1 Neural Networks

Let $\phi(\cdot)$ be non-const, bounded & monotonically incr func. $\forall \epsilon > 0$ & any continuous func $f(\cdot)$ defined on a compact subset of $\mathbb{R}^m$, $\exists$ integer N, real constants $v_i, b_i \in \mathbb{R}$ & real vectors $w_i \in \mathbb{R}^m$ for $i = 1, 2, \ldots, N$ such that:

$$F(x) = \sum_{i=1}^{N} v_i \phi(w_i^T x + b_i) \quad \text{with} \quad |F(x) - F_N(x)| < \epsilon$$

If $\phi(\cdot)$ is a sensible **act func**, then ***any func*** can be approxed by NN with 1 hidden layer & enough neurons. In practice, ϵ is very large & we suffer from *curse of dim*.

1.1 Curse of Dimensionality

As # features/dims grows, data needed to generalise accurately grows exponentially.

> To approximate a **Lipschitz (continuous)** function $f : \mathbb{R}^d \to \mathbb{R}$ to accuracy ϵ requires $O(\epsilon^{-d})$ training samples.

The n-dimensional volume of an interior is α^n times the volume of its original shape, where α is the scaling factor. Therefore, the volume of the rind relative to the original volume is $1 - \alpha^n$. As a function of α, its rate of broth is $\mathrm{d}(1-\alpha^n) = -n\alpha^{n-1}\mathrm{d}\alpha$, where d is the differential operator.

- Beginning with no shrinking ($\alpha = 1$), and that α is decreasing ($\mathrm{d}\alpha < 0$), we see the initial rate of growth of the rind volume is $n\mathrm{d}\alpha$.
- Initially, the volume of the rind grows n times faster than rate at which the object is being shrunk.
- In higher dimensions, tiny changes in distance translate to large changes in volume.

> In higher dimensions n most Euclidean distances between observations in a dataset are nearly the same and close ($\mathcal{O}(\frac{1}{n})$) to the diameter of the region in which they are enclosed.

1.2 Invariance & Equivariance

Shift invariance: system's unchanging response to input shifts. *E.g., recognizing smth regardless of its pos in image.* $f : \mathbb{R}^n \to \mathbb{R}$ is shift invariant for **shift operator** $S_{\vec{v}} : \mathbb{R}^n \to \mathbb{R}^n$ (shifting image by $\vec{v}$) if $\forall x \in \mathbb{R}^n . f(S_{\vec{v}} x) = f(x)$.

Equivariance: change in input $\to$ same change in output. $f : \mathbb{R}^n \to \mathbb{R}^m$ is equivariant for shift operator $S_{\vec{v}} : \mathbb{R}^n \to \mathbb{R}^n$ if $\forall x \in \mathbb{R}^n . f(S_{\vec{v}} x) = S_{\vec{v}} f(x)$.

Deformation Invariance: **warp operator** $D_\tau : \mathbb{R}^n \to \mathbb{R}^n$ warps image by field τ. Then, $\forall x \in \mathbb{R}^n . f(D_\tau x) \approx f(x)$

1.3 Inductive Bias

1. **Translation Invariance**: a shift in the input should lead to a shift in the hidden representation.
2. **Locality**: shouldnt have to look far away from a location to glean relevant information about that area.

These can be applied with a sliding window approach, using correlation $C_{i,j} = \sum_{x=0}^{m-1} \sum_{y=0}^{m-1} \hat{I}_{i,j}(x,y) \cdot K(x,y)$ where $\hat{I}$ is the image patch and K is the kernel/filter.

1.4 Convolutions

For $f, g : [0, \infty) \to \mathbb{R}$:

- **Convolution**: $(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) g(t-\tau) \mathrm{d}\tau$.
- **Correlation**: $(f \star g)(t) = \int_{-\infty}^{\infty} f(\tau) g(t+\tau) \mathrm{d}\tau$.
- **Discrete Convolution**: given arrays u_t and w_t, their convolution is $s_t = \sum_{a=-\infty}^{\infty} u_a w_{t-a}$. When u_t or w_t are undefined they are assumed to be zero.

A convolution is *commutative*, *associative*, *associative with scalar multiplication* and *distributive*.

- **Stride** is the number of pixels by which we slide the filter across the input image.
- **Padding** is the number of pixels added to the border of the input image. *If you need to keep the X and Y dimensions then you can 0-pad the image.*

2 Convolutional Neural Networks

In CNNs, keep *locality* using a $[X, Y, Z] * [I, J, K]$ conv. Input has width X, height Y, depth Z (*channels*). Output (**activation map**) has width $X - I + 1$ & height $Y - J + 1$. *Convolutions are impl as a series of dot products.* Each kernel produces an activation map, these are stacked to produce output volume. With N kernels, output has depth N. *A CNN is a seq of conv layers interleaved with act funcs.*

Each filter has $I \cdot J \cdot K + 1$ params, including the **bias**. A 1×1 conv reduces network depth, aggregating many channels into one. Each convolution decreases X, Y dims, decreasing computational cost of each layer. *Exploitable by using many small filters to approx 1 large filter.*

2.1 Pooling Layer

Aggregates & downsamples input tensor. Pixel order doesnt matter. Introduces *translational invariance*. Some pooling methods are:

- **Max Pooling**: takes the maximum value in each patch. Max pooling can *break shift equivariance*. This can be partially solved by anti-aliasing (blurring) before downsampling.
- **Average Pooling**: takes average value in each patch.

This is applied to each channel separately.

2.2 Fully Connected Layer

A standard dense layer, where each neuron is connected to every neuron in the previous layer.

2.3 Flatten Layer

Connect conv layers to fully connected layers by flattening 3D tensor into a 1D vector.

2.4 Batch Normalization

Loss is calculated at the last layer, so last layer learns quickest. Data input is at 1st layer, so if 1st layer changes, last layer needs to relearn many times, causing slow convergence. This is **internal covariate shift**. To avoid changing last layers while learning 1st layers, fix mean & var:

$$\mu_B = \frac{1}{|B|} \sum_{i \in B} x_i \qquad \sigma_B^2 = \frac{1}{|B|} \sum_{i \in B} (x_i - \mu_B)^2 + \varepsilon$$

Then adjust it separately: $x_{i+1} = \gamma \frac{x_i - \mu_B}{\sigma_B} + \beta$.

Where B is the batch, and γ, β are learned parameters. This speeds up training & acts as a regularizer. To reduce covariate shift, **inject noise** to inputs during training: $x_i = \gamma \frac{x_i - \hat{\mu}_B}{\hat{\sigma}_B} + \beta$. This also removes the need for drouput. *Ideal minibatch size is 64-256.* In dense layer, 1 norm $\forall$ channels. In conv layer, 1 norm per channel. Mean & variance are computed $\forall$ minibatch.

