1 Deep Learning Basics

Deep Learning: write in the front

Deep Learning is a class of **machine learning** methods that use **neural networks** to learn **representations** from raw data.
A DL algorithm *always* consists of neural network (model) structure, training procedure (loss function) and inference procedure.
Always use **backpropagation**.
Always use **mini-batch** of data to compute gradient, and then do **gradient descent** on each dimension; the optimizer can vary, such as SGD, RMSprop or Adam.
Convention: if a function f is a neural network and its parameters are θ, we write $f_\theta(\cdot)$ or $f(\cdot;\theta)$ and say f is **parameterized** by θ.
The loss function is $\mathcal{L}$, the dataset is $\mathcal{D}$.

Deep Learning: about loss function

The loss function should *always* be **differentiable**.
We *always prefer* **tractable** loss. If the loss is not tractable, we have to estimate the gradient (using Monte-Carlo), which introduces instability and error.
What is tractable?
$\mathbb{E}_{x\sim\mathcal{D}}$ and $\mathbb{E}_{z\sim\mathcal{N}(0,I)}$ are both tractable, since we can sample a mini-batch from dataset $\mathcal{D}$ and Gaussian $\mathcal{N}$ (or other known distribution) during training.
The **direct output** of the model.
CAVEAT: If the model inputs z and outputs x, then $p_\theta(x|z)$ is tractable, but $p_\theta(z|x)$ is often not! This is what "direct output" means.

Optimization

Gradient Descent (GD): $\theta \leftarrow \theta - \eta\cdot\nabla_\theta\mathcal{L}$.
Stochastic GD (SGD) (in DL scenario): sample a small batch of data, then compute average loss as the estimation of the true $\mathcal{L}$.
Momentum: $\theta_{t+1}\leftarrow\theta_t-\eta\cdot\nabla_{\theta_t}\mathcal{L}(\theta_t)+\beta\cdot(\theta_t-\theta_{t-1})$, where θ_t is the parameters at step t. **No convergence guarantee**.
Nesterov's Method: $\theta_{t+1}\leftarrow\theta_t-\eta\cdot\nabla_{\theta_t}\mathcal{L}(\theta_t+\beta\cdot(\theta_t-\theta_{t-1}))+\beta\cdot(\theta_t-\theta_{t-1})$. **Faster convergence guarantee in smooth functions**.
AdaGrad: $\theta_{t+1}[i]=\theta_t[i]-\frac{\eta}{\sqrt{G_t[i]+\epsilon}}(\nabla_{\theta_t}\mathcal{L})[i]$, $G_t[i]=\sum_{s<t}|(\nabla_{\theta_s}\mathcal{L})[i]|^2$.
RMSProp: Change $G_t[i]=\gamma\cdot G_{t-1}[i]+(1-\gamma)|(\nabla_{\theta_s}\mathcal{L})[i]|^2$ ("moving average")
Adam: Change the gradient term $\nabla_{\theta_t}\mathcal{L}$ in RMSProp into the momentum version

$$M_t=\delta M_{t-1}+(1-\delta)\nabla_{\theta_t}\mathcal{L}$$

compute bias-corrected first moment and second raw moment estimate

$$\hat{G}_t=\frac{G_t}{1-\delta^t}\quad \hat{M}_t=\frac{M_t}{1-\gamma^t}$$

update rule

$$\theta_{t+1}[i]=\theta_t[i]-\frac{\eta}{\sqrt{\hat{G}_t[i]+\epsilon}}\hat{M}_t[i]$$

RMSProp and Adam are common choices in modern DL.
Essentially, *AdaGrad and RMSProp are making the learning rate in each dimension different (instead of a unified constant η)*. For example, in AdaGrad, the learning rate of dimension i is $\frac{\eta}{\sqrt{G_t[i]+\epsilon}}$.

Optimization: Practices

Why SGD but not GD? Computational affordable; avoid saddle points.
What is the benefit of RMSProp compared to AdaGrad? It can avoid vanishing learning rate.
T/F: Adam, AdaGrad are making the learning rate different and disentangled in different dimension. True.
T/F: Although Nesterov Momentum has no convergence guarantee, it is commonly used in real training. False. None of them is true.
T/F: SGD and GD have the same mean and variance in training. False.
T/F: Second-order optimization is more stable than gradient descent. False. It can diverge for non-convex functions due to negative eigenvalues.

2 Model Architecture

Multi-layer Perceptron (MLP)

Definition: alternating between **linear layer** and **activation layer**.
Linear layer: $x_{\text{output}}=Wx_{\text{input}}+b$, where W and b are learnable.
Activation layer: a non-linear layer applied to each neuron.

- **ReLU**: $f(x)=\max\{x,0\}$
- **LeakyReLU**: $f(x)=\max\{x,kx\}$ $(0<k<1)$
- **Sigmoid**: $f(x)=\frac{1}{1+e^{-x}}$, $f'(x)=f(x)(1-f(x))$.
- **Tanh**: $f(x)=\frac{e^x-e^{-x}}{e^x+e^{-x}}$, $f'(x)=1-f(x)^2$.

Convolutional Layer

Key insight: **invariance** (In Vision, the image is *spatial invariant*; in sequential data, there is *temporal invariance*).
The most common usage is in CV, where the input is an image $H\times W\times C$ (C is the number of **channels**).
Important attributes:

- Kernel (filter) size;
- Padding;
- Stride.

Pytorch: `torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0)`.
Practice: apply `Conv2d(3,4,3,2,1)` on a $15\times15\times3$ image, the output size is $8\times8\times4$, the number of parameters is $3\times4\times3^2+4=112$.

Pooling

Common choice: map a square block to a single value, reducing the size and introducing non-linearity. (Apply independently on each channel)

- Max pooling;
- Average pooling (still linear).

Normalization

Empirically, applying normalization layers can stabilize training.
Another motivation: to solve "covariance shift" across batches.
Many choices: BatchNorm, LayerNorm, GroupNorm, InstanceNorm, $\cdots$. **The ONLY DIFFERENCE is to choose which group of values to normalize.**
For instance, if we group L values $x_1,\cdots,x_L$, first compute the norm and std

$$\mu=\frac{x_1+\cdots+x_L}{L},\quad \sigma^2=\frac{1}{L-1}\sum_{i=1}^{L}(x_i-\mu)^2.$$

Then, x_i will transfer to

$$\gamma\cdot\frac{x_i-\mu}{\sqrt{\sigma^2+\epsilon}}+\beta,$$

where γ,β are two learnable parameters.
For vision, the input size is $N\times H\times W\times C$.

- **BatchNorm**: divide into C groups, each having size $N\times H\times W$.
- **InstanceNorm**: divide into $N\times C$ groups, each having size $H\times W$.
- **LayerNorm** (default): divide into N groups, each having size $H\times W\times C$.
- **GroupNorm**: divide into $N\times G$ (hyperparam) groups, each having size $H\times W\times(C/G)$.
  $\to G=1$ is LayerNorm, $G=C$ is InstanceNorm.

All layers independently operate on each data in a batch, except BatchNorm. For BatchNorm, we also maintain a "moving average" of μ,σ^2, which is used during sampling.

Skip Connection

Motivation: network is hard to learn identity, but easy to learn zero.

- **Residual Block**: for a block of convolutional layers $\mathcal{F}$, let the output be $x+\mathcal{F}(x)$.
- **Dense Block**: concat$(x,\mathcal{F}(x))$. "Take output of all previous layers."

Using skip connections, increasing the depth of a model will increase the performance, even if it is already VERY LARGE (e.g., 100 layers).

CNN for Classification

Apply sequentially the following:

- Several convolutional / residual / dense blocks. Each block has convolution, activation, and normalization (probably pooling, if we want to reduce size).
- Global pooling, getting a C-dim feature.
- A MLP layer to output the final logits.

