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

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

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

$\mathrm{Var}[X] = \mathbb{E}[(X-\mathbb{E}[X])^2] = \int_{\mathcal{X}} (t-\mathbb{E}[X])^2 f_X(t)\,dt = \mathbb{E}[X^2] - \mathbb{E}[X]^2$

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

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

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

"$\mathbf{X}^2 = \mathbf{X}\mathbf{X}^T$" ≥ 0 ((symmetric) positive semidefinite)

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

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

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

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

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

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

Empirical CDF: $\hat{F}_n(t) = \frac{1}{n}\sum_{i=1}^n \mathbb{1}_{\{X_i \leq t\}}$

Empirical PDF: $\hat{f}_n(t) = \frac{1}{n}\sum_{i=1}^n \delta(t - X_i)$ (continuous)

Empirical PDF: $\hat{p}_n(t) = \frac{1}{n}\,|x = t|\; x \in D$ (discrete)

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

$Be(p)$: $\quad pe^t + (1-p)$

$Bin(n,p)$: $(pe^t + (1-p))^n$

$Pois(\lambda)$: $\quad e^{\lambda(e^t-1)}$

$\mathcal{N}(\mu,\sigma)$: $\quad \exp\left(\mu t + \frac{1}{2}\sigma^2 t^2\right)$

$Gam(\alpha,\beta)$: $\left(\frac{1}{a-\beta t}^{\alpha}\right)$ for $t < 1/\beta$

T. If $X_1,\dots,X_n$ are ind. 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 ind., then the p.d.f. of $Z = X+Y$ is the conv. of the p.d.f. of X and Y: $f_Z(z) = \int_{\mathbb{R}} f_X(t) f_Y(z-t)\,dt = \int_{\mathbb{R}} f_X(z-t) f_Y(t)\,dt$

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

$\mathcal{N}(\mathbf{x};\boldsymbol{\mu},\boldsymbol{\Sigma}^{-1}) \propto \exp\left(\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\mathsf{T}\boldsymbol{\Sigma}(\mathbf{x}-\boldsymbol{\mu})\right)$

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

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

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

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

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)

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

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

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

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

(Fundamental Theorem of Calculs)

$f(\mathbf{y}) - f(\mathbf{x}) = \int_{\gamma[\mathbf{x},\mathbf{y}]} \nabla f(\boldsymbol{\tau}) \cdot d\boldsymbol{\tau} = \int_{t=0}^1 \nabla f(\boldsymbol{\gamma}(t))^\mathsf{T}\boldsymbol{\gamma}'(t)\,dt$

$f(\mathbf{y}) - f(\mathbf{x}) = \int_0^1 \nabla f((1-t)\mathbf{x} + t\mathbf{y})^\mathsf{T}(\mathbf{y}-\mathbf{x})\,dt$

Com. Create a path $\boldsymbol{\gamma}$ from $\mathbf{x}$ to $\mathbf{y}$ and integrate the dot product of the gradient of the function-values at the path with the derivative of the path.

D. (Saddle Points etc.)

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

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

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

D. (Lagrangian Formulation) of $f(x,y)$ s.t. $g(x,y) = c$

$\mathcal{L}(x,y,\gamma) = f(x,y) - \gamma(g(x,y) - c)$

3 Linear Algebra

T. (Sylvester Criterion) A $d \times d$ matrix is positive semi-definite if and only if all the upper left $k \times k$ for $k = 1,\dots,d$ 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 $\mathsf{Tr}(\mathbf{A}) = \sum_{i=1}^n a_{ii}$.

4 Derivatives

What is correct???

4.1 Scalar-by-Vector

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

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

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

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

$\frac{\partial}{\partial \mathbf{x}}\left[\mathbf{a}^\mathsf{T}\mathbf{x}\right] = \frac{\partial}{\partial \mathbf{x}}\left[\mathbf{x}^\mathsf{T}\mathbf{a}\right] = \mathbf{a}$

$\frac{\partial}{\partial \mathbf{x}}\left[\mathbf{x}^\mathsf{T}\mathbf{x}\right] = 2\mathbf{x}$

$\frac{\partial}{\partial \mathbf{x}}\left[\mathbf{b}^\mathsf{T}\mathbf{A}\mathbf{x}\right] = \mathbf{A}^\mathsf{T}\mathbf{b}$

$\frac{\partial}{\partial \mathbf{x}}\left[\mathbf{x}^\mathsf{T}\mathbf{A}\mathbf{x}\right] = (\mathbf{A} + \mathbf{A}^\mathsf{T})\mathbf{x}$

$\frac{\partial}{\partial \mathbf{x}}\left[\mathbf{a}^\mathsf{T}\mathbf{f}(\mathbf{x})\right] = \frac{\partial \mathbf{f}}{\partial \mathbf{x}}\mathbf{a}$

$\frac{\partial}{\partial \mathbf{x}}\left[\mathbf{a}^\mathsf{T}\mathbf{x}\mathbf{x}^\mathsf{T}\mathbf{b}\right] = (\mathbf{a}\mathbf{b}^\mathsf{T} + \mathbf{b}\mathbf{a}^\mathsf{T})\mathbf{x}$

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

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

4.2 Vector-by-Vector

$\mathbf{A}, \mathbf{C}, \mathbf{D}, \mathbf{a}, \mathbf{b}, \mathbf{e}$ not a function of $\mathbf{x}$,

$\mathbf{f} = \mathbf{f}(\mathbf{x})$, $\mathbf{g} = \mathbf{g}(\mathbf{x})$, $\mathbf{h} = \mathbf{h}(\mathbf{x})$, $u = u(x)$, $v = v(x)$

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

$\frac{\partial}{\partial \mathbf{x}}[\mathbf{x} \odot \mathbf{a}] = \mathrm{diag}(\mathbf{a})$

$\frac{\partial}{\partial \mathbf{x}}[\mathbf{a}] = \mathbf{0}$

$\frac{\partial}{\partial \mathbf{x}}[\mathbf{x}] = \mathbf{I}$

$\frac{\partial}{\partial \mathbf{x}}[\mathbf{A}\mathbf{x}] = \mathbf{A}$

$\frac{\partial}{\partial \mathbf{x}}\left[\mathbf{x}^\mathsf{T}\mathbf{A}\right] = \mathbf{A}^\mathsf{T}$

$\frac{\partial}{\partial \mathbf{x}}[a\mathbf{f}(\mathbf{x})] = a\frac{\partial \mathbf{f}(\mathbf{x})}{\partial \mathbf{x}} = a\mathbf{J}_f$

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

$\frac{\partial}{\partial \mathbf{x}}[\mathbf{f}(\mathbf{g}(\mathbf{x}))] = \frac{\partial \mathbf{f}}{\partial \mathbf{g}}\frac{\partial \mathbf{g}}{\partial \mathbf{x}} = \mathbf{J}_f(\mathbf{g})\mathbf{J}_g(\mathbf{x})$

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

4.3 Scalar-by-Matrix

$\frac{\partial}{\partial \mathbf{X}}\left[\mathbf{a}^\mathsf{T}\mathbf{X}\mathbf{b}\right] = \mathbf{a}\mathbf{b}^\mathsf{T}$

$\frac{\partial}{\partial \mathbf{X}}\left[\mathbf{a}^\mathsf{T}\mathbf{X}^\mathsf{T}\mathbf{b}\right] = \mathbf{b}\mathbf{a}^\mathsf{T}$

$\frac{\partial}{\partial \mathbf{X}}\left[\mathbf{a}^\mathsf{T}\mathbf{X}\mathbf{a}\right] = \frac{\partial}{\partial \mathbf{X}}\left[\mathbf{a}^\mathsf{T}\mathbf{X}^\mathsf{T}\mathbf{a}\right] = \mathbf{a}\mathbf{a}^\mathsf{T}$

$\frac{\partial}{\partial \mathbf{X}}\left[\mathbf{a}^\mathsf{T}\mathbf{X}^\mathsf{T}\mathbf{X}\mathbf{b}\right] = \mathbf{X}(\mathbf{a}\mathbf{b}^\mathsf{T} + \mathbf{b}\mathbf{a}^\mathsf{T})$

$\frac{\partial}{\partial \mathbf{X}}[\mathsf{Tr}(\mathbf{X})] = \mathbf{I}$

$\frac{\partial}{\partial \mathbf{X}}[\mathsf{Tr}(\mathbf{A}\mathbf{X}\mathbf{B})] = \mathbf{A}^\mathsf{T}\mathbf{B}^\mathsf{T}$

$\frac{\partial}{\partial \mathbf{X}}\left[\mathsf{Tr}\left(\mathbf{A}\mathbf{X}^\mathsf{T}\mathbf{B}\right)\right] = \mathbf{B}\mathbf{A}$

4.4 Vector-by-Matrix (Generalized Gradient)

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

5 General Machine Learning

$$\underbrace{P(\text{model } \theta\,|\,\text{data } D)}_{\text{Posterior}} = \frac{\overbrace{P(\text{data}\,|\,\text{model})}^{\text{Likelihood}} \times \overbrace{P(\text{model})}^{\text{Prior}}}{\underbrace{P(\text{data})}_{\text{Evidence}}}$$

6 Information Theory

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

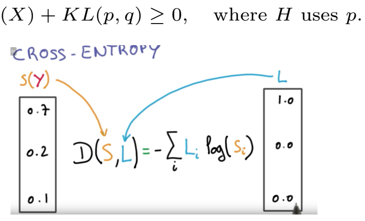
$H(\boldsymbol{X}) = -\sum_{\mathbf{x}\in\mathcal{X}} p(\mathbf{x})\log(p(\mathbf{x})) = \mathbb{E}[I(X)] = \mathbb{E}[-\log(P(X))] \geq 0$.

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

D. (Cross-Entropy) for the 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 second formulation clearly shows why $q := p$ is the minimizer of the cross-entropy (or hence: the maximizer of the likelihood).

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

Relation to Log-Likelihood

In classification problems we want to estimate the probability of different outcomes. If we have the following quantities:

- estimated probability of outcome i is q_i. Now we want to tune q in a way that the data gets the most likely. First, let's just see how good q is doing.
- the frequency (empirical probability) of outcome i in the data is p_i
- n data points

Then the likelihood of the data under p_i is

$\prod_{i=1}^n q_i^{n\cdot p_i}$

since the model estimates event i with probability q_i exactly $n \cot p_i$ times. Now the log-likelihood, divided by n is

$\frac{1}{n}\sum_{i=1}^n np_i\log(q_i) = \sum_{i=1}^n p_i\log(q_i) = -H(p,q)$

Hence, maximizing the log-likelihood corresponds to minimizing the cross-entropy (which is why it's used so often as a loss).

D. (Kullback-Leibler Divergence)

For discrete 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 $\forall \mathbf{x}$: $q(\mathbf{x}) = 0 \implies p(\mathbf{x}) = 0$ (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.

7 NN Functions and their Derivatives

D. (Hard Tan)

HardTanh: $\mathbb{R}^n \to \mathbb{R}^n$

$\mathbf{z} = \mathrm{HardTanh}(\mathbf{x}) = \mathbf{x} \odot \mathbb{1}_{\{\mathbf{x}\in[-1,1]\}} + \mathbb{1}_{\{\mathbf{x}>1\}} - \mathbb{1}_{\{\mathbf{x}<-1\}}$

$\mathbf{z}' = \mathrm{HardTanh}'(\mathbf{x}) = \mathrm{diag}(\mathbb{1}_{\{\mathbf{x}\in[-1,1]\}})$

D. (Max Layer)

max: $\mathbb{R}^n \to \mathbb{R}$

$z = \max(\mathbf{x})$

$z' = \max'(\mathbf{x}) = \mathrm{diag}(\mathbf{e}_i), \quad$ where $i = \arg\max_i(\mathbf{x}_i)$

D. (Softmax)

Now here the output of each activation ? depends on every input, thus the jacobian is not just a diagonal matrix.

$\mathrm{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{i=1}^c e^{x_c}}$

$\frac{\partial\,\mathrm{softmax}(\mathbf{x})_i}{\partial x_j} = \begin{cases} -\mathrm{softmax}(x)_i\,\mathrm{softmax}(x)_j, & i \neq j \\ \mathrm{softmax}(x)_i - \mathrm{softmax}(x)_i\,\mathrm{softmax}(x)_j & i = j \end{cases}$

$\nabla_\mathbf{x}\mathrm{softmax}(\mathbf{x}) = \mathbf{J}_{\mathrm{softmax}}(\mathbf{x}) = \mathrm{diag}(\mathrm{softmax}(\mathbf{x})) - \mathrm{softmax}(\mathbf{x})\,\mathrm{softmax}(\mathbf{x})^\mathsf{T}$

8 Taylor Approximations

T. (Taylor-Lagrange Formula)

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

D. (m-th Taylor Polynomial for f at a)

$$P_m^a(x) = \sum_{k=0}^m \frac{1}{k!}f^{(k)}(a)(x-a)^k$$

D. (Error of m-th Taylor Polynomial for f at a)

$R_m^a(x) := f(x) - P_m^a(x) \quad \iff \quad f(x) = \underbrace{P_m^a(x)}_{\text{approx.}} + \underbrace{R_m^a(x)}_{\text{error}}$

T. (Approximation Quality of Taylor Polynomials) Let $f \in C^m([a,b])$ and let f be $(m+1)$-times differentiable. Then

$\exists \xi \in [a,b]$: $\quad f(x) = P_m^a(x) + \underbrace{\frac{1}{(m+1)!}f^{(m+1)}(\xi)(x-a)^{m+1}}_{R_m^a(x)=}$

Hence, $R_m^a(x) \in \mathcal{O}(\epsilon^{m+1})$ where $\epsilon := x - a$.

$\epsilon := x_{\text{new}} - x \quad$ approximation at x, interpolation to x_{new}

Finite difference method to approximate gradient

$f(x+\epsilon) \approx f(x) + \epsilon\nabla f(x) + \mathcal{O}(\epsilon^2) \iff \nabla f(x) = \frac{f(x+\epsilon)-f(x)}{\epsilon} + \mathcal{O}(\epsilon^2)$

Symmetrical central differences reduces error

$f(x+\epsilon) \approx f(x) + \epsilon^\mathsf{T}\nabla f(x) + \frac{1}{2}\epsilon^T\mathrm{Hess}(f)(x)\epsilon + \mathcal{O}(\epsilon^3)$

$f(x-\epsilon) \approx f(x) - \epsilon^\mathsf{T}\nabla f(x) + \frac{1}{2}\epsilon^T\mathrm{Hess}(f)(x)\epsilon + \mathcal{O}(\epsilon^3)$

$\nabla f(x) \approx \frac{f(x+\epsilon)-f(x-\epsilon)}{2\epsilon} + \mathcal{O}(\epsilon^3)$

2nd-Order Taylor expanion at $\mathbf{x}_0$ (function for $\mathbf{x}$, we want to extrapolate to $f(\mathbf{x})$)

$f(\mathbf{x}) \approx f(\mathbf{x}_0) + (\mathbf{x}-\mathbf{x}_0)\nabla_\mathbf{x} f(\mathbf{x}_0) + (\mathbf{x}-\mathbf{x}_0)^\mathsf{T}\mathrm{Hess}(f)(\mathbf{x}_0)(\mathbf{x}-\mathbf{x}_0)$

9 Newton's Method

$\mathbf{x}^{t+1} = \mathbf{x}^t - Hess(f)(\mathbf{x}^t)^{-1}\nabla_\mathbf{x} f(\mathbf{x}^t)$.

10 Approximation Theory

10.1 Compositional Models

Want to learn: $F^*: \mathbb{R}^n \to \mathbb{R}^m$, **Learning:** Now we reduce this task to learning a function F in some parameter space $\mathbb{R}^d$ that approximates F^* well.

$F: \mathbb{R}^n \times \mathbb{R}^d \to \mathbb{R}^m, \qquad \mathcal{F} := \{F(\cdot,\boldsymbol{\theta})\} \quad \boldsymbol{\theta} \in \mathbb{R}^d$

DL: the composition of simple functions can give rise to very complex functions.

$F: \mathbb{R}^n \xrightarrow{G_1} \mathbb{R}^* \xrightarrow{G_2} \mathbb{R}^* \xrightarrow{G_3} \cdots \xrightarrow{G_L} \mathbb{R}^m$

$F = G_L \circ \cdots \circ G_2 \circ G_1$

$F(\mathbf{x},\boldsymbol{\theta}) = G_L(\cdots G_2(G_1(\mathbf{x};\boldsymbol{\theta}_1);\boldsymbol{\theta}_2);\cdots;\boldsymbol{\theta}_L)$.

10.2 Compositions of Maps

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

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

T. $f: \mathbb{R}^n \to \mathbb{R}$ is linear $\iff f(\mathbf{x}) = \mathbf{w}^\mathsf{T}\mathbf{x}$ 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.

D. (Level Sets) of a function $f: \mathbb{R}^n \to \mathbb{R}$ is a one-parametric family of sets defined as

$$L_f(c) := \{\mathbf{x} \mid f(\mathbf{x}) = c\} = f^{-1}(c) \subseteq \mathbb{R}^n.$$

T. (Comp. of Lin. Maps/- is a Lin. Map/Unit)

Let $F_1,\dots,F_L$ be linear maps, then $F = F_L \circ \cdots \circ F_2 \circ F_1$ is also a linear map.

C. Every L-layer NN of linear layer collapses to a 1-layer NN. Further note that hereby

$\mathrm{rank}(F) \equiv \dim(\mathrm{im}(F)) \leq \min_{i\in\{1,\dots,L\}}\mathrm{rank}(F_i)$.

So this strongly suggests, that we need to move beyond linearity and use generalizations of linear maps (e.g., p.w. linear functions, or ridge functions).

10.3 Universal Approximation with Ridge Functions

D. (Ridge Function) $f: \mathbb{R}^n \to \mathbb{R}$ is a *ridge function*, if it can be written as $f(\mathbf{x}) = \sigma(\mathbf{w}^\mathsf{T}\mathbf{x}+b)$ for some $\sigma: \mathbb{R}\to\mathbb{R}$, $\mathbf{w} \in \mathbb{R}^n$, $b \in \mathbb{R}$.

- $\overline{f}(\mathbf{x}) = \mathbf{w}^\mathsf{T}\mathbf{x} + b$ (linear part)
- $L_f(c) = \bigcup_{d\in\sigma^{-1}(c)} L_{\overline{f}}(d) = \bigcup_{d\in L_\sigma(c)} L_{\overline{f}}(d)$
- if σ is differentiable at $z = \mathbf{w}^\mathsf{T}\mathbf{x} + b$ then
$\nabla_\mathbf{x} f(\mathbf{x}) = \sigma'(z)\nabla_\mathbf{x}\overline{f}(\mathbf{x}) = \sigma'(z)\mathbf{w} = \sigma'(\mathbf{w}^\mathsf{T}\mathbf{x}+b)\mathbf{w}$
- a ridge function picks out one direction of change (linear part), and then models the rate of change in that chosen direction via σ

T. Let $f: \mathbb{R}^n \to \mathbb{R}$ be differentiable at $\mathbf{x}$. Then either $\nabla_\mathbf{x} f(\mathbf{x}) = 0$, or $\nabla_\mathbf{x} f(\mathbf{x}) \perp L_f(f(\mathbf{x}))$

D. (Universe of Ridge Functions for some $\sigma: \mathbb{R}\to\mathbb{R}$)

$\mathcal{G}_\sigma^n := \left\{g \,\middle|\, g(\mathbf{x}) = \sigma(\mathbf{w}^\mathsf{T}\mathbf{x}+b),\ \mathbf{w}\in\mathbb{R}^n,\ b\in\mathbb{R},\ \sigma: \mathbb{R}\to\mathbb{R}\right\}$

D. (Universe of Continuous Ridge Functions)

$\mathcal{G}^n := \bigcup_{\sigma\in C(\mathbb{R})}\mathcal{G}_\sigma^n$.

T. Composition of continuous functions is continuous.

T. (Hence) $\mathcal{G}^n \subseteq C(\mathbb{R}^n)$.

D. (Span of Universe of Continuous Ridge Functions)

$\mathcal{H}^n := \mathrm{span}(\mathcal{G}^n) = \left\{h \,\middle|\, h = \sum_{j=1}^r g_j,\ g_j \in \mathcal{G}^n\right\}$.

