_{t-1})$

Updated ELBO:

$$\log(p(x)) \geq \mathbb{E}_{q_\phi(z_{1:T} \mid x)}\left[\log \frac{p(x, z_{1:T})}{q_\phi(z_{1:T} \mid x)}\right]$$

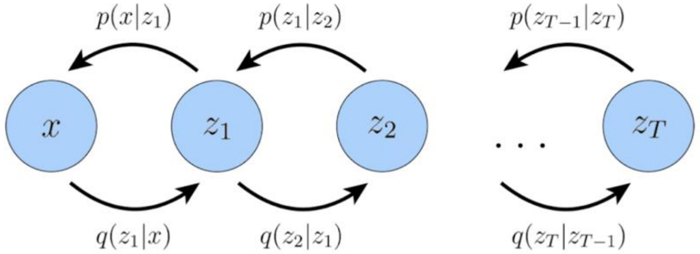


Inference model **q(z|x)** Generative model **p(x,z)**

Fig: https://cs231n.stanford.edu/slides/2023/lecture_15.pdf

Slide: https://deeplearning.cs.cmu.edu/F23/document/slides/lec23.diffusion.updated.pdf

Diffusion Models



Diffusion models are essentially **MHVAEs** with **3 restrictions:**

1. Latent dimension is the same as the data dimension
2. The encoder has no parameters to be learnt. It is defined to be a linear gaussian such that the t^{th} gaussian is centered around the previous latent z_{t-1}
3. The parameters for the gaussians are scheduled such that the final latent is a standard gaussian.

$$z_T \sim \mathcal{N}(z_T;\ 0, \boldsymbol{I})$$

Slide: https://deeplearning.cs.cmu.edu/F23/document/slides/lec23.diffusion.updated.pdf

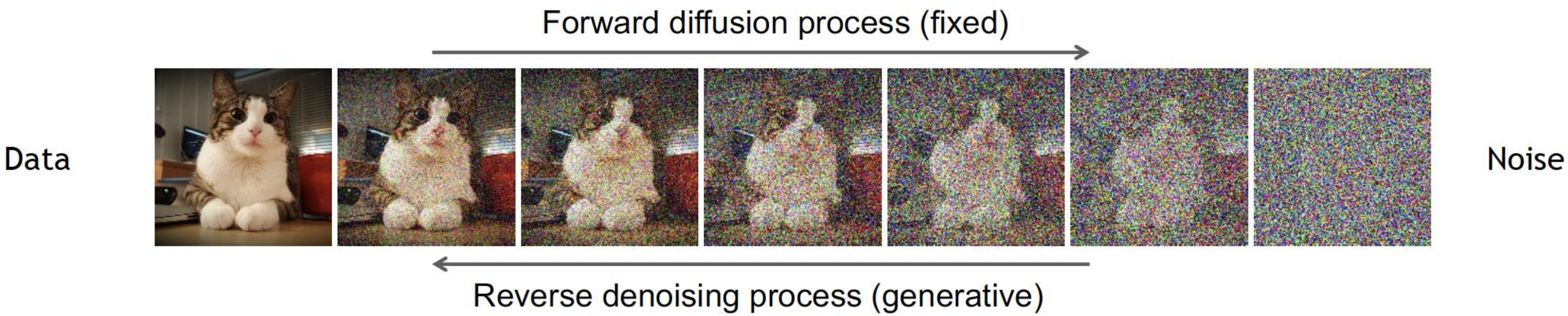
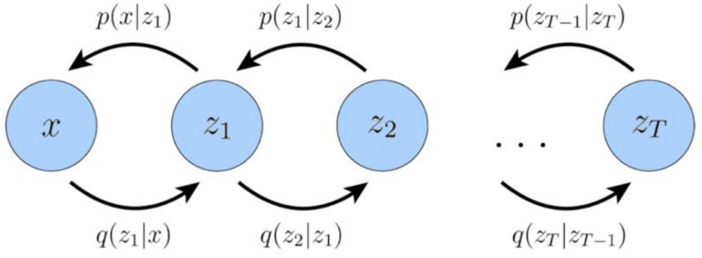


Fig: https://cvpr2022-tutorial-diffusion-models.github.io/

Diffusion Models



Diffusion models are essentially **MHVAEs** with **3 restrictions:**

1. Latent dimension is the same as the data dimension
2. The encoder has no parameters to be learnt. It is defined to be a linear gaussian such that the t^{th} gaussian is centered around the previous latent z_{t-1}
3. The parameters for the gaussians are scheduled such that the final latent is a standard gaussian.

$$z_T \sim \mathcal{N}(z_T;\ 0, \boldsymbol{I})$$

The first restriction allows for some mild abuse of notation:

$$q_\phi(x_{1:T} \mid x_0) = \prod_{t=1}^{T} q_\phi(x_t|x_{t-1})$$

(We are using x instead of z)

$$p(x_{0:T}) = p(x_T) \prod_{t=1}^{T} p_\theta(x_{t-1}|x_t)$$

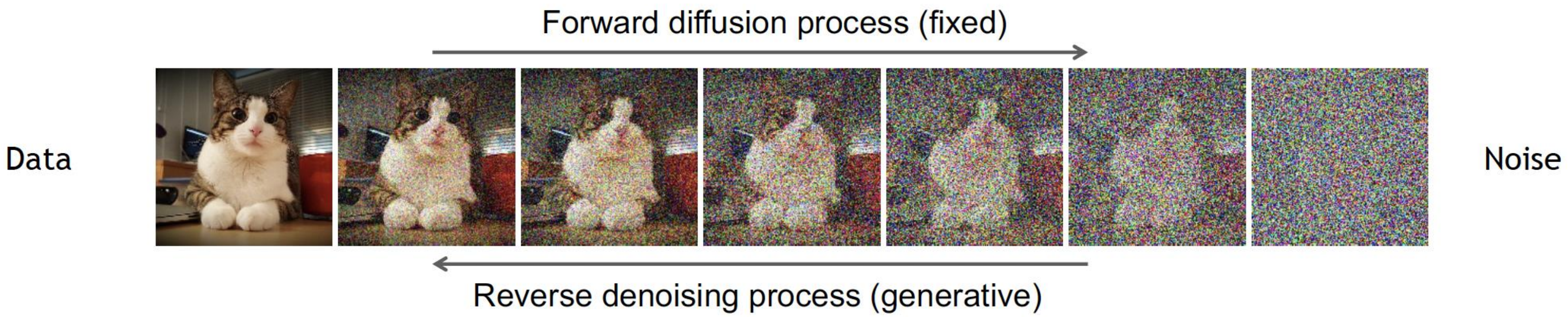
Slide: https://deeplearning.cs.cmu.edu/F23/document/slides/lec23.diffusion.updated.pdf

Denoising Diffusion Models

Learning to generate by denoising

Denoising diffusion models consist of two processes:

- Forward diffusion process that gradually adds noise to input
- Reverse denoising process that learns to generate data by denoising



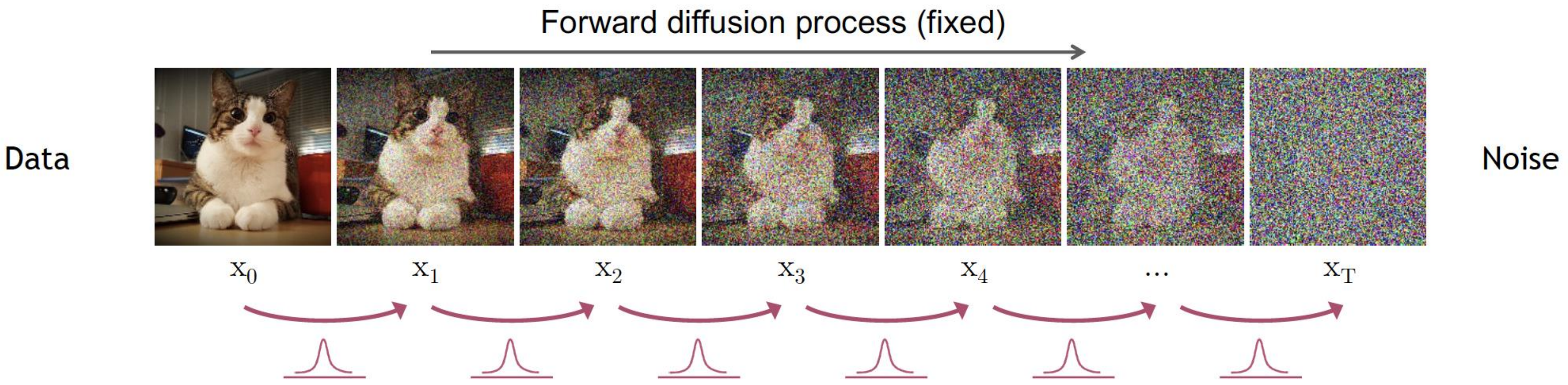
Sohl-Dickstein et al., Deep Unsupervised Learning using Nonequilibrium Thermodynamics, ICML 2015
Ho et al., Denoising Diffusion Probabilistic Models, NeurIPS 2020
Song et al., Score-Based Generative Modeling through Stochastic Differential Equations, ICLR 2021

2024

Slide: https://cvpr2022-tutorial-diffusion-models.github.io/

Forward Diffusion Process

The formal definition of the forward process in T steps:

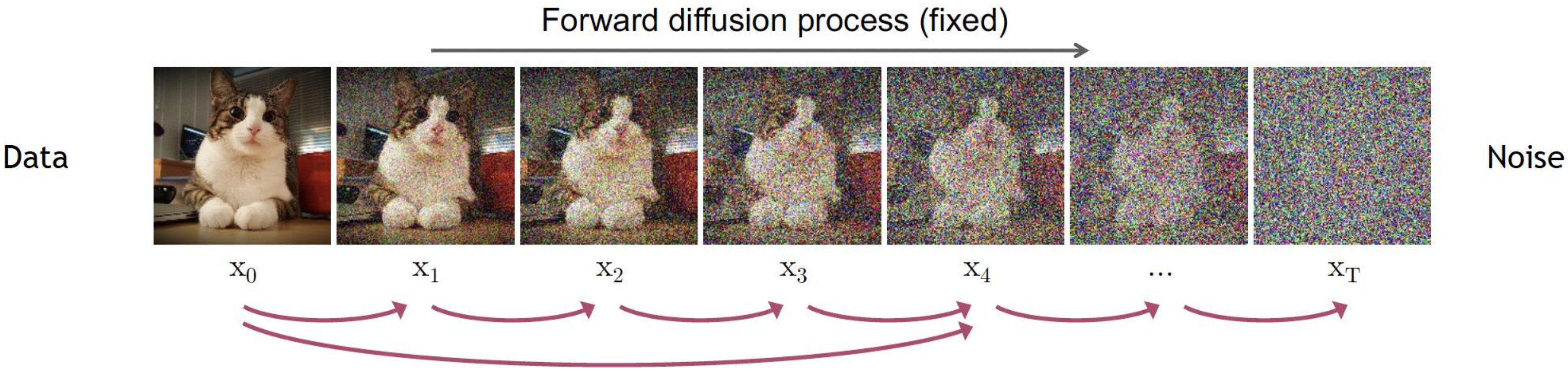


$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x_t}; \sqrt{1-\beta_t}\mathbf{x_{t-1}}, \beta_t\mathbf{I}) \quad \Rightarrow \quad q(\mathbf{x}_{1:T}|\mathbf{x}_0) = \prod_{t=1}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1}) \quad \text{(joint)}$$

2024

Slide: https://cvpr2022-tutorial-diffusion-models.github.io/

Diffusion Kernel



Define $\bar{\alpha}_t = \prod_{s=1}^{t}(1 - \beta_s)$ ➡ $q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1 - \bar{\alpha}_t)\mathbf{I}))$ (Diffusion Kernel)

For sampling: $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\ \mathbf{x}_0 + \sqrt{(1 - \bar{\alpha}_t)}\ \epsilon$ where $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$

β_t values schedule (i.e., the noise schedule) is designed such that $\bar{\alpha}_T \to 0$ and $q(\mathbf{x}_T|\mathbf{x}_0) \approx \mathcal{N}(\mathbf{x}_T; \mathbf{0}, \mathbf{I}))$

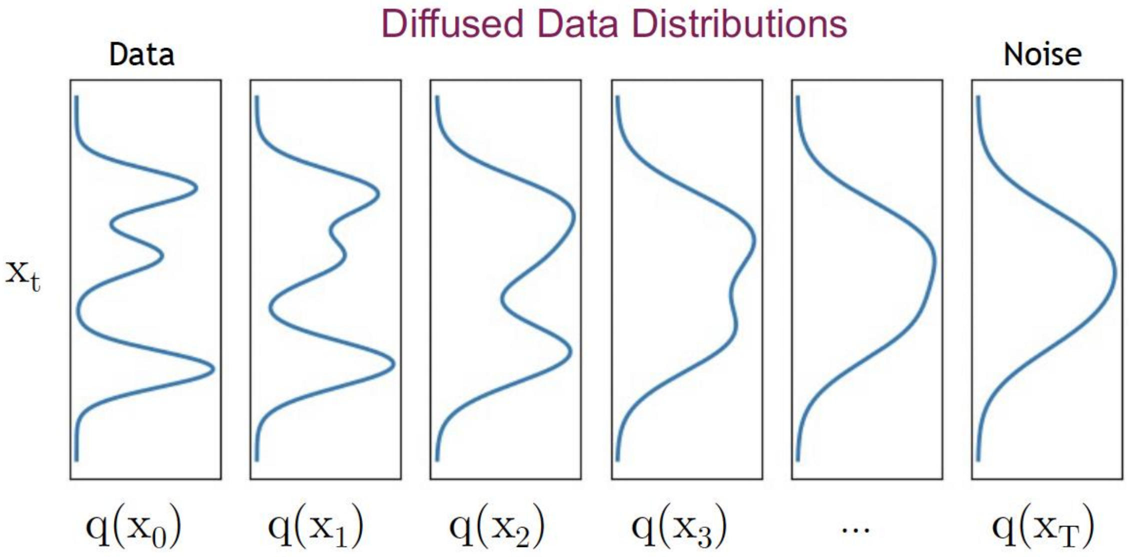
Slide: https://cvpr2022-tutorial-diffusion-models.github.io/

What happens to a distribution in the forward diffusion?

So far, we discussed the diffusion kernel $q(\mathbf{x}_t|\mathbf{x}_0)$ but what about $q(\mathbf{x}_t)$?

$$\underbrace{q(\mathbf{x}_t)}_{\text{Diffused data dist.}} = \int \underbrace{q(\mathbf{x}_0, \mathbf{x}_t)}_{\text{Joint dist.}} \, d\mathbf{x}_0 = \int \underbrace{q(\mathbf{x}_0)}_{\text{Input data dist.}} \underbrace{q(\mathbf{x}_t|\mathbf{x}_0)}_{\text{Diffusion kernel}} \, d\mathbf{x}_0$$

The diffusion kernel is Gaussian convolution.



We can sample $\mathbf{x}_t \sim q(\mathbf{x}_t)$ by first sampling $\mathbf{x}_0 \sim q(\mathbf{x}_0)$ and then sampling $\mathbf{x}_t \sim q(\mathbf{x}_t|\mathbf{x}_0)$ (i.e., ancestral sampling).

Slide: https://cvpr2022-tutorial-diffusion-models.github.io/

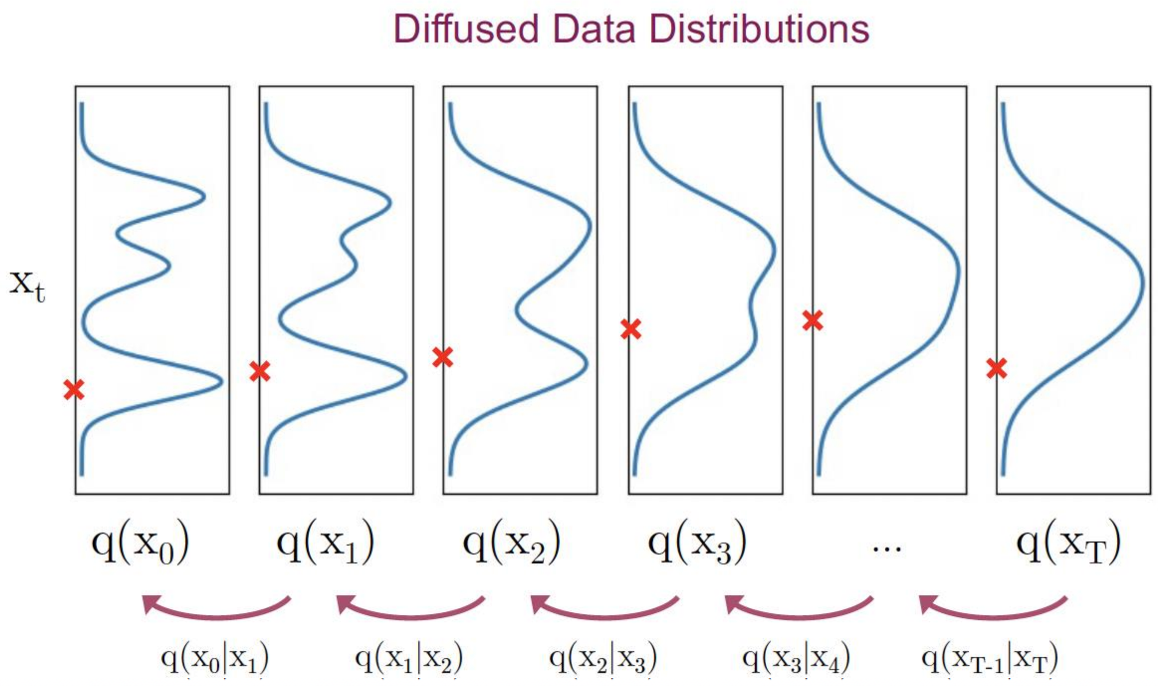
Generative Learning by Denoising

Recall, that the diffusion parameters are designed such that $q(\mathbf{x}_T) \approx \mathcal{N}(\mathbf{x}_T; \mathbf{0}, \mathbf{I})$

Generation:

Sample $\mathbf{x}_T \sim \mathcal{N}(\mathbf{x}_T; \mathbf{0}, \mathbf{I})$

Iteratively sample $\mathbf{x}_{t-1} \sim \underbrace{q(\mathbf{x}_{t-1}|\mathbf{x}_t)}_{\text{True Denoising Dist.}}$



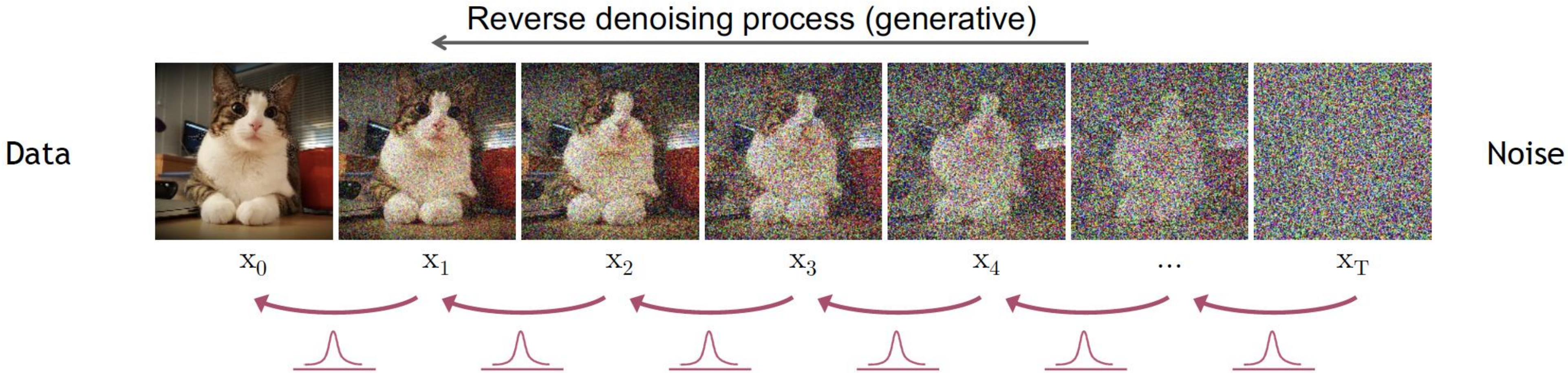
In general, $q(\mathbf{x}_{t-1}|\mathbf{x}_t) \propto q(\mathbf{x}_{t-1})q(\mathbf{x}_t|\mathbf{x}_{t-1})$ is intractable.

Can we approximate $q(\mathbf{x}_{t-1}|\mathbf{x}_t)$? Yes, we can use a Normal distribution if β_t is small in each forward diffusion step.