Important Tricks

- Dropout & early-stopping & regularization $\to$ alleviate overfitting.
- Initialization tricks (Xavier / Kaiming init for MLP; zero-init for residual blocks).
- Learning rate schedule (lr decay; lr warmup).
- Data augmentation.
- Gradient Clipping.

Model Architecture: Practices

- Skip connection is a common technique in neural network design, e.g. ResNet, DenseNet, Transformers. True.
- When training a CNN with BatchNorm, it's equivalent to use batch size 32 for one step and use batch size 2 for 16 steps while accumulating the gradient. False. Intuitively, all data in a batch "communicates" through the batch statistics μ,σ^2.
- In evaluation, BatchNorm normalizes the input by the mean and variance of a batch. False. In evaluation (inference, sampling), we use the moving average of μ,σ^2.
- When the batch size is 1, BatchNorm and LayerNorm are equivalent. False. The group separation is different.
- The input of DenseNet contains more information than ResNet. Why people still use ResNet? DenseNet makes the layer wider, memory consumption will be large. ResNet, however, does not increase the layer width.

3 Generative Models in Vision

Generative Models: Ideas

Universal Principle: learn the data distribution $p(X)$. But we can construct this distribution in different ways:

- Directly model $p(X)$, then do sampling $\to$ **EBM**
- Model the score field $\nabla_X\log p(X)$, then do Langevin Dynamics sampling $\to$ **SBM**
- Model the score field $s(X,\sigma_i)$ for *each noise level*, then gradually denoise in sampling $\to$ **Diffusion & SDE/ODE Models**
- Model $p(X)$ by creating an explicit bijection from $Z\sim\mathcal{N}(0,I)$ to $X\sim\mathcal{D}\to$ **Normalizing Flow**
- Let the prior $Z\sim\mathcal{N}(0,I)$ be the input, and the model outputs an $X=f_\theta(Z)$. Use some clever tricks to train $\to$ **VAE, GAN**

CAVEAT: No specific model structure is involved in this section, only the input and output of the model is specified.

3.1 Energy-Based Models (EBM)

Hopfield Network

A fully-connected network with N neurons (each y_i is +1 or -1), and symmetric weight w_{ij}, define energy E as

$$E(y)=-\sum_{i<j}w_{ij}y_iy_j=-\frac{1}{2}y^TWy$$

Store patterns P using local minimum energy.
Train: minimize energy for $y\in P$, maximize energy for y' as neighborhood of $y\in P$

$$W=\arg\min_W\sum_{y\in P}E(y)-\sum_{y\in P,y'\in N(y)}E(y')$$

$y'\in N(y)$ is sampled by evolution for 2-4 steps from a random pattern in P.
Sample: Introduced later.

EBM: Basic Idea

EBM directly models the distribution $p(X)$: the model takes X as input, and outputs a probability $p_\theta(X)$.
A valid distribution satisfies $\int_X p(X)dX=1$ and $p(X)\ge0,\forall X$.
To force the p_θ satisfy these, the most natural way is:

$$p_\theta(X)\propto e^{-E_\theta(X)}\iff p_\theta(X)=\frac{e^{-E_\theta(X)}}{\int_{X'}e^{-E_\theta(X')}dX'}=:\frac{e^{-E_\theta(X)}}{Z}.$$

$E_\theta(X)$ is the **energy** of the data X, Z is the **normalizing constant**.
Then, the problem divides into two parts: how to learn E_θ and how to sample from the learned p_θ.

EBM: Training

Objective: **negative log-likelihood (NLL)**,

$$\mathcal{L}=-\mathbb{E}_{x\sim\mathcal{D}}\log p(x)=\mathbb{E}_{x\sim\mathcal{D}}E(x)+\log Z.$$

The first term $E(x)$ is tractable, but the second term $\log Z$ is not (involving integration). So we need to **approximate** using **Monte-Carlo sampling** (randomly choose few samples and average).

$$Z=\int_x e^{-E_\theta(x)}dx\implies\nabla_\theta\log Z=-\frac{1}{Z}\int_x e^{-E_\theta(x)}\nabla_\theta E_\theta(x)dx$$
$$=-\mathbb{E}_{x\sim p_\theta(x)}\nabla_\theta E_\theta(x).$$

CAVEAT: Here we choose to approximate $\nabla_\theta\log Z$ since we only care about the gradient instead of the loss!

EBM: Sampling

Given a field $E(X)$, how to sample from $p(X)\propto e^{-E(X)}$?

- Metropolis Hastings (MH) Algorithm
- Gibbs Sampling
- Langevin Dynamics

Sampling Method: MH

Algorithm: Start with a random point X_0; in each step t, propose a new point X_{new}, and decide whether to move ($X_t=X_{\text{new}}$) or not ($X_t=X_{t-1}$).
Suppose we want to sample from $p(X)$. Handcraft a transition distribution $q(X_1\to X_2)$, define acceptance ratio

$$\alpha(X_1\to X_2)=\min\left\{1,\frac{p(X_2)q(X_2\to X_1)}{p(X_1)q(X_1\to X_2)}\right\}.$$

Propose a new point: sample X_{new} from $q(X_{t-1}\to X)$.
Decide whether to move or not: move with probability $\alpha(X_{t-1}\to X_{\text{new}})$.
Why do we use MH:

1. To calculate α, we only need the **ratio** of p, which is exactly $e^{E_\theta(X_2)-E_\theta(X_1)}$ (no normalizing constant here).
2. Usually, there are many or even infinite choices of X, so directly sampling from $p(X)$ is impossible. However, we can use MH to approximate it, and the accuracy can be improved by increasing the number of transition steps.

In practice, we may define the transition q by: $X_2=X_1+\epsilon$, where $\epsilon\sim\mathcal{N}(0,I)$ is a uniform Gaussian noise.

Sampling Method: Gibbs Sampling

Gibbs sampling is a **special case of MH algorithm**.
Assume: the data has the form of $\mathbf{X}=(X_1,X_2,\cdots,X_L)$, where each X_i takes value in a finite set M_i.
Algorithm: Start with a random point $\mathbf{X}^0$. In each step t, randomly select a position $1\le i\le L$, and re-sample the i-th position of $\mathbf{X}^{t-1}$ (other positions remain the same).

- Randomly select a position: any distribution is OK.
- Re-sample: there are $|M_i|$ possibilities $\mathbf{X}^1_{\text{new}},\cdots,\mathbf{X}^{|M_i|}_{\text{new}}$. Sample $\mathbf{X}^j_{\text{new}}$ with probability $\propto p(\mathbf{X}^j_{\text{new}})$.

EBM: Boltzmann Machine

Assume all data is a binary sequence with length n. Define the model $E_\theta(x)=-\frac{1}{2}x^TWx$, W is a $n\times n$ real matrix (the model parameters).

- Sampling: Gibbs sampling (since the data has finite length and finite value set)
- Training: $\nabla_WE_\theta(x)=xx^T$, so the gradient of loss is

$$\nabla_W\mathcal{L}=\nabla_W[\mathbb{E}_{x\sim\mathcal{D}}E(x)+\log Z]=\mathbb{E}_{x\sim\mathcal{D}}xx^T-\mathbb{E}_{x\sim p_\theta(x)}xx^T.$$

EBM: Boltzmann Machine Extensions

- **Hidden neuron**: add k hidden neurons, getting a binary sequence w/ length $n+k$. $\to$ increase representation power.
- **Restricted Boltzmann Machine**: change the "fully-connected" graph to a **bipartite** graph (i.e. restrict some params in W to be 0). One part is visible, another part is hidden. $\to$ Gibbs sampling can be done in parallel, increasing sample efficiency.