2.5 Residual Networks

A **ResNet** ensures func space of each layer includes prev layer, avoiding gradient vanishing problems in deep networks. A **residual block** has a *skip connection* that outputs $y = F(x) + x$, where $F(x)$ is the output of the layer.

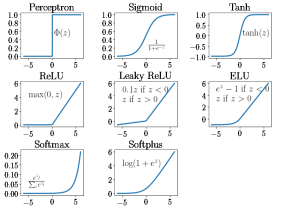
3 Activation Functions

Determine how neurons respond to inputs, *introducing non-linearity* into network. A neuron:

$$\text{output} = \sum (w_i \cdot \text{input}) + b_i$$

Linear act func $c \cdot x$ has const grad, so no relationship to x during backprop. $\therefore$ need non-linear act func, such as:

- **Sigmoid** $f(x) = \frac{1}{1+e^{-x}}$. Steepest at small x, meaning smaller inputs have a larger effect. However, it suffers from *vanishing gradients* for large $|x|$.
- **Tanh** $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ is a scaled sigmoid, outputting $[-1, 1]$. 0 centered, but suffers from *vanishing grads*.
- **ReLU** $f(x) = \max(0, x)$ efficent. Suffers from *dying ReLU*: neurons get stuck outputting 0.
- **Leaky ReLU** $f(x) = \max(ax, x)$ for small a (e.g. 0.01) mitigates the dying ReLU problem.
- **PReLU** $f(x) = \max(ax, x)$, a learned in training.
- **Softplus** $f(x) = \frac{1}{\beta} \ln(1 + e^{\beta x})$: a *smooth approximation* to ReLU. Output > 0. If $\beta \cdot x >$ thresh, then $f(x) \approx x$.
- **LogSigmoid** $f(x) = \ln(\frac{1}{1+e^{-x}})$, numerically stable for large negative x.
- **Softmin** $f(x_i) = \frac{e^{-x_i}}{\sum_j e^{-x_j}}$ operates on vectors, producing a probability distribution.
- **Softmax** $f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$ operates on vectors, producing a probability distribution.
- **LogSoftmax** $f(x_i) = \ln(\frac{e^{x_i}}{\sum_j e^{x_j}})$ is the logarithm of the softmax function, useful for numerical stability in classification tasks.



4 Loss Functions

Quantifies how well the network is doing:

- **L2 Norm (MSE)**: $\ell(x, y) = \|x - y\|_2^2 = \sum_i (x_i - y_i)^2$. Penalizes large errors more than small ones.
- **L1 Norm** $\ell(x, y) = \|x - y\|_1 = \sum_i |x_i - y_i|$. More robust to outliers than L2.
- **Smooth L1 Loss** $L(x, y) = \frac{1}{n} \sum_i z_i$, where $z_i = \begin{cases} 0.5(x_i - y_i)^2, & \text{if } |x_i - y_i| < 1 \\ |x_i - y_i| - 0.5, & \text{otherwise} \end{cases}$. Combines L1 and L2.
- **Negative Log Likelihood**: assume the network's output represents log likelihoods of each class. Then, $\ell(x, y) = \begin{cases} \sum_{n=1}^{N} \frac{l_n}{\sum_{n=1}^{N} w_{y_n}} & \text{if reduction = mean} \\ \sum_{n=1}^{N} l_n & \text{if reduction = sum} \end{cases}$, where $l_n = -w_{y_n} x_{n, y_n}$ and w_c is weight for class c.
- **Cross Entropy Loss** combines *LogSoftmax* and *NLLoss* as $L(x, \text{class}) = -\log(\frac{e^{x_{\text{class}}}}{\sum_j e^{x_j}}) = -x_{\text{class}} + \log(\sum_j e^{x_j})$. This is more numerically stable.
- **Binary Cross Entropy Loss** is CE loss for 2 classes, where $L(x, y) = -\frac{1}{n} \sum_{i=1}^{n} \left[y_i \log(x_i) + (1 - y_i) \log(1 - x_i) \right]$. Can be reduced with *mean* or *sum*. Requires $x_i \in (0, 1)$.
- **Marking Ranking Loss / Ranking Loss / Contrastive Loss**: predicts rel dist between inputs pairs rather than absolute class labels. $L(x, y) = \max(0, -y(x_1 - x_2) + \text{margin})$, where $y \in \{-1, 1\}$ indicates if x_1 should be ranked higher than x_2.
- **Triplet Margin Loss**: samples from the same classes close and different classes far away. Used for metric learning & Siamese networks: $L(x, y) = \{l_1, \cdots, l_N\}$, where $l_i = \max(0, m + \|x_i^a - x_i^p\|_2 - \|x_i^a - x_i^n\|_2)$, x_i^a is anchor, x_i^p is positive sample, x_i^n is negative sample.

The choice of a loss function depends on the desired output.

5 Input Augmentation

Artificially inflates training data size through applying expected transformations during training. *Good regularizer against overfitting.* Can do random *flipping*, *scaling*, *rotation*, *intensity / contrast adj*, *cropping & padding*, *noise*, *affine trans*, *persp trans*.

Anomaly Detection models identify unusual patterns that dont conform to expected behaviour. IA improves robustness by exposing model to wider variety of normal patterns during training. Can be:

- **Unsupervised** - use an autoencoder reconstruction error to identify anomalies.
- **Supervised** - RNNs learn from a labeled dataset of normal and anomalous sequences.

6 Generative Models

Supervised learning learns func $x \to y$ given dataset of IO pairs. **Unsupervised learning** learn patterns without explicit labels. **Generative models** generate new samples. Used for:

- *Probability density estimation:* learn **distribution** $p_\theta(x) \approx p_{\text{data}}(x)$ to generate new samples. **Latent Variable Models (LVMs)** describe sampling of observation x as 2-step process: *(1)* sample **latent var** z from a **prior** dist $p(z)$, *(2)* sample x from a **conditional dist** $p_\theta(x|z)$. Then $p_\theta(x) = \int p_\theta(x|z) p(z) dz$.
- *Dimensionality reduction:* learn a low-dim representation of (sparse) high dim data. **PCA** reduces dim by proj onto **principal component** dirs that capture most variance. **Probablistic PCA** models data as Gaussian with low-rank covariance structure. **Autoencoders** learn *encoder func* $f_\theta(x)$ that maps input to low-dim latent space, and *decoder func* $g_\theta(z)$ that reconstructs input from latent representation. Model trained to min reconstruction error.