D. (Dense Function Class $\mathcal{H}$ in $C(\mathbb{R}^d)$)

A *function class* $\mathcal{H} \subseteq C(\mathbb{R}^d)$ is *dense* in $C(\mathbb{R}^d)$, iff

$$\forall f \in C(\mathbb{R}^d) \quad \forall\epsilon > 0 \quad \forall K \subset \mathbb{R}^d,\ K \text{ compact}$$

$$\exists h \in \mathcal{H}: \max_{x\in K}|f(\mathbf{x}) - h(\mathbf{x})| = \|f - h\|_{\infty,K} < \epsilon.$$

T. (Density of Span of Continuous Ridge Functions)

$\mathcal{H}^n$ is *dense* in $C(\mathbb{R}^n)$. Note: we can absorb the linear combination weights within the functions g_j.

So, we can approximate any $f \in C(\mathbb{R}^n)$ through linear combinations of members in $\mathcal{G}^n$. This gives rise to the idea of building a 1-layer network with adaptive ridge functions (rather impractical). In the following we'll see how we can limit ourselves to one specific σ s.t. $\mathrm{span}(\mathcal{G}_\sigma^n)$ is still dence in $C(\mathbb{R}^n)$. And we'll see how we can simplify the discussion through dimension lifting.

D. (Smooth Function $f \in C^\infty(\mathbb{R})$) A function is "smooth" if it has infinitely many continuous derivatives.

Com. The function may even be just a constant function.

T. (Approximation Theorem, 1993)

$\sigma \in C^\infty(\mathbb{R})$, σ not polynomial $\implies \mathcal{H}_\sigma^1$ is dense in $C(\mathbb{R})$.

C. MLPs with one hidden layer, and any non-polynomial, *smooth* activation function are a universal function approximators.

Com. we don't know how many hidden units are needed; the requirement that f is smooth can be substantially weakened (see results with ReLUs).

L. MLPs with one hidden layer and a polynomial activation function are *not* universal function approximators.

L. (Dimension Lifting)

$\mathcal{H}_\sigma^1$ *dense* in $C(\mathbb{R}) \implies \forall n \geq 1$: $\mathcal{H}_\sigma^n$ *dense* in $C(\mathbb{R}^n)$

Com. So we can lift the *density property of ridge functions* from $C(\mathbb{R})$ to $C(\mathbb{R}^n)$.

Summary

$\underset{\sigma \text{ not polynomial}}{\sigma \in C^\infty(\mathbb{R}),} \overset{\text{Approx. Thm.}}{\Longrightarrow} \mathcal{H}_\sigma^1$ is dense in $C(\mathbb{R}) \overset{\text{Dim. Lifting}}{\Longrightarrow} \forall n$: $\mathcal{H}_\sigma^n$ is dense in $C(\mathbb{R}^n)$

10.4 Sigmoid Networks

D. (Sigmoid Activation Function)

$\sigma(x) := \frac{1}{1+e^{-x}} \in (0;1)$

$\sigma^{-1}(y) = \ln\left(\frac{y}{1-y}\right)$

$\sigma'(x) := \sigma(x)\cdot(1-\sigma(x))$

$\nabla_\mathbf{x}\sigma(\mathbf{x}) = \mathbf{J}_\sigma(\mathbf{x}) = \mathrm{diag}(\sigma(\mathbf{x}) \odot (1-\sigma(\mathbf{x})))$

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

T. $\sigma(-x) = 1 - \sigma(x)$

D. (Tanh Activation Function)

$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \in (-1;1)$

$\tanh'(x) = 1 - \tanh^2(x)$

$\nabla_\mathbf{x}\tanh(\mathbf{x}) = \mathbf{J}_{\tanh}(\mathbf{x}) = \mathbf{I} - \mathrm{diag}(\tanh^2(\mathbf{x}))$

$\tanh'(x) = 4\sigma'(2x) = 4\sigma(2x)(1-\sigma(2x))$

Connection between Sigmoid and Tanh (Equal Representation Strength)

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

$$\begin{aligned}\tanh(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^x}{e^x + e^{-x}}\cdot\frac{e^{-x}}{e^{-x}} - \frac{e^{-x}}{e^x + e^{-x}}\cdot\frac{e^x}{e^x} \\ &= \frac{1}{1+e^{-2x}} - \frac{1}{1+e^{2x}} = \sigma(2x) - \frac{1+e^{2x}-e^{2x}}{1+e^{2x}} \\ &= \sigma(2x) - \frac{1+e^{2x}}{1+e^{2x}} + \frac{e^{2x}}{1+e^{2x}}\cdot\frac{e^{-2x}}{e^{-2x}} = \sigma(2x) - 1 + \frac{1}{1+e^{-2x}} \\ &= 2\sigma(2x) - 1.\end{aligned}$$

10.5 Rectification Networks

D. (Rectified Linear Unit (ReLU))

$(x)_+ := \max(0,x) = \mathrm{ReLU}(x) \in [0,\infty]$

$(x)'_+ := \mathbb{1}_{\{x>0\}}$

$\nabla_\mathbf{x}(\mathbf{x})_+ = \mathbf{J}_{(\cdot)_+}(\mathbf{x}) := \mathrm{diag}(\mathbb{1}_{\{\mathbf{x}>0\}})$

$\underbrace{\partial(z)_+}_{\text{subdiff.}} = \begin{cases}\{1\}, & x > 0, \\ \{0\}, & x < 0, \\ [0;1], & x = 0.\end{cases}$

- a linear function over a half-space $\mathcal{H}$,
- and zero on the complement $\mathcal{H}^c = \mathbb{R}^n - \mathcal{H}$.
- non-smooth

D. (Absolute Value (Rectification) Unit (AVU))

$|z| := \begin{cases} z, & z \geq 0 \\ -z, & z < 0.\end{cases} \qquad \partial|z| = \begin{cases}1, & x > 0, \\ [-1;1], & x = 0, \\ -1, & x < 0.\end{cases}$

Relationship between ReLU and AVU

$(x)_+ = \frac{x + |x|}{2}, \qquad |x| = (x)_+ + (-x)_+ = 2(x)_+ - x$

T. Any $f \in C([0;1])$ can be uniformly approximated to arbitrary precision by a polygonal line (c.f. Shektman, 1982). Or in other words:
$\mathcal{H} = \{$p.w. linear functions$\}$ is *dense* in $C([0,1])$.

T. Lebesgue showed how a polygonal line with m pieces can be written

$g(x) = ax + b + \sum_{i=1}^{m-1} c_k(x - x_i)_+$ (ReLU function approx.)

- Knots: $0 = x_0 < x_1 < \cdots < x_{m-1} < x_m = 1$
- $m+1$ parameters: $a, b, c_1, \dots, c_{m-1}$

With the dimension lifting theorem we can lift this property from 1D to nD.

Note that there's an alternative representation of the above through AVUs

$g(x) = a'x + b' + \sum_{i=1}^m -1c'_i\,|x - x_i|$

T. Networks with one hidden layer of ReLUs or AVUs are universal function approximators.

Com. We can thus use a restricted set of activation functions (ReLUs or AVUs). But still we don't know how many hidden units we need.

Proof. (Sketch)

1. Universally approximate $C(K)$ functions (K, compact) by polygonal lines
2. Represent polygonal lines by (linear function +) linear combinations of $(\cdot)_+$ or $|\cdot|$-functions
3. Apply dimension lifting lemma to show density of the linear span of resulting ridge function families $\mathcal{G}_{(\cdot)_+}^n$ and $\mathcal{G}_{|\cdot|}^n$.

10.5.1 Piecewise Linear Functions and Half Spaces

So the ReLU and the AVU define a *piecewise linear function* with 2 pieces. Hereby, $\mathbb{R}^n$ is partitioned into two open half spaces (and a border face):

- $H^+ := \left\{\mathbf{x} \,\middle|\, \mathbf{w}^\mathsf{T}\mathbf{x} + b > 0\right\} \subset \mathbb{R}^n$
- $H^- := \left\{\mathbf{x} \,\middle|\, \mathbf{w}^\mathsf{T}\mathbf{x} + b < 0\right\} \subset \mathbb{R}^n$
- $H^0 := \left\{\mathbf{x} \,\middle|\, \mathbf{w}^\mathsf{T}\mathbf{x} + b = 0\right\} = \mathbb{R}^n - H^+ - H^- \subset \mathbb{R}^n$

Further note that

- $g_{(\cdot)_+}(H^0) = g_{|\cdot|}(H^0) = 0$
- $g_{(\cdot)_+}(H^-) = 0$
- $g_{|\cdot|}(\mathbf{x}) = g_{|\cdot|}(\mathbf{v} - \mathbf{x})$ with $\mathbf{v} = -2b\frac{\mathbf{w}}{\|\mathbf{w}\|_2^2}$ (mirroring at $\mathbf{w}$: equivalent to subtracting the projection of $\mathbf{x}$ onto $\mathbf{w}$ twice from $\mathbf{x}$)

Partitions of ReLUs go to infinity, even if we have no examples there → weak to extrapolation errors (adversarial examples). However, if you have enough data, then you can overcome this, because there won't be any new examples that lie outside of the training data regions.

10.5.2 Linear Combinations of ReLUs

T. (Zaslavsky, 1975) By linearly combining m rectified units $\mathbb{R}^n$ can be partitioned at most into $R(m)$ cells: $R(m) \leq \sum_{i=0}^{\min\{m,n\}}\binom{m}{i}$

C. If classes $m \leq n$ we have $R(m) = 2^m$ (exponential growth).

C. For any input size n we have $R(m) \in \mathcal{O}(m^n)$ (polynomial slowdown in number of cells, limited by the input space dimension).

10.5.3 Deep Combination of ReLUs

Question: Process n inputs through L ReLU layers with widths $m_1,\dots,m_L \in \mathcal{O}(m)$. Into how many ($= R(m,L)$) cells can $\mathcal{R}^n$ be maximally partitioned?

T. (Montufar et al, 2014) If we process n-dim. inputs through L ReLU layers with widths $m_1,\dots,m_L \in \mathcal{O}(m)$.Then $\mathbb{R}^n$ can be partitioned into at most $R(m,L)$ layers:

$R(m,L) \in \mathcal{O}\left(\left(\frac{m}{n}\right)^{n(L-1)}m^n\right)$

Com. So for a fixed n the exponential growth (that may be lost if classes $m > n$ input dim, and we use one hidden layer) of the number of partitions can be recuperated by increasing the number of layers. Further, by adding layers, one may reduce the total number of hidden units in total (→ less params)

10.5.4 Hinging Hyperplanes

D. (Hinge Function (extension of ReLU))

If $g: \mathbb{R}^n \to \mathbb{R}$ can be written with parameters $\mathbf{w}_1, \mathbf{w}_2 \in \mathbb{R}^n$ and $b_1, b_2 \in \mathbb{R}$ as below it is called a hinge function

$$g(\mathbf{x}) = \max\left(\mathbf{w}_1^\mathsf{T}\mathbf{x} + b_1, \mathbf{w}_2^\mathsf{T}\mathbf{x} + b_2\right)$$

- two hyperplanes, "glued" together at their intersection. So for the intersection it holds that: $\mathbf{w}_1^\mathsf{T}\mathbf{x} + b_1 = \mathbf{w}_2^\mathsf{T}\mathbf{x} + b_2$.
- Representational power: $2\max(f,g) = f + g + |f - g|$.

The good thing is that these hyperplanes (as opposed to the ReLU) don't interact only in one dimension ($\mathbf{w}$), but they interact in two dimensions $\mathbf{w}_1, \mathbf{w}_2$.

T. Given a continuous p.w. linear function in $\mathbb{R}^n$, we can represent the function as

$\mathbf{w}_1^\mathsf{T}\mathbf{x} + b_1 \pm \left|\mathbf{w}_2^\mathsf{T}\mathbf{x} + b_2\right| \pm \left|\mathbf{w}_3^\mathsf{T}\mathbf{x} + b_3 + \left|\mathbf{w}_4^\mathsf{T}\mathbf{x} + b_4\right|\right| \pm \left|\mathbf{w}_5^\mathsf{T}\mathbf{x} + b_5 + \left|\mathbf{w}_6^\mathsf{T}\mathbf{x} + b_6 + \left|\mathbf{w}_7^\mathsf{T}\mathbf{x} + b_7\right|\right|\right| \pm \cdots$

So for a continuous p.w. linear function in $\mathbb{R}^n$ we need n nested (as above) absolute value functions (→ n layers with AVUs needed (same as data dimensionality!)

Com. So, every $\pm$-term has one more nesting.

Luckily, these smart guys managed to prove that if we use a hinge function with k inputs, then the number of AVU nestings can be reduced to logarithmic growth.

D. (k-Hinge Function) $g(\mathbf{x}) = \max(\mathbf{w}_1^\mathsf{T}\mathbf{x} + b_1, \dots, \mathbf{w}_k^\mathsf{T}\mathbf{x} + b_k)$.

T. (Wang and Sun, 2005) Every continuous p.w. linear function from $\mathbb{R}^n \to \mathbb{R}$ can be written as a signed sum of k-Hinges with $k_i \leq \lceil\log_2(n+1)\rceil$. So $f(\mathbf{x}) = \sum_i \theta_i g_i(\mathbf{x})$ where $\theta_i \in \{\pm 1\}$.

Com. This reduces the growth of absolute value nesting to logarithmic growth, instead of linear growth.

C. P.w. linear functions are dense in $C(\mathbb{R}^n)$.

10.5.5 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 $A_1,\dots,A_k$, and then we define the activations $G_j(\mathbf{x})$ for $j \in \{1,\dots,k\}$ as

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

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.

Proof. (Sketch)

1. Wang's theorem: Linear network with two maxout units and a linear output unit (subtraction) can represent any continous p.w. linear function (exactly!)
2. continous p.w. linear function are dense in $C(\mathbb{R}^n)$

11 Feedforward Networks

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

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

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

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

where $\overline{F}^l: \mathbb{R}^{m_{l-1}} \to \mathbb{R}^{m_l}$ 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 $\sigma^l: \mathbb{R}^{m_l} \to \mathbb{R}^{m_l}$ element-wise non-linearity at layer l.

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

D. (Hidden Layer) A layer that is neither the input, nor the output layer is called a *hidden layer*.

So a feedforward neural network represents a *family of functions* $\mathcal{F}$.

The functions $F_\theta \in \mathcal{F}$ are *parametrized* by parameters θ, $\theta = \bigcup_{l=1}^L\left\{\mathbf{W}^l, \mathbf{b}^l\right\}$.

Probability Distribution Perspective

It is often useful to view the output of a FF network as the parameters $\boldsymbol{\mu}$ of some distribution over $\mathcal{Y}$.

$F_\theta: \mathbb{R}^n \to \mathbb{R}^m \to \mathcal{P}(\mathcal{Y})$

$\mathbf{x} \overset{\text{learned}}{\mapsto} \boldsymbol{\mu} \overset{\text{fixed}}{\mapsto} P(\mathbf{y}\,|\,\mathbf{x};F_\theta) = P(\mathbf{y}\,|\,\boldsymbol{\mu} = F_\theta(\mathbf{x})) \quad \mathbf{y} \sim P(\mathbf{y};\boldsymbol{\mu})$

11.1 Output Units and Objectives

Now, how can we find the *most appropriate function* in $\mathcal{F}$ based on training data $\{(\mathbf{x}_1,\mathbf{y}_1),\dots,(\mathbf{x}_N,\mathbf{y}_N)\}$? There are basically two options (both leading to similar loss functions):

- Decision theory (min. some risk, max. some payoff, ...)
- Maximum Likelihood (max. likelihood, min. prob. dens. dist., ...)

11.1.1 Decision Theory

In decision theory, we strive to minimize the expected risk (defined through a loss function ℓ) of a function F.

D. (Expected Risk of a Function F)

$F^* = \arg\min_F \sum_{y\in\mathcal{Y}}\int_{\mathcal{X}} \ell(y, F(\mathbf{x}))p(\mathbf{x},y)\,d\mathbf{x} = \arg\min_F \underbrace{\mathbb{E}_{\boldsymbol{X},Y}[\ell(Y, F(\boldsymbol{X}))]}_{\mathcal{R}^*(F) \text{ expected risk of } F}$

However, we do not know ℓ, we aren't given $p(\mathbf{x},y)$ (only samples).

D. (Loss Function)

$\ell: \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_{\geq 0} \qquad (\underbrace{\mathbf{y}^*}_{\text{true}}, \underbrace{\mathbf{y}}_{\text{pred.}}) \mapsto \ell(\mathbf{y}^*, \mathbf{y})$

s.t. $\forall \mathbf{y} \in \mathcal{Y}$: $\ell(\mathbf{y},\mathbf{y}) = 0 \quad$ and $\quad \forall \mathbf{y}^*, \mathbf{y} \in \mathcal{Y}, \mathbf{y} \neq \mathbf{y}^*$: $\ell(\mathbf{y}^*,\mathbf{y}) > 0..$

D. (Training Risk / Empirical Risk)

$\mathcal{S}_N := \left\{(\mathbf{x}_i, y_i) \overset{\text{i.i.d.}}{\sim} p \,\middle|\, i = 1,\dots,N\right\}$.

$\mathcal{R}(F;\mathcal{S}_N) = \frac{1}{N}\sum_{i=1}^N \ell(y_i, F(\mathbf{x}_i))$

So the *training risk* is the *expected risk* under the empirical distribution induced by the sample $\mathcal{S}_N$.

D. (Empirical Risk Minimization)

$\mathcal{F} = \{F_\theta \mid \theta \in \Theta\} \quad$ (e.g. neural network)

$\hat{F}(\mathcal{S}_N) = \arg\min_{F\in\mathcal{F}}\mathcal{R}(F;S_N)$

11.1.2 Why Regularization is Needed

Wish: $\mathcal{R}(\hat{F};\mathcal{S}_n) \overset{n\to\infty}{\to} \mathcal{R}^*(F^*)$ (no overfitting)

So as n grows we'd want to do as good as if we knew $p(\mathbf{x},\mathbf{y})$.

Law of large numbers guarantees: $\mathcal{R}(F;\mathcal{S}_n) \overset{n\to\infty}{\to} \mathcal{R}^*(F)$ but this doesn't help us to move from some $F \to F^*$.

In order to ultimately get from $\hat{F}$ to F^*, which minimizes $\mathcal{R}^*(F^*)$, we need to *restrict* $\mathcal{F}$ such that $\mathcal{R}(\hat{F};\mathcal{S}_n) \overset{n\to\infty}{\to} \mathcal{R}^*(F^*)$. This will prevent overfitting. However it's not so easy to determine how to constrain $\mathcal{F}$ such that $\hat{F} \to F^*$.

D. (Regularized Empirical Risk)

$\mathcal{R}_r(F;\mathcal{S}_n) := \mathcal{R}(F;\mathcal{S}_n) + \lambda\Omega(\|F\|)$

11.1.3 Regularization Approaches

Weight decay, data augmentation, noise robustness (to weights, inputs, labels (compensates for errors in labeling of data)), semi-supervised learning, dropout, early stopping (prevent overfitting through long training), multi-task learning (to learn general representation or use more data), parameter sharing (CNNs), ensembles (reduces variance, same bias, costly)

Some regularization methods can be proven to be equivalent. However: only in the limit. Thus, it's good to use combinations of them in finite horizons.

12 Backpropagation

Learning in neural networks is about *gradient-based optimization* (with very few exceptions). So what we'll do is compute the gradient of the objective (empirical risk) with regards to the parameters θ: $\nabla_\theta\mathcal{R} = \left(\frac{\partial\mathcal{R}}{\partial\theta_1} \quad \cdots \quad \frac{\partial\mathcal{R}}{\partial\theta_d}\right)^\mathsf{T}$.

12.1 Comp. of Gradient via Backpropagation

A good way to compute $\nabla_\theta\mathcal{R}$ is by exploiting the computational structure of the network through the so-called *backpropagation*.

The basic steps of *backpropagation* are as follows:

1. Forward-Pass: Perform a forward pass (for a given training input $\mathbf{x}$), compute all the activations for all units
2. Compute the gradient of $\mathcal{R}$ w.r.t. the ouput layer activations (for a given target $\mathbf{y}$) (even though we're not interested directly in these gradients, they'll simplify the computations of the gradients in step 4.)
3. Iteratively propagate the activation gradient information from the outputs of the previous layer to the inputs of the current layer.
4. Compute the local gradients of the activations w.r.t. the weights and biases

Since NNs are based on the composition of functions we'll inevitably need the *chain rule*.