EBM: Practices

- T/F: EBM is hard to scale up because of the restriction of the model structure. False. EBM does not restrict the model structure, only defining the training and sampling procedure. In fact, EBM is indeed hard to scale up since calculating the loss requires inefficient Monte-Carlo sampling.
- T/F: The training of EBM involves estimating the value of normalizing constant Z. Thus, we often use Monte-Carlo method to estimate the value of Z. False. We are estimating $\nabla_\theta\log Z$, since we only care about the gradient.
- T/F: Gibbs Sampling is a special case of the MH algorithm, where the acceptance rate is always 1. True.

3.2 Score-Based Models (SBM) & Diffusion Models & SDE/ODE Models

SBM: Langevin Dynamics

We want to sample from a distribution $p(x)$. If we have the **score field** $s(x)=\nabla_x\log p(x)$, starting from a random point x_0, we iteratively sample:

$$x_{t+1}\leftarrow x_t+\epsilon\cdot s(x_t)+\sqrt{2\epsilon}\cdot z,\quad z\sim\mathcal{N}(0,I).$$

To use this sampling method, only the score field $s(x)$ is required.

SBM: Basic Idea

Besides explicitly modeling the distribution $p(X)$, we can represent the distribution $p(X)$ by its score function $s(x)=\nabla\log p(x)$. In EBM, we maximize $\mathbb{E}_{x\sim p_{\text{data}}}\log p_\theta(x)-\log Z(\theta)$. In SBM, we want to force $\nabla\log p_{\text{data}}(x)=\nabla\log p_\theta(x)$.

SBM: Training

To match $\nabla\log p_{\text{data}}(x)=\nabla\log p_\theta(x)$, the most intuitive way is minimizing the MSE:

$$\mathcal{L}(\theta)=\frac{1}{2}\mathbb{E}_{x\sim p_{\text{data}}}\left[|\nabla_x\log p_{\text{data}}(x)-\nabla_x\log p_\theta(x)|_2^2\right]$$

which is the **Fisher divergence** between p_{data} and p_θ. With some math,

$$\mathcal{L}(\theta)=\mathbb{E}_{x\sim p_{\text{data}}}\left[\frac{1}{2}|\nabla_x\log p_\theta(x)|_2^2+\text{tr}(\nabla_x^2\log p_\theta(x))\right]+\text{const}$$

Notice: No partition function $Z(\theta)$ anymore!

SBM: What is the problem?

Instead of using a network to model $p_\theta(x)$, we directly model $s_\theta(x)=\nabla_x\log p_\theta(x)$:

$$\mathcal{L}(\theta)=\mathbb{E}_{x\sim p_{\text{data}}}\left[\frac{1}{2}|s_\theta(x)|_2^2+\text{tr}(\nabla_xs_\theta(x))\right]$$

However, computing $\text{tr}(\nabla_xs_\theta(x))$ requires $O(d)$ backpropagations, which is too expensive. Also, due to the manifold hypothesis, $\nabla\log p_{\text{data}}$ is not well-defined almost everywhere, and in low data density regions, the score is not accurate. Langevin MCMC will have trouble exploring these regions.

Denoising Score Matching

Introduce noise-perturbed distribution:

- Define $q_\sigma(\tilde{x}|x)=\mathcal{N}(\tilde{x}|x,\sigma^2I)$
- $q_\sigma(\tilde{x})=\mathbb{E}_{x\sim p_{\text{data}}}q_\sigma(\tilde{x}|x)$ is the distribution after perturbing data with Gaussian noise

By matching $s_\theta(\tilde{x})$ on $q_\sigma(\tilde{x})$, s_θ covers the full high-dimensional space:

$$\mathcal{L}(\theta)=\frac{1}{2}\mathbb{E}_{x\sim p_{\text{data}},z\sim\mathcal{N}(0,I)}\left[\left|s_\theta(x+\sigma z)+\frac{z}{\sigma}\right|_2^2\right]+\text{const}$$

Pros: Simple and effective, especially for high-dimensional data.
Cons: Can't estimate the score of clean data except as $\sigma\to0$, but the variance diverges as $\sigma\to0$.

Multi-scale Noise Perturbation

Large σ: worse data quality, but better score estimation. Small σ: better data quality, but inaccurate score estimation. Idea: train SBM on multiple noise scales (geometric sequence) $\sigma_1>\sigma_2>\cdots>\sigma_L$:

$$\mathcal{L}(\theta)=\frac{1}{L}\sum_{i=1}^{L}\sigma_i^2\mathbb{E}_{x\sim p_{\text{data}},z\sim\mathcal{N}(0,I)}\left[\left|s_\theta(x+\sigma_iz,\sigma_i)+\frac{z}{\sigma_i}\right|_2^2\right]$$

This is called the **Noise Conditional Score Network (NCSN)**.

Sampling Method: Annealed Langevin Dynamics

Langevin dynamics can mix slowly between modes. Annealed Langevin dynamics uses a sequence of noise scales, starting from large noise and gradually reducing it, to improve mixing and sample quality.

Diffusion Models: Motivations

NCSN trains with the loss:

$$\mathcal{L}(\theta)=\frac{1}{L}\sum_{i=1}^{L}\mathbb{E}_{x\sim p_{\text{data}},z\sim\mathcal{N}(0,I)}\left[\left|\sigma_is_\theta(x+\sigma_iz,\sigma_i)+z\right|_2^2\right]$$

Given a noisy image $x+\sigma_iz$, the model predicts the noise z. Training stability can be enhanced by changing the noise schedule. Sampling:

$$\tilde{x}_{t-1}=\tilde{x}_t-\frac{\epsilon}{2}\sigma_i\epsilon_\theta(\tilde{x}_t,\sigma_i)+\sqrt{\epsilon}\sigma_iz_t$$

Diffusion Models

DDPM (Denoising Diffusion Probabilistic Models) use a denoising loss:

$$\mathcal{L}(\theta)=\mathbb{E}_{x\sim p_{\text{data}},z\sim\mathcal{N}(0,I),t\sim\text{Uniform}(\{1,\cdots,T\})}\left[\|z-\epsilon_\theta(\sqrt{\bar{\alpha}_t}x+\sqrt{1-\bar{\alpha}_t}z,t)\|^2\right]$$

The reverse process is modeled by $p_\theta(x_{t-1}|x_t)=\mathcal{N}(x_{t-1};\mu_\theta(x_{t-1}),\sigma_t^2I)$. Sampling:

$$x_{t-1}=\frac{1}{\sqrt{\alpha_t}}\left(x_t-\frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(x_t,t)\right)+\sigma_tz$$

DDPM only changes the noise/sample schedule of NCSN; both intrinsically predict the same thing.

DDPM: Story

Forward process (pre-defined): gradually add noise,

$$q(x_t|x_0)=\mathcal{N}(\sqrt{\bar{\alpha}_t}x_0,(1-\bar{\alpha}_t)I)$$

Backward process (need to learn): denoise from x_t to x_{t-1}, parameterized by $p_\theta(x_{t-1}|x_t):=\mathcal{N}(\mu_\theta(x_t,t),\sigma_t^2I)$, where σ_t is calculated using α.

DDPM: Math

Key points:

- Loss formulation:

$$\mathcal{L}=\sum_{t=1}^{T}D_{\text{KL}}(q(x_{t-1}|x_t,x_0)\|p_\theta(x_{t-1}|x_t))$$

- Diffusion model learns the posterior average velocity:

$$\epsilon(x_t,t)=\mathbb{E}_{x_0\sim q(\cdot|x_t)}\frac{x_t-\sqrt{\bar{\alpha}_t}x_0}{\sqrt{1-\bar{\alpha}_t}}$$

- Epsilon-prediction is equivalent to the score function:

$$\epsilon(x_t,t)=-\sqrt{1-\bar{\alpha}_t}s(x_t,t)$$

SDE/ODE Models

We can use a continuous version of DDPM with infinitely many noise scales:

$$\mathcal{L}(\theta)=\mathbb{E}_{x\sim p_t,t\sim U(0,T)}\left[\lambda(t)\,|\nabla_x\log p_t(x)-s_\theta(x,t)|_2^2\right]$$

The sample procedure is given by the reverse SDE:

$$dx=-\sigma_t^2s_\theta(x,t)dt+\sigma_td\bar{w}$$

which can be solved by numerical SDE solvers.