Slide: https://cvpr2022-tutorial-diffusion-models.github.io/

Reverse Denoising Process

Formal definition of forward and reverse processes in T steps:



$$p(\mathbf{x}_T) = \mathcal{N}(\mathbf{x}_T; \mathbf{0}, \mathbf{I})$$

$$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \underbrace{\mu_\theta(\mathbf{x}_t, t)}_{\text{Trainable network}}, \sigma_t^2\mathbf{I})$$

➡ $$p_\theta(\mathbf{x}_{0:T}) = p(\mathbf{x}_T)\prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$$

Trainable network
(U-net, Denoising Autoencoder)

Slide: https://cvpr2022-tutorial-diffusion-models.github.io/

Learning Denoising Model

Variational upper bound

For training, we can form variational upper bound that is commonly used for training variational autoencoders:

$$\mathbb{E}_{q(\mathbf{x}_0)}\left[-\log p_\theta(\mathbf{x}_0)\right] \leq \mathbb{E}_{q(\mathbf{x}_0)q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[-\log \frac{p_\theta(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\right] =: L$$

Sohl-Dickstein et al. ICML 2015 and Ho et al. NeurIPS 2020 show that:

$$L = \mathbb{E}_q\left[\underbrace{D_{\mathrm{KL}}(q(\mathbf{x}_T|\mathbf{x}_0)||p(\mathbf{x}_T))}_{L_T} + \sum_{t>1}\underbrace{D_{\mathrm{KL}}(q(\mathbf{x}_{t-1}|\mathbf{x}_t,\mathbf{x}_0)||p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t))}_{L_{t-1}} \underbrace{-\log p_\theta(\mathbf{x}_0|\mathbf{x}_1)}_{L_0}\right]$$

where $q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)$ is the tractable posterior distribution:

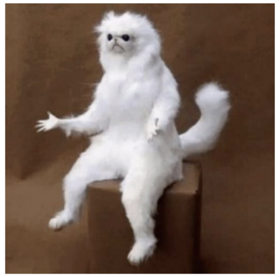
$$q(\mathbf{x}_{t-1}|\mathbf{x}_t,\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_{t-1}; \tilde{\mu}_t(\mathbf{x}_t,\mathbf{x}_0), \tilde{\beta}_t\mathbf{I}),$$

$$\text{where } \tilde{\mu}_t(\mathbf{x}_t,\mathbf{x}_0) := \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}\mathbf{x}_0 + \frac{\sqrt{1-\beta_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\mathbf{x}_t \text{ and } \tilde{\beta}_t := \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$$

Slide: https://cvpr2022-tutorial-diffusion-models.github.io/

Diffusion Models – Updated ELBO

*Derivation in appendix!



$$\log p(x) = \log \int p(x_{0:T}) dx_{0:T}$$

... *

$$= \mathbb{E}_{q(x_1 \mid x_0)}[\log p_\theta(x_0|x_1)] - D_{KL}(q(x_T \mid x_0) || p(x_T)) - \sum_{t=2}^{T} \mathbb{E}_{q(x_t \mid x_0)}\left[D_{KL}\left(q(x_{t-1} \mid x_t, x_0) || p_\theta(x_{t-1}|x_t)\right)\right]$$

Reconstruction | Prior matching | Denoising

- Reconstruction: Reconstruction from least noisy version **(hyperparameter choice can make this arbitrarily small)**
- Prior matching: Moving the posterior closer to the true prior on the final noisy step **(0 for diffusion models)**
- Denoising: Divergence between approximate denoising (p_θ) and true denoising (q) steps

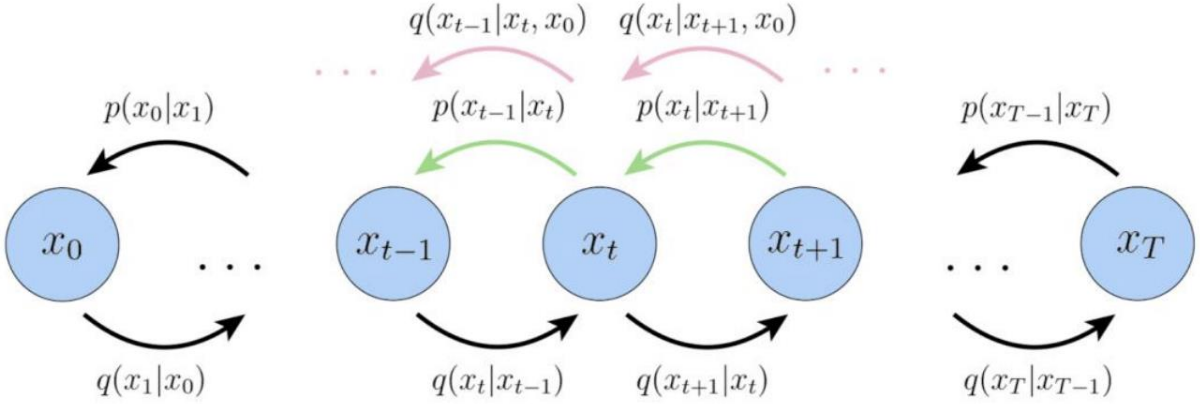
$q(x_{t-1} \mid x_t, x_0)$ is **tractable** and can be calculated **exactly** without any approximation:

$$q(x_{t-1} \mid x_t, x_0) = \mathcal{N}(x_{t-1};\ \tilde{\mu}_t, \Sigma_t \boldsymbol{I})$$

$$\tilde{\mu}_t = \frac{\sqrt{\alpha_t}(1 - \bar{\alpha}_{t-1})x_t + \sqrt{\bar{\alpha}_{t-1}}(1 - \alpha_t)x_0}{1 - \overline{\alpha_t}}, \qquad \Sigma_t = \frac{(1 - \alpha_t)(1 - \bar{\alpha}_{t-1})}{1 - \overline{\alpha_t}}$$

Slide: https://deeplearning.cs.cmu.edu/F23/document/slides/lec23.diffusion.updated.pdf

Diffusion Models – Loss formulation



Loss can focus on the denoising term. Decomposing for each timestep, we can have the t^{th} loss term:

$$L_t = D_{KL}\big(q(x_{t-1}|\, x_t, x_0)\; || p_\theta(x_{t-1}|x_t)\big) + C$$

Since both inputs of the divergence are gaussians, this further simplifies to:

$$\mathrm{L_t} = \mathbb{E}_q\left[\frac{1}{2\Sigma_t}||\bar{\mu_t} \; - \mu_\theta(\mathrm{x}_t, t)||^2\right] + \mathrm{C}$$

Slide: https://deeplearning.cs.cmu.edu/F23/document/slides/lec23.diffusion.updated.pdf

Parameterizing the Denoising Model

Since both $q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)$ and $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$ are Normal distributions, the KL divergence has a simple form:

$$L_{t-1} = D_{\mathrm{KL}}(q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)||p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)) = \mathbb{E}_q\left[\frac{1}{2\sigma_t^2}||\tilde{\mu}_t(\mathbf{x}_t, \mathbf{x}_0) - \mu_\theta(\mathbf{x}_t, t)||^2\right] + C$$

Recall that $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\ \mathbf{x}_0 + \sqrt{(1-\bar{\alpha}_t)}\ \epsilon$. Ho et al. NeurIPS 2020 observe that:

$$\tilde{\mu}_t(\mathbf{x}_t, \mathbf{x}_0) = \frac{1}{\sqrt{1-\beta_t}}\left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon\right)$$

They propose to represent the mean of the denoising model using a *noise-prediction* network:

$$\mu_\theta(\mathbf{x}_t, t) = \frac{1}{\sqrt{1-\beta_t}}\left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(\mathbf{x}_t, t)\right)$$

With this parameterization

$$L_{t-1} = \mathbb{E}_{\mathbf{x}_0 \sim q(\mathbf{x}_0), \epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})}\left[\frac{\beta_t^2}{2\sigma_t^2(1-\beta_t)(1-\bar{\alpha}_t)}||\epsilon - \epsilon_\theta(\underbrace{\sqrt{\bar{\alpha}_t}\ \mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\ \epsilon}_{\mathbf{x}_t}, t)||^2\right] + C$$

2024

Slide: https://cvpr2022-tutorial-diffusion-models.github.io/

Training Objective Weighting

Trading likelihood for perceptual quality

$$L_{t-1} = \mathbb{E}_{\mathbf{x}_0 \sim q(\mathbf{x}_0), \epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})} \Bigg[\underbrace{\frac{\beta_t^2}{2\sigma_t^2 (1-\beta_t)(1-\bar{\alpha}_t)}}_{\lambda_t} ||\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}\, \mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\, \epsilon, t)||^2 \Bigg]$$

The time dependent λ_t ensures that the training objective is weighted properly for the maximum data likelihood training.

However, this weight is often very large for small t's.

Ho et al. NeurIPS 2020 observe that simply setting $\lambda_t = 1$ improves sample quality. So, they propose to use:

$$L_{\text{simple}} = \mathbb{E}_{\mathbf{x}_0 \sim q(\mathbf{x}_0), \epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), t \sim \mathcal{U}(1,T)} \Big[||\epsilon - \epsilon_\theta(\underbrace{\sqrt{\bar{\alpha}_t}\, \mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\, \epsilon}_{\mathbf{x}_t}, t)||^2 \Big]$$

For more advanced weighting see Choi et al., Perception Prioritized Training of Diffusion Models, CVPR 2022.

Slide: https://cvpr2022-tutorial-diffusion-models.github.io/

Ho et al., “Denoising Diffusion Probabilistic Models”, 2020.

The Three Terms

$$L = \mathbb{E}_q \Big[\underbrace{D_{\mathrm{KL}}(q(\mathbf{x}_T|\mathbf{x}_0) \,||\, p(\mathbf{x}_T))}_{L_T} + \sum_{t>1} \underbrace{D_{\mathrm{KL}}(q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) \,||\, p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t))}_{L_{t-1}} \underbrace{- \log p_\theta(\mathbf{x}_0|\mathbf{x}_1)}_{L_0} \Big]$$

3.1 Forward process and L_T

We ignore the fact that the forward process variances β_t are learnable by reparameterization and instead fix them to constants (see Section 4 for details). Thus, in our implementation, the approximate posterior q has no learnable parameters, so L_T is a constant during training and can be ignored.

the standard normal prior $p(\mathbf{x}_T)$. To obtain discrete log likelihoods, we set the last term of the reverse process to an independent discrete decoder derived from the Gaussian $\mathcal{N}(\mathbf{x}_0; \boldsymbol{\mu}_\theta(\mathbf{x}_1, 1), \sigma_1^2\mathbf{I})$:

$$p_\theta(\mathbf{x}_0|\mathbf{x}_1) = \prod_{i=1}^{D} \int_{\delta_-(x_0^i)}^{\delta_+(x_0^i)} \mathcal{N}(x; \mu_\theta^i(\mathbf{x}_1, 1), \sigma_1^2)\, dx$$
$$\delta_+(x) = \begin{cases} \infty & \text{if } x = 1 \\ x + \frac{1}{255} & \text{if } x < 1 \end{cases} \qquad \delta_-(x) = \begin{cases} -\infty & \text{if } x = -1 \\ x - \frac{1}{255} & \text{if } x > -1 \end{cases} \tag{13}$$

$$L_{\mathrm{simple}}(\theta) := \mathbb{E}_{t, \mathbf{x}_0, \boldsymbol{\epsilon}} \Big[\big\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t}\boldsymbol{\epsilon}, t) \big\|^2 \Big] \tag{14}$$

Summary

Training and Sample Generation

Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, \ldots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$$
6: **until** converged

Algorithm 2 Sampling

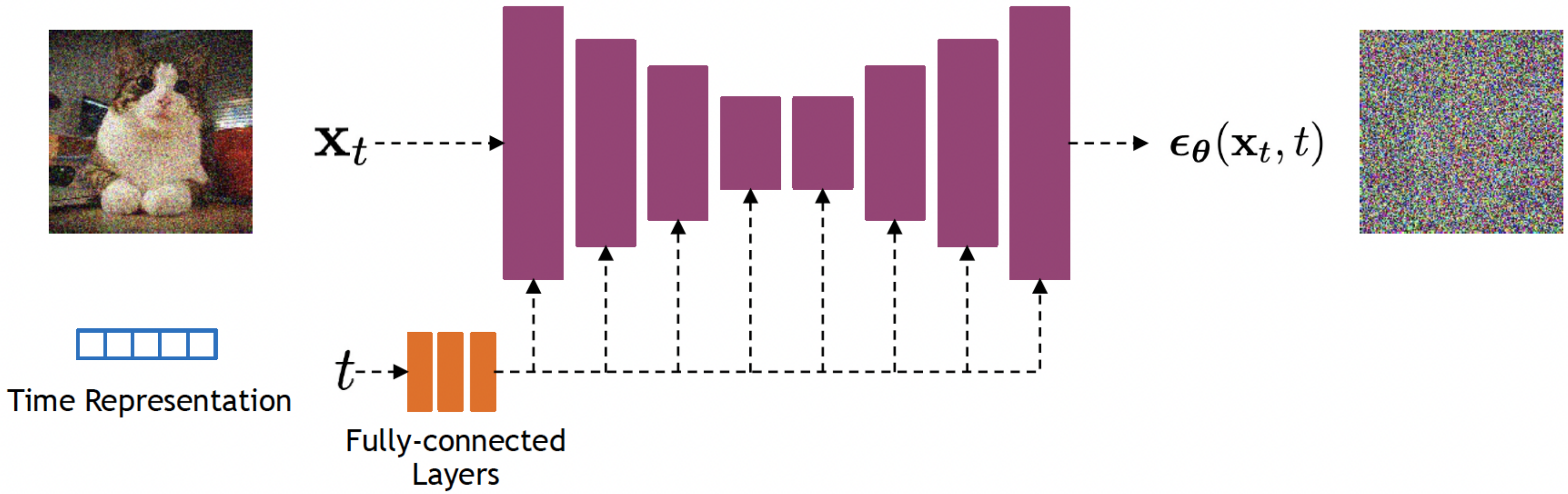
1: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: **for** $t = T, \ldots, 1$ **do**
3: $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
4: $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t) \right) + \sigma_t \mathbf{z}$
5: **end for**
6: **return** $\mathbf{x}_0$

Slide: https://cvpr2022-tutorial-diffusion-models.github.io/

Implementation Considerations

Network Architectures

Diffusion models often use U-Net architectures with ResNet blocks and self-attention layers to represent $\epsilon_\theta(\mathbf{x}_t, t)$



Time representation: sinusoidal positional embeddings or random Fourier features.

Time features are fed to the residual blocks using either simple spatial addition or using adaptive group normalization layers. (see Dharivwal and Nichol NeurIPS 2021)

Slide: https://cvpr2022-tutorial-diffusion-models.github.io/

Diffusion Parameters

Noise Schedule

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x_t}; \sqrt{1-\beta_t}\mathbf{x_{t-1}}, \beta_t\mathbf{I})$$

Data

Noise

$$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \mu_\theta(\mathbf{x}_t, t), \sigma_t^2\mathbf{I})$$

Above, β_t and σ_t^2 control the variance of the forward diffusion and reverse denoising processes respectively.

Often a linear schedule is used for β_t, and σ_t^2 is set equal to β_t.

Kingma et al. NeurIPS 2022 introduce a new parameterization of diffusion models using signal-to-noise ratio (SNR), and show how to learn the noise schedule by minimizing the variance of the training objective.

We can also train σ_t^2 while training the diffusion model by minimizing the variational bound (Improved DPM by Nichol and Dhariwal ICML 2021) or after training the diffusion model (Analytic-DPM by Bao et al. ICLR 2022).

Slide: https://cvpr2022-tutorial-diffusion-models.github.io/

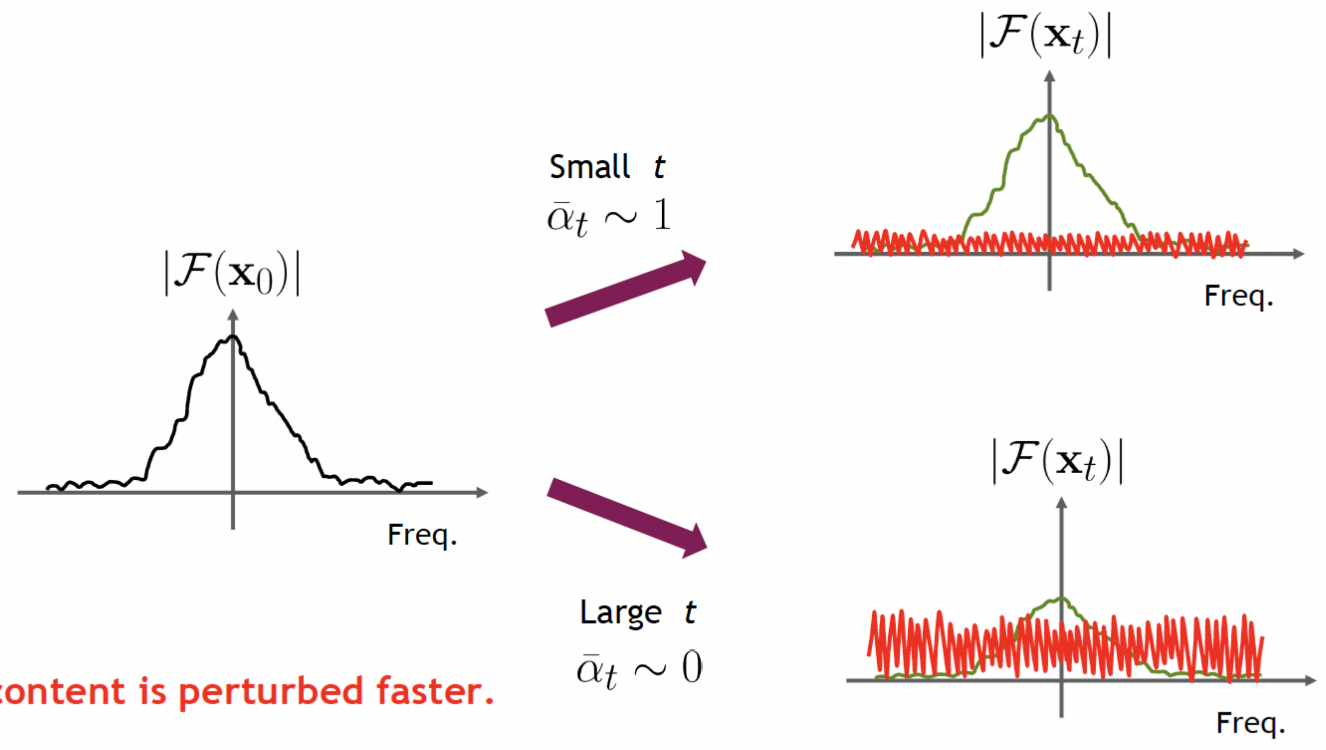
What happens to an image in the forward diffusion process?

Recall that sampling from $q(\mathbf{x}_t|\mathbf{x}_0)$ is done using $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\ \mathbf{x}_0 + \sqrt{(1-\bar{\alpha}_t)}\ \epsilon$ where $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\ \mathbf{x}_0 + \sqrt{(1-\bar{\alpha}_t)}\ \epsilon$$

Fourier Transform

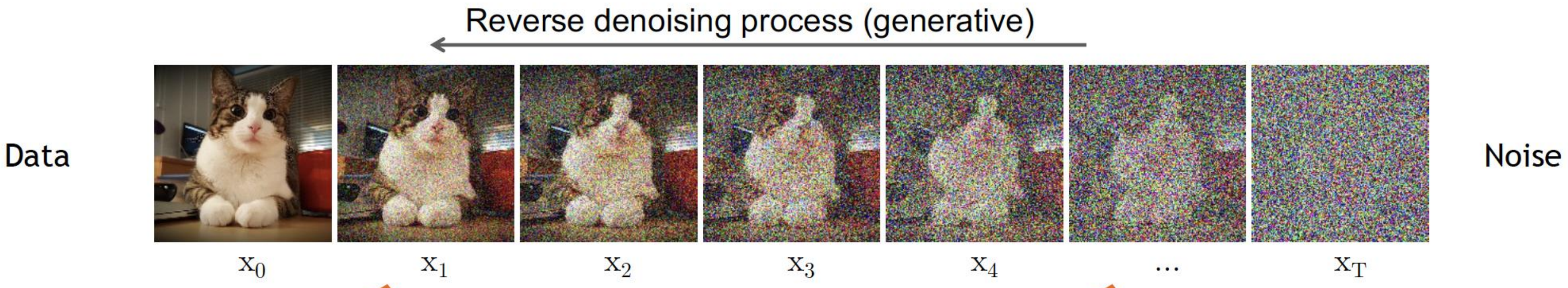
$$\mathcal{F}(\mathbf{x}_t) = \sqrt{\bar{\alpha}_t}\ \mathcal{F}(\mathbf{x}_0) + \sqrt{(1-\bar{\alpha}_t)}\ \mathcal{F}(\epsilon)$$



In the forward diffusion, the high frequency content is perturbed faster.

Slide: https://cvpr2022-tutorial-diffusion-models.github.io/

Content-Detail Tradeoff



The denoising model is specialized for generating the high-frequency content (i.e., low-level details)

The denoising model is specialized for generating the low-frequency content (i.e., coarse content)

The weighting of the training objective for different timesteps is important!

