Deep Reinforcement Learning, VAE

Milan Straka

Apr 29, 2025

EUROPEAN UNION
European Structural and Investment Fund
Operational Programme Research, Development and Education

Charles University in Prague
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics

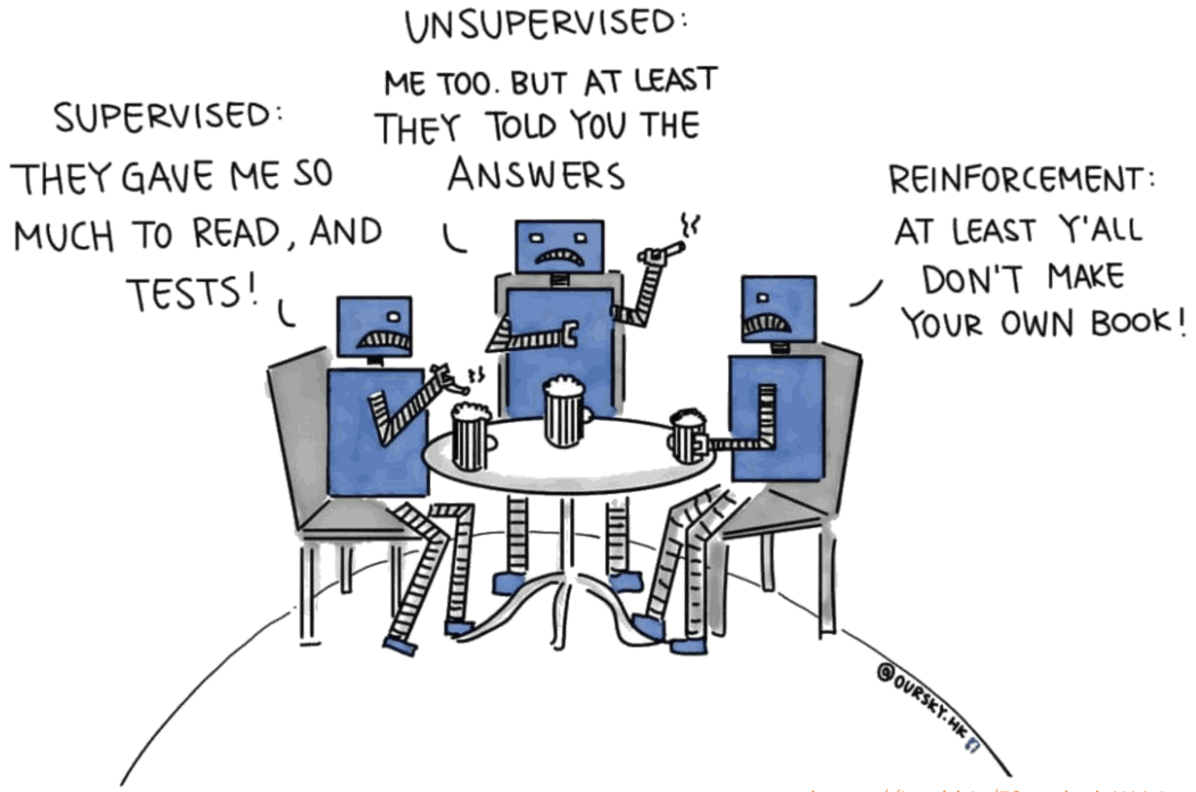
Reinforcement Learning

Develop goal-seeking agent trained using reward signal.

Reinforcement learning is a machine learning paradigm, different from *supervised* and *unsupervised learning*.

The essence of reinforcement learning is to learn from *interactions* with the environment to maximize a numeric *reward* signal.

The learner is not told which actions to take, and the actions may affect not just the immediate reward, but also all following rewards.



https://i.redd.it/50sqtdcyh1j11.jpg

Reinforcement Learning Successes

- Human-level video game playing (*DQN*) – 2013 (2015 Nature), Mnih. et al, Deepmind.
  - After 7 years of development, the *Agent57* beats humans on all 57 Atari 2600 games, achieving a mean score of 4766% compared to human players.

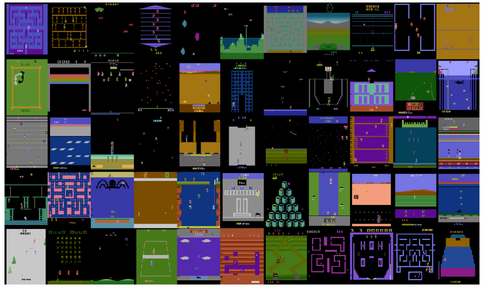


Figure 1 of "A Comparison of learning algorithms on the Arcade Learning Environment", https://arxiv.org/abs/1410.8620

- *AlphaGo* beat 9-dan professional player Lee Sedol in Go in Mar 2016.
  - After two years of development, *AlphaZero* achieved best performance in Go, chess, shogi, being trained using self-play only.

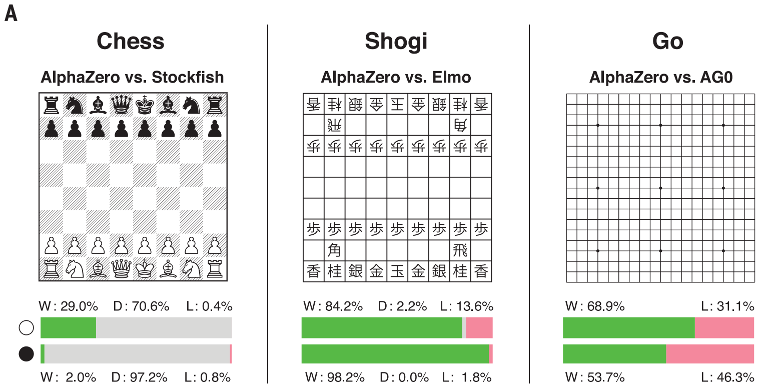
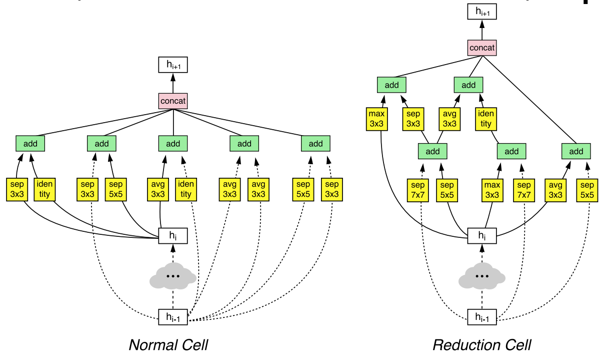


Figure 2 of "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play" by David Silver et al.

- Impressive performance in Dota2, Capture the flag FPS, StarCraft II, ...

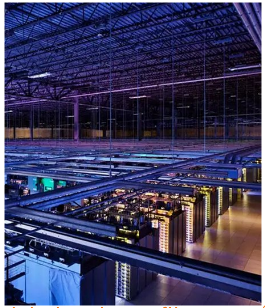
Reinforcement Learning Successes

- Neural Architecture Search – since 2017
  - automatically designing CNN image recognition networks surpassing state-of-the-art performance (*NasNet*, *EfficientNet*, *EfficientNetV2*, ...)
  - also used for other architectures, activation functions, optimizers, ...



Page 3 of "Learning Transferable Architectures for Scalable Image Recognition", https://arxiv.org/abs/1707.07012

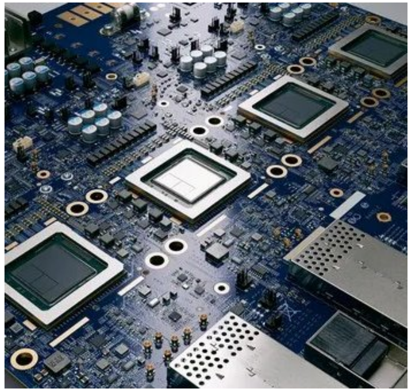
- Controlling cooling in Google datacenters directly by AI (2018)
  - reaching 30% cost reduction
- Improving efficiency of VP9 codec (2022; 4% in bandwidth with no loss in quality)



https://assets-global.website-files.com/621e749a546b7592125f38ed/622690391abb0e8c1ecf4b6a_Data%20Centers.jpg

Reinforcement Learning Successes

- Designing the layout of TPU chips (AlphaChip; since 2021, opensourced)
- Discovering faster algorithms for matrix multiplication (AlphaTensor, Oct 2022), sorting (AlphaDev, June 2023)
- Searching for solutions of mathematical problems (FunSearch, Dec 2023)
- Generally, RL can be used to Optimize nondifferentiable losses
  - Improving translation quality in 2016
  - Reinforcement learning from human feedback (*RLHF*) is used to train chatbots (ChatGPT, …)
  - Improving reasoning of LLMs (DeepSeek R1)
  - Proving math theorems (AlphaGeometry 2)



https://storage.googleapis.com/gweb-uniblog-publish-prod/images/12-11-24_Trillium-Snippet_SocialS.width-600.format-webp.webp

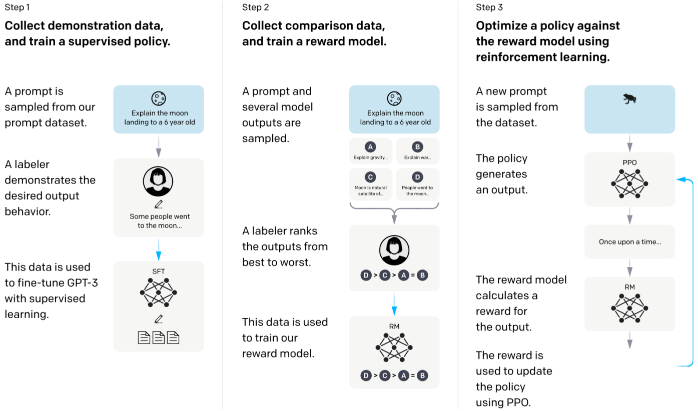
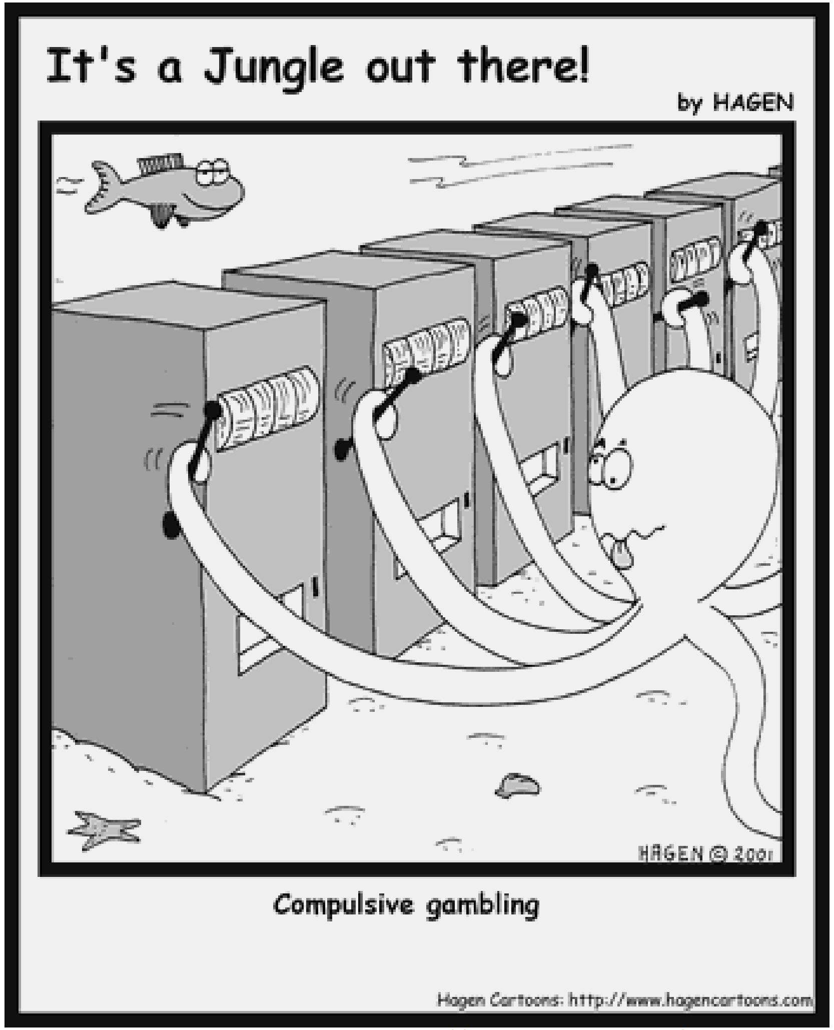


Figure 2 of "Training language models to follow instructions with human feedback", https://arxiv.org/abs/2203.02155