Latent Diffusion Models

Diffusion models can be trained on the latent space encoded by a pretrained VAE. This is the technical foundation of models like Stable Diffusion and DALL-E.

3.3 Normalizing Flow (NF)

NF: Basic Idea

- A general generation formulation:
  - Given z, sample x from $p(x|z)$
  - Compute $p(x)=\int p(z)p(x|z)dz$, often **impossible**
  - Estimate $p(x)$ using ELBO (see VAE as an example later)
- We can directly apply a deterministic process $f:z\to x$, i.e.

$$z\sim\mathcal{N}(0,I),\quad x=f(z)$$

- If $p(x)$ is tractable, then we are done!

NF: Math

Theorem: Given a bijection $f(z):\mathbb{R}^d\to\mathbb{R}^d$, probability of $x=f(z)$

$$p(x)=p(z)\left|\det\frac{\partial f(z)}{\partial z}\right|^{-1}$$

If $f=f_K\circ\cdots\circ f_2\circ f_1$, each f_i is invertible, then

$$\log p(x)=\log p(z_0)-\sum_i\log\left|\det\frac{\partial f_i(z_{i-1})}{\partial z_{i-1}}\right|$$

Notice: $\det\frac{\partial f(z)}{\partial z}$ requires $O(d^3)$ computation

Ways to compute Jacobian

Goal: find f such that computing $\det\frac{\partial f(z)}{\partial z}$ is cheap — f with triangular Jacobian

- NICE (Nonlinear Independent Components Estimation):

$$x_{1:m}=z_{1:m},\quad x_{m+1:d}=z_{m+1:d}-\mu_\theta(z_{1:m}),\quad \det J=1$$

- Real-NVP

$$x_{1:m}=z_{1:m},\quad x_{m+1:d}=(z_{m+1:d}-\mu_\theta(z_{1:m}))\exp(-\alpha_\theta(z_{1:m}))$$
$$\det J=\prod_{i=m+1}^{d}\exp(-\alpha_\theta(z_{1:m})_i)$$

- GLOW (Generative Flow with Invertible 1x1 Convolutions): $x,z\in\mathbb{R}^{h\times w\times c},W\in\mathbb{R}^{c\times c}$

$$x_{ij}=Wz_{ij},\quad \log\det J=hw\log\det W$$

AF & IAF

Autoregressive Flow (AF):

- Forward: fast train $x\to z$,

$$z_i=(x_i-\mu(x_{<i}))\odot\exp(-\alpha(x_{<i}))$$

- Reverse: slow sample $z\to x$,

$$x_i=z_i\odot\exp(\alpha(x_{<i}))+\mu(x_{<i})$$

Inverse Autoregressive Flow (IAF):

- Forward: fast sample $z\to x$,

$$x_i=(z_i-\mu(z_{<i}))\odot\exp(-\alpha(z_{<i}))$$

- Reverse: slow train $x\to z$,

$$z_i=x_i\odot\exp(\alpha(z_{<i}))+\mu(z_{<i})$$

Parallel WaveNet

Inherit the good parts from both AF and IAF

- Teacher: $p_T(x_i|x_{<i},z_i)$ implemented by AF
- Student: $p_S(x_i|z_{<i})$ implemented by IAF

Train procedure:

- Train teacher network p_T (fast training for AF)
- Train student network by minimizing $KL(p_S\|p_T)$
- Use p_S for parallel sampling

NF: Practices

- T/F: In normalizing flow, the term $\log p(z_0)$ can be ignored in training since it is a constant. False. $z_0=f(x_0)$ is not a constant, so does $\log p(z_0)$.
- T/F: Since we need bijection in normalizing flows, we can't use ReLU as activation function in the model. False. Our model is guaranteed to be invertible under previous formulations, no matter how we design μ_θ and α_θ.
- T/F: The likelihood is tractable in both normalizing flow and diffusion models. False. Likelihood of diffusion models is not tractable.

3.4 Variational Autoencoder (VAE) & Vector-Quantized VAE (VQVAE)

VAE: Model and Training

- Two models: **encoder** $q_\phi(z|x)$ and **decoder** $p_\theta(x|z)$.
- The encoder maps an image x to a **latent** z distribution, and the decoder maps a latent z to an image distribution.
- Regularization (KL-divergence) loss:

$$\mathcal{L}_{\text{KL}}=D_{\text{KL}}(q_\phi(z|x)\|p(z)),$$

where the **prior** $p(z)$ is typically $\mathcal{N}(0,I)$.

- Reconstruction loss:

$$\mathcal{L}_{\text{recon}}=-\mathbb{E}_{z\sim q_\phi(z|x)}\log p_\theta(x|z).$$

- Loss function $\mathcal{L}=\mathcal{L}_{\text{recon}}+\mathcal{L}_{\text{KL}}$. $-\mathcal{L}$ is also called **ELBO (evidence lower bound)**.

VAE: Parameterization

The most common choice for $q_\phi(z|x)$ and $p_\theta(x|z)$: **diagonal Gaussian**.

- $q_\phi(z|x)\sim\mathcal{N}(\mu_\phi(x),\text{diag}(\sigma_\phi(x)))$. μ_ϕ,σ_ϕ are neural networks with dimension same as x.
- $p_\theta(x|z)\sim\mathcal{N}(\mu_\theta(z),I)$.

Then $\mathcal{L}_{\text{KL}}$ is the KL divergence of two Gaussians. Closed-form formula:

$$D_{\text{KL}}(\mathcal{N}_0\|\mathcal{N}_1)=\frac{1}{2}\left(\text{tr}(\Sigma_1^{-1}\Sigma_0)+(\mu_1-\mu_0)^T\Sigma_1^{-1}(\mu_1-\mu_0)-d+\log\frac{\det\Sigma_1}{\det\Sigma_0}\right).$$

CAVEAT: We should learn σ_ϕ instead of making $x\to z$ deterministic, although this hurts the performance. This is because the regularization loss needs that. However, we do not care about the variance in p_θ, because we can directly take $\mu_\theta(z)$ during sampling.

VAE: "Reparameterization Trick"

$$\mathcal{L}_{\text{recon}}=-\mathbb{E}_{z\sim q_\phi(z|x)}\log p_\theta(x|z).$$

Sampling does not have gradient information. Solution: $z\sim q_\phi(z|x)$ is equivalent to $z=\mu_\phi(x)+\epsilon\cdot\sigma_\phi(x)$, $\epsilon\sim\mathcal{N}(0,I)$, so

$$\mathcal{L}_{\text{recon}}=-\mathbb{E}_{\epsilon\sim\mathcal{N}(0,I)}\log p_\theta(x|\mu_\phi(x)+\epsilon\cdot\sigma_\phi(x)).$$

VAE: β-VAE

$$\mathcal{L}=\mathcal{L}_{\text{recon}}+\beta\cdot\mathcal{L}_{\text{KL}},\quad \beta>0.$$

- Smaller β ($\beta<1$): $\mathcal{L}_{\text{recon}}$ dominates $\to$ better reconstruction accuracy, but worse regularization.
- Larger β ($\beta>1$): $\mathcal{L}_{\text{KL}}$ is more important $\to$ the latent distribution is more regular, but worse reconstruction.