2024

Slide: https://cvpr2022-tutorial-diffusion-models.github.io/

Latent Diffusion Models (Stable Diffusion)

Main differences:

- Use a pretrained encoder ($\mathcal{E}$) and a decoder ($\mathcal{D}$)
- Conditioning with cross-attention

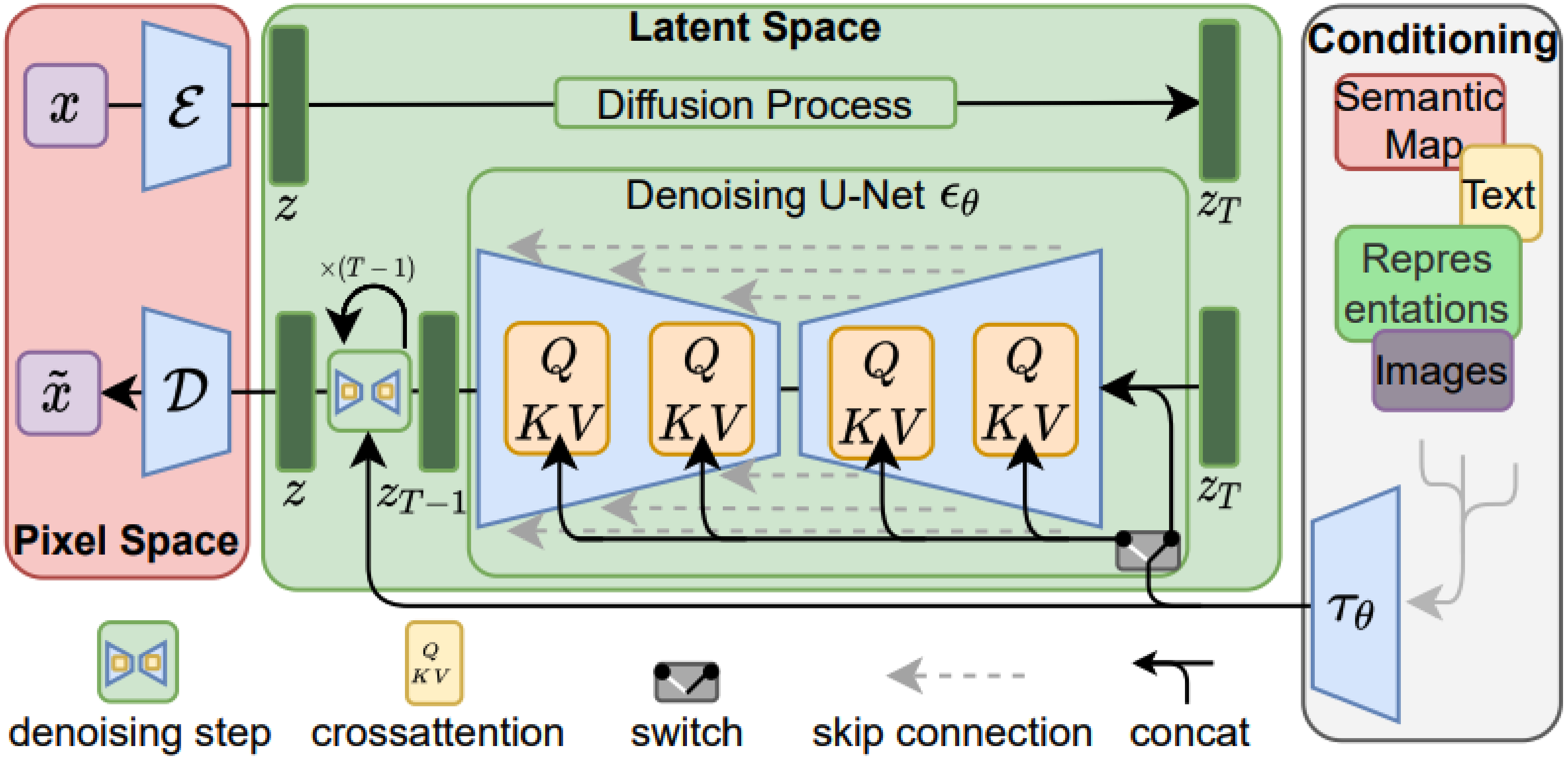


Figure 3. We condition LDMs either via concatenation or by a more general cross-attention mechanism. See Sec. 3.3

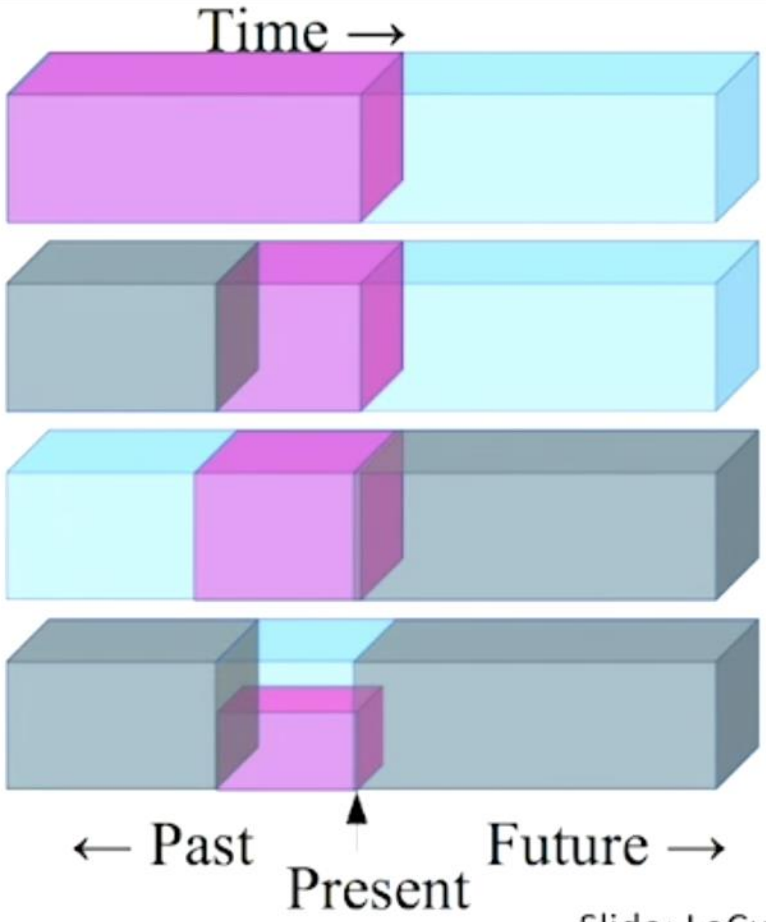
Diffusion models - Summary

- Diffusion models are **Markovian Hierarchical VAEs** with extra restrictions
- The loss is the vanilla VAE ELBO loss with an added denoising term
- The encoder has **0 parameters**
- The true denoising posterior can be **exactly calculated**
- The problem can be reformulated as a noise prediction problem
- There's a ton of math underlying a rather simple intuition

Slide: https://deeplearning.cs.cmu.edu/F23/document/slides/lec23.diffusion.updated.pdf

Self-supervised learning

- Predict any part of the input from any other part.
- Predict the future from the past.
- Predict the future from the recent past.
- Predict the past from the present.
- Predict the top from the bottom.
- Predict the occluded from the visible
- Pretend there is a part of the input you don't know and predict that.



Slide: LeCun

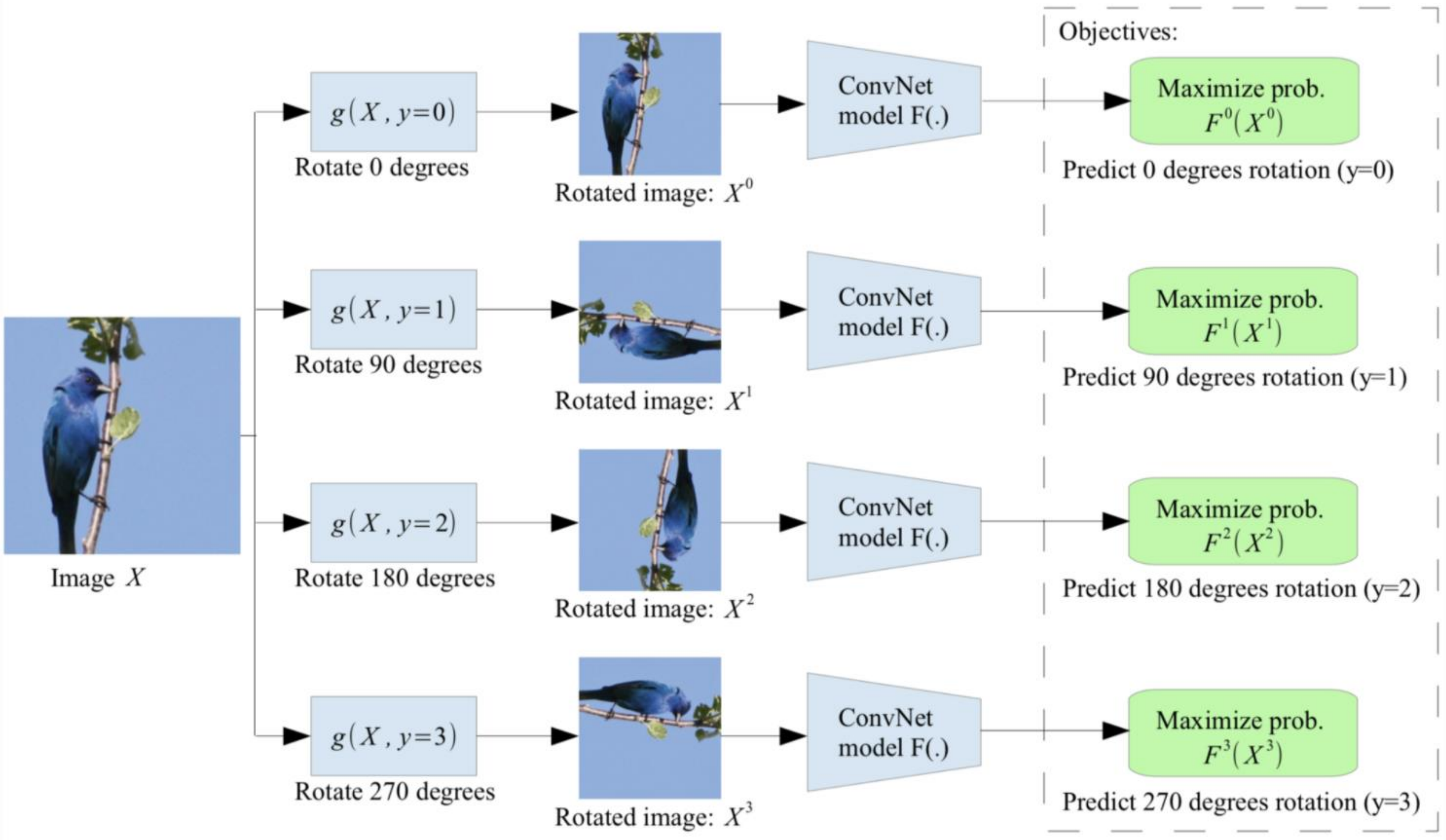


Fig. 3. Illustration of self-supervised learning by rotating the entire input images. The model learns to predict which rotation is applied. (Image source: Gidaris et al. 2018)

From: https://lilianweng.github.io/lil-log/2019/11/10/self-supervised-learning.html

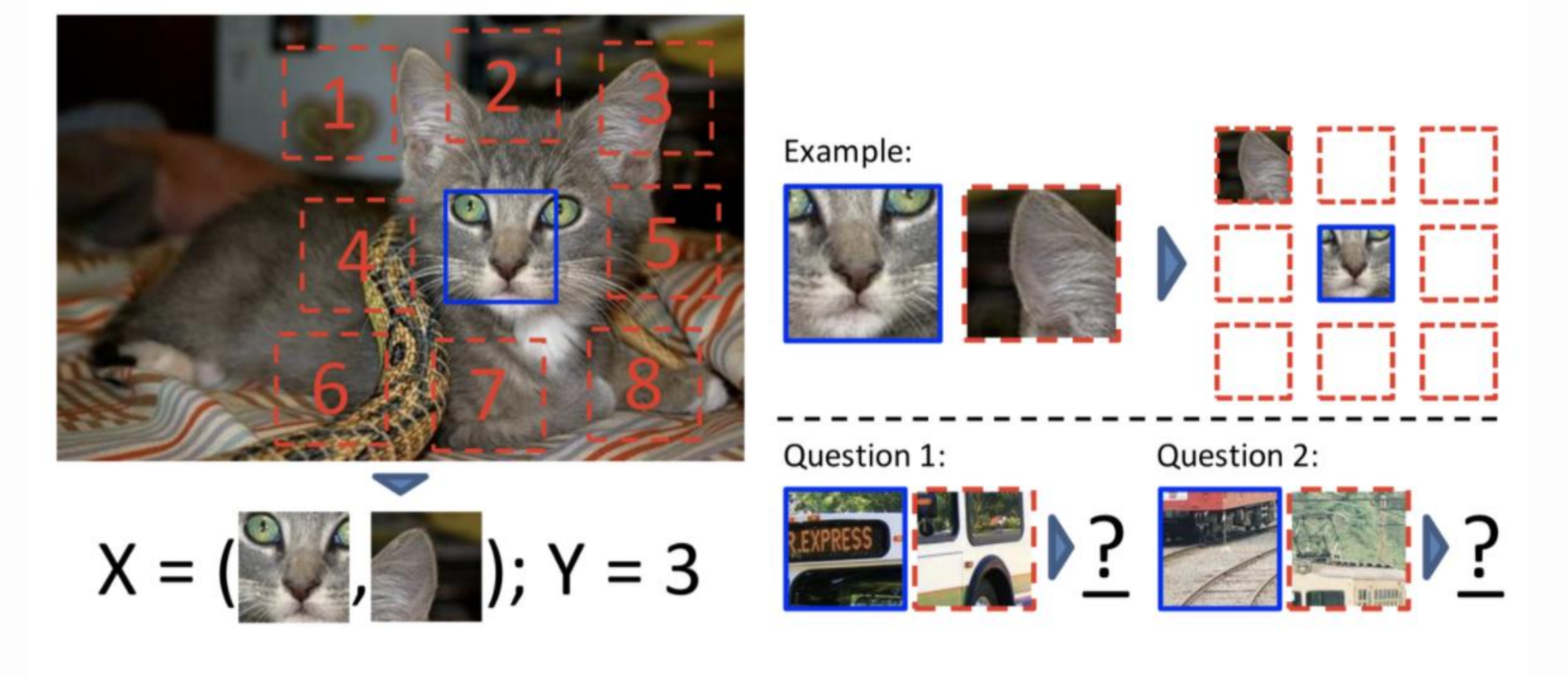


Fig. 4. Illustration of self-supervised learning by predicting the relative position of two random patches. (Image source: Doersch et al., 2015)

From: https://lilianweng.github.io/lil-log/2019/11/10/self-supervised-learning.html

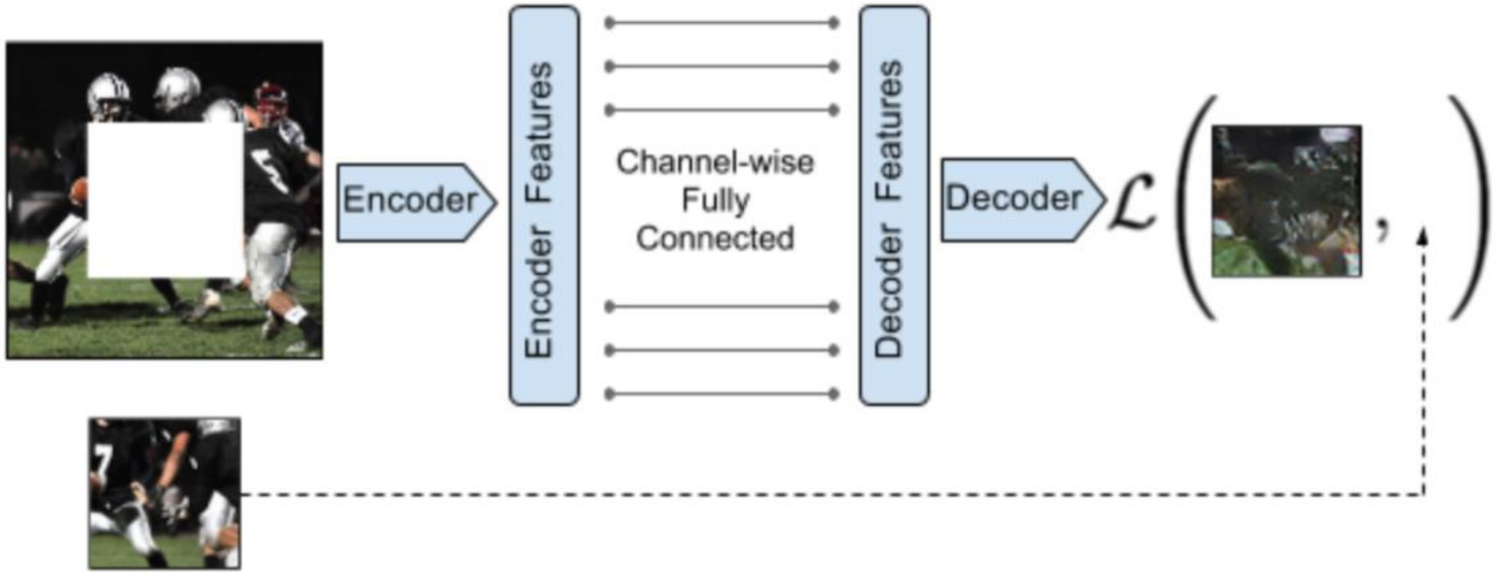


Fig. 8. Illustration of context encoder. (Image source: Pathak, et al., 2016)

From: https://lilianweng.github.io/lil-log/2019/11/10/self-supervised-learning.html

Siamese Networks

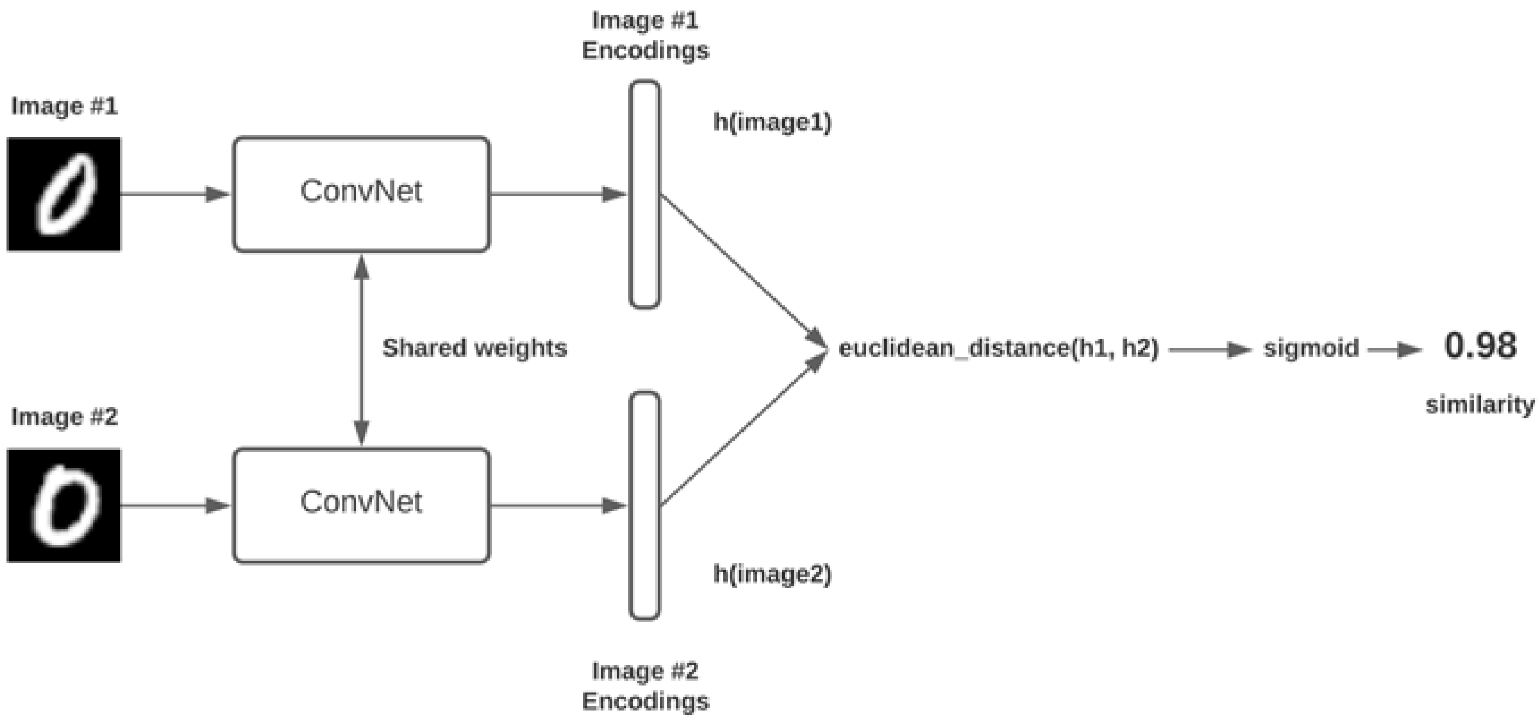


Fig: https://www.pyimagesearch.com/2020/11/30/siamese-networks-with-keras-tensorflow-and-deep-learning/

Contrastive Loss (Chopra et al., 2005)

