1 Probability

Sum Rule $P(X = x_i) = \sum_{j=1}^{J} p(X = x_i, Y = y_i)$

Product rule $P(X, Y) = P(Y|X)P(X)$

Independence $P(X, Y) = P(X)P(Y)$

Bayes' Rule $P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)} = \frac{P(X|Y)P(Y)}{\sum_{i=1}^{k} P(X|Y_i)P(Y_i)}$

Cond. Ind. $X \perp Y|Z \implies P(X, Y \mid Z) = P(X \mid Z)\,P(Y \mid Z)$

Cond. Ind. $X \perp Y|Z \implies P(X \mid Y, Z) = P(X \mid Z)$

$\mathbb{E}[X] = \int_{\mathcal{X}} t \cdot f_X(t)\,dt =: \mu_X$

$\mathrm{Cov}(X, Y) = \mathbb{E}_{x,y}[(X - \mathbb{E}_x[X])(Y - \mathbb{E}_y[Y])]$

$\mathrm{Cov}(X) := \mathrm{Cov}(X, X) = \mathrm{Var}[X]$

X, Y independent $\implies \mathrm{Cov}(X, Y) = 0$

$\mathbf{X}\mathbf{X}^T \geq 0$ (symmetric positive semidefinite)

$\mathrm{Var}[X] = \mathbb{E}[X^2] - \mathbb{E}[X]^2$

$\mathrm{Var}[\mathbf{AX}] = \mathbf{A}\,\mathrm{Var}[\mathbf{X}]\,\mathbf{A}^\top \quad \mathrm{Var}[aX + b] = a^2\,\mathrm{Var}[X]$

$\mathrm{Var}\left[\sum_{i=1}^n a_i X_i\right] = \sum_{i=1}^n a_i^2\,\mathrm{Var}[X_i] + 2\sum_{i<j} a_i a_j\,\mathrm{Cov}(X_i, X_j)$

$\mathrm{Var}\left[\sum_{i=1}^n a_i X_i\right] = \sum_{i=1}^n a_i^2\,\mathrm{Var}[X_i] + \sum_{i\neq j} a_i a_j\,\mathrm{Cov}(X_i, X_j)$

$\frac{\partial}{\partial t} P(X \leq t) = \frac{\partial}{\partial t} F_X(t) = f_X(t)$ (derivative of c.d.f. is p.d.f)

$f_{\alpha Y}(z) = \frac{1}{\alpha} f_Y(\frac{z}{\alpha})$

T. The moment generating function (MGF) $\psi_X(t) = \mathbb{E}\left[e^{tX}\right]$ characterizes the distr. of a rv

$Be(p)$: $pe^t + (1-p)$	$\mathcal{N}(\mu, \sigma)$: $\exp\left(\mu t + \frac{1}{2}\sigma^2 t^2\right)$
$Bin(n, p)$: $(pe^t + (1-p))^n$	$Gam(\alpha, \beta)$: $\left(\frac{1}{\alpha - \beta t}^{\alpha}\right)$ for $t < 1/\beta$
$Pois(\lambda)$: $e^{\lambda(e^t - 1)}$	

T. If $X_1, \ldots, X_n$ are indep. rvs with MGFs $M_{X_i}(t) = \mathbb{E}\left[e^{tX_i}\right]$, then the MGF of $Y = \sum_{i=1}^n a_i X_i$ is $M_Y(t) = \prod_{i=1}^n M_{X_i}(a_i t)$.

T. Let X, Y be indep., then the p.d.f. of $Z = X + Y$ is the conv. of the p.d.f. of X and Y: $f_Z(z) = \int_{\mathbb{R}} f_X(t) f_Y(z - t)\,dt = \int_{\mathbb{R}} f_X(z - t) f_Y(t)\,dt$

$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^d \det(\boldsymbol{\Sigma})}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})\right)$

$\hat{\mu} = \frac{1}{n}\sum_{i=1}^n \mathbf{x}_i \qquad \hat{\boldsymbol{\Sigma}} = \frac{1}{n}\sum_{i=1}^n (\mathbf{x}_i - \hat{\boldsymbol{\mu}})(\mathbf{x}_i - \hat{\boldsymbol{\mu}})^\top$

T. $P\left(\begin{bmatrix}\mathbf{a}_1\\ \mathbf{a}_2\end{bmatrix}\right) = \mathcal{N}\left(\begin{bmatrix}\mathbf{a}_1\\ \mathbf{a}_2\end{bmatrix} \middle| \begin{bmatrix}\mathbf{u}_1\\ \mathbf{u}_2\end{bmatrix}, \begin{bmatrix}\boldsymbol{\Sigma}_{11} & \boldsymbol{\Sigma}_{12}\\ \boldsymbol{\Sigma}_{21} & \boldsymbol{\Sigma}_{22}\end{bmatrix}\right)$

$\mathbf{a}_1, \mathbf{u}_1 \in \mathbb{R}^e$, $\boldsymbol{\Sigma}_{11} \in \mathbb{R}^{e\times e}$ p.s.d. $\boldsymbol{\Sigma}_{12} \in \mathbb{R}^{e\times f}$ p.s.d.

$\mathbf{a}_2, \mathbf{u}_2 \in \mathbb{R}^f$, $\boldsymbol{\Sigma}_{22} \in \mathbb{R}^{f\times f}$ p.s.d. $\boldsymbol{\Sigma}_{21} \in \mathbb{R}^{f\times e}$ p.s.d.

$$P(\mathbf{a}_2 \mid \mathbf{a}_1 = \mathbf{z}) = \mathcal{N}\left(\mathbf{a}_2 \middle| \mathbf{u}_2 + \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}(\mathbf{z} - \mathbf{u}_1), \boldsymbol{\Sigma}_{22} - \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12}\right)$$

T. (Chebyshev) Let X be a rv with $\mathbb{E}[X] = \mu$ and variance $\mathrm{Var}[X] = \sigma^2 < \infty$. Then for any $\epsilon > 0$, we have $P(|X - \mu| \geq \epsilon) \leq \frac{\sigma^2}{\epsilon^2}$.

2 Analysis

Log-Trick (Identity): $\nabla_\theta[p_\theta(\mathbf{x})] = p_\theta(\mathbf{x})\nabla_\theta[\log(p_\theta(\mathbf{x}))]$

T. (Cauchy-Schwarz)

$\forall \mathbf{u}, \mathbf{v} \in V\colon \langle \mathbf{u}, \mathbf{v}\rangle \leq |\langle \mathbf{u}, \mathbf{v}\rangle| \leq \|\mathbf{u}\|\,\|\mathbf{v}\|$.

$\forall \mathbf{u}, \mathbf{v} \in V\colon 0 \leq |\langle \mathbf{u}, \mathbf{v}\rangle| \leq \|\mathbf{u}\|\,\|\mathbf{v}\|$.

Special case: $(\sum x_i y_i)^2 \leq (\sum x_i^2)(\sum y_i^2)$.

Special case: $\mathbb{E}[XY]^2 \leq \mathbb{E}[X^2]\,\mathbb{E}[Y^2]$.

D. (Convex Set) A set $S \subseteq \mathbb{R}^d$ is called *convex* if

$\forall \mathbf{x}, \mathbf{x}' \in \Sigma,\ \forall \lambda \in [0, 1]\colon \quad \lambda\mathbf{x} + (1 - \lambda)\mathbf{x}' \in S$.

Com. Any point on the line between two points is within the set. $\mathbb{R}^d$ is convex.

D. (Convex Function) A function $f\colon S \to \mathbb{R}$ defined on a *convex set* $S \subseteq \mathbb{R}^d$ is called *convex* if

$$\forall \mathbf{x}, \mathbf{x}' \in S,\ \lambda \in [0, 1]\colon \quad f(\lambda\mathbf{x} + (1 - \lambda)\mathbf{x}') \leq \lambda f(\mathbf{x}) + (1 - \lambda)f(\mathbf{x})$$

Com. A function is strictly convex if the line segment between any two points on the graph of the function lies strictly above the graph. This guarantees that there is a unique global minimum.

T. (Properties of Convex Functions)

- $f(y) \geq f(x) + \nabla f(x)^T(y - x)$
- $f''(x) \geq 0$
- Local minima are global minima, strictly convex functions have a unique global minimum
- If f, g are convex then $\alpha f + \beta g$ is convex for $\alpha, \beta \geq 0$
- If f, g are convex then $\max(f, g)$ is convex
- If f is convex and g is convex and non-decreasing then $g \circ f$ is convex

D. (Strongly Convex Function) A function f is μ-strongly convex if it curves up at least as much as a quadratic function with curvature $\mu > 0$. For all x, y:

$$f(y) \geq f(x) + \nabla f(x)^T(y - x) + \frac{\mu}{2}||y - x||^2$$

Relation to Optimization:

- **Guarantee**: Ensures a unique global minimum exists.
- **Convergence**: Gradient Descent on strongly convex (and Lipschitz smooth) functions guarantees a **linear convergence rate** ($O(c^k)$ for some $c < 1$).
- **Condition Number**: The convergence speed depends on the condition number $\kappa = L/\mu$. If κ is large (poor conditioning), convergence slows down.

D. (Condition Number) The condition number $\kappa(A)$ measures the sensitivity of a function's output to small perturbations in the input. For a symmetric positive semi-definite matrix (like the Hessian H of a loss function), it is the ratio of the largest to the smallest eigenvalue:

$$\kappa(H) = \frac{|\lambda_{\max}|}{|\lambda_{\min}|} \geq 1$$

Implications for Optimization:

- **Well-conditioned** ($\kappa \approx 1$): The contours of the loss function are nearly spherical. Gradient Descent converges quickly and directly toward the minimum.
- **Ill-conditioned** ($\kappa \gg 1$): The contours form narrow, elongated ellipses (steep valleys). Gradient Descent tends to oscillate ("zigzag") across the narrow valley rather than moving down the slope, leading to very slow convergence.

Com. Momentum and Adaptive Learning Rate methods (like Adam) are specifically designed to mitigate the issues caused by high condition numbers.

T. (Taylor-Lagrange Formula)

$f(x) = \sum_{k=0}^n \frac{f^{(k)}(x_0)}{k!}(x - x_o)^k + \int_{x_0}^x \frac{f^{(n+1)}(x-t)}{n!}\,dt$

T. (Jensen) f convex/concave, $\forall i\colon \lambda_i \geq 0$, $\sum_{i=1}^n \lambda_i = 1$

$f\left(\sum_{i=1}^n \lambda_i \mathbf{x}_i\right) \leq / \geq \sum_{i=1}^n \lambda_i f(\mathbf{x}_i)$

Special case: $f(\mathbb{E}[X]) \leq \mathbb{E}[f(X)]$.

D. (L-Lipschitz Continous Function) Given two *metric spaces* (X, d_X) and (Y, d_Y), a function $f\colon X \to Y$ is called *Lipschitz continuous*, if there exists a real constant $L \in \mathbb{R}_0^+$ (*Lipschitz constant*), such that

$$\forall \mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^n\colon \quad \|f(\mathbf{x}_1) - f(\mathbf{x}_2)\| \leq L \cdot \|\mathbf{x}_1 - \mathbf{x}_2\|.$$

Com. If the objective function is L-smooth a step size of $\eta = 1/L$ guarantees convergence.

D. (Lagrangian Formulation) of $\arg\max_{x,y} f(x, y)$ s.t. $g(x, y) = c$: $\mathcal{L}(x, y, \gamma) = f(x, y) - \gamma(g(x, y) - c)$

D. (PL Condition) A differentiable function $f(x)$ with global minimum f^* satisfies the μ-Polyak-Lojasiewicz (PL) condition if there exists a constant $\mu > 0$ such that for all x:

$$\frac{1}{2}||\nabla f(x)||^2 \geq \mu(f(x) - f^*)$$

Significance:

- **Gradient Dominance**: It implies that the gradient magnitude dominates the suboptimality. If the gradient is small, the function value must be close to the optimal f^*.
- **Convergence without Convexity**: The PL condition is weaker than strong convexity (it does not require convexity at all). However, it is sufficient to guarantee a **linear convergence rate** for Gradient Descent.
- **In Deep Learning**: Over-parameterized neural networks often satisfy the PL condition in the neighborhood of a minimum, explaining fast convergence despite non-convexity.

Convergence Rate: Gradient Descent with step size $\alpha = 1/L$ (where L is the Lipschitz constant) converges as:

$$f(x_k) - f^* \leq \left(1 - \frac{\mu}{L}\right)^k (f(x_0) - f^*)$$

3 Linear Algebra

Kernels are positive semi-definite matrices.

D. (Positive Semi-Definite Matrix) A symmetric matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ is PSD if for all non-zero vectors $\mathbf{x} \in \mathbb{R}^n$: $\mathbf{x}^T\mathbf{A}\mathbf{x} \geq 0$ **Properties**:

- All eigenvalues $\lambda_i \geq 0$.
- The trace $\mathrm{Tr}(A) \geq 0$ and determinant $\det(A) \geq 0$.
- Cholesky Decomposition exists: $A = LL^T$.

T. (Sylvester Criterion) A $d \times d$ matrix is positive semi-definite if and only if all the upper left $k \times k$ for $k = 1, \ldots, d$ have a positive determinant.
Negative definite: $det < 0$ for all odd-sized minors, and $det > 0$ for all even-sized minors
Otherwise: indefinite.

D. (Trace) of $\mathbf{A} \in \mathbb{R}^{n\times n}$ is $\mathsf{Tr}(\mathbf{A}) = \sum_{i=1}^n a_{ii}$.

Properties:

- $\mathrm{Tr}(A) = \sum_i \lambda_i$ (sum of eigenvalues).
- Cyclic property: $\mathrm{Tr}(ABC) = \mathrm{Tr}(BCA) = \mathrm{Tr}(CAB)$.
- Linear: $\mathrm{Tr}(A + B) = \mathrm{Tr}(A) + \mathrm{Tr}(B)$ and $\mathrm{Tr}(cA) = c\mathrm{Tr}(A)$.
- $\mathrm{Tr}(A) = \mathrm{Tr}(A^T)$.

D. (Frobenius Norm ($||\cdot||_F$)) The square root of the sum of the absolute squares of its elements.

$$||A||_F = \sqrt{\sum_{i,j} |A_{ij}|^2}$$

Properties:

- Relation to Trace: $||A||_F = \sqrt{\mathrm{Tr}(A^T A)}$.
- Invariant under orthogonal rotations: $||QA||_F = ||A||_F$ for orthogonal Q.
- Relation to Singular Values: $||A||_F = \sqrt{\sum_i \sigma_i^2}$.

4 Derivatives

4.1 Numerator and Denominator Convention

Jacobian Layout Convention We use the *denominator-layout*. For a vector-valued function $f\colon \mathbb{R}^n \to \mathbb{R}^m$ we define

$$\left(\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right)_{ij} := \frac{\partial f_j}{\partial x_i}, \qquad \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \in \mathbb{R}^{n\times m}.$$

Hence, gradients of scalar-valued functions are column vectors, and the chain rule takes the form

$$\frac{\partial}{\partial \mathbf{x}}[\mathbf{f}(\mathbf{g}(\mathbf{x}))] = \frac{\partial \mathbf{f}}{\partial \mathbf{g}}\frac{\partial \mathbf{g}}{\partial \mathbf{x}}.$$

Remark. Sometimes the *numerator-layout* is used, where the Jacobian is defined as $(\mathbf{J}_f)_{ij} = \partial f_i/\partial x_j \in \mathbb{R}^{m\times n}$. The two conventions are related by transposition.

4.2 Scalar-by-Vector

Denominator Convention

$\frac{\partial}{\partial \mathbf{x}}[u(\mathbf{x})v(\mathbf{x})] = u(\mathbf{x})\frac{\partial v(\mathbf{x})}{\partial \mathbf{x}} + v(\mathbf{x})\frac{\partial u(\mathbf{x})}{\partial \mathbf{x}}$

$\frac{\partial}{\partial \mathbf{x}}[u(v(\mathbf{x}))] = \frac{\partial u(v)}{\partial v}\frac{\partial v(\mathbf{x})}{\partial \mathbf{x}}$

$\frac{\partial}{\partial \mathbf{x}}\left[\mathbf{f}(\mathbf{x})^\top\mathbf{g}(\mathbf{x})\right] = \frac{\partial \mathbf{f}(\mathbf{x})}{\partial \mathbf{x}}\mathbf{g}(\mathbf{x}) + \frac{\partial \mathbf{g}(\mathbf{x})}{\partial \mathbf{x}}\mathbf{f}(\mathbf{x}) = \mathbf{J}_f\mathbf{g}(\mathbf{x}) + \mathbf{J}_g\mathbf{f}(\mathbf{x})$

$\frac{\partial}{\partial \mathbf{x}}\left[\mathbf{f}(\mathbf{x})^\top\mathbf{A}\mathbf{g}(\mathbf{x})\right] = \frac{\partial \mathbf{f}(\mathbf{x})}{\partial \mathbf{x}}\mathbf{A}\mathbf{g}(\mathbf{x}) + \frac{\partial \mathbf{g}(\mathbf{x})}{\partial \mathbf{x}}\mathbf{A}^\top\mathbf{f}(\mathbf{x})$

$\frac{\partial}{\partial \mathbf{x}}\left[\mathbf{a}^\top\mathbf{x}\right] = \frac{\partial}{\partial \mathbf{x}}\left[\mathbf{x}^\top\mathbf{a}\right] = \mathbf{a}$	$\frac{\partial}{\partial \mathbf{x}}\left[\mathbf{x}^\top\mathbf{A}\mathbf{x}\right] = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$
$\frac{\partial}{\partial \mathbf{x}}\left[\mathbf{x}^\top\mathbf{x}\right] = 2\mathbf{x}$	$\frac{\partial}{\partial \mathbf{x}}\left[\mathbf{a}^\top\mathbf{f}(\mathbf{x})\right] = \frac{\partial \mathbf{f}}{\partial \mathbf{x}}\mathbf{a}$
$\frac{\partial}{\partial \mathbf{x}}\left[\mathbf{b}^\top\mathbf{A}\mathbf{x}\right] = \mathbf{A}^\top\mathbf{b}$	$\frac{\partial}{\partial \mathbf{x}}\left[\mathbf{a}^\top\mathbf{x}\mathbf{x}^\top\mathbf{b}\right] = (\mathbf{a}\mathbf{b}^\top + \mathbf{b}\mathbf{a}^\top)\mathbf{x}$

$\frac{\partial}{\partial \mathbf{x}}\left[(\mathbf{A}\mathbf{x} + \mathbf{b})^\top\mathbf{C}(\mathbf{D}\mathbf{x} + \mathbf{e})\right] = \mathbf{D}^\top\mathbf{C}^\top(\mathbf{A}\mathbf{x} + \mathbf{b}) + \mathbf{A}^\top\mathbf{C}(\mathbf{D}\mathbf{x} + \mathbf{e})$

$\frac{\partial}{\partial \mathbf{x}}\left[\|\mathbf{f}(\mathbf{x})\|_2^2\right] = \frac{\partial}{\partial \mathbf{x}}\left[\mathbf{f}(\mathbf{x})^\top\mathbf{f}(\mathbf{x})\right] = 2\frac{\partial}{\partial \mathbf{x}}[\mathbf{f}(\mathbf{x})]\,\mathbf{f}(\mathbf{x}) = 2\mathbf{J}_f\mathbf{f}(\mathbf{x})$

4.3 Vector-by-Vector

$\mathbf{A}, \mathbf{C}, \mathbf{D}, \mathbf{a}, \mathbf{b}, \mathbf{e}$ not a function of $\mathbf{x}$,
$\mathbf{f} = \mathbf{f}(\mathbf{x})$, $\mathbf{g} = \mathbf{g}(\mathbf{x})$, $\mathbf{h} = \mathbf{h}(\mathbf{x})$, $u = u(x)$, $v = v(x)$

$\frac{\partial}{\partial \mathbf{x}}[u(\mathbf{x})\mathbf{f}(\mathbf{x})] = u(x)\frac{\partial \mathbf{f}(\mathbf{x})}{\partial \mathbf{x}} + \mathbf{f}(\mathbf{x})\frac{\partial u(\mathbf{x})}{\partial \mathbf{x}}$