Multi-armed Bandits

Multi-armed Bandits

http://www.infoslotmachine.com/img/one-armed-bandit.jpg



https://hagencartoons.com/cartoon170.gif

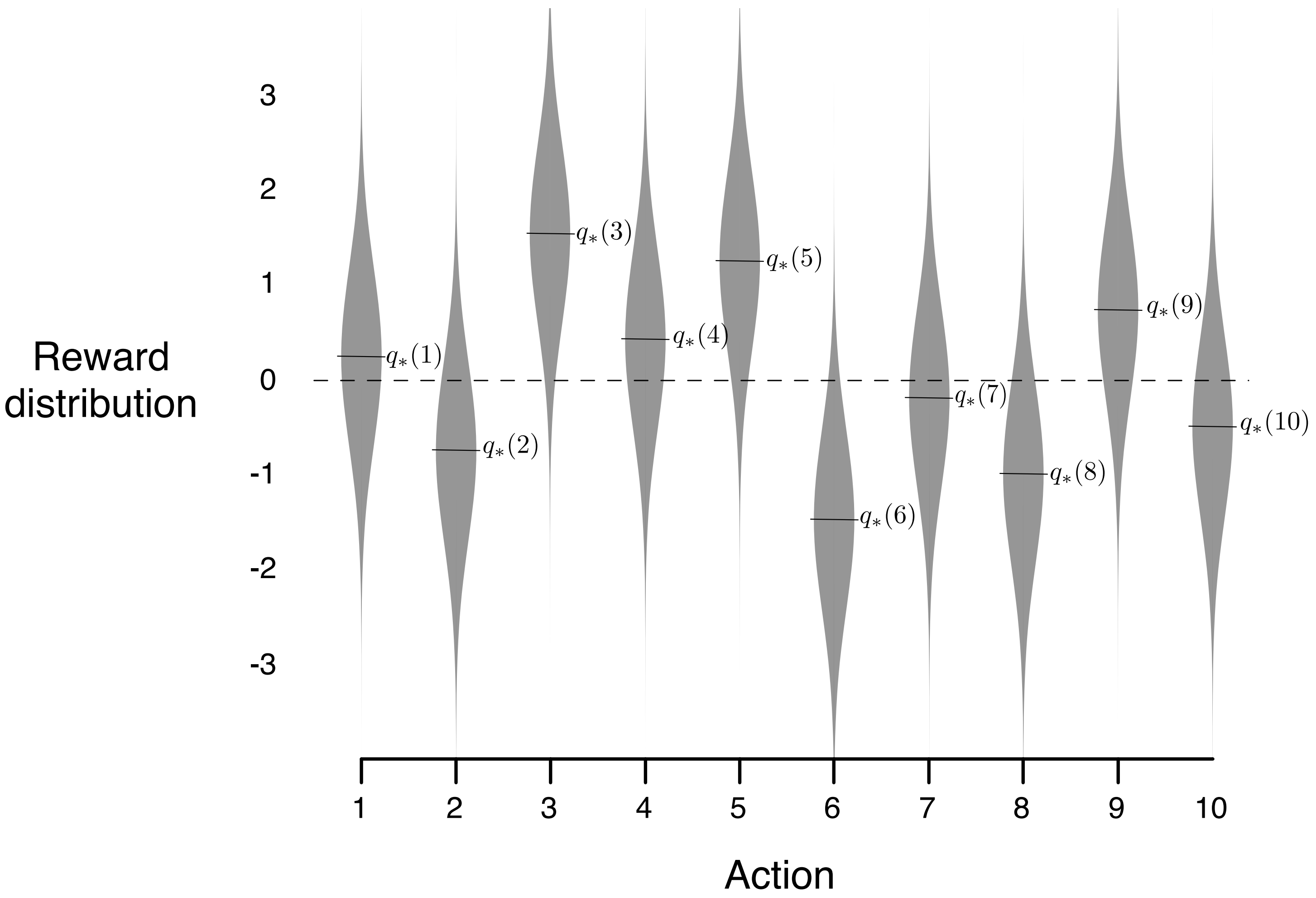


Figure 2.1 of "Reinforcement Learning: An Introduction", http://www.incompleteideas.net/book/the-book-2nd.html

We start by selecting an action A_1 (the index of the arm to use), and we obtain a reward R_1. We then repeat the process by selecting an action A_2, obtaining R_2, selecting A_3, …, with the indices denoting the time step when the actions and rewards occurred.

Let $q_*(a)$ be the real **value** of an action a:

$$q_*(a) = \mathbb{E}[R_t | A_t = a].$$

Assuming our goal is to maximize the sum of rewards $\sum_i R_i$, the optimal strategy is to repeatedly perform the action with the largest value $q_*(a)$.

Multi-armed Bandits

However, we do not know the real action values $q_*(a) = \mathbb{E}[R_t|A_t = a]$.

Therefore, we will try to estimate them, denoting $Q_t(a)$ our estimated value of action a at time t (before taking the trial t).

A natural way to estimate $Q_t(a)$ is to average the observed rewards:

$$Q_t(a) \stackrel{\text{def}}{=} \frac{\text{sum of rewards when action } a \text{ is taken}}{\text{number of times action } a \text{ was taken}}.$$

Utilizing our estimates $Q_t(a)$, we define the **greedy action** A_t as

$$A_t \stackrel{\text{def}}{=} \arg\max_a Q_t(a).$$

When our estimates are accurate enough, the optimal strategy is to repeatedly perform the greedy action.

Law of Large Numbers

Let $X_1, X_2, \ldots, X_n$ are independent and identically distributed (iid) random variables with finite mean $\mathbb{E}\big[X_i\big] = \mu < \infty$, and let

$$\bar{X}_n = \frac{1}{n}\sum_{i=1}^{N} X_i.$$

Weak Law of Large Numbers

The average $\bar{X}_N$ converges in probability to μ:

$$\bar{X}_n \xrightarrow{p} \mu \text{ when } n \rightarrow \infty, \text{ i.e., } \lim_{n\rightarrow\infty} P\big(|\bar{X}_n - \mu| < \varepsilon\big) = 1.$$

Strong Law of Large Numbers

The average $\bar{X}_N$ converges to μ almost surely:

$$\bar{X}_n \xrightarrow{a.s.} \mu \text{ when } n \rightarrow \infty, \text{ i.e., } P\Big(\lim_{n\rightarrow\infty} \bar{X}_n = \mu\Big) = 1.$$

Exploitation versus Exploration

Choosing a greedy action is **exploitation** of current estimates. We however also need to **explore** the space of actions to improve our estimates.

To make sure our estimates converge to the true values, we need to sample every action unlimited number of times.

An *ε-greedy* method follows the greedy action with probability $1-\varepsilon$, and chooses a uniformly random action with probability ε.

ε-greedy Method

Considering the 10-armed bandit problem:

- we generate 2000 random instances
  - each $q_*(a)$ is sampled from $\mathcal{N}(0, 1)$
- for every instance, we run 1000 steps of the ε-greedy method
  - we consider ε of 0, 0.01, 0.1
- we plot the averaged results over the 2000 instances

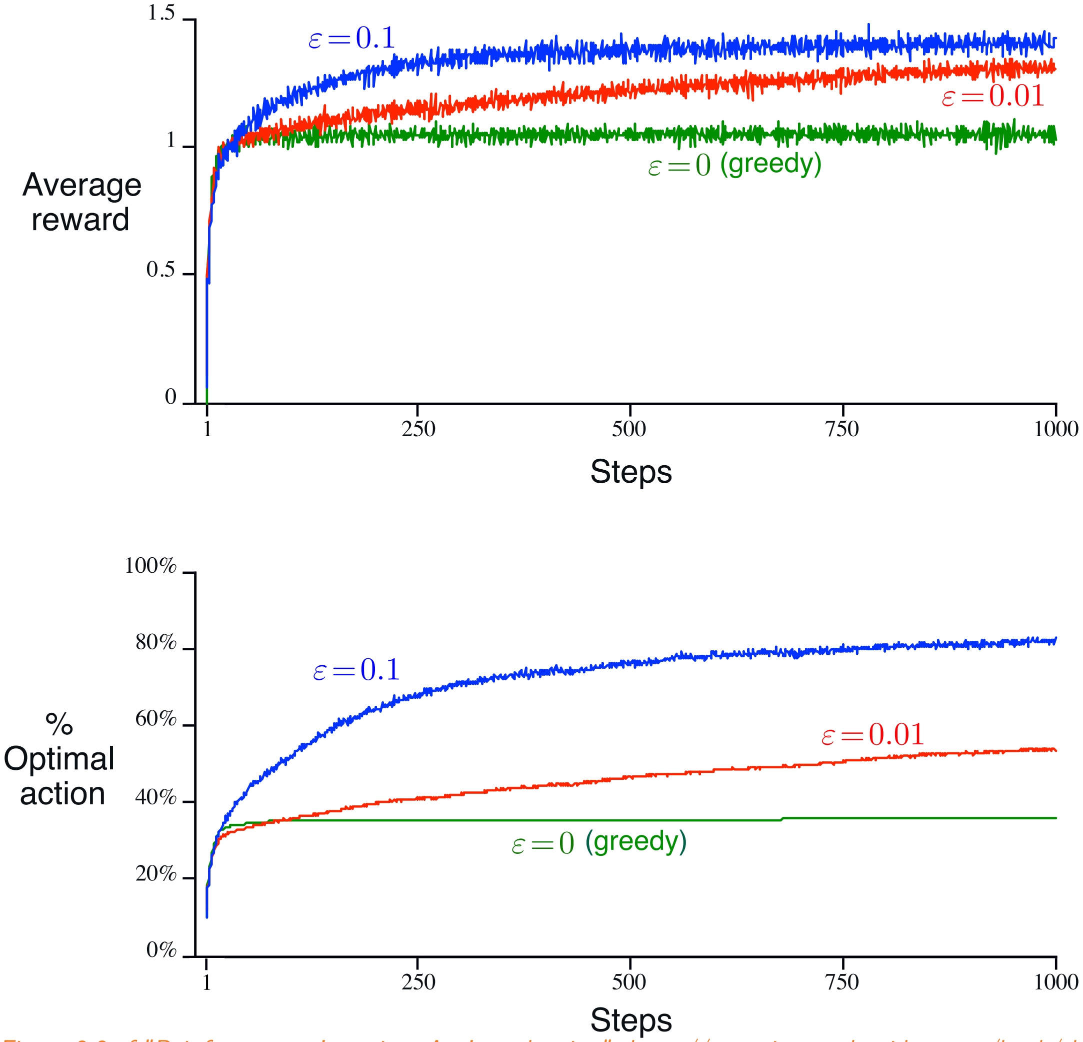
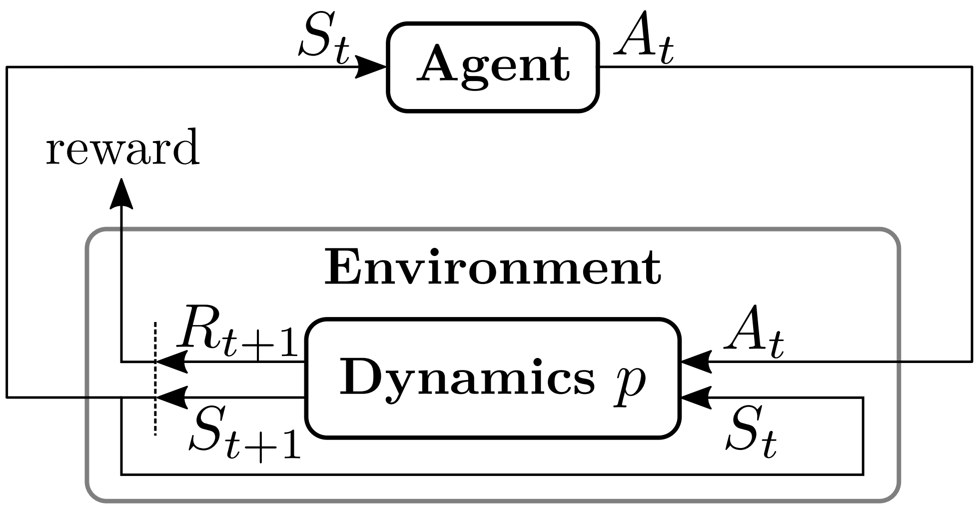


Figure 2.2 of "Reinforcement Learning: An Introduction", http://www.incompleteideas.net/book/the-book-2nd.html

Markov Decision Process

Markov Decision Process



A **Markov decision process** (MDP) is a quadruple $(\mathcal{S}, \mathcal{A}, p, \gamma)$, where:

- $\mathcal{S}$ is a set of states,
- $\mathcal{A}$ is a set of actions,
- $p(S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a)$ is a probability that action $a \in \mathcal{A}$ will lead from state $s \in \mathcal{S}$ to $s' \in \mathcal{S}$, producing a **reward** $r \in \mathbb{R}$,
- $\gamma \in [0, 1]$ is a **discount factor** (we always use $\gamma = 1$ and finite episodes in this course).

Let a **return** G_t be $G_t \stackrel{\text{def}}{=} \sum_{k=0}^{\infty} \gamma^k R_{t+1+k}$. The goal is to optimize $\mathbb{E}[G_0]$.

Episodic and Continuing Tasks

If the agent-environment interaction naturally breaks into independent subsequences, usually called **episodes**, we talk about **episodic tasks**. Each episode then ends in a special **terminal state**, followed by a reset to a starting state (either always the same, or sampled from a distribution of starting states).

In episodic tasks, it is often the case that every episode ends in at most H steps. These **finite-horizon tasks** then can use discount factor $\gamma = 1$, because the return $G \stackrel{\text{def}}{=} \sum_{t=0}^{H} \gamma^t R_{t+1}$ is well defined.

If the agent-environment interaction goes on and on without a limit, we instead talk about **continuing tasks**. In this case, the discount factor γ needs to be sharply smaller than 1.

