Intelligent System for Pattern Recognition

Lorenzo Leuzzi

2023

Contents

Chapter 1

Fundamentals of Pattern Recognition

1.1 Introduction to Signal Processing

1.1.1 Timeseries Formalization

A signal, a time series, is a sequence of measurements in time t. Time series analysis assumes weakly stationary (or second-order stationary) data so properties of the data remain constant over time (constant mean, constant variance).

1.1.2 Time domain analysis

Correlation and Convolution

Assesses how a signal changes over time.
Sample Mean:

$$\hat{\mu} = \frac{1}{N}\sum_{t=1}^{N} x^t \tag{1.1}$$

Auto-covariance is a measure of the linear dependence between two observations of a time series, separated by a certain time lag or time interval $-N \leq \tau \leq N$

$$\hat{\gamma}_x(\tau) = \frac{1}{N}\sum_{t=1}^{n-|\tau|} (x_{t+|\tau|} - \hat{\mu})(x_t - \hat{\mu}) \tag{1.2}$$

Auto-covariance serves to compute **auto-correlation**, the correlation of a signal with itself

$$\hat{\rho}(\tau) = \frac{\hat{\gamma}_x(\tau)}{\hat{\gamma}_x(0)} \tag{1.3}$$

Auto-correlation analysis can reveal repeating patterns such as the presence of a periodic signal hidden by noise.
The **Cross-correlation** is a measure of similarity of $\mathbf{x_1}$ and $\mathbf{x_2}$ as a function of a time lag τ.

$$\phi_{x^1x^2}(\tau) = \sum_{t=max\{0,\tau\}}^{min\{(T_1-1+\tau),(T_2-1)\}} x^1(t-\tau)x^2(t) \tag{1.4}$$

We can normalized it to return an amplitude independent value $\in [-1, +1]$. If the cross-correlation is: +1 they have the exact same shape, -1 the exact same shape but opposite sign, 0 completely uncorrelated signals. Cross-correlation and auto-correlation are **convolutions**.

Auto-regressive Process

A timeseries auto-regressive process (AR) of order K is the linear system (predict, forecast on dependent data, so time series)

$$x_t = \sum_{k=1}^{K} \alpha_k x_{t-k} + \epsilon_t \tag{1.5}$$

where α_k is a linear coefficients and ϵ is random white noise.
Autoregression refers to a process where the value of a variable at a given time point depends on its past values, while moving average refers to a process where the value of a variable at a given time point depends on the past values of an error term. In an **Autoregressive with Moving Average** (ARMA) process, both of these concepts are combined to model a time series as a linear combination of its past values and past error terms. Denotes new information or shocks at time t.

$$x_t = \sum_{k=1}^{K} \alpha_k x_{t-k} + \sum_{q=1}^{Q} \beta_q \epsilon_{t-q} + \epsilon_t \tag{1.6}$$

1.1.3 Spectral Analysis

Analyzing time series in the frequency domain can be helpful. Decompose a time series into a linear combination of sinusoids (and cosines) with random and uncorrelated coefficients.

Fourier Transform

Discrete Fourier transform (DFT) transforms a time series from the time domain to the frequency domain. It can be easily inverted (back to the time domain) and it is useful to handle periodicity in the time series.
Given an orthonormal system we can represent any function by a linear combination of the basis. Using $cos(kx) - isin(kx) = e^{-ikx}$ with $i = \sqrt{-1}$ we can rewrite the Fourier series as

$$\sum_{k=-\infty}^{\infty} X_k e^{ikx} \tag{1.7}$$

in the orthonormal system $1, e^{ix}, e^{-ix}, e^{2ix}, e^{-2ix}, ...$ Given a time series $\mathbf{x}$ its Discrete Fourier Transform (DFT) and the inverse transform are defined as:

$$X_k = \sum_{n=1}^{N-1} x_n e^{\frac{-2\pi i n k}{N}} \tag{1.8}$$

$$x_n = \frac{1}{N} \sum_{k=1}^{N-1} X_k e^{\frac{2\pi i n k}{N}} \tag{1.9}$$

We would like to measure relevance/strength/contribution of a target frequency bin k. **Amplitude** and **Power** are defined as follows:

$$A_k = |X_k| \tag{1.10}$$

$$P_k = \frac{|X_k|^2}{N} \tag{1.11}$$

Use the DFT elements $X_1, ..., X_k$ as representation of the signal to train predictor/classifier. Representation in spectral domain can reveal patterns that are not clear in time domain.

Spectral Centroid

The spectral centroid is a commonly used measure in spectral analysis of time series data. It is a measure of the center of mass of the power spectrum of a signal, which reflects the dominant frequency or frequencies in the signal. Mathematically, the spectral centroid is defined as the weighted mean of the frequencies in the power spectrum of a signal, where the weights are the corresponding amplitudes or power of each frequency component. Serves to measure brightness: e.g. music analysis (high hat detection) and genre classification. Spectral-weighted average frequency (between frequency bands b_1 and b_2)

$$\mu = \frac{\sum_{k=b_1}^{b_2} f_k s_k}{\sum_{k=b_1}^{b_2} s_k} \tag{1.12}$$

where f_k is the k-th frequency and s_k is the corresponding spectral weight (amplitude or power spectrum).

1.2 Image Processing - Visual Descriptors

1.2.1 Spatial Feature Descriptors

We have to choose a way that is informative, invariant to photometric and geometric transformations and efficient for indexing and querying.

Image Histograms

Represent the distribution of some visual information on the whole image (colors, edges, corners).