$\frac{\partial}{\partial \mathbf{x}}[\mathbf{x} \odot \mathbf{a}] = \mathrm{diag}(\mathbf{a})$	$\frac{\partial}{\partial \mathbf{x}}[a\mathbf{f}(\mathbf{x})] = a\frac{\partial \mathbf{f}(\mathbf{x})}{\partial \mathbf{x}} = a\mathbf{J}_f$
$\frac{\partial}{\partial \mathbf{x}}[\mathbf{a}] = \mathbf{0}$	$\frac{\partial}{\partial \mathbf{x}}[\mathbf{A}\mathbf{f}(\mathbf{x})] = \mathbf{A}\frac{\partial \mathbf{f}(\mathbf{x})}{\partial \mathbf{x}}$
$\frac{\partial}{\partial \mathbf{x}}[\mathbf{x}] = \mathbf{I}$	$\frac{\partial}{\partial \mathbf{x}}[\mathbf{f}(\mathbf{g}(\mathbf{x}))] = \frac{\partial \mathbf{f}}{\partial \mathbf{g}}\frac{\partial \mathbf{g}}{\partial \mathbf{x}} = \mathbf{J}_f(\mathbf{g})\mathbf{J}_g(\mathbf{x})$
$\frac{\partial}{\partial \mathbf{x}}[\mathbf{A}\mathbf{x}] = \mathbf{A}$	$\frac{\partial}{\partial \mathbf{x}}[\mathbf{f}(\mathbf{g}(\mathbf{h}(\mathbf{x})))] = \frac{\partial \mathbf{f}(\mathbf{g})}{\partial \mathbf{g}}\frac{\partial \mathbf{g}(\mathbf{h})}{\partial \mathbf{h}}\frac{\partial \mathbf{h}}{\partial \mathbf{x}}$
$\frac{\partial}{\partial \mathbf{x}}\left[\mathbf{x}^\top\mathbf{A}\right] = \mathbf{A}^\top$	

4.4 Scalar-by-Matrix

$\frac{\partial}{\partial \mathbf{X}}\left[\mathbf{a}^\top\mathbf{X}\mathbf{b}\right] = \mathbf{a}\mathbf{b}^\top$	$\frac{\partial}{\partial \mathbf{X}}\left[\mathbf{a}^\top\mathbf{X}^\top\mathbf{X}\mathbf{b}\right] = \mathbf{X}(\mathbf{a}\mathbf{b}^\top + \mathbf{b}\mathbf{a}^\top)$
$\frac{\partial}{\partial \mathbf{X}}\left[\mathbf{a}^\top\mathbf{X}^\top\mathbf{b}\right] = \mathbf{b}\mathbf{a}^\top$	$\frac{\partial}{\partial \mathbf{X}}[\mathsf{Tr}(\mathbf{X})] = \mathbf{I}$
$\frac{\partial}{\partial \mathbf{X}}\left[\mathbf{a}^\top\mathbf{X}\mathbf{a}\right] = \frac{\partial}{\partial \mathbf{X}}\left[\mathbf{a}^\top\mathbf{X}^\top\mathbf{a}\right] = \mathbf{a}\mathbf{a}^\top$	$\frac{\partial}{\partial \mathbf{X}}[\mathsf{Tr}(\mathbf{A}\mathbf{X}\mathbf{B})] = \mathbf{A}^\top\mathbf{B}^\top$
	$\frac{\partial}{\partial \mathbf{X}}\left[\mathsf{Tr}\left(\mathbf{A}\mathbf{X}^\top\mathbf{B}\right)\right] = \mathbf{B}\mathbf{A}$

4.5 Vector-by-Matrix (Generalized Gradient)

$\frac{\partial}{\partial \mathbf{X}}[\mathbf{X}\mathbf{a}] = \mathbf{X}^\top$

5 Information Theory

D. (Entropy) Let $\boldsymbol{X}$ be a random variable distributed according to $p(\boldsymbol{X})$. Then the entropy of $\boldsymbol{X}$

$H(\boldsymbol{X}) = \mathbb{E}[-\log(P(X))] = -\sum_{\mathbf{x}\in\mathcal{X}} p(\mathbf{x})\log(p(\mathbf{x})) \geq 0$.

It describes the expected information content $I(\boldsymbol{X})$ of $\boldsymbol{X}$.

D. (Cross-Entropy) For distributions p and q over a given set is $H(p, q) = -\sum_{\mathbf{x}\in\mathcal{X}} p(\mathbf{x})\log(q(\mathbf{x})) = \mathbb{E}_{\mathbf{x}\sim p}[-\log(q(\mathbf{x}))] \geq 0$.

$H(X; p, q) = H(X) + KL(p, q) \geq 0, \quad$ where H uses p.

Com. The minimizer of the cross-entropy is $q := p$, due to the second formulation.

Com. Usually, q is the approximation of the unknown p.

D. (Kullback-Leibler Divergence)

For probability distributions p and q defined on the same probability space, the KL-divergence between p and q is defined as

$KL(p, q) = -\sum_{\mathbf{x}\in\mathcal{X}} p(\mathbf{x})\log\left(\frac{q(\mathbf{x})}{p(\mathbf{x})}\right) = \sum_{\mathbf{x}\in\mathcal{X}} p(\mathbf{x})\log\left(\frac{p(\mathbf{x})}{q(\mathbf{x})}\right) \geq 0$.

$KL(p, q) = -\mathbb{E}_{\mathbf{x}\sim p}\left[\log\left(\frac{q(\mathbf{x})}{p(\mathbf{x})}\right)\right] = \mathbb{E}_{\mathbf{x}\sim p}\left[\log\left(\frac{p(\mathbf{x})}{q(\mathbf{x})}\right)\right] \geq 0$.

$KL(X; p, q) = H(p, q) - H(X), \quad$ where H uses p.

The KL-divergence is defined only if $\forall \mathbf{x}\colon q(\mathbf{x}) = 0 \implies p(\mathbf{x}) = 0$ (absolute continuity). Whenever $p(\mathbf{x})$ is zero the contribution of the corresponding term is interpreted as zero because $\lim_{x\to 0^+} x\log(x) = 0$.

In ML it is a measure of the amount of information lost, when q (model) is used to approximate p (true).

Com. $KL(p, y) = 0 \iff p \equiv q$.

Com. Note that the KL-divergence is not symmetric!

D. (Jensen-Shannon Divergence)

$JSD(P, Q) = \frac{1}{2}KL(P, M) + \frac{1}{2}KL(Q, M) \in [0, \log(n)?] \quad M = \frac{1}{2}(P + Q)$

C. The JSD is symmetric!

Com. The JSD is a symmetrized and smoothed version of the KL-divergence.

6 Activation Functions

Activation functions are non-linear functions that are applied element-wise to the output of a linear function.

D. (Tanh)

$$\tanh(\mathbf{x}) = \frac{e^{\mathbf{x}} - e^{-\mathbf{x}}}{e^{\mathbf{x}} + e^{-\mathbf{x}}} \qquad \tanh'(\mathbf{x}) = 1 - \tanh^2(\mathbf{x})$$

D. (ReLU)

$$\mathrm{ReLU}(x) = \max(x, 0) \qquad \mathrm{ReLU}'(x) = \begin{cases}1 & x \geq 0\\ 0 & x \leq 0\end{cases}$$

D. (Sigmoid/Logistic)

$$\sigma(\mathbf{x})_i = \frac{1}{1 + e^{-x_i}} \in (0, 1) \qquad \frac{\partial \sigma(\mathbf{x})}{\partial \mathbf{x}} = \sigma(\mathbf{x}) \odot (1 - \sigma(\mathbf{x}))$$

$$\sigma^{-1}(y) = \ln\left(\frac{y}{1 - y}\right) \qquad \nabla_{\mathbf{x}}\sigma(\mathbf{x}) = \mathbf{J}_\sigma(\mathbf{x}) = \mathrm{diag}(\sigma(\mathbf{x})\odot(1 - \sigma(\mathbf{x})))$$

$$\sigma'(x) = \frac{1}{4}\tanh'(\frac{1}{2}x) = \frac{1}{4}(1 - \tanh^2(\frac{1}{2}x)) \qquad \sigma(-x) = 1 - \sigma(x)$$

Connection between Sigmoid and Tanh (Equal Representation Strength)

$$\sigma(x) = \frac{1}{2}\tanh\left(\frac{1}{2}x\right) + \frac{1}{2} \iff \tanh(x) = 2\sigma(2x) - 1$$

D. (Softmax) $\mathrm{softmax}(\mathbf{x})_i = f(x)_i$

$$f(x)_i = \frac{e^{x_i}}{\sum_{j=1}^K e^{x_j}} \qquad \frac{\partial f(x)_i}{\partial x_j} = \begin{cases}-f(x)_i f(x)_j & i \neq j\\ f(x)_i - f(x)_i f(x)_j & i = j\end{cases}$$

$$\nabla_x f(x) = \mathbf{J}_f(x) = \mathrm{diag}(f(x)) - f(x)f(x)^T$$

7 Connectionism

McCulloch & Pitts (1943): MP-Neuron. Abstract model of neurons as linear threshold units. Inputs $x \in \{0, 1\}^n$, synapses $\sigma \in \{-1, 1\}^n$. $f(x) = \mathbb{I}(\sum \sigma_i x_i \geq \theta)$. No learning (fixed weights).

Perceptron (Rosenblatt 1958): Pattern recognition. $f(\mathbf{x}) = \mathrm{sign}(\mathbf{w}^\top\mathbf{x} + b)$. Update rule (if misclassified): $\mathbf{w}_{new} \leftarrow \mathbf{w}_{old} + y_i\mathbf{x}_i$, $b_{new} \leftarrow b_{old} + y_i$. Cannot learn the XOR function.

T. (Novikoff) Guaranteed convergence if data is linearly separable.

T. (Cover) Capacity of perceptron in $\mathbb{R}^n$ is $2n$ random patterns. Max dichotomies of S (s points in gen. pos.) by linear classifier: $C(s + 1, n) = 2\sum_{i=0}^n \binom{s}{i}$.

Assymptotic Shattering:

$$\frac{C(s, n)}{2^s} = \begin{cases}1 & \text{if } s \leq n\\ 1 - \mathcal{O}(e^{-n}) & \text{if } n < s < 2n\\ \frac{1}{2} & \text{if } s = 2n\\ \mathcal{O}(e^{-n}) & \text{otherwise}\end{cases}$$

Hopfield Networks: Associative Memory. Hebbian Learning: "Neurons that fire together, wire together". $w_{ij} \propto \sum x_i x_j$. Energy Min.: $E(\mathbf{x}) = -\frac{1}{2}\mathbf{x}^\top\mathbf{W}\mathbf{x}$.

PDP (Rumelhart et al. 1986): Parallel Distributed Processing. Introduction of Backpropagation (Generalized Delta Rule). Differentiable activations allow δ propagation to hidden layers.

8 Regression

D. (Linear Regression): $f(x) = w^T x + b$

$$\ell(w) = \frac{1}{2n}\sum_{i=1}^n (y_i - f(x_i))^2$$

Analytically solvable: $w^* = (X^T X)^{-1}X^T y$

D. (Ridge Regression): $f(x) = w^T x + b$ with ℓ_2 regularization

$$\ell(w) = \frac{1}{2n}\sum_{i=1}^n (y_i - f(x_i))^2 + \lambda\|w\|^2$$

Analytically solvable: $w^* = (X^T X + \lambda I)^{-1}X^T y$

D. (Lasso Regression): $f(x) = w^T x + b$ with ℓ_1 regularization

$$\ell(w) = \frac{1}{2n}\sum_{i=1}^n (y_i - f(x_i))^2 + \lambda\|w\|_1$$

D. (Logistic Regression): $f(x) = \sigma(w^T x + b)$

It minimizes the cross-entropy loss (negative log-likelihood), which acts as a surrogate loss for the 0/1 classification error.

$$\ell(w) = -\sum_{i=1}^n (y_i\log(\sigma(w^T x_i)) + (1 - y_i)\log(1 - \sigma(w^T x_i)))$$

9 Approximation Theory

9.1 Compositions of Maps

D. (Linear Function) A function $f\colon \mathbb{R}^n \to \mathbb{R}^m$ is a linear function if the following properties hold

- $\forall \mathbf{x}, \mathbf{x}' \in \mathbb{R}^n\colon f(\mathbf{x} + \mathbf{x}') = f(\mathbf{x}) + f(\mathbf{x}')$
- $\forall \mathbf{x} \in \mathbb{R}\,\forall \alpha \in \mathbb{R}\colon f(\alpha\mathbf{x}) = \alpha f(\mathbf{x})$

T. $f\colon \mathbb{R}^n \to \mathbb{R}$ is linear $\iff f(\mathbf{x}) = \mathbf{w}^\top\mathbf{x}$ for some $\mathbf{w} \in \mathbb{R}^n$

D. (Hyperplane)

$H := \{\mathbf{x} \mid \langle \mathbf{w}, \mathbf{x} - \mathbf{p}\rangle = 0\} = \{\mathbf{x} \mid \langle \mathbf{w}, \mathbf{x}\rangle = b\}$

where $b = \langle \mathbf{w}, \mathbf{p}\rangle$. $\mathbf{w}$ =normal vector, $\mathbf{p}$ points onto a point on the plane.

9.2 Universal Approximation

T. (Universal Approximation Theorem) MLPs with one hidden layer and a non-polynomial activation function can approximate any continuous function on a compact subset of $\mathbb{R}^d$ to arbitrary accuracy $\epsilon > 0$. Let φ be a non-constant, bounded, and continuous activation function (e.g., Sigmoid, ReLU). There exist weights v, w, b and an integer N (number of neurons) such that the network output $F(x)$ satisfies the condition:

$$|f(x) - F(x)| < \epsilon, \quad \forall x \in K$$

Uniform Approximation: This specific type of convergence guarantees that the maximum error across the *entire* domain K is bounded by ϵ. It is stricter than pointwise approximation.

T. (Weierstrass Approximation) Let f be a continuous real-valued function defined on a closed interval $[a, b]$. For every $\epsilon > 0$, there exists a polynomial $P(x)$ such that for all $x \in [a, b]$:

$$|f(x) - P(x)| < \epsilon$$

Implication: Polynomials are dense in the space of continuous functions $C[a, b]$. This is the foundation for Universal Approximation, as neural networks can approximate polynomials.

T. (Dimension Lifting (Leshno's Theorem)) By the universal approximation theorem, we know that we can approximate $C(\mathbb{R})$ with a single hidden layer of ReLUs. The lifting theorem then allows us to lift the dimension of the function space to a higher dimension, i.e. $\mathrm{span}(\{\phi(ax + b) : a, b \in \mathbb{R}\})$ universally approximates $C(\mathbb{R}^n)$.

T. (Montufar (Linear Regions)) The number of linear regions carved out by a Deep ReLU Network grows exponentially with depth L, but only polynomially with width n. For a network with L layers and width n (where $n \geq d$):

$$\#\text{Regions} = O\left(\left(\frac{n}{d}\right)^{(L-1)d} n^d\right)$$

Com. Deep networks are exponentially more expressive (in terms of complex decision boundaries) than shallow networks of the same parameter count.

T. (Shekhtman (ReLU Basis)) Any continuous piecewise linear function $g(x)$ on $[0, 1]$ (a polygonal line) with breakpoints (knots) $t_1 < \cdots < t_k$ can be represented exactly as a weighted sum of ReLU units:

$$g(x) = a + bx + \sum_{i=1}^k c_i\mathrm{ReLU}(x - t_i)$$

Com. A single hidden layer of ReLUs acts as a universal basis for 1D splines. This establishes the theoretical link between splines and ReLU networks.

T. (Barron's Theorem) For a function f with a finite Fourier moment C_f (a measure of smoothness), there exists a neural network f_n with one hidden layer of n sigmoidal units such that the integrated squared error is bounded by:

$$||f - f_n||_{L_2}^2 \leq \frac{(2C_f)^2}{n}$$

Com. The approximation error decays at a rate of $O(1/\sqrt{n})$, which is independent of the input dimension d. This suggests neural networks can overcome the "Curse of Dimensionality" for specific classes of functions.

T. (Zaslavsky's Theorem) The number of distinct regions $R(m, d)$ created by an arrangement of m hyperplanes in a d-dimensional space is bounded by:

$$R(m, d) \leq \sum_{i=0}^d \binom{m}{i}$$

Com. In the context of ReLU networks, this bounds the number of linear regions a single layer with m neurons can create. For large m, this approximates to $O(m^d)$, showing polynomial growth with width (contrast with Montufar's exponential growth with depth).

9.2.1 Maxout Networks

In 2013 k-Hinges were re-discovered under the name of *Maxout* by Goodfellow et al.

D. (Maxout) is just the max non-linearity applied to k groups of linear functions. So the input $[1 : d]$ (of the previous layer) is *partitioned* into k sets $A_1, \ldots, A_k$, and then we define the activations $G_j(\mathbf{x})$ for $j \in \{1, \ldots, k\}$ as

$$G_j(\mathbf{x}) = \max_{i\in A_j}\left\{\mathbf{w}_i^\top\mathbf{x} + b_i\right\} \qquad (i \in \{1, \ldots, d\})$$

Com. So, here we apply the nonlinearity in $\mathbb{R}^d$ (among some set members A_j) instead applying the nonlinearity in $\mathbb{R}$ (as with ridge functions).

T. (Goodfellow, 2013) Maxout networks with two maxout units that are applied to $2m$ linear functions are universal function approximators.

10 Feedforward Networks

10.1

D. (Feedforward Network) is a set of computational units arranged in a DAG (layer-wise processing)

$F = F^L \circ \cdots \circ F^1$

where each layer $l \in \{1, \ldots, L\}$ is a composition of the following functions

$F^l\colon \mathbb{R}^{m_{l-1}} \to \mathbb{R}^{m_l} \qquad F^l = \sigma^l \circ \overline{F}^l$

where $\overline{F}^l\colon \mathbb{R}^{m_{l-1}} \to \mathbb{R}^{m_l}$ is the linear function in layer l

$\overline{F}^l(\mathbf{h}^{l-1}) = \mathbf{W}^l\mathbf{h}^{l-1} + \mathbf{b}, \quad \mathbf{W}^l \in \mathbb{R}^{m_l\times m_{l-1}}, \quad \mathbf{b} \in \mathbb{R}^{m_l}$

and $\sigma^l\colon \mathbb{R}^{m_l} \to \mathbb{R}^{m_l}$ element-wise non-linearity at layer l.

Note that $\mathbf{h}^0 := \mathbf{x}$.

T. Feedforward Networks are invariant under permutations of units: The units within a hidden layer are interchangeable (along with their corresponding weights).

T. The units of feedforward networks are invariant along directions orthogonal to the weight vector $F[\mathbf{w}, \mathbf{b}](\mathbf{x}) = F[\mathbf{w}, \mathbf{b}](\mathbf{x} + \delta\mathbf{x})$: $\forall \delta\mathbf{x} \perp \mathbf{w}$

T. A layer of width m can be embedded into a layer of width $m + 1$ by adding "barren" (unused) units or by cloning existing units.

10.2 Empirical & Population Risk

In learning, we are interested in minimizing the population risk (test risk). We use the training risk (empirical risk) as an estimate of the population risk.

D. (Expected (training) Risk of a Function F)

$\ell[\theta](S) = \frac{1}{N}\sum_{i=1}^N \ell[\theta](x_i, y_i)$

D. (Population Risk/Test Risk)

$\mathcal{R}(\ell[\theta]; p) = \mathbb{E}_{(\mathbf{x},y)\sim p}\ell(\mathbf{x}, y)$

10.2.1 Linear Networks

Linear networks are composed of linear maps (with activation function ϕ = id). However, they do not gain representational power from depth because affine maps are closed under composition. They are primarily used for dimensionality reduction (contraction), c.f. Autoencoders.

10.3 Autoencoders

Goal: *Compress* the data into m-dim. ($m \leq d$) representation.

D. (Autoencoder) tries to learn the identity function.

$$\mathcal{R}(\theta) = \frac{1}{2n}\sum_{i=1}^n \|\mathbf{x} - F_\theta(\mathbf{x})\|_2^2 = \mathbb{E}_{\mathbf{x}\sim p_{emp}}[\ell(\mathbf{x}, (H \circ G)(\mathbf{x}))]$$

Typically, a linear autoencoder can be broken into two parts G and H such that $F = H \circ G \approx \mathbf{x} \mapsto \mathbf{x}$, where $G \in \mathbb{R}^{m\times n}$ is the encoder and $H \in \mathbb{R}^{n\times m}$ is the decoder.