A **policy** π computes a distribution of actions in a given state, i.e., $\pi(a|s)$ corresponds to a probability of performing an action a in state s.

We will model a policy using a neural network with parameters $\boldsymbol\theta$:

$$\pi(a|s;\boldsymbol\theta).$$

If the number of actions is finite, we consider the policy to be a categorical distribution and utilize the $\operatorname{softmax}$ output activation as in supervised classification.

(State-)Value and Action-Value Functions

To evaluate a quality of a policy, we define **value function** $v_\pi(s)$, or **state-value function**, as

$$\begin{aligned} v_\pi(s) &\stackrel{\text{def}}{=} \mathbb{E}_\pi\left[G_t \middle| S_t = s\right] = \mathbb{E}_\pi\left[\sum\nolimits_{k=0}^\infty \gamma^k R_{t+k+1} \middle| S_t = s\right] \\ &= \mathbb{E}_{A_t\sim\pi(s)} \mathbb{E}_{S_{t+1},R_{t+1}\sim p(s,A_t)} \big[R_{t+1} + \gamma \mathbb{E}_{A_{t+1}\sim\pi(S_{t+1})} \mathbb{E}_{S_{t+2},R_{t+2}\sim p(S_{t+1},A_{t+1})} \big[R_{t+2} + \ldots\big]\big] \end{aligned}$$

An **action-value function** for a policy π is defined analogously as

$$q_\pi(s, a) \stackrel{\text{def}}{=} \mathbb{E}_\pi\left[G_t \middle| S_t = s, A_t = a\right] = \mathbb{E}_\pi\left[\sum\nolimits_{k=0}^\infty \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right].$$

The value function and the state-value function can be easily expressed using one another:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_{a\sim\pi}\big[q_\pi(s, a)\big], \\ q_\pi(s, a) &= \mathbb{E}_{s',r\sim p}\big[r + \gamma v_\pi(s')\big]. \end{aligned}$$

Optimal state-value function is defined as

$$v_*(s) \stackrel{\text{def}}{=} \max_\pi v_\pi(s),$$

and **optimal action-value function** is defined analogously as

$$q_*(s, a) \stackrel{\text{def}}{=} \max_\pi q_\pi(s, a).$$

Any policy π_* with $v_{\pi_*} = v_*$ is called an **optimal policy**. Such policy can be defined as $\pi_*(s) \stackrel{\text{def}}{=} \arg\max_a q_*(s, a) = \arg\max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a]$. When multiple actions maximize $q_*(s, a)$, the optimal policy can stochastically choose any of them.

Existence

In finite-horizon tasks or if $\gamma < 1$, there always exists a unique optimal state-value function, a unique optimal action-value function, and a (not necessarily unique) optimal policy.

The REINFORCE Algorithm

Policy Gradient Methods

We train the policy

$$\pi(a|s;\boldsymbol\theta)$$

by maximizing the expected return $v_\pi(s)$.

To that account, we need to compute its **gradient** $\nabla_{\boldsymbol\theta} v_\pi(s)$.

Policy Gradient Theorem

Assume that $\mathcal{S}$ and $\mathcal{A}$ are finite, $\gamma = 1$, and that maximum episode length H is also finite.

Let $\pi(a|s;\boldsymbol\theta)$ be a parametrized policy. We denote the initial state distribution as $h(s)$ and the on-policy distribution under π as $\mu(s)$. Let also $J(\boldsymbol\theta) \stackrel{\text{def}}{=} \mathbb{E}_{s\sim h} v_\pi(s)$.

Then

$$\nabla_{\boldsymbol\theta} v_\pi(s) \propto \sum_{s'\in\mathcal{S}} P(s \rightarrow \ldots \rightarrow s'|\pi) \sum_{a\in\mathcal{A}} q_\pi(s', a) \nabla_{\boldsymbol\theta} \pi(a|s';\boldsymbol\theta)$$

and

$$\nabla_{\boldsymbol\theta} J(\boldsymbol\theta) \propto \sum_{s\in\mathcal{S}} \mu(s) \sum_{a\in\mathcal{A}} q_\pi(s, a) \nabla_{\boldsymbol\theta} \pi(a|s;\boldsymbol\theta),$$

where $P(s \rightarrow \ldots \rightarrow s'|\pi)$ is the probability of getting to state s' when starting from state s, after any number of 0, 1, … steps.

Proof of Policy Gradient Theorem

$$\begin{aligned}\nabla v_\pi(s) &= \nabla \Big[\sum\nolimits_a \pi(a|s;\boldsymbol\theta) q_\pi(s, a)\Big] \\ &= \sum\nolimits_a \Big[q_\pi(s, a) \nabla \pi(a|s;\boldsymbol\theta) + \pi(a|s;\boldsymbol\theta) \nabla q_\pi(s, a)\Big] \\ &= \sum\nolimits_a \Big[q_\pi(s, a) \nabla \pi(a|s;\boldsymbol\theta) + \pi(a|s;\boldsymbol\theta) \nabla \big(\sum\nolimits_{s',r} p(s', r|s, a)(r + v_\pi(s'))\big)\Big] \\ &= \sum\nolimits_a \Big[q_\pi(s, a) \nabla \pi(a|s;\boldsymbol\theta) + \pi(a|s;\boldsymbol\theta) \big(\sum\nolimits_{s'} p(s'|s, a) \nabla v_\pi(s')\big)\Big]\end{aligned}$$

We now expand $v_\pi(s')$.

$$\begin{aligned} &= \sum\nolimits_a \Big[q_\pi(s, a) \nabla \pi(a|s;\boldsymbol\theta) + \pi(a|s;\boldsymbol\theta) \Big(\sum\nolimits_{s'} p(s'|s, a)\Big(\\ &\qquad \sum\nolimits_{a'} \Big[q_\pi(s', a') \nabla \pi(a'|s';\boldsymbol\theta) + \pi(a'|s';\boldsymbol\theta) \big(\sum\nolimits_{s''} p(s''|s', a') \nabla v_\pi(s'')\big)\Big]\Big)\Big)\Big]\end{aligned}$$

Continuing to expand all $v_\pi(s'')$, we obtain the following:

$$\nabla v_\pi(s) = \sum\nolimits_{s' \in \mathcal{S}} \sum\nolimits_{k=0}^{H} P(s \rightarrow s' \text{ in } k \text{ steps } |\pi) \sum\nolimits_{a \in \mathcal{A}} q_\pi(s', a) \nabla_{\boldsymbol\theta} \pi(a|s';\boldsymbol\theta).$$

Proof of Policy Gradient Theorem

To finish the proof of the first part, it is enough to realize that

$$\sum\nolimits_{k=0}^{H} P(s \rightarrow s' \text{ in } k \text{ steps } | \pi) \propto P(s \rightarrow \ldots \rightarrow s' | \pi).$$

For the second part, we know that

$$\nabla_{\boldsymbol\theta} J(\boldsymbol\theta) = \mathbb{E}_{s \sim h} \nabla_{\boldsymbol\theta} v_\pi(s) \propto \mathbb{E}_{s \sim h} \sum_{s' \in \mathcal{S}} P(s \rightarrow \ldots \rightarrow s' | \pi) \sum_{a \in \mathcal{A}} q_\pi(s', a) \nabla_{\boldsymbol\theta} \pi(a | s'; \boldsymbol\theta),$$

therefore using the fact that $\mu(s') = \mathbb{E}_{s \sim h} P(s \rightarrow \ldots \rightarrow s' | \pi)$ we get

$$\nabla_{\boldsymbol\theta} J(\boldsymbol\theta) \propto \sum_{s \in \mathcal{S}} \mu(s) \sum_{a \in \mathcal{A}} q_\pi(s, a) \nabla_{\boldsymbol\theta} \pi(a | s; \boldsymbol\theta).$$

Finally, note that the theorem can be proven with infinite $\mathcal{S}$ and $\mathcal{A}$; and also for infinite episodes when discount factor $\gamma < 1$.

REINFORCE Algorithm

The REINFORCE algorithm (Williams, 1992) directly uses the policy gradient theorem, minimizing $-J(\boldsymbol\theta) \stackrel{\text{def}}{=} -\mathbb{E}_{s\sim h} v_\pi(s)$. The loss gradient is then

$$\nabla_{\boldsymbol\theta} - J(\boldsymbol\theta) \propto -\sum_{s\in\mathcal{S}} \mu(s) \sum_{a\in\mathcal{A}} q_\pi(s,a) \nabla_{\boldsymbol\theta} \pi(a|s;\boldsymbol\theta) = -\mathbb{E}_{s\sim\mu} \sum_{a\in\mathcal{A}} q_\pi(s,a) \nabla_{\boldsymbol\theta} \pi(a|s;\boldsymbol\theta).$$

However, the sum over all actions is problematic. Instead, we rewrite it to an expectation which we can estimate by sampling:

$$\nabla_{\boldsymbol\theta} - J(\boldsymbol\theta) \propto \mathbb{E}_{s\sim\mu} \mathbb{E}_{a\sim\pi} q_\pi(s,a) \nabla_{\boldsymbol\theta} - \log \pi(a|s;\boldsymbol\theta),$$

where we used the fact that

$$\nabla_{\boldsymbol\theta} \log \pi(a|s;\boldsymbol\theta) = \frac{1}{\pi(a|s;\boldsymbol\theta)} \nabla_{\boldsymbol\theta} \pi(a|s;\boldsymbol\theta).$$

REINFORCE therefore minimizes the loss $-J(\boldsymbol\theta)$ with gradient

$$\mathbb{E}_{s\sim\mu}\mathbb{E}_{a\sim\pi} q_\pi(s,a)\nabla_{\boldsymbol\theta} - \log\pi(a|s;\boldsymbol\theta),$$

where we estimate the $q_\pi(s,a)$ by a single sample.

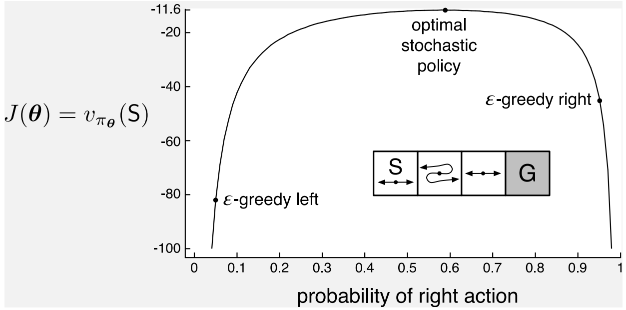
Note that the loss is just a weighted variant of negative log-likelihood (NLL), where the sampled actions play a role of gold labels and are weighted according to their return.

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol\theta)$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol\theta \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol\theta)$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\boldsymbol\theta \leftarrow \boldsymbol\theta + \alpha G \nabla \ln\pi(A_t|S_t,\boldsymbol\theta)$

Modified from Algorithm 13.3 of "Reinforcement Learning: An Introduction", http://www.incompleteideas.net/book/the-book-2nd.html by removing γ^t from the update of θ



Example 13.1 of "Reinforcement Learning: An Introduction, Second Edition".

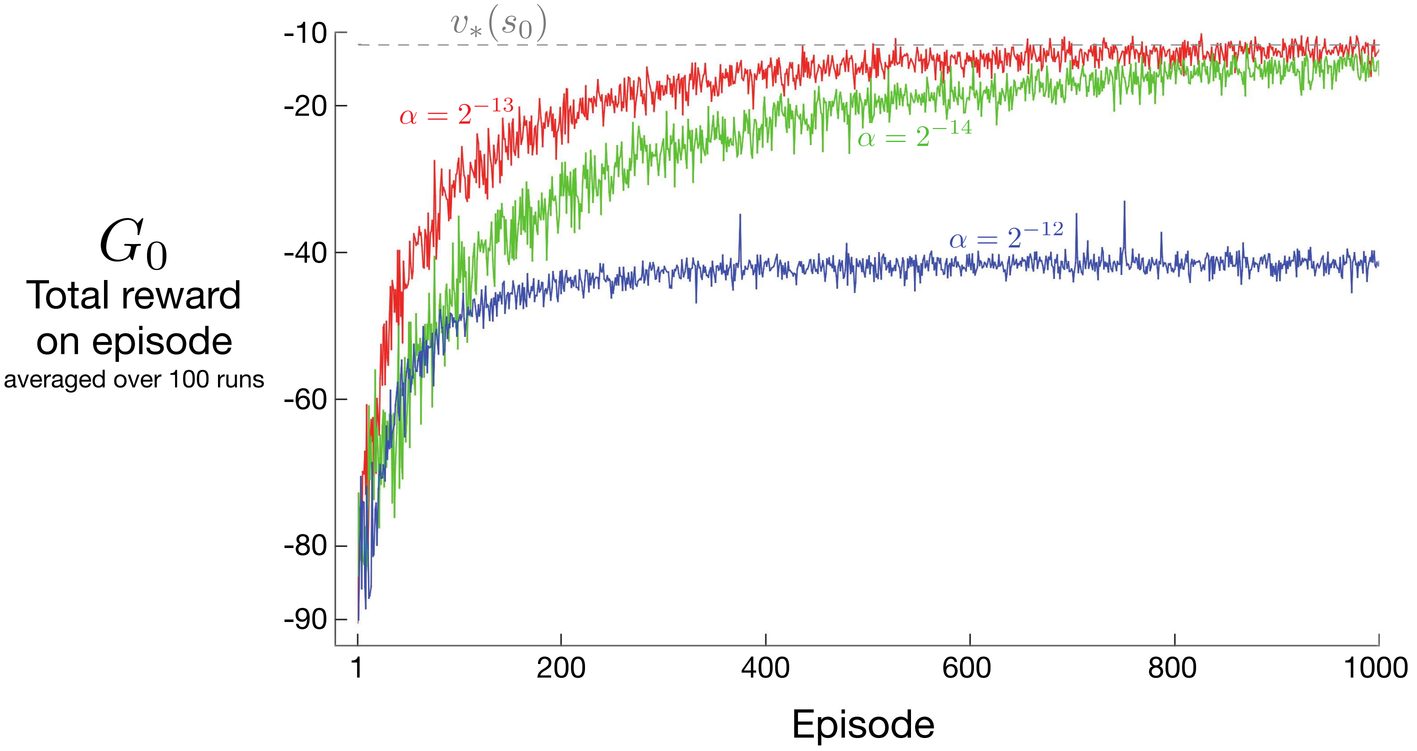


Figure 13.1 of "Reinforcement Learning: An Introduction, Second Edition".