7 Variational Auto Encoders

Divergence min: finding a dist p_θ thas close to target dist p_{data} by min a divergence measure $D[p_{\text{data}} \| p_\theta]$ ($\theta^* = \arg\min_\theta D[p_{\text{data}} \| p_\theta]$):

$$D[p\|q] = 0 \Leftrightarrow p = q \qquad p \neq q \Rightarrow D[p\|q] > 0$$

Kullback Leibler (KL) divergence $D_{KL}[p\|q] = \int p(x) \log \frac{p(x)}{q(x)} dx = E_{p(x)}[\log \frac{p(x)}{q(x)}]$. Not symmetric, i.e., $D_{KL}[p\|q] \neq D_{KL}[q\|p]$. To verify above conditions hold, use **Jensen's inequality** ($\log \mathbb{E}[X] \geq \mathbb{E}[\log X]$).

To use, simplify expr to $E_{p_{\text{data}}(x)}[\log p_{\text{data}}(x)] - E_{p_{\text{data}}(x)}[\log p_\theta(x)]$. 1st term constant wrt θ, so min KL div is equiv to max expected log-likelihood of data under model, i.e., $\theta^* = \arg\max_\theta E_{p_{\text{data}}(x)}[\log p_\theta(x)]$. In practice, this means computing $\theta^* = \arg\max_\theta \frac{1}{N} \sum_{i=1}^{N} \log p_\theta(x_i)$ where $x_1, \cdots, x_N \sim p_{\text{data}}$.

7.1 Fitting a LVM

MLE function above intractible, so optimise for a variational lower bound instead:

- $\log p_\theta(x) = \log \int p_\theta(x|z) p(z) dz$ *(see intro to LVMs)*.
- $\log p_\theta(x) = \log \int q_\phi(z|x) \frac{p_\theta(x|z) p(z)}{q_\phi(z|x)} dz$ *(introduce a variational distribution $q_\phi(z|x)$)*.
- $\log p_\theta(x) \geq \int q_\phi(z|x) \log \frac{p_\theta(x|z) p(z)}{q_\phi(z|x)} dz$ *(apply Jensen's inequality)*.
- $\log p_\theta(x) \geq E_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{\text{KL}}[q_\phi(z|x) \| p(z)] = L(x, \phi, \theta)$ *(rearrange terms)*.

The goal of a **VAE** is to find $\theta^*, \phi^* = \arg\max_{\theta,\phi} L(\theta, \phi)$, where $L(\theta, \phi) = E_{p_{\text{data}}(x)}[E_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{\text{KL}}[q_\phi(z|x) \| p(z)]]$. VAE trains gen model $p_\theta(x|z)$ & variational dist $q_\phi(z|x)$ simul to max variational lower bound on the log-likelihood of the data. *In English, VAE learns to encode data into a latent space & decode from that latent space back to the original data, while ensuring that latent space follows a specified prior dist.*

However, $E_{q_\phi(z|x)}[\log p_\theta(x|z)]$ is *intractable* as it requires computing $\forall$ possible z. Use **Monte Carlo estimation** to approximate this instead:

$$E_{q_\phi(z|x)}[\log p_\theta(x|z)] \approx \log p_\theta(x|z), z \sim q_\phi(z|x)$$

Differentiate this to obtain MC gradient w.r.t. θ, which allows learning. To learn w.r.t. ϕ, use **reparameterization trick**: express z as a deterministic func of ϕ & some noise ϵ that is independent of ϕ:

$$z \sim q_\phi(z|x) \Leftrightarrow z = \mu_\phi(x) + \sigma_\phi(x) \odot \epsilon, \epsilon \sim \mathcal{N}(0, I)$$

Hence, by letting $z = T_\phi(x, \epsilon) = \mu_\phi(x) + \sigma_\phi(x) \odot \epsilon$, we can rewrite the expectation as:

$$E_{q_\phi(z|x)}[\log p_\theta(x|z)] \approx \log p_\theta(x|T_\phi(x, \epsilon)), \epsilon \sim \mathcal{N}(0, I)$$

Now, we can differentiate this w.r.t. ϕ to learn it as well.

7.2 Designing the q Distribution

A **Factorized Gaussian** dist $q_\phi(z|x) = \mathcal{N}(z; \mu_\phi(x), \text{diag}(\sigma_\phi^2(x)))$: z follows dist with mean $\mu_\phi(x)$ & diag cov matrix $\text{diag}(\sigma_\phi^2(x))$. $\mu_\phi(x)$ & $\sigma_\phi(x)$ are parameterized by NN with params ϕ. Variance ensured ≥ 0. Using this, express the analytic form of the KL regularizer, with $p(z) = \mathcal{N}(z; 0, I)$, $z \in R^d$ and $\langle a, b \rangle$ is the dot product between two vectors:

$$D_{\text{KL}}[q_\phi(z|x) \| p(z)] = \frac{1}{2} \left(\|\mu_\phi(x)\|_2^2 + \|\sigma_\phi(x)\|_2^2 - d - 2\langle \log \sigma_\phi(x), 1 \rangle \right)$$

7.3 Designing the VAE

Combining the above, we want to find $\theta^*, \phi^* = \arg\max_{\theta,\phi} L(\theta, \phi)$, where:

$$L(\theta, \phi) \triangleq E_{p_{\text{data}}(x)}\left[-E_{\mathcal{N}(\epsilon; 0, I)}\left[\frac{1}{2\sigma^2} \left\| G_\theta(T_\phi(x, \epsilon)) - x \right\|_2^2 \right] - D_{\text{KL}}[q_\phi(z|x) \| p(z)] \right] = D_{KL}[p_{\text{data}}(x) \| \tilde{p}(x)] + D_{KL}[p_\theta(x) \| \tilde{p}(x)] - 2\log 2$$

where $\tilde{p}(x) = \frac{p_{\text{data}}(x) + p_\theta(x)}{2}$

- $T_\phi(x, \epsilon) = \mu_\phi(x) + \sigma_\phi(x) \odot \epsilon$ is the reparam. of z.
- $G_\theta(z)$: *decoder func* mapping latent variable z to the data space, parameterized by θ.
- $G_\theta(T_\phi(x, \epsilon))$: **stochastic autoencoder** that reconstructs the input x from the latent representation z.
- $E_{\mathcal{N}(\epsilon; 0, I)}[\cdots]$: **reconstruction loss** measuring how well model reconstructs input from latent repr.
- $-D_{\text{KL}}[q_\phi(z|x) \| p(z)]$: **KL regularization term** encouraging learned latent dist to be closer to prior dist, preventing $\sigma_\phi(x)$ from collapsing to 0.
- ϵ is a **noise variable** that allows for stochastic sampling of the latent variable z during training.