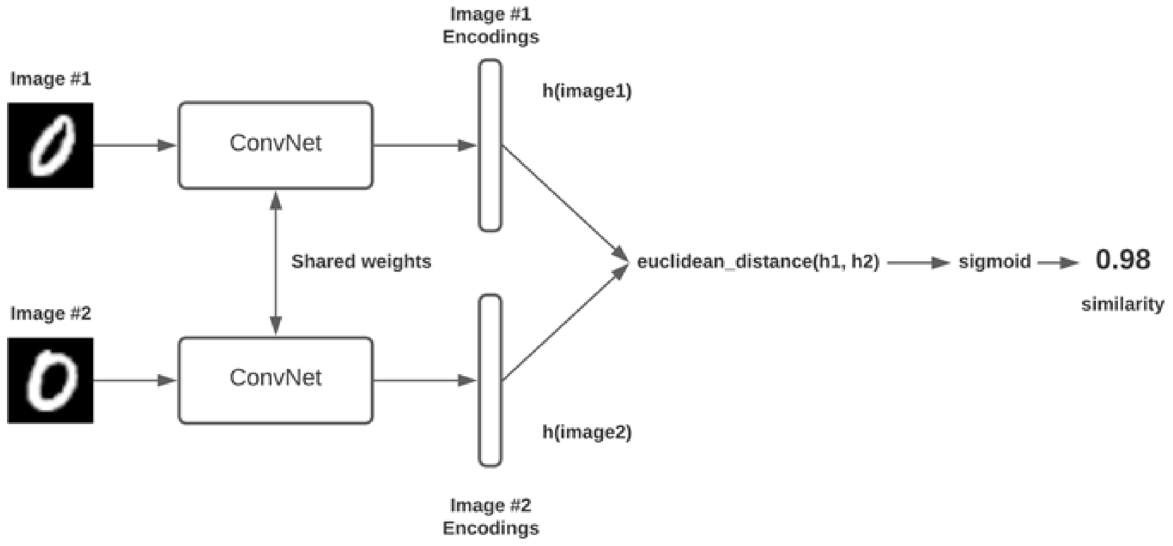


Fig: https://www.pyimagesearch.com/2020/11/30/siamese-networks-with-keras-tensorflow-and-deep-learning/

$y = 1$ for "similar" pairs:

$$\mathcal{L}_{\text{cont}}(\mathbf{x}_i, \mathbf{x}_j, \theta) = \mathbb{1}[y_i = y_j]\|f_\theta(\mathbf{x}_i) - f_\theta(\mathbf{x}_j)\|_2^2 + \mathbb{1}[y_i \neq y_j]\max(0, \epsilon - \|f_\theta(\mathbf{x}_i) - f_\theta(\mathbf{x}_j)\|_2)^2$$

From: https://lilianweng.github.io/lil-log/2019/11/10/self-supervised-learning.html

Triplet Loss (Schroff et al., 2015)

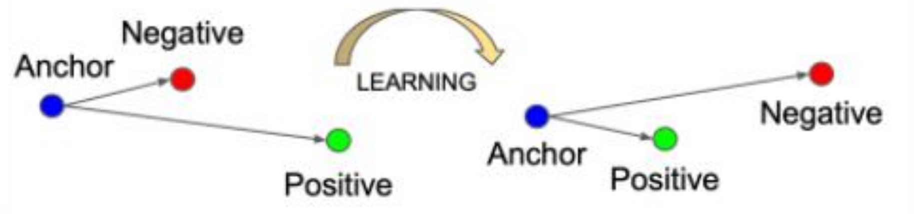
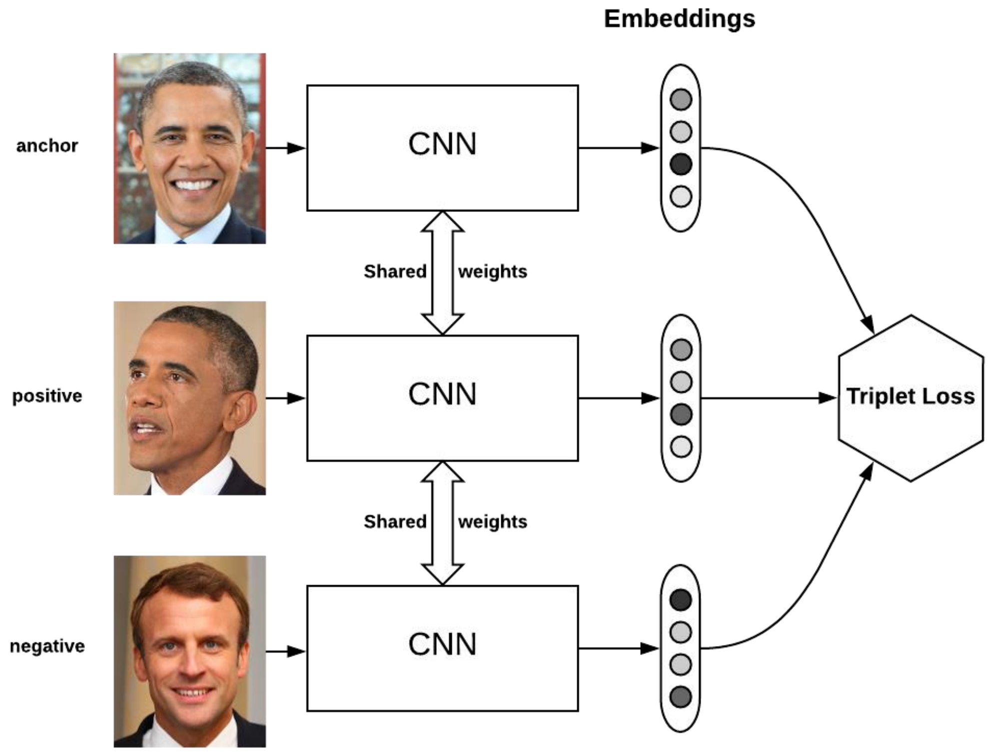


Fig. 1. Illustration of triplet loss given one positive and one negative per anchor. (Image source: Schroff et al. 2015)

$$\mathcal{L}_{\text{triplet}}(\mathbf{x}, \mathbf{x}^+, \mathbf{x}^-) = \sum_{\mathbf{x} \in \mathcal{X}} \max\big(0, \|f(\mathbf{x}) - f(\mathbf{x}^+)\|_2^2 - \|f(\mathbf{x}) - f(\mathbf{x}^-)\|_2^2 + \epsilon\big)$$



https://omoindrot.github.io/triplet-loss

From: https://lilianweng.github.io/lil-log/2019/11/10/self-supervised-learning.html

Lifted Structure Loss (Song et al., 2015)

Let $D_{ij} = \|f(\mathbf{x}_i) - f(\mathbf{x}_j)\|_2$, a structured loss function is defined as

$$\mathcal{L}_{\text{struct}} = \frac{1}{2|\mathcal{P}|} \sum_{(i,j)\in\mathcal{P}} \max(0, \mathcal{L}^{(ij)}_{\text{struct}})^2$$

$$\text{where } \mathcal{L}^{(ij)}_{\text{struct}} = D_{ij} + \max\Big(\max_{(i,k)\in\mathcal{N}} \epsilon - D_{ik}, \max_{(j,l)\in\mathcal{N}} \epsilon - D_{jl}\Big)$$

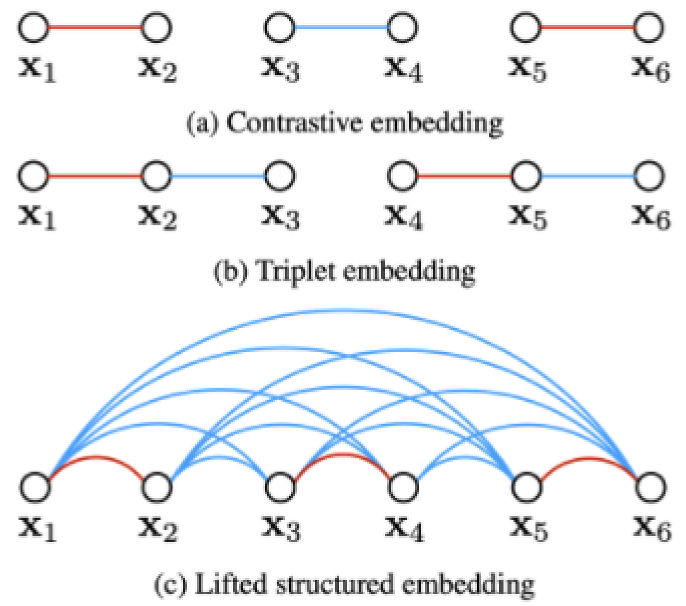


Fig. 2. Illustration compares contrastive loss, triplet loss and lifted structured loss. Red and blue edges connect similar and dissimilar sample pairs respectively. (Image source: Song et al. 2015)

From: https://lilianweng.github.io/lil-log/2019/11/10/self-supervised-learning.html

N-pair Loss (Sohn 2016)

$$\mathcal{L}_{\text{N-pair}}(\mathbf{x}, \mathbf{x}^+, \{\mathbf{x}_i^-\}_{i=1}^{N-1}) = \log\left(1 + \sum_{i=1}^{N-1} \exp(f(\mathbf{x})^\top f(\mathbf{x}_i^-) - f(\mathbf{x})^\top f(\mathbf{x}^+))\right)$$

$$= -\log \frac{\exp(f(\mathbf{x})^\top f(\mathbf{x}^+))}{\exp(f(\mathbf{x})^\top f(\mathbf{x}^+)) + \sum_{i=1}^{N-1} \exp(f(\mathbf{x})^\top f(\mathbf{x}_i^-))}$$

From: https://lilianweng.github.io/lil-log/2019/11/10/self-supervised-learning.html

Momentum Contrast for Unsupervised Visual Representation Learning

Kaiming He Haoqi Fan Yuxin Wu Saining Xie Ross Girshick 2019

Momentum Contrast

- Contrastive learning as dictionary lookup:

for q). With similarity measured by dot product, a form of a contrastive loss function, called InfoNCE [46], is considered in this paper:

$$\mathcal{L}_q = -\log \frac{\exp(q\cdot k_+/\tau)}{\sum_{i=0}^{K} \exp(q\cdot k_i/\tau)} \tag{1}$$

where τ is a temperature hyper-parameter per [61]. The sum is over one positive and K negative samples. Intuitively, this loss is the log loss of a $(K+1)$-way softmax-based classifier that tries to classify q as k_+. Contrastive loss functions

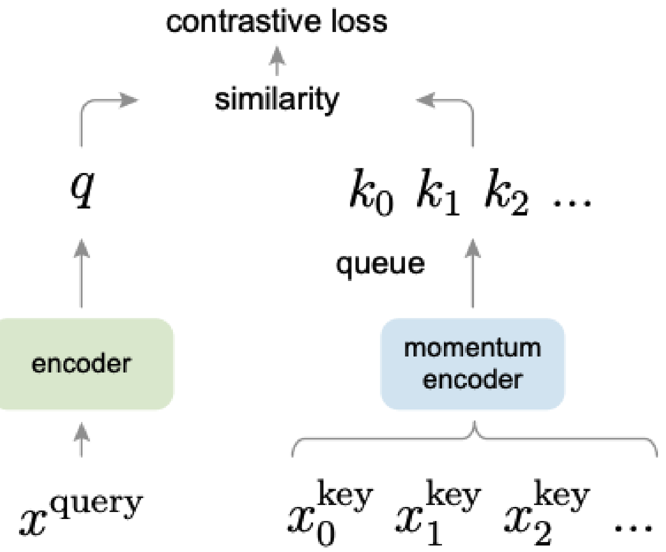
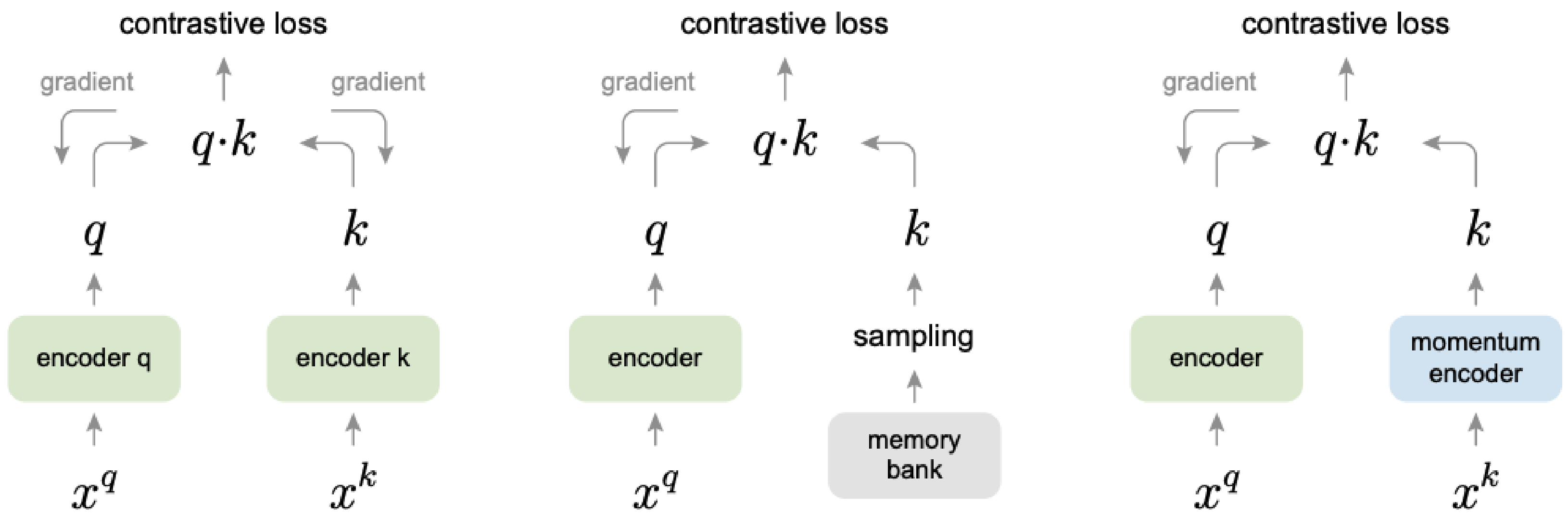


Figure 1. Momentum Contrast (MoCo) trains a visual representation encoder by matching an encoded query q to a dictionary of encoded keys using a contrastive loss. The dictionary keys $\{k_0, k_1, k_2, ...\}$ are defined on-the-fly by a set of data samples. The dictionary is built as a queue, with the current mini-batch enqueued and the oldest mini-batch dequeued, decoupling it from the mini-batch size. The keys are encoded by a slowly progressing encoder, driven by a momentum update with the query encoder. This method enables a large and consistent dictionary for learning visual representations.

Momentum Contrast for Unsupervised Visual Representation Learning

Kaiming He Haoqi Fan Yuxin Wu Saining Xie Ross Girshick 2019

Momentum Contrast



(a) end-to-end (b) memory bank (c) MoCo

Figure 2. **Conceptual comparison of three contrastive loss mechanisms** (empirical comparisons are in Figure 3 and Table 3). Here we illustrate one pair of query and key. The three mechanisms differ in how the keys are maintained and how the key encoder is updated. **(a)**: The encoders for computing the query and key representations are updated ***end-to-end*** by back-propagation (the two encoders can be different). **(b)**: The key representations are sampled from a ***memory bank*** [61]. **(c)**: ***MoCo*** encodes the new keys on-the-fly by a momentum-updated encoder, and maintains a queue (not illustrated in this figure) of keys.

Momentum Contrast

Momentum Contrast for Unsupervised Visual Representation Learning

Kaiming He Haoqi Fan Yuxin Wu Saining Xie Ross Girshick 2019

- Dictionary
  - A queue of data samples
  - Encoded keys from immediately preceding mini-batches
  - Decouples dictionary size from batchsize
  - Samples are progressively replaced: Current batch is added and the oldest is removed.

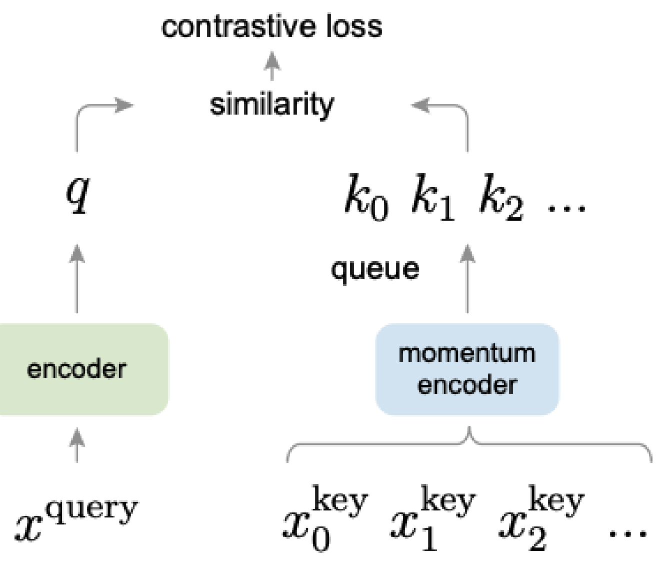


Figure 1. Momentum Contrast (MoCo) trains a visual representation encoder by matching an encoded query q to a dictionary of encoded keys using a contrastive loss. The dictionary keys $\{k_0, k_1, k_2, ...\}$ are defined on-the-fly by a set of data samples. The dictionary is built as a queue, with the current mini-batch enqueued and the oldest mini-batch dequeued, decoupling it from the mini-batch size. The keys are encoded by a slowly progressing encoder, driven by a momentum update with the query encoder. This method enables a large and consistent dictionary for learning visual representations.

Momentum Contrast

Momentum Contrast for Unsupervised Visual Representation Learning

Kaiming He Haoqi Fan Yuxin Wu Saining Xie Ross Girshick 2019

- Momentum update
  - Queue size can be a limiting factor especially if we backpropagate to the samples in the queue as well
  - Naïve solution: Copy image encoder to the queue encoder => Does not work well.
  - Effective solution: Update the queue encoder with the image encoder with momentum update

Formally, denoting the parameters of f_k as θ_k and those of f_q as θ_q, we update θ_k by:

$$\theta_k \leftarrow m\theta_k + (1-m)\theta_q. \quad (2)$$

Here $m \in [0, 1)$ is a momentum coefficient. Only the parameters θ_q are updated by back-propagation. The momen-

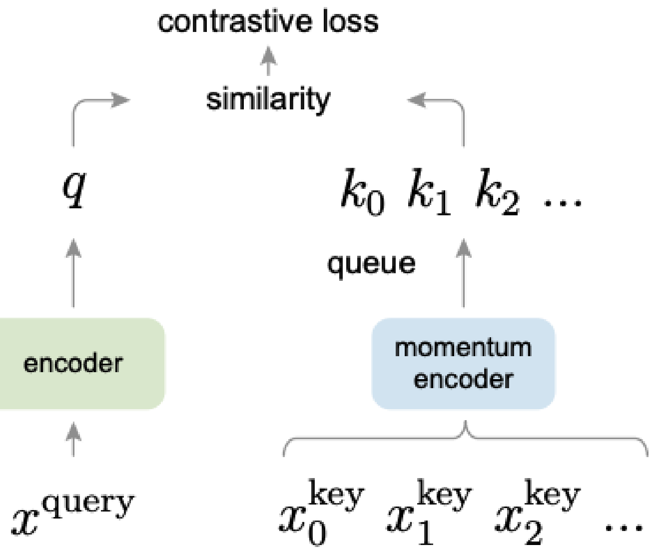


Figure 1. Momentum Contrast (MoCo) trains a visual representation encoder by matching an encoded query q to a dictionary of encoded keys using a contrastive loss. The dictionary keys $\{k_0, k_1, k_2, ...\}$ are defined on-the-fly by a set of data samples. The dictionary is built as a queue, with the current mini-batch enqueued and the oldest mini-batch dequeued, decoupling it from the mini-batch size. The keys are encoded by a slowly progressing encoder, driven by a momentum update with the query encoder. This method enables a large and consistent dictionary for learning visual representations.

Momentum Contrast

Momentum Contrast for Unsupervised Visual Representation Learning

Kaiming He Haoqi Fan Yuxin Wu Saining Xie Ross Girshick 2019

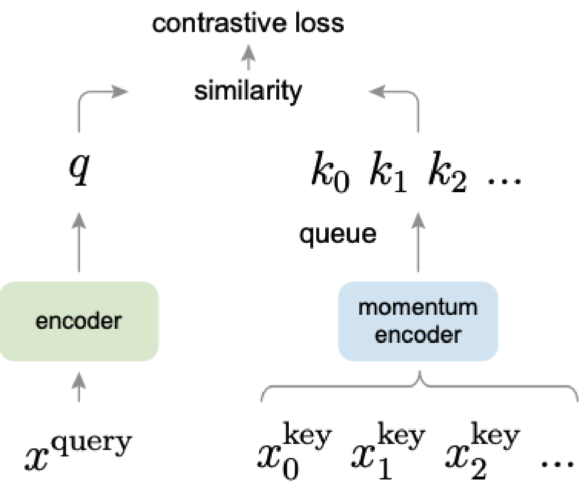


Figure 1. Momentum Contrast (MoCo) trains a visual representation encoder by matching an encoded query q to a dictionary of encoded keys using a contrastive loss. The dictionary keys $\{k_0, k_1, k_2, ...\}$ are defined on-the-fly by a set of data samples. The dictionary is built as a queue, with the current mini-batch enqueued and the oldest mini-batch dequeued, decoupling it from the mini-batch size. The keys are encoded by a slowly progressing encoder, driven by a momentum update with the query encoder. This method enables a large and consistent dictionary for learning visual representations.