Scale Invariant Feature Transform

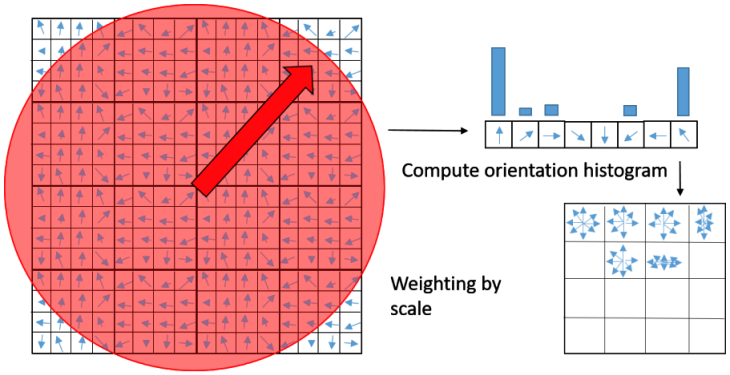
SIFT is a widely used algorithm for detecting and describing local features in digital images. Produces intensity gradient histogram.
Steps: **1.** Center the image patch on a pixel x, y of image I, **2.** represent image at scale σ (how close we look at an image) **convolving** the image with a Gaussian filter with std σ.

$$L_\sigma(x,y) = G(x,y,\sigma) * I(x,y) \qquad (1.13) \qquad G(x,y,\sigma) = exp(-\frac{x^2+y^2}{2\sigma^2}) \qquad (1.14)$$

3. Compute the **gradient** of intensity in the patch, with $G_x = \begin{bmatrix}1 & 0 & -1\end{bmatrix} * L_\sigma(x,y)$ and $G_y = \begin{bmatrix}1 & 0 & -1\end{bmatrix}^T * L_\sigma(x,y)$ the magnitude m and the orientation θ are computed as follows:

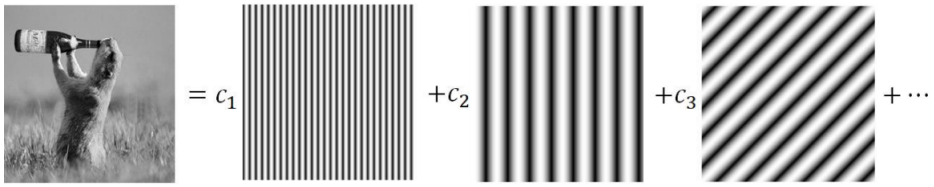
$$m_\sigma = \sqrt{G_x^2 + G_y^2} \qquad (1.15) \qquad \theta_\sigma(x,y) = tan^{-1}(\frac{G_y}{G_x}) \qquad (1.16)$$

4. Create **gradient histogram**



1.2.2 Spectral Analysis

Images are functions returning intensity values $I(x,y)$ on the 2D plane. I can write my image as sum of sine and cosine waves of varying frequency in x and y directions.



The Fourier transform of the convolution of two functions is the product of their Fourier transforms. Suppose we are given an image I (a function) and a filter g (a function as well) their convolution $I * g$ can be conveniently computed as: $I * g = F^1(F(I)F(g))$.

1.3 Image Processing - Visual Detectors

1.3.1 Edges and Gradients

The image gradient (greylevel) is the direction of change of intensity. Edges are pixel regions where the intensity of the gradient changes abruptly. $\nabla I = [\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}] = [G_x, G_y]$.

1.3.2 Blob Detection

Blobs are connected pixels regions with little gradient variability. **The Laplace of Gaussian** (LoG) is a commonly used method for detecting blobs in digital images. The LoG method involves convolving the image with a Gaussian filter and then computing the Laplacian of the filtered image. The resulting image highlights regions of the original image that have a similar intensity to a Gaussian function at a particular scale $\sqrt{2}\sigma$. $\nabla^2 g_\sigma(x,y) = \frac{\partial^2 g_\sigma}{\partial x^2} + \frac{\partial^2 g_\sigma}{\partial y^2}$. Laplacian-based detectors are invariant to scale but not to affine transformations.

Maximally Stable Extremal Regions

MSER extracts covariant regions (blobs) that are stable connected components of intensity sets of the image. The basic idea behind MSER is to identify regions of the image that remain connected and have a consistent intensity under thresholding at different levels. These regions are called maximally stable extremal regions because they are connected components of the image that have a stable and extreme response to thresholding.

1.3.3 Image Segmentation

The process of partitioning an image into set of homogeneous pixels, hoping to match object or their subparts.

Normalized Cuts

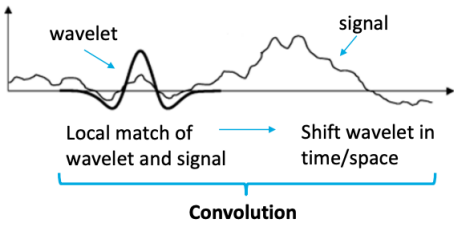
Normalized Cut (Ncut) is a popular graph-based image segmentation technique that uses spectral clustering to partition an image into disjoint regions. Ncut aims to find a segmentation that minimizes the dissimilarity between the regions while maximizing the similarity within the regions.

Superpixels

Pixels in image are a lot! An efficiency trick is to group together similar pixels and applying segmentation/fusion algorithms to the obtained superpixels.

1.4 Wavelets

Sometimes we might need localized frequencies rather than global frequency analysis. Split signal in frequency bands only if they exist in specific time-intervals or portion of the space.



Wavelets are similar to Fourier transforms in that they represent signals as a combination of sine and cosine waves with different frequencies and amplitudes but they use a set of basis functions that can be adapted to the local features of the signal. Split the signal using an orthonormal basis generated by translation and dilation of a mother wavelet. Many different possible choices for the mother wavelet function.

Chapter 2

Generative Learning

2.1 Generative and Graphical Models

ML models that represent knowledge inferred from data under the form of **probabilities**.
Graphical Model Representation: graphs whose nodes (vertices) are random variables whose edges (links) represent probabilistic relationships between the variables.