Once trained, generate new sample images from model by sampling $z \sim p(z)$ & then computing $G_\theta(z)$. Different dims of z encode diff features of the generated image.

7.4 Training

To compute $\arg\max_{\theta,\phi} L(\theta, \phi)$: initialize θ & ϕ randomly. Set learning rate γ & total T iters for **stochastic grad descent**. Then, for $t = 1, \cdots, T$, ENCODE (*perform approximate posterior inference*) then DECODE (*reconstruct the data*):

1. Sample **mini-batch** (size M): $x_1, \cdots, x_M \sim p_{\text{data}}(x)$.
2. ENCODE: Compute $\mu_\phi(x_m)$ & $\sigma_\phi(x_m)$ for $m = 1, \cdots, M$. These are params of variational dist $q_\phi(z|x)$ for each data point in the mini-batch.
3. ENCODE: Apply reparam trick: $z_m = \mu_\phi(x_m) + \sigma_\phi(x_m) \odot \epsilon_m$ where $\epsilon_m \sim \mathcal{N}(0, I)$ for each $m = 1, \cdots, M$. Allows sampling from variational dist in a way that is differetiable wrt ϕ.
4. DECODE: Find $\hat{x}_m = G_\theta(z_m)$ for each $m = 1, \cdots, M$. This is the reconstruction of the input data point x_m from the latent representation z_m.
5. Finally, update network params. Compute **variational lower bound** as $L = \frac{1}{M} \sum_{m=1}^{M} \left[-\frac{1}{2\sigma^2} \|x_m - \hat{x}_m\|_2^2 - D_{\text{KL}}(q_\phi(z_m|x_m) \| p(z_m)) \right]$. Then, update network parameters as $(\theta, \phi) \leftarrow (\theta, \phi) + \gamma \nabla_{\theta,\phi} L$.

> In the variational lower bound, $D_{\text{KL}}[q_\phi(z_m|x_m) \| p(z_m)]$ has an **analytic form** if both q and the prior $p(z)$ are Gaussian. If there is no analytic form, we can use **Monte Carlo estimation** to approximate it.

8 Generative Adverserial Networks

Goal: fit **prob model** p_θ to **underlying dist** p_{data} s.t. $p_\theta(x) \approx p_{\text{data}}(x)$. Want to **min divergence measure**: $\theta^* = \arg\min_\theta D[p_{\text{data}} \| p_\theta]$.

A **GAN)** samples **latent variable** z from **prior** $p(z)$, then passes it through **generator** (*neural network*) G_θ to produce **sample** $x = G_\theta(z) \sim p_\theta(x)$. Now, a **discriminator** (*neural network*) D_ϕ takes in sample x & underlying dist p_{data}, to produce binary value (x real/fake). Objective func is a **minmax** on ϕ & θ:

$$\min_\theta \max_\phi L(\theta, \phi) = E_{p_{\text{data}}(x)}[\log D_\phi(x)] + E_{p_\theta(x)}[\log(1 - D_\phi(x))]$$

where $D_\phi(x) = P(x \text{ is real})$, $1 - D_\phi(x) = P(x \text{ is fake})$

Fixing θ & training D_ϕ is a **supervised learning problem**, where we max **negative cross entropy** loss between discriminator's predictions & labels. Fixing ϕ & training G_θ is a **reinforcement learning problem**, where we min **-ve log prob** of discriminator being correct, i.e., max $\log D_\phi(G_\theta(z))$.

When fixing θ, the optimal discriminator is $\phi^* = \max_\phi L(\theta, \phi)$, which satisfies $D_{\phi^*}(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_\theta(x)}$. Substituting back into objective func, we get:

$$L(\theta, \phi^*(\theta)) = E_{p_{\text{data}}(x)}\left[\log \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_\theta(x)} \right] + E_{p_\theta(x)}\left[\log \frac{p_\theta(x)}{p_{\text{data}}(x) + p_\theta(x)} \right]$$

> This is valid divergence measure, **Jensen-Shannon divergence**. Hence, min $L(\theta, \phi^*(\theta))$ is equiv to min $D_{JS}[p_{\text{data}} \| p_\theta]$.

8.1 Training

Training a GAN uses a *double loop* algorithm:

1. **Inner Loop**: fixed θ, optimise ϕ for a few gradient ascent iters: $\max_\phi E_{p_{\text{data}}(x)}[\log D_\phi(x)] + E_{p_\theta(x)}[\log(1 - D_\phi(x))]$.
2. **Outer Loop**: fixed ϕ, optimise θ for **JUST ONE** gradient descent iter: $\min_\theta E_{p_\theta(x)}[\log(1 - D_\phi(x))]$

This is repeated until convergence. Num inner loop iters & learning rates are hyperparams. Computing $E_{p \cdots}[\cdot]$ is intractable, so use **Monte Carlo estimation** to approx by sampling **minibatches** from the respective dists:

- $E_{p_{\text{data}}(x)}[\log D_\phi(x)] \approx \frac{1}{M} \sum_{m=1}^{M} \log D_\phi(x_m)$ where $x_m \sim p_{\text{data}}(x)$.

- $E_{p_\theta(x)}[\log(1-D_\phi(x))] \approx \frac{1}{M}\sum_{m=1}^{M}\log(1-D_\phi(G_\theta(z_m)))$ where $z_m \sim p(z)$.

The full algorithm is:

1. Initialize θ, ϕ; learning rates γ_D, γ_G, num of inner & outer loop iters T, K.
2. Repeat steps 3-8 for $t = 1, \cdots, T$:
3. [GENERATOR] Repeat 4-5 for $k = 1, \cdots, K$:
4. Sample minibatches $z_1, \cdots, z_M \sim p(z)$ and $x_1, \cdots, x_M \sim p_{\text{data}}(x)$.
5. $\phi \leftarrow \phi + \gamma_D \nabla_\phi [\frac{1}{M}\sum_{m=1}^{M}\log D_\phi(x_m) + \frac{1}{M}\sum_{m=1}^{M}\log(1-D_\phi(G_\theta(z_m)))]$.
6. [DISCRIM] Sample minibatch $z_1, \cdots, z_M \sim p(z)$.
7. [DISCRIM] $\forall j \in \{1, \cdots, J\}$. $\left[\tilde{x}_j = G_\theta(z_j)\right]$.
8. [DISCRIM] $\theta \leftarrow \theta - \gamma_G \nabla_\theta \left[\frac{1}{J}\sum_{j=1}^{J}\log(1-D_\phi(\tilde{x}_j))\right]$.