The representation of G and H is not unique and by the Eckhart-Young theorem we can find the optimal representation (under the squared reconstruction error) by performing the SVD of the data matrix $\mathbf{X}$ (PCA).

$$\mathbf{X} = \begin{bmatrix} | & | & & | \\ \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_k \\ | & | & & | \end{bmatrix}$$

is of the following form:

$$\mathbf{X} = \underset{n\times n}{\mathbf{U}}\underbrace{\mathrm{diag}^\dagger(\sigma_1, \ldots, \sigma_{\min(n,k)})}_{=:\boldsymbol{\Sigma}\in\mathbb{R}^{n\times k}}\underset{k\times k}{\mathbf{V}^\top}.$$

And the matrices $\mathbf{U}$ and $\mathbf{V}$ are orthogonal; we have an orthogonal basis.

T. (Non-uniqueness) For any invertible matrix $\mathbf{A}$ we have

$$\underbrace{(\mathbf{U}_m\mathbf{A}^{-1})}_{\mathbf{D}}\underbrace{(\mathbf{A}\mathbf{U}_m^\top)}_{\mathbf{C}} = \mathbf{U}_m\mathbf{U}_m^\top$$

T. (Eckhart-Young) For $m \leq \min(n, k)$ and the objective

$$\underset{\hat{\mathbf{X}}\colon \mathrm{rank}(\hat{\mathbf{X}})=m}{\arg\min}\left\|\mathbf{X} - \hat{\mathbf{X}}\right\|_F^2 = \mathbf{U}_m\mathrm{diag}(\sigma_1, \ldots, \sigma_m)\mathbf{V}_m^\top$$

where the subscript m refers to the matrices of the SVD pruned to m columns.

C. This means that a linear auto-encoder with m hidden units cannot improve the SVD since $\mathrm{rank}(\mathbf{CD}) \leq m$. However, the auto-encoder can achieve the result of the SVD.

10.4 Residual Networks

D. Residual Networks learn an incremental improvement: $F(x) = x + [\phi(Wx + b)]$. They utilize "skip connections to propagate inputs forward, allowing to mitigate the vanishing/exploding gradient problem. It allows us to increase depth even more. (Important paper: ResNet 2015). The skip connection can also be transformed with a matrix $\mathbf{A}$ to allow for more flexibility $F(x) = \mathbf{A}x + [\phi(Wx + b)]$.

10.4.1 Regularization Approaches

D. Weight Decay (L_2 Regularization): Adds a penalty to the loss function proportional to the square of the weights' magnitude to constrain complexity.

$$\ell(\theta) = \ell_{emp}(\theta) + \frac{\lambda}{2}\|\theta\|^2$$

D. Data Augmentation: Artificially expands the training set by creating modified versions of existing data (e.g., flipping, rotating) to teach invariance.

D. Noise Robustness: Injects random noise into inputs, weights, or gradients to make the model insensitive to small perturbations.

D. Semi-supervised Learning: Uses a small set of labeled data alongside unlabeled data. Unlabeled data helps learn the distribution $P(x)$ to regularize $P(y|x)$.

D. Dropout: Prevents co-adaptation by randomly deactivating neurons.

- **Training**: Each neuron is kept active with probability p (and dropped with $1 - p$). This is mathematically applied using a mask vector $\mathbf{m}$:
$$y = \mathbf{m} \odot h, \quad \text{where } \mathbf{m} \sim \mathrm{Bernoulli}(p)$$
- **Inference (Weight Scaling)**: During testing, all neurons are active. To match the expected output magnitude from training (since $E[y_{train}] = p \cdot h$), we must scale the weights down:
$$W_{test} = p \cdot W_{train}$$

D. Early Stopping: Monitors validation error and stops training when performance degrades, effectively selecting parameters before overfitting occurs. Rigorous analysis shows that early stopping approximates a form of L_2 regularization.

D. Multi-task Learning: Trains on multiple tasks simultaneously using shared representations, introducing inductive bias that aids generalization.

D. Parameter Sharing: Constraints the model by forcing sets of parameters to be identical (e.g., filter weights in CNNs).

D. Ensembles: Combines predictions of multiple independent models to average out individual errors.

$$y_{final} = \frac{1}{k}\sum_{i=1}^k M_i(x)$$

D. Batch Normalization (BN) A technique to stabilize training by normalizing the inputs to a layer for each mini-batch. It addresses *Internal Covariate Shift* (the changing distribution of layer inputs during training).

1. **Normalization**: Calculate batch mean μ_B and variance σ_B^2 to normalize input x:
$$\hat{x} = \frac{x - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$
2. **Scale and Shift**: The network learns parameters γ and β to restore representation power (allows the network to undo the normalization if needed):
$$y = \gamma\hat{x} + \beta$$

Benefits: Allows higher learning rates, reduces sensitivity to initialization, and acts as a weak regularizer.

10.4.2 Initialization Techniques

Proper initialization is crucial to prevent vanishing or exploding gradients at the start of training.

D. (LeCun Initialization)

- **Goal**: Keep the variance of the input and output unchanged for efficient backprop in linear networks.
- **Use Case**: SELU activation, or standard Sigmoid (older standard).
- **Variance**: $\mathrm{Var}(W) = \frac{1}{n_{in}}$

D. (Glorot Initialization (Xavier))

- **Goal**: Maintains the variance of activations during forward pass and gradients during backward pass. Assumes linear or Tanh/Sigmoid activations.
- **Use Case**: Tanh, Sigmoid, Softmax.
- **Variance**: $\mathrm{Var}(W) = \frac{2}{n_{in} + n_{out}}$

D. (He Initialization (Kaiming))

- **Goal**: accounts for the fact that ReLU zeroes out half of the inputs (reducing variance by half). It doubles the variance compared to LeCun to compensate.
- **Use Case**: ReLU, Leaky ReLU.
- **Variance**: $\mathrm{Var}(W) = \frac{2}{n_{in}}$

D. (Orthogonal Initialization)

- **Goal**: Initialize weight matrices as orthogonal matrices ($W^T W = I$). This ensures that the matrix eigenvalues have absolute value 1.
- **Use Case**: Recurrent Neural Networks (RNNs) and very deep networks. It prevents gradients from vanishing or exploding even after many matrix multiplications (time steps).

11 Backpropagation and Optimization

11.1 Automatic Differentiation

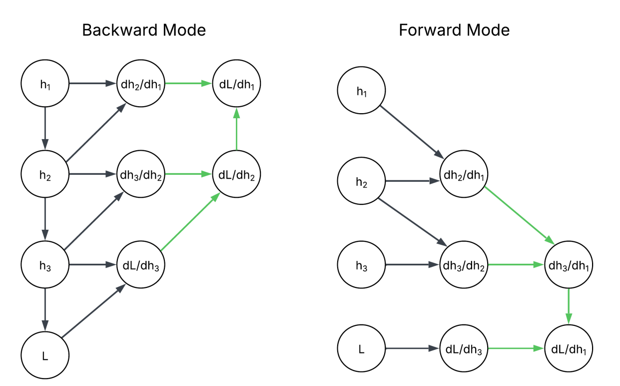
In neural networks we can efficiently compute the gradient of the loss w.r.t. the parameters $\nabla_\theta\mathcal{R}$ by exploiting the computational structure of the network.

Forward Mode AD (Forward Propagation of Tangents) Computes the derivative simultaneously with the function evaluation.

- **Mechanism**: Propagates perturbations $\dot{x}$ forward through the graph. For every node v_i, it computes the value v_i and the derivative $\dot{v}_i = \frac{\partial v_i}{\partial x}$.
- **DAG Flow**: Derivatives flow in the *same direction* as data (Input $\to$ Output). No need to store the full graph in memory.
- **Efficiency**: Efficient for functions $f\colon \mathbb{R}^n \to \mathbb{R}^m$ where $n \ll m$ (few inputs, many outputs).
- **Cost**: requires n passes to compute gradients for n parameters. In Deep Learning (millions of parameters), this is computationally prohibitive.
- $O(n)$ passes

Reverse Mode AD (Backpropagation) Computes derivatives after a full function evaluation by traversing the graph in reverse.

- **Mechanism**: First runs a *Forward Pass* to compute and store intermediate values (Primals). Then runs a *Backward Pass* to propagate adjoints $\bar{v}_i = \frac{\partial L}{\partial v_i}$.
- **DAG Flow**: Derivatives flow in the *opposite direction* (Output $\to$ Input). Requires storing the DAG and intermediate activations (memory intensive).
- **Efficiency**: Efficient for functions $f\colon \mathbb{R}^n \to \mathbb{R}^m$ where $n \gg m$ (many inputs, few outputs).
- **Relevance**: This is the standard in Deep Learning because we map millions of parameters (n) to a single scalar loss ($m = 1$). It computes all gradients in a single backward pass.
- $O(m)$ passes



11.2 Jacobian Matrix

D. (Jacobi Matrix of a Map) The Jacobian $\mathbf{J}_F$ of a map $F\colon \mathbb{R}^n \to \mathbb{R}^m$ is defined as (Numerator Layout)

$$\mathbf{J}_F := \begin{pmatrix} —(\nabla F_1)^\top— \\ —(\nabla F_2)^\top— \\ \vdots \\ —(\nabla F_m)^\top— \end{pmatrix} = \begin{pmatrix} \frac{\partial F_1}{\partial x_1} & \frac{\partial F_1}{\partial x_2} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \frac{\partial F_2}{\partial x_1} & \frac{\partial F_2}{\partial x_2} & \cdots & \frac{\partial F_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial F_m}{\partial x_1} & \frac{\partial F_m}{\partial x_2} & \cdots & \frac{\partial F_m}{\partial x_n} \end{pmatrix} \in \mathbb{R}^{m\times n}$$

T. (Jacobi Matrix Chain Rule)

$G\colon \mathbb{R}^n \to \mathbb{R}^q \quad H\colon \mathbb{R}^q \to \mathbb{R}^m$

$F := H \circ G$

Chain-rule for a single component-function $F_i\colon \mathbb{R}^n \to \mathbb{R}$ of F:

$$\left.\frac{\partial F_i}{\partial x_j}\right|_{\mathbf{x}=\mathbf{x}_0} = \left.\frac{\partial(H \circ G)_i}{\partial x_j}\right|_{\mathbf{x}=\mathbf{x}_0} = \sum_{k=1}^q \left.\frac{\partial H_i}{\partial z_k}\right|_{\mathbf{z}=G(\mathbf{x}_0)} \cdot \left.\frac{\partial G_k}{\partial x_j}\right|_{\mathbf{x}=\mathbf{x}_0}$$

11.3 Gradient Descent

D. (Gradient Descent (GD)) Iteratively moves parameters θ in the direction of the negative gradient of the loss function $J(\theta)$.

$$\theta_{t+1} = \theta_t - \eta\nabla_\theta J(\theta_t)$$

Com. In Stochastic GD (SGD), the gradient is approximated using a single sample (or mini-batch) to introduce noise and escape local minima. Altough, the gradient is unbiased it adds variance, this can help to escape local minima and saddle points.

Com. Gradient Flow can be seen as the numerical integration of the continuous-time ordinary differential equation (ODE) $\dot{x} = -\nabla f(x)$.

D. (Polyak Averaging (Averaged SGD)) Instead of using the final parameter vector θ_T, this method uses the arithmetic mean of the parameters traversed during training.

$$\bar{\theta}_t = \frac{1}{t}\sum_{i=1}^t \theta_i \quad \text{or} \quad \bar{\theta}_t = (1 - \beta)\bar{\theta}_{t-1} + \beta\theta_t$$

Benefits:

- It effectively increases the effective batch size and reduces the variance of the estimate.
- Allows the use of larger learning rates (longer steps) while still converging to the optimal solution asymptotically.
- Often achieves the optimal convergence rate of $O(1/t)$ for convex problems.

D. (Learning Rate (Robbins-Monro Conditions)) For Stochastic Gradient Descent (SGD) to guarantee convergence to a local minimum (in non-convex cases) or global minimum (in convex cases), the step size schedule α_t must satisfy two conditions:

1. **Explore Forever**: The steps must sum to infinity to ensure the algorithm can reach the optimum from any starting point, no matter how far.
$$\sum_{t=1}^\infty \alpha_t = \infty$$
2. **Decay Fast Enough**: The squared steps must sum to a finite value to ensure the variance (noise) of the updates tends to zero, preventing the parameters from oscillating forever around the minimum.
$$\sum_{t=1}^\infty \alpha_t^2 < \infty$$

Example: A schedule of $\alpha_t = \frac{1}{t}$ satisfies both, whereas $\alpha_t = \frac{1}{\sqrt{t}}$. satisfies the first but not the second.

D. (Momentum) Accelerates SGD by navigating along the relevant direction and softening oscillations in irrelevant directions. It maintains a velocity vector v (exponential moving average of past gradients).

$$\theta^{t+1} = \theta^t - \eta\nabla J(\theta^t) + \beta(\theta^t - \theta^{t-1})$$

where $\beta \in [0, 1)$ is the momentum term (friction).

D. (Nesterov Accelerated Gradient (NAG)) A "look-ahead" version of GD. It computes the gradient at the *approximate future position* of the parameters rather than the current position.

$$\vartheta^{t+1} = \theta^t + \beta(\theta^t - \theta^{t-1})$$
$$\theta^{t+1} = \vartheta^{t+1} - \eta\nabla f(\vartheta^{t+1})$$

D. (Adaptive Learning Rate Methods) These methods adjust the learning rate for each parameter individually, scaling them based on the history of gradient magnitudes.

D. (RMSProp) Designed to resolve the diminishing learning rates of Adagrad. It uses a decaying average of squared gradients.

$$E[g^2]_t = \beta E[g^2]_{t-1} + (1 - \beta)(\nabla J(\theta_t))^2$$
$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{E[g^2]_t + \epsilon}}\nabla J(\theta_t)$$

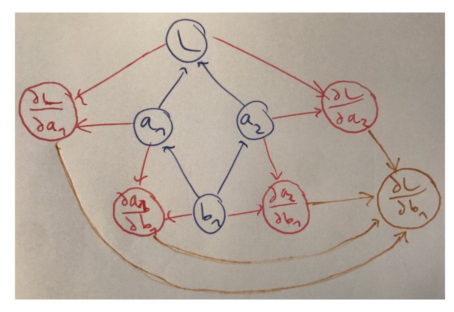
D. (Adam (Adaptive Moment Estimation)) Combines Momentum (first moment m_t) and RMSProp (second moment v_t). It also includes bias correction terms $\hat{m}_t, \hat{v}_t$ to account for initialization at zero.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla J(\theta_t)$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla J(\theta_t))^2$$
$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

11.4 Backpropagation Graphs

The approach a backpropagation graph for a loss L from a FF network graph is as follows:

1. (RED) Starting at the loss L backward: for each node n and for each of its outputs o which has a path to the loss, add the node $\frac{\partial o}{\partial n}$ (at the height of node n). Connect the output o and that node n to that newly created node.
2. (BROWN) Starting at the loss backward: for each node n create a node $\frac{\partial L}{\partial n}$ (if it doesn't exist already) and connect each previously created partial node ($\frac{\partial o}{\partial n}$) to it, and the previously crated (in this step) $\frac{\partial L}{\partial o}$'s too.



12 Convolutional Neural Networks

12.1 Convolutional Layers

D. (Transform) A transform T is a mapping from one function space $\mathcal{F}$ to another function space $\mathcal{F}'$. So $T\colon \mathcal{F} \to \mathcal{F}'$.

D. (Linear Transform) A transform T is linear, if for all functions f, g and scalars α, β, $T(\alpha f + \beta g) = \alpha(Tf) + \beta(Tg)$.

D. (Integral Transform) An *integral transform* is any transform T of the following form

$$(Tf)(u) = \int_{t_1}^{t_2} K(t, u)f(t)\,dt.$$

Com. The fourier transform is an example of an integral transform.

T. Any integral transform is a linear transform.

D. (Convolution) Given two functions $f, h\colon \mathbb{R} \to \mathbb{R}$, their convolution is defined as

$$(f * h)(u) := \int_{-\infty}^{\infty} h(t)f(u - t)\,dt = \int_{-\infty}^{\infty} h(u - t)f(t)\,dt$$

Com. Whether the convolution exists depends on the properties of f and h (the integral might diverge). However, a typical use is f = signal, and h = fast decaying kernel function.

T. (Convolution Theorem) Any linear, translation-invariant transformation T can be written as a *convolution* with a suitable h.

T. (Convolutions are commutative and associative)

T. (Convolutions are shift-invariant), we define $f_\Delta(t) := f(t + \Delta)$. Then

$$(f_\Delta * h)(u) = (f * h)_\Delta(u)$$

12.2 Discrete Time Convolutions

D. (Discrete Convolution)

For $f, h\colon \mathbb{Z} \to \mathbb{R}$, we can define the discrete convolution via

$$(f * h)[u] := \sum_{t=-\infty}^{\infty} f[t]h[u - t] = \sum_{t=-\infty}^{\infty} f[u - t]h[t]$$

Com. Note that the use of rectangular brackets suggests that we're using "arrays" (discrete-time samples).

Com. Typically we use a h with finite support (window size).

D. (Multidimensional Discrete Convolution)

For $f, h\colon \mathbb{R}^d \to \mathbb{R}$ we have

$$(f * h)[u_1, \ldots, u_d] = \sum_{t_1=-\infty}^{\infty} \cdots \sum_{t_d=-\infty}^{\infty} f(t_1, \ldots, t_d)h(u_1 - t_1, \ldots, u$$
$$= \sum_{t_1=-\infty}^{\infty} \cdots \sum_{t_d=-\infty}^{\infty} f(u_1 - t_1, \ldots, u_d - t_d)h(t_1, \ldots$$

D. (Discrete Cross-Correlation)

Let $f, h\colon \mathbb{Z} \to \mathbb{R}$, then

$$(h \star f)[u] := \sum_{t=-\infty}^{\infty} h[t]f[u + t] = \sum_{t=-\infty}^{\infty} h[-t]f[u - t]$$
$$= (\overline{h} * f)[u] = (f * \overline{h})[u] \qquad \text{where } \overline{h}(t) = h(-t).$$

aka "sliding inner product", non-commutative, kernel "flipped over" ($u + t$ instead of $u - t$). If kernel symmetric: cross-correlation = convolution.

12.3 Convolution via Matrices

Represent the input signal, the kernel and the output as *vectors*. Copy the kernel as columns into the matrix ofsetting it by one more very time (gives a band matrix (special case of Toeplitz matrix)). Then the convolution is just a matrix-vector product.

12.4 Border Handling

There are different options to do this

- **D. (Padding of p)** Means we extend the image (or each dimension) by p on both sides (so $+2p$) and just fill in a constant there (e.g., zero).
- **D. (Same Padding)** Padding with zeros = *same padding* ("same" constant, i.e., 0, and we'll get a tensor of the "same" dimensions)
- **D. (Valid Padding)** Only retain values from windows that are fully-contained within the support of the signal f (see 2D example below) = *valid padding*

12.5 Backpropagation for Convolutions

D. (Receptive Field $\mathcal{I}_i^l$ of x_i^l)

The *receptive field* $\mathcal{I}_i^l$ of node x_i^l is defined as $\mathcal{I}_i^l := \left\{j \,\middle|\, W_{ij}^l \neq 0\right\}$ where $\mathbf{W}^l$ is the Toeplitz matrix of the convolution at layer l.

Com. Hence, the receptive field of a node x_i^l are just nodes the which are connected to it and have a non-zero weight.

Com. One may extend the definition of the receptive field over several layers. The further we go back in layer, the bigger the receptive field becomes due to the nested convolutions. The receptive field may be even the entire image after a few layers. Hence, the convolutions have to be small.

Obviously, we have $\forall j \neq \mathcal{I}_i^l\colon \frac{\partial x_i^l}{\partial x_j^{l-1}} = 0$, simply because

- a node x_j^{l-1} may not be connected to x_i^l,
- or a node x_j^{l-1} may be connected to x_i^l through an edge with zero weight, so $W_{ij} = 0$ - hence, tweaking x_j^{l-1} has no effect on x_i^l.