REINFORCE with Baseline

REINFORCE with Baseline

The returns can be arbitrary: better-than-average and worse-than-average returns cannot be recognized from the absolute value of the return.

Hopefully, we can generalize the policy gradient theorem using a baseline $b(s)$ to

$$\nabla_{\boldsymbol\theta} J(\boldsymbol\theta) \propto \sum_{s\in\mathcal{S}} \mu(s) \sum_{a\in\mathcal{A}} \big(q_\pi(s,a) - \boldsymbol{b(s)}\big) \nabla_{\boldsymbol\theta} \pi(a|s;\boldsymbol\theta).$$

The baseline $b(s)$ can be a function or even a random variable, as long as it does not depend on a, because

$$\sum_a b(s) \nabla_{\boldsymbol\theta} \pi(a|s;\boldsymbol\theta) = b(s) \sum_a \nabla_{\boldsymbol\theta} \pi(a|s;\boldsymbol\theta) = b(s) \nabla_{\boldsymbol\theta} \sum_a \pi(a|s;\boldsymbol\theta) = b(s) \nabla_{\boldsymbol\theta} 1 = 0.$$

REINFORCE with Baseline

A good choice for $b(s)$ is $v_\pi(s)$, which can be shown to minimize the variance of the gradient estimator. Such baseline reminds centering of the returns, given that

$$v_\pi(s) = \mathbb{E}_{a\sim\pi} q_\pi(s, a).$$

Then, better-than-average returns are positive and worse-than-average returns are negative.

Of course, we need a way to estimate the $v_\pi(s)$ baseline. The usual approach is to approximate it by another neural network model. That a model model is trained using mean square error of the predicted and observed returns.

REINFORCE with Baseline

In REINFORCE with baseline, we train:

1. the *policy network* using the REINFORCE algorithm, and
2. the *value network* by minimizing the mean squared error.

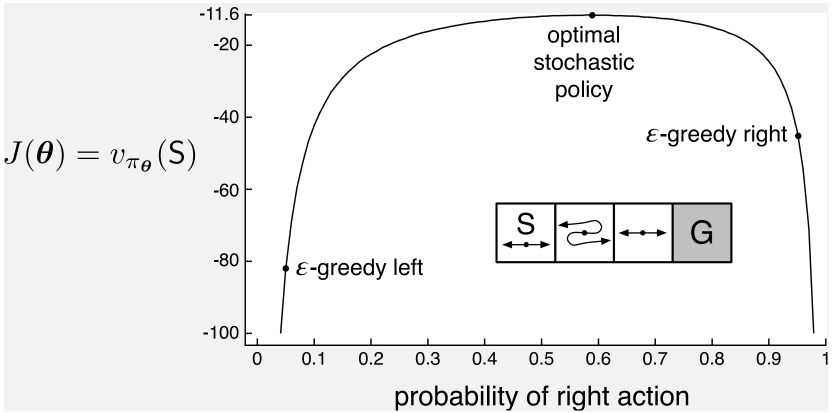
REINFORCE with Baseline (episodic), for estimating $\pi_{\boldsymbol\theta} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol\theta)$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol\theta} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol\theta \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol\theta)$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\delta \leftarrow G - \hat{v}(S_t,\mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t,\mathbf{w})$
    $\boldsymbol\theta \leftarrow \boldsymbol\theta + \alpha^{\boldsymbol\theta} \delta \nabla \ln \pi(A_t|S_t, \boldsymbol\theta)$

Modified from Algorithm 13.4 of "Reinforcement Learning: An Introduction", http://www.incompleteideas.net/book/the-book-2nd.html by removing γ^t from the update of θ

REINFORCE with Baseline Example Performance



Example 13.1 of "Reinforcement Learning: An Introduction, Second Edition".

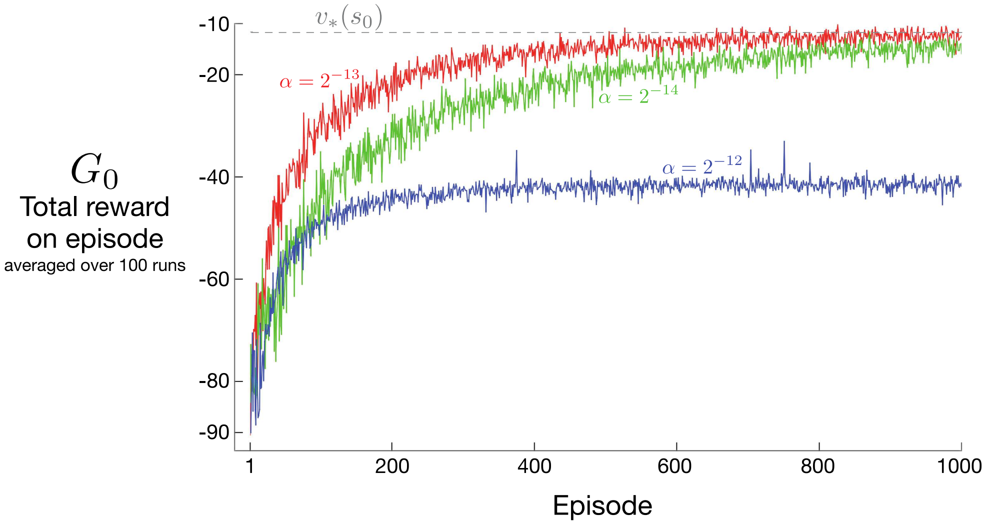


Figure 13.1 of "Reinforcement Learning: An Introduction, Second Edition".

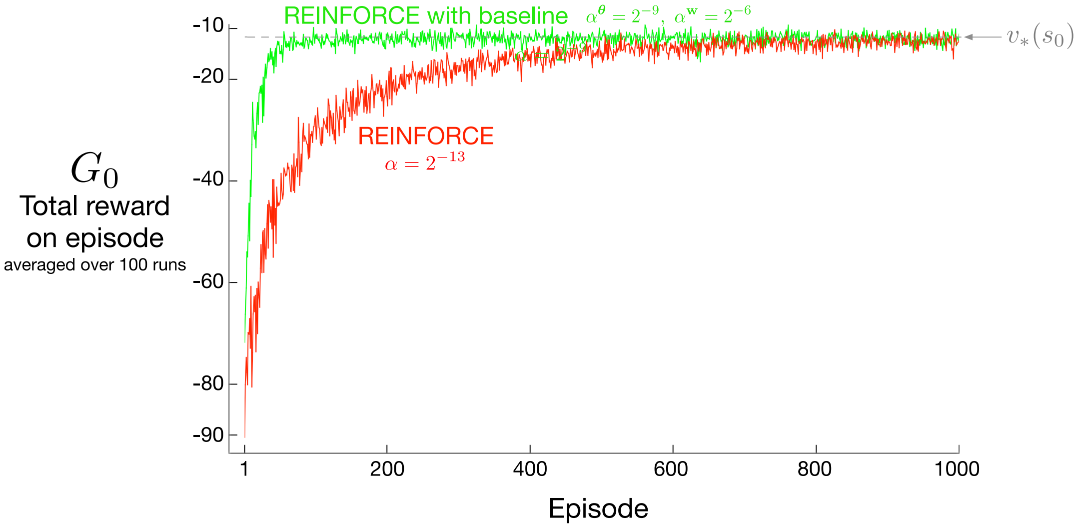


Figure 13.2 of "Reinforcement Learning: An Introduction", http://www.incompleteideas.net/book/the-book-2nd.html

Neural Architecture Search

Neural Architecture Search: NASNet, 2017

- We can design neural network architectures using reinforcement learning.
- The designed network is encoded as a sequence of elements, and is generated using an **RNN controller**, which is trained using the REINFORCE with baseline algorithm.

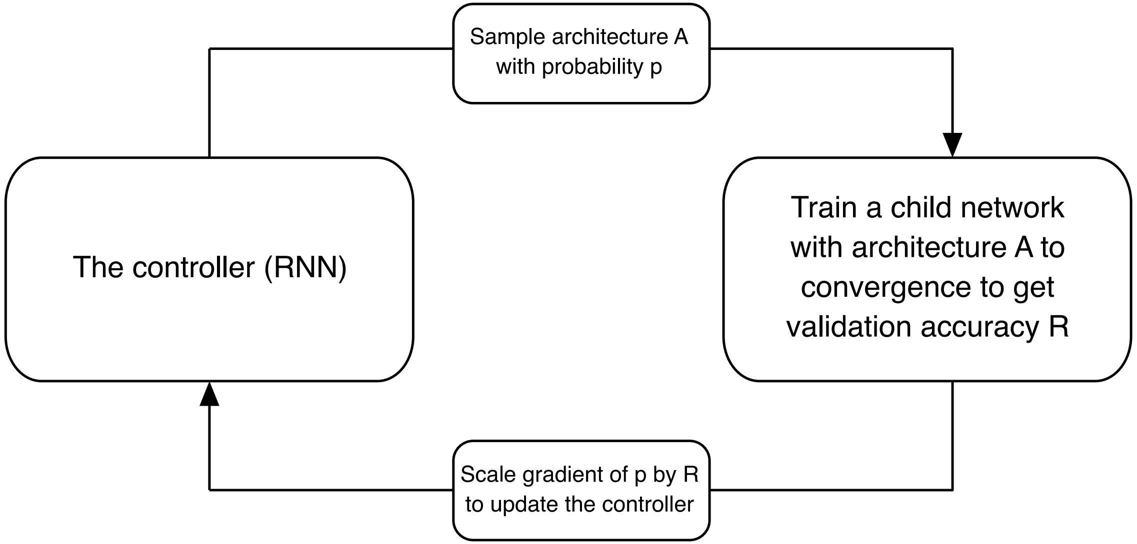


Figure 1 of "Learning Transferable Architectures for Scalable Image Recognition", https://arxiv.org/abs/1707.07012

- For every generated sequence, the corresponding network is trained on CIFAR-10 and the development accuracy is used as a return.

The overall architecture of the designed network is fixed and only the Normal Cells and Reduction Cells are generated by the controller.

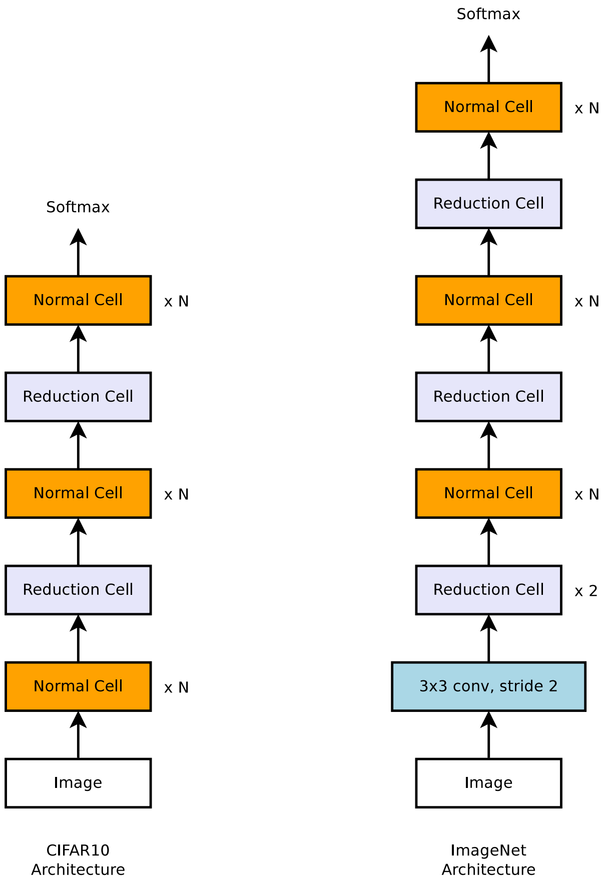


Figure 2 of "Learning Transferable Architectures for Scalable Image Recognition", https://arxiv.org/abs/1707.07012

Neural Architecture Search: NASNet, 2017

- Each cell is composed of B blocks ($B = 5$ is used in NASNet).
- Each block is designed by a RNN controller generating 5 parameters.