8.2 Non Saturate Loss

Generator maxxes prob of samples classified as real by discrim. Instead of minimising $\log(1 - D_\phi(G_\theta(z)))$, use **non-saturating loss**:

$$\theta^* = \min_\theta -E_{p_\theta(x)}[\log D_\phi(x)]$$

8.3 Conditional LVMs

To construct **conditional LVMs**, specify cond dist $p_\theta(x|y)$ where y is some observed var. *e.g., generating input with specific label.*

Goal is to learn gen model $p_\theta(x|y)$ where x is data to generate, y is **label** that gen process is conditioned on. Make (z, y) input for network: $p_\theta(x|y=c) = \int p_\theta(x|z, y=c)p(z)dz$. A **conditional VAE** is *parameter efficient, works on continuous y*. To train on data $\{(x_n, y_n)\}_{n=1}^N \sim p_{\text{data}}(x, y)$, use **conditional variational lower bound**, derived from MLE:

$$\max_\theta E_{p_{\text{data}}(x,y)}[\log p_\theta(x|y)]$$

$$\geq \text{minimize } E_{p_{\text{data}}(x,y)}\Big[E_{q_\phi(z|x,y)}[\log p_\theta(x|z,y)]$$

$$-D_{\text{KL}}[q_\phi(z|x,y)\|p(z)]\Big] \text{ wrt } \theta, \phi$$

Now, the encoder must now take $q_\phi(z|x, y)$ as input. The decoder must take $p_\theta(x|z, y)$ as input.

9 Diffusion Models

To make **latent dist** q in a VAE more flexible, use **heirarchical LVMs**. Instead of using 1 z, use a heirarchy of latent vars $z_T, \cdots, z_1$:

- Starting from a **gaussian prior** $p(z_T) = \mathcal{N}(z_T; 0, I)$.
- Latent variables are *transformed*: $p_\theta(z_{t-1}|z_t) = \mathcal{N}(z_{t-1}; f_\theta(z_t), v_\theta(z_t))$ for $t = T, \cdots, 1$.

ELBO (Evidence Lower Bound) learning requires designing $q_\phi(z_{1:T}|x)$. We can do this:

- **Bottom up**: $z_0 \triangleq x$, $q_\phi(z_{1:T}|x) = \prod_{t=1}^T q_\phi(z_t|z_{t-1})$. Unstable, inconsistent across datasets & arch's.
- **Top down**: $q_\phi(z_{1:T}|x) = q_\phi(z_T|x)\prod_{t=2}^T q_\phi(z_{t-1}|z_t, x)$. More stable, and is the basis of **diffusion models**.

9.1 Fixed Forward Diffusion Process

Instead of learning inference dist q_ϕ, **fix the forward diffusion process** q, avoiding instability during training caused by chasing a changing posterior in hierarchical VAEs. This keeps **same dim at every step**, i.e. $\dim(x_t) = \dim(x_0)$, and gradually adds Gaussian noise to the data. The forward transition is defined as:

$$q(x_t|x_{t-1}) = \mathcal{N}\left(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t I\right)$$
$$\Leftrightarrow x_t = \sqrt{1-\beta_t}x_{t-1} + \sqrt{\beta_t}\epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0, I)$$

where β_t is a **variance schedule** that determines how much noise is added at each timestep. Since thisis linear & Gaussian, the marginal dist $q(x_t|x_0)$ has a **closed-form expression**, allowing us to sample x_t directly from x_0 without iterating through all steps. This marginal dist is:

$$q(x_t|x_0) = \mathcal{N}\left(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t)I\right), \bar{\alpha}_t = \prod_{s=1}^t(1-\beta_s)$$

and sampling can be written as:

$$x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

As t incr, $\bar{\alpha}_t$ decr toward 0, so signal from x_0 gradually disappears & $x_t \to$ **Gaussian noise**. If $q(x_0) := p_{data}(x_0)$, then dist at time t becomes:

$$q(x_t) = \int q(x_t|x_0)p_{data}(x_0)dx_0$$

i.e. **progressively smoothing data dist**: starting from complex data dist at $t = 0$, gradually transforming it into approx standard Gaussian $t = T$.

9.2 Reverse Denoising Process

After defining q, goal is to **learn reverse process** that gradually removes noise & reconstructs data. I.e. design parameterized model $p_\theta(x_{t-1}|x_t)$ that learns to **denoise** x_t step-by-step until we recover x_0. Using a **top-down decomposition** of the variational posterior:

$$q(x_{1:T}|x_0) = q(x_T|x_0)\prod_{t=2}^T q(x_{t-1}|x_t, x_0)$$

we can derive ELBO objective for training diff model. The resulting variational lower bound becomes:

$$L(x,\theta) = E_{q(x_1|x_0)}[\log p_\theta(x_0|x_1)] - KL\left(q(x_T|x_0)\|p(x_T)\right)$$
$$-\sum_{t=2}^T E_{q(x_t|x_0)}\left[KL\left(q(x_{t-1}|x_t, x_0)\|p_\theta(x_{t-1}|x_t)\right)\right]$$

Encouraging learned reverse trans $p_\theta(x_{t-1}|x_t)$ to approx true posterior $q(x_{t-1}|x_t, x_0)$.

To make learning tractable, design reverse model $p_\theta(x_{t-1}|x_t)$ to **share the same functional form** as true posterior $q(x_{t-1}|x_t, x_0)$. The true reverse conditional is Gaussian: $q(x_{t-1}|x_t, x_0) = \mathcal{N}(x_{t-1}; \tilde{\mu}_t(x_t, x_0), \tilde{\beta}_t I)$. x_0 unknown during gen, so model **predicts x_0 from x_t** using NN $x_\theta(x_t, t)$; allowing parameterization of learned reverse transition as: $p_\theta(x_{t-1}|x_t) = \mathcal{N}\left(x_{t-1}; \tilde{\mu}_t(x_t, x_\theta(x_t, t)), \tilde{\beta}_t I\right)$. Using this parameterization, the KL term between true reverse posterior & learned reverse model becomes $\propto$ **MSE objective**:

$$E_{q(x_t|x_0)}\left[D_{KL}\left(q(x_{t-1}|x_t, x_0)\|p_\theta(x_{t-1}|x_t)\right)\right]$$
$$\propto E_{q(x_t|x_0)}\left[\|x_0 - x_\theta(x_t, t)\|_2^2\right]$$

This shows training diff model $\equiv$ *predicting original clean data x_0 from noisy sample x_t at timestep t*. Instead of directly minimizing KL divergences between dists, learning problem reduces to simple regression where a NN learns to remove noise from progressively corrupted inputs, leading to stable & effective training.