2.1.1 Probability Refresher

The world and observations are **represented** as: random variables $X_1, ..., X_N$ and joint probability distribution $P(X_1 = x_1, ..., X_N = x_n)$.
There's rules for **manipulating** the probabilistic knowledge: Sum-Product, Marginalization, Bayes, Conditional Independence. **Learning** is about discovering the values for $P(X_1 = x_1, ..., X_N = x_n)$. Given a set of observations $\mathbf{d}$ and a probabilistic model of a given structure, how do I find the parameters θ of its distribution.

2.1.2 Inference and learning with probabilities

Approaches to Inference

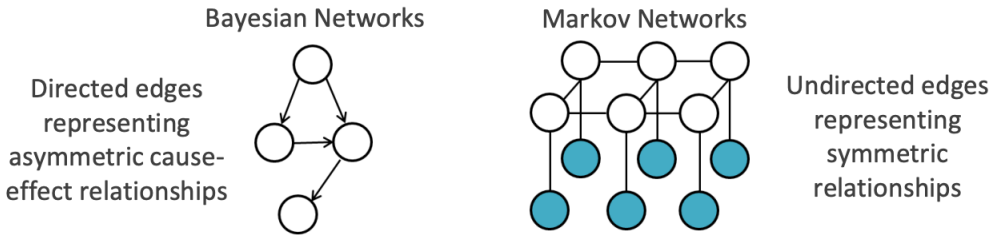
Bayesian considers all hypotheses weighted by their probability $P(X|\mathbf{d}) = \sum_i P(X|h_i)P(h_i|\mathbf{d})$.
Maximum a-Posteriori infers X from $P(X|h_{MAP})$ where $h_{MAP} = \arg\max_{h \in H} P(h|\mathbf{d}) = \arg\max_{h \in H} P(\mathbf{d}|h)P(h)$.
Maximum Likelihood assumes uniform priors $P(h_i) = P(h_j)$ and estimates $P(X|h_{ML})$ where $h_{ML} = \arg\max_{h \in H} P(\mathbf{d}|h)$.
ML and MAP are point estimates of the Bayesian since they infer based only on one most likely hypothesis. MAP is a regularization (penalize complexity) of the ML estimation.

Maximum Likelihood Learning

Find the model θ that is most likely to have **generated** the data $\mathbf{d}$: $\theta_{ML} = \arg\max_\theta P(\mathbf{d}|\theta)$. The Likelihood function for the optimization problem is $L(\theta|x) = P(x|\theta)$ can be addressed by solving $\frac{\partial L(\theta|x)}{\partial \theta} = 0$.

2.1.3 Graphical Models

Compact graphical representation for **exponentially large joint distributions**. Simplifies marginalization and inference algorithms. Allow to incorporate **prior knowledge** concerning causal relationships and associations between random variables (RVs): directed Graphical Models (Bayesian Networks), undirected Graphical Models (Markov Random Fields).



2.2 Conditional independence and Causality

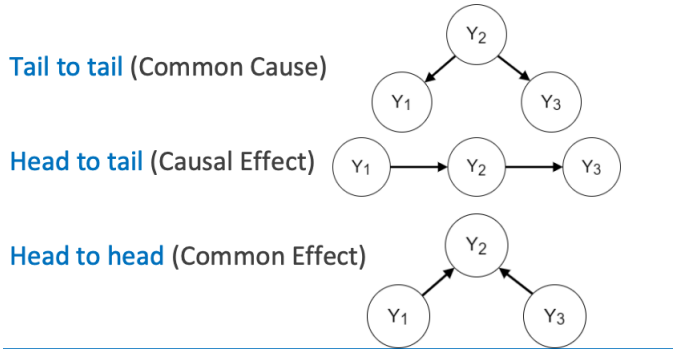
2.2.1 Bayesian Network

Direct Acyclic Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ are nodes that represent random variables (hidden or observed) and $\mathcal{E}$ are edges that represent conditional dependence relationship. If the N nodes have a maximum of L children then $(k-1)^L \times N$ independent parameters.

Markov Property

Each node is conditionally independent of all its non-descendants given a joint state of its parents. The **Markov Blanket** $Mb(A)$ of a node A is is the minimal set of vertices that shield the node from the rest of Bayesian Network. The behavior of a node can be completely determined and predicted from the knowledge of its Markov blanket.

Fundamentals Bayesian Network Structures



A Bayesian Network represents the local relationships encoded by the 3 basic structures plus the derived relationships.

d-separation

Let r be an undirected path bewtween two nodes then r is d-separated by Z if there exist at least one node $Y_c \in Z$ for which path r is blocked. The Markov blanket $Mb(Y)$ is the minimal set of nodes which d-separates a node Y from all other nodes.

2.2.2 Markov Random Fields

Bayesian Networks are used to model asymmetric dependencies (e.g. causal). What if we want to model symmetric dependencies: bidirectional effects, need undirected approaches.

Clique

A subset of nodes $\mathcal{C}$ in graph $\mathcal{G}$ such that $\mathcal{G}$ contains an edge between all pair of nodes in $\mathcal{C}$. A **Maximal Clique** is a clique that cannot include any further node from the graph without ceasing to be a clique.

What is the undirected equivalent of conditional probability factorization in directed models? **Maximal Clique Factorization**. Define $\mathbf{X}$ as the RVs associated to the N nodes in the undirected graph.

$$P(\mathbf{X}) = \frac{1}{Z} \prod_C \psi(\mathbf{X}_C) \tag{2.1}$$

where $\mathbf{X}_C$ are RVs associated with nodes in the maximal clique C, ψ is the potential function over the maximal clique, Z is the partition function ensuring normalization $Z = \sum_X \prod_X \psi(\mathbf{X}_C$ (serious computational issues). So the factorization is in terms of generic potential functions (not probabilities).

