Removing RL from RLHF

Recall we want to maximize the following objective in RLHF

$$\mathbb{E}_{\hat{y}\sim p_\theta^{RL}(\hat{y}|x)}[RM_\phi(x,\hat{y}) - \beta \log \left(\frac{p_\theta^{RL}(\hat{y}|x)}{p^{PT}(\hat{y}|x)}\right)]$$

There is a closed form solution to this:

$$p^*(\hat{y}|x) = \frac{1}{Z(x)} p^{PT}(\hat{y}|x) \exp(\frac{1}{\beta} RM(x,\hat{y}))$$

- Rearrange this via a log transformation

$$RM(x,\hat{y}) = \beta\,(\log p^*(\hat{y}|x) - \log p^{PT}(\hat{y}|x)) + \beta \log Z(x) = \beta \log \frac{p^*(\hat{y}|x)}{p^{PT}(\hat{y}|x)} + \beta \log Z(x)$$

- This holds true for any arbitrary LMs, thus

$$RM_\theta(x,\hat{y}) = \beta \log \frac{p_\theta^{RL}(\hat{y}|x)}{p^{PT}(\hat{y}|x)} + \beta \log Z(x)$$

Putting It Together for DPO

{ √DPO: like finetune, direct use human data.
RLHF: train Reward Model, sample from reward model.

- Derived reward model: $RM_\theta(x,\hat{y}) = \beta \log \frac{p_\theta^{RL}(\hat{y}|x)}{p^{PT}(\hat{y}|x)} + \beta \log Z(x)$

- Final DPO loss via the Bradley-Terry model of human preferences:

$$J_{DPO}(\theta) = -\mathbb{E}_{(x,y_w,y_l)\sim D}[\log \sigma(RM_\theta(x,y_w) - RM_\theta(x,y_l))]$$

Log Z term cancels as the loss only measures differences in rewards

$$= -\mathbb{E}_{(x,y_w,y_l)\sim D}\left[\log \sigma(\beta \log \underbrace{\frac{p_\theta^{RL}(y_w|x)}{p^{PT}(y_w|x)}}_{\text{Reward for winning sample}} - \beta \log \underbrace{\frac{p_\theta^{RL}(y_l|x)}{p^{PT}(y_l|x)}}_{\text{Reward for losing sample}})\right]$$

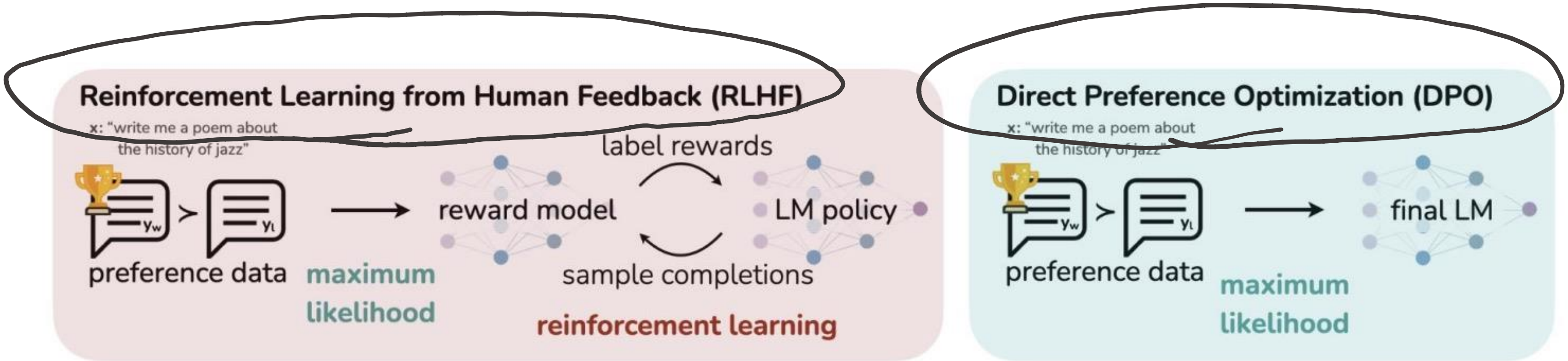
DPO don't need to train Reward Model like RLHF,

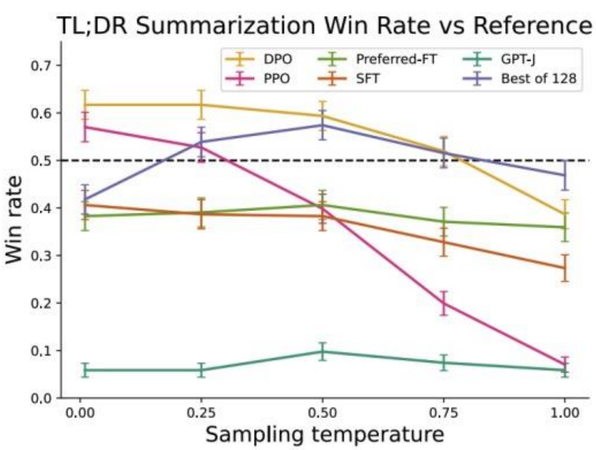
RLHF: $RM_\varphi(x, \hat{y})$ about RM_φ [Rafailov+ 2023] (PPO)