So due to the *weight-sharing*, the kernel weight h_j^l is re-used for every unit in the target layer at layer l, so when computing the derivative $\frac{\partial \mathcal{R}}{\partial h_j^l}$ we just build an additive combination of all the derivatives (note that some of them might be zero).

$$\frac{\partial \mathcal{R}}{\partial h_j^l} = \sum_{i=1}^{m_l}\frac{\partial \mathcal{R}}{\partial x_i^l}\frac{\partial x_i^l}{\partial h_j^l}$$

Backpropagations of Convolutions as Convolutions

$\mathbf{y}^{(l)}$ output of l-th layer $\mathbf{y}^{(l-1)}$ output of $(l-1)$-th layer / input to l-th layer $\mathbf{w}$ convolution filter $\frac{\partial \mathcal{R}}{\partial \mathbf{y}^{(l)}}$ known $\mathbf{y}^{(l+1)} = \mathbf{y}^{(l)} * \mathbf{w}$

$$\frac{\partial \mathcal{R}}{\partial w_i} = \sum_k \frac{\partial \mathcal{R}}{\partial y_k^{(l)}}\frac{\partial y_k^{(l)}}{\partial w_i} = \sum_k \frac{\partial \mathcal{R}}{\partial y_k^{(l)}}\frac{\partial}{\partial w_i}\left[\mathbf{y}^{(l)} * \mathbf{w}\right]_k$$
$$= \sum_k \frac{\partial \mathcal{R}}{\partial y_k^{(l)}}\frac{\partial}{\partial w_i}\left[\sum_{o=-p}^{p} y_{k-o}^{(l-1)}w_o\right] = \sum_k \frac{\partial \mathcal{R}}{\partial y_k^{(l)}}y_{k-i}^{(l-1)}$$
$$= \sum_k \frac{\partial \mathcal{R}}{\partial y_k^{(l)}}y_{-(k-i)}^{(l-1)} = \sum_k \frac{\partial \mathcal{R}}{\partial y_k^{(l)}}\mathrm{rot180}(y^{(l-1)})_{k-i}$$
$$= \left(\frac{\partial \mathcal{R}}{\partial \mathbf{y}^{(l)}} * \mathrm{rot180}(y^{(l-1)})\right)_i$$

The derivative $\frac{\partial \mathcal{R}}{\partial \mathbf{y}^{(l)}}$ is analogous.

Note that we just used generalized indices i, k, o which may be multi-dimensional.
This example omits activation functions and biases, but that could be easily included with the chain-rule.

D. (Rotation180) $\forall i\colon \mathrm{rot180}(\mathbf{x})_i = \mathbf{x}_{(-i)}$.

12.6 Efficient Comp. of Convolutional Activities

A naive way to compute the convolution of a signal of length n and a kernel of length m gives an effort of $\mathcal{O}(m \cdot n)$. A faster way is to transform both with the FFT and then just do element-wise multiplication (effort: $\mathcal{O}(n\log n)$). However, this is rarely done in CNNs as the filters usually are small ($m \ll n$, $m \approx \log(n)$).

12.7 Typical Convolutional Layer Stages

A typical setup of a convolutional layer is as follows:

1. Convolution stage: affine transform
2. Detector stage: nonlinearity (e.g., ReLU)
3. Pooling stage: locally combine activities in some way (max, avg, ...)
Locality of the item that activated the neurons isn't too important, further we profit from dimensionality reduction. Alternative: do convolution with stride. Another thing that turns out to be so is that most of the kernels that are learned resemble a low-pass filter. Hence, when we sub-sample the images most of the information is still contained.

12.8 Pooling

There are min, max, avg, and softmax pooling. Max pooling is the most frequently used one.

D. (Max-Pooling)

- 1D: $x_i^{\max} = \max\{x_{i+k} \mid 0 \leq k < r\}$
- 2D: $x_{ij}^{\max} = \max\{x_{i+k,j+l} \mid 0 \leq k, l < r\}$

12.9 Sub-Sampling (aka "Strides")

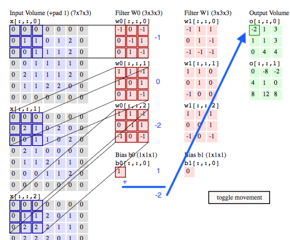
Often, it is desirable to reduce the size of the feature maps. That's why sub-sampling was introduced.

D. (Sub-Sampling) Hereby the temporal/spatial resolution is reduced.

12.10 Channels

Ex. Here we have

- an input signal that is 2D with 3 channels (7x7x3) (image x channels)
- and we want to learn two filters $W0$ and $W1$, which each process the 3 channels, and sum the results of the convolutions across each channel leading to a tensor of size 3x3x2 (convolution result x num convolutions)



Usually we convolve over all of the channels together, such that each convolution has the information of all channels at its disposition and the order of the channels hence doesn't matter.

12.11 CNNs in Computer Vision

So the typical use of convolution that we have in vision is: a sequence of convolutions

1. that *reduce* the spatial dimensions (sub-sampling)
2. that *increase* the number of channels

The deeper we go in the network, we transform the spatial information into a semantic representation. Usually, most of the parameters lie in the fully connected layers

12.12 Famous CNN Architectures

12.12.1 LeNet, 1989

MNIST, 2 Convolutional Layers + 2 Fully-connected layers

12.12.2 LeNet5

MNIST, 3 Convolutional Layers (with max-pool subsampling) + 1 Fully connected layer

12.12.3 AlexNet

ImageNet: similar to LeNet5, just deeper and using GPU (performance breakthrough)

12.12.4 Inception Module

Now, a problem that arose with this ever deeper and deeper channels was that the filters at every layer were getting longer and longer and lots of their coefficients were becoming zero (so no information flowing through). So, Arora et al. came up with the idea of an inception module.

What this inception module does is just taking all the channels for one element in the space, and reduces their dimensionality. Such that we don't get too deep channels, and also compress the information (learning the low-dimensional manifold).

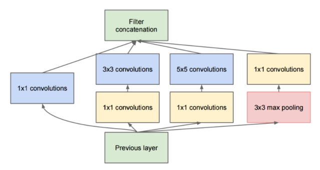
This is what gave rise to the inception module:

D. (Dimension Reduction) m channels of a 1x1xk convolution $m \leq k$:

$$\mathbf{x}^{+}ij = \sigma(\mathbf{W}\mathbf{x}_{ij}), \quad \mathbf{W} \in \mathbf{R}^{m \times k}.$$

So it uses a 1x1 filter over the k input channels (which is actually not a convolution), aka "network within a network".

12.12.5 Google Inception Network



The Google Inception Network uses many layers of this inception module along with some other tricks

- dimensionality reduction through the inception modules
- convolutions at various sizes, as different filter sizes turned out to be useful
- a max-pooling of the previous layer, and a dimensionality reduction of the result
- 1x1 convs for dimension reduction before convolving with largerkernels
- then all these informations are passed to the next layer
- gradient shortcuts: connect softmax layer at intermediate stages to have the gradient flow until the beginnings of the network
- decomposition of convolution kernels for computational performance
- all-in-all the dimensionality reductions improved the efficiency

12.13 Networks Similar to CNNs

D. (Locally Connected Network) A locally connected network has the same connections that a CNN would have, however, the parameters are not shared. So the output nodes do not connect to all nodes, just to a set of input nodes that are considered "near" (locally connected).

12.14 Comparison of #Parameters (CNNs, FC, LC)

Ex. input image $m \times n \times c$ (c = number of channels)

K convolution kernels: $p \times q$ (valid padding and stride 1)

output dimensions: $(m-p+1) \times (n-q+1) \times K$

#parameters CNN: $K(pqc+1)$

#parameters of fully-conn. NN with same number of outputs as CNN:
$mnc((m-p+1)(n-q+1)+1)K$

#parameters of locally-conn. NN with same connections as CNN:
$pqc((m-p+1)(n-q+1)+1)K$

Ex. Assume we have an $m \times n$ image (with one channel).

And we convolve it with a filter $(2p+1) \times (2q+1)$

Then the convolved image has dimensions (assuming stride 1)

- valid padding (only where it's defined): $(m-2p) \times (n-2q)$
- same padding (extend image with constant): $m \times n$ where the extended image has size $(m+2p) \times (n+2q)$.

13 Optimization

13.1 Objectives as Expectations

$$\nabla_\theta \mathcal{R}(D) = \mathbb{E}_{\mathcal{S}_N \sim p_D}\left[\nabla_\theta \mathcal{R}(\mathcal{S}_N)\right] = \mathbb{E}\left[\frac{1}{N}\sum_{i=1}^{N} \nabla_\theta \mathcal{R}(\theta; \{\mathbf{x}[i], \mathbf{y}[i]\})\right]$$

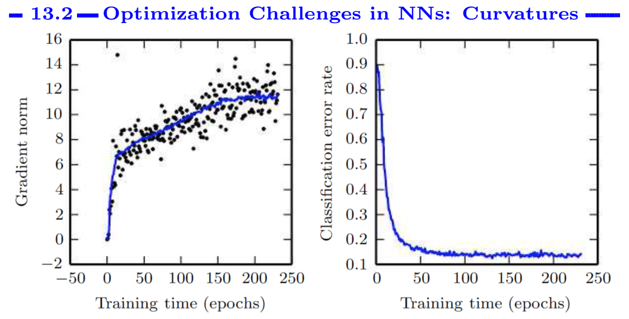
T. We have the following chain of inclusions for functions over a *closed* and *bounded* (i.e., compact) subset of the real line.

Continously differentiable $\subsetneq$ Lipschitz continuous $\subsetneq$ (Uniformly) continous

T. If we use Nesterov acceleration (in the general case), then we get a polynomial convergence rate of $\mathcal{O}(t^{-2})$.

Com. The trick used in the Nesterov approach is *momentum*.

13.2 Optimization Challenges in NNs: Curvatures



13.2.1 Convergence Rates

Under certain conditions SGD converges to the optimum:

- If we have a convex, or strongly convex objective,
- and if we have Lipschitz continous gradients,
- and a deaying learning rate, s.t.

$$\underbrace{\sum_{t=1}^{\infty} \eta_t = \infty,}_{\text{we get far enough}} \qquad \underbrace{\sum_{t=1}^{\infty} \eta_t^2 < \infty}_{\text{our steps get always smaller}}$$

typically $\eta_t = Ct^{-\alpha}$, $\frac{1}{2} < \alpha < 1$ (c.f. harmonic series.)

- or we use iterate (Polyak) averaging (once we start jumping around, we average the solutions over time).

Then, we can get the following convergence rates:

- strongly-convex case: can achieve a $\mathcal{O}(1/t)$ suboptimality rate (only polynomial convergence)
- non-strongly convex case: $\mathcal{O}(1/\sqrt{t})$ suboptimality rate (even worse than polynomial convergence)

So, even if the convergence rates are not super nice, thanks to the cheap gradient computation (only one example at the time), we may even converge faster than computing the gradient on the full dataset everytime.

14 Natural Language Processing

Similarities between text and image processing: local information.

Differences between text and image processing: texts have various lengths, texts may have long-term interactions, language is a man-made conceptions on how to communicate with each other / pictures capture the reality, pictures capture the reality / sentences may mean different things in different contexts

14.1 Word Embeddings

Basic Idea: Map symbols over a vocabulary $\mathcal{V}$ to a vector representation = *embedding* into an (euclidean) vector space (see lookup table in architecture overview).

$$\text{embedding map: (vocabulary) } \mathcal{V} \mapsto \mathbb{R}^d \text{ (embeddings)}$$
$$\text{(symbolic) } w \mapsto \mathbf{x}_w \text{ (quantitative)}$$

$$\text{word } w \in \mathcal{V} \to \text{one - hot } w \in \{0,1\}^{|V|} \to \text{embedding } \mathbf{x}_w.$$

$m := |\mathcal{V}|$, usually $|\mathcal{V}| = 10^5$

d = dimensionality of embedding, $d \ll m$

So for each of the m words in $\mathcal{V}$ we have a corresponding embedding in $\mathcal{R}^d$, which can be stored in a shared lookup table:

$\mathcal{R}^{d \times m}$ shared lookup table

Any sentence of k words can then be represented as a $d \times k$ matrix (a sequence of k embedding vectors in $\mathbb{R}^d$).

Now, how should an embedding be? Ideally, the embedding carries the information/structure that we need in order to go from the input text to the question that we want to solve. Typical questions are:

- Clustering based on context (co-occurrence)
- Sentiment analysis (group words according to mood/feelings)
- Translation (group by meaning)
- Part-of-Speech tagging (understand the structure of text, e.g., location, time, actor, ..., or, noun, verb, adjective, ...)

14.1.1 Bi-Linear Models

The first thing that we could do is to use an information theoretic quantity: the so-called *mutual information*. The mutual information is described in information theory as *how much information one random variable has about another random variable.* If two variables are independent, then, the mutual information will be zero.

So, if we put two words nearby, it's because they have to be related somehow in the *meaning* of the sentence. Hence, we expect them to have a larger mutual information.

D. (Pointwise Mutual Information)

$$\text{pmi}(v,w) = \log\left(\frac{P(v,w)}{P(v)\,P(w)}\right) = \log\left(\frac{P(v\,|\,w)}{P(v)}\right) \approx \mathbf{x}_v^\top \mathbf{x}_w + \text{const}$$

Com. As you can see this metric is bi-linear.

So we interpret the vectors as *latent variables* and link them to the observable probabilistic model. So the pointwise mutual information is related to the inner product between the latent vectors (the more related, the more co-linear the latent representations have to be).

Now, how do we compute the pointwise mutual information? One thing that we could do is to just look for words that are nearby and compute these probabilities empirically. This leads us to the idea of skip-grams.

D. (Skip Grams) The skip-gram approach is an approach to look at co-occurrences of words within a window size R (instead of looking at subsequences of some length n as with n grams). So we're only interested in the co-occurrence within some window size of words R, rather than a precise sequence.

D. (Co-Occurrence Set) Here we look at the *pairwise occurrences* of words in a *context window* of size R. So, if we have a long sequence of words $\mathbf{w} = (w_1, \ldots, w_T)$, then the co-occurrence index set is defined as

$$\mathcal{C}_R := \{(i,j) \mid 1 \leq |i-j| \leq R\}.$$

D. (Co-Ocurrence Matrix) Note that in order to get an (empirical) idea of the co-ocurrence frequencies one could compute the co-ocurrence matrix

$$\mathbf{C} \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}, \quad \text{where } C_{ij} = \begin{array}{c}\#\text{of co - occurrences of } w_i \text{ and } w_j \\ \text{within window size } R.\end{array}$$

Properties: $\mathbf{C} = \mathbf{C}^\top$ (symmetric), peaky, sparse.

One approach for embeddings: do PCA of $\mathbf{C}$ and use k eigenvectors corresponding to largest eigenvalues of $\mathbf{C}$. Note that we have

$$C_{ij} = \underbrace{\text{one-hot}(w_i)}_{\mathbf{o}_i} \mathbf{C} \underbrace{\text{one-hot}(w_j)}_{\mathbf{o}_j} = \mathbf{o}_i \underbrace{\mathbf{V}\boldsymbol{\Lambda}\mathbf{V}^\top}_{\text{SVD of } \mathbf{C}} \mathbf{o}_j$$

$$\approx \mathbf{o}_i \underbrace{\mathbf{V}_k \boldsymbol{\Lambda}_k \mathbf{V}_k^\top}_{k \text{ PCs}} \mathbf{o}_j = \underbrace{\mathbf{o}_i \mathbf{V}_k \boldsymbol{\Lambda}_k^{\frac{1}{2}}}_{\text{emb. } \mathbf{x}_{w_i}} \underbrace{\boldsymbol{\Lambda}_k^{\frac{1}{2}} \mathbf{V}_k^\top \mathbf{o}_j}_{\text{emb. } \mathbf{x}_{w_j}}$$

de-embedding: $\mathbf{V}\boldsymbol{\Lambda}_k^{-\frac{1}{2}}$ (then find nearest neighbour)

Problem: $\mathbf{C}$ is huge ($|\mathcal{V}|^2$), hence matrix-factorization becomes prohibitively expensive!

Solution: Use skip-gram approach to avoid computing $\mathbf{C}$ at all!

The solution to this is pretty simple: we train a model that tries to predict for one word w_t the preceding and following words:

$$w_{t-c}, w_{t-c+1}, \ldots, w_{t-1}, w_t, w_{t+1}, \ldots, wt+c-1, w_{t+c}.$$

Here's an illustration of the model for $t = 3$: input $w(t) \to$ projection $\to w(t-2), w(t-1), w(t+1), w(t+2)$ output

Note that the assumption (or simplification) of this model is that it assumes that the words W_i, W_j within the window $+c$, $-c$ of W_t are conditionally independent of each other given W_t.

$$W_i \perp W_j \mid W_t \qquad (i \neq j \wedge i \neq t \wedge j \neq t)$$

That might be too much of an assumption but you can see that sometimes when we're talking about something we may change the order of the words and still mean the same thing (e.g., "I was born in 1973.", "1973 is the year I was born."). So in a way we're just trying to capture the meaning of W_t with this. So this gives us an idea of the context of W_t and might relieve the structure we're looking for. So, it's not as optimal as computing $\mathbf{C}$, but it's a way to start.

So actually, what we want to do is we want to maximize the likelihood of the co-occurrences in our dataset:

$$\theta^* = \arg\max_\theta \prod_{(i,j) \in \mathcal{C}_R} P_\theta(w_i \,|\, w_j)$$

Now our approach to approximate the probability $P_\theta(w_i \,|\, w_j)$ as follows: it should be something that is related to the dot product of the embeddings, so $\mathbf{x}_{w_i}^\top \mathbf{z}_{w_j}$ (note how we use two different embeddings as the conditional probability is asymmetric), but in order to make the probability positive we'll take the exponential of it and normalize. Further, for SGD it's always better to optimize a sum: so we'll optimize the log-likelihood of co-ocurrent words in our dataset $\mathbf{w} = (w_1, \ldots, w_T)$:

$$= \arg\max_\theta \sum_{(i,j) \in \mathcal{C}_R} \log\left(\frac{\exp\left(\mathbf{x}_{w_i}^\top \mathbf{z}_{w_j}\right)}{\sum_{u \in \mathcal{V}} \exp\left(\mathbf{x}_u^\top \mathbf{z}_{w_j}\right)}\right)$$

$$= \arg\max_\theta \sum_{(i,j) \in \mathcal{C}_R} \mathbf{x}_{w_i}^\top \mathbf{z}_{w_j} - \log\Bigg(\underbrace{\sum_{u \in \mathcal{V}} \exp\left(\mathbf{x}_u^\top \mathbf{z}_w\right)}_{\text{partition function}}\Bigg)$$

So the idea is to use two different latent vectors $\mathbf{x}_w$ and $\mathbf{z}_w$ per word (to allow for the asymmetry for the conditional probability)

$$\theta = (\mathbf{x}_w, \mathbf{z}_w)_{w \in \mathcal{V}}$$

- $\mathbf{x}_w$ is used to predict w's conditional probability, and
- $\mathbf{z}_w$ is used to use w as an evidence in the cond. prob.

Note that $\mathbf{C}$ is actually symmetric (as it represents the joint probabilies), but the probabilities that we're computing are asymmetric (conditional probability).

Problem: Note however that it's too expensive to compute the *partition function* as we have to do a full sum over $\mathcal{V}$ (can be $\sim 10^5$ up to 10^7). And we'd have to do this every time we pass a new batch-sample $(w_{t+\ell}, w_t)$ through the network.

Brilliant idea of skip-grams: instead of computing the partition function, turn the problem of determining $P_\theta(w_i \,|\, w_j)$ into a classification problem (logistic regression). So, we create a classifier that determines the co-occuring likelihood of the words on a scale from 0 to 1.