2.2.3 Structure Learning

Observations are given for a set of fixed random variables but network structure is not specified, learning to infer **multivariate causation relationships** from data.

Search and Score

It is a model selection approach. Search the space Graph(Y) of graphs G_k, score each structure by $S(G_k)$ and return the highest scoring graph G^*. The scoring function S has to have the following properties: consistency (same score for graphs in the same equivalence class), decomposability (can be locally computed). Finding maximal scoring structures is NP complete so we have to constrain both the search and the space.

Constrain-based Models

We test conditional independence, $I(X_i, X_y|Z)$ determine edge presence (network skeleton). Use deterministic rules based on local Markovian dependencies to determine edge orientation. Choice of the testing order is fundamental for avoiding a super-exponential complexity. Testing can be: level-wise (order of increasing size of the conditioning set Z, PC algorithm) or node-wise (single edge at the time, exhausting independence checks on all conditioning variables).

Hybrid Models

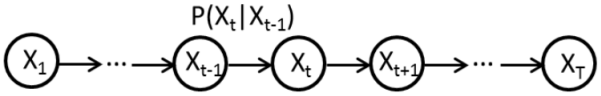
Multi-stage algorithms combining previous approaches. Independence tests to find a sub-optimal skeleton (good starting point) then search and score starting from the skeleton.

2.3 Hidden Markov Model

2.3.1 Introduction to Hidden Markov Model

Markov Chain

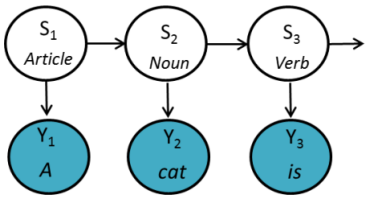
Directed graphical model for sequences s.t. element X_t only depends on its predecessor in the sequence (First-Order).



Joint probability factorizes as $P(\mathbf{X}) = P(X_1, ..., X_T) = \underbrace{(X_1)}_{\text{prior}} \prod_{t=2}^{T} \underbrace{P(X_t|X_{t-1})}_{\text{transition}}$.

Definition

Stochastic process where transition dynamics is disentangled from observations generated by the process.



State transition is an unobserved (hidden/latent) process characterized by the hidden state variables S_t: $A_{ij} = P(S_t = i|S_{t-1} = j)$ and $\pi_i = P(S_1 = i)$.
Observation are generated by the **emission distribution**: $b_i(y_t) = P(Y_t = y_t|S_t = i)$.

Joint Probability Factorization for HMMs

$$P(\boldsymbol{Y} = \boldsymbol{y}) = \sum_{\boldsymbol{s}} P(\boldsymbol{Y} = \boldsymbol{y}, \boldsymbol{S} = \boldsymbol{s})$$
$$= \sum_{s_1,...,s_T} \left\{ P(S_1 = s_1)P(Y_1 = y_1|S_1 = s_1) \prod_{t=2}^{T} P(S_t = s_t|S_{t-1} = s_{t-1})P(Y_t = y_t|S_t = s_t) \right\}$$

Inference Problems

Smoothing: Given a model θ and an observed sequence $\mathbf{y}$, determine the distribution of the hidden state at time t $P(S_t|\mathbf{Y} = \mathbf{y}, \theta)$.
Learning: Given a dataset of N observed sequences $\mathcal{D} = \mathbf{y}^1, ..., \mathbf{y}^N$ and the number of hidden states C, find the parameters π, A, B that maximize the probability of the model θ having generated the sequences $\mathcal{D}$.
Optimal State Assignment: Given a model θ and an observed sequence $\mathbf{y}$, find an optimal state assignment $\mathbf{s} = s_1^*, ..., s_T^*$ for the hidden Markov chain.

2.3.2 Forward-Backward Algorithm

How do we determine the posterior $P(S_t = i|\mathbf{y})$? The forward-backward algorithm is a dynamic programming algorithm used to do smoothing inference, determine the probability of one observed state (S_t)given the observations ($\mathbf{y}$). Reminder that A_{ij} and b_i are respectively the transition and emission distributions.

$$P(S_t = i|\mathbf{y}) = P(S_t = i, \mathbf{Y_{1:t}})P(\mathbf{Y_{t+1:T}}|S_t = i) = \alpha_t(i)\beta_t(i) \tag{2.2}$$

α-term computed as a part of the **forward recursion** ($\alpha_1(i) = b_i(y_1)\pi_i$)

$$\alpha_t(i) = P(S_t = i, \mathbf{Y}_{1:t}) = b_i(y_t) \sum_{j=1}^{C} A_{ij}\alpha_{t-1}(j) \tag{2.3}$$

In this phase, we compute the probability of observing the first t outputs and being in state i at time t. This is done recursively using the probabilities from the previous time step.
β-term computed as a part of the **backward recursion** ($\beta_T(i) = 1$)

$$\beta_t(j) = P(\mathbf{Y_{t+1:T}}|S_t = j) = \sum_{i=1}^{C} b_i(y_{t+1})\beta_{t+1}(i)A_{ij} \tag{2.4}$$

In this phase, we compute the probability of observing the remaining outputs after time t, given that we are in state i at time t. This is done recursively using the probabilities from the next time step.