Algorithm 1 Pseudocode of MoCo in a PyTorch-like style.

```
# f_q, f_k: encoder networks for query and key
# queue: dictionary as a queue of K keys (CxK)
# m: momentum
# t: temperature

f_k.params = f_q.params # initialize
for x in loader: # load a minibatch x with N samples
    x_q = aug(x) # a randomly augmented version
    x_k = aug(x) # another randomly augmented version

    q = f_q.forward(x_q) # queries: NxC
    k = f_k.forward(x_k) # keys: NxC
    k = k.detach() # no gradient to keys

    # positive logits: Nx1
    l_pos = bmm(q.view(N,1,C), k.view(N,C,1))

    # negative logits: NxK
    l_neg = mm(q.view(N,C), queue.view(C,K))

    # logits: Nx(1+K)
    logits = cat([l_pos, l_neg], dim=1)

    # contrastive loss, Eqn.(1)
    labels = zeros(N) # positives are the 0-th
    loss = CrossEntropyLoss(logits/t, labels)

    # SGD update: query network
    loss.backward()
    update(f_q.params)

    # momentum update: key network
    f_k.params = m*f_k.params+(1-m)*f_q.params

    # update dictionary
    enqueue(queue, k) # enqueue the current minibatch
    dequeue(queue) # dequeue the earliest minibatch
```

bmm: batch matrix multiplication; mm: matrix multiplication; cat: concatenation.

Momentum Contrast for Unsupervised Visual Representation Learning

Kaiming He Haoqi Fan Yuxin Wu Saining Xie Ross Girshick 2019

Momentum Contrast

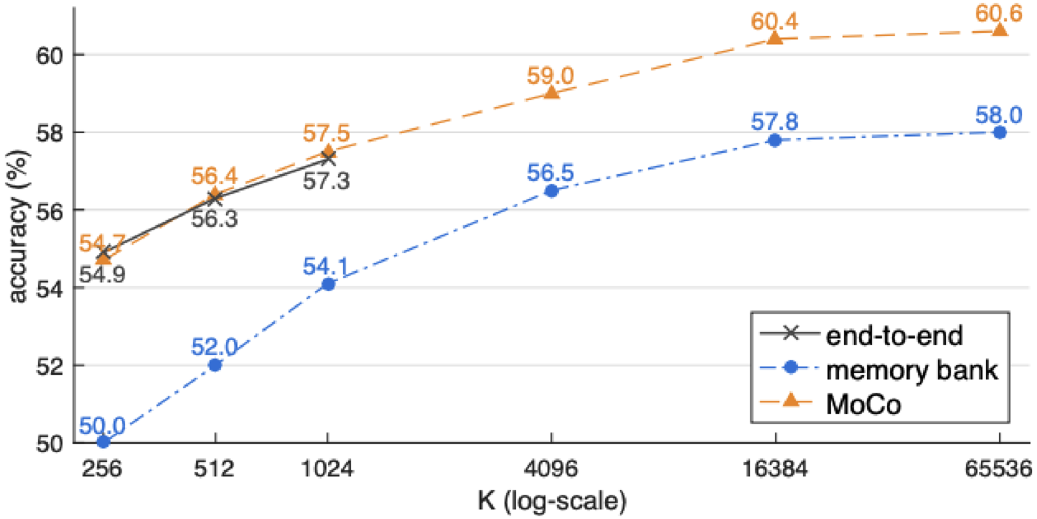


Figure 3. **Comparison of three contrastive loss mechanisms** under the ImageNet linear classification protocol. We adopt the same pretext task (Sec. 3.3) and only vary the contrastive loss mechanism (Figure 2). The number of negatives is K in memory bank and MoCo, and is $K-1$ in end-to-end (offset by one because the positive key is in the same mini-batch). The network is ResNet-50.

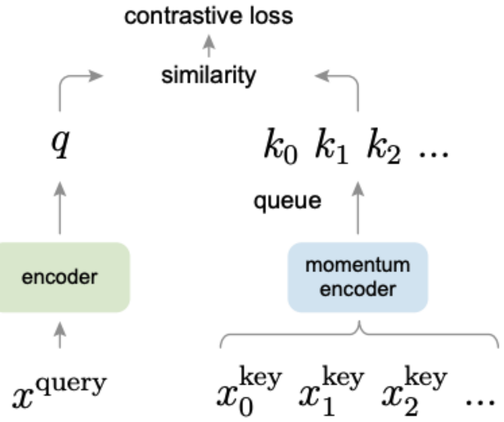


Figure 1. Momentum Contrast (MoCo) trains a visual representation encoder by matching an encoded query q to a dictionary of encoded keys using a contrastive loss. The dictionary keys $\{k_0, k_1, k_2, ...\}$ are defined on-the-fly by a set of data samples. The dictionary is built as a queue, with the current mini-batch enqueued and the oldest mini-batch dequeued, decoupling it from the mini-batch size. The keys are encoded by a slowly progressing encoder, driven by a momentum update with the query encoder. This method enables a large and consistent dictionary for learning visual representations.

Momentum Contrast

Momentum Contrast for Unsupervised Visual Representation Learning

Kaiming He Haoqi Fan Yuxin Wu Saining Xie Ross Girshick 2019

pre-train	R50-dilated-C5 AP_{50}	R50-dilated-C5 AP	R50-dilated-C5 AP_{75}	R50-C4 AP_{50}	R50-C4 AP	R50-C4 AP_{75}
end-to-end	79.2	52.0	56.6	80.4	54.6	60.3
memory bank	79.8	52.9	57.9	80.6	54.9	60.6
MoCo	**81.1**	**54.6**	**59.9**	**81.5**	**55.9**	**62.6**

Table 3. **Comparison of three contrastive loss mechanisms** on PASCAL VOC object detection, fine-tuned on `trainval07+12` and evaluated on `test2007` (averages over 5 trials). All models are implemented by us (Figure 3), pre-trained on IN-1M, and fine-tuned using the same settings as in Table 2.

pre-train	COCO keypoint detection AP^{kp}	AP^{kp}_{50}	AP^{kp}_{75}
random init.	65.9	86.5	71.7
super. IN-1M	65.8	86.9	71.9
MoCo IN-1M	66.8 (+1.0)	87.4 (+0.5)	72.5 (+0.6)
MoCo IG-1B	66.9 (+1.1)	87.8 (+0.9)	73.0 (+1.1)

pre-train	COCO dense pose estimation AP^{dp}	AP^{dp}_{50}	AP^{dp}_{75}
random init.	39.4	78.5	35.1
super. IN-1M	48.3	85.6	50.6
MoCo IN-1M	50.1 (+1.8)	86.8 (+1.2)	53.9 (+3.3)
MoCo IG-1B	50.6 (+2.3)	87.0 (+1.4)	54.3 (+3.7)

pre-train	LVIS v0.5 instance segmentation AP^{mk}	AP^{mk}_{50}	AP^{mk}_{75}
random init.	22.5	34.8	23.8
super. IN-1M†	24.4	37.8	25.8
MoCo IN-1M	24.1 (−0.3)	37.4 (−0.4)	25.5 (−0.3)
MoCo IG-1B	24.9 (+0.5)	38.2 (+0.4)	26.4 (+0.6)

pre-train	Cityscapes instance seg. AP^{mk}	Cityscapes instance seg. AP^{mk}_{50}	Semantic seg. (mIoU) Cityscapes	Semantic seg. (mIoU) VOC
random init.	25.4	51.1	65.3	39.5
super. IN-1M	32.9	59.6	74.6	74.4
MoCo IN-1M	32.3 (−0.6)	59.3 (−0.3)	75.3 (+0.7)	72.5 (−1.9)
MoCo IG-1B	32.9 (0.0)	60.3 (+0.7)	75.5 (+0.9)	73.6 (−0.8)

Table 6. **MoCo *vs.* ImageNet supervised pre-training, fine-tuned on various tasks.** For each task, the same architecture and schedule are used for all entries (see appendix). In the brackets are the gaps to the ImageNet supervised pre-training counterpart. In green are the gaps of at least +0.5 point.

Simple Contrastive Learning

A Simple Framework for Contrastive Learning of Visual Representations

Ting Chen [1] **Simon Kornblith** [1] **Mohammad Norouzi** [1] **Geoffrey Hinton** [1]

2020

as negative examples. Let $\text{sim}(\boldsymbol{u}, \boldsymbol{v}) = \boldsymbol{u}^\top \boldsymbol{v} / \|\boldsymbol{u}\| \|\boldsymbol{v}\|$ denote the dot product between ℓ_2 normalized $\boldsymbol{u}$ and $\boldsymbol{v}$ (i.e. cosine similarity). Then the loss function for a positive pair of examples (i, j) is defined as

$$\ell_{i,j} = -\log \frac{\exp(\text{sim}(\boldsymbol{z}_i, \boldsymbol{z}_j)/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(\text{sim}(\boldsymbol{z}_i, \boldsymbol{z}_k)/\tau)} , \quad (1)$$

where $\mathbb{1}_{[k \neq i]} \in \{0, 1\}$ is an indicator function evaluating to 1 iff $k \neq i$ and τ denotes a temperature parameter. The fi-

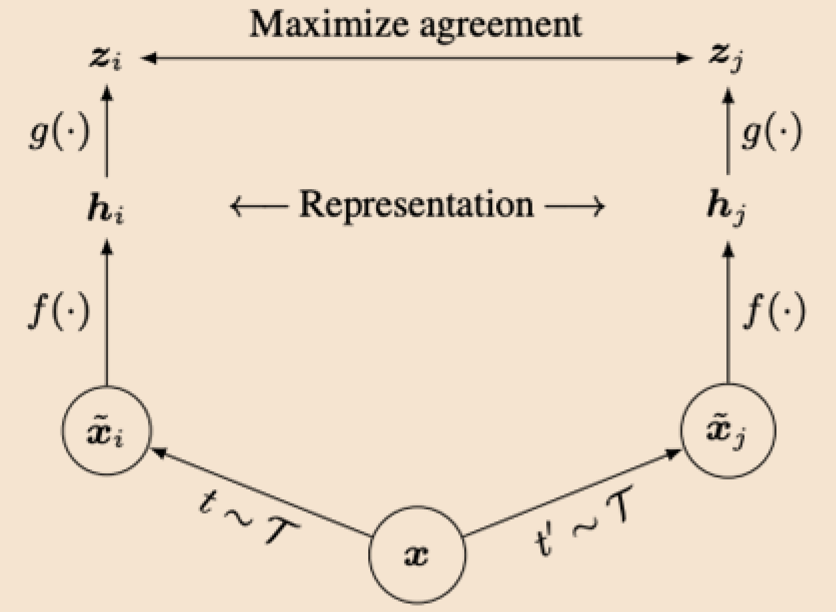


Figure 2. A simple framework for contrastive learning of visual representations. Two separate data augmentation operators are sampled from the same family of augmentations ($t \sim \mathcal{T}$ and $t' \sim \mathcal{T}$) and applied to each data example to obtain two correlated views. A base encoder network $f(\cdot)$ and a projection head $g(\cdot)$ are trained to maximize agreement using a contrastive loss. After training is completed, we throw away the projection head $g(\cdot)$ and use encoder $f(\cdot)$ and representation $\boldsymbol{h}$ for downstream tasks.

Simple Contrastive Learning

Algorithm 1 SimCLR's main learning algorithm.

input: batch size N, constant τ, structure of f, g, $\mathcal{T}$.
for sampled minibatch $\{\boldsymbol{x}_k\}_{k=1}^N$ **do**
  for all $k \in \{1, \ldots, N\}$ **do**
    draw two augmentation functions $t \sim \mathcal{T}$, $t' \sim \mathcal{T}$
    # the first augmentation
    $\tilde{\boldsymbol{x}}_{2k-1} = t(\boldsymbol{x}_k)$
    $\boldsymbol{h}_{2k-1} = f(\tilde{\boldsymbol{x}}_{2k-1})$ # representation
    $\boldsymbol{z}_{2k-1} = g(\boldsymbol{h}_{2k-1})$ # projection
    # the second augmentation
    $\tilde{\boldsymbol{x}}_{2k} = t'(\boldsymbol{x}_k)$
    $\boldsymbol{h}_{2k} = f(\tilde{\boldsymbol{x}}_{2k})$ # representation
    $\boldsymbol{z}_{2k} = g(\boldsymbol{h}_{2k})$ # projection
  end for
  for all $i \in \{1, \ldots, 2N\}$ and $j \in \{1, \ldots, 2N\}$ **do**
    $s_{i,j} = \boldsymbol{z}_i^\top \boldsymbol{z}_j / (\|\boldsymbol{z}_i\| \|\boldsymbol{z}_j\|)$ # pairwise similarity
  end for
  define $\ell(i, j)$ **as** $\ell(i,j) = -\log \frac{\exp(s_{i,j}/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(s_{i,k}/\tau)}$
  $\mathcal{L} = \frac{1}{2N} \sum_{k=1}^{N} [\ell(2k-1, 2k) + \ell(2k, 2k-1)]$
  update networks f and g to minimize $\mathcal{L}$
end for
return encoder network $f(\cdot)$, and throw away $g(\cdot)$

A Simple Framework for Contrastive Learning of Visual Representations

Ting Chen [1] **Simon Kornblith** [1] **Mohammad Norouzi** [1] **Geoffrey Hinton** [1]

2020

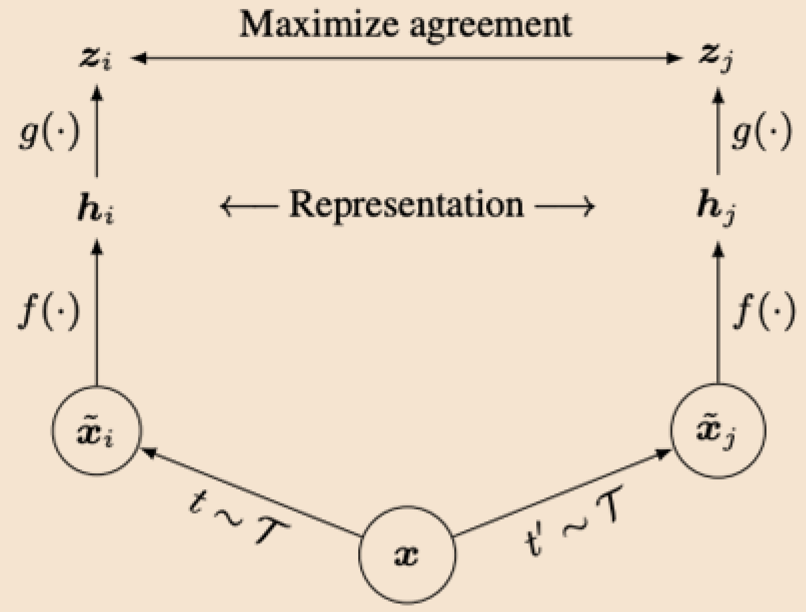


Figure 2. A simple framework for contrastive learning of visual representations. Two separate data augmentation operators are sampled from the same family of augmentations ($t \sim \mathcal{T}$ and $t' \sim \mathcal{T}$) and applied to each data example to obtain two correlated views. A base encoder network $f(\cdot)$ and a projection head $g(\cdot)$ are trained to maximize agreement using a contrastive loss. After training is completed, we throw away the projection head $g(\cdot)$ and use encoder $f(\cdot)$ and representation $\boldsymbol{h}$ for downstream tasks.

Simple Contrastive Learning

A Simple Framework for Contrastive Learning of Visual Representations

Ting Chen[1] Simon Kornblith[1] Mohammad Norouzi[1] Geoffrey Hinton[1]

2020

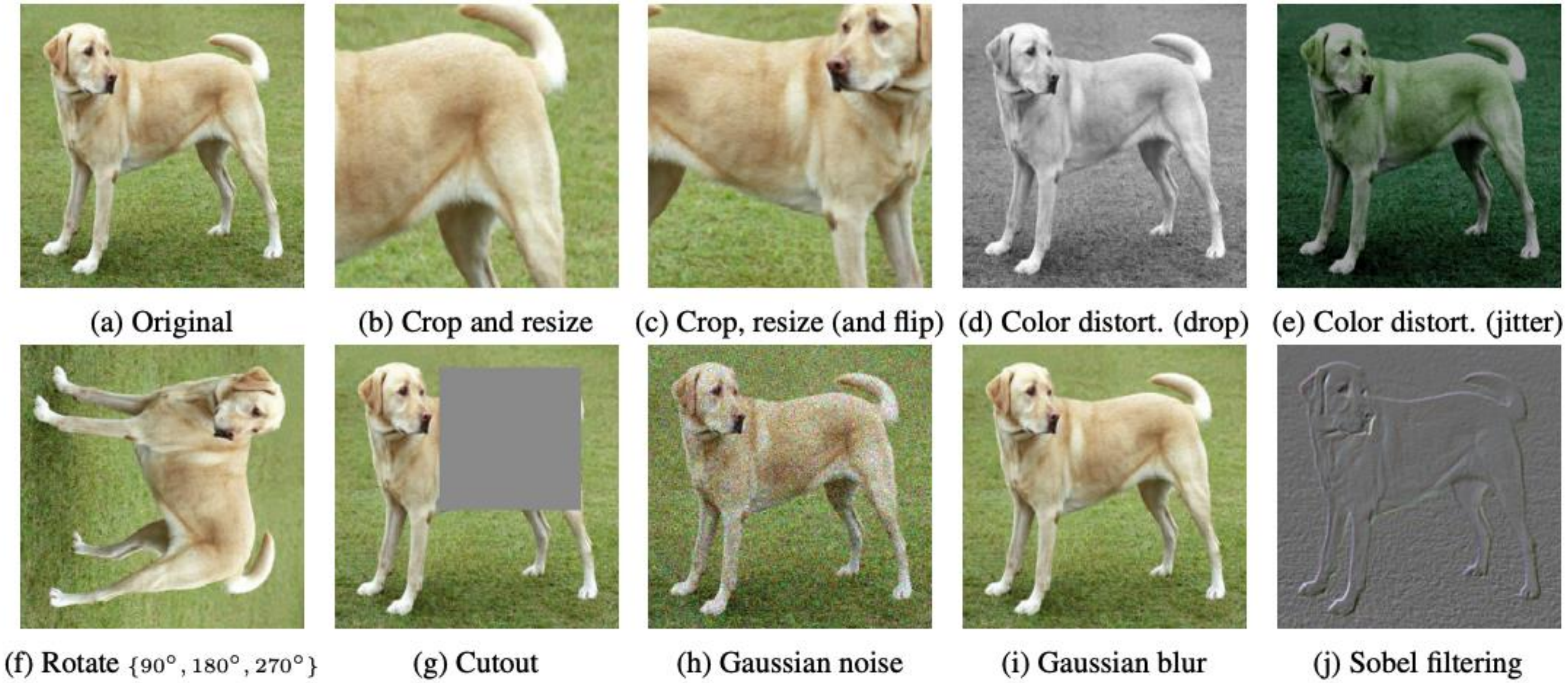


Figure 4. Illustrations of the studied data augmentation operators. Each augmentation can transform data stochastically with some internal parameters (e.g. rotation degree, noise level). Note that we *only* test these operators in ablation, the *augmentation policy used to train our models* only includes *random crop (with flip and resize)*, *color distortion*, and *Gaussian blur*. (Original image cc-by: Von.grzanka)

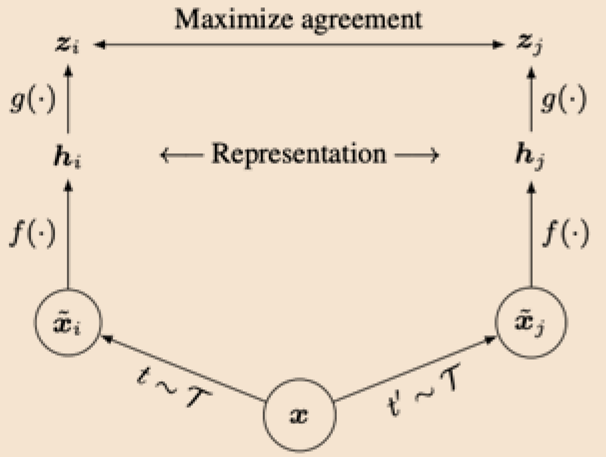


Figure 2. A simple framework for contrastive learning of visual representations. Two separate data augmentation operators are sampled from the same family of augmentations ($t \sim \mathcal{T}$ and $t' \sim \mathcal{T}$) and applied to each data example to obtain two correlated views. A base encoder network $f(\cdot)$ and a projection head $g(\cdot)$ are trained to maximize agreement using a contrastive loss. After training is completed, we throw away the projection head $g(\cdot)$ and use encoder $f(\cdot)$ and representation $\boldsymbol{h}$ for downstream tasks.