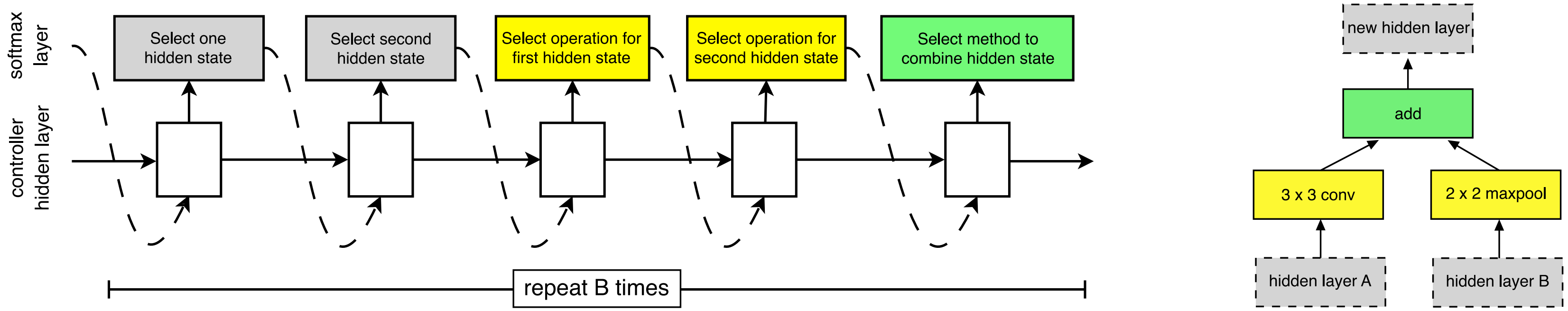


Figure 3. Controller model architecture for recursively constructing one block of a convolutional cell. Each block requires selecting 5 discrete parameters, each of which corresponds to the output of a softmax layer. Example constructed block shown on right. A convolutional cell contains B blocks, hence the controller contains $5B$ softmax layers for predicting the architecture of a convolutional cell. In our experiments, the number of blocks B is 5.

Figure 3 of "Learning Transferable Architectures for Scalable Image Recognition", https://arxiv.org/abs/1707.07012

Step 1. Select a hidden state from h_i, h_{i-1} or from the set of hidden states created in previous blocks.

Step 2. Select a second hidden state from the same options as in Step 1.

Step 3. Select an operation to apply to the hidden state selected in Step 1.

Step 4. Select an operation to apply to the hidden state selected in Step 2.

Step 5. Select a method to combine the outputs of Step 3 and 4 to create a new hidden state.

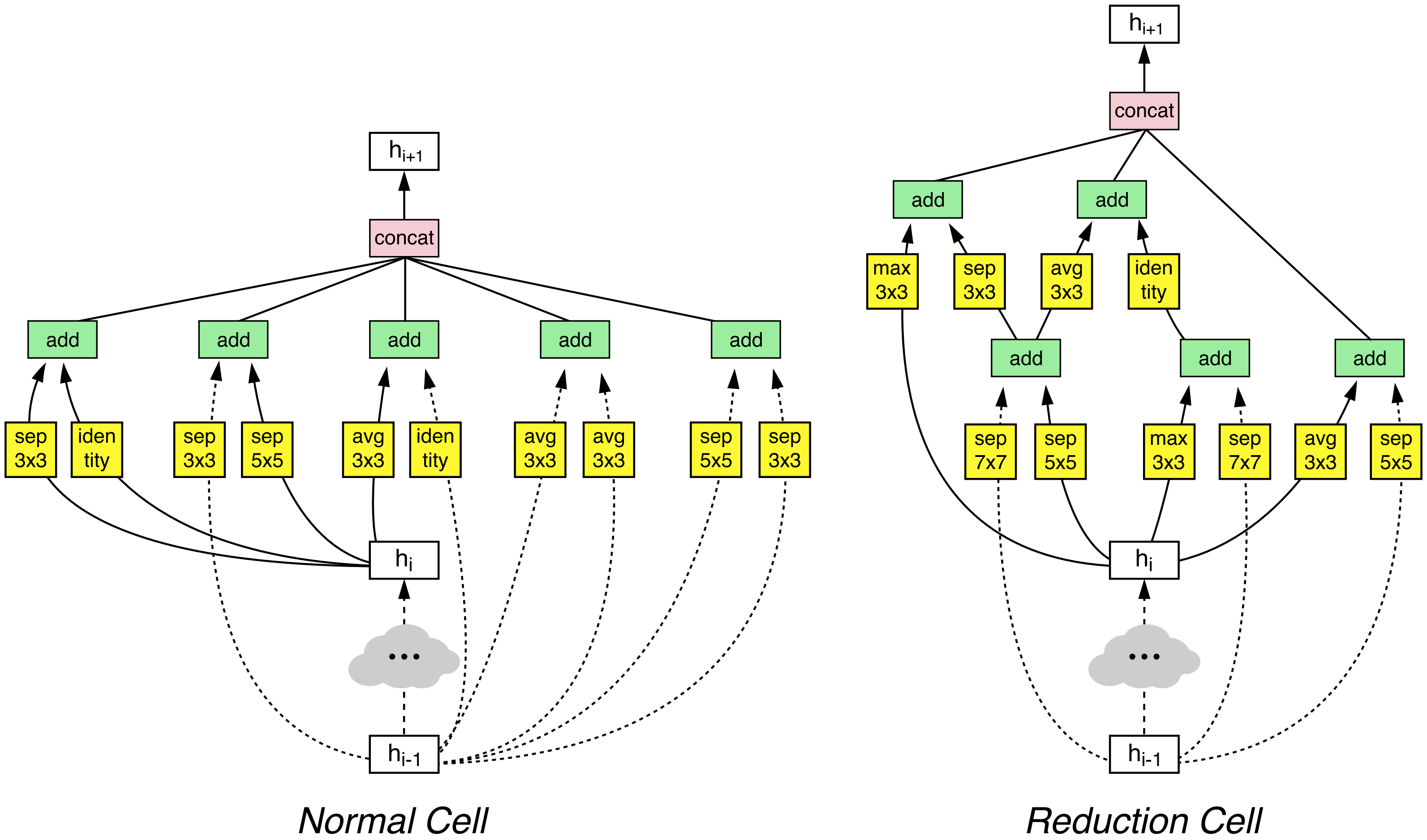
Page 3 of "Learning Transferable Architectures for Scalable Image Recognition", https://arxiv.org/abs/1707.07012

- identity
- 1x7 then 7x1 convolution
- 3x3 average pooling
- 5x5 max pooling
- 1x1 convolution
- 3x3 depthwise-separable conv
- 7x7 depthwise-separable conv
- 1x3 then 3x1 convolution
- 3x3 dilated convolution
- 3x3 max pooling
- 7x7 max pooling
- 3x3 convolution
- 5x5 depthwise-seperable conv

Figure 2 of "Learning Transferable Architectures for Scalable Image Recognition", https://arxiv.org/abs/1707.07012

Neural Architecture Search: NASNet, 2017

The final Normal Cell and Reduction Cell chosen from 20k architectures (500GPUs, 4days).



Page 3 of "Learning Transferable Architectures for Scalable Image Recognition", https://arxiv.org/abs/1707.07012

EfficientNet Search

EfficientNet changes the search in three ways.

- Computational requirements are part of the return. Notably, the goal is to find an architecture m maximizing

$$\text{DevelopmentAccuracy}(m) \cdot \left(\frac{\text{TargetFLOPS=400M}}{\text{FLOPS}(m)}\right)^{0.07},$$

  where the constant 0.07 balances the accuracy and FLOPS (*the constant comes from an empirical observation that doubling the FLOPS brings about 5% relative accuracy gain, and* $1.05 = 2^\beta$ gives $\beta \approx 0.0704$).

- It uses a different search space allowing to control kernel sizes and channels in different parts of the architecture (compared to using the same cell everywhere as in NASNet).
- Training directly on ImageNet, but only for 5 epochs.

In total, 8k model architectures are sampled, and PPO algorithm is used instead of the REINFORCE with baseline.

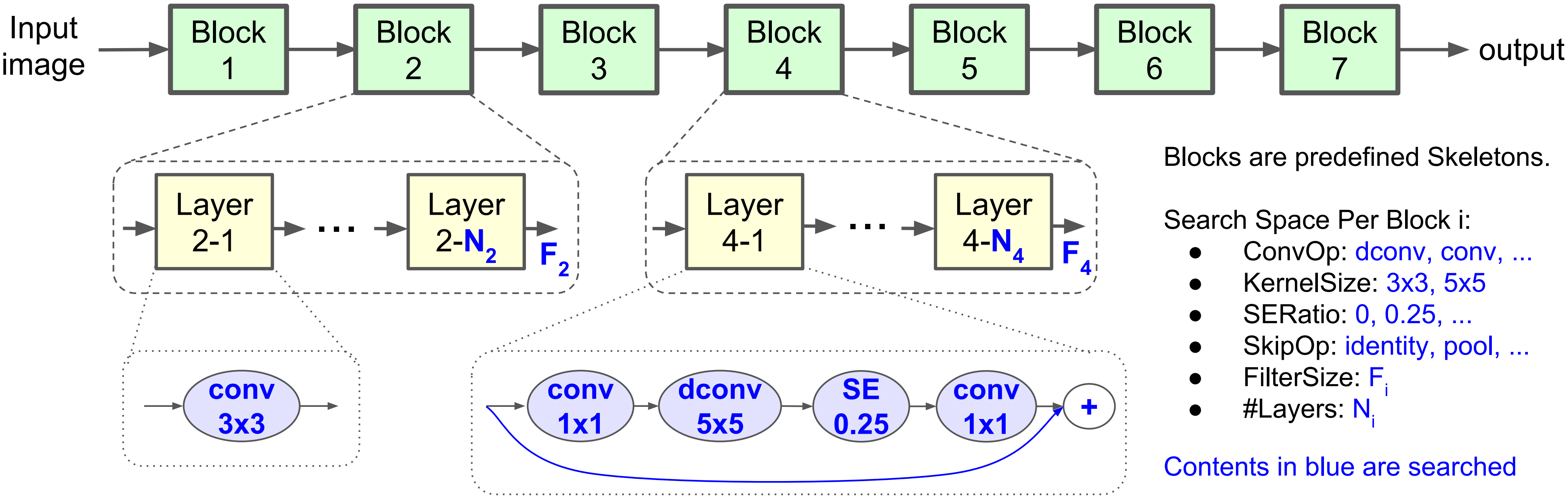


Figure 4 of "MnasNet: Platform-Aware Neural Architecture Search for Mobile", https://arxiv.org/abs/1807.11626

The overall architecture consists of 7 blocks, each described by 6 parameters – 42 parameters in total, compared to 50 parameters of the NASNet search space.

- Convolutional ops $ConvOp$: regular conv (conv), depthwise conv (dconv), and mobile inverted bottleneck conv [29].
- Convolutional kernel size $KernelSize$: 3x3, 5x5.
- Squeeze-and-excitation [13] ratio $SERatio$: 0, 0.25.
- Skip ops $SkipOp$: pooling, identity residual, or no skip.
- Output filter size F_i.
- Number of layers per block N_i.

Page 4 of "MnasNet: Platform-Aware Neural Architecture Search for Mobile", https://arxiv.org/abs/1807.11626

EfficientNet-B0 Baseline Network

Stage i	Operator $\hat{\mathcal{F}}_i$	Resolution $\hat{H}_i \times \hat{W}_i$	#Channels $\hat{C}_i$	#Layers $\hat{L}_i$
1	Conv3x3	224×224	32	1
2	MBConv1, k3x3	112×112	16	1
3	MBConv6, k3x3	112×112	24	2
4	MBConv6, k5x5	56×56	40	2
5	MBConv6, k3x3	28×28	80	3
6	MBConv6, k5x5	14×14	112	3
7	MBConv6, k5x5	14×14	192	4
8	MBConv6, k3x3	7×7	320	1
9	Conv1x1 & Pooling & FC	7×7	1280	1

Table 1 of "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks", https://arxiv.org/abs/1905.11946

What Next

If you find deep reinforcement learning interesting, I have a whole course dedicated to it: **NPFL139 – Deep Reinforcement Learning**.

- It covers a range of reinforcement learning algorithms, from the basic ones to more advanced algorithms utilizing deep neural networks.
- Summer semester, 3/2 C+Ex, 8 e-credits, similar structure as Deep learning.
- An elective (povinně volitelný) course in the programs:
  - Artificial Intelligence,
  - Language Technologies and Computational Linguistics.

Generative Models

Generative Models

https://images.squarespace-cdn.com/content/v1/6213c340453c3f502425776e/0715034d-4044-4c55-9131-e4bfd6dd20ca/2_4x.png

https://i.kym-cdn.com/photos/images/original/002/470/247/37b.jpg

https://i.redd.it/now-that-hands-are-better-heres-a-meme-update-v0-73j3ez3wi0oa1.png?s=bf6ea761fea5d1d44ccf34d5961b23aeea1b19bc

Generative models are given a set of realizations of a random variable $\mathbf{x}$ and their goal is to estimate $P(\boldsymbol{x})$.

Usually the goal is to be able to sample from $P(\mathbf{x})$, but sometimes an explicit calculation of $P(\boldsymbol{x})$ is also possible.