2.3.3 Learning in HMM

Learning HMM parameters $\theta = (\pi, A, B)$ by maximization of the **complete likelihood**.

$$\mathcal{L}(\theta) = \log \prod_{n=1}^{N} P(\boldsymbol{Y}^n|\theta) = \underset{\text{iterates thru training samples}}{\log \prod_{n=1}^{N}} \left\{ \sum_{s_1^n,\ldots,s_{T_n}^n} \overbrace{P(S_1^n)P(Y_1^n|S_1^n)}^{\text{emission}} \underbrace{\prod_{t=2}^{T_n} P(S_t^n|S_{t-1}^n)P(Y_t^n|S_t^n)}_{\text{iterates thru states of the sequence}} \right\}$$

Introducing **indicator variables**

$$z_{ti}^n = \begin{cases} 1 \text{ if the } n\text{-th chain is in state } i \text{ at time } t \\ 0 \text{ otherwise} \end{cases} \tag{2.5}$$

$$\mathcal{L}_c(\theta) = \log P(\mathcal{X}, \mathcal{Z}|\theta) = \log \prod_{n=1}^{N} \left\{ \prod_{i=1}^{C} [P(S_1 = 1)P(Y_1^n|S_1 = i)]^{z_{1i}^n} \right.$$
$$\left. \prod_{t=2}^{T_n} \prod_{i,j=1}^{C} P(S_t = i|S_{t-1} = j)^{z_{ti}^n z_{(t-1)j}^n} P(Y_t^n|S_t = i)^{z_{ti}^n} \right\}$$
$$= \sum_{n=1}^{N} \left\{ \sum_{i=1}^{C} z_{1i}^n \overbrace{\log \pi_i}^{\text{prior}} + \sum_{t=2}^{T_n} \sum_{i,j=1}^{C} z_{ti}^n z_{(t-1)j}^n \overbrace{\log A_{ij}}^{\text{transition}} + \sum_{t=1}^{T_n} \sum_{i=1}^{C} z_{ti}^n \overbrace{\log b_i(y_t^n)}^{\text{emission}} \right\}$$

Iterates thru finite number of hidden states

2.3.4 Expectation Maximization

A 2-step iterative algorithm for the maximization of complete likelihood.

E-step

Given the current estimate of the model parameters θ , compute

$$Q^{(k+1)}(\theta|\theta^{(k)}) = \mathbb{E}_{\mathcal{Z}|\mathcal{X},\theta^{(t)}}[\log P(\mathcal{X}, \mathcal{Z}|\theta)] \tag{2.6}$$

To compute that conditional expectation for the complete HMM log-likelihood we need to estimate $E_{\mathcal{Z}|\mathbf{Y},\theta^{(t)}}[z_{ti}] = P(S_t = i|\mathbf{y})$ and $E_{\mathcal{Z}|\mathbf{Y},\theta^{(t)}}[z_{ti}z_{(t-1)j}] = P(S_t = i, S_{t-1} = j|\mathbf{y})$.
We know how to compute those posteriors by the **forward-backward algorithm**.

M-step

Find the new estimate of the model parameters, using the posteriors computed in the E-step and solving the **optimization problem**.

$$\theta^{k+1} = \arg\max_{\theta} Q^{k+1}(\theta|\theta^{(k)}) \tag{2.7}$$

$$\frac{\partial Q^{(k+1)}(\theta|\theta^{(k)})}{\partial \theta} \tag{2.8}$$

2.3.5 Decoding Problem

Find an **optimal hidden state assignment** $\mathbf{s} = s_1^*, ..., s_T^*$ for an observed sequence $\mathbf{y}$ given a trained HMM. This can be done two ways: identify the single hidden states s_t that maximize the posterior $P(S_t = i|Y)$ or find the most likely joint hidden state assignment $\mathbf{s}^* = \arg\max_s P(\mathbf{Y}, \mathbf{S} = \mathbf{s})$.

Viterbi algorithm

The last problem is addressed by the Viterbi algorithm. It is an efficient dynamic programming algorithm based on a backward-forward recursion.

Recursive backward term

$$\epsilon(s_{t-1}) = \max_{s_t} P(Y_t|S_t = s_t)\, P(S_t = s_t|S_{t-1} = s_{t-1})\epsilon(s_t),$$

Root optimal state

$$s_1^* = \arg\max_s P(Y_t|S_1 = s)\, P(S_1 = s)\epsilon(s).$$

Recursive forward optimal state

$$s_t^* = \arg\max_s P(Y_t|S_t = s)\, P(S_t = s|S_{t-1} = s_{t-1}^*)\epsilon(s).$$

2.3.6 Dynamic Bayesian Networks

A graphical model whose **structure changes** to reflect information with variable size and connectivity patterns. Suitable for modeling structured data (sequences, tree, ...).

2.4 Markov Random Fields

2.4.1 Undirected Graphical Models

Markov Random Fields (MFRs) are undirected graph where nodes $v \in V$ represent random variables and edges $e \in \mathcal{E}$ describe bi-directional dependencies between variables (constraints).

2.4.2 Likelihood Factorization

Define $\mathbf{X} = X_1, ..., X_N$ as the RVs associated to the N nodes in the undirected graph.

$$P(\mathbf{X}) = \frac{1}{Z} \prod_c \psi_C(\mathbf{X}_C) \tag{2.9}$$

The **maximal clique factorization** of an MRF is useful because it allows us to factor the joint probability distribution of the MRF as a product of smaller, more manageable terms.

Potential Functions

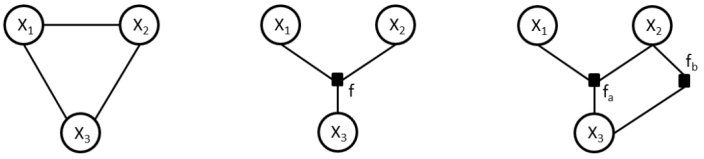
Potential functions $\psi_c(\mathbf{X}_C)$ are not probabilities, they express which configurations of the local variables are preferred. In other words they encode the local interactions between the variables in the clique C, and the product of the potential functions over all maximal cliques captures the global dependencies between the variables in the MRF. If we restrict to strictly positive potential functions, the Hammersley-Clifford theorem provides guarantees on the distribution that can be represented by the clique factorization.
In general, we will assume to work with MRF where the partition functions factorize as

$$\psi_C(\mathbf{X}_C) = exp(\sum_k \theta_{Ck} f_{Ck}(\mathbf{X}_C)) \tag{2.10}$$

where f_{Ck} are feature functions or sufficient statistics to compute the potential of clique C, $\theta_{CK} \in \mathbb{R}$ are parameters, k indexes over the available feature functions.