9.3 Predicting Noise

A more effective parameterization: have NN **predict noise ϵ_t directly** rather than x_0. The forward process is $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon_t$, where $\epsilon_t \sim \mathcal{N}(0, I)$. Now rearrange to estimate x_0 if we know the noise: $x_0 = \frac{1}{\sqrt{\bar{\alpha}_t}}\left(x_t - \sqrt{1-\bar{\alpha}_t}\epsilon_t\right)$. Thus, the model learns a NN $\epsilon_\theta(x_t, t)$ that **predicts the noise added at timestep t**. Substituting this estimate into the reverse Gaussian mean gives the learned reverse transition:

$$E_{q(x_t|x_0)}\left[D_{KL}\left(q(x_{t-1}|x_t, x_0)\|p_\theta(x_{t-1}|x_t)\right)\right]$$
$$\propto E_{\epsilon_t \sim \mathcal{N}(0,I)}\|\epsilon_t - \epsilon_\theta(x_t, t)\|_2^2$$

Now, training obj simplifies to predicting true noise that produced x_t. The loss becomes a simple **MSE between true & predicted noise**: $E_{x_0,\epsilon,t}\left[\|\epsilon - \epsilon_\theta(x_t, t)\|_2^2\right]$. This makes diff models stable to train: network simply learns to **identify and remove Gaussian noise at diff noise levels**, so model can iteratively denoise samples from pure noise $x_T \sim \mathcal{N}(0, I)$ back to realistic data x_0.

9.4 Training

To train diff model, repeat **until convergence**:

1. Sample $x_0 \sim q(x_0)$ from the data distribution.
2. Let $t \sim \text{Uniform}\{1, \cdots, T\}$ and $\epsilon \sim \mathcal{N}(0, I)$.
3. Take a **gradient descent** step on $\nabla_\theta\|\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, t)\|_2^2$.

9.5 Sampling

To sample from the trained model:

1. Take $x_T \sim \mathcal{N}(0, I)$ as input.
2. Do steps 3-4 for $t = T, \cdots, 1$:
3. Sample $z \sim \mathcal{N}(0, I)$.
4. Let $x_{t-1} = \frac{1}{\sqrt{\alpha_t}}(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(x_t, t)) + \sigma_t z$.
5. Return x_0 as the generated sample.

10 Recurrent Neural Networks

RNNs work on *seqdata*, maintains **recurrent state** $h_t = \phi_h(W_h h_{t-1} + W_x x_t + b_h)$, where h_{t-1} = prev state, x_t is curr input, ϕ_h is act func. The output $y_t = \phi_y(W_y h_t + b_y)$.

10.1 Training

The loss function is $L_{\text{tot}}(\theta) = \sum_{t=1}^T L(y_t)$, where $\theta = \{W_h, W_x, W_y, b_h, b_y\}$ are the parameters of the RNN. To compute the loss gradient $\frac{\mathrm{d}}{\mathrm{d}\theta}L_{\text{tot}}(\theta) = \sum_{t=1}^T \frac{\mathrm{d}}{\mathrm{d}\theta}L(y_t)$, we will need:

- Update W_y: $\frac{\mathrm{d}}{\mathrm{d}W_y}L_{\text{tot}}(\theta) = \sum_{t=1}^T \frac{\mathrm{d}L(y_t)}{\mathrm{d}W_y}$, where $\frac{\mathrm{d}L(y_t)}{\mathrm{d}W_y} = \frac{\mathrm{d}L(y_t)}{\mathrm{d}y_t}\cdot\frac{\mathrm{d}y_t}{\mathrm{d}W_y}$.
- Update W_x: $\frac{\mathrm{d}}{\mathrm{d}W_x}L_{\text{tot}}(\theta) = \sum_{t=1}^T \frac{\mathrm{d}L(y_t)}{\mathrm{d}W_x}$, $\frac{\mathrm{d}L(y_t)}{\mathrm{d}W_x} = \frac{\mathrm{d}L(y_t)}{\mathrm{d}y_t}\cdot\frac{\mathrm{d}y_t}{\mathrm{d}h_t}\cdot\frac{\mathrm{d}h_t}{\mathrm{d}W_x}$, $\frac{\mathrm{d}h_t}{\mathrm{d}W_x} = \frac{\partial h_t}{\partial W_x} + \frac{\partial h_t}{\partial h_{t-1}}\cdot\frac{\mathrm{d}h_{t-1}}{\mathrm{d}W_x}$.
- Update W_h: $\frac{\mathrm{d}}{\mathrm{d}W_h}L_{\text{tot}}(\theta) = \sum_{t=1}^T \frac{\mathrm{d}L(y_t)}{\mathrm{d}W_h}$, $\frac{\mathrm{d}L(y_t)}{\mathrm{d}W_h} = \frac{\mathrm{d}L(y_t)}{\mathrm{d}y_t}\cdot\frac{\mathrm{d}y_t}{\mathrm{d}h_t}\cdot\frac{\mathrm{d}h_t}{\mathrm{d}W_h}$, $\frac{\mathrm{d}h_t}{\mathrm{d}W_h} = \frac{\partial h_t}{\partial W_h} + \frac{\partial h_t}{\partial h_{t-1}}\cdot\frac{\mathrm{d}h_{t-1}}{\mathrm{d}W_h}$.

This is **backprop through time (BPTT)**. By expanding $\frac{\mathrm{d}h_t}{\mathrm{d}W_h}$: $\frac{\mathrm{d}h_t}{\mathrm{d}W_h} = \sum_{\tau=1}^t\left(\prod_{l=\tau}^{t-1}\frac{\mathrm{d}h_{l+1}}{\mathrm{d}h_l}\right)\cdot\frac{\partial h_\tau}{\partial W_h}$.

> Depending on the non-linearity of ϕ_h, the term $\prod_{l=\tau}^{t-1}\frac{\mathrm{d}h_{l+1}}{\mathrm{d}h_l}$ can either vanish or explode, which is called the **vanishing/exploding gradient problem**, especially when $t \gg \tau$. Also, this means the dependency of y_t on x_1 gets harder to learn as t increases, which is called the **long-term dependency problem**.