For this reason we'll introduce the following function

$$D_{w_i,w_j} = \begin{cases} 1, & \text{if } (i,j) \in \mathcal{C}_R, \\ 0, & \text{if } (i,j) \notin \mathcal{C}_R. \end{cases}$$

and we'll squash the dot-product to something between 0 and 1 to make it serve as a probability

$$P_\theta(w_i \,|\, w_j) \cong P_\theta\left(D_{w_i,w_j} = 1 \,\middle|\, \mathbf{x}_{w_i}, \mathbf{z}_{w_j}\right) = \sigma(\mathbf{x}_{w_i}^\top \mathbf{z}_{w_j})$$

and the opposite event is given by:

$$P_\theta\left(D_{w_i,z_j} = 0 \,\middle|\, \mathbf{x}_{w_i}, \mathbf{z}_{w_j}\right) = 1 - \sigma(\mathbf{x}_{w_i}^\top \mathbf{z}_{w_j}) = \sigma(-\mathbf{x}_{w_i}^\top \mathbf{z}_{w_j})$$

Further, instead of just maximizing the likelihood over the dataset of co-occurrences D, we'll also maximize the log likelihood over a dataset of non-co-occurrences $\overline{D}$ (negative samples). So, we'll have the following maximization problem

$$\begin{aligned}
\theta^* &= \arg\max_\theta \sum_{(i,j) \in D} \log\left(P_\theta\left(D_{w_i,w_j} = 1 \,\middle|\, \mathbf{x}_{w_i}, \mathbf{z}_{w_j}\right)\right) \\
&\quad + \sum_{(i,j) \in \overline{D}} \log\left(1 - P_\theta\left(D_{w_i,w_j} = 0 \,\middle|\, \mathbf{x}_{w_i}, \mathbf{z}_{w_j}\right)\right) \\
&= \arg\max_\theta \sum_{(i,j) \in D} \log\left(\sigma(\mathbf{x}_{w_i}^\top \mathbf{z}_{w_j})\right) + \sum_{(i,j) \in \overline{D}} \log\left(\sigma(-\mathbf{x}_{w_i}^\top \mathbf{z}_{w_j})\right) \\
&= \arg\max_\theta \sum_{(i,j) \in \mathcal{C}_R} \Bigg[\underbrace{\log(\sigma(\mathbf{x}_{w_i}^\top \mathbf{z}_{w_j}))}_{\text{positive examples}} + \underbrace{k \cdot \mathbb{E}_{v \sim p_n}\left[\log(\sigma(-\mathbf{x}_{w_i}^\top \mathbf{z}_v))\right]}_{\text{negative examples}}\Bigg]
\end{aligned}$$

As we can see in the last step, instead of computing these sums separately, we'll do it as follows: we'll compute the log likelihood for one concrete positive example (w_i, w_j) and produce some negative examples (w_i, v), so we'll approximate the expectation below and weigh it a bit higher (oversampling facotr).

The negative examples are just sampled from the negative sampling distribution $p_n(w)$: for that we determine relative frequencies of words $p(w)$ and dampen them with a factor $\alpha < 1$ (usually: $\alpha = \frac{3}{4}$). Then $p_n(w) = \alpha p(w)$ to increase the chance of true negatives. Even if by chance we might produce false negative examples this event is rather rare. The factor k is just tu weight the negative examples a bit more in the loss. Usually $k \approx 2$ to 10 (oversampling factor).

14.2 From Embedding Words to Embedding Sequences of Words

Question: Can we extend word embeddings to embeddings for *sequences of words*? So what we're after is *understanding the sentence.*

Why is this relevant? This is the fundamental question of *statistical language modeling* (cf. Shannon). So we'd like to estimate the probability of a sequence of words in a certain order:

$$\text{estimate } P(w_1, \ldots, w_T) \overset{\text{prod. rule}}{=} \prod_{t=1}^{T} P(w_t \,|\, w_{t-1}, \ldots, w_1)$$

As we can see, thanks to the product rule this problem decomposes to predicting the next word of a sequence of words.

This problem has been adressed in different ways. Here are the traditional approach, and the more modern approaches.

1. **Traditional Approach:** k-th order Markov assumption

$$P(w_t \,|\, w_{t-1}, \ldots, w_1) \overset{\text{Markov ass.}}{\approx} P(w_t \,|\, w_{t-1}, \ldots, w_{t-k}) \overset{\text{emp. dist.}}{\approx} (k+1) - \text{gram counts}$$

   In practice we often use 5-grams, i.e., $k = 4$ (last 4 words).

2. **Modern Approach:** Create language models via embeddings

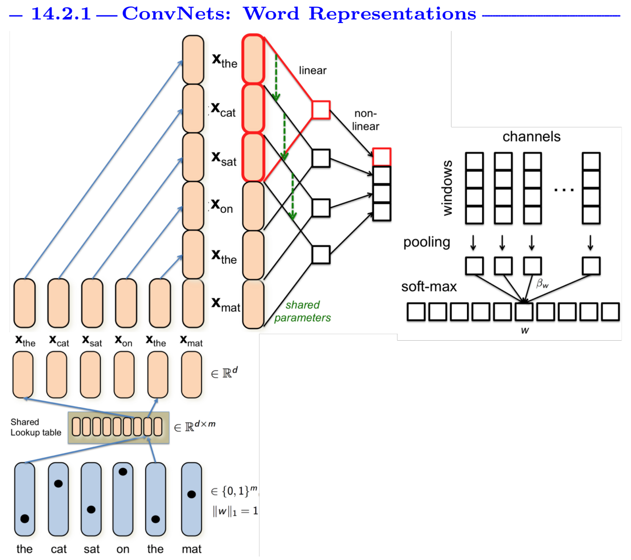
$$\log(P(w_t \,|\, \mathbf{w} := w_{t-1}, \ldots, w_1)) = \mathbf{x}_{w_t}^\top \mathbf{z}_{\mathbf{w}} + \text{const}$$

   where
   - $\mathbf{x}_{w_t}$ is the word embedding
   - $\mathbf{z}_{\mathbf{w}}$ is the sequence embedding (predicts the context)

There are three main approaches to construct *sequence embeddings*:

1. CNNs
   + conceptually simple, fast to train
   − limited range of memory
2. RNNs
   + active memory management via gated units
   − more difficult to optimize, larger datasets needed
3. Recursive networks (in combination with parsers)

14.2.1 ConvNets: Word Representations



The main stages in th *"ConvNet"* architecture aka *"time-delay NN with max-over-time pooling"* are:

1. **Embed Words:** Map word sequence (of n words) to a sequence of embedded vectors in $\mathbb{R}^d$. Store these into the *sentence matrix* in $\mathbb{R}^{d \times n}$.
2. **Concatenation:** Then, we'll concatenate the columns of the sentence matrix, which will give us a variable-length embedded *sentence vector* of length in $\mathbb{R}^{n \cdot d}$:
3. **Convolution:** Then we'll do a convolution over the sentence vector. Hereby, we'll convolve a number of embedded words (e.g., 3) at each step. Of course the convolution parameters are shared.
   - Filter size: $(dw) \times c$
     w = window size, e.g., 3-5 words, and
     k = #channels. (hyperparameter that we'll have to tune)
   - Stride: d (1 word)
   - Non-linearity: simple one like tanh or ReLU.

   So the convolution will transform the stacked vector of embeddings as follows:

$$f\colon \mathbb{R}^{n \cdot d} \to \mathbb{R}^{(n-w+1) \times k}$$

4. **Max-Pooling over Time:** Now what is really different from text to images is that texts are of different lengths (and if the images are not we can easily convert them to the same size). So the pooling is done a bit different here: what is done is that we have k channels and $n-w+1$ convolution results for the sentences. So then, for each channel we do a max-pooling in time over all the convolution results.

$$\mathbb{R}^{(n-w+1) \times k} \to \mathbb{R}^k$$

   So, suddenly, in this step we remove the temporal information. Further note, how through this mapping every word sequence now maps to a fixed-length representation.

5. **2-Layer Fully Coonnected + Softmax:** Once we have the output of the max-pooling, we just put a two-layer fully-connected layer and a softmax at the end. This will predict the following-word probabilities via soft-max as follows:

$$P(w_{t+1} \,|\, \mathbf{w}) = \frac{\exp\left(\mathbf{y}_{w_t}^\top \mathbf{z}_{\mathbf{w}}\right)}{\sum_{u \in \mathcal{V}} \exp\left(\mathbf{y}_u^\top \mathbf{z}_{\mathbf{w}}\right)}.$$

One thing to note about this architecture is that most of the parameters were in the embedding.

14.2.2 Dynamic CNNs

Kalchbrenner et al. suggested Dynamic CNNs in 2014 (as an alternative to ConvNet). They are exactly the same as ConvNets except for one thing: before doing the max-pooling over time (to get a fixed-size representation), they do a dynamic max-pooling (dynamic since it depends on the input size) over the sentence and another convolution.

14.3 Recurrent Networks (RRNs)

Disadvantage of CNNs: need to pick right convolution size, too small: no context, too large: a lot of data needed, anyways: loss of memory at some point.

Advantage of RNNs: capture better the time component, lossy memorization of past in hidden state.

Given an observation sequence $\mathbf{x}^1, \ldots, \mathbf{x}^T$. We want to identify the hidden activites $\mathbf{h}^t$ with the state of a dynamical system. The discrete time evolution of the *hidden state sequence* is expressed as a HMM with a non-linearity.

$$\theta = (\mathbf{U}, \mathbf{W}, \mathbf{b}, \mathbf{V}, \mathbf{c})$$

$$\mathbf{h}^t = F(\mathbf{h}^{t-1}, \mathbf{x}^t; \theta), \qquad \mathbf{h}^0 = \mathbf{o}.$$

$$F := \sigma \circ \overline{F}, \qquad \sigma \in \{\text{logistic,tanh,ReLU,}\ldots\}$$

$$\overline{F}(\mathbf{h}, \mathbf{x}; \theta) := \mathbf{W}\mathbf{h} + \mathbf{U}\mathbf{x} + \mathbf{b},$$

$$\mathbf{y}^t = H(\mathbf{h}^t; \theta) := \sigma(\mathbf{V}\mathbf{h}^t + \mathbf{c}),$$

There are two scenarios for producing outputs

1. Only one output at the end:

$$\mathbf{h}^T \mapsto \mathbf{H}(\mathbf{h}^T; \theta) = \mathbf{y}^T = \mathbf{y}$$

   And then we just pass this $\mathbf{y}$ to the loss $\mathcal{R}$.

2. Output a prediction at every timestep: $\mathbf{y}^1, \ldots, \mathbf{y}^T$. And then use an additive loss function

$$\mathcal{R}(\mathbf{y}^1, \ldots, \mathbf{y}^T) = \sum_{i=1}^{T} \mathcal{R}(\mathbf{y}^T) = \sum_{i=1}^{T} \mathcal{R}(H(\mathbf{h}^T; \theta))$$

- **Markov Property:** hidden state at time t depends on input of time t as well as the privous hidden state (but we don't need the oder hidden states).
- **Time-Invariance:** the state evolution function F is independent of t (it's just parametrized by θ).

Feedforward VS Recurrent Networks: RNNs process inputs in sequence, parameters shared between layers (same H and F at every timestep).

Backpropagation in Recurrent Networks

The backpropagation is straightforward: we propagate the derivatives *backwards through time*. So, the parameter sharing leads to a sum over t when dealing with the derivatives of the weights:

Algorithm 1: Backpropagation in RNNs

(Blue terms only need to be comp. for multiple-output RNNs)

// Compute derivative w.r.t. outputs

Compute $\frac{\partial \mathcal{R}}{\partial \mathbf{y}^1}, \frac{\partial \mathcal{R}}{\partial \mathbf{y}^2}, \ldots, \frac{\partial \mathcal{R}}{\partial \mathbf{y}^T}$ $\quad \left(= \frac{\partial L}{\partial \mathbf{y}^i}\right)$

// Compute the gradient w.r.t. all hidden states

$\frac{\partial \mathcal{R}}{\partial \mathbf{h}^T} \leftarrow \sum_i \frac{\partial \mathcal{R}}{\partial \mathbf{y}_i^T} \frac{\partial \mathbf{y}_i^T}{\partial \mathbf{h}^T}$

for $t \leftarrow (T-1)$ *down to* 1 **do**

  $\frac{\partial \mathcal{R}}{\partial \mathbf{h}^t} \leftarrow \sum_i \frac{\partial \mathcal{R}}{\partial \mathbf{h}_i^{t+1}} \frac{\partial \mathbf{h}_i^{t+1}}{\partial \mathbf{h}^t} + \sum_i \frac{\partial \mathcal{R}}{\partial \mathbf{y}_i^t} \frac{\partial \mathbf{y}_i^t}{\partial \mathbf{h}^t}$

// Do back-propagation over time for weights and biases

$$\frac{\partial \mathcal{R}}{\partial w_{ij}} \leftarrow \sum_{i=1}^{T} \frac{\partial \mathcal{R}}{\partial h_j^{t-1}} \frac{\partial h_j^{t-1}}{\partial w_{ij}} = \sum_{i=1}^{T} \frac{\partial \mathcal{R}}{\partial h_j^{t-1}} \cdot \dot{\sigma}_i^t \cdot h_j^{t-1},$$

$$\frac{\partial \mathcal{R}}{\partial u_{ij}} \leftarrow \sum_{i=1}^{T} \frac{\partial \mathcal{R}}{\partial h_j^{t-1}} \frac{\partial h_j^{t-1}}{\partial u_{ij}} = \sum_{i=1}^{T} \frac{\partial \mathcal{R}}{\partial h_j^{t-1}} \cdot \dot{\sigma}_i^t \cdot x_j^t,$$

$$\frac{\partial \mathcal{R}}{\partial b_i} \leftarrow \sum_{t=1}^{T} \frac{\partial \mathcal{R}}{\partial \mathbf{h}_i^t} \frac{\partial \mathbf{h}_i^t}{\partial b_i} = \sum_{t=1}^{T} \frac{\partial \mathcal{R}}{\partial \mathbf{h}_i^t} \cdot \dot{\sigma}_i^t,$$

$$\text{where } \dot{\sigma}_i^t := \sigma'(\overline{F}_i(\mathbf{h}^{t-1}, \mathbf{x}^t)).$$

$$\frac{\partial \mathcal{R}}{\partial v_{ij}} \leftarrow \sum_{i=1}^{T} \frac{\partial \mathcal{R}}{\partial y_j^{t-1}} \frac{\partial y_j^{t-1}}{\partial v_{ij}} = \sum_{i=1}^{T} \frac{\partial \mathcal{R}}{\partial y_j^{t-1}} \cdot \dot{\sigma}_i^t \cdot y_j^t,$$

$$\frac{\partial \mathcal{R}}{\partial c_{ij}} \leftarrow \sum_{i=1}^{T} \frac{\partial \mathcal{R}}{\partial y_j^{t-1}} \frac{\partial y_j^{t-1}}{\partial v_{ij}} = \sum_{i=1}^{T} \frac{\partial \mathcal{R}}{\partial y_j^{t-1}} \cdot \dot{\sigma}_i^t,$$

$$\text{where } \dot{\sigma}_i^t := \sigma'(\overline{H}_i(\mathbf{h}^{t-1}, \mathbf{x}^t)).$$

Note that $\frac{\partial \mathcal{R}}{\partial w_{ij}} \leftarrow \sum_{i=1}^{T} \sum_k \frac{\partial \mathcal{R}}{\partial h_k^{t-1}} \frac{\partial h_k^{t-1}}{\partial w w_{ij}} = \sum_{i=1}^{T} \frac{\partial \mathcal{R}}{\partial h_j^{t-1}} \frac{\partial h_j^{t-1}}{\partial w_{ij}}$.

Since for $k \neq j$ the summand is zero (similarly for u_{ij} and v_{ij}).

Exploding and/or Vanishing Gradients

One of the typical problems that RNNs have is that the gradients may explode or vanish. Remember that the gradients that we with MLPs were:

$$\nabla_{\mathbf{x}} \mathcal{R} = \mathbf{J}_{F^1} \cdot \cdots \cdot \mathbf{J}_{F^L} \nabla_{\mathbf{y}} \mathcal{R}.$$

Since we're sharing the parameters we have $\forall t$: $F^t = F$, yet evaluated at different points. Now, if the sequence is very long (large T) then we're multiplying a lot of times the same jacobian (yet evaluated at different points) by itself.

D. (Spectral Matrix Norm (Largest Singular Value))

$$\|\mathbf{A}\|_2 := \max_{\mathbf{x}, \|\mathbf{x}\|_2 = 1} \|\mathbf{A}\mathbf{x}\|_2 =: \sigma_{\max}(\mathbf{A}).$$

T. $\|\mathbf{A}\mathbf{B}\|_2 \leq \|\mathbf{A}\|_2 \cdot \|\mathbf{B}\|_2$.

Now, let's have a look at the product of Jacobians for RNNs (single-output case, otherwise it would just be a sum of longer and longer products of Jacobians). Let's have a look at the gradient of $\mathcal{R}$ w.r.t. some input $\mathbf{x}^t$ at iteration t: This is a good formula for backpropagation through time!! If we did this w.r.t. the parameters we'd have to sum it up over $t = 1$ to T.

$$\begin{aligned}
\nabla_{\mathbf{x}^t} \mathcal{R} &= \frac{\partial \mathbf{h}^t}{\partial \mathbf{x}^t} \frac{\partial \mathbf{h}^{t+1}}{\partial \mathbf{h}^t} \frac{\partial \mathbf{h}^{t+2}}{\partial \mathbf{h}^{t+1}} \cdots \frac{\partial \mathbf{h}^{T-1}}{\partial \mathbf{h}^{T-2}} \frac{\partial \mathbf{h}^T}{\partial \mathbf{h}^{T-1}} \frac{\partial \mathbf{y}^T}{\partial \mathbf{h}^T} \frac{\partial \mathcal{R}}{\partial \mathbf{y}^T} \\
&= \mathbf{J}_F^{\mathbf{x}}\big|_{\mathbf{x}^t, \mathbf{h}^{t-1}} \mathbf{J}_F^{\mathbf{h}}\big|_{\mathbf{x}^{t+1}, \mathbf{h}^t} \cdot \cdots \cdot \mathbf{J}_F^{\mathbf{h}}\big|_{\mathbf{x}^T, \mathbf{h}^{T-1}} \mathbf{J}_H\big|_{\mathbf{h}^T} \nabla_{\mathbf{y}^T} \mathcal{R}. \\
&= \mathbf{J}_F^{\mathbf{x}}\big|_{\mathbf{x}^t, \mathbf{h}^{t-1}} \left(\prod_{s=t}^{T-1} \mathbf{J}_F^{\mathbf{h}}\big|_{\mathbf{x}^{s+1}, \mathbf{h}^s}\right) \mathbf{J}_H\big|_{\mathbf{h}^T} \nabla_{\mathbf{y}^T} \mathcal{R} \\
&= \mathbf{J}_F^{\mathbf{x}}\big|_{\mathbf{x}^t, \mathbf{h}^t} \left(\prod_{s=t}^{T-1} \underbrace{\text{diag}(\sigma'(\mathbf{W}\mathbf{h}^s + \mathbf{U}\mathbf{x}^{s+1} + \mathbf{b}))}_{= \mathbf{S}_{\mathbf{x}^{s+1}, \mathbf{h}^s}} \mathbf{W}\right) \mathbf{J}_H\big|_{\mathbf{h}^T} \nabla
\end{aligned}$$

Note that when talking about $\mathbf{J}_F^{\mathbf{h}}$ we mean the Jacobian w.r.t. $\mathbf{h}$ (and analogously $\mathbf{x}$). We need to make this explicit, since F acutally has two arguments. Now, the Jacobians were just computed as follows

$$(\mathbf{J}_F)_{ij}^{\mathbf{h}}\Big|_{\mathbf{x}^t, \mathbf{h}^{t-1}} = \frac{\partial F_i}{\partial h_j^{t-1}} = \frac{\partial h_i^t}{\partial h_j^{t-1}} = \sigma'(\mathbf{W}\mathbf{h}^{t-1} + \mathbf{U}\mathbf{x}^t + \mathbf{b})_i \cdot w_{ij}$$

$$\mathbf{J}_F^{\mathbf{h}}\Big|_{\mathbf{x}^t, \mathbf{h}^{t-1}} = \underbrace{\text{diag}(\sigma'(\mathbf{W}\mathbf{h}^{t-1} + \mathbf{U}\mathbf{x}^t + \mathbf{b}))}_{= \mathbf{S}_{\mathbf{x}^t, \mathbf{h}^{t-1}}} \cdot \mathbf{W}$$

Now, for the eigenvalues of these Jacobians, we have that

$$\mathbf{S}_{\mathbf{x}^t, \mathbf{h}^{t-1}} \leq \mathbf{I} \qquad (\circ)$$

if $\sigma \in \{\text{logistic,tanh,ReLU}\}$. Hence $\mathbf{S}_{\mathbf{x}^t, \mathbf{h}^{t-1}}$ will make the gradients vanish over time (when the products gets large, so t is small) if $\mathbf{W}$ doesn't have big enough eigenvalues.