2.4.3 Factor Graph

Undirected graphical models do not express the factorization of potentials into feature functions, factor graphs do.



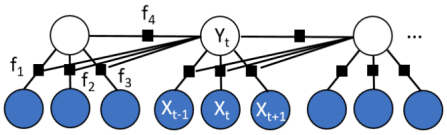
$$\psi(X_1, X_2, X_3) = f(X_1, X_2, X_3)$$
$$\psi(X_1, X_2, X_3) = f_a(X_1, X_2, X_3) f_b(X_2, X_3)$$

2.4.4 Conditional Random Field

In ML a part of the random variables, the input data $\mathbf{X}$, can be assumed to be always observable.

$$P(\mathbf{Y}|\mathbf{X}, \theta) = \frac{1}{Z(\mathbf{X})} \prod_k exp\{\theta_k, f_k(\mathbf{X_k}, \mathbf{Y_k})\} \tag{2.11}$$

(Linear) Conditional Random Field can be seen as the generalized version of a HMM.



General LCRF likelihood

$$P(\mathbf{Y}|\mathbf{X}, \theta) = \frac{1}{Z(\mathbf{X})} \prod_t \prod_k exp\{\theta_k f_k(Y_t, Y_{t-1}, \mathbf{X_t})\} \tag{2.12}$$

Our model takes into account what is the current hidden state Y_t the past hidden state Y_{t-1} and some observation $\mathbf{X}_t$.

Inference in LCRF

Smoothing The posterior inference $P(Y, Y_{t-1}|\mathbf{X})$ is solved by (exact) forward-backward inference.

$$P(Y_t, Y_{t-1}|\mathbf{X}) \simeq \underbrace{\alpha_{t-1}(Y_{t-1})}_{\text{Forward message}} \underbrace{\psi_t(Y_t, Y_{t-1}, X_t)}_{\text{Clique weighting}} \underbrace{\beta_t(Y_t)}_{\text{Backward message}} \tag{2.13}$$

Clique weighting

$$\psi_t(Y_t, Y_{t-1}, X_t) = \exp\{\theta_e f_e(X_t, Y_t) + \theta_t f_t(Y_{t-1}, Y_t)\}$$

Forward Message

$$\alpha_t(i) = \sum_j \psi_t(i, j, X_t) \alpha_{t-1}(j)$$

Backward Message

$$\beta_t(j) = \sum_i \psi_{t+1}(i, j, X_{t+1}) \beta_{t+1}(i)$$

Max-product inference can be performed as in the **Viterbi algorithm** for HMM. If the CRF does not have a chain structure we cannot perform the exact inference because it's computationally impracticable, so we just need to approximate this inference and we can do it with Monte-Carlo methods (for example).

Training LCRF

Maximum (conditional) log-likelihood

$$\max_\theta \mathcal{L}(\theta) = \max_\theta \sum_n \log P(\mathbf{y^n}|\mathbf{x^n}, \theta) \tag{2.14}$$

Substituting LCRF conditional formulation and penalizing with a regularization term.

$$\mathcal{L}(\theta) = \overline{\sum_{n}} \overline{\sum_{t}} \overline{\sum_{k}} \theta_k f_k(Y_t^n, Y_{t-1}^n, \boldsymbol{X}_t^n) - \overline{\sum_{n}} \log Z(\boldsymbol{X}^n) - \overline{\sum_{k}} \frac{\theta_k^2}{2\sigma^2}$$

samples states potential fun

The gradient is:

$$\frac{\partial \mathcal{L}(\theta)}{\partial \theta_k} = \underbrace{\sum_{n,k} f_k(Y_t^n, Y_{t-1}^n, X_t^n)}_{\mathbb{E}_{y,y' \sim \text{ dataset}}[f_k(y,y',X_t^n)]} - \underbrace{\sum_{n,t} \sum_{y,y'} f_k(y, y', X_t^n) P(y, y' \mid X_t^n)}_{\mathbb{E}_{P(Y \mid X,\theta)}[f_k(y,y',X_t^n)]} - \frac{\theta_k}{\sigma^2} \tag{2.15}$$

We have sum of expectations: the first term is $\mathbb{E}[f_k]$ when Y are samples drawn from a finite dataset (**empirical distribution**); the right term is $\mathbb{E}[f_k]$ using the posterior so the expectation of the feature function under the (**model distribution**). We need to match those two expectations, meaning that when the gradient is zero these are equal.

$$\frac{\partial \mathcal{L}(\theta)}{\partial \theta_k} = \mathbb{E}_{y,y' \sim \text{ dataset}}[f_k(y, y', x_t^n)] - \mathbb{E}_{P(Y|X,\theta)}[f_k(y, y', x_t^n)] \tag{2.16}$$

Optimizing the likelihood via **stochastic gradient descent**: $\theta^m = \theta^{m-1} - v_m \nabla \mathcal{L}_n(\theta^{m-1})$