10.2 Long Term Short Memory (LTSM)

An **LTSM** performs better than RNNs in learning long-term deps. Each cell has **cell state** c_t and recurrent state h_t. At each timestep, the LSTM computes:

1. **Forget gate**: discards info from cell state. $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$, σ = sigmoid func, $[\cdot,\cdot]$ denotes concat. *Depends on prev hidden state h_{t-1} & curr input x_t.*
2. **Input gate**: adds new info to cell state. $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$.
3. **Candidate cell state update gate**: Finds candidate update to cell state: $\tilde{c}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$.
4. **Cell state update**: new state $c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$, $\odot$ is element-wise mult. Here, f_t, i_t control how much of prev cell state c_{t-1} to keep and how much of candidate update $\tilde{c}_t$ to add.
5. **Hidden state update**: new hidden state $h_t = o_t \odot \tanh(c_t)$ where *output gate* $o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$ controls how much of the cell state to output.

Prediction of y_t same as RNNs: $y_t = \phi_y(W_y h_t + b_y)$.

10.3 BPTT in LSTMs

Now, we require computing $\prod_{l=1}^{t-1}\frac{\mathrm{d}c_{l+1}}{\mathrm{d}c_l} = \prod_{l=1}^{t-1}(f_{l+1} + o_l \odot \frac{\mathrm{d}\tanh(c_l)}{\mathrm{d}c_l}\cdot\frac{\mathrm{d}c_{l+1}}{\mathrm{d}h_l})$. Since f_{l+1} is sigmoid, its in range $(0, 1)$, helping prevent exploding grads. Also, $o_l \odot \frac{\mathrm{d}\tanh(c_l)}{\mathrm{d}c_l}\cdot\frac{\mathrm{d}c_{l+1}}{\mathrm{d}h_l}$ can be small, preventing vanishing grads. So, LSTMs better capture long-term deps than RNNs.

10.4 Gated Recurrente Unit (GRU)

GRU: simplified LSTM, combining forget & input gates into an **updatee**. *Faster to compute.* No conclusive evidence that LSTM or GRU performs better, so start with LSTMs & switch to GRUs need speed, or have overfitting issues.

10.5 Stacking LSTMs

Stack multiple LSTM layers to incr model capacity. Output of l-th layer at timestep t is fed as input to $(l+1)$-th layer at t. Allows model to learn more complex repr of the seq data. Don't need to wait for prev layer to finish forward pass before starting next, since they are processing same t. Having some layers with **reverse** dir (*i.e. processing seq in reverse*) helps capture deps from both dirs (**bidirect. LSTM**). However, need wait for prev layer to finish forward pass before starting next, since they are process diff t.

10.6 Sequence to Sequence Models

Machine Translation: input $x_{1:T} = (x_1, \cdots, x_T)$, output $y_{1:L} = (y_1, \ldots, y_L)$. Learn cond dist $p(y_{1:L}|x_{1:T})$by defining **autoregressive model**:

$$p_\theta(y_{1:L}|x_{1:T}) = \prod_{l=1}^L p_\theta(y_l|y_{<l}, v) \qquad v = \text{enc}(x_{1:T})$$

A **seq enc** summarizes info in input seqeuence, by:

1. Map each input token x_t to embedding e_t, add ending token.
2. Use (Stacked) LSTM to process each embedding e_t sequentially, taking final hidden state h_T as seq enc v.
3. Last LSTM used to produce prob dist of $p_\theta(y_1)$.
4. Then passed thru more LSTMs to produce prob dist of $p_\theta(y_2|y_1, v)$, and so on.

Need to pass $(x_{1:T}, y_{1:L})$ as inputs to model. In training, y_l is provided output in the supervision sequence. In testing, y_l is the output generated by the model at the previous time step. This is called **teacher forcing**.

11 Attention

In seq→seq model, decoder is $p_\theta(y_{1:L}|x_{1:T}) = \prod_{t=1}^L p_\theta(y_t|y_{<l}, v)$, v = vector repr of input $x_{1:T}$. Instead, use v_l at diff encoding steps, where:

$$v_l = \sum_{t=1}^T \alpha_{lt} f_t \qquad \alpha_{lt} = \text{softmax}(e_l)$$

$$e_l = (e_{l1}, e_{l2}, ..., e_{lT}) \qquad e_{lt} = a(h_{l-1}^d, f_t)$$

- h_{l-1}^d: RNN hidden state of the decoder at step $l-1$.
- f_t is the RNN encoder feature output at step t.
- $a(\cdot,\cdot)$ is the **alignment model**, scoring similarity between the decoder state and the encoder output.

This is **attention**, allowing decoder to focus on diff parts of input at each step. Att weights α_{lt} give importance to each enc output f_t when generating decoder output at step l.

11.1 Single Head Attention

$$\text{Att}(Q, K, V; a) = a(\frac{QK^T}{\sqrt{d_k}})V \text{ where:}$$

- $Q \in \mathbb{R}^{N \times d_q}$ has N query inputs of dim d_q.
- $K \in \mathbb{R}^{M \times d_q}$ has M key vectors of dim d_q.
- $V \in \mathbb{R}^{M \times d_v}$ has M value vectors of dim d_v.
- $a(\cdot)$ is a row-wise applied act func.
- *In **self-attention**, $K = Q$.*
- *In **hard-attention** $a(x) = \text{onehot}(\text{argmax}(x))$ for $x \in (x_1, \cdots, x_d)$.* Selects single most relevant key $\forall$ query, resulting in sparse attention distribution.
- *In **soft-attention** $a(x) = \text{softmax}(x)$ for $x \in (x_1, \cdots, x_d)$.* Produces dense attention dist, allowing model to attend to multiple keys simul, capturing complex relationships in the data.

Masked Att used in decoder of *Transformer* to prevent attending to future positions in input sequence during training. Applies mask to att scores, so each pos can only attend to prev positions and itself: $\text{MaskedAtt}(Q, K, V; a) = a\left(\text{mask}\left(\frac{QK^T}{\sqrt{d_k}}, M\right)\right)V$. Masking matrix M_{nm} takes values 0 (*mask out*) or 1 (*keep in*). The time complexity is $\mathcal{O}(MNd_q + MNd_v)$. The space complexity is $\mathcal{O}(MN + Nd_y)$.