DPO: $E[\log \sigma(RM_\theta(x, y_w) - RM_\theta(x, y_l))]$

about y_w and y_l (Adam)

DPO Performs Better





- You can replace the complex RL part with a very simple weighted MLE objective
- Other variants (KTO, IPO) now emerging too
- TL;DR summarization win rates vs. human-written summaries (GPT-4 as a judge)

Convergence of SGD

Assumption 3.2 f is a **convex function** and

$$\mathbb{E}_\xi[g(\theta,\xi)] = \nabla f(\theta),$$
$$\mathbb{E}_\xi[\|g(\theta,\xi)\|^2] \le B^2, \ \forall\theta.$$

where B is a given parameters. Variance Reduce Algorithm:

SVRG, SAGA

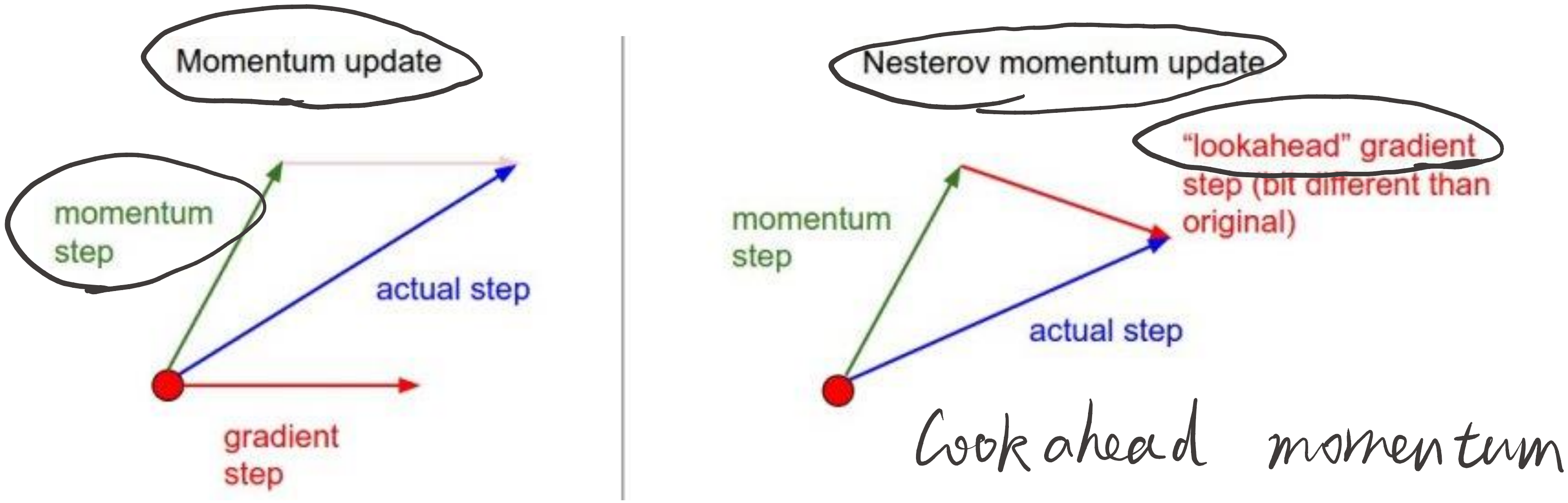
Theorem 3.2 Let $\{\theta^k\}$ be the sequence generated by SGD with step size $\alpha_k > 0$, under Assumption 3.2, for any T > 0,

$$\mathbb{E}[f(\bar{\theta}^T) - f^*] \le \frac{\|\theta^0 - \theta^*\|^2 + B^2 \sum_{j=0}^{T} \alpha_j^2}{2\sum_{j=0}^{T} \alpha_j},$$

where

$$\lambda_k = \sum_{j=0}^{k} \alpha_j, \ \bar{\theta}^k = \lambda_k^{-1} \sum_{j=0}^{k} \alpha_j \theta^j.$$

Nesterov Momentum



Start from some $\theta^0 \in \mathbb{R}^s$, $v_0 = g(\theta^0, \xi_0)$, for $k \geq 0$:

$$\begin{aligned} \vartheta^k &= \theta^k - \beta_k v^k, \\ v^{k+1} &= \beta_k v^k + \alpha_k g(\vartheta^k, \xi_k), \\ \theta^{k+1} &= \theta^k - v^{k+1}. \end{aligned}$$

An advantage: prevent overshot!

AdaGrad: Adaptive Learning Rates

Key idea: Rescale the learning rate of each coordinate by the historical progress.

2nd order moment : $E(X \cdot X^T) = \begin{pmatrix} E(x_1^2) & \cdots & E(x_1 x_n) \\ \vdots & & \vdots \\ E(x_n x_1) & \cdots & E(x_n^2) \end{pmatrix}$

σ^2

Start from some $\theta^0 \in \mathbb{R}^s$, $n_g = 0$, for $k \geq 0$:

$$\begin{aligned} n_g &= n_g + g(\theta^k, \xi_k) .* g(\theta^k, \xi_k), \\ \theta^{k+1} &= \theta^k - \alpha_k g(\theta^k, \xi_k) ./ (n_g + 10^{-8}). \end{aligned}$$

(n_g go to ∞ quickly)

Issue: The learning rate (step size) goes to zero quickly.