T. (1-D Chain Rule) $y = f(\underbrace{g(x)}_{z})$

$(f \circ g)' = (f' \circ g)\cdot g' \qquad \left.\frac{d(f\circ g)}{dx}\right|_{x=x_0} = \left.\frac{df}{dz}\right|_{z=g(x_0)} \cdot \left.\frac{dg}{dx}\right|_{x=x_0}$

D. (Jacobi Matrix of a Map)

The Jacobian $\mathbf{J}_F$ of a map $F: \mathbb{R}^n \to \mathbb{R}^m$ is defined as

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

So each component function $F_i: \mathbb{R}^n \to \mathbb{R}$ of F, for $i \in \{1,\dots,m\}$ has a respective gradient. Putting these gradients together as rows of a matrix gives us the *Jacobian* of F.

So we have that

$$(\mathbf{J}_F)_{ij} = \frac{\partial F_i}{\partial x_j}.$$

T. (Jacobi Matrix Chain Rule)

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

$F := H \circ G$

$$F: \mathbb{R}^n \overset{G}{\to} \mathbb{R}^q \overset{H}{\to} \mathbb{R}^m$$

$$\mathbf{x} \overset{G}{\mapsto} \mathbf{z} := G(\mathbf{x}) \overset{H}{\mapsto} \mathbf{y} := H(\mathbf{z}) = H(G(\mathbf{x}))$$

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

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

This gives us the following lemma for Jacobi matrices of compositions of functions

L. (Jacobi Matrix Chain Rule)

$$\mathbf{J}_F|_{\mathbf{x}=\mathbf{x}_0} = \mathbf{J}_{H\circ G}|_{\mathbf{x}=\mathbf{x}_0} = \mathbf{J}_H|_{\mathbf{z}=G(\mathbf{x}_0)} \cdot \mathbf{J}_G|_{\mathbf{x}=\mathbf{x}_0}$$

Proof. This just follows from the upper relationship

$$\begin{aligned}(\mathbf{J}_{H\circ G})_{ij}|_{\mathbf{x}=\mathbf{x}_0} &= \left.\frac{\partial(H\circ G)_i}{\partial x_j}\right|_{\mathbf{x}=\mathbf{x}_0} \\ &= \sum_{k=1}^q \left.\frac{\partial H_i}{\partial z_k}\right|_{\mathbf{z}=G(\mathbf{x}_0)} \cdot \left.\frac{\partial G_k}{\partial x_j}\right|_{\mathbf{x}=\mathbf{x}_0} \\ &= (\mathbf{J}_H)_{i,:}|_{\mathbf{z}=G(\mathbf{x}_0)}\,(\mathbf{J}_G)_{:,j}|_{\mathbf{x}=\mathbf{x}_0} \\ &= (\nabla_\mathbf{z} H_i)^\mathsf{T}\Big|_{\mathbf{z}=G(\mathbf{x}_0)}\,(\mathbf{J}_G)_{:,j}|_{\mathbf{x}=\mathbf{x}_0}\end{aligned}$$

– Special Case: Function Composition

Let's have a look at the special case where we compose a map with a function

$$G: \mathbb{R}^n \to \mathbb{R}^m, \quad h: \mathbb{R}^m \to \mathbb{R}, \quad h \circ G: \mathbb{R}^n \to \mathbb{R}$$

And let's use the more intuitive variable notation

$$\mathbf{x} \overset{G}{\mapsto} \mathbf{y} \overset{h}{\mapsto} z$$

Then we have that

$$\nabla_\mathbf{x} z = \sum_{k=1}^m \frac{\partial z}{\partial y_k}\cdot\frac{\partial y_k}{\partial \mathbf{x}} = (\mathbf{J}_G)^\mathsf{T}\nabla_\mathbf{y} z$$

– Important Notation

Note that we have a lot of indicies. Always use the following convention:

- index of a layer: put as a *superscript*
- index of a dimension of a vector: put as a *subscript*
- futher we use the following shorthand for layer activations

$$\mathbf{x}^l := (F^l \circ \cdots \circ F^1)(\mathbf{x}) \in \mathbb{R}^{m_l}$$

$$x_i^l \in \mathbb{R}\text{: activation of } i\text{-th unit in layer } l$$

- the index of a data point (i.e., the index of one of the N samples), is omitted where possible, but if we really need to we'll use rectangular brackets (to not confuse it with the vector indices). So,

$$(\mathbf{x}[i], \mathbf{y}[i]) \text{ is the } i\text{-th sample point.}$$

Note that in the following we'll be taking gradients w.r.t. one datapoint (just in order to not get another index in this mess). So, one data sample $(\mathbf{x},\mathbf{y})$. If we wanted to take the gradient w.r.t. some larger batch, then the resulting gradient would just be the average of the gradients for each sample datapoint in the batch.

– Deep Function Compositions

$$F_\theta: \mathcal{X} = \mathbb{R}^n \to \mathbb{R}^m = \mathcal{Y}$$
$$F_\theta = F_\theta^L \circ \cdots \circ F_\theta^1$$

$$\mathcal{R}: \underbrace{(\mathcal{X} \overset{F_\theta}{\to} \mathcal{Y})}_{\substack{\text{we're interested in} \\ \text{the predictions that} \\ \text{result from choosing } \theta}} \times \underbrace{(\mathcal{X}\times\mathcal{Y})}_{\substack{\text{evaluation set } \mathcal{S} \\ \text{(induces emp. dist.)}}} \times \underbrace{(\mathcal{Y}\times\mathcal{Y}\to\mathbb{R}_{\geq 0})}_{\text{loss } \ell} \to \mathbb{R}_{\geq 0}$$

$$\mathbf{x} =: \mathbf{x}^0 \overset{F_\theta^1}{\mapsto} \mathbf{x}^1 \overset{F_\theta^2}{\mapsto} \mathbf{x}^2 \overset{F_\theta^3}{\mapsto} \cdots \overset{F_\theta^L}{\mapsto} \mathbf{x}^L =: \mathbf{y} \overset{\mathcal{R}}{\mapsto} \mathcal{R}(F_\theta;\{\mathbf{x},\mathbf{y}^*\}) \in \mathbb{R}_{\geq 0}$$

$\mathbf{x}^l = \sigma^l\left(\mathbf{W}^l\mathbf{x}^{l-1} + \mathbf{b}^l\right)$

$\mathbf{e}^l = \nabla_{\mathbf{x}^l}\mathcal{R}$

$$\mathbf{e}^l = \nabla_{\mathbf{x}^l}\mathcal{R} = \frac{\partial\mathcal{R}}{\partial\mathbf{x}^l} = \sum_{k=1}^{m_{l+1}} \underbrace{\frac{\partial\mathcal{R}}{\partial x_k^{l+1}}}_{e_k^{l+1}} \cdot \underbrace{\frac{\partial x_k^{l+1}}{\partial\mathbf{x}^l}}_{(\mathbf{J}_{F^{l+1}}^\mathsf{T})_{:,k}} = \mathbf{J}_{F^{l+1}}^\mathsf{T}\mathbf{e}_k^{l+1}.$$

$\mathbf{e}^0 = \mathbf{J}_{F^1}^T \cdot \mathbf{J}_{F^2}^T \cdots \mathbf{J}_{F^{L-1}}^T \cdots \mathbf{J}_{F^L}^T\mathbf{e}^L$

Backpropagation just gives us another network in the reversed direction. $\mathbf{e}^L \overset{\mathbf{J}_{F^L}^\mathsf{T}}{\mapsto} \mathbf{e}^{L-1} \overset{\mathbf{J}_{F^{L-1}}^\mathsf{T}}{\mapsto} \cdots \mathbf{e}^{l+1} \overset{\mathbf{J}_{F^{l+1}}^\mathsf{T}}{\mapsto} \mathbf{e}^l \mapsto \cdots \overset{\mathbf{J}_{F^1}^\mathsf{T}}{\mapsto} \mathbf{e}^0$.

Algorithm 1: Activity Backpropagation

```
$\mathbf{e}^L \leftarrow \nabla_\mathbf{y}\mathcal{R}|_\mathbf{y} = \frac{1}{N}\sum_{(\mathbf{x},\mathbf{y}^*)\in\mathcal{S}_N} \nabla_\mathbf{y}\ell(\mathbf{y} = F_\theta(\mathbf{x}), \mathbf{y}^*)|_\mathbf{y}$
for $l \leftarrow (L-1)$ (down) to 0 do
    $\mathbf{e}^l \leftarrow \mathbf{J}_{F^{l+1}}^\mathsf{T}\mathbf{e}_k^{l+1}$
```

Jacobians of Ridge Functions

$\mathbf{x}^l = F^l(\mathbf{x}^{l-1}) = \sigma^l\left(\mathbf{W}^l\mathbf{x}^{l-1} + \mathbf{b}^l\right) = \sigma^l(\mathbf{z}^l)$.

$\mathbf{z}^l = \mathbf{W}^l\mathbf{x}^{l-1} + \mathbf{b}^l$

$\frac{\partial x_i^l}{\partial x_j^{l-1}} = \sigma'(z_i^l)\cdot\frac{\partial z_i^l}{\partial x_j^{l-1}} = \sigma'(z_i^l)\cdot w_{ij}^l$

$$\mathbf{J}_{F^l} = \begin{pmatrix} | & | & & | \\ \sigma'(\mathbf{z}^l) & \sigma'(\mathbf{z}^l) & \cdots & \sigma'(\mathbf{z}^l) \\ | & | & & | \end{pmatrix} \odot \mathbf{W}^l = \mathrm{diag}(\sigma'(\mathbf{z}^t))\mathbf{W}^t$$

Sidenote: if σ =ReLU, then $\sigma'(z) \in \{0,1\}$ which makes $\mathbf{J}_{F^l}$ a sparsified version of $\mathbf{W}^l$ (0-rows, or just the rows of $\mathbf{W}^l$).

Now we'll se how we can get from the gradients w.r.t. the activations to the gradients w.r.t. the weights. Actually, that is very easy - we just need to apply the chain rule one more time locally:

Just draw a picture of the network graph if you need to understand it better.

$$\frac{\partial\mathcal{R}}{\partial w_{ij}^l} = \underbrace{\frac{\partial\mathcal{R}}{\partial x_i^l}}_{=e_i^l}\frac{\partial x_i^l}{\partial w_{ij}^l} = e_i^l\underbrace{\frac{\partial x_i^l}{\partial z_i^l}}_{=\sigma'(z_i^l)}\underbrace{\frac{\partial z_i^l}{\partial w_{ij}^l}}_{=x_j^{l-1}} = e_i^l\sigma'(z_i^l)x_j^{l-1}$$

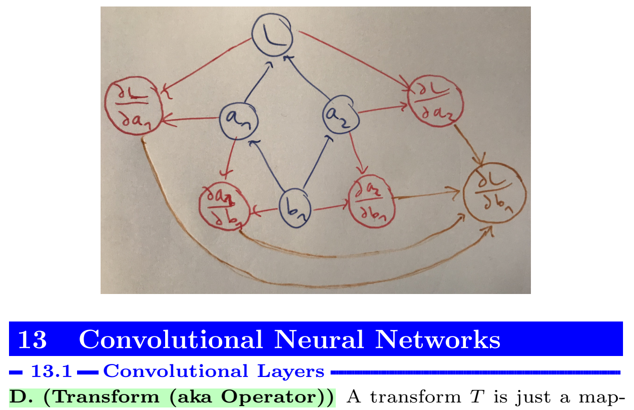
$$\frac{\partial\mathcal{R}}{\partial b_i^l} = \underbrace{\frac{\partial\mathcal{R}}{\partial x_i^l}}_{=e_i^l}\frac{\partial x_i^l}{\partial b_i^l} = e_i^l\underbrace{\frac{\partial x_i^l}{\partial z_i^l}}_{=\sigma'(z_i^l)}\underbrace{\frac{\partial z_i^l}{\partial b_i^l}}_{=1} = e_i^l\sigma'(z_i^l).$$

Note that in a way, the size of the gradient depends on the error that we have times the strength of the network's output at some node.

12.2 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.


L
$\frac{\partial L}{\partial a_1}$
$\frac{\partial L}{\partial a_2}$
a_1
a_2
$\frac{\partial a_1}{\partial b_1}$
$\frac{\partial a_2}{\partial b_1}$
b_1
$\frac{\partial L}{\partial b_1}$

13 Convolutional Neural Networks

13.1 Convolutional Layers

D. (Transform (aka Operator)) A transform T is just a mapping from one function space $\mathcal{F}$ (or a cross product of it) to another function space $\mathcal{F}'$. So $T: \mathcal{F} \to \mathcal{F}'$.

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

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

The input of this transform is a function f of t, and the output is another function Tf in terms of some other variable u.

Note that the integral boundaries, the class of input function f, and the kernel K must be defined such that Tf exists for any f in order for T to be an integral operator.

There are numerous useful integral transforms. Each is specified by a choice of the function K in two *variables*, the *kernel function*, *integral kernel*, or *nucleus* of the transform.

Some kernels have an associated *inverse kernel* $K^{-1}(u,t)$, which (roughly speaking) yields an inverse transform:

$$f(t) = \int_{u_1}^{u_2} K^{-1}(u,t)(Tf)(u)\,du.$$

T. Any integral transform is a linear transform.

C. The expectation operator is a linear operator.

Proof. Trivially follows from the linearity of the integral.

D. (Symmetric Kernel) A *symmetric kernel* K is one that is unchanged when the two variables are *permuted*. So, K is symmetric, if $K(t,u) = K(u,t)$.

Mathematical motivation: some problems are easier to solve if transformed and solved in another domain (and then possibly the solution is transformed back).

DL motivation: we'll learn the kernels.

D. (Convolution) Given two functions $f, h: \mathbb{R} \to \mathbb{R}$, 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. (Every Conv. can be Written as an Integral Transf.)

Now, a convolution of f with some function h can be seen as an *integral operator* with a kernel

$$K(u,t) = h(u-t).$$

Note that we can say the anologous thing for the convolution of h with f. The definition of the convolution shows us directly that it's *commutative*!

L. The convolution $T(\cdot) = (\cdot * g)$ is a linear transform (with $K(u,t) = g(u-t)$).

Proof. Trivially follows from the fact that we can express every convolution as an integral transform.

T. (Convolution Properties)

- Associativity $(f * g) * h = f * (g * h)$
- Commutativity $f * g = g * f$
- Bilinearity $(\alpha f + \beta g) * (\gamma y + \delta z) = \alpha\gamma(f * y) + \alpha\delta(f * z) + \cdots$ (follows from commutativity and linearity)

D. (Translation- (or Shift-) Invariant) A transform T is translation (or shift) invariant, if for any f and scalar τ,

$$f_\tau(t) := f(t-\tau) \qquad \text{(Def. shift operator } s_\tau)$$

$$(Tf_\tau)(t) = (Tf)(t-\tau). \qquad \text{(commuting of operators holds)}$$

So, an operator T is shift-invariant iff it commutes with the shift operator s_Δ.

$$\forall f: (T(s_\Delta f)) = (s_\Delta(Tf)).$$

So, the commutative diagram for this is:

$$\begin{tikzcd} f \arrow[r, "s_\Delta"] \arrow[d, "T"'] & s_\Delta f = f_\Delta \arrow[d, "T"] \\ Tf \arrow[r, "s_\Delta"] & s_\Delta(Tf) = T(f_\Delta) \end{tikzcd}$$

T. (Convolution is Translation- (or Shift-) Invariant)

Proof. $\tau_{(s,t)}((f * g)(u,v)) = (\tau_{(s,t)}(f) * g)(u,v) = (f * \tau_{(s,t)}(g))(u,v)$

$$\begin{aligned}\tau_{(s,t)}((f * g)(u,v)) &= \tau_{(s,t)}\left(\left(\sum_{i=-p}^p\sum_{j=-q}^q f(u-i, v-j)g(i,j)\right)(u,\right. \\ &= \sum_{i=-p}^p\sum_{j=-q}^q f(u+s-i, v+t-j)g(i,j) \\ &= \sum_{i=-p}^p\sum_{j=-q}^q \tau_{(s,t)}(f(u-i, v-j))g(i,j) \\ &= (\tau_{(s,t)}(f) * g)(u,v)\end{aligned}$$

□

So, to summarize a *convolution* is a *linear shift-invariant integral transform*.

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

Proof.

Let → represent the input-output relationship of a linear system.

By definition $\delta(t) \to h(t)$

Using linearity we have $a\delta(t) \to ah(t)$

Let $a = x(t')$ then $x(t')\delta(t) \to x(t')h(t)$

By shift invariance we have $\delta(t-t') \to h(t-t')$

Combining linearity and shift invariance $x(t')\delta(t-t') \to x(t')h(t-t')$

Again by linearity, we can sum many terms of this kind.

$\int_{-\infty}^{\infty} x(t')\delta(t-t')\,dt' \to \int_{-\infty}^{\infty} x(t')h(t-t')\,dt'$

But by definition of $\delta(t)$ we have that the LHS is $x(t)$, so

$x(t) \to \int_{-\infty}^{\infty} x(t')h(t-t')\,dt'$

By a change of variable on the RHS to $\tilde{t} = t - t'$ we also have

$x(t) \to \int_{-\infty}^{\infty} x(t-t')h(t')\,dt' = y(t)$ □

13.2 Discrete Time Convolutions

D. (Discrete Convolution)

For $f, h: \mathbb{Z} \to \mathbb{R}$, 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 (windows size).

D. (Multidimensional Discrete Convolution)

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

$$\begin{aligned}(f * h)[u_1,\dots,u_d] &= \sum_{t_1=-\infty}^{\infty}\cdots\sum_{t_d=-\infty}^{\infty} f(t_1,\dots,t_d)h(u_1-t_1,\dots,u_d-t_d) \\ &= \sum_{t_1=-\infty}^{\infty}\cdots\sum_{t_d=-\infty}^{\infty} f(u_1-t_1,\dots,u_d-t_d)h(t_1,\dots,t_d)\end{aligned}$$

D. (Discrete Cross-Correlation)

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

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

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

13.3 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.

13.4 Why to use Convolutions in DL

Transforms in NNs are usually: linear transform + nonlinearity. (given in convolution).

Many signals obey translation invariance, so we'd like to have translation invariant feature mpas. If the relationship of translation invariance is given in the input-output relation then this is perfect.

13.5 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)** our definition: 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*

13.6 Backpropagation for Convolutions

Exploits structural sparseness.

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 $\forall j \neq \mathcal{I}_i^l$: $\quad \frac{\partial x_i^l}{\partial x_j^{l-1}} = 0$, 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

$\mathbf{y}^{(l+1)} = \mathbf{y}^{(l)} * \mathbf{w}$

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

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

I.ote 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) $\forall i$: $\mathrm{rot180}(\mathbf{x})_i = \mathbf{x}_{(-i)}$.

13.7 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 ($m \ll n$, $m \approx \log(n)$).

13.8 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.

13.9 Pooling

The most frequently used pooling function is: *max pooling*. But one can imagine using other pooling functions, such as: min, avg, softmax,

D. (Max-Pooling)

Max pooling works, as follows, if we define a window size of $r = 3$ (in 1D or 2D), then

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

So, in general we just take the maximum over a small "patch"/"neighbourhood" of some units.

T. (Max-Pooling: Invariance)

Let $\mathcal{T}$ be the set of invertible transformations (e.g., integral transforms, integal operators). Then $\mathcal{T}$ forms a group w.r.t. function composition: $\langle \mathcal{T}, \circ, ^{-1}, \mathrm{id} \rangle$.

13.10 Sub-Sampling (aka "Strides")

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

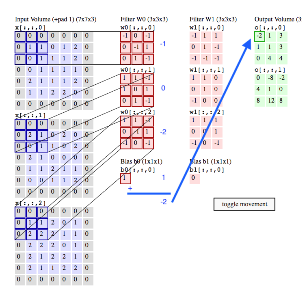
D. (Sub-Sampling) Hereby the temporal/spatial resolution is reduced.
Com. Often, the sub-sampling is done via a max-pooling according to some interval step size (a.k.a. stride)

− Loss of information
+
+ Dimensionality reduction
+ Increase of efficiency

13.11 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.

13.12 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), and
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

13.13 Famous CNN Architectures

13.13.1 LeNet, 1989

MNIST, 2 Convolutional Layers + 2 Fully-connected layers

13.13.2 LeNet5

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

13.13.3 AlexNet

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

13.13.4 Inception Module

Now, a problem that arose with this ever deeper and deeper channels were 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).