Ex. Concretely, if $\mathbf{x} := \mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^T$ was a movie review, then the gradients for the first part of the review would vanish, and only the gradients of the last part of the review would exist.

So, let's have a look at what happens to the spectral norm of this product of Jacobians:

$$\left\|\prod_{s=t}^{T-1} \mathbf{S}_{\mathbf{x}^{s+1}, \mathbf{h}^s} \mathbf{W}\right\|_2 \overset{(\circ)}{\leq} \left\|\prod_{s=t}^{T-1} \mathbf{W}\right\|_2 = \left\|\mathbf{W}^{T-t-1}\right\|_2 \leq \|\mathbf{W}\|_2^{T-t-1} = \sigma_{\max}(\mathbf{W})^{T-t-1}$$

Now, this means that

- If $\sigma_{\max}(\mathbf{W}) < 1$, then the gradients will vanish

$$\|\nabla_{\mathbf{x}^t} \mathcal{R}\|_2 \leq C \cdot \sigma_{\max}(\mathbf{W})^{T-t-1} \overset{(T-t-1) \to \infty}{\to} 0,$$

  where C is just some constant that depends on the other matrix and vector norms, but not on $T-t$.
- Conversely, if $\sigma_{\max}(\mathbf{W}) > 1$ the gradient may, or may not explode - we can't really say something.

This was observed and published in a paper by Pascanu, Mikolov, Bengion in 2013.

Fixing Exploding/Vanishing Gradients

- When the gradients are exploding a common heuristic is to clip the gradients (shring the gradient norm once it gets too big), so-called gradient clipping.

$$\nabla_{\mathbf{x}} \mathcal{R} \leftarrow \nabla_{\mathbf{x}} \mathcal{R} \cdot \frac{\gamma_{\max}}{\max\left(\|\nabla_{\mathbf{x}}\| \, \mathcal{R}, \gamma_{\max}\right)}$$

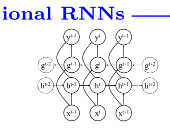
  This ensures that the gradient norm is never greater than $\gamma_{\max}$.
- However, when have vanishing gradients over time, then this means that the RNN is forgetting the past after a few timesteps, and usually then the RNN is not performing very well. This is harder to fix, and we'll see later how this is solved with LSTMS.

14.3.1 Backprop Over Time

For multi-output loss

$$\frac{L}{\theta} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \sum_{\ell=1}^{T} \left[\prod_{k=\ell+1}^{T} \frac{\partial h_k}{\partial h_{k-1}}\right] \frac{\partial h_t}{\partial \theta}$$

14.3.2 Bi-Directional RNNs



So, additionally, we're define a *reverse order sequence*

$$\mathbf{g}^t := G(\mathbf{x}^t, \mathbf{g}^{t+1}; \theta) = \sigma(\mathbf{P}\mathbf{x}^t + \mathbf{Q}\mathbf{g}^{t+1} + \mathbf{d}), \quad \text{with } \mathbf{g}^T = \mathbf{o}$$

the function F to compute the normal order sequence stays the same, and the output transform H becomes now a function of both hidden states:

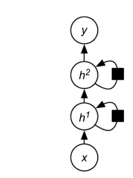
$$\mathbf{y}^t = H(\mathbf{h}^t, \mathbf{g}^t; \theta).$$

The nice thing is that we can compute both sequences (the forward and backward sequence) in parallel (or independently) - just verify this in the graph. The back-propagation through time is done in reverse order for the reverse order sequence.

14.3.3 Deep Recurrent Networks

Deep recurrent networks (DRNNs) just use a deeper network for the evolution function. So we have hierarchical hidden states. Note, that this can also be combined with bi-directionality.

$$\begin{aligned}
\mathbf{h}^{t,1} &= F(\mathbf{h}^{t-1,1}, \mathbf{x}^t; \theta) \\
&\vdots \\
\mathbf{h}^{t,l} &= F(\mathbf{h}^{t-1,l}, \mathbf{h}^{t,l-1}; \theta) \\
\mathbf{y}^T &= H(\mathbf{y}^{t,L}; \theta).
\end{aligned}$$



14.3.4 Probability Distributions over Sequences

Goal: Define a conditional probability distribution over output sequence $\mathbf{y}^{1:T}$, given input sequence $\mathbf{x}^{1:T}$.

So the idea would be to do a step-by-step prediction:

$$P\left(\mathbf{y}^{1:T} \,\middle|\, \mathbf{x}^{1:T}\right) \approx \prod_{t=1}^{T} P\left(\mathbf{y}^t \,\middle|\, \mathbf{x}^{1:t}, \mathbf{y}^{1:(t-1)}\right)$$

Now, in the naive RNN implementation $P(\mathbf{y}^t)$ only depends on $\mathbf{y}^{1:(t-1)}$ through $\mathbf{h}^t$ since

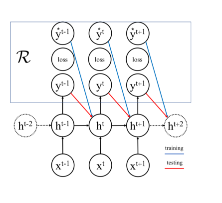
$$\mathbf{x}^{1:t} \overset{F}{\mapsto} \mathbf{h}^T \overset{H}{\mapsto} \boldsymbol{\mu}^t \mapsto P\left(\mathbf{y}^T\right).$$

Problems when learning RNNs One of the problems that we may have when we're learning to predict sequences to sequences is that the elements of the predicted sequence are somewhat uncorrelated if we don't train the right way. Actually, it would be better to consider what we predicted for the neighbouring elements when predicting an element of a sequence. Further, if the prediction was wrong it would be actually misleading, and so the error would get propagated. A better approach to do this is to feed in the right prediction as the neighbouring prediction during training time such that the training and sequence doesn't get fully of track because of one misprediction.

D. (Teacher Forcing) That's why output feedback was introduced, with output feedback the output function takes an additional input $\mathbf{y}^{t-1}$ (for unidirectional RNNs) such that we can consider what the previous sequence prediction was. So

$$\mathbf{y}^t = H(\mathbf{h}^t, \mathbf{y}^{t-1}).$$

As we can see in the picture we feed in the true value during training, and we use the predicted value at test time.



This technique is called *teacher forcing* (even if we do a wrong prediction, we force it to be the true value).

D. (Curriculum Learning) Another related idea is *curricular learning* where we alternate randomly between teacher forcing and using the actual (and maybe wrong) prediction. So even if there are some wrong predictions, we force the network to come to the right prediction at some point and it may recover from the small errors.

15 Memory & Attention

15.1 Memory Units

Problem with RNNs: vanishing gradients (changes in the input a long time ago won't really affect the output/loss), so it's hard to learn long-term dependencies as the information in $\mathbf{h}_t$ fades-out when combined with current input $\mathbf{x}_t$.

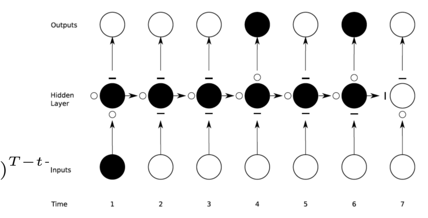
Goal of Memory Units: model long-term dependencies. have some kind of memory.

15.1.1 LSTMs

It would be nice to have something like a gated unit

D. (Gated Unit)

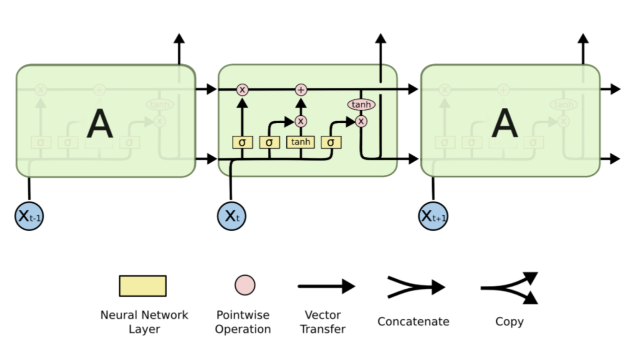
The following picture illustrates how we have a memory unit where we can store, read and delete information illustrated by the *gated units.*



Advantages: information can be remembered for a long time (doesn't fade away as with RNNs). Further, we can delete something in memory in one timestep (without having to fade it out).

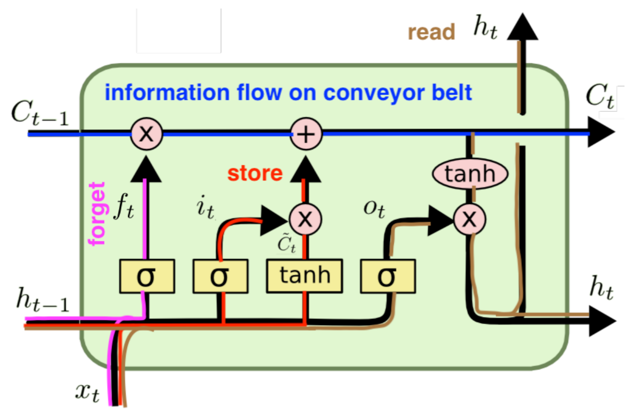
D. (LSTM) Long Short-Term Memory: Remembering information for long time and forgetting it fast.

An LSTM is just a complex unit for memory management to achieve these objectives. It has the following computation graph:



Now an LSTM has two memories:

- C_t is the *main memory* in the LSTM (a vector). That's where we store all of our information. The memory we'll control through the gated units.
- h_t is our previous output. So it's not our memory, but it's what the sequence model outputs at timestep t. This could then passed further to a prediction network.



The LSTM adds three more gates. The idea is that the information C_t flows on a **conveyor-belt** where we selectively forget, store and output as follows:

- **Forget Gate:** This is just a one-layer neural network, where

$$\mathbf{f}^t = \sigma(\mathbf{W}_f \mathbf{h}^{t-1} + \mathbf{U}_f \mathbf{x}^t + \mathbf{b}_f) \qquad \text{(forget)}$$

  Using the previous output vector, and the current input it computes some weights, which are used to multiply the content of the memory (by a number between 0 and 1) in order to selectively forget some entries in the memory.
- **Store/Input Gate:** Here we'll compute

$$\mathbf{i}^t := \sigma(\mathbf{W}_i \mathbf{h}^{t-1} + \mathbf{U}_i \mathbf{x}^t + \mathbf{b}_i) \qquad \text{(input)}$$

  which determines how much we want to input into the memory (how much we want to store). In many cases, this is just mapped as $\mathbf{i}^t = 1 - \mathbf{f}^t$. But this model illustrates the flexibility that we may want to keep. Further, given that we'll be opening the gate, what should we store there? This is what is computed through

$$\overline{C}^t = \tanh(\mathbf{W}_C \mathbf{h}^{t-1} + \mathbf{U}_C \mathbf{x}^t + \mathbf{b}_C).$$

  And in the end we'll update the memory by combining the weighted sums of the stored and new information

$$C^t = \underbrace{\mathbf{f}^t}_{\text{forget factors}} \odot \underbrace{C^{t-1}}_{\text{current mem}} + \underbrace{\mathbf{i}^t}_{\text{write factors}} \odot \underbrace{\overline{C}^t}_{\text{net in}}.$$

- **Read/Output Gate:** In this gate we decide which information that we want to get out into our output (the new hidden state):

$$\mathbf{o}^t = \sigma(\mathbf{W}_o \mathbf{h}^{t-1} + \mathbf{U}_o \mathbf{x}^t + \mathbf{b}_o) \qquad \text{(output)}$$

$$\mathbf{h}^t = \underbrace{\mathbf{o}^t}_{\substack{\text{what we want} \\ \text{to let out}}} \odot \tanh(\underbrace{C^t}_{\text{new mem.}}).$$

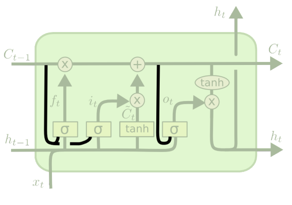
Note that when stacking $\mathbf{h}^{t-1}$ and $\mathbf{x}^t$ we may just represent things as follows:

$$[\mathbf{W}^\kappa \quad \mathbf{U}^\kappa] \begin{bmatrix} \mathbf{h}^{t-1} \\ \mathbf{x}^t \end{bmatrix} + \mathbf{b}^\kappa = \mathbf{W}^k \mathbf{h}^{t-1} + \mathbf{U}^k \mathbf{x}^t + \mathbf{b}^\kappa.$$

15.1.2 Other Variants of LSTMs

LSTMs with Peepholes

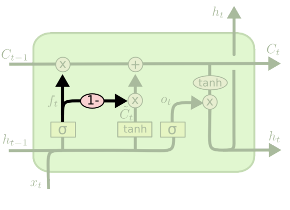
There are a bunch of other LSTMs that have been invented. A change that was done with LSTMS was: *LSTS with peepholes* where each of the gates also is allowed to look at the memory (Gers & Schmidhuber, 2000).



$$f_t = \sigma\left(W_f \cdot [\mathbf{C_{t-1}}, h_{t-1}, x_t] + b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\mathbf{C_{t-1}}, h_{t-1}, x_t] + b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\mathbf{C_t}, h_{t-1}, x_t] + b_o\right)$$

This makes a lot of sense if we start multiplying by a lot of zeros when we're carrying the information out.

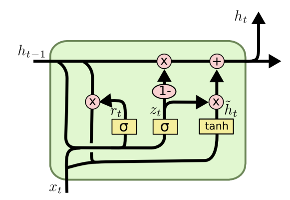
Coupled Forget and Input Gates



$$C_t = f_t * C_{t-1} + (\mathbf{1} - \mathbf{f_t}) * \tilde{C}_t$$

Another variation is to use coupled forget and input gates. Instead of separately deciding what to forget and what we should add new information to, we make those decisions together. We only forget when we're going to input something in its place. We only input new values to the state when we forget something older.

GRU Networks (Gated Memory Unit)



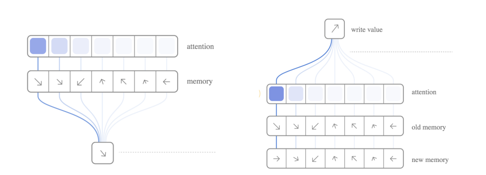
$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

This is basically a simplification of the LSTM that we've seen where we have only one state. Usually these ones tend to be much faster to train. It combines the forget and input gates into a single "update gate." It also merges the cell state and hidden state, and makes some other changes.

15.2 Differentiable Memory

D. (Neural Turing Machine)

NTMs are very reminiscent of a Turing machine, but: each cell $M_i \in \mathbb{R}^d$.



In contrast to RNNs, NTMs use an external memory to which they read and write according to some determined probabilities.Now here's an example how that can become very useful. We may have these vectors in memory, and then we may search for some vector. The greater the dot-product is with a vector, the greater our attention will be for that vector in the memory (via the attention distribution).

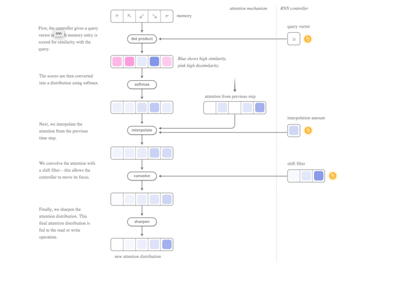
So the operations that can be done are the following:

1. Compute the attention distribution: $(\alpha_i)_i$, $\alpha_i \geq 0$ s.t. $\sum_{\alpha_i} = 1$.
2. Read: out expected memory content: $r \leftarrow \sum_i \alpha_i M_i$.
3. Write: Now, given that we have one value that we'd like to write, we'll write it to the memory location with the biggest attention:

$$(\beta_i)_i, \quad \beta_i \in [0;1], \quad M_i \leftarrow (1-\beta_i) M_i + \beta_i w.$$

   Usually, there's one value where we have a lot of attention (curse of dimensionality), and lots where we have little attention. Further, as we can see then we'll write a lot so somwhere and a little bit to everywhere else. The reason we're using a slowly varying β_i is that we want to be able to take the derivative (if it was just 0 or 1 we couldn't take the derivative). So this is why these things tend to be soft.

The typical memory controller works as follows:



1. For a *query* vector see which memory cells get the most attention.
2. Normalize the attention distribution
3. Interpolate the attention with the previous attention
4. Convolve the the attention if needed (ability to shift attention relative to content-selected locations)
5. Additionally sharpen the final attention distribution.
6. Then do your operation (read or write) according to the attention distribution.

These operations are all differentiable, as we're using probabilities and not only the values 0 or 1. Other resembling architectures are: neural random access machines, differentiable data structures (stacks, queues).

Now, NTM architectures can learn loops and simple programs. However, no real-world applications have been developed with this so-far.

15.3 Attention Mechanisms

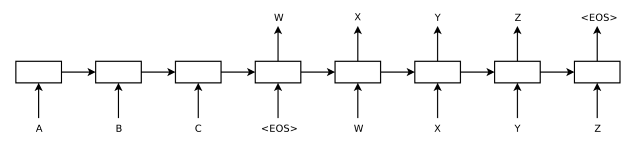
D. (Attention Mechanisms) offer a simple way to overcome some challenges of RNN-based memorization. With attention mechanisms we selectively attend to *inputs* or *feature representations* computed from inputs.

- RNNs: learn to encode information relevant for the future.
- Attention: selects what is relevant from the past in hindsight!

Both ideas can be combined!

Ex. If we have a sentence in English and one in German the question is how do we match one to the other. The problem with CTC was that if things are changed in order, then CTC cannot deal with it. Because the CTC doesn't process every input before it produces an output. Attention will provide a mechanism to deal with this.

So we'll see how we can do sequence to sequence learning. The idea fairly simple: Let's say we have a sequence ABC and we want to map it to $WXYZ$. To achieve this we'll use the so-called *encoder-decoder architecture*:



So what we'll do is

- we'll *encode* the sequence (e.g., sentence) into a vector, and then
- we'll *decode* the sequence (e.g., translate) from the vector (w/ output feedback) into another sequence.

So the probability that we want to determine is

$$P\left(\mathbf{y}^1, \ldots, \mathbf{y}^{T_y} \,\middle|\, \mathbf{x}_1, \ldots, \mathbf{x}^{T_x}, F(\mathbf{x}^{T_x})\right).$$

The issue that we have here is that T_x and T_y have variable lengths, and the difference between the two lengths is not always the same. So it's very hard to match one sequence to another. Now, sequence learning will compute a function

$$F(\mathbf{x}^1, \ldots, \mathbf{x}^{T_x}) = \text{"thought vector"}$$

which will be a vector which will have all the information that we need from the input sequence to compute the output sequence. This F is the so-called "thought vector" (Hinton). So F will be computed via an LSTM.

To produce the output sequence we'll use another LSTM that takes as input the thought vector F plus the output that we'll be producing (output feedback).

How to make the RNN Encoder/Decoder Work?

The following things were discovered by Sutskever, Vinals & Le in 2014:

- Use Deep LSTMs (multiple layers, e.g., 4)
- Use different RNNs for encoding and decoding
- Apply beam search for decoding
- Reverse the order of the source sequence
- Ensemble-ing

For a machine translation task this gave state-of-the-art results on WMT benchmarks. However, traditional approaches use *sentence alignment models.* We still don't know what is the equivalent in a neural architecture.

— 15.3.1 — Seq2Seq with Attention

The issue with the encoder-decoder architecture is that if we're translating a very long sequence, it might have the issue that suddenly we have to store the entire sequence in a single vector. But when we as humans translate we translate small parts into small parts. In order to understand this better let's have a look at a concrete example. Let's say that we want to translate the following sentence from English to French.

- bi-directionality (it's good to know future and past context)
- select useful hidden states based on atention
- sizes of sentences might not be the same
- outputted workds might have slightly different order
- Note that if we don't have dependencies that are out of order we can use the CTC approach.

— 15.4 — Recursive Networks

Good to process tree-structure, e.g., from a parser (more depth efficient $\mathcal{O}(\log(n))$). Gives a single output at the root.

$F\colon \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^d$

$\mathbf{h}^n = F(\mathbf{h}^{n_\text{left}}, \mathbf{h}^{n_\text{right}})$

16 Unsupervised Learning

— 16.1 — Density Estimation

D. (Density Estimation) is used to learn the distribution of the data. Classically, we use a *parametric family of densities* $\{p_\theta \mid \theta \in \Theta\}$ to describe the set of densities that we may model. Usually, the parameters are estimated with MLE (expectation w.r.t. the empirical distribution)

$$\theta^* = \arg\max_\theta \mathbb{E}_{\mathbf{x}\sim p_\text{emp.}}\left[\log(p_\theta(\mathbf{x}))\right].$$

However, real data is rarely gaussian, laplacian, ... e.g., images. So the fact that in general we cannot solve for p_θ for a parametric function makes this task quite complicated.