2.5 Bayesian Learning and Variational Inference

2.5.1 Introduction to the problem

Variational learning is a framework for **approximate inference** in probabilistic models. It is a family of techniques that involve approximating complex probability distributions with simpler ones that are easier to work with. The goal of variational learning is to find an approximation to the posterior distribution of the latent variables in a probabilistic model given the observed data.
A model is said to be **intractable** to compute if its evaluation, optimization, or inference requires an impractically large amount of computational resources, either in terms of time or space.
Latent variables are unobserved random variables that define a hidden generative process of observed data. They explain the complex relations between many observable variables. Examples of latent variables are the hidden states in HMM/CRF. Latent variable models likelihood:

$$P(x) = \int_{\mathbf{s}} \prod_{i=1}^{N} P(x_i|\mathbf{z}) P(\mathbf{z}) d\mathbf{z} \tag{2.17}$$

2.5.2 Kullback-Leibler Divergence

Approximating the posterior is achieved by minimizing a **measure of the distance between two distribution**, the true posterior and the approximating distribution, such as the Kullback-Leibler (KL) divergence.

$$KL(q||p) = \mathbb{E}[\log \frac{q(z)}{p(z|x)}] = \mathbb{E}_q[\log q(z)] - \mathbb{E}_q[\log p(z|x)] = \langle \log q(z) \rangle_q - \langle \log p(z|x) \rangle_q \tag{2.18}$$

If q high and p high $\Rightarrow$ happy (KL close to 0), if q high and p low $\Rightarrow$ unhappy, if q low $\Rightarrow$ don't care (we are taking expectation). It's a divergence $\Rightarrow$ it's not symmetric.

2.5.3 Jensen Inequality

Property of linear operators on convex/concave functions applied in our **probabilistic setting**

$$f(\mathbb{E}[X]) \geq \mathbb{E}[f(X)] \tag{2.19}$$

Intuitively, the Jensen inequality states that the expectation of a convex function (log) is always less than or equal to the convex function of the expectation. In other words, if we apply a convex function to the expected value of a random variable, the result is always greater than or equal to the expected value of the convex function applied to the random variable.

2.5.4 Evidence Lower Bound (ELBO)

Bounding Log-likelihood with Jensen

The log-likelihood for a model with a single hidden variable Z and parameters θ with a single sample assumed for simplicity is the following:

$$\log P(x|\theta) = \log \int_z P(x,z|\theta)dz = \log \int_z \frac{Q(z|\phi)}{Q(z|\phi)} P(x,z|\theta)dz \tag{2.20}$$

Q is a distribution, used over z with parameters ϕ and $Q(z|\phi) \neq 0$. That is the definition of expectation, we have $\int_z Q(z|\phi)\frac{1}{Q(z|\phi)}P(x,z|\theta)$ which is $\sum_z q \cdot g(z)$.
Using Jensen we can get rid of the fact that was a nasty logarithm over a summation and rewrite it as:

$$\log P(x|\theta) = \log \mathbb{E}_Q \left[\frac{P(x,z)}{Q(z)}\right] \geq \mathbb{E}_Q \left[\log \frac{P(x,z)}{Q(z)}\right] = \tag{2.21}$$

$$= \underbrace{\mathbb{E}_Q \left[\log P(x,z)\right]}_{\text{Expectation of Joint Distribution}} - \underbrace{\mathbb{E}_Q \left[\log Q(z)\right]}_{\text{Entropy}} = \mathcal{L}(x,\theta,\phi) \tag{2.22}$$

The log likelihood of the model is lower bounded $P(x|\theta) \geq \mathcal{L}(x,\theta,\phi)$. But how good is this lower bound? $P(x|\theta) - \mathcal{L}(x,\theta,\phi) = ?$.

$$\log P(x|\theta) - \mathcal{L}(x,\theta,\phi) = ?$$

Inserting the definition of $\mathcal{L}(x,\theta,\phi)$

$$\log P(x|\theta) - \int_z Q(z) \log \frac{P(x,z)}{Q(z)} dz$$

Introducing $Q(z)$ by marginalization ($\int_z Q(z) = 1$)

$$\int_z Q(z) \log P(x) dz - \int_z Q(z) \log \frac{P(x,z)}{Q(z)} dz =$$

$$\mathbb{E}_Q \left[\log \frac{Q(z)}{P(z|x)}\right] = KL\big(Q(z|\phi)||P(z|x,\theta)\big)$$

We can assume the existence of a probability $Q(z|\phi)$ which allows to bound the likelihood $P(x|\theta)$ from below using the term $\mathcal{L}(x,\theta,\phi)$ called **variational bound or evidence lower bound (ELBO)**. The optimal bound is obtained when the $KL(Q(z|\phi)||P(z|x,\theta)) = 0$ so if we choose $Q(z|\phi) = P(z|x,\theta)$. Minimizing KL is equivalent to maximize the ELBO so change a sampling problem with an optimization problem.

2.5.5 Variational View of Expecatation Maximization

Maximum likelihood learning with hidden variables can be approached by maximization of the ELBO.

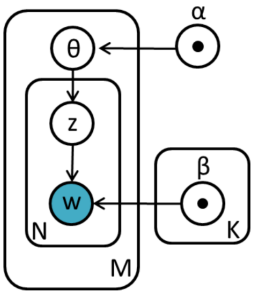
$$\max_{\theta,\phi} \sum_{n=1}^{N} \mathcal{L}(x_n,\theta,\phi) \tag{2.23}$$

If $P(z|x,\theta)$ is tractable then we use it as $Q(z|\phi)$ (optimal ELBO). Otherwise we choose $Q(z|\phi)$ as a tractable family of distributions: find ϕ that minimize $KL(Q(z|\phi)\|P(z|x,\theta))$ or find ϕ that maximize $L(\cdot,\phi)$.

2.6 Latent Dirichlet Allocation

Latent topic models consider documents (i.e. item containers) as a **mixture of topics**.
Latent Dirichlet Allocation (LDA) is a generative probabilistic model that is used for **topic modeling**. The goal of LDA is to identify a set of latent topics that best explain a collection of documents

assigning one topic z to each item w with probability $P(w|z,\beta)$ and picking one topic for the whole document with probability $P(z|\theta)$. Each document has its personal topic proportion θ sampled from a distribution.