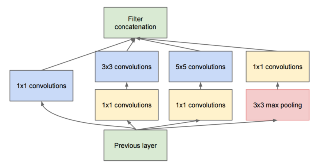
This is what gave rise to the inception module:

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 no convolution), aka "network within a network".

13.13.5 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.

13.14 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).

13.15 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 $(m + 2p) \times (n + 2q)$.

14 Optimization

14.1 Learning as Optimization

Machine learning uses optimization, but it's *not equal* to optimization for two reasons:

1. The empirical risk is only a *proxy* for the expected risk
2. The loss function may only be a *surrogate*

14.2 Objectives as Expectations

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

The typical structure of a learning objective in a NN is a *large finite sum* (over all training instances). Accuracy-complexity trade-off: in practice we subsample terms in the sum, by using *mini-batches* of the training data (so we'll get something close to the true gradient - but not exactly). The idea behind it, is that everything will work out in expectation. Further, we favour cheap and imprecise computations over many datapoints rather than precise and expensive computations over a few datapoints.

14.3 Gradient Descent

$\theta(t + 1) = \theta(t) - \eta_t \nabla_\theta \mathcal{R}$,

$\dot{\theta} = -\eta_t \nabla_\theta \mathcal{R}(\theta)$

14.4 Gradient Descent: Classic Analysis

In classical machine learning we have a *convex* objective $\mathcal{R}$. And we denote

- $\mathcal{R}^*$ as the minimum of $\mathcal{R}$
- θ^* as the optimal set of parameters (the minimizer of $\mathcal{R}$)

So we have $\forall \theta \neq \theta^*$: $\mathcal{R}^* := \mathcal{R}(\theta^*) \leq \mathcal{R}(\theta)$.

D. (Strictly Convex Objective) → objective has only one (a unique) minimum.

$$\forall \theta \neq \theta^*: \mathcal{R}^* := \mathcal{R}(\theta^*) < \mathcal{R}(\theta).$$

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

$$\forall x_1, x_2 \in X: \quad d_Y(f(x_1), f(x_2)) \leq L \cdot d_X(x_1, x_2).$$

Hereby, L is referred to as a *Lipschitz constant* for f.

- If $L = 1$ the function is called a *short map*
- If $0 \leq L < 1$ and f maps a metric space to itself, so $f\colon X \to X$, then the function f is called a *contraction*.

In particular, a map $f\colon \mathbb{R}^n \to \mathbb{R}^m$ is called Lipschitz continous if there exists a $L \in \mathbb{R}_0^+$ such that

$$\forall \mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^n: \quad \underbrace{\|f(\mathbf{x}_1) - f(\mathbf{x}_2)\|}_{=d_{\mathbb{R}^m}(f(\mathbf{x}_1), f(\mathbf{x}_2))} \leq L \cdot \underbrace{\|\mathbf{x}_1 - \mathbf{x}_2\|}_{d_{\mathbb{R}^n}(\mathbf{x}_1, \mathbf{x}_2)}.$$

Ex. So for our risk function $\mathcal{R}$, we say that the gradient of it

$$\nabla_\theta \mathcal{R}\colon \Omega \to \Omega \quad \text{where } \Omega = \mathbb{R}^n$$

is L-Lipschitz continous, if it holds that

$$\forall \theta_1, \theta_2 \in \Theta: \quad \|\nabla_\theta \mathcal{R}(\theta_1) - \nabla_\theta \mathcal{R}(\theta_2)\| \leq L \|\theta_1 - \theta_2\|$$

Com. So, the L tells us how big the gradient could be.

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 ⊆ Lipschitz continuous ⊆ (Uniformly) continous

Com. It's important that the space is bounded. Because for example only on a compact subset $[a, b] \subset \mathbb{R}$ the function e^x is Lipschitz continous. On $\mathbb{R}$ the function e^x is not Lipschitz continuous, as it gets arbitrarily steep.

`https://en.wikipedia.org/wiki/Lipschitz_continuity#Properties`

T. An everywhere differentiable function $f\colon \mathbb{R} \to \mathbb{R}$ is Lipschitz continuous (with $L = \sup |f'(x)|$) iff it has *bounded first derivatives*.

T. In particular any continously differentiable function is *locally* Lipschitz continuous. As continuous functions are bounded on an interval, so its gradient is locally bounded as well.

T. If $\mathcal{R}$ is convex with L-Lipschitz-continuous gradients then we have that

$$\mathcal{R}(\theta(t)) - \mathcal{R}^* \leq \frac{2L}{t+1} \|\theta(0) - \theta^*\|^2 \in \mathcal{O}(t^{-1})$$

Com. So we have a polynomial (linear) convergence rate of θ towards the optimal parameter θ^* (note: just in the convex setting!). As we can see, the convergence time is bounded by a time that depends on our initial guess, and the Lipschitz constant L.

Com. Usually one value for η that people use in this setting is $\eta := \frac{1}{L}$ or $\eta := \frac{2}{L}$.

Proof. See here

D. (Convex Set) A set $S \subseteq \mathbb{R}^d$ is called *convex* if
$\forall \mathbf{x}, \mathbf{x}' \in \Sigma$, $\forall \lambda \in [0, 1]$: $\quad \lambda \mathbf{x} + (1 - \lambda)\mathbf{x}' \in S$.

Com. Any point on the line between two points is within the set.

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

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

Com. convex combination of two points < evaluation of convex combination of two points.

Com. Another way to formulate that f is convex function is to say that the epi-graph of f is a convex set.

T. Every local optimum of a convex function is a global optimum.

T. (Operations that Preserve Convexity)

- $-f$ is concave if and only if f is convex
- nonnegative weighted sums
- point/element-wise maximum $\max(f_1(\mathbf{x}), \ldots, f_n(\mathbf{x}))$
- composition with non-decreasing function, e.g. $e^{f(\mathbf{x})}$
- composition with affine mapping: $f(\mathbf{A}\mathbf{x} + \mathbf{b})$
- restriction to a line (of convex set domain)

D. (Strictly Convex Function) f is called *strictly convex* if

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

D. (Strongly Convex Function) A differentiable function f is called *μ-strongly convex* if the following inequality holds for all points $\mathbf{x}, \mathbf{y}$ in its domain:

$$\forall \mathbf{x}\mathbf{y}: \quad \langle \nabla f(\mathbf{y}) - \nabla f(\mathbf{x}), \mathbf{y} - \mathbf{x} \rangle \geq \mu \|\mathbf{y} - \mathbf{x}\|_2^2$$

where $\|\cdot\|$ is any norm. An equivalent condition is the following:

$$\forall \mathbf{x}, \mathbf{y}: \quad f(\mathbf{y}) \geq f(\mathbf{x}) + \nabla f(\mathbf{x})^\mathsf{T}(\mathbf{y} - \mathbf{x}) + \frac{\mu}{2}\|\mathbf{y} - \mathbf{x}\|_2^2.$$

Com. The concept of strong convexity extends and parametrizes the notion of strict convexity. A strongly convex function is also strictly convex, but not vice versa. Notice how the definition of strong convexity approaches the definition for strict convexity as $\mu \to 0$, and is identical to the definition of a convex function when $\mu = 0$. Despite this, functions exist that are strictly convex, but are not strongly convex for any $\mu > 0$.

T. Now, when $\mathcal{R}$ is μ-strongly convex in θ and its gradient is L-Lipschitz continuous $\Longrightarrow$

$$\mathcal{R}(\theta(t)) - \mathcal{R}^* \leq \left(1 - \frac{\mu}{L}\right)^t (\mathcal{R}(\theta(0)) - \mathcal{R}^*) \in \mathcal{O}\left(\left(1 - \frac{\mu}{L}\right)^t\right)$$

So we have

- an exponential convergence ("linear rate")
- and the rate depends adversely on the condition number $\frac{L}{\mu}$. So we want the maximum gradient to be small, and we want the curvature to be large (which are somewhat contrary desires, but ideally the condition number is very close to 1).

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

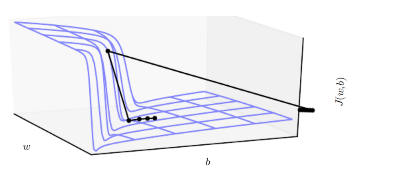
Com. The trick used in the Nesterov approach is *momentum*.

14.5 Optimization Challenges in NNs: Curvatures

When it comes to NNs the objective is usually non-convex. So this is for example an objective that we may get that is non-convex. Still, we can apply gradient descent in this setting. And if we respect the rule of choosing the learning rate as $\eta = \frac{1}{L}$ where L is the Lipschitz-constant of the function, then usually, we're fine.

Models with multiplication of many weights (depth, recurrence): sharp non-linearities

- Very large Lipschitz constant
- Would theoretically require very small gradient steps → very slow optimization



Motivates gradient clipping heuristics and learning rate decay.

So the problem is if we have sharp non-linearities, then there are two approaches to solve this

- one is to be very conservative and only do small update steps by choosing a very small learning rate.
- or we are courageous and due huge jumps as depicted in the image.

So this is kindof the typical problem that, at least some people think, happens with NNs.

Typical approaches are to clip the gradient when it gets too large, or use a decreasing learning rate (in terms of time).

Now, the problem is not that the cliff is very steep. The problem is the curvature. Because when we take the gradient, the gradient is actually constant on the wall of the cliff. Let's have a look at this through some equations.

Now, let's evaluate what the risk is at some point, plus some gradient step. If we do the 2nd-order Taylor expansion of that, then we get

$$\mathcal{R}(\theta - \eta \nabla_\theta \mathcal{R}(\theta)) \overset{\text{Taylor}}{\approx} \mathcal{R}(\theta) - \eta \|\nabla_\theta \mathcal{R}(\theta)\|_2^2 + \frac{\eta^2}{2} \underbrace{\nabla_\theta \mathcal{R}(\theta)^\mathsf{T} \mathbf{H} \nabla_\theta \mathcal{R}(\theta)}_{\|\nabla_\theta \mathcal{R}(\theta)\|_{\mathbf{H}}^2}$$

where

$$\mathbf{H} := \nabla^2 \mathcal{R}(\theta) \qquad \text{(Hessian matrix)}$$

Now, what we want is that the rest of the sum is negative. If that is the case, because then we're improving our cost function. If not, then we're basically diverging.

So,

- the first term $-\eta \|\nabla_\theta \mathcal{R}(\theta)\|_2^2$ will obviously be negative, as it's a negative factor of a norm
- the second term will always be positive, as the hessian matrix is positive semi-definite. Fortunately, we're squaring η, which may be already small, so the term is small. However, if the Hessian is ill-conditioned (as in the cliff-situation (curvature)). Then we can have a very large positive value in the second term. So what then can happen is that

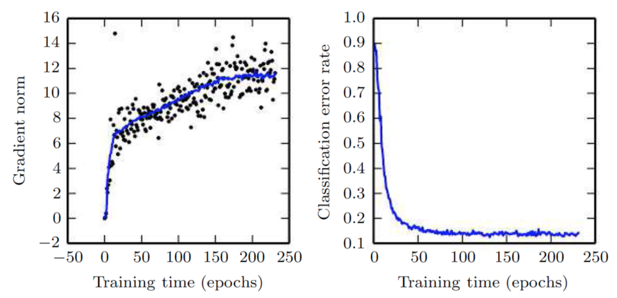
$$\frac{\eta}{2}\|\nabla_\theta \mathcal{R}(\theta)\|_{\mathbf{H}}^2 \gtrsim \|\nabla_\theta \mathcal{R}(\theta)\|^2$$

So, the hessian term becomes much larger than the gradient. So we're not improving our cost function.

- and the remaining terms, will be negative (as defined by the Taylor sum)

So a typical remedy for first-order methods is to take very small step sizes η.

However, things bececome even stranger because of the curvature. As we can see, the gradient norm gets larger and larger the more we train (can be checked empirically). And the gradient norm also tends to have larger fluctuations. And as we can see, starting at some point, the error just fluctuates around at a certain level. Actually, one might assume that as we're getting closer to the minimum, the gradient should get smaller and smaller, as the objective gets flatter and flatter at the optimal point - but that's actually not the case!



This is probably so, because we're dealing with large curvatures when reaching (or getting close to) the optimal parameters. So with gradient descent we may not arrive at a critical point of any kind, and this also motivates to use more and more decreasing learning rates, the closer we get to the optimal parameters. Note that this graph was built using the MNIST dataset and some CNN.

Note that there exist many architectures where the final gradient was very large, and still they are used in practice, and people are quite happy with them.

14.6 Optimization Challenges in NNs: Local Minima

At the beginning people were happy when they were doing convex optimization because there was a single optimum and it was reachable. And then when people started using non-convex optimization they were afraid of getting into non-optimal local minima and getting stuck there.

Neural network cost functions can have many local minima and/or saddle points - and this is typical. Gradient descent can get stuck.

Questions that have been looked at are

- Ar local minima a practical issue? Somtimes not: Gori & Tesi, 1992
- Do local minima even exist? Sometimes not (auto encoder): Baldi & Hornik, 1989
- Are local minima typically worse? often not (large networks): e.g., Choromanska et al, 2015
- Can we understand the learning dynamics? Deep linear case has similarities with non-linear case, e.g., Saxe et al., 2013

So it turns out that the non-convexity is actually not so much of an issue. It turns out that when we go to very high dimensions, the number of local minima VS the number of saddle points (gradient is zero, but non-optimal) is very small - so we're much more likely to end up in a saddle point. However, in practice, if we do SGD there is some stochasticity that will make our gradient move. Then, after waiting for a while we'll exit the saddle point.

Now, next we'll look at the insights that were gained by the paper of Saxe.

14.7 Least Squares: Single Layer Lin. Netw.

Let's assume that we have some inputs $\mathbf{x} \in \mathbb{R}^n = \mathcal{X}$ and some outputs $\mathbf{y} \in \mathbb{R}^m = \mathcal{Y}$, and we have a very simple architecture, that will take our input $\mathbf{x}$ and transform it to some output in $\mathcal{Y}$ the output space

$$F(\mathbf{x}) = \mathbf{A}\mathbf{x}, \qquad \mathbf{A} \in \mathbb{R}^{m \times n}$$

So then, we can define our risk as

$$\mathcal{R}(\mathbf{A}) = \mathbb{E}\left[\|\mathbf{y} - \mathbf{A}\mathbf{x}\|_2^2\right].$$

so we're taking the expectation with regard to the empirical distribution (averaginv over all $(\mathbf{x}, \mathbf{y})$-training pairs).

Now, to make things simpler, we assume that the inputs are whitened, that means that

$$\mathbb{E}\left[\mathbf{x}\mathbf{x}^\mathsf{T}\right] = \mathbf{I}$$

So if the mean $\mu = \mathbf{o}$, then our input is uncorrelated, and every feature is of variance 1.

Further, have a look at the following trace identities

$$\mathbf{v}^\mathsf{T}\mathbf{w} = \sum_i v_i w_i = \mathsf{Tr}\left(\mathbf{v}\mathbf{w}^\mathsf{T}\right) = \mathsf{Tr}\left(\mathbf{w}\mathbf{v}^\mathsf{T}\right)$$

$$\mathsf{Tr}\left(\mathbf{A} + \mathbf{B}\right) = \mathsf{Tr}\left(\mathbf{A}\right) + \mathsf{Tr}\left(\mathbf{B}\right)$$

$$\mathbb{E}\left[\mathsf{Tr}\left(\boldsymbol{X}\right)\right] = \mathsf{Tr}\left(\mathbb{E}\left[\boldsymbol{X}\right]\right) \qquad \text{(linearity of trace and exp.)}$$

Now let's see if we can rewrite the risk differently

$$\begin{aligned}\mathcal{R}(\underbrace{\mathbf{A}}_{=\theta}) &= \mathbb{E}\left[\|\mathbf{y} - \mathbf{A}\mathbf{x}\|_2^2\right] \\ &= \mathbb{E}\left[\mathsf{Tr}\left((\mathbf{y} - \mathbf{A})(\mathbf{y} - \mathbf{A})^\mathsf{T}\right)\right]\end{aligned}$$

Using the linearity of the expectation and the trace, and some trace identities leads us to

$$\begin{aligned}&= \mathsf{Tr}\left(\mathbb{E}\left[\mathbf{y}\mathbf{y}^\mathsf{T}\right]\right) - 2\mathsf{Tr}(\mathbf{A}\underbrace{\mathbb{E}\left[\mathbf{x}\mathbf{y}^\mathsf{T}\right]}_{=:\mathbf{\Gamma}^\mathsf{T},\ \mathbf{\Gamma} \in \mathbb{R}^{m \times n}}) + \mathsf{Tr}(\mathbf{A}\underbrace{\mathbb{E}\left[\mathbf{x}\mathbf{x}^\mathsf{T}\right]}_{=\mathbf{I}\text{ (by ass.)}}\mathbf{A}^\mathsf{T}) \\ &= \underbrace{\mathsf{Tr}\left(\mathbb{E}\left[\mathbf{y}\mathbf{y}^\mathsf{T}\right]\right)}_{\text{indep. of }\mathbf{A}} - 2\mathsf{Tr}\left(\mathbf{A}\mathbf{\Gamma}^\mathsf{T}\right) + \mathsf{Tr}\left(\mathbf{A}\mathbf{A}^\mathsf{T}\right)\end{aligned}$$

Now, let's see how we can minimize the risk

$$\begin{aligned}\mathbf{A}^* &= \arg\min_{\mathbf{A}} \mathcal{R}(\mathbf{A}) \\ &= \arg\min_{\mathbf{A}} -2\mathsf{Tr}\left(\mathbf{A}\mathbf{\Gamma}^\mathsf{T}\right) + \mathsf{Tr}\left(\mathbf{A}\mathbf{A}^\mathsf{T}\right)\end{aligned}$$

Now it's hard to continue from here, so we'll just do it via computing the gradient (generalized) and setting it equal to zero:

$$\begin{aligned}\mathcal{R}(\mathbf{A}) &= \mathsf{Tr}\left(\mathbb{E}\left[\mathbf{y}\mathbf{y}^\mathsf{T}\right]\right) - 2\mathsf{Tr}\left(\mathbf{A}\mathbf{\Gamma}^\mathsf{T}\right) + \mathsf{Tr}\left(\mathbf{A}\mathbf{A}^\mathsf{T}\right) \\ \nabla_{\mathbf{A}}\mathcal{R}(\mathbf{A}) &= -2\mathbf{\Gamma} + 2\mathbf{A} = 2(\mathbf{A} - \mathbf{\Gamma})\end{aligned}$$

So, obviously, the derivative is zero for

$$\mathbf{A}^* = \mathbf{\Gamma} = \mathbb{E}\left[\mathbf{x}\mathbf{y}^\mathsf{T}\right]^\mathsf{T} = \mathbb{E}\left[\mathbf{y}\mathbf{x}^\mathsf{T}\right] \overset{\text{emp. dist}}{=} \frac{1}{N}\sum_{i=1}^{N} \mathbf{y}[i]\mathbf{x}[i]^\mathsf{T}.$$

Note that when computing the derivative we've used the following trace differentiation rules (cf. wikipedia, The Matrix Cookbook)

$$\nabla_{\mathbf{A}}\mathsf{Tr}\left(\mathbf{A}\mathbf{A}^\mathsf{T}\right) = 2\mathbf{A} \qquad \nabla_{\mathbf{A}}\mathsf{Tr}\left(\mathbf{A}\mathbf{B}\right) = \mathbf{B}^\mathsf{T}$$

Note that one could have solved the solution also through the following way, by recognizing that $\mathbf{A}(t)$ follows the following differential equation

$$\dot{\mathbf{A}}(t) = -\eta \nabla_{\mathbf{A}}\mathcal{R}(\mathbf{A}(t)) = 2\eta(\mathbf{\Gamma} - \mathbf{A}(t)) = 2\eta\mathbf{\Gamma} - 2\eta\mathbf{A}(t).$$

So rearranging the equation for $\mathbf{A}(t)$ we get

$$\mathbf{A}(t) = -\frac{1}{2}\eta\dot{\mathbf{A}}(t) + \mathbf{\Gamma}$$

And now, since we're using gradient descent, we'll converge, and the gradients will go to zero, hence

$$\lim_{t \to \infty} \mathbf{A}(t) = -\frac{1}{2}\eta\underbrace{\left(\lim_{t \to \infty}\dot{\mathbf{A}}(t)\right)}_{\to 0} + \mathbf{\Gamma} = \mathbf{\Gamma}.$$

So $\mathbf{A}(t)$ will converge to $\mathbf{\Gamma}$.