2nd order optimization :

$\nabla^2 f(x) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \\ & \ddots & \\ \cdots & & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}$ Hessian, Newton descent method.

ADAM

Momentum + Adagrad

Key idea: Consider momentum and adaptive learning rate (second-order momentum) together.

Momentum + Adagrad

1st order moment + 2nd order moment

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

Empirical risk minimization

Empirical risk minimization: to find a function $f(\cdot)$ to minimize

$$\frac{1}{n}\sum_{i=1}^{n} L(f(X_i,\theta),Y_i)$$

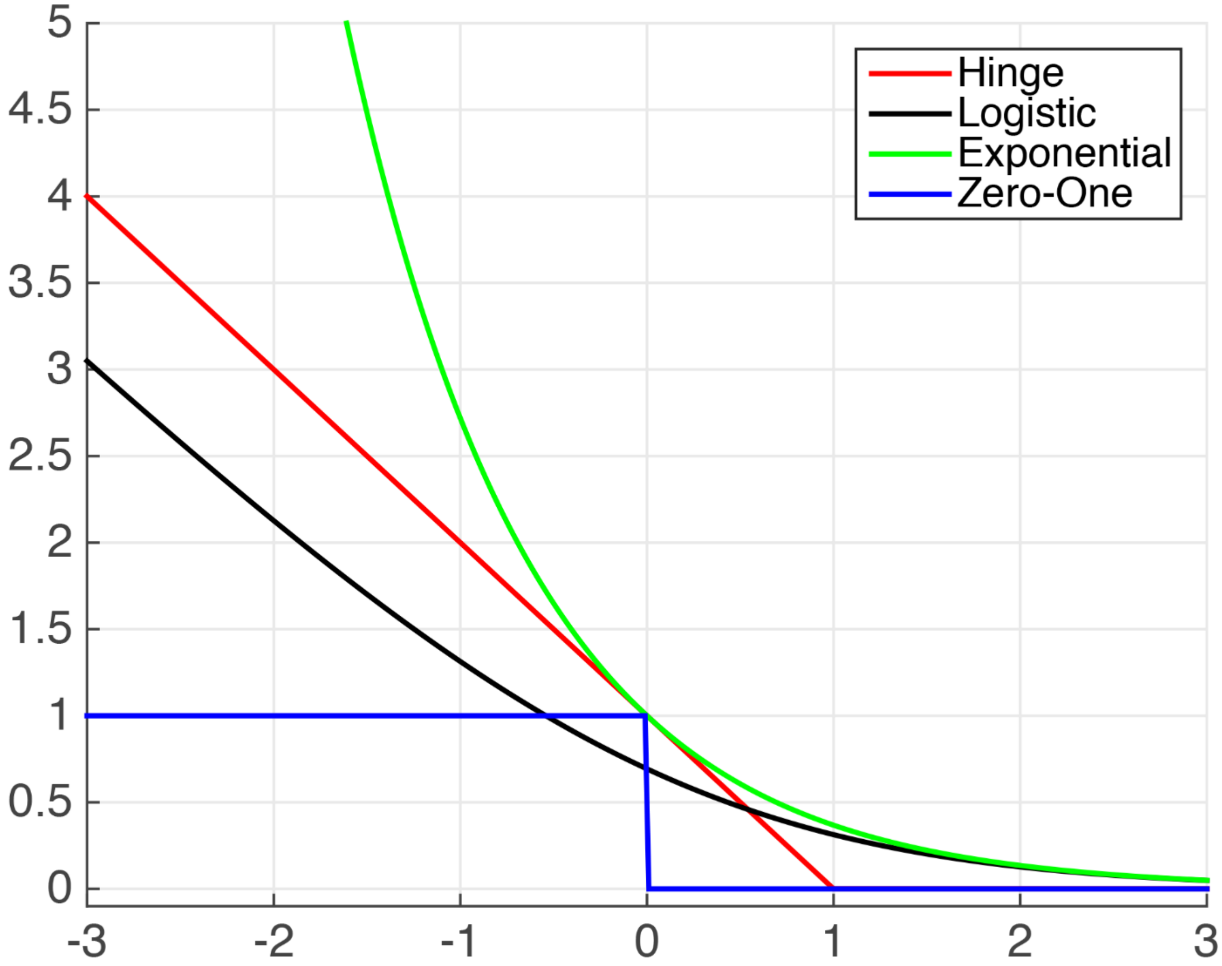
over

$$\mathcal{F} = \{f: f(x;\theta)\ is\ a\ neural\ network\ parameterized\ by\ \theta \in \mathbb{R}^s\ outputs\ real\ values\}.$$

We expect

- **Surrogate Loss function $L(\cdot,\cdot)$: continuous, smooth**
- **Neural network $f(\cdot;\theta)$: output continuous value**
- **The estimation easy to implement and explain**

Feasible loss functions



0-1 loss: $\phi(y \cdot f(x, \theta)) = I(y \cdot f(x, \theta) < 0)$

Exponential loss (AdaBoost): $\phi(y \cdot f(x, \theta)) = exp(-y \cdot f(x, \theta))$

Logistic loss : $\phi(y \cdot f(x, \theta)) = log\{1 + exp[-y \cdot f(x, \theta)]\}$

Hinge loss (SVM): $\phi(y \cdot f(x, \theta)) = max\{1 - y \cdot f(x, \theta), 0\}$

A derivation of Xavier Initialization

Consider $y = w_1x_1 + w_2x_2 + \cdots + w_nx_n$, x_i are i.i.d. with zero mean, w_i are i.i.d with zero mean.