VAE: Applications

- **Generation**: only use the decoder. Sample $z\sim\mathcal{N}(0,I)$, use p_θ to sample an image x according to this z. From here we see why regularization loss is necessary: we must ensure that the latent distribution (determined by the encoder) is close to $\mathcal{N}(0,I)$, otherwise this sampling procedure is nonsense!
- **Representation**: Since the latent size is smaller, a small reconstruction loss will give a great compression of the original image. Usually β is EXTREMELY SMALL in this case.

VAE: Math

We want a model p_θ to take $z\sim\mathcal{N}(0,I)$ as input and output an image x. We want to maximize log-likelihood $\mathbb{E}_{x\sim\mathcal{D}}\log p_\theta(x)$. Using Bayes rule,

$$\log p_\theta(x)=\log\frac{p(z)p_\theta(x|z)}{p_\theta(z|x)}=\log p_\theta(x|z)+\log\frac{p(z)}{q(z)}-\log\frac{p_\theta(z|x)}{q(z)}.$$

Thus,

$$\log p_\theta(x)=\int_z q(z)\log p_\theta(x)dz$$
$$=\mathbb{E}_{z\sim q(z)}\log p_\theta(x|z)-D_{\text{KL}}(q(z)\|p(z))+D_{\text{KL}}(q(z)\|p(z|x))$$
$$\ge\mathbb{E}_{z\sim q(z)}\log p_\theta(x|z)-D_{\text{KL}}(q(z)\|p(z))=\text{ELBO}.$$

The difference is $D_{\text{KL}}(q(z|x)\|p_\theta(z|x))$. Alternatively,

$$\log p_\theta(x)-D_{\text{KL}}(q(z|x)\|p_\theta(z|x)).$$

Fix p and optimize q, we get $q(z|x)=p(z|x)$. So the maximization of ELBO w.r.t. p and q implies the maximization of NLL.

Conditioned VAE

To generate image x using a label y: **add y to the latent**. So: encoder $q_\phi(z,y|x)$ and decoder $p_\theta(x|z,y)$. Still train with ELBO:

$$\mathbb{E}_{z,y\sim q_\phi(z,y|x)}\log p_\theta(x|z,y)-D_{\text{KL}}(q_\phi(z,y|x)\|p(z,y)).$$

- Let $q(z,y|x)=q(z|x)\cdot q(y|x)$ (z,y are independent), and $p(z,y)=p(z)\cdot p(y)$, then

$$D_{\text{KL}}(q_\phi(z,y|x)\|p(z,y))=D_{\text{KL}}(q_\phi(z|x)\|p(z))+D_{\text{KL}}(q_\phi(y|x)\|p(y)),$$

which is tractable.

- Reconstruction loss:

$$\mathcal{L}_{\text{recon}} = -\mathbb{E}_{\epsilon\sim\mathcal{N}(0,I),y\sim q_\phi(y|x)}[\log p_\theta(x|\mu(x)+\epsilon\cdot\sigma_\phi(x),y)]$$

Backpropagation through $\mathbb{E}_{y\sim q_\phi(y|x)}$ via expanding expectation

VAE: Practices

- T/F: Optimizing ELBO w.r.t. p, q will implicitly maximize NLL. Thus, assuming infinite power of neural networks, if we train a VAE, the margin of NLL and ELBO will finally converge to 0. False. $q_\phi(z|x) = p_\theta(z|x)$ will never hold since $q_\phi(z|x)$ is restricted to be a Gaussian, but $p_\theta(z|x)$ is not.
- T/F: We can still directly use VAE to generate when training with $\beta = 0$. False. The latent distribution is unknown: we don't know how to sample z!($\beta = 0$ is standard AE)
- T/F: If we train a VAE to reconstruct a randomly masked image, this model can do impainting. True.
- T/F: Intuitively, large β in β-VAE makes the latent space disentangled. True.
- T/F: The reconstruction loss of VAE is commonly the ℓ_2 loss between the input image and the predicted image. True, using the formulation we have introduced.

Discrete Latent VAE: Gumbel-softmax trick

Method for discrete latent, not widely used today. Want the latent distribution $p(z) = \text{Uniform}(1, 2, \cdots, K)^L$. The regularization loss is naturally $D_{\text{KL}}(q(z|x)\|p(z))$;

$$\mathcal{L}_{\text{recon}} = -\mathbb{E}_{z\sim q_\phi(z|x)} \log p_\theta(x|z),$$

where $q_\phi(z_i|x) = \text{softmax}(l_{\phi,i}(x))$. For the backpropagation of reconstruction loss, we need to reparameterize the "sampling" from a discrete distribution $q(z|x)$.

Discrete Latent VAE: Gumbel-softmax trick

Theorem: Define $Q(0, 1)$ as the probability distribution $p(x) = \exp\{-x - \exp\{-x\}\}$. Sample i.i.d. $\epsilon_1, \cdots, \epsilon_n \sim Q(0, 1)$. Given $x_1, \cdots, x_n \in \mathbb{R}$, define a random variable

$$k = \arg\max_{1\le i\le n} \{x_i + \epsilon_i\}.$$

Then, $p(k) = \text{softmax}(\{x_i\}_{1\le i\le n})[k], \quad 1 \le k \le n$.
Sample $\epsilon_1, \cdots, \epsilon_K \sim Q(0, 1)$, using the Theorem above

$$v = \lim_{\tau\to 0} \text{softmax}\left\{\frac{x_i+\epsilon_i}{\tau}\right\}_{1\le i\le K} = \text{one-hot}(\arg\max_i\{x_i + \epsilon_i\})$$

equivalent to sample from one-hot vector from $\text{softmax}(x_1, \cdots, x_K)$, hence

$$\mathcal{L}_{\text{recon}} = -\mathbb{E}_{z\sim q_\phi(z|x)} \log p_\theta(x|z)$$
$$\approx -\mathbb{E}_{\epsilon\sim Q(0,1)} \log p_\theta\left(x|\text{softmax}\left(\frac{l_\phi(x)+\epsilon}{\tau}\right)\right) \quad (\tau \ll 1)$$

Vector Quantized VAE (VQVAE): Motivation

- Idea: make the prior $p(z)$ **discrete**.
- Motivation: $p(z) \sim \mathcal{N}(0, I)$ is "unimodal"; but a categorical distribution $\{1, \cdots, K\}^L$ (L is the latent size) is naturally "multimodal".
- Specifically, given an image x, the encoder should output a discrete vector $v = (v_1, \cdots, v_L) \in \{1, 2, \cdots, K\}^L$. Then, using this discrete vector, we want a decoder to reconstruct x.

VQVAE: Encoder & Decoder

Using a normal encoder, we can encode x into a latent $z = (z_1, \cdots, z_L)$. However, z is not discrete; so we need to **quantize** it.

- KEY IDEA: learn a **codebook** $\mathbf{Z} = \{e_1, e_2, \cdots, e_K\}$.
- Each z_i is encoded into a discrete value $v_i \in \{1, 2, \cdots, N\}$ by finding the nearest neighbor:

$$v_i = \arg\min_{1\le j\le K} \|e_j - z_i\|.$$

Encoder: $x \to z\ (z_e) \to v$; $x \to z$ uses a neural network encoder; $z \to v$ uses **quantization** by a learned codebook. **Decoder**: in reverse order, $v \to z\ (z_q) \to x$. $z \to x$ part can be implemented by another NN; for $v \to z$, we can **look-up** the codebook, $z_i = e_{v_i}$.

VQVAE: Loss

We need to define the **reconstruction loss** and the **regularization loss**.

- **Reconstruction**: just the ℓ_2 between input and output.
- **Regularization**: minimize the quantization error, i.e., the ℓ_2 distance between z_e and z_q. In the original paper:

$$\mathcal{L}_{\text{reg}} = \|z_e - \text{sg}(z_q)\|^2 + \beta \cdot \|z_q - \text{sg}(z_e)\|^2.$$

Essentially, this β is **to separate the learning rate for z_e and z_q**.