11.2 Multi-Head Attention

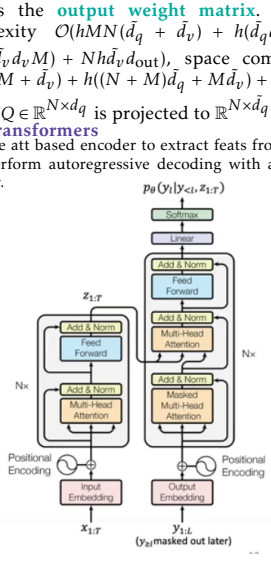
Allows jointly attending info from diff repr subspaces at diff pos:

$$\text{MultiHead}(Q, K, V, a) = \text{concat}(\text{head}_1, ..., \text{head}_h)W^O$$
$$\text{head}_i = \text{Att}(QW_i^Q, KW_i^K, VW_i^V; a)$$
$$\text{Att}(Q, K, V; a) = a(\frac{QK^T}{\sqrt{d_k}})V$$

Where W_i^Q, W_i^K, W_i^V are **projection params**, W^O is the **output weight matrix**. Time complexity $\mathcal{O}(hMN(\tilde{d}_q + \tilde{d}_v) + h(\tilde{d}_q d_q(M+N) + \tilde{d}_v d_v M) + Nh\tilde{d}_v d_{\text{out}})$, space complexity $\mathcal{O}(hN(M + \tilde{d}_v) + h((N+M)\tilde{d}_q + M\tilde{d}_v) + Nd_{\text{out}})$, where $Q \in \mathbb{R}^{N \times d_q}$ is projected to $\mathbb{R}^{N \times \tilde{d}_q}$, etc.

12 Transformers

First use att based encoder to extract feats from input, then perform autoregressive decoding with att based decoder.



Positional Encoding injects ordering info into input embeddings: $PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{\text{model}}})$, $PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{\text{model}}})$, where i is the dim. **Add & Normalize** layers used stabilize training & improve convergence. **Layer Normalization** similar to batch except performed within single hidden layer output. The output is LayerNorm(x+Sublayer(x)).

13 Efficient Deep Learning

Mem required to load model with P params, B bits/param is: $M_{\text{inf}} = \frac{PB}{8}$ bytes. For training:

$$M_{\text{train}} = \underbrace{\frac{PB}{8}}_{\text{weights}} + \underbrace{\frac{PB}{8}}_{\text{gradients}} + \underbrace{2\frac{PB}{8}}_{\text{optimiser}} + M_{\text{act}} \text{ bytes}$$

Find all stored tensors for backward pass (*usually inputs to ops*). $\forall$ tensor with batch size b, seq length n and dim d, $M = bnd$. Then $M_{\text{act}} = \frac{MB}{8}$B.

13.1 FLOPs

For $w, x \in \mathbb{R}^n, W \in \mathbb{R}^{m \times p}, X \in \mathbb{R}^{p \times n}$:

- $w + x$ is n FLOPs.
- $w^T x$ (*dot product*) is $2n - 1$ FLOPs.
- WX is $2npm$ FLOPs.

For a model layer:

1. **Layer Norm**: doing mean, var, normalisation, scale, and shift uses many elem-wise passes over $n \times d$ input $\approx 20nd$ FLOPs.
2. **QKV projections** ($\mathbf{Q}, \mathbf{K}, \mathbf{V} = \mathbf{X}W_Q, \mathbf{X}W_K, \mathbf{X}W_V$): three matmuls of $(n \times d) \times (d \times d) \to 3 \times 2nd^2 = 6nd^2$ FLOPs.
3. **Attention scores** ($\mathbf{Q}\mathbf{K}^\top/\sqrt{d_k}$): per head, $(n \times d_k) \times (d_k \times n)$; across n_h heads $\to n_h \times 2n^2 d_k = 2n^2 d$ FLOPs (since $n_h d_k = d$). Scaling by $1/\sqrt{d_k}$ is negligible.
4. **Softmax**: applied to each row of $n \times n$ score mat $\forall$ heads. Each row req. n exponents, n adds, n divs $\to 3n_h n^2$ FLOPs (negligible).
5. **Attention × Values** ($\mathbf{AV}$): per head, $(n \times n) \times (n \times d_k)$; across n_h heads $\to n_h \times 2n^2 d_k = 2n^2 d$ FLOPs.
6. **Output projection** (W_O): $(n \times d) \times (d \times d) \to 2nd^2$ FLOPs.
7. **First linear** ($d \to 4d$): $(n \times d) \times (d \times 4d) \to 2n \cdot d \cdot 4d = 8nd^2$ FLOPs.
8. **GeLU activation**: elem-wise over $n \times 4d$ elements (negligible).
9. **Second linear** ($4d \to d$): $(n \times 4d) \times (4d \times d) \to 2n \cdot 4d \cdot d = 8nd^2$ FLOPs.

13.2 Training Efficiency

- (**ALWAYS**) **Grad Accumulation**: iteratively compute grads over smaller batches, accumulate before update. Optimiser steps once. *Mem scales linearly with batch size. Small BS leads to unstable training.*
- (**ALWAYS**): **Grad Checkpointing** trade compute for mem, only storing subset of intermediate act during forward pass. During backprop, missing acts recalc as needed, incr training time but decr mem usage.
- **Mixture of Experts** increases params without increasing compute. Multiple *expert* sub-networks, only top k activated based on **router** (finds the similarity between input and expert **centroids** (learned during training)). **Routing collapse**: router selects same experts $\forall$ inputs. To mitigate, *token dropping* (*randomly drop tokens in training*) & *auxiliary loss* (add loss term encouraging fair router dist) & *bias term* (*add to routing scores, encourages underused experts*).

13.3 Finetuning (Low Rank Adaptation)

After pretraining we may want to adapt the model to a specific downstream task. **LoRA** adds low-rank mats to original weights, allowing for efficient adaptation with fewer trainable params. For weight mat $W_0 \in \mathbb{R}^{D \times D}$, learn $\Delta W \in \mathbb{R}^{D \times D}$, where $Y = XW_X\Delta W \approx XW_0 + XL_1L_2$, where $L_1 \in \mathbb{R}^{D \times r}$ & $L_2 \in \mathbb{R}^{r \times D}$ with $r \ll D$. Reduces mem to $M_{\text{lora}} \approx \frac{PB}{8}$B. **QLoRA** decr mem by quantising (*map cont. range to finite set of discrete vals*) weights to k bits, keeping LoRA params full precision.

13.4 Mixed Precision

Use lower-precision for parts of training, keeping critical ops same. Decr memory usage, incr speed without sacrificing perf. Must maintain numerical stability using higher precision for ops that are sensitive to rounding errors, i.e. grad accumulation & weight updates.

13.5 KV Caching

To avoid recalc full att over all previous tokens in autoregressive model,use **key-value (KV) cache**: store key & value projections $\forall$ previously generated tokens. Insight comes from causal att mask: when generating t, att scores for $1, \ldots, t-1$ dont depend on this, so caching avoids recomputing. For model with L layers, batch size b, seq length n, KV head dim d_{kv}, # of KV heads n_{kv}, stored at B bits/elem, KV cache requires: $M_{\text{KV}} = 2 \cdot L \cdot b \cdot n \cdot n_{\text{kv}} \cdot d_{\text{kv}} \cdot \frac{B}{8}$B