Target: Compute $Var[y]$.

Lemma $Var[w_ix_i] = (E[w_i])^2Var[x_i] + (E[x_i])^2Var[w_i] + Var[w_i]Var[x_i]$.

Thus, $Var[w_ix_i] = Var[w_i]Var[x_i]$ and

$$Var[y] = Var[w_1x_1 + w_2x_2 + \cdots + w_nx_n] = \sum_{i=1}^{n} Var[w_ix_i] = nVar[w_i]Var[x_i]$$

Thus,

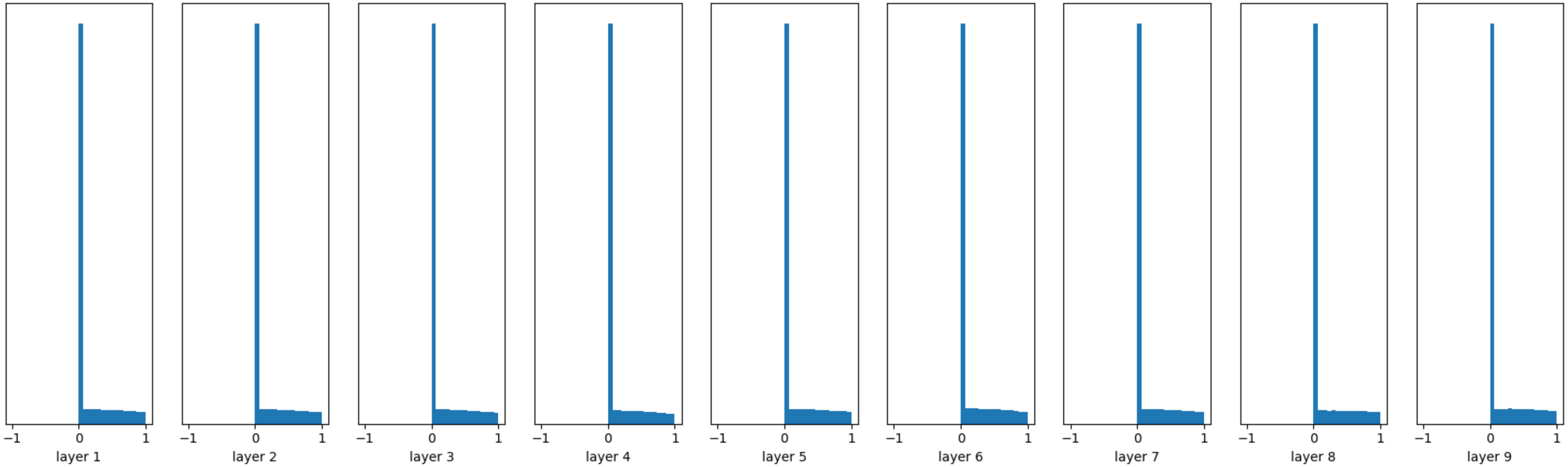
$$Var[w_i] = 1/n$$

$w_i \sim N(0,1)/\sqrt{n}$

He’s Xavier Initialization for ReLu Activation Function

Key Motivation: Assume that only a half of the neurons are activated in each layer.

He’s Xavier initialization: **N(0, 1)/sqrt(n_in/2)**.