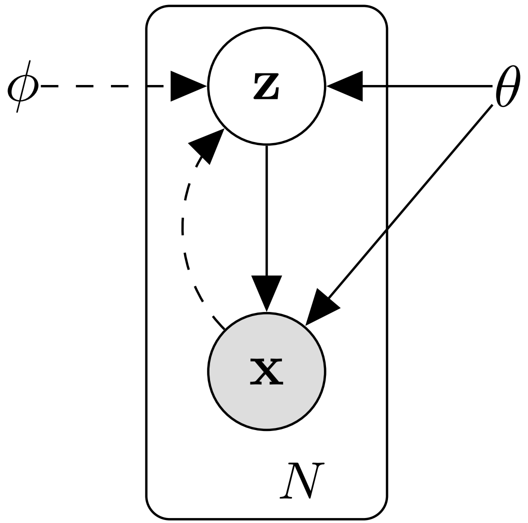


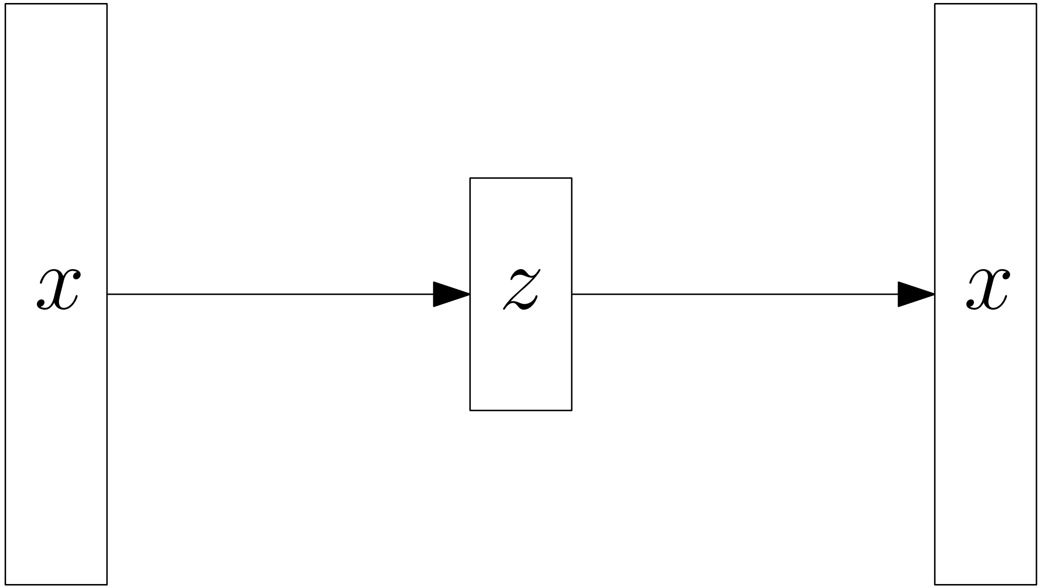
Figure 1 of "Auto-Encoding Variational Bayes", https://arxiv.org/abs/1312.6114

One possible approach to estimate $P(\boldsymbol{x})$ is to assume that the random variable $\mathbf{x}$ depends on a **latent variable** $\mathbf{z}$:

$$P(\boldsymbol{x}) = \sum_{\boldsymbol{z}} P(\boldsymbol{z}) P(\boldsymbol{x}|\boldsymbol{z}) = \mathbb{E}_{\boldsymbol{z} \sim P(\mathbf{z})} P(\boldsymbol{x}|\boldsymbol{z}).$$

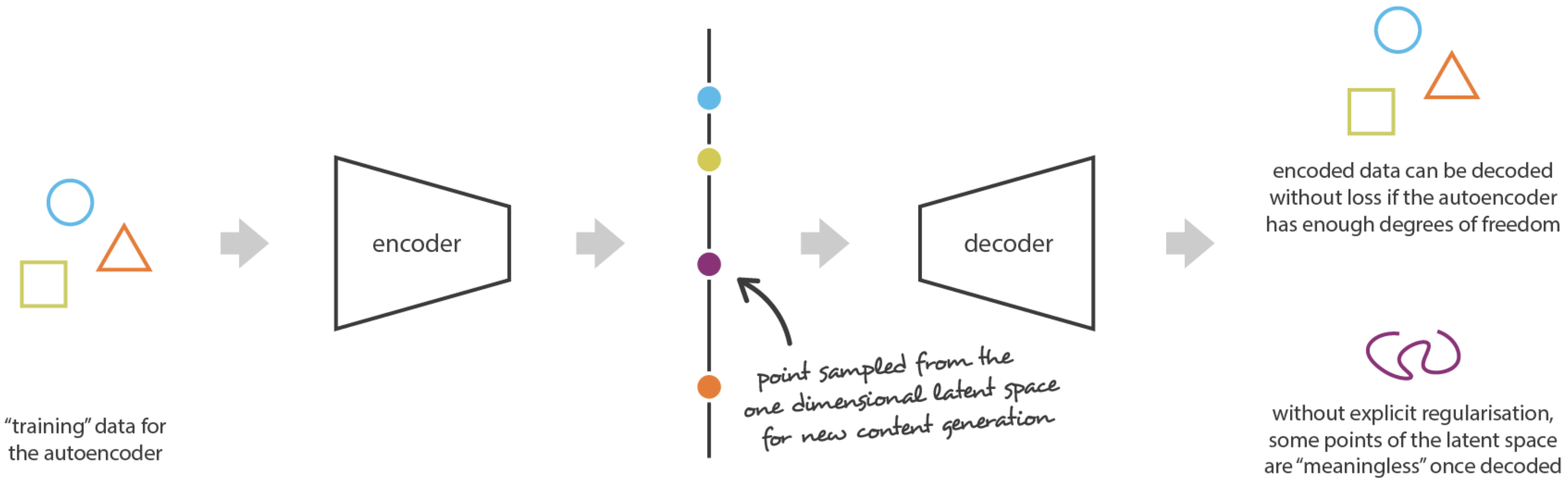
We use neural networks to estimate the conditional probability $P_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$.

AutoEncoders



- Autoencoders are useful for unsupervised feature extraction, especially when performing input compression (i.e., when the dimensionality of the latent space $\boldsymbol{z}$ is smaller than the dimensionality of the input).
- When $\boldsymbol{x} + \boldsymbol{\varepsilon}$ is used as input, autoencoders can perform denoising.
- However, the latent space $\boldsymbol{z}$ does not need to be fully covered, so a randomly chosen $\boldsymbol{z}$ does not need to produce a valid $\boldsymbol{x}$.

AutoEncoders



*https://miro.medium.com/max/3608/1*iSfaVxcGi_ELkKgAG0YRlQ@2x.png*

Variational AutoEncoders

Variational AutoEncoders

We assume $P(\mathbf{z})$ is fixed and independent on $\mathbf{x}$.

We approximate $P(\boldsymbol{x}|\boldsymbol{z})$ using a neural network $P_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$, the **decoder**.

However, in order to train an autoencoder, we need to know the "inverse" $P_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})$, which cannot be usually computed directly.

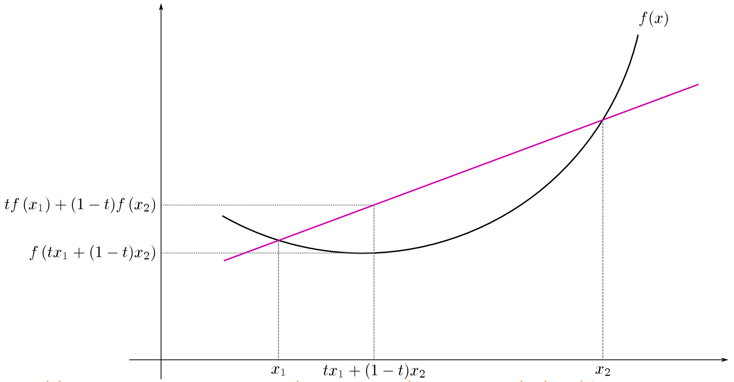
Therefore, we approximate $P_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})$ by another trainable neural network $Q_{\boldsymbol{\varphi}}(\boldsymbol{z}|\boldsymbol{x})$, the **encoder**.

Jensen's Inequality

To derive a loss for training variational autoencoders, we first formulate the Jensen's inequality.

Recall that convex functions by definition fulfil that for $\boldsymbol{u}, \boldsymbol{v}$ and real $0 \leq t \leq 1$,

$$f(t\boldsymbol{u} + (1-t)\boldsymbol{v}) \leq tf(\boldsymbol{u}) + (1-t)f(\boldsymbol{v}).$$



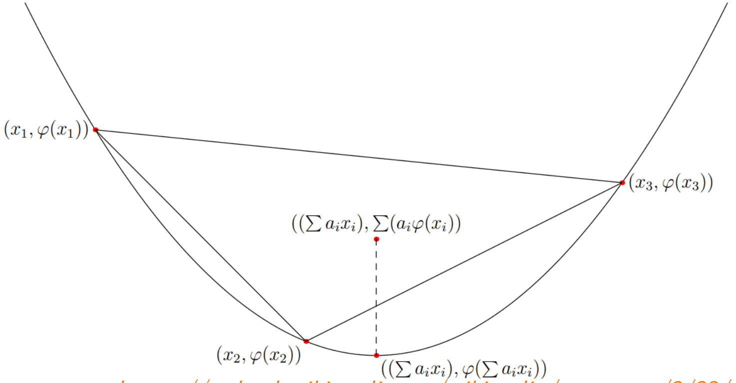
https://upload.wikimedia.org/wikipedia/commons/c/c7/ConvexFunction.svg

The **Jensen's inequality** generalizes the above property to any *convex* combination of points: if we have $\boldsymbol{u}_i \in \mathbb{R}^D$ and weights $w_i \in \mathbb{R}^+$ such that $\sum_i w_i = 1$, it holds that

$$f\Big(\sum_i w_i \boldsymbol{u}_i\Big) \leq \sum_i w_i f\big(\boldsymbol{u}_i\big).$$

The Jensen's inequality can be formulated also for probability distributions (whose expectation can be considered an infinite convex combination):

$$f\big(\mathbb{E}[\mathbf{u}]\big) \leq \mathbb{E}_{\mathbf{u}}\big[f(\mathbf{u})\big].$$



https://upload.wikimedia.org/wikipedia/commons/3/33/Jensen%27s_Inequality_Proof_Without_Words.png

VAE – Loss Function Derivation

Our goal will be to maximize the log-likelihood as usual, but we need to express it using the latent variable $\boldsymbol{z}$:

$$\log P_{\boldsymbol\theta}(\boldsymbol{x}) = \log \mathbb{E}_{P(\boldsymbol{z})}\big[P_{\boldsymbol\theta}(\boldsymbol{x}|\boldsymbol{z})\big].$$

However, approximating the expectation using a single sample has monstrous variance, because for most $\boldsymbol{z}$, $P_{\boldsymbol\theta}(\boldsymbol{x}|\boldsymbol{z})$ will be nearly zero.

We therefore turn to our *encoder*, which is able for a given $\boldsymbol{x}$ to generate "its" $\boldsymbol{z}$:

$$\begin{aligned}\log P_{\boldsymbol\theta}(\boldsymbol{x}) &= \log \mathbb{E}_{P(\boldsymbol{z})}\big[P_{\boldsymbol\theta}(\boldsymbol{x}|\boldsymbol{z})\big] \\ &= \log \mathbb{E}_{Q_{\boldsymbol\varphi}(\boldsymbol{z}|\boldsymbol{x})}\Big[P_{\boldsymbol\theta}(\boldsymbol{x}|\boldsymbol{z}) \cdot \frac{P(\boldsymbol{z})}{Q_{\boldsymbol\varphi}(\boldsymbol{z}|\boldsymbol{x})}\Big]\end{aligned}$$

now we use the Jensen's inequality

$$\geq \mathbb{E}_{Q_{\boldsymbol\varphi}(\boldsymbol{z}|\boldsymbol{x})}\Big[\log P_{\boldsymbol\theta}(\boldsymbol{x}|\boldsymbol{z}) + \log \frac{P(\boldsymbol{z})}{Q_{\boldsymbol\varphi}(\boldsymbol{z}|\boldsymbol{x})}\Big]$$

VAE – Variational (or Evidence) Lower Bound

The resulting **variational lower bound** or **evidence lower bound** (ELBO), denoted $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\varphi}; \mathbf{x})$, can be also defined explicitly as:

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\varphi}; \mathbf{x}) = \log P_{\boldsymbol{\theta}}(\boldsymbol{x}) - D_{\mathrm{KL}}\big(Q_{\boldsymbol{\varphi}}(\boldsymbol{z}|\boldsymbol{x}) \| P_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})\big).$$

Because KL-divergence is nonnegative, $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\varphi}; \mathbf{x}) \leq \log P_{\boldsymbol{\theta}}(\boldsymbol{x})$.