VQVAE: "Reparameterization"

NOTICE: taking arg min and look-up are not differentiable! We should make the whole process differentiable, so that the gradient of the reconstruction loss can backpropagate. **"Straight-through" trick**: $z_q \leftarrow z_e + \text{sg}(z_q - z_e)$. Then, the gradient will flow back to z_e, bypassing the quantization step.

VQVAE: Sampling

To sample images, we first need to sample v, then use the decoder to turn v into z_q, and then x. NOTICE: We do not know the latent distribution $p(v)$! However, we can learn it. For instance, regard $p(v) = p(v_1, \cdots, v_L)$ in an autoregressive manner, then we can use a Transformer decoder to learn

$$p(v_t|v_{1:t-1}), \quad 1 \le t \le L.$$

VQVAE: Practices

- After training a VQ-VAE, can we sample the latent z from a uniform distribution and do generation? If not, what should we do? We cannot sample directly from a uniform distribution. We need the "second stage": train the distribution of z (v).
- T/F: If the latent has high dimension (e.g. 10^3), dictionary embedding is better than the Gumbel softmax trick. True.

3.5 Generative Adversarial Networks (GAN)

GAN: Motivation

All generative models share a core question: How to approximate $p(x)$ for training? Why is $p(x)$ necessary? A good sampler $G(\cdot)$ may be enough. The goal is: $z \sim \mathcal{N}(0, I)$, $G_\theta(z) \sim p_{\text{data}}$. The idea is to minimize $D(G_\theta(z), p_{\text{data}})$ with a likelihood-free metric D.

GAN: Motivation (cont'd)

Our problem: how to choose D and G? G and D should be differentiable. D should be a distance metric to measure "whether $G_\theta(z)$ is a good sample". The idea: use a neural classifier D_ϕ to measure whether "an image looks like an image".

GAN: Training

Two core questions:

- Given D_ϕ, how to train G_θ? Maximize the score given by the classifier:

$$\theta = \arg\max \mathbb{E}_{z\sim\mathcal{N}(0,I)} D_\phi(G_\theta(z))$$

- Given G_θ, how to train D_ϕ? Try to classify images from dataset and generated by G_θ:

$$\phi = \arg\max \mathbb{E}_{x\sim p_{\text{data}}} \log D_\phi(x) + \mathbb{E}_{z\sim\mathcal{N}(0,I)} \log(1 - D_\phi(G_\theta(z)))$$

GAN: Objective

The goal:

$$\min_\theta \max_\phi \mathbb{E}_{x\sim p_{\text{data}}} \log D_\phi(x) + \mathbb{E}_{z\sim\mathcal{N}(0,I)} \log(1 - D_\phi(G_\theta(z)))$$

The optimal discriminator:

$$D_\phi(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_G(x)}$$

Under the optimal discriminator, the goal becomes:

$$\min_\theta 2\,\text{JSD}(p_G, p_{\text{data}})$$

where JSD is the Jensen-Shannon divergence.

GAN: Evaluation

Since $p_G(x)$ is not tractable, likelihood-based evaluation is impossible. Common metrics are based on a classifier (e.g., Inception v3 pretrained on ImageNet):

- **IS (Inception Score)**: $IS = \exp(\mathbb{E}_{x\sim G}[KL(f(y|x)||p_f(y))]$, higher is better.
- **FID (Fréchet Inception Distance)**: $FID = \|\mu_p - \mu_G\|^2 + \text{tr}(\Sigma_p + \Sigma_G - 2\sqrt{\Sigma_p\Sigma_G})$, lower is better.

GAN: Problems

Two main issues:

- Mode collapse: generating a few samples can cheat the loss.
- Training instability: discriminator and generator may keep oscillating.

Solutions include DCGAN, improved training techniques, WGAN, and more extensions.

DCGAN

Tricks for stability:

- Fully convolutional network (no pooling / MLP).
- Batch normalization.
- Leaky ReLU activation.
- Small learning rate and momentum.

Improved GAN

Additional tricks:

- Match features of the final activation layer in D_ϕ.
- D_ϕ classifies a sample via batch information.
- Historical averaging: $\mathcal{L}(\theta) = \|\theta - \frac{1}{T}\sum_t \theta_t\|^2$.
- Change positive label to 0.9.
- Use a fixed, pre-selected batch for normalization.

WGAN

Change the distance metric from JSD to Wasserstein distance (more smooth):

$$W(p_G, p_{\text{data}}) = \sup_{|f|_L\le 1} \mathbb{E}_{x\sim p_{\text{data}}} f(x) - \mathbb{E}_{x\sim p_G} f(x)$$

The goal:

$$\min_\theta \max_{\phi,|D_\phi|_L\le 1} \mathbb{E}_{x\sim p_{\text{data}}} D_\phi(x) - \mathbb{E}_{z\sim\mathcal{N}(0,I)} D_\phi(G_\theta(z))$$

Improved WGAN adds a gradient penalty.

GAN: A Bit More

Improvements:

- BigGAN: Large model + tricks.
- GigaGAN: Larger model + tricks.
- R3GAN: New loss and architectures.

Extensions:

- Semi-Supervised Learning: assign fake label for data without label.
- Representation Learning: similar distribution for $(z, G(z))$ and $(E(x), x)$.
- Style transfer: sample z from another dataset (see CycleGAN for unpaired data).

GAN: Practices

- T/F: In GAN training, we should first train the discriminator to make it converge, then train the generator. False. We should train generator and discriminator jointly.
- T/F: We want to fit a multimodal distribution by normal distribution and using KL loss function, but this results in collapsing to one mode. Change the loss function to JSD can fix this problem. False. JSD also suffers from mode collapse issue, that's why we need WGAN.

4 Sequential Models

4.1 Recurrent Networks (RNN & LSTM) and Language Models

Language Models

seq2seq model: input: sequence $x = (x_1, x_2, \cdots)$; output: sequence $y = (y_1, y_2, \cdots)$.
In a language model, sequence x, y are natural language sentences.
Each x_i or y_i is a **token** but not a word.
There is often a finite set of possible tokens (of size N). Then, each $x_i, y_i \in \{1, 2, \cdots, N\}$.
We want to learn a N-dimensional probability distribution for each position y_i, say $p_{\theta,i}(y_i)$. The loss function is the NLL loss

$$\mathcal{L} = \sum_i -\log p_{\theta,i}(y_i).$$

A sequential **generative model** is commonly modeled in a **autoregressive** manner:

$$p_\theta(y_{1:L}|x) = p_\theta(y_1|x) \cdot p_\theta(y_2|x, y_1) \cdots p_\theta(y_L|x, y_{1:L-1}).$$

This is called "**next-token prediction**". In this way, the NLL loss is equivalent to $-\log p(y|x)$.
Model structures:

- RNN;
- LSTM;
- Transformer.

Other important concepts:

- Word embedding;
- Beam search.

Recurrent NN (RNN): Idea

Idea: maintain a "latent (hidden) state" h_i, combining information of $x_1, \cdots, x_i$. *Intuitively, h represents what your brain is thinking when you hear the first i words.*

- h_{i+1} is inherited from h_i, can be influenced by x_{i+1};
- y_i is directly computed using h_i.

RNN: Formulation

- Hidden state transition: $h_{i+1} = f_1(W_{hx}x_{i+1} + W_{hh}h_i + b_h)$;
- Output: $y_i = f_2(W_{yh}h_i + b_y)$.
- f_1, f_2 are activations. W, b are learnable parameters.

RNN: BPTT

Note that the parameters are used for L times in a sequence of length L. So, when we backpropagate the gradient, the gradient will accumulate L times. This is called "**backpropagation through time (BPTT)**".
PROBLEM: gradient vanishing & exploding (toy example: all layers are linear, then the gradient flow accumulates like $I \to W \to W^2 \to \cdots$)

- Solve exploding: gradient clipping; identity init; **truncated** BPTT (only consider the last few tokens).
- Solve vanishing: LSTM (introduce memory) / Transformer (enable pairwise interaction).