Generative models

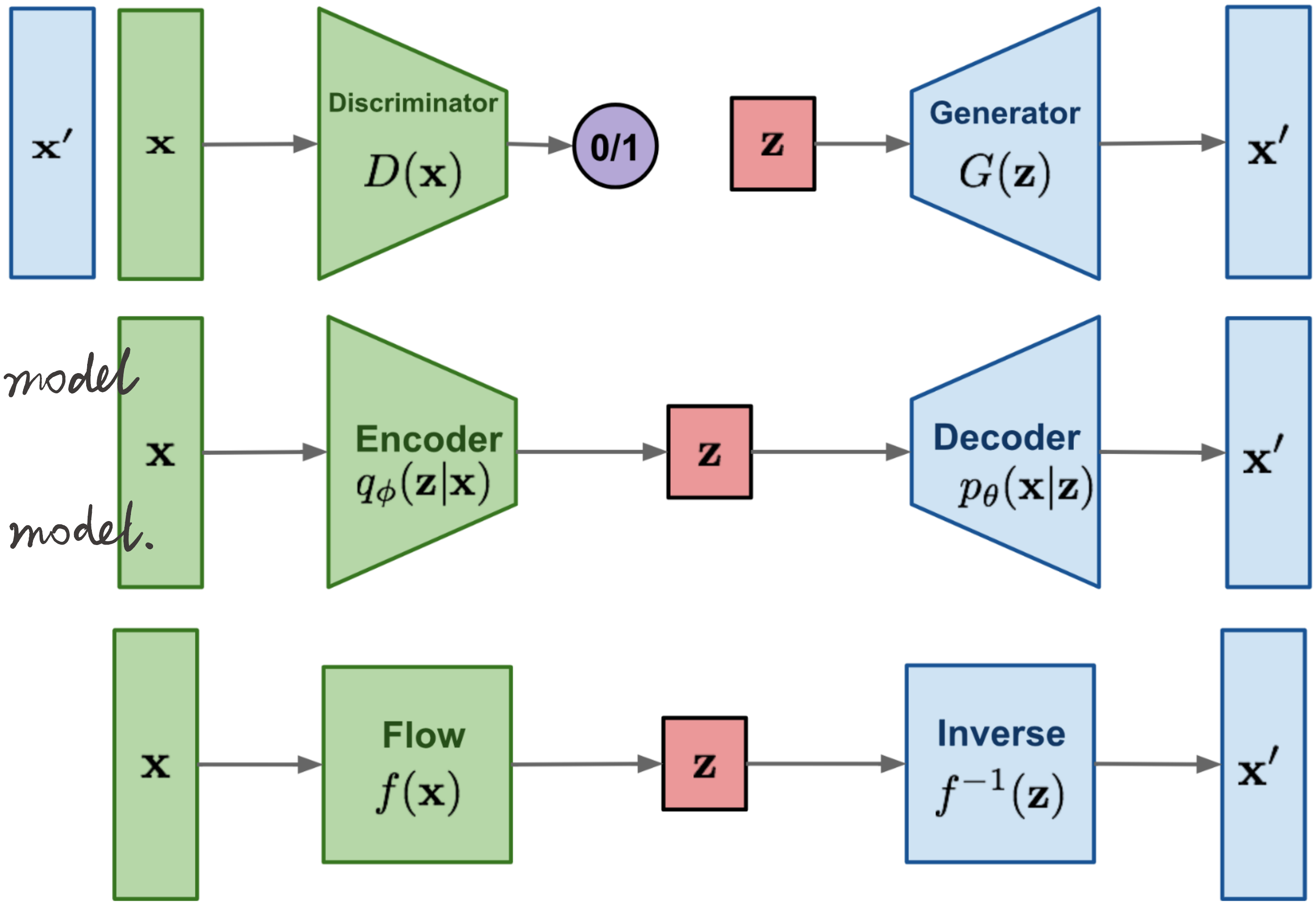
GAN: minimax the classification error loss.

probability model

VAE: maximize ELBO.

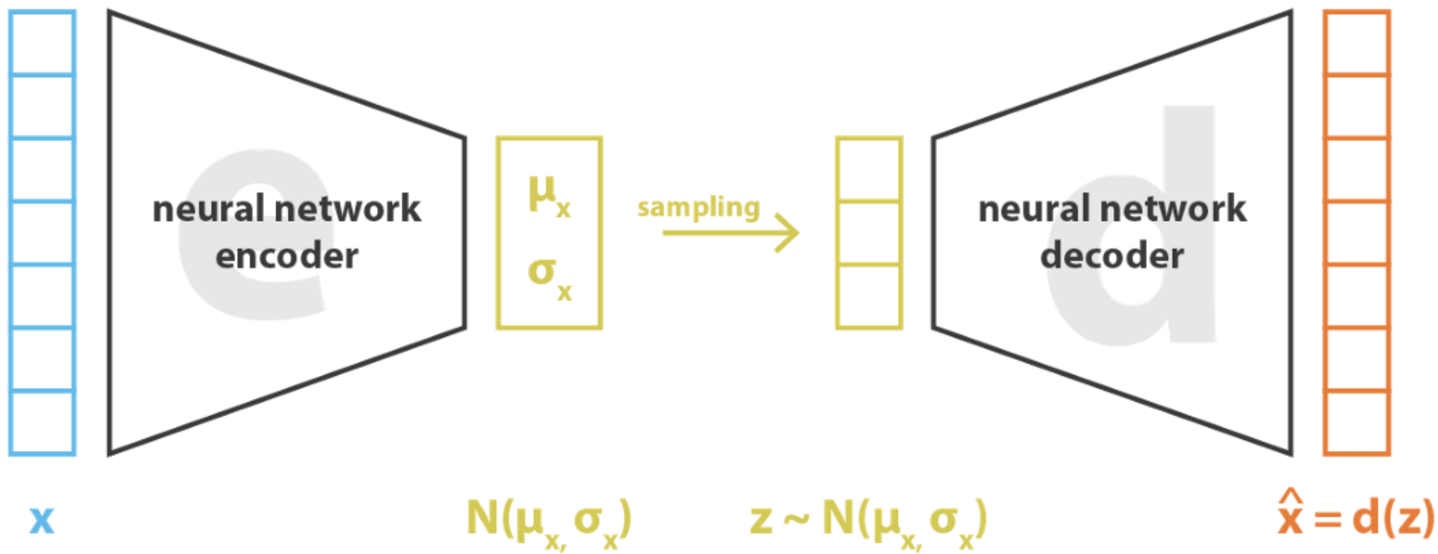
AE: compression model.