14.8 Least Squares: Two-Layer Lin. Netw.

Now, the question is what happens, when we build a two-layer linear network (again with the squared error), with no nonlinearity. So we'll have a linear mapping (that is composed of two linear mappings)

$$F(\mathbf{x}) = \mathbf{A}\mathbf{x} = \mathbf{Q}\mathbf{W}\mathbf{x}$$

Now, from what we've seen before, we can express the risk as (due to trace identities, trace linearity, etc.) just by replacing $\mathbf{A} = \mathbf{Q}\mathbf{W}$, so

$$\mathcal{R}(\mathbf{Q}, \mathbf{W}) = \text{const.} + \mathsf{Tr}\left((\mathbf{Q}\mathbf{W})(\mathbf{Q}\mathbf{W})^\mathsf{T}\right) - 2\mathsf{Tr}\left(\mathbf{Q}\mathbf{W}\mathbf{\Gamma}^\mathsf{T}\right)$$

Now, taking the derivatives w.r.t. the parameters, we get (using the chain rule)

$$\begin{aligned}\nabla_{\mathbf{Q}}\mathcal{R}(\mathbf{Q}, \mathbf{W}) &= \frac{\partial \mathcal{R}(\mathbf{A})}{\partial \mathbf{A}}\frac{\partial \mathbf{A}}{\partial \mathbf{Q}} \\ \nabla_{\mathbf{W}}\mathcal{R}(\mathbf{Q}, \mathbf{W}) &= \frac{\partial \mathcal{R}(\mathbf{A})}{\partial \mathbf{A}}\frac{\partial \mathbf{A}}{\partial \mathbf{W}}\end{aligned}$$

Which in the end gives us

$$\begin{aligned}\nabla_{\mathbf{Q}}\mathcal{R}(\mathbf{Q}, \mathbf{W}) &= 2\underbrace{\mathbf{Q}\mathbf{W}}_{=\mathbf{A}}\mathbf{W}^\mathsf{T} - 2\mathbf{\Gamma}\mathbf{W}^\mathsf{T} = 2(\mathbf{A} - \mathbf{\Gamma})\mathbf{W}^\mathsf{T} \\ \nabla_{\mathbf{W}}\mathcal{R}(\mathbf{Q}, \mathbf{W}) &= 2\mathbf{Q}^\mathsf{T}\underbrace{\mathbf{Q}\mathbf{W}}_{=\mathbf{A}} - 2\mathbf{Q}^\mathsf{T}\mathbf{\Gamma} = 2\mathbf{Q}^\mathsf{T}(\mathbf{A} - \mathbf{\Gamma})\end{aligned}$$

Now, what we'll do is we'll perform the SVD of $\mathbf{\Gamma}$ (we can do this since $\mathbf{\Gamma}$ only depends on the data, it was the correlation matrix between the inputs and the outputs). So,

$$\mathbf{\Gamma} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\mathsf{T}.$$

Now, we'll linearly transform the variables:

$$\begin{aligned}\widetilde{\mathbf{Q}} = \mathbf{U}^\mathsf{T}\mathbf{Q} &\iff \mathbf{Q} = \mathbf{U}\widetilde{\mathbf{Q}} \\ \widetilde{\mathbf{W}} = \mathbf{W}\mathbf{V} &\iff \mathbf{W} = \widetilde{\mathbf{W}}\mathbf{V}^\mathsf{T}\end{aligned}$$

Then, we have that the common term in the gradients $\mathbf{A} - \mathbf{\Gamma}$ can be written as follows

$$\begin{aligned}\mathbf{A} - \mathbf{\Gamma} &= \mathbf{Q}\mathbf{W} - \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\mathsf{T} \\ &= \mathbf{U}\underbrace{\mathbf{U}^\mathsf{T}\mathbf{Q}}_{=\widetilde{\mathbf{Q}}}\underbrace{\mathbf{W}\mathbf{V}}_{=\widetilde{\mathbf{W}}}\mathbf{V}^\mathsf{T} - \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\mathsf{T} \\ &= \mathbf{U}(\widetilde{\mathbf{Q}}\widetilde{\mathbf{W}} - \mathbf{\Sigma})\mathbf{V}^\mathsf{T}\end{aligned}$$

And we can re-express the risk in terms of $\widetilde{\mathbf{Q}}, \widetilde{\mathbf{W}}$ as follows:

$$\begin{aligned}\mathcal{R}(\widetilde{\mathbf{Q}}, \widetilde{\mathbf{W}}) = \text{const.} &+ \mathsf{Tr}\left((\mathbf{U}^\mathsf{T}\widetilde{\mathbf{Q}}\widetilde{\mathbf{W}}\mathbf{V}^\mathsf{T})(\mathbf{U}^\mathsf{T}\widetilde{\mathbf{Q}}\widetilde{\mathbf{W}}\mathbf{V}^\mathsf{T})^\mathsf{T}\right) \\ &- 2\mathsf{Tr}\left((\mathbf{U}^\mathsf{T}\widetilde{\mathbf{Q}}\widetilde{\mathbf{W}}\mathbf{V}^\mathsf{T})\mathbf{\Gamma}^\mathsf{T}\right)\end{aligned}$$

And we can compute the corresponding projected gradients in terms of $\widetilde{\mathbf{Q}}, \widetilde{\mathbf{W}}$ as follows.

$$\begin{aligned}\nabla_{\widetilde{\mathbf{Q}}}\mathcal{R}(\widetilde{\mathbf{Q}}, \widetilde{\mathbf{W}}) = \mathbf{U}^\mathsf{T}\nabla_{\mathbf{Q}}\mathcal{R}(\mathbf{Q}, \mathbf{W}) = \cdots \\ \cdots = \cdots\end{aligned}$$

So, as we can easily see, the gradients can be computed through the rules of linearity.

14.9 Stochastic Gradient Descent

So, how do we modify the gradient descent approach to work with batches?
The idea in *stochastic* gradient descent is to choose the update directin $\boldsymbol{V}$ *at random*, such that

$$\mathbb{E}\left[\boldsymbol{V}\right] = -\nabla_\theta \mathcal{R}.$$

So, the randomization scheme is unbiased.

So SGD works via subsampling. So we pick a random subset

$$\mathcal{S}_K \subseteq \mathcal{S}_N, \qquad K \leq N. \quad \text{(usually } K \ll N\text{)}$$

And since we're picking $\mathcal{S}_K$ at random, note how

$$\mathbb{E}\left[\mathcal{R}(\mathcal{S}_K)\right] = \mathcal{R}(\mathcal{S}_N)$$

And thus it also holds for the gradient that

$$\nabla_\theta \mathbb{E}\left[\mathcal{R}(\mathcal{S}_K)\right] \overset{\text{lin.}}{=} \mathbb{E}\left[\nabla_\theta \mathcal{R}(\mathcal{S}_K)\right] = \nabla_\theta \mathcal{R}(\mathcal{S}_N).$$

So, with SGD, we just do gradient descent, where at each t we'll do a randomization of $\mathcal{S}_K$, so

$$\theta(t + 1) = \theta(t) - \eta \nabla_\theta \mathcal{R}(\theta(t); \mathcal{S}_K(t))$$

In practice, what is done is we *permute* the instances, and break them up into mini-batches. So we're actually not doing random sampling at every timestep. We're rather doing a random partitioning of the training instances into batches. This gives rise to the following definitions:

D. (Epoch = one sweep through the whole data)

- take the data batch by batch
- compute one gradient by batch
- harder to analyze theoretically
- typically works better in practice
- no permutation → danger of "unlearning". Let's suppose we're training for MNIST, and we're first doing the gradient steps for the 1s, then for the 2s, etc. This will completely bias the gradient and lead us into the wrong direction at every gradient step. In the end we'll never converge to a good solution.

Com. It happens that this way SGD is a bit harder to analyze theoretically, but for NNs it works quite well in practice.

D. (Minibatch Size)

- "Standard SGD": $k = 1$, this is for classical SGD. However, if we only take one instance, the error on the gradient direction will be too large.
- but: larger k is better for utilizing concurrency in GPUs or multicore CPUs. And we'll also get more accuracy. Of course, this requires more computation per gradient step, so we'll have to compute more to do one step, but it pays of in terms of accuracy of the gradient (and it can be parallelized anyways).

Com. In practice we just need to ensure that the batch is sufficiently big to have a representative subsample to compute a reliable estimate of the gradient (in order to converge). Further, we usually use batch-sizes of 2^k for some $k \in \mathbb{N}$.

14.9.1 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.

14.9.2 Practicalities

Now, let's have a look at some of the practicalities:

- Almost none of the analysis applies to the non-convex case
- Choosing a learning rate schedule can be a nuisance
- Fast decay schedules may lead to super-slow convergence
- In practice, we tend to use larger step sizes and level out at a minimal step size. The justification behind this that the SGD with a fixed step size is known to converge to a ball around the optimum (strongly convex case). So we may use

$$\eta_t = \max(0.001, \frac{1}{t}).$$

- Further, there's the common belief that the stochasticity of the SGD is a "feature", since it may help to escape from reagions with small gradients via perturbations.

14.9.3 Momentum

Accumulate the gradient over several updates (as a geometrically weighted average). The momentum (averaging) keeps the gradient moving better towards the optimum (instead of zig-zagging).

Initialization: $\alpha = 0.95$ (typically), $\mathbf{m}(0) = \mathbf{o}$.

Then at every timestep $t = 1, 2, \ldots$

$\mathbf{m}(t) = \alpha\mathbf{m}(t - 1) - (1 - \alpha)\nabla_\theta \mathcal{R}(\boldsymbol{\theta}(t - 1))$,

$\widehat{\mathbf{m}}(t) = \frac{\mathbf{m}(t)}{(1-\alpha)^t}$ $\quad$ (bias correction, otw. gradient is too small at beginning)

$\boldsymbol{\theta}(t) = \boldsymbol{\theta}(t - 1) - \eta\widehat{\mathbf{m}}(t)$ $\quad$ (update parameters)

Usually it's good to choose a small alpha (0.5) at the beginning, and only towards the end, we'll increase alpha to 0.99 to accumulate and average the speed.

14.9.4 Nesterov Momentum

First jump, and then compute the momentum based on the gradient at the place that we'll land (seems to work better in practice).

$\boldsymbol{\theta}(t + 1) = \boldsymbol{\theta}(t) + \eta\alpha\widehat{\mathbf{m}}(t)$ (jump first)

$\mathbf{v}(t+1) = \alpha\mathbf{v}(t) + \epsilon\nabla_\theta \mathcal{R}(\boldsymbol{\theta}(t) + \alpha\widehat{\mathbf{m}}(t))$. (and then correct the jump with the gradient at the place that we jumped to)

14.9.5 AdaGrad

With AdaGrad we consider the entire history of gradients and put all the gradients ito a *gradient matrix*, so

$$\theta \in \mathbb{R}^d, \quad \mathbf{G} \in \mathbb{R}^{d \times t_{\max}}, \quad g_{it} = \left.\frac{\partial \mathcal{R}(t)}{\partial \theta_i}\right|_{\theta = \theta(t)}$$

Then we compute the (partial) row sums of squares of $\mathbf{G}$ (note: not the gradient norms! → rows!)

$$\gamma_i^2(t) := \sum_{s=1}^{t} g_{is}^2.$$

And then we adapt the gradient stepsize for each dimension as follows:

$$\theta_i(t + 1) = \theta_i(t) - \frac{\eta}{\delta + \gamma_i(t)}\nabla_\theta \mathcal{R}(t), \qquad \delta > 0 \text{ (small)}$$

This will transform the gradient such as if the loss landscape would be in a more isometric shape. It will scale the gradient appropriately into each dimension. So instead of having a valley, we'll have a nice round hole again. This avoids this typical situation where the gradient descent boundes left and right in the valley, instead of walking down the valley.

In practice a variant of AdaGrad (RMSprop) is used. Intuitively: the learning rate decays faster for weights that have seen significant updates.

Theoretical justification: regret bounds for convex objectives (Duchi, Hazan, Singer, 2011) (out of scope for this lecture).

So, Tieleman & Hinton came up with a "non-convex variant of AdaGrad" in 2012:

$$\gamma_i^2(t) := \sum_{s=1}^{t} \rho^{t-s} g_{is}^2, \qquad \rho < 1.$$

This is just a moving average, which is exponentially weighted. The weight decays exponentially over time, and thus .

It turns out that this optimizer works very nice some times.

14.9.6 ADAM

Adam is probably the most popular optimizer today. It takes the best of both worlds: AdaGrad (adapting the gradient) + Momentum. However, more parameters to tune (β_1, β_2).

Initialization: $\mathbf{m}(0) = \mathbf{o}$, $\mathbf{v}(0) = \mathbf{o}$

Typical values: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$

Then at every timestep $t = 1, 2, \ldots$

$\mathbf{g}(t) = \nabla_\theta \mathcal{R}(\boldsymbol{\theta}(t - 1))$ (get the gradient)

$\mathbf{m}(t) = \beta_1\mathbf{m}(t - 1) + (1 - \beta_1)\mathbf{g}(t)$ (update the biased first moment estimate)

$\mathbf{v}(t) = \beta_2\mathbf{v}(t-1) + (1-\beta_2)\mathbf{g}(t)^2$ (update the b. second raw moment estimate)

$\hat{\mathbf{m}}(t) = \mathbf{m}(t)(1 - \beta_1^t)$ (bias correction first moment estimate)

$\hat{\mathbf{v}}(t) = \mathbf{v}(t)(1 - \beta_2^t)$ (bias correction second raw moment estimate)

$\boldsymbol{\theta}(t) = \boldsymbol{\theta}(t - 1) + \eta\hat{\mathbf{m}}(t)/(\sqrt{\hat{\mathbf{v}}(t)} + \epsilon)$ (update params)

14.10 Optimization Heuristics

Polyak Averaging may bring us good guarantees if we have a convex loss (on average). However, for DL it's not ideal. The reason are

- if we may want to have an idea over what the gradient over the whole dataset would be, then we'd have to swipe over all the dataset which will take a lot of time. So, it's nood a good idea (we'll get a better but slower convergence). So what people do in practice instead is that they just run a weighted average to forget what was in the past. Usually, the weighted
-

14.10.1 Batch Normalization

Batch normalization (Ioffe & Szegedy, 2015) is one of the most controversial but most useful tricks in DL.

One of the big problems that we have when we optimize NNs, is that usually there exist strong dependencies between the weights in various layers (recall: we also saw that the gradients interact with each other through complex dynamics). So it's hard to find a suitable learning rate for all the situations of the weights. The dynamics were fine in this case, but if we have a large network it might not work out, and we may have to wait a long time until the dynamics diminish and lead to the solution. What batch normalization tries to achieve is to remove the dependencies between the layers. So the learning algorithm can optimize the weights of each layer independently. Of course, that's not really what happens, but anyways that's the idea behind it.

Let's have alook at a toy example to illustrate this is: a deep linear network with one unit per layer:

$$y = w_1 w_2 \cdot \cdots \cdot w_L x = \left(\prod_{l=1}^{L} w_l\right) x$$

For later notation, let us collect all the weights in a set

$$W := \{w_1, \ldots, w_L\}$$

After the gradient step we'll have the following situation:

$$y^{\text{new}} = \left(\prod_{l=1}^{L}\left(w_l - \eta\frac{\partial \mathcal{R}}{\partial w_l}\right)\right) x$$

So what actually happens is that if we take the term for y^{new} and we expand it this leads to terms of up to order L.

$$\begin{aligned}&= \left(w_1 - \eta\frac{\partial \mathcal{R}}{\partial w_1}\right)\left(w_2 - \eta\frac{\partial \mathcal{R}}{\partial w_2}\right)\cdots\left(w_L - \eta\frac{\partial \mathcal{R}}{\partial w_L}\right) x \\ &= \left(\left(\prod_{l=1}^{L} w_l\right) - \eta\frac{\partial \mathcal{R}}{\partial w_1}\left(\prod_{l=2}^{L} w_l\right) + \cdots + (-\eta)^L\left(\prod_{l=1}^{L}\frac{\partial \mathcal{R}}{\partial w_l}\right)\right) x \\ &= \underbrace{\left(\prod_{l=1}^{L} w_l\right) x}_{=y} - \eta\frac{\partial \mathcal{R}}{\partial w_1}\left(\prod_{l=2}^{L} w_l\right) x + \cdots + (-\eta)^L\left(\prod_{l=1}^{L}\frac{\partial \mathcal{R}}{\partial w_l}\right) x \\ &= y \underbrace{- \eta\frac{\partial \mathcal{R}}{\partial w_1}\left(\prod_{l=2}^{L} w_l\right) x + \cdots + (-\eta)^L\left(\prod_{l=1}^{L}\frac{\partial \mathcal{R}}{\partial w_l}\right) x}_{(*)\text{ significant?}} \\ &= y + \underbrace{\sum_{S \in \mathcal{P}(W)}\left[(-\eta)^{|W-S|}\left(\prod_{w \in S} w\right)\left(\prod_{w \in W-S}\frac{\partial \mathcal{R}}{\partial w}\right) x\right]}_{(*)\text{ significant?}}\end{aligned}$$

Hence, the higher order terms in terms of η $(*)$ may become significant, despite the damping.

The key idea of batch-normalization is to normalize the layer activations (→ batch normalization) and then to backpropagate through the normalization. So it "keeps the same distribution" at each layer. And if we optimize the weights of a layer, it should not affect the distribution at the end of the layer.

So what we do is

- we fix a layer l,
- and we fix a set of examples $I \subset [1 : N]$
- and compute the mean activities and a vector of the standard deviations

$$\begin{aligned}\boldsymbol{\mu}^l &:= \frac{1}{|I|}\sum_{i \in I}(F^l \circ \cdots \circ F^1)(\mathbf{x}[i]) \in \mathbb{R}^{m_l} \\ \boldsymbol{\sigma}^l &\in \mathbb{R}^{m_l} \\ \sigma_j^l &:= \sqrt{\delta + \frac{1}{|I|}\sum_{i \in I}\left((F_j^l \circ \cdots \circ F^1)(\mathbf{x}[i]) - \mu_j^l\right)^2}, \quad \delta > 0,\end{aligned}$$

- then we remove the mean and divide by the standard deviation to normalize the activities.

$$\tilde{x}_j^l := \frac{x_j^l - \mu_j^l}{\sigma_j^l}$$

However, when we do this, what happens is that we can represent less than before. We may only have distributions with mean zero, and variance one (because we enforce this through the normalization). So we need to do something to regain the representational power. What we do is we multiply by some coefficients α_j and β_j

$$\tilde{\tilde{x}}_j^l = \alpha_j^l \tilde{x}_j^l + \beta_j^l$$

So since $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are functions of the weights and they can be differentiated.
A further note about batch-normalization is that it doesn't change the information in the data, because since we have $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ we could theoretically recuperate the original activations.

Now, some implementation details:

- The bias term before the batch normalization should be removed (since we're removing the mean it makes no sense).
- At training time, the statistics are computed per batch, hence they're very noisy. So what people do in practice (e.g., when they're predicting just one sample) is that they keep a running average over the batch batch-norm statistics. So, at test time, $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are replaced by the running averages that were collected during training time. An alternative, is to pass through the whole dataset at the end of the training and re-compute the statistics - that may work even better (but it takes a lot of time).

What is not very clear is why batch-normalization works. The original paper about batch-normalization (BN) said that BN reduces the internal covariance shift of the data. What they meant by this is that: let's say that we have a very simple classifier, that will basically classify everything that is negative to one class, and everything that is positive to another class. Then, when we just shift the data by a constant vector, then, without batch-normalization we'd shift all the datapoints into one class. However, with BN since the mean is removed we'll remove that constant shift the BN layer and it will work out. So BN reduces the covariance shift. That was the effect that the inventors of BN described.
However, it turns out that some other people came later on and said the following: They didn't negate the effect of the covariance-shift reduction, but the reason they said that BN works is that it makes the landscape of the loss more smooth. Hence, the optimization works better and gives better results.

However, no-one *really* knows why BN works so well.