So when using a *prescribed model* p_θ we have to

- ensure that p_θ defines a proper density:

$$\int p_\theta(\mathbf{x})\, d\mathbf{x} = 1.$$

- and to be able to evaluate the density p_θ at *various sample points* $\mathbf{x}$
  - this may be trivial for models such as exponential families (simple formulas)
  - but impractical for complex models (Markov networks, DNNs)

Now, the question is what strategies can we use for more complex models.

A typical example for an non-parametric and unnormalized model is kernel-density estimation.

D. (Kernel Density Estimator) Let $\mathbf{x}_1, \dots, \mathbf{x}_n$ be a sample, and k a kernel with bandwidth $h > 0$ then the estimator is defined as:

$$\overline{p}_\theta(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^n k_h(\mathbf{x}-\mathbf{x}_i) = \frac{1}{nh}\sum_{i=1}^n k\left(\frac{\mathbf{x}-\mathbf{x}_i}{h}\right).$$

The problem with this is that the rate of convergence is $\log(\log(n))$ - this is extremely painfully slow. This is just a guarantee in general when we know nothing about our density.

An alternative is to use *unnormalized models* (non-parametric: the number of parameters depends on dataset size). These then represent improper density functions:

$$\underbrace{\overline{p}_\theta(\mathbf{x})}_{\text{represented}} = \underbrace{c_\theta}_{\text{unknown}} \cdot \underbrace{p_\theta(\mathbf{x})}_{\text{normalized}}.$$

Finding the normalization constant c_θ might be really complicated, so we can only evaluate relative probabilities. Further, here we cannot use the log-likelihood, because scaling up $\overline{p}_\theta$ leads to an unbounded likelihood.

So the question still is: is there an alternative *estimation method* for unnormalized models?

What we do in practice is we do not look for the exact p_θ, but we look for properties of p_θ. In many cases these properties depend on our prior knowledge of p_θ. We need to understand what the problem is in order to put the prior knowledge into the model that we want to do. This was already important in supervised learning (e.g., CNNs with several layers for images), and is even more important in unsupervised learning. We have to do the same thing there without knowing what our final goal is.

Finally, Hyvarinen came up with the following idea in 2005. He asked himself whether there's an *operator* that we can apply to $\overline{p}_\theta$ that does not depend on normalization. - The answer was yes! Instead of estimating p_θ, we estimate $\log p_\theta$.

D. (Score Matching (Hyvarinen 2005))

$$\psi_\theta := \nabla_\mathbf{x} \log \overline{p}_\theta, \quad \psi = \nabla_\mathbf{x} \log p$$

Minimize the criterion

$$J(\theta) = \mathbb{E}\left[\|\psi_\theta - \psi\|\right]^2$$

or equivalently (by eliminating ψ by integration by parts)

$$J(\theta) = \mathbb{E}\left[\sum_i \partial_i \psi_{\theta,1} - \frac{1}{2}\psi_{\theta,i}^2\right].$$

This expectation can be approximated by sampling.

The main problem with this is that it assumes that the two normalization constants are the same!

— 16.2 — Factor Analysis

— 16.2.1 — Latent Variable Analysis

Latent Variable Analysis provides a generic way of defining probabilistic, i.e., *generative models* - the so-called *latent variable models*. They usually work as follows

1. Define a *latent variable* $\mathbf{z}$, with a distribution $p(\mathbf{z})$
2. Define *conditional models* for the observables $\mathbf{x}$ conditoned on the latent variable: $p(\mathbf{x} \mid \mathbf{z})$
3. Construct the *observed data model* by integrating/summing out the latent variables

$$p(\mathbf{x}) = \int p(\mathbf{z})p(\mathbf{x}\mid\mathbf{z})\,\mu(d\mathbf{z}) = \begin{cases}\int p(\mathbf{z})p(\mathbf{x}\mid\mathbf{z})\,d\mathbf{z}, & \mu = \text{Lebesgue}\\ \sum_\mathbf{z} p(\mathbf{z})p(\mathbf{x}\mid\mathbf{z}), & \mu = \text{counting}\end{cases}$$

Ex. (Gaussian Mixture Models GMMs)

$\mathbf{z} \in \{1, \dots, K\}$, $p(\mathbf{z})$ =mixing proportions

$p(\mathbf{x} \mid \mathbf{z})$: conditional densities (Gaussians for GMMs)

The idea of latent variable models is very similar to the one of autoencoders.

The idea is to have some

- $\mathbf{x} \in \mathbb{R}^d$
- and we want to embed it into $\mathbb{R}^k$ ($k \ll d$)
- so we'll use $\mathbf{z} \in \mathbb{R}^k$ (latent-space)
- and look at the conditional probabilities $p(\mathbf{x} \mid \mathbf{z})$ for some $\mathbf{x}$

Depending on whether $\mathbf{z}$ is continuous (e.g., as with PCA) or discrete random variable (e.g., GMMs) we'll be using the Lebesgue integral or counting to integrate/sum it out.

A typical approach to for latent variable models is *linear factor analysis*.

— Linear Factor Analysis

The idea of linear factor analysis is to explain the data through some low-dimensional isotropic gaussian. And the data is mapped/reconstructed through some linear map to/from the lower-dimensional space. The reconstruction is done via a linear map $\mathbf{W}$ and then different gaussian noises are added to the reconstructed vector (via $\boldsymbol{\eta}$).

So the *latent variable prior* is $\mathbf{z} \in \mathbb{R}^m$ where

$$\mathbf{z} \sim \mathcal{N}(\mathbf{o}, \mathbf{I})$$

and we have a linear *observation model* for $\mathbf{x} \in \mathbb{R}^n$

$$\mathbf{x} = \boldsymbol{\mu} + \underset{n\times m}{\mathbf{W}}\mathbf{z} + \boldsymbol{\eta}, \quad \boldsymbol{\eta} \sim \mathcal{N}(\mathbf{o}, \boldsymbol{\Sigma}), \quad \boldsymbol{\Sigma} := \operatorname{diag}(\sigma_1^2, \dots, \sigma_n^2)$$

Further note that

- $\boldsymbol{\mu}$ and $\mathbf{z}$ are *independent*
- typically $m \ll n$ (fewer factors than features)
- so few factors account for the depencencies between many observables
- The vector $\boldsymbol{\mu}$ is computed through MLE on the training set

$$\hat{\boldsymbol{\mu}} = \frac{1}{k}\sum_{i=1}^k \mathbf{x}_i$$

Usually we assume centered data, so $\boldsymbol{\mu} = \mathbf{o}$. Since $\boldsymbol{\mu}$ only complicates the notation and is actually easy to determine.

Recall, that in the previous part when we were doing autoencoders, the deviations that we were having for each of the components was the same one . So we wanted the error to be the same for each of the components. Now, with this model, with $\boldsymbol{\eta}$ we're allowing for additional flexibility for the error. There will be some components that we'll be able to explain with less error, and some with more.

So, $\mathbf{z}$ shoud capture everything that is important to explain the data, and $\boldsymbol{\eta}$ can be viewn as noise.

Altough we're assuming that here everything is gaussian, in general we may view $\mathbf{z}$ as a clustering mechanism, where $\mathbf{z}$ determines some cluster components that are selected.

T. The distribution of the *observation model* is

$$\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \mathbf{W}\mathbf{W}^\top + \boldsymbol{\Sigma}).$$

— Non-Identifiability of Factors

Now this seems to be nice, but again we have the *non-identifiability problem*, since there exist an infinite amount of solutions for any $\mathbf{W}$ that is a solution. Just let $\mathbf{Q}$ be an orthogonal $m \times m$-matrix. Then $\mathbf{WQ}$ is also a solution, because

$$(\mathbf{WQ})(\mathbf{WQ})^\top = MW\mathbf{Q}\mathbf{Q}^\top\mathbf{W}^\top = \mathbf{W}\mathbf{W}^\top.$$

The consequence of this is that the factors of the linear factor analysis are only identifieable up to some rotations/relfections in $\mathbb{R}^m$. Since we care what the factors in $\mathbf{z}$ mean we need to factor the *rotations* to get a better "interpretability" of the representation of the data in the latent space.

— Data Compression View

Now, how is the factor analysis related to data compression?

Encoder Step: Implicitly defined by posterior distribution

$$p(\mathbf{z}\mid\mathbf{x}) = \frac{p(\mathbf{x}\mid\mathbf{z})\,p(\mathbf{z})}{p(\mathbf{x})} \qquad \text{(Bayes)}$$

So,

$$p(\mathbf{z}\mid\mathbf{x}) = \mathcal{N}(\mathbf{z}; \boldsymbol{\mu}_{\mathbf{z}|\mathbf{x}}, \boldsymbol{\Sigma}_{\mathbf{z}|\mathbf{x}}),$$

where

$$\boldsymbol{\mu}_{\mathbf{z}|\mathbf{x}} = \mathbf{W}^\top\left(\mathbf{W}\mathbf{W}^\top + \boldsymbol{\Sigma}\right)^{-1}(\mathbf{x} - \boldsymbol{\mu})$$
$$\boldsymbol{\Sigma}_{\mathbf{z}|\mathbf{x}} = \mathbf{I} - \mathbf{W}^\top\left(\mathbf{W}\mathbf{W}^\top + \boldsymbol{\Sigma}\right)^{-1}\mathbf{W}$$

Further, if we assume that $\boldsymbol{\Sigma} = \sigma^2\mathbf{I}$ and we let $\sigma^2 \to 0$ (the reconstruction-error-variance for all the components is the same and we let the reconstruction error go to zero), then the following expression just reduces to the pseudo-invers:

$$\mathbf{W}^\top\left(\mathbf{W}\mathbf{W}^\top + \sigma^2\mathbf{I}\right)^{-1} \overset{\sigma^2\to 0}{\to} =: \mathbf{W}^\dagger \in \mathbb{R}^{m\times n}.$$

Consequently with the assumption of zero reconstruction error:

$$\boldsymbol{\mu}_{\mathbf{z}|\mathbf{x}} \to \mathbf{W}^\dagger(\mathbf{x} - \boldsymbol{\mu})$$
$$\boldsymbol{\Sigma}_{\mathbf{z}|\mathbf{x}} \to \mathbf{0}$$

So, if we know $\mathbf{W}$ and $\boldsymbol{\Sigma}$ is assumed to be isotropic with the error going to zero the encoding distribution gets very easy to compute.

— Maximum Likelihood Estimation

Now, how do we estimate $\mathbf{W}$ and $\boldsymbol{\Sigma}$? The idea is fairly simple.

Let's assume that $\mathbf{x}_1, \dots, \mathbf{x}_k \overset{\text{i.i.d.}}{\sim} \mathcal{N}(\mathbf{0}, \mathbf{A})$. Further let's define the data matrix $\mathbf{X}$ as

$$\mathbf{X} = \begin{bmatrix} | & | & & | \\ \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_k \\ | & | & & | \end{bmatrix}$$

and the empirical co-variance matrix as

$$\mathbf{S} := \frac{1}{k}\sum_{i=1}^k \mathbf{x}_i\mathbf{x}_i^\top = \frac{1}{k}\mathbf{X}\mathbf{X}^\top.$$

Then, the log-likelihood of the data $\mathbf{X}$, given $\mathbf{A}$ can be written as:

$$\log\left(P(\mathbf{X};\mathbf{A})\right) = -\frac{k}{2}\left(\operatorname{Tr}\left(\mathbf{S}\mathbf{A}^{-1}\right) - \log\left(\det(\mathbf{A})\right)\right) + \underset{\text{i.T. of }\mathbf{A}}{\text{const.}}$$

Note: this can be verified by using the definition of $\mathbf{S}$, the cyclic property of the trace, and then just write down the matrix-product as block-matrices and see what is the diagonal of the resulting matrix.

Now, let's compute the matrix gradients w.r.t. $\mathbf{A}$ to know the equations that we need to compute the maximum likelihood:

$$\nabla_\mathbf{A}\operatorname{Tr}\left(\mathbf{S}\mathbf{A}^{-1}\right) = -\mathbf{A}^{-1}\mathbf{S}\mathbf{A}^{-1}$$
$$\nabla_\mathbf{A}\log\left(\det(\mathbf{A})\right) = \mathbf{A}^{-1}$$

Now, setting the gradient of the log-likelihood to zero gives us the following condition:

$$\nabla_\mathbf{A}\log\left(P(\mathbf{X};\mathbf{A})\right) \overset{!}{=} 0 \iff \mathbf{S}\mathbf{A}^{-1} = \mathbf{I}.$$

So, the MLE for $\mathbf{A}$ is just $\mathbf{A} = \mathbf{S}$.

But recall, that what we want is not $\mathbf{A}$, but we want $\mathbf{W}$ and $\boldsymbol{\Sigma}$.

However, we know that $\mathbf{A}$ is just the empirical covariance matrix, and $\mathbf{W}$ will be the mapping to the low-dimensional space and $\boldsymbol{\Sigma}$ is the reconstruction error.

$$\mathbf{A} = \mathbf{W}\mathbf{W}^\top + \boldsymbol{\Sigma}$$

Now, using the chain rule we get:

$$\nabla_\mathbf{W}\mathbf{A} = 2\mathbf{W}$$
$$\nabla_{\boldsymbol{\Sigma}}\mathbf{A} = \mathbf{I}$$

This gives us the following stationary condition for $\mathbf{W}$ given $\boldsymbol{\Sigma}$

$$\mathbf{S}(\boldsymbol{\Sigma} + \mathbf{W}\mathbf{W}^\top)^{-1}\mathbf{W} = \mathbf{W}.$$

In general, finding $\mathbf{W}$ is not easy. However, a special case is if we assume that $\boldsymbol{\Sigma} = \sigma^2\mathbf{I}$ (isotropic reconstruction error/noise) and $\mathbf{W}^\top\mathbf{W} = \operatorname{diag}(\rho_i^2)$, then by Woodbury's formula we have simplifies to:

$$\left(\sigma^2\mathbf{I}_n + \mathbf{W}\mathbf{W}^\top\right)^{-1}\mathbf{W} = \mathbf{W}\operatorname{diag}\left(\frac{1}{\sigma^2 + \rho_i^2}\right).$$

Putting this back into the stationary condition, for each column $\mathbf{w}_i$ of $\mathbf{W}$ we get an eigenvector equation:

$$\mathbf{S}\mathbf{w}_i = (\sigma^2 + \rho_i^2)\mathbf{w}_i \qquad \mathbf{SW} = \operatorname{diag}(\boldsymbol{\lambda})\mathbf{W}.$$

Then, if $\mathbf{u}_i$ is the i-th eigenvector of $\mathbf{S}$, then

$$\mathbf{w}_i = \rho_i\mathbf{u}_i, \qquad \rho_i^2 = \max\left\{0, \lambda_i - \sigma^2\right\}.$$

This gives us the *probabilistic interpretation PCA* and showed us how we can derive the PCA as a special case for $\sigma^2 \to 0$ (Tipping & Bishop, 1999).

— Refresher on MGFs and Gaussians

D. (Moment Generating Function (MGF)) The MGF $M_\boldsymbol{X}$ of a random vector $\boldsymbol{X} \in \mathbb{R}^n$ is defined as

$$M_\boldsymbol{X}\colon \mathbb{R}^n \to \mathbb{R}$$
$$\mathbf{t} \mapsto \mathbb{E}_\boldsymbol{X}\left[e^{\mathbf{t}^\top\boldsymbol{X}}\right].$$

The reason $M_\boldsymbol{X}$ is called *moment* generating function is because it represents the *moments* of $\mathbf{x}$ in the following way: Let $k_1, \dots, k_n \in \mathbb{N}$, then

$$\mathbb{E}_\boldsymbol{X}\left[x_1^{k_1}x_2^{k_2}\cdots x_n^{k_n}\right] = \frac{\partial k}{\partial t_1^{k_1}\partial t_2^{k_2}\cdots\partial t_n^{k_n}}M_\boldsymbol{X}\Big|_{\mathbf{t}=\mathbf{o}}.$$

T. (Uniqueness Theorem) If $M_\boldsymbol{X}$ and $M_\boldsymbol{Y}$ exist for the RVs $\boldsymbol{X}$ and $\boldsymbol{Y}$ and $M_\boldsymbol{X} = M_\boldsymbol{Y}$ then $\forall\mathbf{t}\colon P(\boldsymbol{X} = \mathbf{t}) = P(\boldsymbol{Y} = \mathbf{t})$ (distributions are the same).

Now, every distribution has its unique kind of MGF form. Hence, MGFs can be very useful to deal with *sums of i.i.d. random variables*:

T. If $\boldsymbol{X}, \boldsymbol{Y}$ are i.i.d. then $M_{\boldsymbol{X}+\boldsymbol{Y}} = M_\boldsymbol{X} \cdot M_\boldsymbol{Y}$.

— 16.3 — Latent Variable Models

— 16.3.1 — DeFinetti's Theorem

There's another way of looking at latent variable models which is by the DeFinetty exchangeable theorem from the 1930s. This is one of the foundations of Bayesian probability (although there is nothing Bayesian in this theorem).

T. (DeFinetti's Theorem) For *exchangeable* data (order of dataset doesn't matter and they come from the same distribution), we can decompose the data by a *latent variable model*

$$P(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N) = \int\prod_{i=1}^N p_\theta(\mathbf{x}_i \mid \mathbf{z})\,p_\theta(\mathbf{z})\,d\mathbf{z}.$$

We *expect* that those hidden variables are: interpretable and actionable and even show causal relations.

Later we'll put our Bayesian priors into the distributions $P(\mathbf{z})$ and we then hope that the latent structure will tell us something about the data that we didn't know before.

— 16.3.2 — Latent Variable Models

Classically we define complex models via the *marginalization* of a *latent variable model*

$$p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{x}, \mathbf{z})\,d\mathbf{z} \text{ or } p_\theta(\mathbf{x}) = \sum_\mathbf{z} p_\theta(\mathbf{x}, \mathbf{z})$$

— 16.3.3 — Dimensionality Reduction

One of the recurring things that we see in all of these models is *dimensionality reduction.* So we have that

$$\mathbf{X} = f(\mathbf{ZB})$$

where

- $\mathbf{X}$ is $N \times D$,
- $\mathbf{Z}$ is $N \times K$,
- $\mathbf{B}$ is $K \times D$, and
- $K \ll D$.

So we have the data $\mathbf{X}$ that we're trying to understand. We'll try to understand this data by a tall matrix $\mathbf{Z}$ and a fat matrix $\mathbf{B}$. The tall matrix are the latent factors that we've talking about (how do we summarize the information of each sample). And the matrix $\mathbf{B}$ is telling us how we can recover the original data from the summary. Most of the unsupervised algorithms can be captured in this general framework.

Depending on $f(\cdot)$, $\mathbf{Z}$ and $\mathbf{B}$, we arrive at different models:

- Principal Component Analysis / Factor Analysis (f linear)
- Nonnegative Matrix Factorization (f "psomolu" or Bernoulli model, and bot $\mathbf{Z}$ and $\mathbf{B}$ have to be a nonnegative matrix).
- LLE/Isomap/GPLVM (here we also try to do PCA or Factor analysis with nonlinear components (with p.w. linear components))
- Restricted Boltzmann Machine (the idea is that $\mathbf{Z}$ is discrete)
- Dirichlet Process (aka Chinese Restaurant Process)
- Beta Process (aka Indian Buffet Process)
- Implicit Models (e.g., Generative Adversarial Networks)(here all the information is moved to the function f instead of computing the matrices $\mathbf{B}$ and $\mathbf{C}$)

— 16.3.4 — Implicit Models

Here we develop statistical models via: *generating stochastic mechanism* or *simulation process.*

Deep implicit models

- latent code $\mathbf{z} \in \mathbb{R}^d$, $\mathbf{z} \sim \pi(\mathbf{z})$, e.g. $\pi(\mathbf{z}) = \mathcal{N}(\mathbf{o}, \mathbf{I})$
- parametrized mechanism: $F_\theta\colon \mathbb{R}^d \to \mathbb{R}^m$
- induced distribution $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{x} \sim p_\theta(\mathbf{x})$
- sampling is easy: random vector + forward propagation.