Flow-based generative models: minimize the negative log-likelihood



Source:https://lilianweng.github.io/posts/2018-10-13-flow-models/

Variational Auto-encoder

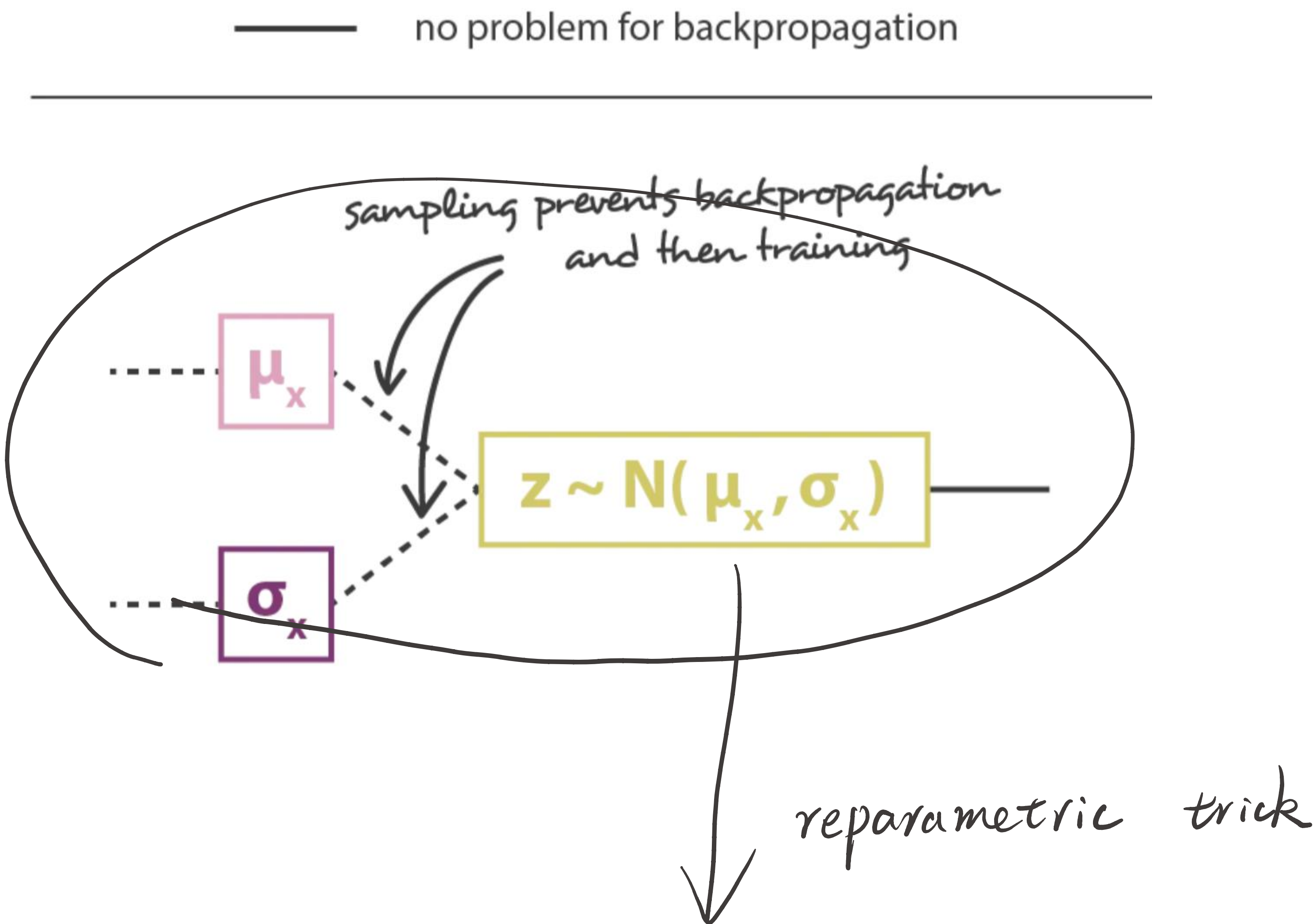


$$\text{loss} = \| x - \hat{x} \|^2 + \mathrm{KL}[\, N(\mu_x, \sigma_x), N(0, I) \,] = \| x - d(z) \|^2 + \mathrm{KL}[\, N(\mu_x, \sigma_x), N(0, I) \,]$$

The loss function is composed of
a reconstruction term and a regularisation term.