14.10.2 Other Heuristics

- **Curriculum learning and non-uniform sampling** of data points → focus on the most relevant examples (Bengio, Louradour, Collobert, Weston, 2009) (DL-Book: 8.7.6) Or increase hardness of tasks (corner-cases9 as NN improves
- **Continuation methods**: define a family of (simpler) objective functions and track solutions, gradually change hardness of loss (DL-Book: 8.7.6)
- **Heuristics for initialization** (DL-Book: 8.4) scale the weights of each layer in a way that at at the end of the layer, the data has more or less the same energy (and gradient norms are more or less the same at each layer).
- **pre-training** (DL-Book: 8.7.4). for better initialization, to avoid local minima (less relevant today).

14.11 Norm-Based Regularization

$\mathcal{R}_\Omega(\theta; \mathcal{S}) = \mathcal{R}(\theta; \mathcal{S}) + \Omega(\theta)$,

where Ω is a functional (function from a vector-space to the field over which it's defined) that does not depend on the training data.

D. (L_2 Frobenius-Norm Penalty (Weight Decay))

$\Omega(\theta) = \frac{1}{2}\sum_{l=1}^{L}\lambda^l\|\mathbf{W}\|_F^2, \quad \lambda^l \geq 0$

Com. It's common practice to only penalize the weights, and not the biases.

So, the assumption here is that the weights have to be small. So we'll only allow a big increase in the weights, if it comes at a much bigger increase in performance. Regularization based on the L_2-norm is also called *weight-decay*, as

$\frac{\partial \boldsymbol{\Omega}}{\partial w_{ij}^l} = \lambda^l w_{ij}^l$,

which means that the weights in the l-th layer get pulled towards zero with "gain" λ^l. What happens in the gradient-update step is

$$\begin{aligned}\theta(t + 1) &= \theta(t) - \nabla_\theta \mathcal{R}_\Omega(\theta; \mathcal{S}) \\ &= \underbrace{(1 - \eta\lambda^l)\theta(t)}_{\text{weight decay}} - \underbrace{\eta}_{\substack{\text{step}\\\text{size}}}\ \underbrace{\nabla_\theta \mathcal{R}}_{\text{data dep.}}.\end{aligned}$$

and also note that we require $\eta\lambda^l < 1$.

Let's analyze the weight decay: The Quadratic (Taylor) approximation of $\mathcal{R}$ around the optimal θ^* would be

$$\begin{aligned}\mathcal{R}(\theta) &\approx \mathcal{R}(\theta^*) + \underbrace{\nabla_\theta \mathcal{R}(\theta^*)^\mathsf{T}}_{=\mathbf{o}}(\theta - \theta^*) + \frac{1}{2}(\theta - \theta^*)^\mathsf{T}\mathbf{H}(\theta - \theta^*) \\ &= \mathcal{R}(\theta^*) + \frac{1}{2}(\theta - \theta^*)^\mathsf{T}\mathbf{H}(\theta - \theta^*) \qquad (\star)\end{aligned}$$

where $\mathbf{H}_{\mathcal{R}}$ is the hessian of $\mathcal{R}$, so

$$(\mathbf{H}_{\mathcal{R}})_{i,j} = \frac{\partial^2 \mathcal{R}}{\partial \theta_i \partial \theta_j}$$

and $\mathbf{H}$ is the evaluation of $\mathbf{H}_{\mathcal{R}}$ at θ^*:

$$\mathbf{H} := \mathbf{H}_{\mathcal{R}}(\theta^*).$$

So now we have the upper quadratic approximation of the cost function $(\star)$ (so we're assuming it is a parabola and that we know θ^*). Now, let's compute the gradient of that upper approximation of $\mathcal{R}$ (in $(\star)$).

$$\nabla_\theta\left[\mathcal{R}(\theta^*) + \frac{1}{2}(\theta - \theta^*)^\mathsf{T}\mathbf{H}(\theta - \theta^*)\right] = -\mathbf{H}\theta^* + \mathbf{H}\theta \qquad (*)$$

Further, recall that

$$\nabla_\theta \Omega = \boldsymbol{\lambda} \odot \theta = \mathrm{diag}(\boldsymbol{\lambda})\theta$$

So, now, let's set $\nabla_\theta \mathcal{R}_\Omega$ (with $\nabla_\theta \mathcal{R}$ approximated as in $(*)$) equal to zero.

$$\begin{aligned}&\nabla_\theta \mathcal{R}_\Omega \overset{!}{\approx} 0 \\ \iff\quad &-\mathbf{H}\theta^* + \mathbf{H}\theta + \mathrm{diag}(\boldsymbol{\lambda})\theta \overset{!}{=} 0 \\ \iff\quad &(\mathbf{H} + \mathrm{diag}(\boldsymbol{\lambda}))\theta = \mathbf{H}\theta^*\end{aligned}$$

Since both $\mathbf{H}$ and $\mathrm{diag}(\boldsymbol{\lambda})$ are s.p.s.d. we can invert their sum

$$\iff \qquad \theta = (\mathbf{H} + \mathrm{diag}(\boldsymbol{\lambda}))^{-1}\mathbf{H}\theta^*$$

Now, what we can directly see here is that if we use no L2-regularization $\theta = \theta^*$. Now, since $\mathbf{H}$ is s.p.s.d. we can diagonalize it to $\mathbf{H} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\mathsf{T}$ where $\mathbf{\Lambda} = \mathrm{diag}(\epsilon_1, \ldots, \epsilon_d)$ and plug this in which gives us

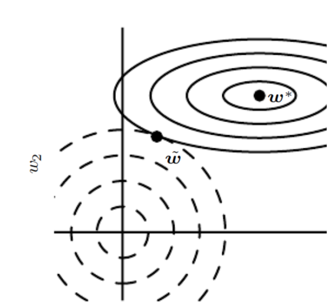
$$\begin{aligned}\theta &= (\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T + \mathrm{diag}(\boldsymbol{\lambda}))^{-1}\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T\theta^* \\ &= \mathbf{Q}\quad\underbrace{(\mathbf{\Lambda} + \mathrm{diag}(\boldsymbol{\lambda}))^{-1}\mathbf{\Lambda}}_{=\mathrm{diag}\left(\frac{\epsilon_1}{\epsilon_1 + \lambda_1}, \ldots, \frac{\epsilon_d}{\epsilon_d + \lambda_d}\right)}\quad\mathbf{Q}^\mathsf{T}\theta^*.\end{aligned}$$

So this gives us an idea what happens with θ^* in the directions of the eigenvectors of the hessian $\mathbf{H}$ if we use L2-regularization:

- if $\epsilon_i \gg \lambda_i$: **effect vanishes**: along directions in parameter space with *large* eigenvalues ϵ_i the weights are almost not reduced

- if $\epsilon_i \ll \lambda_i$: **shrinking effect**: along the directions in parameter space with small eigenvalues ϵ_i the weights are shrunk to nearly zero magnitude.

The following picture illustrates this better:



The isometric balls illustrate the regularization loss (L2) for any choice of θ (or w), and the ellipsoid curves illustrate the risk (for a parabolic risk). So $\tilde{w}$ is the point with the least loss for its specific regularization loss. As we can see, at that point

- downwards the risk has a large eigenvalue, as the risk increases rapidly. And as we've stated above, the value of w along that dimension is not reduced that much.
- from right to left (starting at w^*) the risk has a very low eigenvalue, and hence $\tilde{w}$ is reduced much more along that dimension.

D. (L1-Regularization (sparsity inducing))

$\Omega(\theta) = \sum_{l=1}^{L} \lambda^l \left\|\mathbf{W}^l\right\|_1 = \sum_{l=1}^{L} \lambda^l \sum_{i,j} |w_{ij}|, \qquad \lambda^l \geq 0$

– 14.11.1 — Regularization via Constrained Optimization –

An alternative view on regularization is for a given $r > 0$, solve

$$\min_{\theta, \|\theta\| \leq r} \mathcal{R}(\theta.)$$

So we're also constraining the size of the coefficients indirecty, be constraining θ to some ball.

The simple optimization approach to this is: *projected* gradient descent

$$\theta(t+1) = \Pi_r(\theta(t) - \eta\nabla\mathcal{R}), \qquad \Pi_r(\mathbf{v}) := \min\left\{1, \frac{r}{\|\mathbf{v}\|}\right\}\mathbf{v}$$

So we're essentially clipping the weights.

Actually, for each λ in L2-Regularization there is a radius r that would make the two problems equivalent (if the loss is convex).

Hinton made some research in 2013 and realized that

- the constraints do not affect the initial learning (as the weights are assumed to be small at the beginning), so we won't clip the weights. So the constraints only become active, once the weights are large.
- alternatively, we may just constrain the norm of the incoming weights for each unit (so use row-norms for the weight matrices). This had some practical success in stabilizing the optimization.

– 14.11.2 — Early Stopping

Gradient descent usually evolves solutions from: simple + robust → complex + sensitive. Hence, it makes sense to stop training early (as soon as validation loss flattens/increases). Also: computationally attractive.

Since the weights are initialized to small values (and grow and grow to fit/overfit) we're kindof clipping/constraining the weight sizes by stopping the learning process earlier.

Let's analyze the situation closer: If we study the gradient descent trajectories through a quadratic approximation of the loss around the optimal set of parameters θ^*. We've derived previously already (and show it here again with slightly different notation) that:

$$\nabla_\theta\mathcal{R}|_{\theta_0} \approx \nabla_\theta\mathcal{R}|_{\theta^*} + \mathbf{J}_{\nabla\mathcal{R}_\theta}\big|_{\theta^*}(\theta_0 - \theta^*) = \mathbf{H}(\theta_0 - \theta^*).$$

ꞁnis is just because the Jacobian of the gradient map is the Hessian $\mathbf{H}_\mathcal{R}$ from before.

So (as seen previously) we have that

$$\theta(t+1) = \theta(t) - \eta\nabla_\theta\mathcal{R}|_{\theta(t)} \approx \theta(t) - \eta\mathbf{H}(\theta(t) - \theta^*).$$

Now, subtracting θ^* on both sides gives us

$$\theta(t+1) - \theta^* \approx (\mathbf{I} - \eta\mathbf{H})(\theta(t) - \theta^*)$$

Now we'll use the same trick as before that we can diagonalize the hessian $\mathbf{H}$ as it's s.p.s.d., so $\mathbf{H} = \mathbf{Q\Lambda Q}^\top$. Inserting this gives us:

$$\theta(t+1) - \theta^* \approx (\mathbf{I} - \eta\mathbf{Q\Lambda Q}^\top)(\theta(t) - \theta^*)$$

Now let's have a look at everything w.r.t the eigenbasis of $\mathbf{H}$, let's define $\tilde{\theta} = \mathbf{Q}^\top\theta$. Then

$$\tilde{\theta}(t+1) - \tilde{\theta}^* \approx (\mathbf{I} - \eta\mathbf{\Lambda})(\tilde{\theta}(t) - \tilde{\theta}^*)$$

Now, assuming $\theta(0) = \mathbf{o}$ (and inserting and using it) and a small η ($\forall i$: $|1 - \eta\lambda_i| < 1$) one gets explicitly

$$\tilde{\theta}(t) = \tilde{\theta}^* - \underbrace{(\mathbf{I} - \eta\mathbf{\Lambda})^t\tilde{\theta}^*}_{\to 0 \text{ with upper ass. on eigenvalues}}.$$

Thus (comparing to the previous analysis) if we can choose t, η s.t.

$$(\mathbf{I} - \eta\Lambda)^t \overset{!}{=} \lambda(\mathbf{\Lambda} + \lambda\mathbf{I})^{-1}$$

which for $\eta\epsilon_i \ll 1$, and $\epsilon_i \ll \lambda$ can be achieved approximately via performing $t = \frac{1}{\eta\lambda}$ steps.

So early stopping (up to the first order) can thus be seen as an approximate L_2-regularizer.

– 14.12 — Dataset Augmentation

Applying some transformations to the input data such that we know that the output is not affected. E.g., for images: mirroring, slight rotations, scaling, slight shearing, brightness changes. Blows up data, but: there are approaches to incorporating this into the gradient instead of the input data.

– 14.12.1 — Invariant Architectures

Instead of augmenting the dataset one could build an architecture that is invariant to certain transformations of the data.

First, we distinguish the following terms: Let's say we have some $\mathbf{x}$ and apply the transformation $\mathbf{x}' := \tau(\mathbf{x})$. Then for our neural network F

- **D. (Invariance)** means that $F(\mathbf{x}) = F(\tau(\mathbf{x}))$.
- **D. (Equivariance)** means that $\tau(F(\mathbf{x})) = F(\tau(\mathbf{x}))$.
  So applying the transformation before or after applying F doesn't change a thing (e.g., convolutions and translations are equivariant).

E.g. NNs where the first layer is a convolution are invariant to image translation. Hence, it would make no sense to augment the dataset of images with translations. It also saves computation and memory not to do this. So if we have an architecture that is invariant to certain dataset augmentations the augmentations become obsolete. So, if you can, choose an invariant architecture to make your life easier in the first place.

– 14.12.2 — Injection of Noise

At various places: inputs (noise robustness), weights (regularization), targets (network becomes more careful)

– 14.12.3 — Semi-Supervised Training

If we have a lot of data, but only a few datapoints are labeled. Then semi-supervised training may become useful. You may build a generative model or an autoencoder to learn how to represent your data (learn features). Then, we train a supervised model on top of these representations.

– 14.12.4 — Multi-Task Learning

If we have different tasks that we may want to solve, we may share the intermediate representations across the tasks and then learn jointly (i.e., minimize the combined objective). A typical architecture would be to share the low-level representations, lern the high-level representations per task.

– 14.13 — Dropout

Dropout idea: randomly "drop" subsets of the units in the network.

So more preciely, we'll define a "keep" probability π_i^l for unit i in layer l.

- typically: $\pi_i^0 = 0.8$ (inputs), $\pi^{l\geq 1} = 0.5$ (hidden units)
- realization: sampling bit mask and zeroing out activations
- effectively defines an exponential ensemble of networks (each of which is a sub-network of the original one), just that we sample these models at training-time (instead of during prediction) and we *share* the parameters
- all modles share the same weights
- standard backpropagation applies.
- This prevents complex co-adaptions in which a feature detector is only helpful in the context of several other specific feature detectors. Instead, each neuron learns to detect a feature that is generally helpful for producing the correct answer given the combinatorially large variety of internal contexts in which it must operate. (Hinton et al., 2012). This enforces the features to be redundant (not too specific about one thing in the image) and also to build on top of *all* the features of the previous layer (since we never know if some are absent).

Benefits: benefits of ensembles with the runtime complexity of the training of one network. The network gets trained to have many different paths through it to get the right result (as neurons are turned off).

Equivalent to: adding multiplicative noise to weights or training exponentially many sub-networks $\sum_{i=1}^{n}\binom{n}{i} = 2^n$ wher n is the number of compute units (so at each iteration we turn some nodes off according to some probability). So we're getting the benefits of ensembles with the runtime complexity of just training one network.

Ensembling corresponds to taking geometric mean (instead of usual arithmetic) (must have to do with exponential growth of networks) of the ensembles:

$P_{\text{ensemble}}(y \mid \mathbf{x}) = \sqrt[d]{\prod_\mu P(\boldsymbol{\mu})\,P(y \mid \mathbf{x}, \boldsymbol{\mu})}$

Having to sample several sub-networks for a prediction is somewhat inconvenient, so the idea that Hinton et al. came up with is: scaling each weight w_{ij}^l by the probability of the unit j being active

$$\tilde{w}_{ij}^l \leftarrow \pi_j^{l-1}w_{ij}$$

This makes sure that the net (total) input to unit x_i^l is calibrated, i.e.,

$$\sum_j \tilde{w}_{ij}^l x_j^{l-1} \overset{!}{=} \mathbb{E}_{Z\sim P(Z)}\left[\sum_j Z_j^{l-1}w_{ij}x_j^{l-1}\right]\sum_j \pi_j^{l-1}w_{ij}x_j^{l-1} \qquad ✓$$

It can be shown that this approach leads to a (sometimes exact) approximation of a gemoetrically averaged ensemble (see DL-Book, 7.12).

Ex. Let's say that at the end we selected each unit with a probability of 0.5. Then when typically when we're finished with training our neural network, we're going to multiply all the weights that we obtained with 0.5 to reduce the contribution of each of the features (since we'll have all of them). So with this trick for the prediction we can just do a single forward pass.

15 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

– 15.1 — 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).

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

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

$m := |\mathcal{V}|$, 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, ...)

– 15.1.1 — 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)

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

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

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

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

de-embedding: $\mathbf{V\Lambda}_k^{-\frac{1}{2}}$ (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:

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

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

$$\theta^* = \arg\max_\theta \prod_{(i,j)\in C_R} P_\theta(w_i \mid w_j)$$

Now our approach to approximate the probability $P_\theta(w_i \mid w_j)$ 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-occurrent words in our dataset $\mathbf{w} = (w_1, \ldots, w_T)$:

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

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

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)

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

where

- $\mathbf{x}_w$ 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.

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

For this reason we'll introduce the following function

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

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

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

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

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

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).

– 15.2 — 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:

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

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

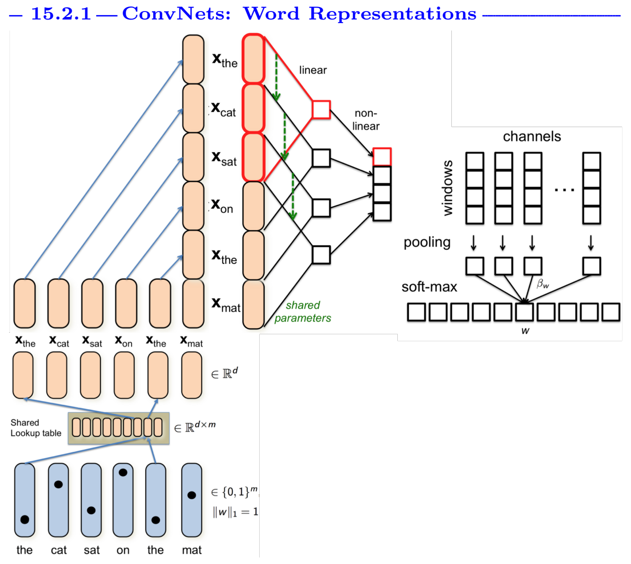
This problem has been adressed in different ways. Here are the traditional approach, and the more modern approaches.

1. **Traditional Approach:** k-th order Markov assumption
   $$P(w_t \mid w_{t-1}, \ldots, w_1) \overset{\text{Markov ass.}}{\approx} P(w_t \mid w_{t-1}, \ldots, w_{t-k}) \overset{\text{emp. dist.}}{\approx} (k+1) - \text{gram counts}$$
   In practice we often use 5-grams, i.e., $k = 4$ (last 4 words).
2. **Modern Approach:** Create language models via embeddings
   $$\log\left(P(w_t \mid \mathbf{w} := w_{t-1}, \ldots, w_1)\right) = \mathbf{x}_{w_t}^\top\mathbf{z}_\mathbf{w} + \text{const}$$
   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)

– 15.2.1 — 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:
   $$f: \mathbb{R}^{n\cdot d} \to \mathbb{R}^{(n-w+1)\times k}$$
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:
   $$P(w_{t+1} \mid \mathbf{w}) = \frac{\exp\left(\mathbf{y}_{w_t}^\top\mathbf{z}_\mathbf{w}\right)}{\sum_{u\in\mathcal{V}}\exp\left(\mathbf{y}_u^\top\mathbf{z}_\mathbf{w}\right)}.$$

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

– 15.2.2 — 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.

– 15.3 — 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.