Simple Contrastive Learning

A Simple Framework for Contrastive Learning of Visual Representations

Ting Chen [1] **Simon Kornblith** [1] **Mohammad Norouzi** [1] **Geoffrey Hinton** [1]

2020

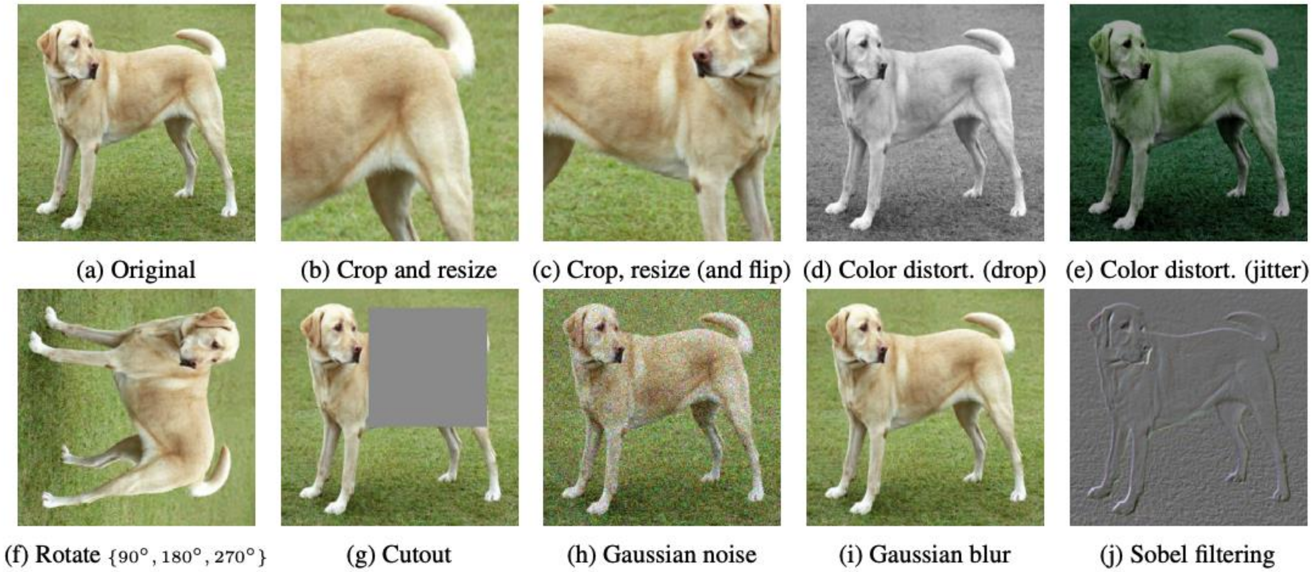


Figure 4. Illustrations of the studied data augmentation operators. Each augmentation can transform data stochastically with some internal parameters (e.g. rotation degree, noise level). Note that we *only* test these operators in ablation, the *augmentation policy used to train our models* only includes *random crop (with flip and resize)*, *color distortion*, and *Gaussian blur*. (Original image cc-by: Von.grzanka)

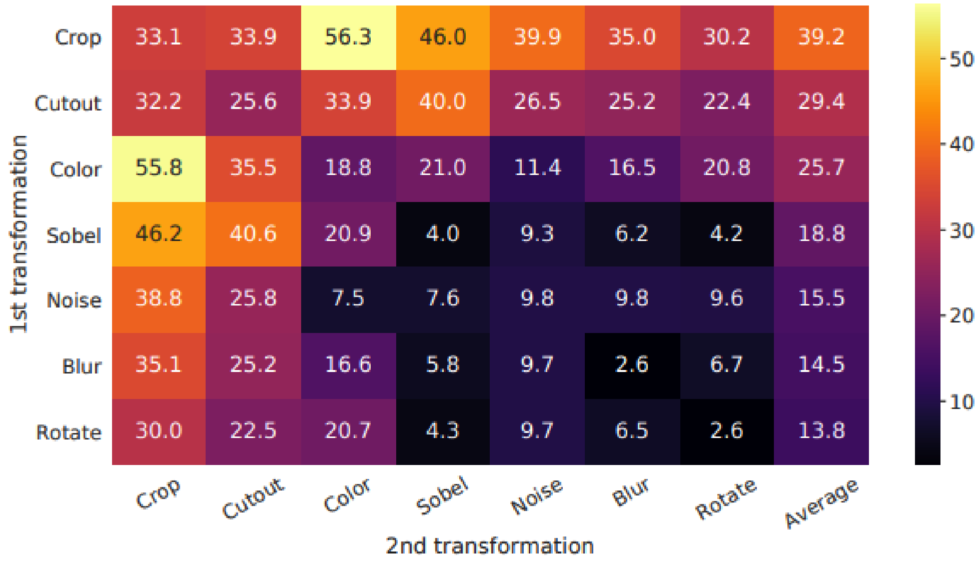


Figure 5. Linear evaluation (ImageNet top-1 accuracy) under individual or composition of data augmentations, applied only to one branch. For all columns but the last, diagonal entries correspond to single transformation, and off-diagonals correspond to composition of two transformations (applied sequentially). The last column reflects the average over the row.

Simple Contrastive Learning

A Simple Framework for Contrastive Learning of Visual Representations

Ting Chen [1] **Simon Kornblith** [1] **Mohammad Norouzi** [1] **Geoffrey Hinton** [1]

2020

Methods	Color distortion strength 1/8	1/4	1/2	1	1 (+Blur)	AutoAug
SimCLR	59.6	61.0	62.6	63.2	64.5	61.1
Supervised	77.0	76.7	76.5	75.7	75.4	77.1

Table 1. Top-1 accuracy of unsupervised ResNet-50 using linear evaluation and supervised ResNet-50[5], under varied color distortion strength (see Appendix A) and other data transformations. Strength 1 (+Blur) is our default data augmentation policy.

	Food	CIFAR10	CIFAR100	Birdsnap	SUN397	Cars	Aircraft	VOC2007	DTD	Pets	Caltech-101	Flowers
Linear evaluation:												
SimCLR (ours)	**76.9**	**95.3**	80.2	48.4	**65.9**	60.0	61.2	**84.2**	**78.9**	89.2	**93.9**	**95.0**
Supervised	75.2	**95.7**	**81.2**	**56.4**	64.9	**68.8**	**63.8**	83.8	**78.7**	**92.3**	**94.1**	94.2
Fine-tuned:												
SimCLR (ours)	**89.4**	**98.6**	**89.0**	**78.2**	**68.1**	**92.1**	**87.0**	**86.6**	**77.8**	92.1	**94.1**	97.6
Supervised	88.7	98.3	**88.7**	**77.8**	67.0	91.4	**88.0**	86.5	**78.8**	**93.2**	**94.2**	**98.0**
Random init	88.3	96.0	81.9	**77.0**	53.7	91.3	84.8	69.4	64.1	82.7	72.5	92.5

Table 8. Comparison of transfer learning performance of our self-supervised approach with supervised baselines across 12 natural image classification datasets, for ResNet-50 (4×) models pretrained on ImageNet. Results not significantly worse than the best ($p > 0.05$, permutation test) are shown in bold. See Appendix B.8 for experimental details and results with standard ResNet-50.

Simple Contrastive Learning

A Simple Framework for Contrastive Learning of Visual Representations

Ting Chen [1] Simon Kornblith [1] Mohammad Norouzi [1] Geoffrey Hinton [1]

2020

- Requires large batchsize

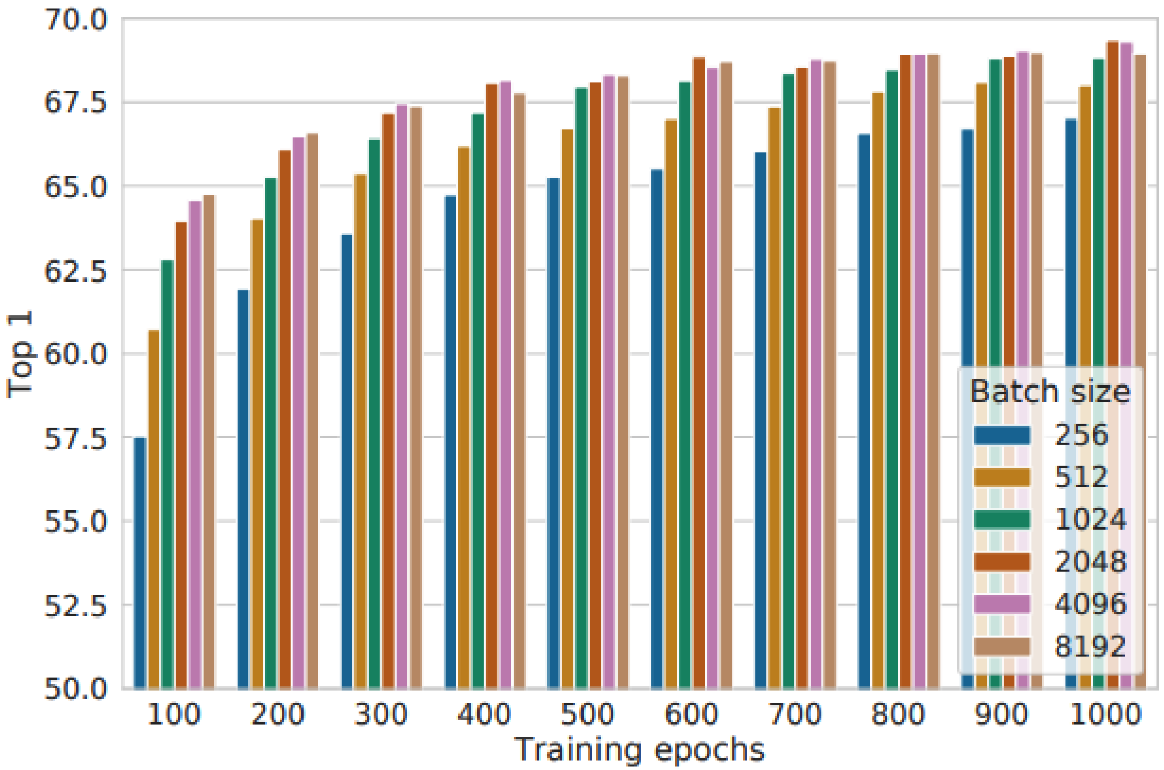


Figure 9. Linear evaluation models (ResNet-50) trained with different batch size and epochs. Each bar is a single run from scratch.[10]

MoCo v2

Improved Baselines with Momentum Contrastive Learning

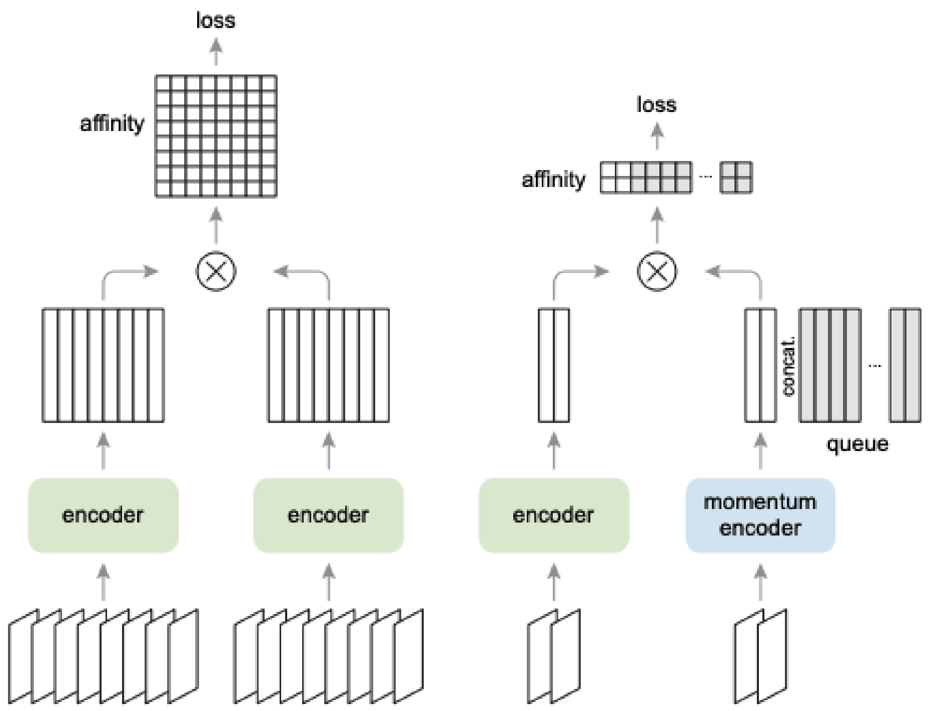
Xinlei Chen Haoqi Fan Ross Girshick Kaiming He 2020

Abstract

Contrastive unsupervised learning has recently shown encouraging progress, e.g., in Momentum Contrast (MoCo) and SimCLR. In this note, we verify the effectiveness of two of SimCLR's design improvements by implementing them in the MoCo framework. With simple modifications to MoCo—namely, using an MLP projection head and more data augmentation—we establish stronger baselines that outperform SimCLR and do not require large training batches. We hope this will make state-of-the-art unsupervised learning research more accessible. Code will be made public.

case	unsup. pre-train					ImageNet
	MLP	aug+	cos	epochs	batch	acc.
MoCo v1 [6]				200	256	60.6
SimCLR [2]	✓	✓	✓	200	256	61.9
SimCLR [2]	✓	✓	✓	200	8192	66.6
MoCo v2	✓	✓	✓	200	256	**67.5**
*results of **longer** unsupervised training follow:*						
SimCLR [2]	✓	✓	✓	1000	4096	69.3
MoCo v2	✓	✓	✓	800	256	**71.1**

Table 2. **MoCo *vs.* SimCLR**: ImageNet linear classifier accuracy (**ResNet-50, 1-crop 224×224**), trained on features from unsupervised pre-training. "aug+" in SimCLR includes blur and stronger color distortion. SimCLR ablations are from Fig. 9 in [2] (we thank the authors for providing the numerical results).



(a) end-to-end (b) Momentum Contrast

Figure 1. A **batching** perspective of two optimization mechanisms for contrastive learning. Images are encoded into a representation space, in which pairwise affinities are computed.

An Empirical Study of Training Self-Supervised Vision Transformers

Xinlei Chen* Saining Xie* Kaiming He

2021

MoCo v3 = MoCo v2 with ViT

framework	model	params	acc. (%)
linear probing:			
iGPT [9]	iGPT-L	1362M	69.0
iGPT [9]	iGPT-XL	6801M	72.0
MoCo v3	ViT-B	86M	76.7
MoCo v3	ViT-L	304M	77.6
MoCo v3	ViT-H	632M	78.1
MoCo v3	ViT-BN-H	632M	79.1
MoCo v3	ViT-BN-L/7	304M	**81.0**
end-to-end fine-tuning:			
masked patch pred. [16]	ViT-B	86M	79.9†
MoCo v3	ViT-B	86M	83.2
MoCo v3	ViT-L	304M	**84.1**

Table 1. **State-of-the-art Self-supervised Transformers** in ImageNet classification, evaluated by linear probing (top panel) or end-to-end fine-tuning (bottom panel). Both iGPT [9] and masked patch prediction [16] belong to the masked auto-encoding paradigm. MoCo v3 is a contrastive learning method that compares two (224×224) crops. ViT-B, -L, -H are the Vision Transformers proposed in [16]. ViT-BN is modified with BatchNorm, and "/7" denotes a patch size of 7×7. †: pre-trained in JFT-300M.

Bootstrap Your Own Latent (BYOL – Grill et al., 2020)

- Does not use negative samples

Given an image $\mathbf{x}$, the BYOL loss is constructed as follows:

- Create two augmented views: $\mathbf{v} = t(\mathbf{x}); \mathbf{v}' = t'(\mathbf{x})$ with augmentations sampled $t \sim \mathcal{T}, t' \sim \mathcal{T}'$;
- Then they are encoded into representations, $\mathbf{y}_\theta = f_\theta(\mathbf{v}), \mathbf{y}' = f_\xi(\mathbf{v}')$;
- Then they are projected into latent variables, $\mathbf{z}_\theta = g_\theta(\mathbf{y}_\theta), \mathbf{z}' = g_\xi(\mathbf{y}')$;
- The online network outputs a prediction $q_\theta(\mathbf{z}_\theta)$;
- Both $q_\theta(\mathbf{z}_\theta)$ and $\mathbf{z}'$ are L2-normalized, giving us $\bar{q}_\theta(\mathbf{z}_\theta) = q_\theta(\mathbf{z}_\theta)/\|q_\theta(\mathbf{z}_\theta)\|$ and $\bar{\mathbf{z}}' = \mathbf{z}'/\|\mathbf{z}'\|$;
- The loss $\mathcal{L}_\theta^{\text{BYOL}}$ is MSE between L2-normalized prediction $\bar{q}_\theta(\mathbf{z})$ and $\bar{\mathbf{z}}'$;
- The other symmetric loss $\tilde{\mathcal{L}}_\theta^{\text{BYOL}}$ can be generated by switching $\mathbf{v}'$ and $\mathbf{v}$; that is, feeding $\mathbf{v}'$ to online network and $\mathbf{v}$ to target network.
- The final loss is $\mathcal{L}_\theta^{\text{BYOL}} + \tilde{\mathcal{L}}_\theta^{\text{BYOL}}$ and only parameters θ are optimized.

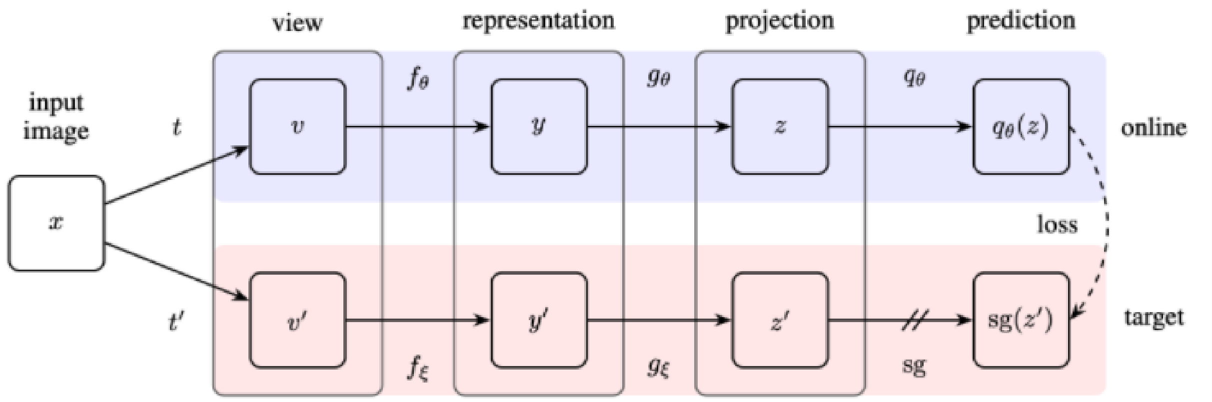


Fig. 10. The model architecture of BYOL. After training, we only care about f_θ for producing representation, $y = f_\theta(x)$, and everything else is discarded. sg means stop gradient. (Image source: Grill, et al 2020)

$$\xi \leftarrow \tau\xi + (1 - \tau)\theta.$$

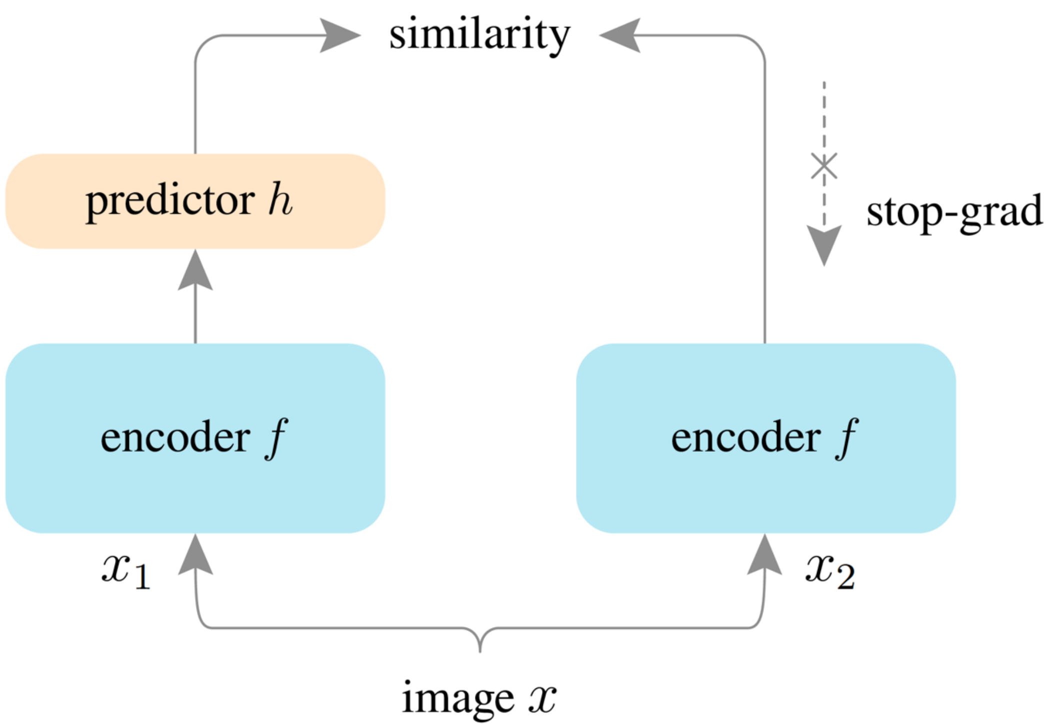
https://lilianweng.github.io/lil-log/2021/05/31/contrastive-representation-learning.html

Simple Siamese Representation Learning (SimSiam – Chen et al., 2020)

- "BYOL without momentum encoder"