By using simple properties of conditional and joint probability, we get that

$$\begin{aligned}
\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\varphi}; \mathbf{x}) &= \mathbb{E}_{Q_{\boldsymbol{\varphi}}(\boldsymbol{z}|\boldsymbol{x})}\big[\log P_{\boldsymbol{\theta}}(\boldsymbol{x}) + \log P_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}) - \log Q_{\boldsymbol{\varphi}}(\boldsymbol{z}|\boldsymbol{x})\big] \\
&= \mathbb{E}_{Q_{\boldsymbol{\varphi}}(\boldsymbol{z}|\boldsymbol{x})}\big[\log P_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z}) - \log Q_{\boldsymbol{\varphi}}(\boldsymbol{z}|\boldsymbol{x})\big] \\
&= \mathbb{E}_{Q_{\boldsymbol{\varphi}}(\boldsymbol{z}|\boldsymbol{x})}\big[\log P_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) + \log P(\boldsymbol{z}) - \log Q_{\boldsymbol{\varphi}}(\boldsymbol{z}|\boldsymbol{x})\big] \\
&= \mathbb{E}_{Q_{\boldsymbol{\varphi}}(\boldsymbol{z}|\boldsymbol{x})}\big[\log P_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\big] - D_{\mathrm{KL}}\big(Q_{\boldsymbol{\varphi}}(\boldsymbol{z}|\boldsymbol{x}) \| P(\boldsymbol{z})\big).
\end{aligned}$$

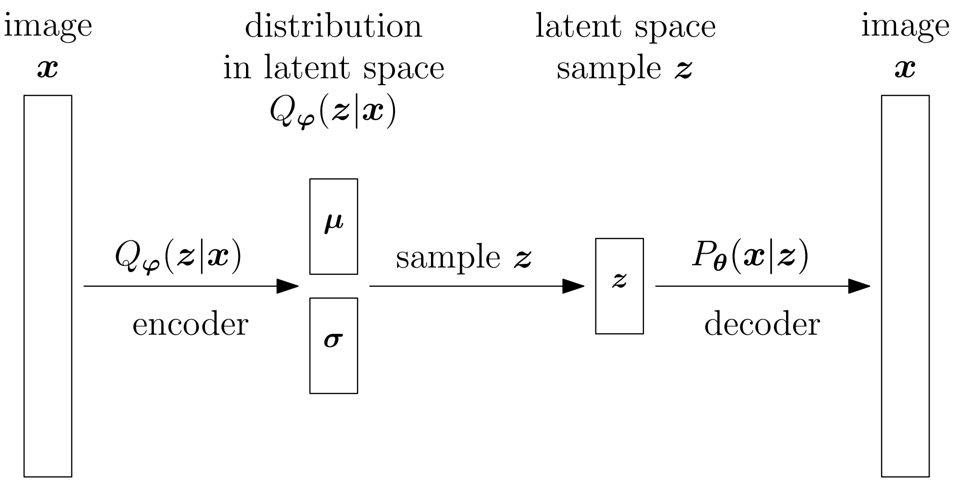
Variational AutoEncoders Training

$$-\mathcal{L}(\boldsymbol\theta, \boldsymbol\varphi; \mathbf{x}) = \mathbb{E}_{Q_{\boldsymbol\varphi}(\boldsymbol z|\boldsymbol x)}\big[-\log P_{\boldsymbol\theta}(\boldsymbol x|\boldsymbol z)\big] + D_\textrm{KL}\big(Q_{\boldsymbol\varphi}(\boldsymbol z|\boldsymbol x) \| P(\boldsymbol z)\big)$$

- We train a VAE by minimizing the $-\mathcal{L}(\boldsymbol\theta, \boldsymbol\varphi; \mathbf{x})$.
- The $\mathbb{E}_{Q_{\boldsymbol\varphi}(\boldsymbol z|\boldsymbol x)}$ is estimated using a single sample.
- The distribution $Q_{\boldsymbol\varphi}(\boldsymbol z|\boldsymbol x)$ is parametrized as a normal distribution $\mathcal{N}(\boldsymbol z|\boldsymbol\mu, \boldsymbol\sigma^2)$, with the model predicting $\boldsymbol\mu$ and $\boldsymbol\sigma$ given $\boldsymbol x$.
  - In order for $\boldsymbol\sigma$ to be positive, we can use $\exp$ activation function (so that the network predicts $\log\boldsymbol\sigma$ before the activation), or for example a $\operatorname{softplus}$ activation function.
  - The normal distribution is used, because we can sample from it efficiently, we can backpropagate through it and we can compute $D_\textrm{KL}$ analytically; furthermore, if we decide to parametrize $Q_{\boldsymbol\varphi}(\boldsymbol z|\boldsymbol x)$ using mean and variance, the maximum entropy principle suggests we should use the normal distribution.
- We use a prior $P(\boldsymbol z) = \mathcal{N}(\mathbf{0}, \boldsymbol I)$.

Variational AutoEncoders Training

$$-\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\varphi}; \mathbf{x}) = \mathbb{E}_{Q_{\boldsymbol{\varphi}}(\boldsymbol{z}|\boldsymbol{x})}\Big[-\log P_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\Big] + D_{\mathrm{KL}}\Big(Q_{\boldsymbol{\varphi}}(\boldsymbol{z}|\boldsymbol{x}) \| P(\boldsymbol{z})\Big)$$



Note that the loss has 2 intuitive components:

- **reconstruction loss** – starting with $\boldsymbol{x}$, passing though $Q_{\boldsymbol{\varphi}}$, sampling $\boldsymbol{z}$ and then passing through $P_{\boldsymbol{\theta}}$ should arrive back at $\boldsymbol{x}$;
- **latent loss** – over all $\boldsymbol{x}$, the distribution of $Q_{\boldsymbol{\varphi}}(\boldsymbol{z}|\boldsymbol{x})$ should be as close as possible to the prior $P(\boldsymbol{z}) = \mathcal{N}(\mathbf{0}, \boldsymbol{I})$, which is independent on $\boldsymbol{x}$.

Variational AutoEncoders – Reparametrization Trick

In order to backpropagate through $\boldsymbol{z} \sim Q_\varphi(\boldsymbol{z}|\boldsymbol{x})$, note that if

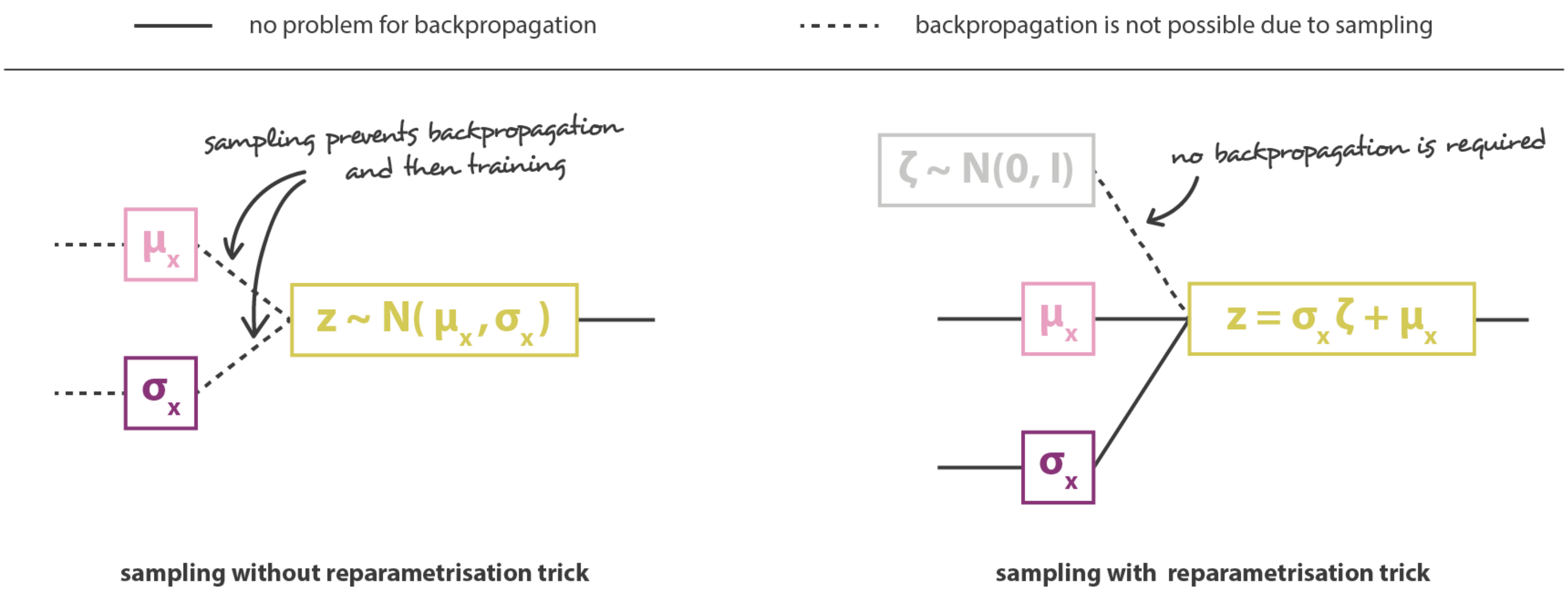
$$\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^2),$$

we can write $\boldsymbol{z}$ as

$$\boldsymbol{z} \sim \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \mathcal{N}(\mathbf{0}, \boldsymbol{I}).$$

Such formulation then allows differentiating $\boldsymbol{z}$ with respect to $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ and is called a **reparametrization trick** (Kingma and Welling, 2013).

Variational AutoEncoders – Reparametrization Trick



*https://miro.medium.com/max/3704/1*S8CoO3TGtFBpzv8GvmgKeg@2x.png*

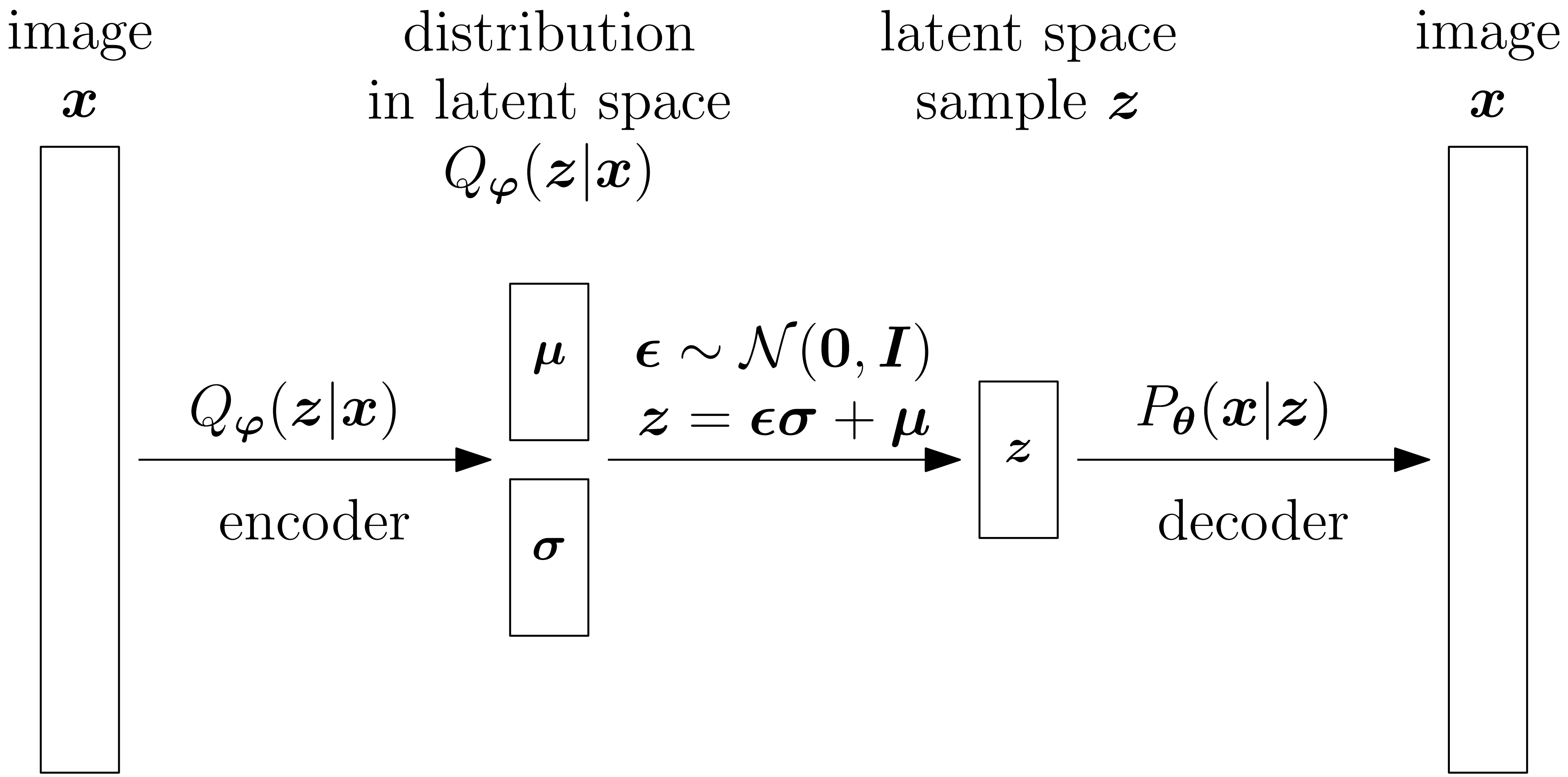
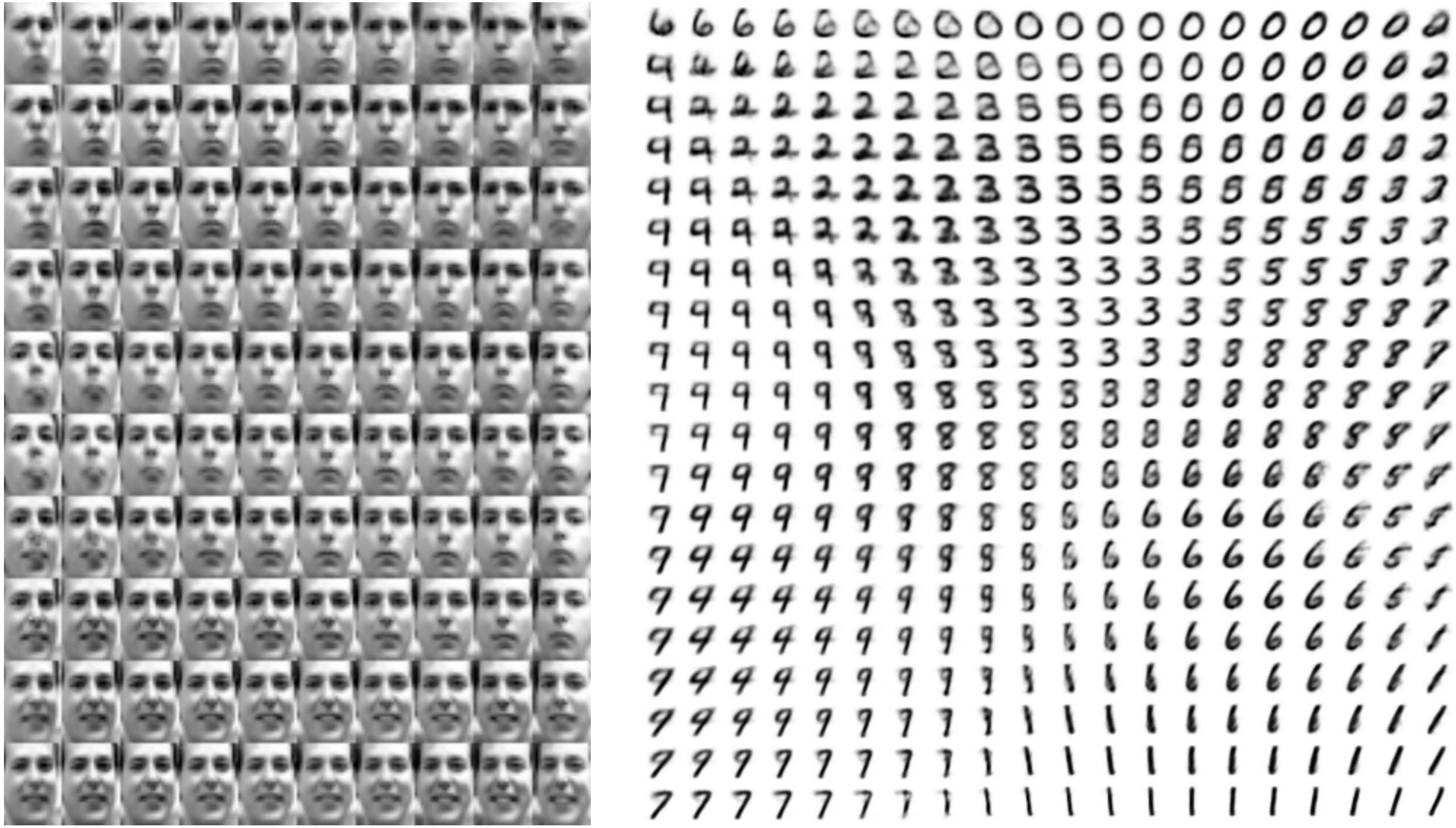