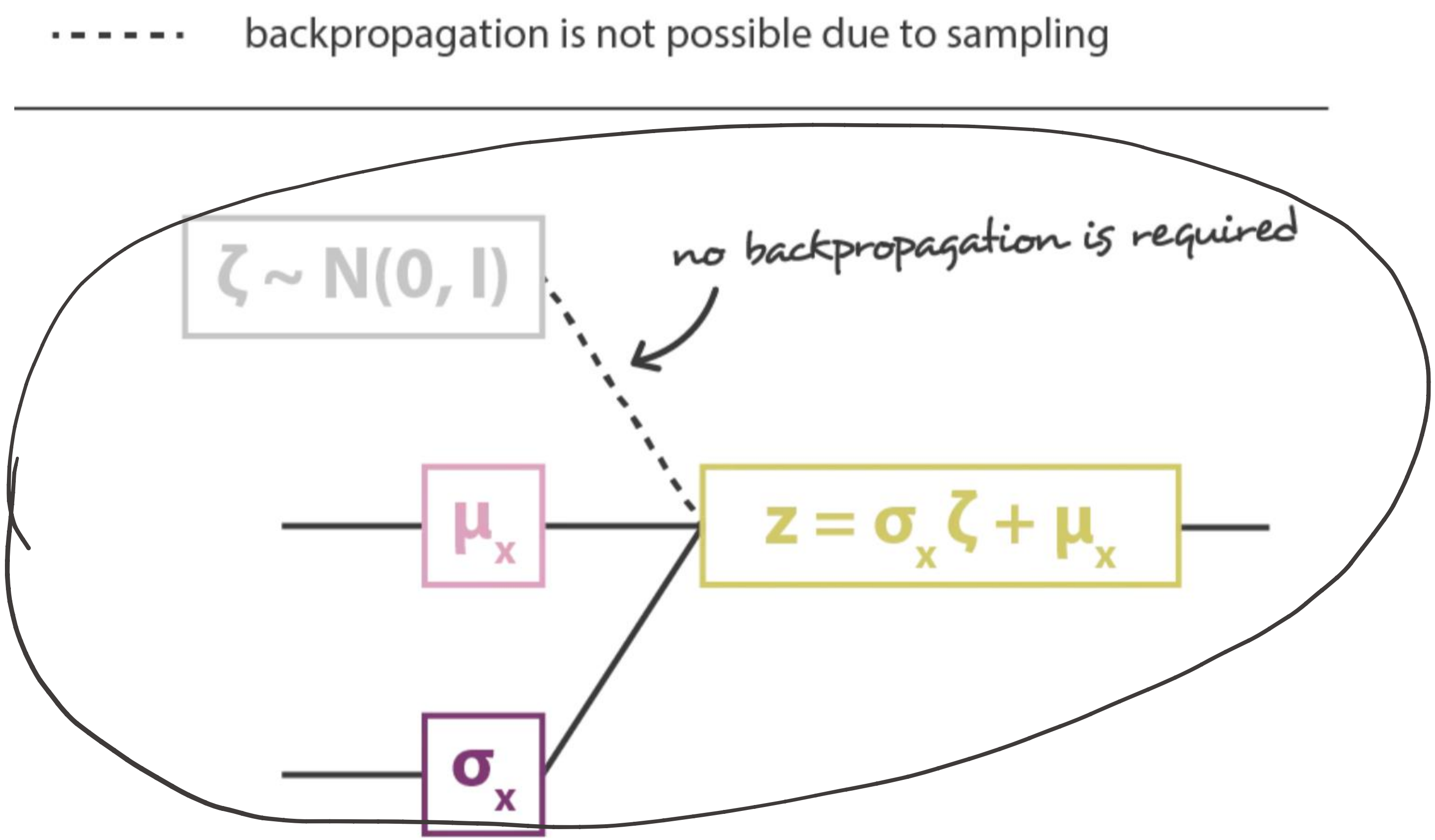
Variational Auto-encoder

Sampling prevents backpropagation and the training



Reparameterization trick

$$z = \sigma_x \zeta + \mu_x \qquad \zeta \sim N(0, I)$$



Score-based models

diffusion model

- Train score-based models by minimizing the Fisher divergence

$$\mathbb{E}_{p(\mathbf{x})}[\|\nabla_\mathbf{x} \log p(\mathbf{x}) - \mathbf{s}_\theta(\mathbf{x})\|_2^2]$$

- Once trained a model $s_\theta(x) \approx \nabla_x \log p(x)$, we can use an iterative procedure called Langevin dynamics to draw samples from it.

- It initializes the chain from an arbitrary prior distribution $x_0 \sim \pi(x)$, and then iterates the following

$$\mathbf{x}_{i+1} \leftarrow \mathbf{x}_i + \epsilon \nabla_\mathbf{x} \log p(\mathbf{x}) + \sqrt{2\epsilon}\, \mathbf{z}_i, \quad i = 0, 1, \cdots, K,$$

where z_i follow standard Gaussian distribution.

- When $\epsilon \to 0$ and $K \to \infty$, x_K obtained from the procedure converges to a sample from $p(x)$ under some regularity conditions

Recurrent Neural Network

Elman Network

X_1: He → $h_1 = \sigma_h(W_h X_1 + U_h h_0 + b_h)$ → $Y_1 = \sigma_y(W_y h_1 + b_y)$

X_2: is → $h_2 = \sigma_h(W_h X_2 + U_h h_1 + b_h)$ → $Y_2 = \sigma_y(W_y h_2 + b_y)$

X_3: a → $h_3 = \sigma_h(W_h X_3 + U_h h_2 + b_h)$ → $Y_3 = \sigma_y(W_y h_3 + b_y)$

X_4: lucky → $h_4 = \sigma_h(W_h X_4 + U_h h_3 + b_h)$ → $Y_4 = \sigma_y(W_y h_4 + b_y)$

X_5: dog → $h_5 = \sigma_h(W_h X_5 + U_h h_4 + b_h)$ → $Y_5 = \sigma_y(W_y h_5 + b_y)$