Algorithm 1 SimSiam Pseudocode, PyTorch-like

```
# f: backbone + projection mlp
# h: prediction mlp

for x in loader: # load a minibatch x with n samples
    x1, x2 = aug(x), aug(x) # random augmentation
    z1, z2 = f(x1), f(x2) # projections, n-by-d
    p1, p2 = h(z1), h(z2) # predictions, n-by-d

    L = D(p1, z2)/2 + D(p2, z1)/2 # loss

    L.backward() # back-propagate
    update(f, h) # SGD update

def D(p, z): # negative cosine similarity
    z = z.detach() # stop gradient

    p = normalize(p, dim=1) # l2-normalize
    z = normalize(z, dim=1) # l2-normalize
    return -(p*z).sum(dim=1).mean()
```



https://github.com/facebookresearch/simsiam

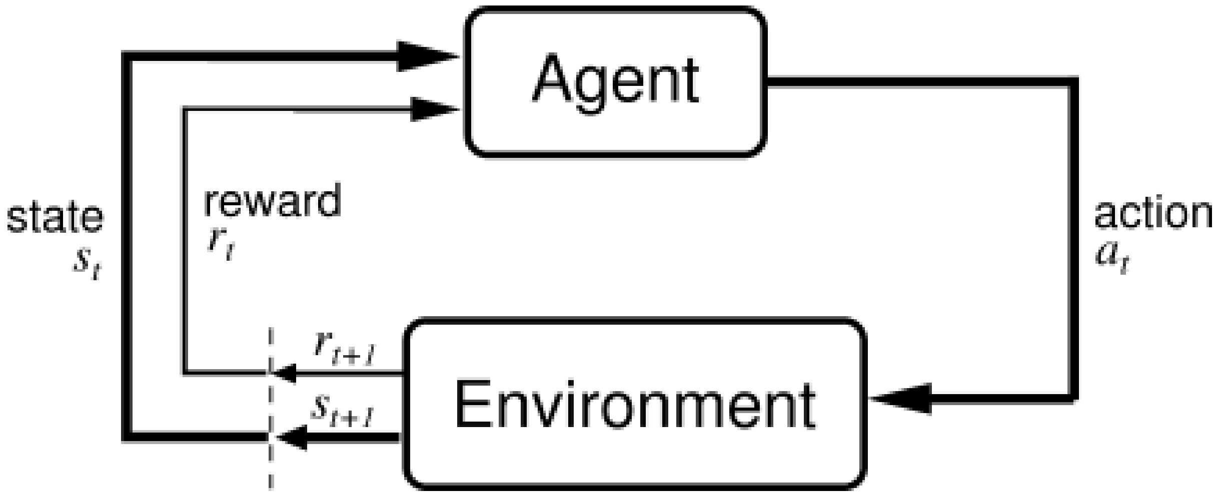
Resources on SSL

- The rise of SSL, by Y. Lecun: https://www.youtube.com/watch?v=05wUrb5Ej8Q&t=21252s

- Self-supervised representation learning: https://lilianweng.github.io/lil-log/2019/11/10/self-supervised-learning.html

Deep reinforcement learning

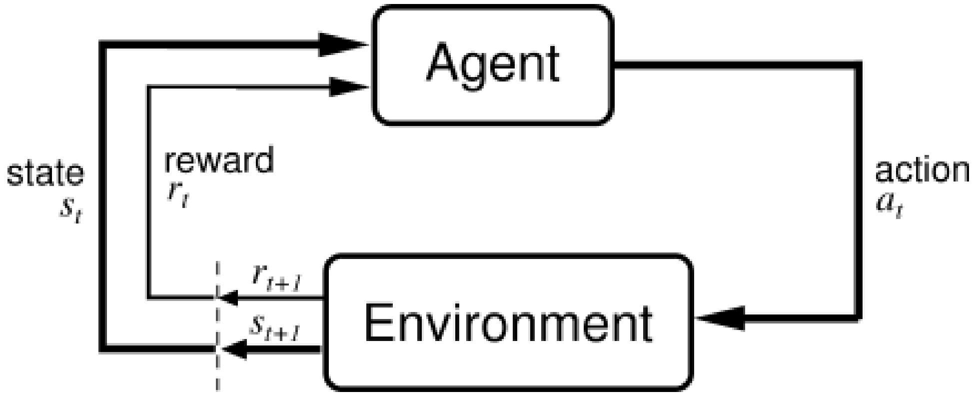
Reinforcement Learning



The agent receives reward r_t for its actions.

More formally

- An agent's behavior is defined by a policy, π:
  $$\pi: \mathcal{S} \rightarrow P(\mathcal{A})$$
  $\mathcal{S}$: The space of states.
  $\mathcal{A}$: The space of actions.
- The "return" from a state is usually:
  $$R_t = \sum_{i=t}^{T} \gamma^{(i-t)} r(s_i, a_i)$$
  $r(s_i, a_i)$: the reward for action a_i in state s_i.
  γ: discount factor.
- Goal: Learn a policy that maximizes the expected return from the starting position:
  $$\mathbb{E}_{r_i, s_i \sim E, a_i \sim \pi}[R_1]$$



http://www.cs.ubc.ca/~murphyk/Bayes/pomdp.html

More formally

- We can define an expected return for taking action a_t at state s_t:

$$Q^\pi(s_t, a_t) = \mathbb{E}_{r_{i \geq t}, s_{i>t} \sim E, a_{i>t} \sim \pi}\left[R_t \mid s_t, a_t\right]$$

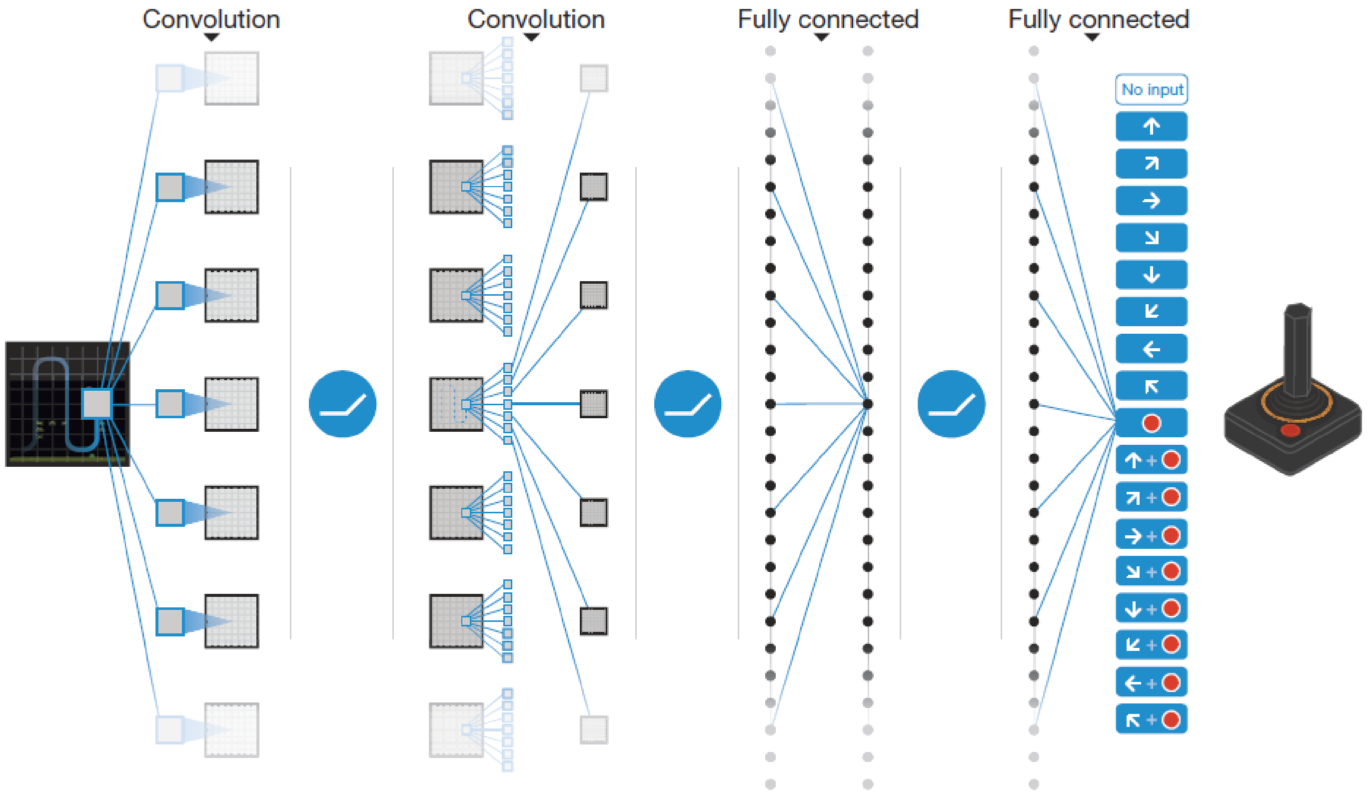
- This can be rewritten as (called the Bellman equation):

$$Q^\pi(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E}\left[r(s_t, a_t) + \gamma\, \mathbb{E}_{a_{t+1} \sim \pi}\left[Q^\pi(s_{t+1}, a_{t+1})\right]\right]$$

http://www.cs.ubc.ca/~murphyk/Bayes/pomdp.html

Reinforcement Learning with Deep Networks

- Two general approaches:
  - Value gradients
  - Policy gradients



Q values of actions are predicted at the output.

Figure 1 | Schematic illustration of the convolutional neural network. The details of the architecture are explained in the Methods. The input to the neural network consists of an 84 × 84 × 4 image produced by the preprocessing map ϕ, followed by three convolutional layers (note: snaking blue line symbolizes sliding of each filter across input image) and two fully connected layers with a single output for each valid action. Each hidden layer is followed by a rectifier nonlinearity (that is, $\max(0,x)$).

2015

LETTER

doi:10.1038/nature14236

Human-level control through deep reinforcement learning

Volodymyr Mnih[1]*, Koray Kavukcuoglu[1]*, David Silver[1]*, Andrei A. Rusu[1], Joel Veness[1], Marc G. Bellemare[1], Alex Graves[1], Martin Riedmiller[1], Andreas K. Fidjeland[1], Georg Ostrovski[1], Stig Petersen[1], Charles Beattie[1], Amir Sadik[1], Ioannis Antonoglou[1], Helen King[1], Dharshan Kumaran[1], Daan Wierstra[1], Shane Legg[1] & Demis Hassabis[1]

network. We refer to a neural network function approximator with weights θ as a Q-network. A Q-network can be trained by adjusting the parameters θ_i at iteration i to reduce the mean-squared error in the Bellman equation, where the optimal target values $r+\gamma \max_{a'} Q^*(s',a')$ are substituted with approximate target values $y=r+\gamma \max_{a'} Q(s',a';\theta_i^-)$, using parameters θ_i^- from some previous iteration. This leads to a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i,

$$L_i(\theta_i)=\mathbb{E}_{s,a,r}\left[\left(\mathbb{E}_{s'}[y|s,a]-Q(s,a;\theta_i)\right)^2\right]$$

LETTER

doi:10.1038/nature14236

Human-level control through deep reinforcement learning

Volodymyr Mnih[1]*, Koray Kavukcuoglu[1]*, David Silver[1]*, Andrei A. Rusu[1], Joel Veness[1], Marc G. Bellemare[1], Alex Graves[1], Martin Riedmiller[1], Andreas K. Fidjeland[1], Georg Ostrovski[1], Stig Petersen[1], Charles Beattie[1], Amir Sadik[1], Ioannis Antonoglou[1], Helen King[1], Dharshan Kumaran[1], Daan Wierstra[1], Shane Legg[1] & Demis Hassabis[1]

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode = 1, M **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For $t = 1$,T **do**

    With probability ε select a random action a_t

    otherwise select $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

    Execute action a_t in emulator and observe reward r_t and image x_{t+1}

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

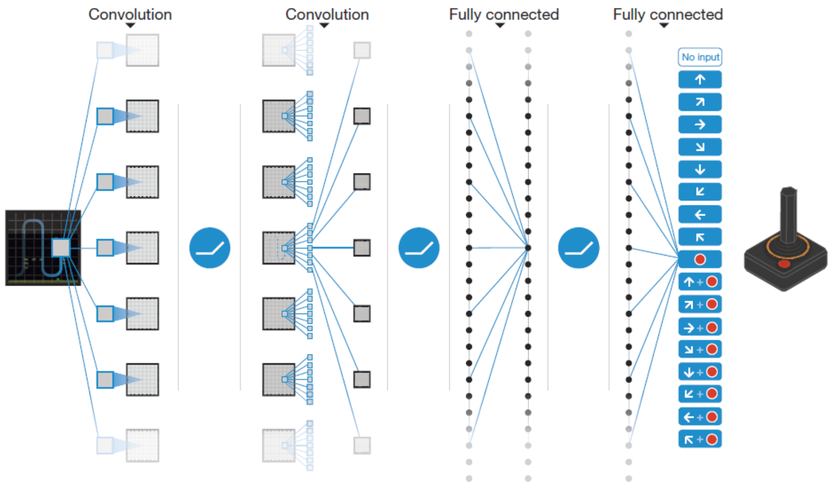
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

    Every C steps reset $\hat{Q} = Q$

  End For

End For

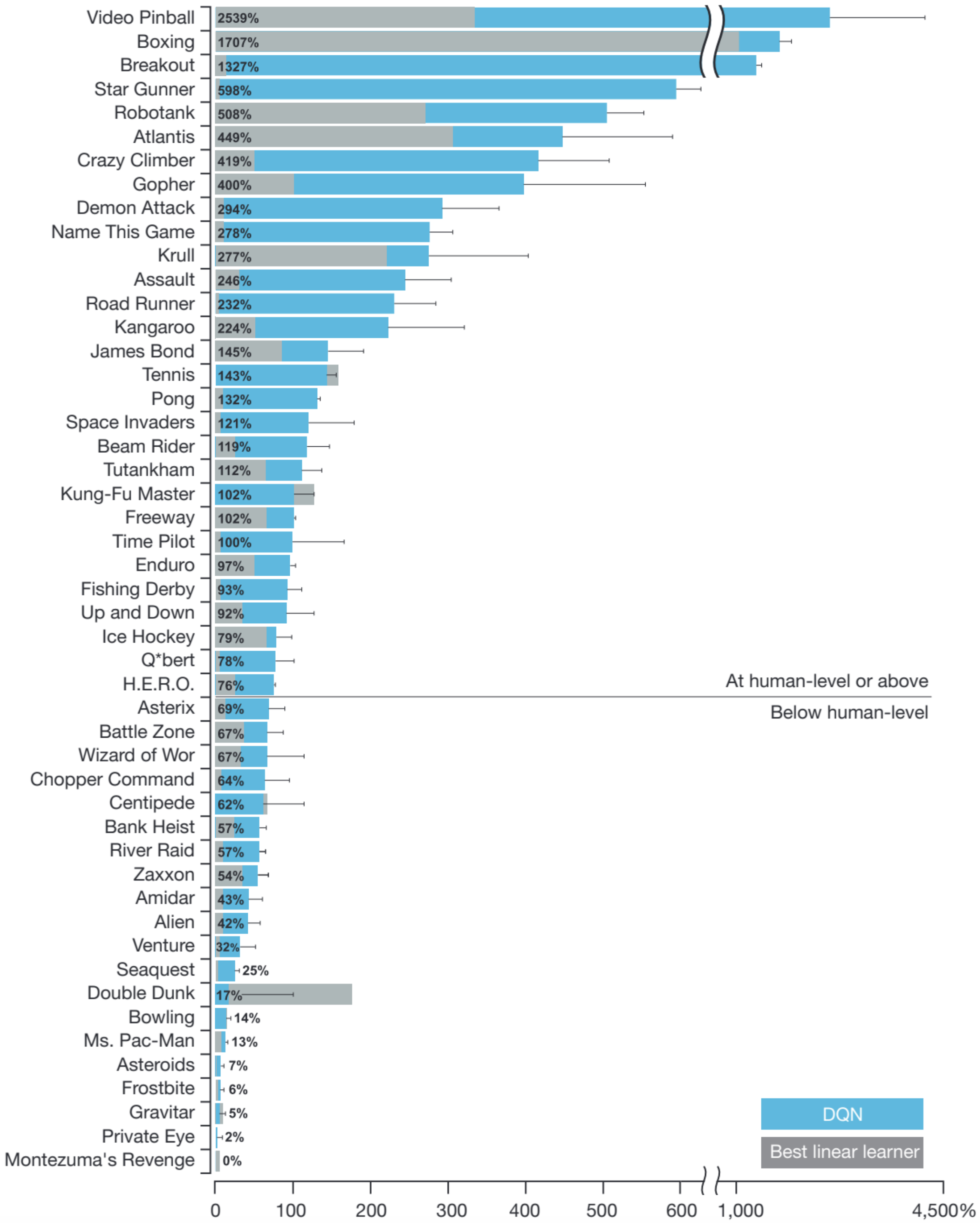


LETTER

doi:10.1038/nature14236

Human-level control through deep reinforcement learning

Volodymyr Mnih[1]*, Koray Kavukcuoglu[1]*, David Silver[1]*, Andrei A. Rusu[1], Joel Veness[1], Marc G. Bellemare[1], Alex Graves[1], Martin Riedmiller[1], Andreas K. Fidjeland[1], Georg Ostrovski[1], Stig Petersen[1], Charles Beattie[1], Amir Sadik[1], Ioannis Antonoglou[1], Helen King[1], Dharshan Kumaran[1], Daan Wierstra[1], Shane Legg[1] & Demis Hassabis[1]



LETTER

doi:10.1038/nature14236

Human-level control through deep reinforcement learning

Volodymyr Mnih[1]*, Koray Kavukcuoglu[1]*, David Silver[1]*, Andrei A. Rusu[1], Joel Veness[1], Marc G. Bellemare[1], Alex Graves[1], Martin Riedmiller[1], Andreas K. Fidjeland[1], Georg Ostrovski[1], Stig Petersen[1], Charles Beattie[1], Amir Sadik[1], Ioannis Antonoglou[1], Helen King[1], Dharshan Kumaran[1], Daan Wierstra[1], Shane Legg[1] & Demis Hassabis[1]

Deep Reinforcement Learning with Double Q-learning

Double DQN

Hado van Hasselt and **Arthur Guez** and **David Silver** 2015
Google DeepMind

Problem with DQN (and Q learning):

- Over-optimistic estimation owing to the max because the environment is noisy

Q-learning

$Q(s, a; \boldsymbol{\theta}_t)$. The standard Q-learning update for the parameters after taking action A_t in state S_t and observing the immediate reward R_{t+1} and resulting state S_{t+1} is then

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha(Y_t^{\mathrm{Q}} - Q(S_t, A_t; \boldsymbol{\theta}_t))\nabla_{\boldsymbol{\theta}_t} Q(S_t, A_t; \boldsymbol{\theta}_t) . \quad (1)$$

where α is a scalar step size and the target Y_t^{Q} is defined as

$$Y_t^{\mathrm{Q}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{\theta}_t) . \quad (2)$$

This update resembles stochastic gradient descent, updating the current value $Q(S_t, A_t; \boldsymbol{\theta}_t)$ towards a target value Y_t^{Q}.

DQN

work, and the use of experience replay. The target network, with parameters $\boldsymbol{\theta}^-$, is the same as the online network except that its parameters are copied every τ steps from the online network, so that then $\boldsymbol{\theta}_t^- = \boldsymbol{\theta}_t$, and kept fixed on all other steps. The target used by DQN is then

$$Y_t^{\mathrm{DQN}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{\theta}_t^-) . \quad (3)$$

Double DQN

Deep Reinforcement Learning with Double Q-learning

Hado van Hasselt and **Arthur Guez** and **David Silver**
Google DeepMind

2015

Solution

- Separate action selection (actor) from action evaluation (critic)

Double Q-learning

The Double Q-learning error can then be written as

$$Y_t^{\text{DoubleQ}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}'_t) .$$

Double DQN (DDQN)

to the resulting algorithm as Double DQN. Its update is the same as for DQN, but replacing the target Y_t^{DQN} with

$$Y_t^{\text{DoubleDQN}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(S_{t+1}, a; \boldsymbol{\theta}_t), \boldsymbol{\theta}_t^-) .$$

In comparison to Double Q-learning (4), the weights of the second network $\boldsymbol{\theta}'_t$ are replaced with the weights of the target network $\boldsymbol{\theta}_t^-$ for the evaluation of the current greedy policy. The update to the target network stays unchanged from DQN, and remains a periodic copy of the online network.