image
$\boldsymbol{x}$
distribution
in latent space
$Q_\varphi(\boldsymbol{z}|\boldsymbol{x})$
latent space
sample $\boldsymbol{z}$
image
$\boldsymbol{x}$
$Q_\varphi(\boldsymbol{z}|\boldsymbol{x})$
encoder
$\boldsymbol{\mu}$
$\boldsymbol{\sigma}$
$\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$
$\boldsymbol{z} = \boldsymbol{\epsilon}\boldsymbol{\sigma} + \boldsymbol{\mu}$
$\boldsymbol{z}$
$P_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$
decoder

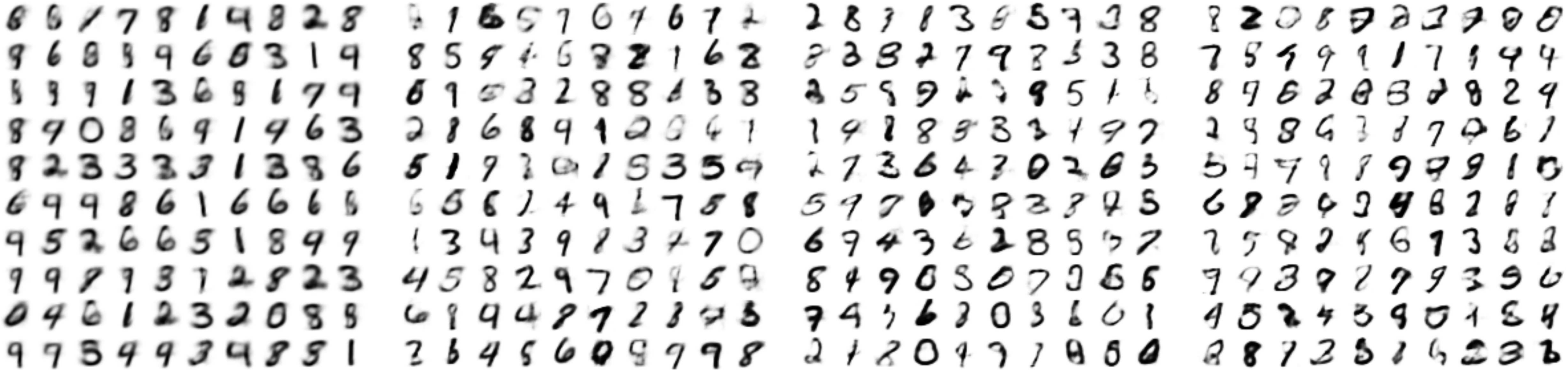


(a) Learned Frey Face manifold

(b) Learned MNIST manifold

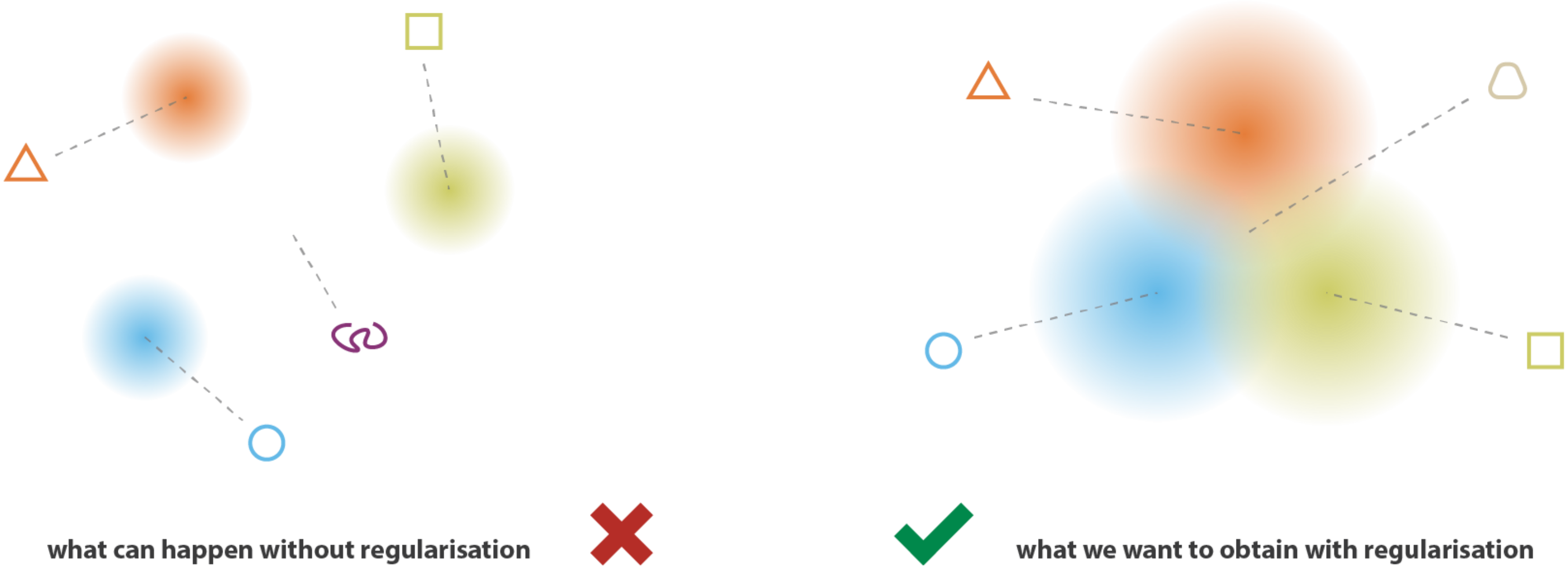
Figure 4 of "Auto-Encoding Variational Bayes", https://arxiv.org/abs/1312.6114

Variational AutoEncoders



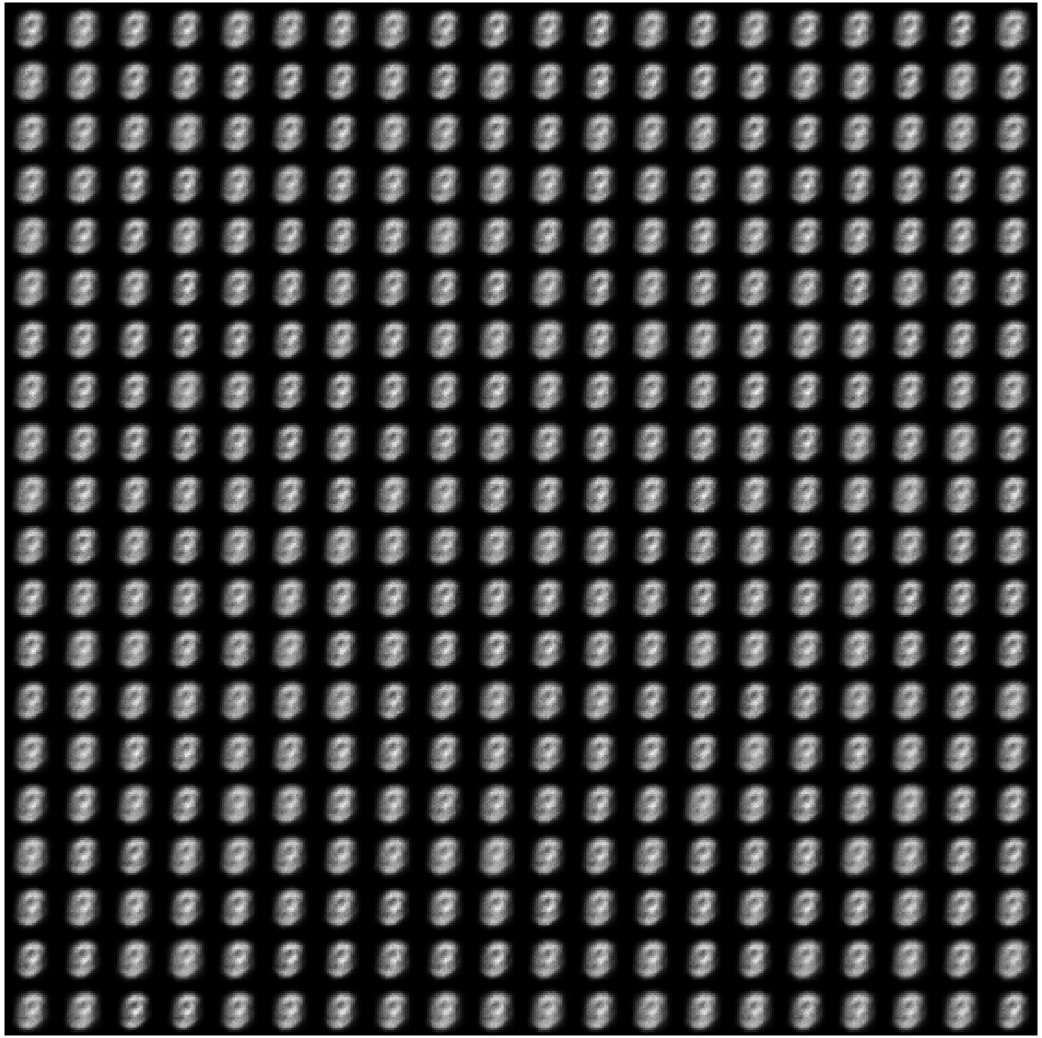
(a) 2-D latent space (b) 5-D latent space (c) 10-D latent space (d) 20-D latent space

Figure 5 of "Auto-Encoding Variational Bayes", https://arxiv.org/abs/1312.6114



*https://miro.medium.com/max/3742/1*9ouOKh2w-b3NNOVx4Mw9bg@2x.png*

Variational AutoEncoders – Too High Latent Loss



Variational AutoEncoders – Too High Reconstruction Loss