LSTM

KEY IDEA: maintain a "memory" c_t together with the latent h_t.

- **forget gate** $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \in [0, 1]$.
- **input gate** $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \in [0, 1]$.
- **new information** $\tilde{c}_t = \tanh(W_c[h_{t-1}, x_t] + B_c)$.
- **MEMORY TRANSITION:** $c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$ ($\odot$ is pairwise product). Intuition: keep some info at the last time step, strength controlled by the forget gate f; update some new info, controlled by the input gate i.
- **output gate** $o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$.
- **LATENT TRANSITION:** $h_t = o_t \odot \tanh(c_t)$.

LSTM: As a Language Model

LSTM can be used as a language model by predicting the next token in a sequence, using the hidden state and memory cell to capture long-term dependencies.

Seq2Seq

Two RNNs: f_{enc} and f_{dec}, encode process

$$X \to f_{enc} \to h \to f_{dec} \to Y$$

- f_{enc}: (possibly) bidirectional RNN, produce final hidden state h
- f_{dec}: autoregressive LM conditions on the hidden state h

Attention: produce h_t based on h_{t-1} and attention output h_{att}

$$h_{att} = \sum_i f_{enc}(X_i|X_{1:i-1}) \cdot P_{att}(X_i|h_{t-1})$$

Word Embedding

KEY IDEA: The input token $\{1, 2, \cdots, N\}$ has no semantic meaning. We can learn an **embedding** for each token, $x_1, \cdots, x_N$, which can show some semantics.
For instance, if word i and j has similar meaning, the cosine similarity of x_i and x_j,

$$D(x_i, x_j) = \cos\langle x_i, x_j\rangle = \frac{x_i^T x_j}{\|x_i\| \cdot \|x_j\|}.$$

should be large; if they are irrelevant, these two vectors may be almost orthogonal.

Word Embedding: Continuous Bag-of-Words (CBOW)

KEY IDEA: learn an embedding w together with a **context embedding** c for each word. For adjacent $2k + 1$ tokens $\mathbf{x} = (x_{-k} \cdots x_{-1}x_0x_1 \cdots x_k)$, we want the similarity of c_i ($1 \le |i| \le k$) and w_0 to be large. That is, the context information should align with the actual word.

$$\mathcal{L}(w, c) = \mathbb{E}_{(w,c)\in\mathcal{D}} \log p(w|c),$$

Where (w, c) is sampled from $(c_{-k} \cdots c_{-1}wc_1 \cdots c_k)$, and

$$p(w|c) = \prod_i p(w|c_i), \quad p(w|c_i) = \frac{e^{w^T c_i}}{\sum_{w'} e^{w'^T c_i}}.$$

We may estimate the gradient of the denominator using Monte-Carlo.

Word Embedding: Skip-Gram

KEY IDEA: In the inverse of CBOW, we regard the center word as the context: $(w_{-k} \cdots w_{-1}w_1 \cdots w_k)$. In this way,

$$p(w|c) = \prod_i p(w_i|c), \quad p(w_i|c) = \frac{e^{w_i^T c}}{\sum_{w'} e^{w'^T c}}.$$

- Skip-Gram performs better than CBOW, especially for rare words;
- Skip-Gram is a harder task, thereby harder to overfit.
- CBOW converges faster.

Language Model Sampling: Beam Search

After training a generative autoregressive model θ, we want to sample from $p_\theta(y_{1:})$. In theory, sampling y_i from $p_\theta(y_i|y_{1:i-1})$ yields the correct distribution. However, we may prefer the sample with the highest probability, i.e., $\arg\max_y p_\theta(y)$.
IDEA: for each t, keep "top-K candidates". When sampling $t + 1$, find top-K tokens for each candidate, getting K^2 samples; keep K samples with the highest probability, and repeat.

4.2 Transformers and LLMs

Transformer

Transformer block: Attention block + MLP.
Attention Block:

- $[q_i, k_i, v_i] = [W_q, W_k, W_v]x_i$;
- **Dot-product attention**:

$$o_i = (v_1, \cdots, v_L) \odot \text{softmax}(\text{masked}(q_i^T k_j/\sqrt{d}))_{1\le j\le L}$$

, where d is the dimension of q, k, v.
When we do autoregression, since position i cannot gain information from position $j > i$, so all terms $q_i^T k_j$ ($j > i$) are masked.

Modern architectures use **multi-head attention**: essentially, split q_i, k_i, v_i into h parts (w/ equal length), and do dot-product attention on each part, then concatenate the output vectors o.

Transformer: Details

- **Positional Embedding (PE)**: sinusoidal PE & learnable PE. Each position i has its own PE_i. This is essential for Transformer since the other part (attention & MLP) is *symmetric among different positions*.
- **Encoder & Decoder**: encoder is **bidirectional**, meaning no mask in dot-product attention; decoder is **causal / autoregressive**, meaning $q_i^T k_j$ for $j > i$ is masked.
- **Self-attention & Cross-attention**: in self-attention, q, k, v are derived from the same input x; however, in cross-attention, $q = W_q x$ and $[k, v] = [W_k, W_v]y$. We often say the output is *conditioned on y using cross-attention*.

Transformer: Pretrain

Idea: pretrain large transformers on a large dataset lead to good representations
Question: How to pretrain?

- Encoder: bi-directional
- Decoder: auto-regressive
- Both encoder and decoder: powerful LM

Transformer: Pretrain Encoder

BERT: Pre-training of Deep Bidirectional Transformers
Train:

- Choose 15% of word tokens
  - Mask 80% (replace by $[M]$)
  - Replace 10% with random tokens
  - 10% remain unchanged

  predict these chosen tokens
- Next sentence prediction (not introduced in class)

Success on many down-stream tasks

Transformer: Pretrain Decoder

Most famous & successful pretraining work!

- GPT (117M): Improving Language Understanding by Generative Pre-Training
- GPT-2 (1.5B): Language Models are Unsupervised Multitask Learners
- GPT-3 (175B): Language Models are Few-shot Learners
- ImageGPT, ChatGPT, GPT-4...

Scaling Law: larger model / dataset / computation, higher performance Finetune & Zero-shot (prompting) & Few-shot (in-context learning)

Transformer: Pretrain Encoder-Decoder

- T5 (Text-to-Text Transfer Transformer):
  - Multi-task learning
  - Generalize to general QA
- FLAN-T5:
  - Finetune T5
  - Large-scale Training

Mixture-of-Expert

More efficient architecture with better scaling law

- Expert: small FFNs
- Router: select which experts to process
- Only a few parameters are activated each time

Finetune: SFT

In most case, we want an instruction following specialist, instead of a general next token prediction model

- Q: How to finetune (post-train) our pretrained model?
- A: Most natural idea: SFT
- A popular method: LoRA ($W + AB$)
- However, often cause hallucination

Finetune: RLHF

RLHF solves hallucination by aligning the LLM to human preferences

- Collect demonstration and train a supervised policy
- Collect human preference and train a reward model
- Optimize policy against reward model via RL

LLMs: A Bit More

Other modern ways to avoid hallucination and improve performance

- RAG: retrieval before generating
- Inference-time scaling: think longer, e.g. OpenAI o1
- CoT: "Let's think step by step."
- Post-time Scaling: Longer RL training

Transformers and LLMs: Practices

- T/F: Flamingo, gpt-4v, llava are LLMs that support interleaved images in texts. True. They are all modern multi-modal VLMs.
- T/F: LoRA makes it possible to do fine-tuning with limited GPU memory. True. LoRA is a parameter-efficient fine-tuning technique that significantly lowers GPU memory requirements.
- Suppose you have a pretrained BERT model, and say the probability distribution of it is p. You are given a sentence with middle part missing, say $x_1x_2 \cdots x_l???x_r \cdots x_n$, and you want to reconstruct it. The vanilla approach is just set all the middle words to be the mask token and sample from the model. However, this might be problematic. Explain why. The logits generated by BERT is decoupled among tokens. Directly sample each token does not yield the correct joint distribution.