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

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

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

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

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

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 2: Backpropagation in RNNs

```
(Blue terms only need to be comp. for multiple-output RNNs)
// Compute derivative w.r.t. outputs
Compute $\frac{\partial\mathcal{R}}{\partial\mathbf{y}^1}, \frac{\partial\mathcal{R}}{\partial\mathbf{y}^2}, \ldots, \frac{\partial\mathcal{R}}{\partial\mathbf{y}^T}$   $\left(= \frac{\partial L}{\partial\mathbf{y}^i}\right)$
// 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
$\frac{\partial\mathcal{R}}{\partial w_{ij}} \leftarrow \sum_{i=1}^{T}\frac{\partial\mathcal{R}}{\partial h_j^{t-1}}\frac{\partial h_j^{t-1}}{\partial w_{ij}} = \sum_{i=1}^{T}\frac{\partial\mathcal{R}}{\partial h_j^{t-1}}\cdot\dot{\sigma}_i^t\cdot h_j^{t-1},$
$\frac{\partial\mathcal{R}}{\partial u_{ij}} \leftarrow \sum_{i=1}^{T}\frac{\partial\mathcal{R}}{\partial h_j^{t-1}}\frac{\partial h_j^{t-1}}{\partial u_{ij}} = \sum_{i=1}^{T}\frac{\partial\mathcal{R}}{\partial h_j^{t-1}}\cdot\dot{\sigma}_i^t\cdot x_j^t,$
$\frac{\partial\mathcal{R}}{\partial b_i} \leftarrow \sum_{t=1}^{T}\frac{\partial\mathcal{R}}{\partial\mathbf{h}_i^t}\frac{\partial\mathbf{h}_i^t}{\partial b_i} = \sum_{t=1}^{T}\frac{\partial\mathcal{R}}{\partial\mathbf{h}_i^t}\cdot\dot{\sigma}_i^t,$
        where $\dot{\sigma}_i^t := \sigma'(\overline{F}_i(\mathbf{h}^{t-1}, \mathbf{x}^t))$.
$\frac{\partial\mathcal{R}}{\partial v_{ij}} \leftarrow \sum_{i=1}^{T}\frac{\partial\mathcal{R}}{\partial y_j^{t-1}}\frac{\partial y_j^{t-1}}{\partial v_{ij}} = \sum_{i=1}^{T}\frac{\partial\mathcal{R}}{\partial y_j^{t-1}}\cdot\dot{\sigma}_i^t\cdot y_j^t,$
$\frac{\partial\mathcal{R}}{\partial c_{ij}} \leftarrow \sum_{i=1}^{T}\frac{\partial\mathcal{R}}{\partial y_j^{t-1}}\frac{\partial y_j^{t-1}}{\partial v_{ij}} = \sum_{i=1}^{T}\frac{\partial\mathcal{R}}{\partial y_j^{t-1}}\cdot\dot{\sigma}_i^t,$
        where $\dot{\sigma}_i^t := \sigma'(\overline{H}_i(\mathbf{h}^{t-1}, \mathbf{x}^t))$.
```

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

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))

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

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

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.

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

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

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

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

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 $\sigma \in$ {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

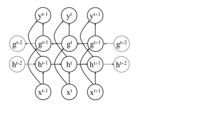
- When the gradients are exploding a common heuristic is to clip the gradients (shring the gradient norm once it gets too big), so-called gradient clipping.
  $$\nabla_\mathbf{x}\mathcal{R} \leftarrow \nabla_\mathbf{x}\mathcal{R}\cdot\frac{\gamma_{\max}}{\max\left(\|\nabla_\mathbf{x}\|\,\mathcal{R}, \gamma_{\max}\right).}$$
  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.

– 15.3.1 — Backprop Over Time

For multi-output loss

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

– 15.3.2 — Bi-Directional RNNs



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

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

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:

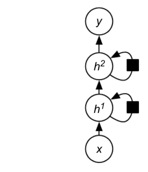
$$\mathbf{y}^t = H(\mathbf{h}^t, \mathbf{g}^t; \theta).$$

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.

– 15.3.3 — 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.

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



– 15.3.4 — 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 $P\left(\mathbf{y}^t\right)$ only depends on $\mathbf{y}^{1:(t-1)}$ through $\mathbf{h}^t$ since

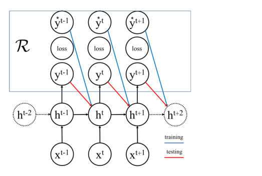
$$\mathbf{x}^{1:t}\overset{F}{\mapsto}\mathbf{h}^T\overset{H}{\mapsto}\boldsymbol{\mu}^t \mapsto P\left(\mathbf{y}^T\right).$$

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.

16 Memory & Attention

– 16.1 — 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 $\mathbf{x}_t$.

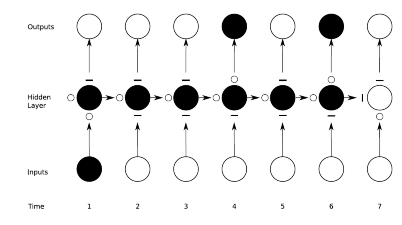
Goal of Memory Units: model long-term dependencies. have some kind of memory.

– 16.1.1 — LSTMs

It would be nice to have something like a gated unit

D. (Gated Unit)

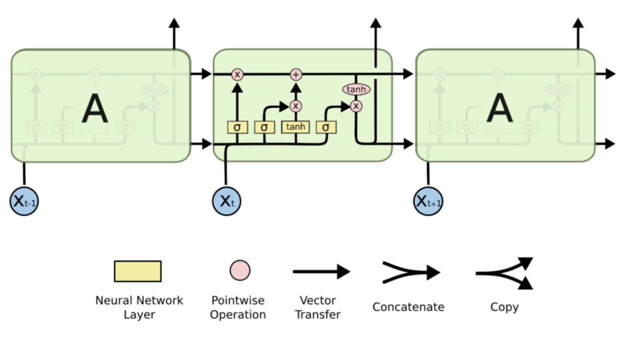
The following picture illustrates how we have a memory unit where we can store, read and delete information illustrated by the *gated units*.



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).

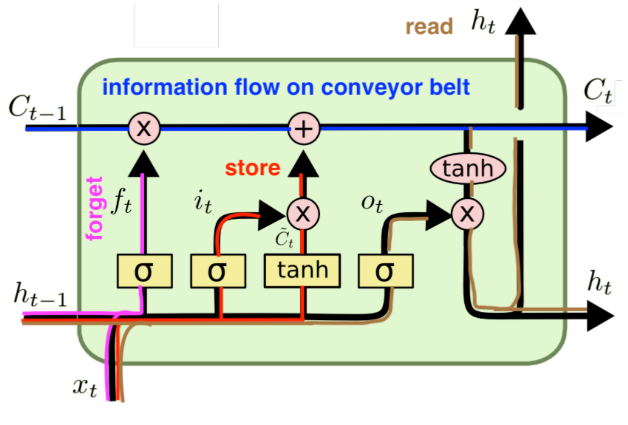
D. (LSTM) Long Short-Term Memory: Remembering information for long time and forgetting it fast.

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):
  $$\mathbf{o}^t = \sigma(\mathbf{W}_o\mathbf{h}^{t-1} + \mathbf{U}_o\mathbf{x}^t + \mathbf{b}_o) \qquad \text{(output)}$$
  $$\mathbf{h}^t = \underbrace{\mathbf{o}^t}_{\substack{\text{what we want}\\ \text{to let out}}}\odot\tanh(\underbrace{C^t}_{\text{new mem.}}).$$

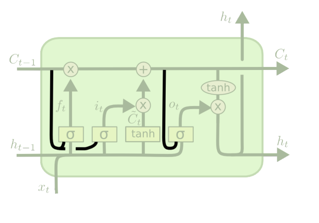
Note that when stacking $\mathbf{h}^{t-1}$ and $\mathbf{x}^t$ we may just represent things as follows:

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

– 16.1.2 — Other Variants of LSTMs

– LSTMs with Peepholes

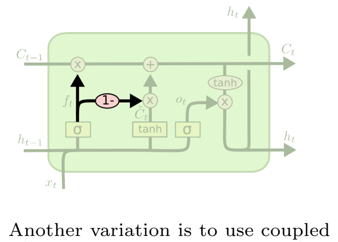
There are a bunch of other LSTMs that have been invented. A change that was done with LSTMS was: *LSTS with peepholes* where each of the gates also is allowed to look at the memory (Gers & Schmidhuber, 2000).



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

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

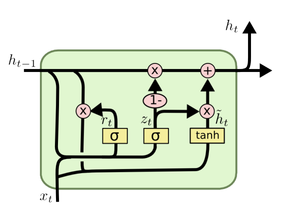
– Coupled Forget and Input Gates



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

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.

– 16.1.3 — Unsegmented Sequences

Problem: different durations for same thing

"Theee houssse iis new."

"The hoouse iis new."

– 16.1.4 — Connectionist Temporal Classification

Let's have a look an approach with LSTMs on hwo to solve the problem of unsegmented sequences.

The connectionist temporal classification allows to estimate the sequences of unsegmented data:

Acutally, it's a very simplified model (huge simplification)

$$P(\pi \mid \mathbf{x}) = \prod_{t=1}^{T} y_{\pi_t}$$

where

- $\mathbf{x}$ is the sound that we had, segmented every few milliseconds,
- π_t is a distribution over all possible lables + blank. Note that a blank doesn't mean no sound, it means I don't know.

So it assumes that each of the labels that we're getting are independent - so this is really the point where we can say that that doesn't make any sens! Even though it's a huge simplification it works very well.

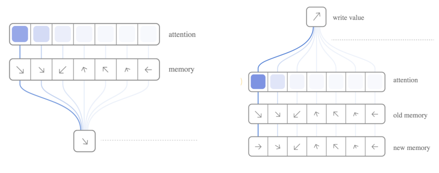
The next step is to take the sequence of the predictions (if whe have two "a" "a", then most likely there should be just one "a" and the next should be something that correlates to a).

So the next idea is after having the sequence $\pi_1, \ldots, \pi_T$ of predictions we'll add all the potential probabilities that actually mean the same thing.

– 16.2 — Differentiable Memory

D. (Neural Turing Machine)

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

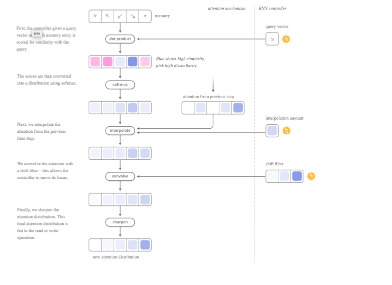


In contrast to RNNs, NTMs use an external memory to which they read and write according to some determined probabilities.Now here's an example how that can become very useful. We may have these vectors in memory, and then we may search for some vector. The greater the dot-product is with a vector, the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

16.3 — Attention Mechanisms

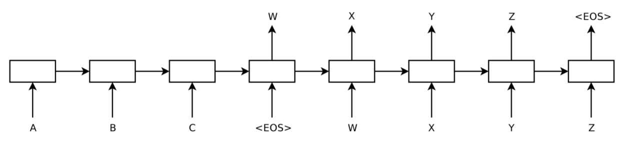
D. (Attention Mechanisms) offer a simple way to overcome some challenges of RNN-based memorization. With attention mechanisms we selectively attend to *inputs* or *feature representations* computed from inputs.

- 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 acheve 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

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

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

$$F(\mathbf{x}^1,\dots,\mathbf{x}^{T_x}) = \text{“thought vector”}$$

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.

16.3.1 — 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.

16.4 — 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.

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

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

17 Unsupervised Learning

Here we'll look at what we can say about a distribution of $\boldsymbol{X}$, when we have some samples $\mathbf{x}_1,\dots,\mathbf{x}_N$. Unsupervised learning is the most dangerous thing that we can do (dangerous if we don't know what we're doing). Unsupervised learning usually is hard, because we don't have a goal. The final goal of unsupervised learning is *density estimation* - so, understand the distribution that the data is coming from. Other things we might strive for is interpretability of the results we've learned about $p(x)$. Another key aspect of unsupervised learning is: "I don't know what I'm looking for until I find it."

17.1 — Density Estimation

D. (Density Estimation) is a standard problem in statistics and unsupervised learning. It's 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 stimated 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:

$$\overline{p}_\theta(\mathbf{x}) = \frac1n\sum_{i=1}^n k_h(\mathbf{x}-\mathbf{x}_i) = \frac{1}{nh}\sum_{i=1}^n k\left(\frac{\mathbf{x}-\mathbf{x}_i}{h}\right).$$

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)

$$J(\theta) = \mathbb{E}\left[\sum_i \partial_i\psi_{\theta,\imath} - \frac12\psi^2_{\theta,i}\right].$$

This expectation can be approximated by sampling.

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

17.2 — Autoencoders

Given: data points $\{\mathbf{x}_1,\dots,\mathbf{x}_n\}\subset\mathbb{R}^d$

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

D. (Autoencoder) any NN that aims to learn the *identity map*.

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

$$\ell(\mathbf{x},\hat{\mathbf{x}}) = \frac12\|\mathbf{x}-\hat{\mathbf{x}}\|_2^2$$

Typically, the network can be broken into two parts G and H such that

- $F = H\circ G \approx \mathbf{x}\mapsto\mathbf{x}$
- *Encoder*: $G = F_l\circ\cdots\circ F_1:\mathbb{R}^n\to\mathbb{R}^m$, $\mathbf{x}\mapsto\mathbf{z}=\mathbf{x}^l$
- *Decoder*: $H = F_L\circ\cdots F_{l+1}:\mathbb{R}^m\to\mathbb{R}^n$, $\mathbf{z}\mapsto\mathbf{y}=\hat{\mathbf{x}}$.
- layer l is usually a "bottleneck" layer.

Com. Just a special case of a feedforward NN, that can be trained through backpropagation.

Autoencoders provide a canonical way of *representation learning* (since NNs naturally do this). Note, how the data compression (learning compressed representation) is just a "proxy" and not the real learning objective of the network (identity function).

17.2.1 — Linear Autoencoding

D. (Linear Autoencoder)

A linear autoencoder just consists of two linear maps: an encoder $\mathbf{C}\in\mathbb{R}^{m\times d}$ and a decoder $\mathbf{D}^{d\times m}$. The objective it minimizes is then:

$$\mathcal{R}(\theta) = \frac{1}{2n}\sum_{i=1}^n\|\mathbf{x}_i-\mathbf{DC}\mathbf{x}_i\|_2^2.$$

So it's a NN with one hidden layer (no biases and linear activation functions) which will contain the compressed representation $\mathbf{z}=\mathbf{Cx}\in\mathbb{R}^m$.

D. (Linear Autoencoder with Coupled Weights)

Then, we define $\mathbf{D} := \mathbf{C}^\top$.

D. (Singular Value Decomposition)

Recall that the SVD of a data matrix

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

is of the following form:

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

And the matrices $\mathbf{U}$ and $\mathbf{V}$ are orthogonal - so we have an orthogonal basis. Further recall that via the SVD we can get the best rank k approximation of a linear mapping. It also is a decomposition that preserves as much of the variance (or energy) of the data for a predefined number of desired basis vectors to represent it.

– Optimal Linear Compression

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

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

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 $\operatorname{rank}(\mathbf{CD})\le m$. However, the auto-encoder can achieve the result of the SVD.

T. Given the SVD of the data $\mathbf{X} = \mathbf{U}\operatorname{diag}(\sigma_1,\dots,\sigma_n)\mathbf{V}^\top$. The choice $\mathbf{C} = \mathbf{U}_m^\top$ and $\mathbf{D} = \mathbf{U}_m$ minimizes the squared reconstruction error of a two-layer linear auto-encoder with m hidden units.

Proof.

$$\mathbf{DCX} = \mathbf{U}_m\mathbf{U}_m^\top\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T = \mathbf{U}_m\begin{bmatrix}\mathbf{I}_m & 0\end{bmatrix}\boldsymbol{\Sigma}\mathbf{V}^T = \mathbf{U}_m\begin{bmatrix}\boldsymbol{\Sigma}_m & 0\end{bmatrix}\mathbf{V}^T = \mathbf{U}_m\boldsymbol{\Sigma}_m\mathbf{V}_m^T.$$

And as we know from the Eckhart-Young theorem $\hat{\mathbf{X}} = \mathbf{U}_m\boldsymbol{\Sigma}_m\mathbf{V}_m^T$ is the best m-dimensional approximation of the original data $\mathbf{X}$.

Now, since $\mathbf{C} = \mathbf{U}_m^\top$ and $\mathbf{D} = \mathbf{U}_m$ that means that we can do weight sharing between the decoder and encoder network, since $\mathbf{C} = \mathbf{D}^\top$.
Another thing to note is that the solution is *not unique!* For any invertible matrix $\mathbf{A}\in GL(m)$ we have

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

Now, restricting through weight sharing that $\mathbf{D} = \mathbf{C}^\top$ will enforce that

$$\mathbf{A}^{-1} = \mathbf{A}^\top$$

hencem $\mathbf{A}\in O(m)$ (orthogonal group, rotation matrices). Then the mapping $\mathbf{x}\to\mathbf{z}$ is determined (up some rotation that we do in-between, rotation and its inverse).

– Principal Component Analysis

A way to solve this problem is through PCA. First, we center the data (pre-processing) as follows:

$$\mathbf{x}_i\mapsto\mathbf{x}_i\sum_{i=1}^k\mathbf{x}_i$$

Then we define

$$\mathbf{S} := \mathbf{X}\mathbf{X}^\top$$

which is the sample covariance matrix. And then, in order to get $\mathbf{U}$ we just do the singular value decomposition of $\mathbf{S}$. If we relate it to the SVD of $\mathbf{X}$ we can see that

$$\mathbf{S} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\top\mathbf{V}\boldsymbol{\Sigma}\mathbf{U}^\top = \mathbf{U}\boldsymbol{\Sigma}^2\mathbf{U}^\top.$$

So, the column vectors of $\mathbf{U}$ are the eigenvectors of the covariance matrix. And $\mathbf{U}_m\mathbf{U}_m^\top$ is the orthogonal projection onto m principal components of $\mathbf{S}$.
Note that if we wanted to get $\mathbf{V}$ the we'd just do the PCA with $\mathbf{S} = \mathbf{X}^\top\mathbf{X}$.

17.2.2 — Non-Linear Autoencoders

Non-linear autoencoders allow us to learn powerful non-linear generalizations of the PCA.

D. (Non-Linear Autoencoder) contains many hidden layers with nonlinear-activation functios as we want (as long as there's a bottleneck layer) and train the parameters via MLE.

17.2.3 — Regularized Autoencoders

One may also regularize the code $\mathbf{z}$ via a regularizers $\Omega(\mathbf{z})$. This will give us a regularized autoencoder.
There are various flavours of regularization:

- standard L_2 penalty: ability to learn "overcomplete" codes
- **D. (Code Sparseness)** e.g., via $\Omega(\mathbf{z}) = \lambda\|\mathbf{z}\|_1$
- **D. (Contractive Autoencoders)** $\Omega(\mathbf{z}) = \lambda\left\|\frac{\partial\mathbf{z}}{\partial\mathbf{x}}\right\|_F^2$. This penalizes the Jacobian and generalizes weight decay (cf. Rifai et al, 2011)

17.2.4 — Denoising Autoencoders

Autoencoders allso allow us to separate the signal from noise: Denoising autoencoders aim to learn features of the original data representation that are robust under noise.

D. (Denoising Autoencoder) we perturb the inputs

$$\mathbf{x}\mapsto\mathbf{x}_{\boldsymbol{\eta}},$$

where η *is a random noise vector*, e.g., additive (white) noise

$$\mathbf{x}_\eta = \mathbf{x}+\boldsymbol{\eta},\qquad \boldsymbol{\eta}\sim\mathcal{N}(\mathbf{o},\sigma^2\mathbf{I})$$

and instead of the original objective, we minimize the following

$$\mathbb{E}_\mathbf{x}\left[\mathbb{E}_{\boldsymbol{\eta}}\left[\ell(\mathbf{x},(H\circ G)(\mathbf{x}_{\boldsymbol{\eta}}))\right]\right]$$

The hope is that we'll achieve *de-noising*, which happens if

$$\|\mathbf{x}-H(G(\mathbf{x}_{\boldsymbol{\eta}}))\|^2\,.<\|\mathbf{x}-\mathbf{x}_{\boldsymbol{\eta}}\|^2$$

So this would mean that the reconstruction error of the noisy data is less than the error we created by the noise we've added (then the de-noising works).

17.3 — Factor Analysis

17.3.1 — 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: $p(\mathbf{x}\,|\,\mathbf{z})$
3. Construct the *observed data model* by integrating/summing out the latent variables

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

Ex. (Gaussian Mixture Models GMMs)

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

$p(\mathbf{x}\,|\,\mathbf{z})$: 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 $p(\mathbf{x}\,|\,\mathbf{z})$ 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 $\mathbf{x}\in\mathbb{R}^n$

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

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

$$\hat{\boldsymbol{\mu}} = \frac1k\sum_{i=1}^k\mathbf{x}_i$$

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

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

Proof. This can be proven in three steps

1. We use the insights on MGFs of and their properties.
2. We use the insights on MGFs of multivariate normal distributions
3. Then the proof is straightforward.