Double DQN

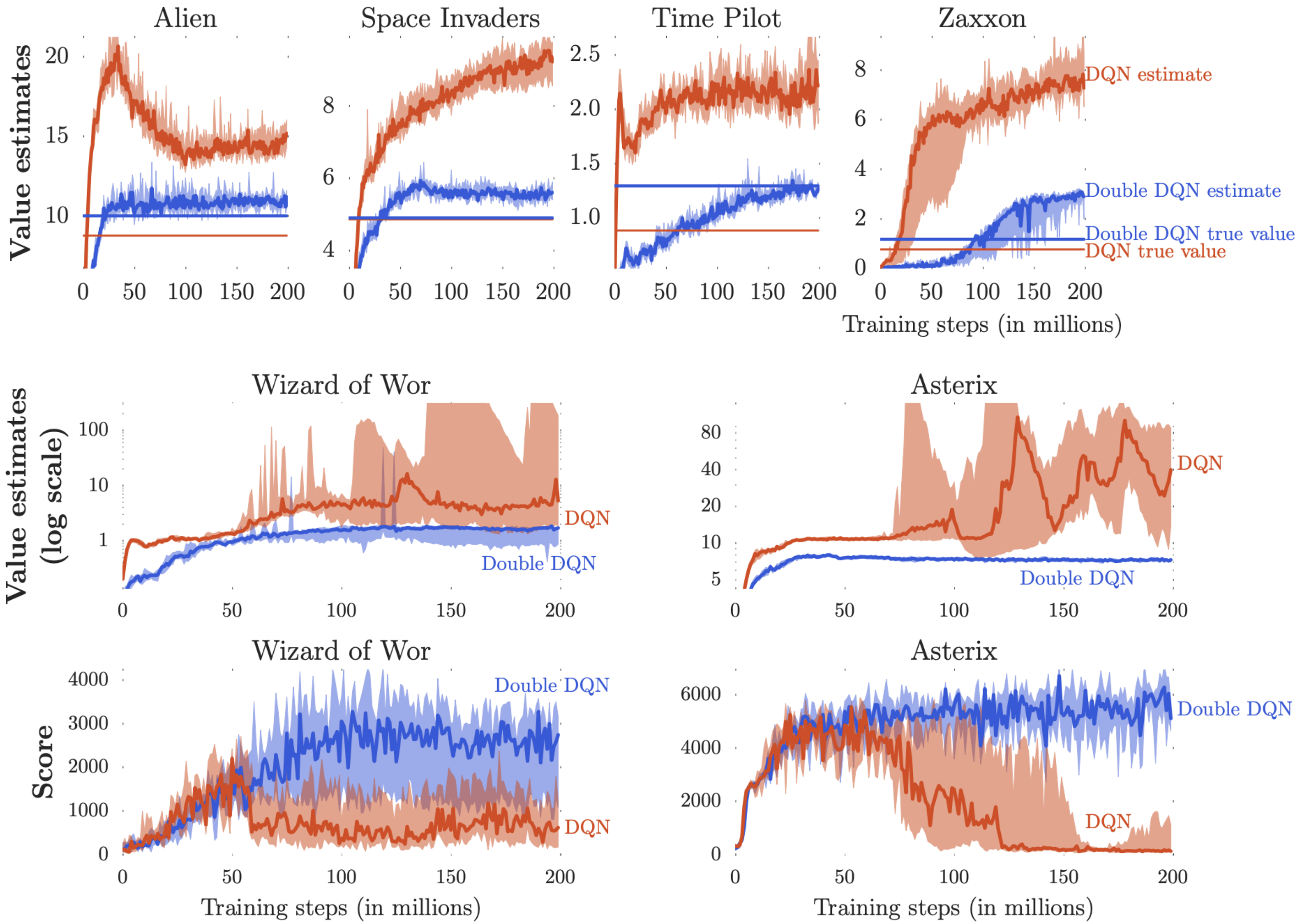
Deep Reinforcement Learning with Double Q-learning

Hado van Hasselt and **Arthur Guez** and **David Silver**
Google DeepMind

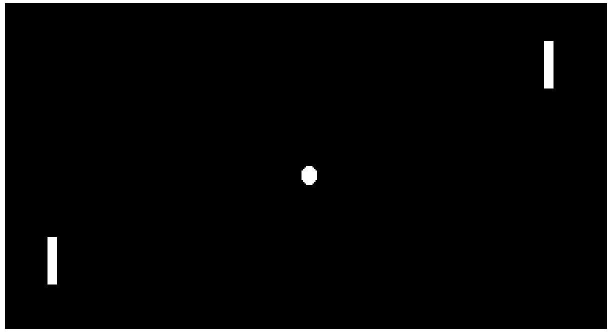
2015

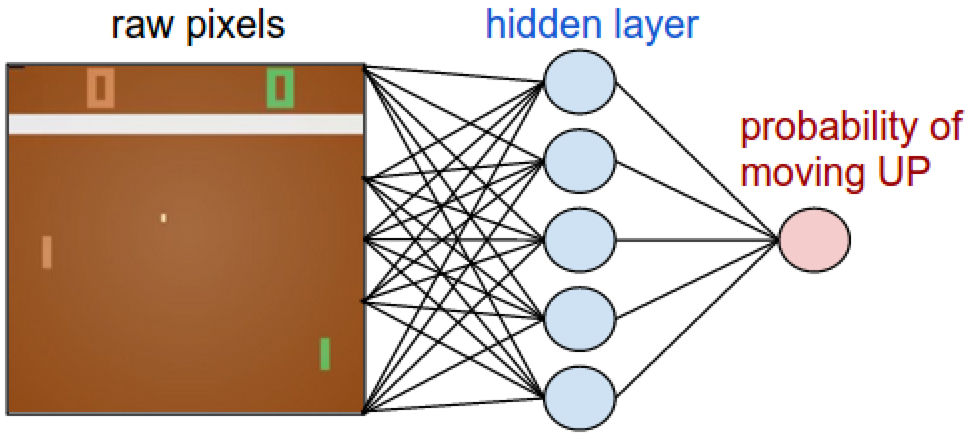
	DQN	Double DQN	Double DQN (tuned)
Median	47.5%	88.4%	116.7%
Mean	122.0%	273.1%	475.2%

Table 2: Summary of normalized performance up to 30 minutes of play on 49 games with human starts. Results for DQN are from Nair et al. (2015).



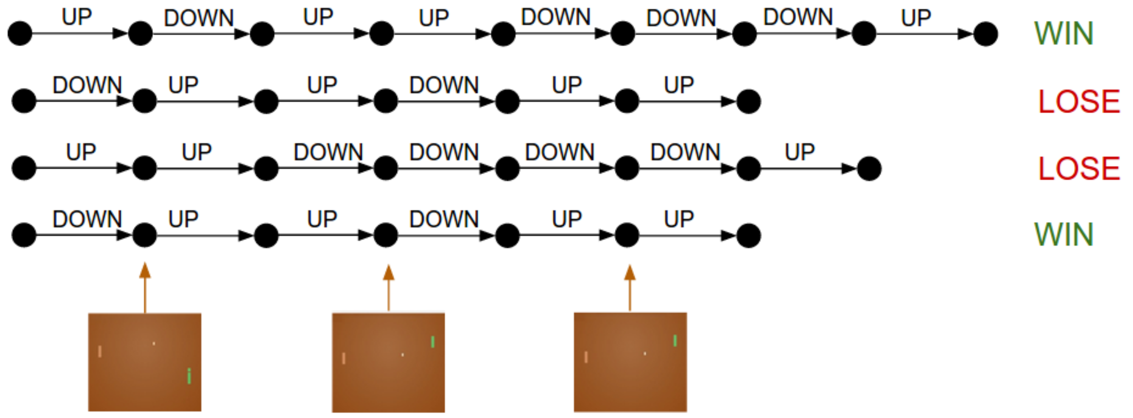
Policy gradients

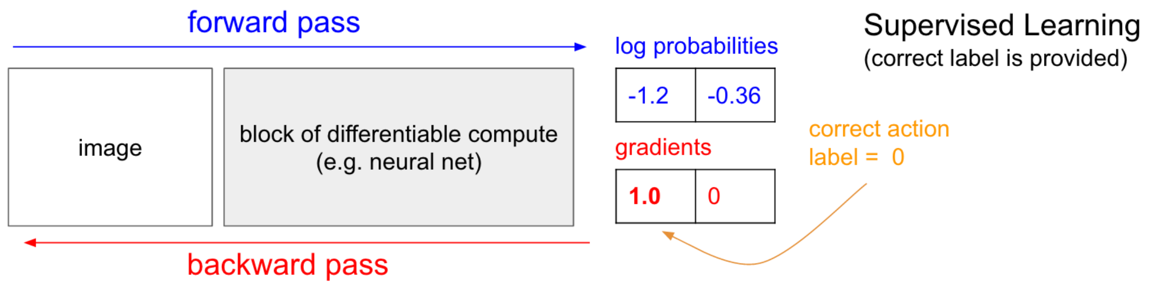




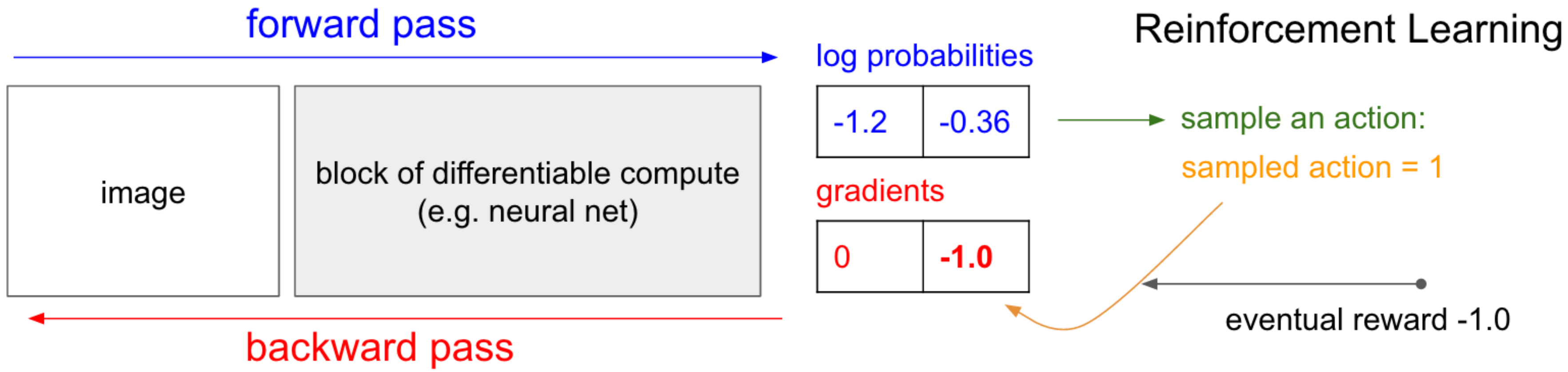
http://karpathy.github.io/2016/05/31/rl/

Policy gradients





$$\nabla_W \log p(y = UP \mid x)$$



http://karpathy.github.io/2016/05/31/rl/

Policy gradients

Let us start with the defined objective function $J(\theta)$. We can expand the expectation as:

$$J(\theta) = \mathbb{E}[\sum_{t=0}^{T-1} r_{t+1} | \pi_\theta]$$
$$= \sum_{t=i}^{T-1} P(s_t, a_t | \tau) r_{t+1}$$

where i is an arbitrary starting point in a trajectory, $P(s_t, a_t|\tau)$ is the probability of the occurrence of s_t, a_t given the trajectory τ.

https://medium.com/@thechrisyoon/deriving-policy-gradients-and-implementing-reinforce-f887949bd63

Policy gradients

Differentiate both sides with respect to policy parameter θ:

$$\text{Using} \quad \frac{d}{dx} log f(x) = \frac{f'(x)}{f(x)},$$

$$\nabla_\theta J(\theta) = \sum_{t=i}^{T-1} \nabla_\theta P(s_t, a_t|\tau) r_{t+1}$$

$$= \sum_{t=i}^{T-1} P(s_t, a_t|\tau) \frac{\nabla_\theta P(s_t, a_t|\tau)}{P(s_t, a_t|\tau)} r_{t+1}$$

$$= \sum_{t=i}^{T-1} P(s_t, a_t|\tau) \nabla_\theta log P(s_t, a_t|\tau) r_{t+1}$$

$$= \mathbb{E}[\sum_{t=i}^{T-1} \nabla_\theta log P(s_t, a_t|\tau) r_{t+1}]$$

→ This however does not depend on the policy network

https://medium.com/@thechrisyoon/deriving-policy-gradients-and-implementing-reinforce-f887949bd63

Policy gradients

By rewriting the probability as:

$$P(s_t, a_t|\tau) = P(s_0, a_0, s_1, a_2, ..., s_{t-1}, a_{t-1}, s_t, a_t|\pi_\theta)$$
$$= P(s_0)\pi_\theta(a_1|s_0)P(s_1|s_0, a_0)\pi_\theta(a_2|s_1)P(s_2|s_1, a_1)\pi_\theta(a_3|s_2)$$
$$...P(s_{t-1}|s_{t-2}, a_{t-2})\pi_\theta(a_{t-1}|s_{t-2})P(s_t|s_{t-1}, a_{t-1})\pi_\theta(a_t|s_{t-1})$$

Taking the logarithm and the derivative:

$$\nabla_\theta log P(s_t, a_t|\tau) = 0 + \nabla_\theta log \pi_\theta(a_1|s_0) + 0 + \nabla_\theta log \pi_\theta(a_2|s_1) + 0 + \nabla_\theta log \pi_\theta(a_3|s_2) +$$
$$... + 0 + \nabla_\theta log \pi_\theta(a_{t-1}|s_{t-2}) + 0$$
$$= \nabla_\theta log \pi_\theta(a_1|s_0) + \nabla_\theta log \pi_\theta(a_2|s_1) + \nabla_\theta log \pi_\theta(a_3|s_2) +$$
$$... + \nabla_\theta log \pi_\theta(a_{t-1}|s_{t-2}) + log \pi_\theta(a_t|s_{t-1})$$
$$= \sum_{t'=0}^{t} \nabla_\theta log \pi_\theta(a_{t'}|s_{t'})$$

https://medium.com/@thechrisyoon/deriving-policy-gradients-and-implementing-reinforce-f887949bd63

Policy gradients

Incorporating the discount factor $\gamma \in [0,1]$ into our objective (in order to weight immediate rewards more than future rewards):

$$J(\theta) = \mathbb{E}[\gamma^0 r_1 + \gamma^1 r_2 + \gamma^2 r_3 + ... + \gamma^{T-1} r_T | \pi_\theta]$$

We can perform a similar derivation to obtain

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T-1} \nabla_\theta log \pi_\theta(a_t|s_t) (\sum_{t'=t+1}^{T} \gamma^{t'-t-1} r_{t'})$$

and simplifying $\sum_{t'=t+1}^{T} \gamma^{t'-t-1} r_{t'}$ to G_t,

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T-1} \nabla_\theta log \pi_\theta(a_t|s_t) G_t$$

https://medium.com/@thechrisyoon/deriving-policy-gradients-and-implementing-reinforce-f887949bd63

Actor-Critic Networks

Two main components in policy gradient are the policy model and the value function. It makes a lot of sense to learn the value function in addition to the policy, since knowing the value function can assist the policy update, such as by reducing gradient variance in vanilla policy gradients, and that is exactly what the **Actor-Critic** method does.

Actor-critic methods consist of two models, which may optionally share parameters:

- **Critic** updates the value function parameters w and depending on the algorithm it could be action-value $Q_w(a|s)$ or state-value $V_w(s)$.
- **Actor** updates the policy parameters θ for $\pi_\theta(a|s)$, in the direction suggested by the critic.

https://lilianweng.github.io/posts/2018-04-08-policy-gradient/

Actor-Critic Networks

1. Initialize s, θ, w at random; sample $a \sim \pi_\theta(a|s)$.
2. For $t = 1 \ldots T$:
   1. Sample reward $r_t \sim R(s, a)$ and next state $s' \sim P(s'|s, a)$;
   2. Then sample the next action $a' \sim \pi_\theta(a'|s')$;
   3. Update the policy parameters: $\theta \leftarrow \theta + \alpha_\theta Q_w(s, a) \nabla_\theta \ln \pi_\theta(a|s)$;
   4. Compute the correction (TD error) for action-value at time t:
      $\delta_t = r_t + \gamma Q_w(s', a') - Q_w(s, a)$
      and use it to update the parameters of action-value function:
      $w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s, a)$
   5. Update $a \leftarrow a'$ and $s \leftarrow s'$.

https://lilianweng.github.io/posts/2018-04-08-policy-gradient/

Actor-Critic Networks

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\, G_t]$$ REINFORCE

$$= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\ Q^w(s,a)]$$ Q Actor-Critic

$$= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\ A^w(s,a)]$$ Advantage Actor-Critic

$$= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\ \delta]$$ TD Actor-Critic

From CMU CS10703 lecture slides

Introducing baseline $b(s)$:

$$\nabla_\theta J(\theta) = \mathbb{E}[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)(G_t - b(s_t))]$$

https://medium.com/@thechrisyoon/deriving-policy-gradients-and-implementing-reinforce-f887949bd63

Actor-Critic Networks

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\, G_t]$$ REINFORCE

$$= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\; Q^w(s,a)]$$ Q Actor-Critic

$$= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\; A^w(s,a)]$$ Advantage Actor-Critic

$$= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\; \delta]$$ TD Actor-Critic

From CMU CS10703 lecture slides

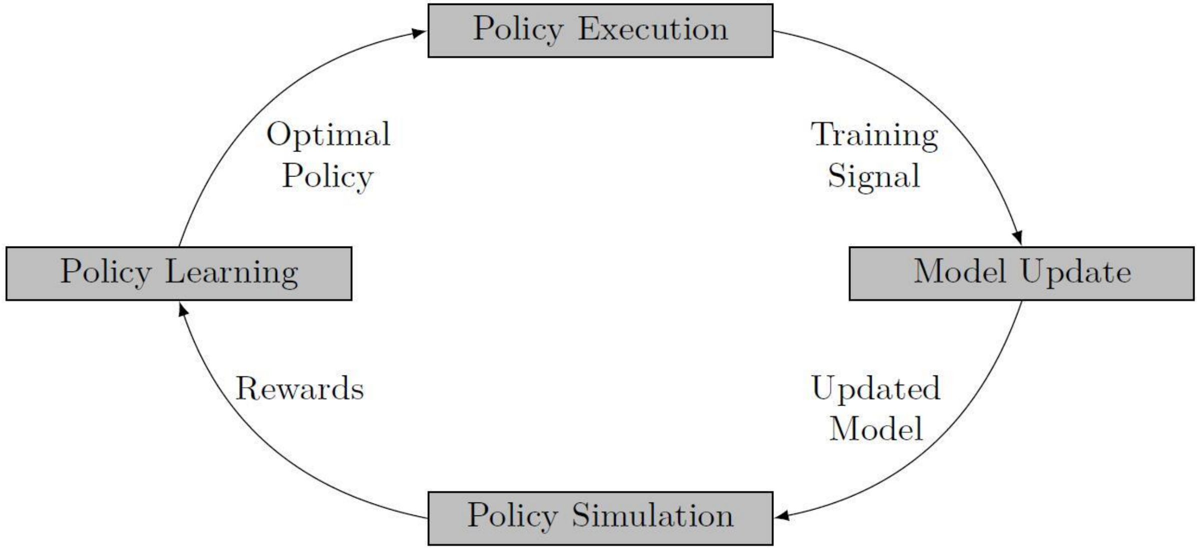
$$A(s_t, a_t) = Q_w(s_t, a_t) - V_v(s_t)$$
$$= r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t)$$

$$V^\pi(s) = E_{a\sim\pi}\left\{\sum_{t=0}^{\infty} \gamma^t R_{t+1} \middle| S_0 = s\right\}$$

https://medium.com/@thechrisyoon/deriving-policy-gradients-and-implementing-reinforce-f887949bd63

Model-based vs. Model-free

- Model-free
  - Learn the Q values
- Model-based
  - Learn the Q values and the transition probabilities (the model of how the environment would change)



RL Methods	Advantages	Disadvantages
Model-based RL	– Small number of interactions between robot & environment – Faster convergence to optimal solution.	–Depend on transition models – Model accuracy has a big impact on learning tasks
Model-free RL	– No need for prior knowledge of transitions – Easily implementable	– Slow learning convergence – High wear & tear of the robot – High risk of damage

Table & Fig: https://www.chenshiyu.top/blog/2019/06/12/An-Overview-of-Model-Based-Reinforcement-Learning/

On-policy vs. Off-policy

This brings us to the key difference between on-policy and off-policy learning: ***On-policy algorithms attempt to improve upon the current behavior policy that is used to make decisions and therefore these algorithms learn the value of the policy carried out by the agent, Q^{π}. Off-policy algorithms learn the value of the optimal policy, Q^{*}, and can improve upon a policy that is different from the behavior policy.*** Determining if the update and behavior policy are the same or different can give us insight into whether or not the algorithm is on-policy or off-policy. If the update policy and the behavior policy are the same, then this suggest but does not guarantee that the learning method is on-policy. If they are different, this suggests that the learning method is off-policy.

	On-Policy	Off-Policy
Advantages	• Learns safer strategy • Often converges faster • Often has better online performance	• More likely to find optimal policy • Less likely to get stuck in local minimum • Can utilize experience replay • Data can be collected via various method
Disadvantages	• May become trapped in local minima • Less likely to find optimal policy • Data must be collected following current policy	• Policy learned may not be as safe • May not perform as well online

https://core-robotics.gatech.edu/2022/02/28/bootcamp-summer-2020-week-4-on-policy-vs-off-policy-reinforcement-learning/

Today

- (Deep) Generative Models
  - Diffusion Models
- Self-Supervised Learning
- Deep Reinforcement Learning

CENG796 DEEP GENERATIVE MODELS

Course Code:	5710796
METU Credit (Theoretical-Laboratory hours/week):	3(3-0)
ECTS Credit:	8.0
Department:	Computer Engineering
Language of Instruction:	English
Level of Study:	Graduate
Course Coordinator:	Assoc.Prof.Dr. RAMAZAN GÖKBERK CİNBİŞ
Offered Semester:	Fall Semesters.

Course Objectives

At the end of the course, the students will be expected to:

- Comprehend a variety of deep generative models.
- Apply deep generative models to several problems.
- Know the open issues in learning deep generative models, and have a grasp of the current research directions.

Course Content

Deep generative modeling with Autoregressive models; Energy-based models; Adversarial models; Variational models.