Jordan Network

X_1: He $\rightarrow$ $h_1 = \sigma_h(W_h X_1 + U_h Y_0 + b_h)$ $\rightarrow$ $Y_1 = \sigma_y(W_y h_1 + b_y)$

X_2: is $\rightarrow$ $h_2 = \sigma_h(W_h X_2 + U_h Y_1 + b_h)$ $\rightarrow$ $Y_2 = \sigma_y(W_y h_2 + b_y)$

X_3: a $\rightarrow$ $h_3 = \sigma_h(W_h X_3 + U_h Y_2 + b_h)$ $\rightarrow$ $Y_3 = \sigma_y(W_y h_3 + b_y)$

X_4: lucky $\rightarrow$ $h_4 = \sigma_h(W_h X_4 + U_h Y_3 + b_h)$ $\rightarrow$ $Y_4 = \sigma_y(W_y h_4 + b_y)$

X_5: dog $\rightarrow$ $h_5 = \sigma_h(W_h X_5 + U_h Y_4 + b_h)$ $\rightarrow$ $Y_5 = \sigma_y(W_y h_5 + b_y)$

Long Short-Term Memory

The forward pass of an LSTM cell with a forget gate are

$$
\begin{aligned}
f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\
\tilde{c}_t &= \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\
h_t &= o_t \odot \sigma_h(c_t)
\end{aligned}
$$

where the initial values are $\boldsymbol{c_0} = \mathbf{0}$ and $\boldsymbol{h_0} = \mathbf{0}$, and the operator $\odot$ denotes the Hadamard product (element-wise product).

- $x_t \in \mathbb{R}^d$: input vector to the LSTM unit
- $f_t \in (0,1)^h$: forget gate's activation vector
- $i_t \in (0,1)^h$: input/update gate's activation vector
- $o_t \in (0,1)^h$: output gate's activation vector
- $h_t \in (-1,1)^h$: hidden state vector
- $\tilde{c}_t \in (-1,1)^h$: cell input activation vector
- $c_t \in \mathbb{R}^h$: cell state vector

Word2Vec Algorithm

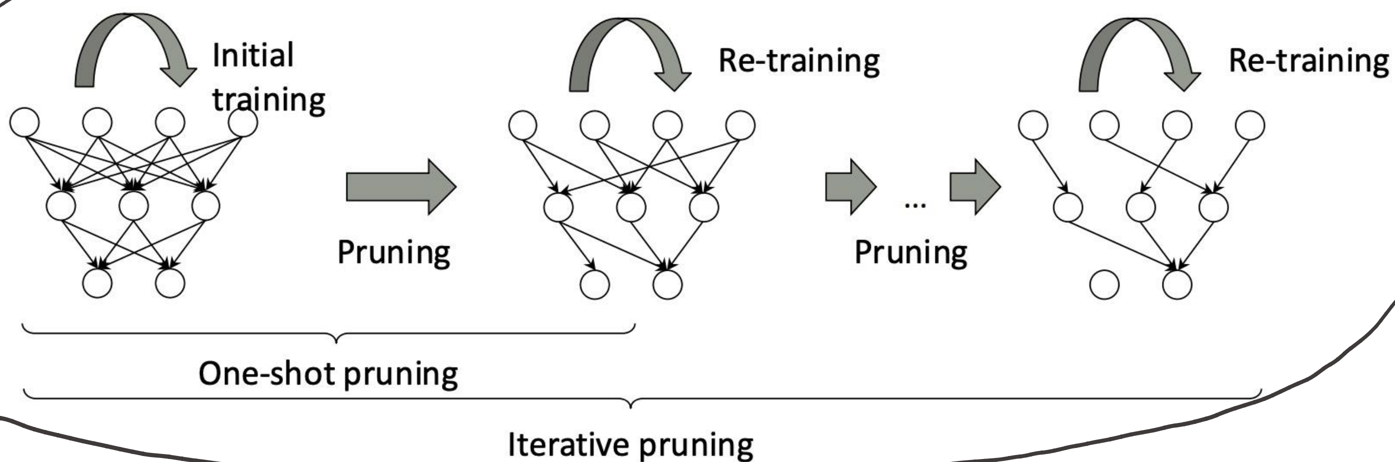
- **Continuous Skip-gram Model:**
  predicts words within a certain range before and after the current word in the same sentence.

- **Continuous Bag-of-Words Model (CBOW):**
  predicts the middle word based on surrounding context words. The context consists of a few words before and after the current (middle) word. This architecture is called a bag-of-words model as the order of words in the context is not important.

To train a neural network with a single hidden layer to perform a prediction task. But the goal is to learn the weights of the hidden layer–these weights are the "word vectors".

Pruning

- During pruning, a fraction of the lowest-magnitude weights are removed
- The non-pruned weights are re-trained
- Pruning for multiple iterations is more common (Frankle & Carbin, 2019)



The Lottery Ticket Hypothesis

- Dense, randomly-initialized models **contain subnetworks** ("winning tickets") that—when trained in isolation—**reach test accuracy comparable to the original network** in a similar number of iterations [Frankle & Carbin, 2019]

Frankle, J., & Carbin, M. (2018, September). The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. In International Conference on Learning Representations.