If you need some refresher of some core definitions just have a look at them (after the proof).

Let $\widetilde{\mathbf{x}} := \mathbf{Wz}$ s.t. $\mathbf{x} = \boldsymbol{\mu}+\widetilde{\mathbf{x}}+\boldsymbol{\eta}$. Now let's determine the MGF of $\widetilde{\mathbf{x}}$:

$$M_{\widetilde{\mathbf{x}}}(\mathbf{t}) = \mathbb{E}_{\widetilde{\mathbf{x}}}\left[e^{\mathbf{t}^\top\widetilde{\mathbf{x}}}\right] = \mathbb{E}_\mathbf{z}\left[e^{\mathbf{t}^\top\mathbf{Wz}}\right] = \mathbb{E}_\mathbf{z}\left[e^{\left(\mathbf{W}^\top\mathbf{t}\right)^\top\mathbf{z}}\right] = M_\mathbf{z}(\mathbf{W}^\top\mathbf{t}).$$

Now, since $\mathbf{z}\sim\mathcal{N}(\mathbf{o},\mathbf{I})$ and we know the form of a MGF of a normal distribution, we can just plug in:

$$M_\mathbf{z}(\mathbf{W}^\top\mathbf{t}) = \exp\left(\frac12(\mathbf{W}^\top\mathbf{t})^\top\mathbf{I}(\mathbf{W}^\top\mathbf{t})\right) = \exp\left(\frac12\mathbf{t}^\top(\mathbf{W}\mathbf{W}^\top)\mathbf{t}\right).$$

So this gives us

$$M_{\widetilde{\mathbf{x}}}(\mathbf{t}) = \exp\left(\frac12\mathbf{t}^\top(\mathbf{W}\mathbf{W}^\top)\mathbf{t}\right).$$

which btw shows us that $\widetilde{\mathbf{x}} = \mathbf{Wz}\sim\mathcal{N}(\mathbf{o},\mathbf{W}\mathbf{W}^\top)$.
Now, we defined that $\mathbf{x} = \widetilde{\mathbf{x}}+\boldsymbol{\eta}+\boldsymbol{\mu}$. Now, in order to determine the distribution $P(\mathbf{x})$, we can just use the fact that the MGF of the addition of two or more random variables is just the multiplication of their MGFs. So,

$$\begin{aligned} M_\mathbf{x} &= M_{\widetilde{\mathbf{x}}+\boldsymbol{\eta}+\boldsymbol{\mu}}\\ &= M_{\widetilde{\mathbf{x}}}\cdot M_{\boldsymbol{\eta}}\cdot M_{\boldsymbol{\mu}}\\ &= \exp\left(\frac12\mathbf{t}^\top\mathbf{W}\mathbf{W}^\top\mathbf{t}\right)\exp\left(\frac12\mathbf{t}^\top\boldsymbol{\Sigma}\mathbf{t}\right)\exp\left(\mathbf{t}^\top\boldsymbol{\mu}\right)\\ &= \exp\left(\mathbf{t}^\top\boldsymbol{\mu}+\frac12\mathbf{t}^\top(\mathbf{W}\mathbf{W}^\top+\boldsymbol{\Sigma})^\top\mathbf{t}\right).\end{aligned}$$

From the form of the MGF $M_\mathbf{x}$ we can conclude that

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

□

– 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

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

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

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

One can prove that the posterior also follows a normal distribution (see http://cs229.stanford.edu). So,

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

where

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

Further, if we assume that $\boldsymbol{\Sigma} = \sigma^2\mathbf{I}$ and we let $\sigma^2\to0$ (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:

$$\mathbf{W}^\top\left(\mathbf{W}\mathbf{W}^\top+\sigma^2\mathbf{I}\right)^{-1}\overset{\sigma^2\to0}{\to}=:\mathbf{W}^\dagger\in\mathbb{R}^{m\times n}.$$

Consequently with the assumption of zero reconstruction error:

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

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

$$\mathbf{S} := \frac1k\sum_{i=1}^k\mathbf{x}_i\mathbf{x}_i^\top = \frac1k\mathbf{X}\mathbf{X}^\top.$$

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:

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

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

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

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:

$$\begin{aligned}\nabla_\mathbf{W}\mathbf{A} &= 2\mathbf{W}\\ \nabla_{\boldsymbol{\Sigma}}\mathbf{A} &= \mathbf{I}\end{aligned}$$

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 $\sigma^2\to0$ (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

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

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

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

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 $\forall\mathbf{t}$: $P(\boldsymbol{X}=\mathbf{t}) = P(\boldsymbol{Y}=\mathbf{t})$ (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 $M_{\boldsymbol{X}+\boldsymbol{Y}} = M_{\boldsymbol{X}}\cdot M_{\boldsymbol{Y}}$.

D. (Multivariable Normal Distribution)

$\boldsymbol{X}\sim\mathcal{N}(\boldsymbol{\mu},\boldsymbol{\Sigma})$, $\boldsymbol{X}\in\mathbb{R}^n$

$\mu = \mathbb{E}[\boldsymbol{X}]$ (mean)

$\boldsymbol{\Sigma} = \mathbb{E}\left[(\mathbf{x}-\boldsymbol{\mu})(\mathbf{x}-\boldsymbol{\mu})^\top\right]$ (variance-covariance matrix)

PDF:

$$p(\mathbf{x};\boldsymbol{\mu},\boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^n\cdot\det(\boldsymbol{\Sigma})}}e^{-\frac12(\mathbf{x}-\boldsymbol{\mu})^\top\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})}$$

MGF:

$$M_{\boldsymbol{X}}(\mathbf{t}) = \exp\left(\mathbf{t}^\top\boldsymbol{\mu}+\frac12\mathbf{t}^\top\boldsymbol{\Sigma}\mathbf{t}\right)$$

17.4 — Latent Variable Models

17.4.1 — DeFinetti's Theorem

There's another way of at 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*

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

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.

The follwing paragraph of a paper shows why interpretability is important:

F. Doshi-Veletz et al. (NIPS 2015)

> "Objectives such as data exploration present unique challenges and opportunities for problems in unsupervised learning. While in more typical scenarios, the discovered latent structures are simply required for some downstream task – such as features for a supervised prediction problem – in data exploration, the model must provide information to a domain expert in a form that they can readily interpret. It is not sufficient to simply list what observations are part of which cluster; one must also be able to explain why the data partition in that particular way. These explanations must necessarily be succinct, as people are limited in the number of cognitive entities that they can process at one time."

17.4.2 — Latent Variable Models

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

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

17.4.3 — 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 $f(\cdot)$ and $\mathbf{Z}$ 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 $\mathbf{C}$)

17.4.4 — 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: $F_\theta:\mathbb{R}^d\to\mathbb{R}^m$
- induced distribution $\mathbf{x}\in\mathbb{R}^m$, $\mathbf{x}\sim p_\theta(\mathbf{x})$
- sampling is easy: random vector + forward propagation.

18 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)

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

$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial W}\times_{d_3,d_4}\frac{\partial y}{\partial W}$ then we have: $T_{i,j,k,l} = \frac{\partial y_{i,j}}{\partial W_{k,l}}$

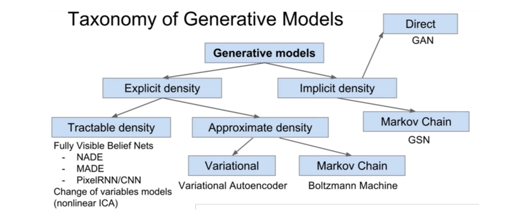
19 Generative Models

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.

19.1 — Noise Contrastive Estimation (NCE)

Approach: Reduce unsupervised problem of estimating $p(\mathbf{x})$ to binary classification problem. The MLE of the classification problem is asymptotically consistent with estimator of original problem. Asymptotically consistent: with increasing number of datapoints, the resulting estimates become more and more concentrated near the true value of the estimated parameter.

Noise contrastive estimation is an explicit density estimation method. It works by turing a unnormalized probability distribution into a normalized probability distribution. Instead of computing the partition function NCE solves the normalization probelm by extending everything to a joint distribution which has a switch variable that selects either the real distribution or a noise distribution. The same thing is done for the training distribution. Then everything is trained using MLE.

Many probabilistic models are defined by an unnormalized probability distribution $\tilde{p}(\mathbf{x};\theta)$. We must normalize $\tilde{p}$ by dividing by a partition function $Z(\theta)$ to obtain a valid probability distribution

$p(\mathbf{x};\theta) = \frac{1}{Z(\theta)}\tilde{p}(\mathbf{x};\theta)$.

The partition function is an integral or sum over the unnormalized probability of all states: $Z(\theta) = \int_\mathcal{X}\tilde{p}(\mathbf{x})\,d\mathbf{x}$. This operation is intractable in many cases.

So what makes learning undirected models by maximum likelihood particularly difficult is that the partition function depends on the parameters. Thus, the gradient of the log-likelihood w.r.t. the parameters has a term corresp. to the gradient of the partition function:

$\nabla_\theta\log(p(\mathbf{x};\theta)) = \nabla_\theta\log(\tilde{p}(\mathbf{x};\theta))-\nabla_\theta\log(Z(\theta))$.

This is a well-known decomposition into the *positive phase* and the *negative phase* of learning.

Most techniques for estimating modesl with an intractable partition function do not provide an estimate of the partition function.

Noise Constrastive Estimation (NCE) takes a different strategy: In this approach, the probability distribution estimated by the model is represented explicitly as

$\log(p_\text{model}(\mathbf{x})) = \log(\tilde{p}(\mathbf{x};\theta))+\underbrace{c}_{=-\log(Z(\theta))}$.

So we have the relationship

$\frac{1}{Z(\theta)} = e^c$.

Now, rather than estimating only θ, NCE treats c as just another parameter and estimates θ and c simultaneously, using the same algorithm for both. The resulting $\log(p_\text{model}(\mathbf{x}))$ may thus not correspond exactly to a valid probability distribution, but it will become closer and closer to being valid as the estimate of c improves.

We cannot optimize this with MLE, because MLE would just set c arbitrarily high. (we want: c to be s.t. we get a valid probability distr.)

NCE works by reducing the unsupervised learning problem of estimating $p(\mathbf{x})$ to that of learning a probabilistic binary classifier in which one of the categories corresponds to the data generated by the model. This supervised learning problem is constructed in such a way that the MLE defines an asymptotically consistent estimator of the original problem.

Specifically, we introduce a second distribution, the *noise contrastive distribution* $p_\text{noise}(\mathbf{x})$. The noise distribution should be tractable to evaluate and to sample from. We can now construct a model over both $\mathbf{x}$ and a new *binary* class variable $y\sim Be(p)$. In the new joint probability model we specify that

$$\begin{aligned}p_\text{joint}(y=1) = p_\text{joint}(y=0) &= \frac12\\ p_\text{joint}(\mathbf{x}|\mathbf{y}=1) = p_\text{model}(\mathbf{x}) &= e^c\tilde{p}(\mathbf{x};\theta)\\ p_\text{joint}(\mathbf{x}|\mathbf{y}=0) &= p_\text{noise}(\mathbf{x})\end{aligned}$$

In other words, y is a switch variable that determines whether we will generate $\mathbf{x}$ from the model or from the noise distribution.

We can construct a similar joint model p_train of the training data. In this case, the switch variable y determines whether we draw $\mathbf{x}$ from the data or from the noise distribution. Formally,

$$\begin{aligned}p_\text{train}(y=1) = p_\text{train}(y=0) &= \frac12\\ p_\text{train}(\mathbf{x}|y=1) &= p_\text{data}(\mathbf{x})\\ p_\text{train}(\mathbf{x}|y=0) &= p_\text{noise}(\mathbf{x})\end{aligned}$$

We can now just use standard maximum likelihood learning on the *supervised* learning problem of fitting p_joint to p_train.

$$(\theta^*,c^*) = \arg\max_{\theta,c}\mathbb{E}_{\mathbf{x},y\sim p_\text{train}}\left[\log\left(p_\text{joint}(y|\mathbf{x})\right)\right].$$

The distribution p_joint is essentially a logistic regression model applied to the difference in log probabilities of the model and the noise distribution:

$$\begin{aligned}p_\text{joint}(y=1|\mathbf{x}) &= \frac{p_\text{model}(\mathbf{x})}{p_\text{model}(\mathbf{x})+p_\text{noise}(\mathbf{x})}\\ &= \frac{e^c\tilde{p}_\text{model}(\mathbf{x})}{e^c\tilde{p}_\text{model}(\mathbf{x})+p_\text{noise}(\mathbf{x})}\\ &= \sigma(\log(p_\text{model}(\mathbf{x}))-\log(p_\text{noise}(\mathbf{x}))).\end{aligned}$$

Now, the estimate $p_\text{joint}(\mathbf{x})$ is bayes optimal if

$$e^c\tilde{p}_\text{model}(\mathbf{x};\theta) = p_\text{data}(\mathbf{x})$$

Hence, optimizing the upper loss will yield us the right c and θ.

T. The estimator for θ is consistent.

T. The estimator for θ is generally not statistically efficient.

19.2 — 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.

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

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).

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

Note that different factorizations of the distributions would also intractable

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

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

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

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 $q_\phi(\mathbf{z}\,|\,\mathbf{x})$ 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 $p_\theta(\mathbf{x}\,|\,\mathbf{z})$ 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 decoter netw. 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)

$$\begin{aligned}\log(p_\theta(\mathbf{x})) &= \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}\,|\,\mathbf{x})}\left[\log(p_\theta(\mathbf{x}))\right]\quad(\log(p_\theta(\mathbf{x}))\text{ does not depend on }\mathbf{z})\\ &= \mathbb{E}_\mathbf{z}\left[\log\left(\frac{p_\theta(\mathbf{x}\,|\,\mathbf{z})\,p_\theta(\mathbf{z})}{p_\theta(\mathbf{z}\,|\,\mathbf{x})}\right)\right]\quad\text{(Bayes Rule)}\\ &= \mathbb{E}_\mathbf{z}\left[\log\left(\frac{p_\theta(\mathbf{x}\,|\,\mathbf{z})\,p_\theta(\mathbf{z})}{p_\theta(\mathbf{z}\,|\,\mathbf{x})}\frac{q_\phi(\mathbf{z}\,|\,\mathbf{x})}{q_\phi(\mathbf{z}\,|\,\mathbf{x})}\right)\right]\quad\text{(Muliply by 1)}\\ &= \mathbb{E}_\mathbf{z}\left[\log(p_\theta(\mathbf{x}\,|\,\mathbf{z}))\right]-\mathbb{E}_\mathbf{z}\left[\log\left(\frac{q_\phi(\mathbf{z}\,|\,\mathbf{x})}{p_\theta(\mathbf{z})}\right)\right]+\mathbb{E}_\mathbf{z}\left[\log\left(\frac{q_\phi(\mathbf{z}\,|\,\mathbf{x})}{p_\theta(\mathbf{z}\,|\,\mathbf{x})}\right)\right]\\ &= \underbrace{\underbrace{\mathbb{E}_\mathbf{z}\left[\log(p_\theta(\mathbf{x}\,|\,\mathbf{z}))\right]}_{(1)}-\underbrace{KL(q_\phi(\mathbf{z}\,|\,\mathbf{x}),p_\theta(\mathbf{z}))}_{(2)}}_{\mathcal{L}(\mathbf{x},\theta,\phi)}+\underbrace{KL(q_\phi(\mathbf{z}\,|\,\mathbf{x}),p_\theta(\mathbf{z}\,|\,\mathbf{x}))}_{(3)\ge0}\end{aligned}$$

(1) Decoder network gives us $p_\theta(\mathbf{x}\,|\,\mathbf{z})$ 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) $p_\theta(\mathbf{z}\,|\,\mathbf{x})$ 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

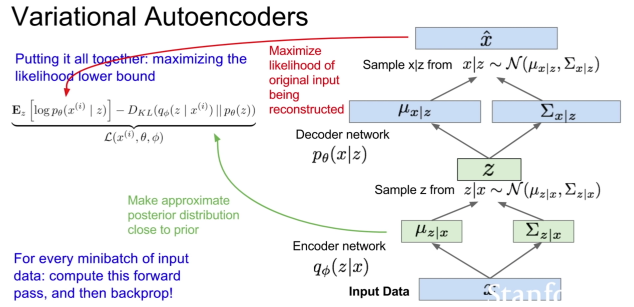
$\mathcal{L}(\mathbf{x},\theta,\phi)\le\log(p_\theta(\mathbf{x}))$

and we can take its gradient to optimize:

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

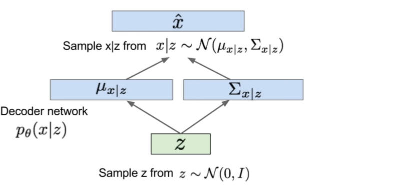
Reparametrization Trick

Forward Pass and Backpropagation


Variational Autoencoders
Putting it all together: maximizing the likelihood lower bound
$\mathbf{E}_z\left[\log p_\theta(x^{(i)}\,|\,z)\right]-D_{KL}(q_\phi(z\,|\,x^{(i)})\,||\,p_\theta(z))$
$\mathcal{L}(x^{(i)},\theta,\phi)$
Maximize likelihood of original input being reconstructed
$\hat{x}$
Sample x|z from $x|z\sim\mathcal{N}(\mu_{x|z},\Sigma_{x|z})$
$\mu_{x|z}$
$\Sigma_{x|z}$
Decoder network
$p_\theta(x|z)$
z
Sample z from $z|x\sim\mathcal{N}(\mu_{z|x},\Sigma_{z|x})$
Make approximate posterior distribution close to prior
$\mu_{z|x}$
$\Sigma_{z|x}$
Encoder network
$q_\phi(z|x)$
For every minibatch of input data: compute this forward pass, and then backprop!
Input Data

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.


$\hat{x}$
Sample x|z from $x|z\sim\mathcal{N}(\mu_{x|z},\Sigma_{x|z})$
$\mu_{x|z}$
$\Sigma_{x|z}$
Decoder network
$p_\theta(x|z)$
z
Sample z from $z\sim\mathcal{N}(0,I)$

19.3 — Deep Latent Gaussian Models

19.4 — 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:

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

$$\mathcal{R}^{(D)} = -\frac12\mathbb{E}_{\mathbf{x}\sim p_\text{data}}\left[\log(D(\mathbf{x}))\right]-\frac12\underbrace{\mathbb{E}_{\mathbf{z}\sim\mathcal{N}(\boldsymbol{\mu},\boldsymbol{\Sigma})}\left[\log(1-D(G(\mathbf{z})))\right]}_{=\mathbb{E}_{\mathbf{x}\sim p_\text{generator}}\left[\log(1-D(\mathbf{x}))\right]}$$

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

- 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.

$\frac{\partial\mathcal{R}^{(D)}}{\partial D(\mathbf{x})} = -\frac12\mathbb{E}_{\mathbf{x}\sim p_\text{data}}\left[\frac{1}{D(\mathbf{x})}\right]+\frac12\mathbb{E}_{\mathbf{x}\sim p_\text{generator}}\left[\frac{1}{1-D(\mathbf{x})}\right]\overset{!}{=}0$

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

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

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.

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

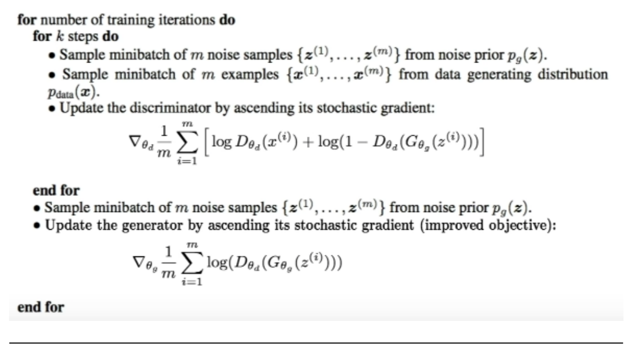
$\theta_g^{t+1}\leftarrow\theta_g^t-\eta_t\nabla_{\theta_g}\mathcal{R}^{(G)}(\theta_g)$

Optional: run $k\ge1$ 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 $\{z^{(1)},\dots,z^{(m)}\}$ from noise prior $p_g(z)$.
        • Sample minibatch of m examples $\{x^{(1)},\dots,x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
        • 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 $\{z^{(1)},\dots,z^{(m)}\}$ from noise prior $p_g(z)$.
    • 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
```