17 Chain-Rule and Jacobians for Tensors

D. (k-Dimensional Tensor) $\mathsf{T} \in \mathbb{R}^{d_1\times d_2\times\cdots\times d_k}$

D. (Tensor Multiplication)

$$\underset{\in\mathbb{R}^{(a+c)}}{\mathsf{T}} = \underset{\in\mathbb{R}^{(a+b)}}{\mathsf{P}} \times_b \underset{\in\mathbb{R}^{(b+c)}}{\mathsf{Q}}$$

$$\underset{r_1\times\cdots\times r_a\times s_1\times\cdots\times s_c}{\mathsf{T}} = \underset{r_1\times\cdots\times r_a\times s_1\times\cdots\times s_b}{\mathsf{P}} \times_{s_1,\dots,s_b} \underset{s_1\times\cdots\times s_b\times t_1\times\cdots\times t_c}{\mathsf{Q}}$$

where each entry of T is computed as follows: $T_{i_1,\dots,i_a,k_1,\dots,k_c} := \sum_{j_1,\dots,j_b} P_{i_1,\dots,i_a,j_1,\dots,j_b}Q_{j_1,\dots,j_b,k_1,\dots,k_c}$

Note that this is just the sum of the multiplications of two numbers which are in corresponding locations in P and Q. Essentially, it's the dot product across the dimensions $s_1, \dots, s_b$.

Note how this tensor-tensor-multiplication is isomorphic to some matrix-matrix product:

$$T_{\underbrace{i_1,\dots,i_a}_{\text{“i”}},\underbrace{k_1,\dots,k_c}_{\text{“k”}}} := \sum_{\underbrace{j_1,\dots,j_b}_{\text{“j”}}} P_{\underbrace{i_1,\dots,i_a}_{\text{“i”}},\underbrace{j_1,\dots,j_b}_{\text{“j”}}}Q_{\underbrace{j_1,\dots,j_b}_{\text{“j”}},\underbrace{k_1,\dots,k_c}_{\text{“k”}}}$$

T. (Tensor Chain Rule)

$y(W)\colon \mathbb{R}^{d_1\times d_2} \to \mathbb{R}^{d_3\times d_4}$, $L(y)\colon \mathbb{R}^{d_3\times d_4} \to \mathbb{R}$

$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial W} \times_{d_3,d_4} \frac{\partial y}{\partial W}$

then we have: $T_{i,j,k,l} = \frac{\partial y_{i,j}}{\partial W_{k,l}}$

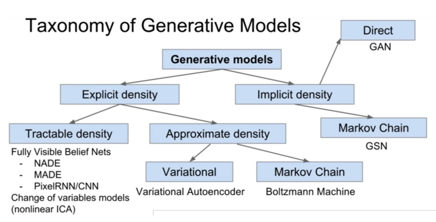
18 Generative Models

In unsupervised learning our goal is to learn some underlying hidden *structure* of the data (clustering, dimensionality reduction, feature learning, density estimation). Now, generative modeling has the following goal:

Goal: given data D, generate new samples from the same distribution p_data. We want to learn p_model similar to p_data.

The nice thing is that the traning data is cheap, as we need no labels. However, it's a hard task.

In some way or another, any generative model has to cope with density estimation (which is the hard task). This problem is tackled in different ways by the several flavours of generative models:



- **explicit density estimation:** explicitly define and solve for $p_\text{model}(\mathbf{x})$
  - **tractable** we can comute $p_\text{model}(\mathbf{x})$
  - **approximate** we approximate $p_\text{model}(\mathbf{x})$ in some way
- **implicit density estimation:** learn a model that can sample from $p_\text{model}(\mathbf{x})$ without explicitly defining it.

— 18.1 — Variational Autoencoders (VAEs)

Relation to Autoencoders

Recall, that with autoencoders, we had defined a concatenation of two differentiable (non-linear) mappings $\mathbf{x} \overset{E}{\mapsto} \mathbf{z} \overset{D}{\mapsto} \widehat{\mathbf{x}}$ (an encoder E and a decoder D) and trained it with the following loss $\|\mathbf{x} - \widehat{\mathbf{x}}\|_2^2$ (approximating identity function) in order to learn some compressed representation $\mathbf{z}$ of $\mathbf{x}$ which just contains the essence of $\mathbf{x}$ according to some meaningful feature-dimensions.

Further, we could then use E to bootstrap a classification model, by first applying E, and then a classification network C and fine-tuning them jointly on the cross-entropy loss.

So the lower-dimensional features $\mathbf{z}$ capture the factors of variation in the data. And we can reconstruct an $\mathbf{x}$ from its compressed representation $\mathbf{x}$.

Now the question is can we us a similar kind of setup to use new images?

VAEs define an *intractable* density function $p_\text{model}(\mathbf{x})$ with a latent $\mathbf{z}$. Having this latent variable allows us to build a network similar to an autoencoder. A tractable lower bound for the intractable density function is then derived and optimized.

$$p_\text{model}(\mathbf{x}) := p_\theta(\mathbf{x}) = \int_\mathbf{z} \underbrace{p_\theta(\mathbf{z})}_{\substack{\text{“latent representation”}\\ \text{sample from prior}\\ \text{(simple, e.g., MV - Gaussian)}}} \overbrace{p_\theta(\mathbf{x}\mid\mathbf{z})}^{\substack{\text{“Decoder”}\\ \text{sample from conditional}\\ \text{(complex NN)}}}\,d\mathbf{z}$$

With VAEs we assume that our data is generated from some underlying unobserved representation $\mathbf{z}$. We first sample $\mathbf{z}$ and then generate some $\mathbf{x}$ from the conditional distribution. Now, we just have to learn the parameters θ that maximize the likelihood of the training data. Unfortunately, we cannot optimize the likelihood of our model for the data directly (as it's intractable).

$\int_\mathcal{Z} p_\theta(\mathbf{z})p_\theta(\mathbf{x}\mid\mathbf{z})\,d\mathbf{z}$ intractable!, tractable

Note that different factorizations of the distributions would also intractable

$p_\theta(\mathbf{z}\mid\mathbf{x}) = \frac{p_\theta(\mathbf{x}\mid\mathbf{z})p_\theta(\mathbf{z})}{p_\theta(\mathbf{x})}$

Now, the solution to this is that in addition to the decoder network modeling $p_\theta(\mathbf{x}\mid\mathbf{z})$, we define an additional encoder network $q_\phi(\mathbf{z}\mid\mathbf{x})$ that approximates $p_\theta(\mathbf{z}\mid\mathbf{x})$.

$p_\phi(\mathbf{z}\mid\mathbf{x}) \approx p_\theta(\mathbf{z}\mid\mathbf{x}) = \frac{p_\theta(\mathbf{x}\mid\mathbf{z})p_\theta(\mathbf{z})}{p_\theta(\mathbf{x})}$

This will allow us to derive a lower bound on the data likelihood that is tractable, which we can optimize.

A common way to define these distributions is as follows: We assume the latent variable to follow a multivariate gaussian (assumed to be a reasonable prior for latent attributes).

Prior: $\mathbf{z} \sim \mathcal{N}(\mathbf{o}, \mathbf{I})$.

Enc. Netw. E: models $q_\phi(\mathbf{z}\mid\mathbf{x})$ with params ϕ and maps $\mathbf{x} \overset{E}{\mapsto} (\boldsymbol{\mu}_{\mathbf{z}|\mathbf{x}}, \boldsymbol{\Sigma}_{\mathbf{z}|\mathbf{x}})$

Dec. Netw. D: models $p_\theta(\mathbf{x}\mid\mathbf{z})$ with params θ and maps $\mathbf{z} \overset{D}{\mapsto} (\boldsymbol{\mu}_{\mathbf{x}|\mathbf{z}}, \boldsymbol{\Sigma}_{\mathbf{x}|\mathbf{z}})$

Note that we use both *diagonal* covariance matrices for $\boldsymbol{\Sigma}_{\mathbf{z}|\mathbf{x}}$ and $\boldsymbol{\Sigma}_{\mathbf{x}|\mathbf{z}}$. So the output of both networks are just two vectors (one for mean, other for diagonal).

Com. Encoder and decoder networks are also called "recognition/inference" and "generation" networks.

Now, equipped with our encoder and decoder networks, we can rewrite the data (log) likelihood as follows (note that we omit the product for all points - you'd just have to put a sum over all the instances in front of everything)

$$\begin{aligned}\log(p_\theta(\mathbf{x})) &= \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}\mid\mathbf{x})}\left[\log(p_\theta(\mathbf{x}))\right] \quad (\log(p_\theta(\mathbf{x})) \text{ does not depend on } \mathbf{z})\\ &= \mathbb{E}_\mathbf{z}\left[\log\left(\frac{p_\theta(\mathbf{x}\mid\mathbf{z})\,p_\theta(\mathbf{z})}{p_\theta(\mathbf{z}\mid\mathbf{x})}\right)\right] \quad \text{(Bayes Rule)}\\ &= \mathbb{E}_\mathbf{z}\left[\log\left(\frac{p_\theta(\mathbf{x}\mid\mathbf{z})\,p_\theta(\mathbf{z})}{p_\theta(\mathbf{z}\mid\mathbf{x})}\frac{q_\phi(\mathbf{z}\mid\mathbf{x})}{q_\phi(\mathbf{z}\mid\mathbf{x})}\right)\right] \quad \text{(Muliply by 1)}\\ &= \mathbb{E}_\mathbf{z}\left[\log(p_\theta(\mathbf{x}\mid\mathbf{z}))\right] - \mathbb{E}_\mathbf{z}\left[\log\left(\frac{q_\phi(\mathbf{z}\mid\mathbf{x})}{p_\theta(\mathbf{z})}\right)\right] + \mathbb{E}_\mathbf{z}\left[\log\left(\frac{q_\phi(\mathbf{z}\mid\mathbf{x})}{p_\theta(\mathbf{z}\mid\mathbf{x})}\right)\right]\\ &= \underbrace{\underbrace{\mathbb{E}_\mathbf{z}\left[\log(p_\theta(\mathbf{x}\mid\mathbf{z}))\right]}_{(1)} - \underbrace{KL(q_\phi(\mathbf{z}\mid\mathbf{x}), p_\theta(\mathbf{z}))}_{(2)}}_{\mathcal{L}(\mathbf{x},\theta,\phi)} + \underbrace{KL(q_\phi(\mathbf{z}\mid\mathbf{x}), p_\theta(\mathbf{z}\mid\mathbf{x}))}_{(3)\geq 0}\end{aligned}$$

(1) Decoder network gives us $p_\theta(\mathbf{x}\mid\mathbf{z})$ and we can compute estimates of this term through sampling. (Sampling differentiable through *reparametrization trick!*)
This term ensures that we reconstruct the data well.

(2) This KL term (between Gaussians for encoder and $\mathbf{z}$ prior) has a nice closed-form solution.
This term ensures that the approximate posterior distribution is close to prior.

(3) $p_\theta(\mathbf{z}\mid\mathbf{x})$ is intractable (as seen earlier). But we know that the KL-divergence is ≥ 0.

Now what we have is a *tractable lower bound* $\mathcal{L}$ (so-called variational lower bound, or evidence lower bound "ELBO") for the likelihood

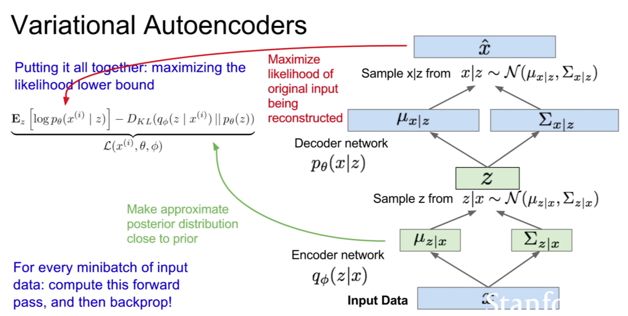
$\mathcal{L}(\mathbf{x}, \theta, \phi) \leq \log(p_\theta(\mathbf{x}))$

and we can take its gradient to optimize:

$(\theta^*, \phi^*) = \arg\max_{(\theta,\phi)}\sum_{i=1}^n \mathcal{L}(\mathbf{x}^{(i)}, \theta, \phi)$.

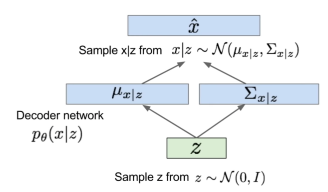
Reparametrization Trick

Forward Pass and Backpropagation



So, first we do the whole backpropagation. And then we just compute the updates to the parameters θ and ϕ via backpropagation.

Generating Data Here we just sample from the prior, and pass it through the decoder network to get the posterior distribution's parameters, then we sample from that one.



— 18.2 — Deep Latent Gaussian Models

— 18.3 — Generative Adversarial Networks (GANs)

GANs do not try to model a density function but directly aim to build a function to generated data (implicit generative method). The whole optimization motivated by a game-theoretic approach in a 2-player game. GANs sample from a simple random noise distribution and try to learn a transformation (via a NN) to a data distribution.

D. (Discriminator D) must be a differentiable function parametrized by θ_d

$\mathbf{x} \overset{D}{\mapsto} P(\mathbf{x} \text{ comes from true data distr.}) \in [0, 1]$

D. (Generator G) must be a differentiable function parametrized by θ_g

$\mathbf{z} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \overset{G}{\mapsto} \mathbf{x}$ (one falsified data sample)

(one may also use another prior)

$\mathbf{z}$ is sampled from the prior distr. over latent vars (source of randomness)

D tries to make $D(G(\mathbf{z}))$ near 0 (for fake data)

D tries to make $D(\mathbf{x})$ near 1 ($\mathbf{x}$ sampled from true data)

G tries to make $D(G(\mathbf{z}))$ near 1

Com. In some sense G implicitly tries to make $D(\mathbf{x})$ near 0 (since it uses the negative loss of D) see Minimax game VS Non-Saturating game.

Minimax Game: Both D and G try to minimize and maximize the same value function:

$$\begin{aligned}V(G, D) &= \min_G\max_D \mathcal{R}^{(D)}\\ &= \min_{\theta_g}\max_{\theta_d}\mathbb{E}_{\mathbf{x}\sim p_\text{data}}\left[\log(D_{\theta_d}(\mathbf{x}))\right] + \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}\left[\log(1 - D_{\theta_d}(G_{\theta_g}(\mathbf{z})))\right]\end{aligned}$$

$\mathcal{R}^{(D)} = -\frac{1}{2}\mathbb{E}_{\mathbf{x}\sim p_\text{data}}\left[\log(D(\mathbf{x}))\right] - \frac{1}{2}\underbrace{\mathbb{E}_{\mathbf{z}\sim\mathcal{N}(\boldsymbol{\mu},\boldsymbol{\Sigma})}\left[\log(1 - D(G(\mathbf{z})))\right]}_{=\mathbb{E}_{\mathbf{x}\sim p_\text{generator}}[\log(1-D(\mathbf{x}))]}$

$\mathcal{R}^{(G)} = -\mathcal{R}^{(D)}$

- The loss $\mathcal{R}^{(D)}$ is simply the cross-entropy between D's predictions and the correct labels in the binary classification task (real/fake)
- The equilibrium of this game is saddle point of the discriminator loss
- If we look for this equilibrium the whole procedure resembles minimizing the Jensen-Shannon divergence between the true data distribution and the generator distribution.
- So G minimizes the log-probability of D being correct

What is the solution $D(\mathbf{x})$ in terms of p_data and $p_\text{generator}$ at the equilibrium?

In the equilibrium it must hold that the gradient of the discriminator is zero, because the discriminator otherwise would improve (thus change) itself.

$\frac{\partial\mathcal{R}^{(D)}}{\partial D(\mathbf{x})} = -\frac{1}{2}\mathbb{E}_{\mathbf{x}\sim p_\text{data}}\left[\frac{1}{D(\mathbf{x})}\right] + \frac{1}{2}\mathbb{E}_{\mathbf{x}\sim p_\text{generator}}\left[\frac{1}{1-D(\mathbf{x})}\right] \overset{!}{=} 0$

$\iff \mathbb{E}_{\mathbf{x}\sim p_\text{data}}\left[\frac{1}{D(\mathbf{x})}\right] = \mathbb{E}_{\mathbf{x}\sim p_\text{generator}}\left[\frac{1}{1-D(\mathbf{x})}\right]$

$\iff \int_\mathcal{X} p_\text{data}(\mathbf{x})\frac{1}{D(\mathbf{x})}\,d\mathbf{x} = \int_\mathcal{X} p_\text{generator}(\mathbf{x})\frac{1}{1-D(\mathbf{x})}\,d\mathbf{x}$

Recall: we can get rid of the integral as it is just a function operator. Using the inverse of the operator, we can get rid of it and the solution constraints on the optimal $D(\mathbf{x})$ still remain the same.

$\iff p_\text{data}(\mathbf{x})\frac{1}{D(\mathbf{x})} = p_\text{generator}(\mathbf{x})\frac{1}{1-D(\mathbf{x})}$

Then we get the following stationarity condition: The optimal $D(\mathbf{x})$ for any $p_\text{data}(\mathbf{x})$ and $p_\text{generator}(\mathbf{x})$ is always

$D(\mathbf{x}) = \frac{p_\text{data}(\mathbf{x})}{p_\text{data}(\mathbf{x}) + p_\text{generator}(\mathbf{x})}$

Note that by stationarity condition we mean that this must hold in the Nash equilibrium (where both players stop adapting themselves).

Estimating this ratio using supervised learning is the key approximation mechanism used by GANs.

What assumptions are needed to obtain this solution?

We need to assume that both densities are nonzero everywhere. If we don't make this assumption then there's this issue that the discriminator's input space might never be sampled during its training process. Then these points would have an undefined behaviour since they're never trained.

Training Procedure: Use SGD-like algorithm of choice (ADAM) on two minibatches simultaneously. At each iteration, we choose:

- a minibatch of true data samples
- a minibatch of noise vectors to produce minibatch generated samples

Then compute both losses and perform gradient updates.

$\theta_d^{t+1} \leftarrow \theta_d^t - \eta_t\nabla_{\theta_d}\mathcal{R}^{(D)}(\theta_d)$

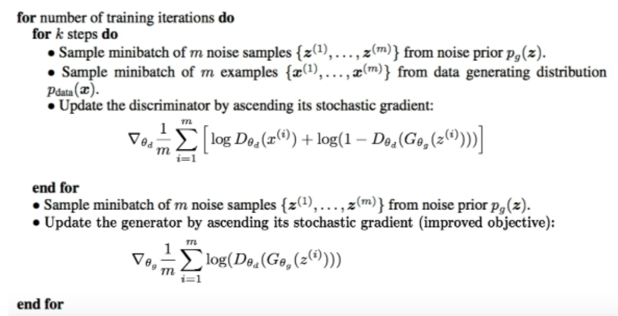
$\theta_g^{t+1} \leftarrow \theta_g^t - \eta_t\nabla_{\theta_g}\mathcal{R}^{(G)}(\theta_g)$

Optional: run $k \geq 1$ update steps of D for every iteration (and only 1 update step for G).

So we alternate between

- **Gradient ascent** on D
  $\max_{\theta_d}\mathbb{E}_{\mathbf{x}\sim p_\text{data}}\left[\log(D_{\theta_d}(\mathbf{x}))\right] + \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}\left[\log(1 - D_{\theta_d}(G_{\theta_g}(\mathbf{z})))\right]$
- **Gradient descent** on G
  $\min_{\theta_g}\mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}\left[\log(1 - D_{\theta_d}(G_{\theta_g}(\mathbf{z})))\right]$

Note that this training algorithm uses a heuristically motivated loss (that is a bit different) for the generator to have better gradients when the discriminator is good:



for number of training iterations **do**
  for k steps **do**
  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \dots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_\text{data}(\boldsymbol{x})$.
  - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d}\frac{1}{m}\sum_{i=1}^m\left[\log D_{\theta_d}(x^{(i)}) + \log(1 - D_{\theta_d}(G_{\theta_g}(z^{(i)})))\right]$$

  end for
  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \dots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by ascending its stochastic gradient (improved objective):

$$\nabla_{\theta_g}\frac{1}{m}\sum_{i=1}^m\log(D_{\theta_d}(G_{\theta_g}(z^{(i)})))$$

end for