5 Other Topics: Graph Learning, Representation Learning, and Meta-Learning

Graph Learning: Basics

- Practical problem: Given a partially labeled graph, how to predict labels for unknown nodes?
- Homophily assumption: u and v are "similar" if they are connected.
- Goal: learn embedding z for each node, such that similar nodes have similar embeddings.
- Application: AlphaFold series.

Graph Convolutional Networks (GCN): Definition

Graph convolution aggregates information from neighbors h_i as follows:

$$h_i^l = \sigma\left(W\sum_{j\in N(i)} \frac{h_j^{l-1}}{|N(i)|} + Bh_i^{l-1}\right)$$

In matrix form, with $D = \text{diag}(|N(i)|)$ and A as the adjacency matrix:

$$H^l = GC(G, H^{l-1}) = \sigma(D^{-1}AH^{l-1}W^T + H^{l-1}B^T)$$

GCN: Tips

- Shared neighbors quickly grow with l.
- Adding more GC layers may cause over-smoothing.
- Tips:
  - Increase the capability within each GC layer.
  - Add pre/post processing layers.
  - Add virtual edges ($A \to A + A^2$) or nodes.
  - Residual connection: $H^l = GC(G, H^{l-1}) + H^{l-1}$.

GCN: Improvement

- Vanilla GCN: sum over neighbors,

$$h_i^l = \sigma\left(W\sum_{j\in N(i)} \frac{h_j^{l-1}}{|N(i)|} + Bh_i^{l-1}\right)$$

- GraphSage: AGG as aggregation operation like MLP + Max-Pooling,

$$h_i^l = \sigma\left(W^T\left[h_i^{l-1}, AGG\left(\{h_j^{l-1}|j \in N(i)\}\right)\right]\right)$$

- Graph Attention: ATT as attention over neighbors,

$$h_i^l = \sigma\left(W^T\left[ATT\left(h_i^{l-1}, \{h_j^{l-1}|j \in N(i)\}\right)\right]\right)$$

- Graph Isomorphism Network: stronger representation power,

$$h_i^l = MLP^l\left(\left(1 + \epsilon^l\right)h_i^{l-1} + \sum_{j\in N(i)} h_j^{l-1}\right)$$

Representation Learning: Key Idea

- Learn a representation z for any data (e.g., image) x. Ideally, z should capture semantic information (e.g., similar scenes have close representations).
- For downstream tasks (e.g., image classification/segmentation), use z as input and a lightweight model (e.g., linear probing) to achieve the task.
- Method 1: Prediction-based self-supervised learning. Predict masked part; use the intermediate layer output as the representation.
- Method 2: Contrastive learning. Features of similar images should align, while features of irrelevant images should be almost orthogonal.

Representation Learning: Examples

- Context prediction: give two image patches, predict the spatial relation.
- Image colorization: give a gray image, predict the color.
- **CPC**: use context to predict future embeddings through contrastive learning
- **SimCLR**: predefine a set of data augmentations. Apply two random augmentations on a single image, the repr should align. (Note: need negative samples)
- **MAE**: random mask $\sim 70\%$ patches in an image, predict pixel values for missing patches.
- The most famous image repr today: **CLIP** (image & text repr alignment); **SigLIPv2**.

Representation Learning: MoCo

Momentum Contrastive Learning:

- Learn a query encoder f_{θ_q} and a key encoder f_{θ_k}.
- For each training step, sample a "similar" image pair (x, y_+) and negative samples $y_{-,1}, \cdots, y_{-,T}$.
- Let $q = f_{\theta_q}(x)$, $k = \text{sg}(f_{\theta_k}(y))$.
- Train θ_q using InfoNCE loss:

$$\mathcal{L}_q = -\log\frac{\exp(q \cdot k_+/\tau)}{\exp(q \cdot k_+/\tau) + \sum_i \exp(q \cdot k_{-,i}/\tau)}$$

- θ_k is updated using exponential moving average:

$$\theta_k = m\theta_k + (1 - m)\theta_q$$

World Model

World model: build model on video data by predicting actions

- VPT: Learning to Act by Watching Videos
- Latent World Models
- Genie: Generative Interactive Environments
- Dreamer: Mastering diverse control tasks through world models

Meta-Learning

- Goal: learn an "efficient learner" (few-shot learning). A meta-learner is formulated as $y = f_\theta(x|S_T)$, where $S_T = \{(x_i, y_i)\}$ contains few-shot samples for task $T \in \mathcal{D}$.
- General objective: B_T is the "test batch".

$$\theta^* = \arg\min_\theta \mathbb{E}_{T\in\mathcal{D},(x,y)\in B_T}[\text{Div}(f_\theta(x|S_T), y)]$$

- Method 1: learn a representation network ϕ_θ; in inference, find the label y according to the feature distance of x to few-shot samples x_i (e.g., nearest neighbor).
- Method 2: use sequential models and feed in S_T and x (e.g., RNN).
- Method 3: gradient descent the network parameters during inference (**MAML**).

Meta-Learning: MAML

We want θ to learn fast on any few-shot learning task. Suppose we have a meta-learning dataset $\mathcal{D}$, each $S \in \mathcal{D}$ includes several examples (x_i, y_i).
One gradient step: $\theta \to \theta - \eta \cdot \nabla_\theta\mathcal{L}(S; \theta)$. The objective is

$$\mathcal{L}_{\text{MAML}} = \mathbb{E}_{S\in\mathcal{D}}\,\mathcal{L}(S; \theta - \eta \cdot \nabla_\theta\mathcal{L}(S; \theta))$$

Inference: do gradient descent on the few-shot samples for a few steps, then use the updated θ to predict y from x.

6 Weakness of Deep Learning

Overview

Issues and weaknesses of deep learning models:

- Adversarial examples
- Security and privacy
- Explainability
- Fairness and bias
- Safety

Adversarial Example

Given a model $f_\theta(x)$, attacker: $\arg\max_{|\epsilon|\le c}|f_\theta(x + \epsilon) - f_\theta(x)| \propto \nabla f_\theta(x)$
How to defend against adversarial examples?

- White box (accessible gradient):

$$\mathcal{L}(\theta) = \mathcal{L}(f_\theta(x + \epsilon\nabla f_\theta(x)), y)$$

- Black box (inaccessible gradient): "steal" gradient by imitation learning.

Besides, many attacks to LLMs (backdoor attacks, "jailbreaks") and alignment for safety (rule-based rewards, reasoning).

Security and Privacy

Data leakage:

- Machine learning models remember too much.
- Training data can be extracted from diffusion models.
- Language models can remember sensitive information.

Temporary solution: Federated Learning

- Data stays local, only gradients are synchronized.
- However, data can be recovered from distributed gradients.

Explainability

Most deep learning algorithms are "black box".
How to understand what features contribute to the output of a neural network?

- Visualize attention maps.
- Search for the pattern that activates a neuron the most via back-propagation.
- Remember the difference between correlation and causation.

Practices

- T/F: There's no safety issue to use generative models as the things they generate are fake. False. Rumors are more ferocious than tigers.
- Suppose you trained a LLM and put its weights on the internet. Somebody downloads it and makes small modifications, claiming it's their model. How can you show the plagiarization? (Hint: small modification means adding small Gaussian noise to the weights, or doing neuron permutation that doesn't affect the result. Assume no further training.) (1) Watermarking: embed a unique, often imperceptible signal into the model's generated text. (2) Check weight/activation similarity (need to be agnostic to neuron permutation).