$P(w|z,\beta)$ multinomial item-topic distribution.
$P(z|\theta)$ multinomial topic distribution with document-specific parameter θ.
$P(\theta|\alpha)$ Dirichlet distribution with hyperparameter α.

2.6.1 Dirichlet Distribution

Why a Dirichlet distribution? The Dirichlet distribution is a conjugate prior for the multinomial distribution. We use this particular distribution because we can interpret it as a way to introduce a prior distribution to a multinomial one. If the likelihood is multinomial, it's guaranteed that using a Dirichlet Prior, the posterior will be Dirichlet itself. This property makes it easy to update the topic mixture distribution as new data is observed, thus provides a natural way to model the distribution of topics in a corpus of text data.
This distribution uses a hyperparameter α to set the sparsity level.

2.6.2 LDA Generative Process

For each M documents choose $\theta \sim Dirichlet(\alpha)$; for each of the N items, choose a topic $z \sim$ Multinomial(θ) and pick an item w_j with multinomial probability $P(w_j|z,\beta)$. α and β are hyper-parameters, in particular β is a multinomial topic-item parameter matrix.

$$P(\theta, \mathbf{z}, \mathbf{w}|\alpha, \beta) = P(\theta|\alpha) \prod_{j=1}^{N} P(z_j|\theta) P(w_j|z_j, \beta) \tag{2.24}$$

2.6.3 Learning in LDA

Learning with Latent Dirichlet Allocation (LDA) models involves estimating the latent topic structure of a corpus of text data given the observed words in the documents.

$$P(\mathbf{w}|\alpha,\beta) = \int \sum_{z} P(\theta, \mathbf{z}, \mathbf{w}|\alpha, \beta) d\theta = \int P(\theta|\alpha) \prod_{j=1}^{N} \sum_{z_j=1}^{k} P(z_j|\theta) P(w_j|z_j, \beta) d\theta \tag{2.25}$$

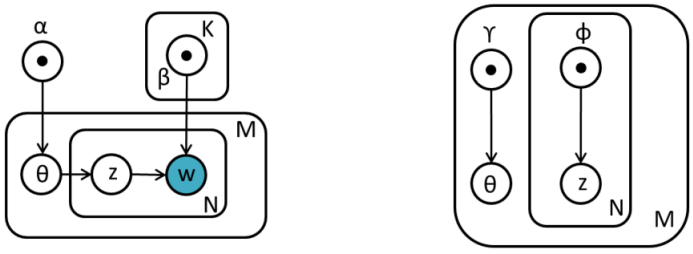
Given $\mathbf{w_1}, ..., \mathbf{w_M}$ find (α, β) maximizing

$$\mathcal{L}(\alpha, \beta) = \log \prod_{i=1}^{M} P(\mathbf{w}_i|\alpha, \beta) \tag{2.26}$$

Since we are learning with random variables we need the **Expectation-Maximization** algorithm which requires the posterior $P(\theta, \mathbf{z}|\mathbf{w}, \alpha, \beta)$ to be computed. The key problem is that it's (of course) **untractable** due to a combinatorial sum over all the possible couples between β and θ. You can approximate this inference in LDA using: variational inference (fast convergence but it is an approximation) or sampling approaches (slow convergence but it is as accurate as you wish).

Variational Inference

Write a $Q(\mathbf{z}|\phi)$ function that is sufficiently similar to the posterior but tractable and fit parameter ϕ so that $Q(\mathbf{z}|\phi)$ is close to $P(\mathbf{w}|\alpha, \beta)$ according to KL. Optimal ELBO is achieved when Q is equal to the latent variable posterior P.



Given $\Phi = \{\gamma, \phi\}$ as variational approximation parameters.

$$Q(\theta, \mathbf{z}|\Phi) = Q(\theta|\gamma) \prod_{n=1}^{N} Q(z_n|\phi_n) \tag{2.27}$$

Then we have the model parameters β of sample distribution $P(\theta, \mathbf{z}, \mathbf{w}|\beta)$.

Variational Expectation-Maximization

Find the Φ, β that maximize the ELBO

$$\mathcal{L}(\mathbf{w}, \Phi, \beta) = \mathbb{E}_Q[\log P(\theta, \mathbf{z}, \mathbf{w}|\beta)] - \mathbb{E}_Q[\log Q(\theta, \mathbf{z}|\Phi)] \tag{2.28}$$

by iterating EM algorithm: fix β update variational parameters Φ^* (**E-step**), fix $\Phi = \Phi^*$ update model parameter β^* (**M-step**).

2.7 Sampling methods

2.7.1 Sampling

Sampling consists in drawing a set of realizations $X = x_1, .., x_L$ of a random variable x with distribution $p(x)$. Why do we need sampling? For **approximating expectations**, we cannot compute $E_{p(x)}[f(x)]$ enumerating alll states of x if $p(x)$ is intractable. Also for **learning parameters** by sampling their posteriors.
Drawing samples from a **univariate distribution** is easy! We only need a random number generator R which produces a value uniformly at random in $[0, 1]$.

Properties of Sampling

The empirical distribution **converges** almost surely to the **true distribution**.
Let $\tilde{p}(x)$ the distribution over all possible realizations of the sampling set X, then $\hat{f}_X$ is an **unbiased** estimator if the approximation is exact on average $\Leftarrow \tilde{p}(x^l) = p(x^l)$.
The sampling approximation $\hat{f}_X$ of the expectation can have **low variance**, the variance tells us how much we can rely on the approximation, if variance is low $\hat{f}(X)$ is (quite) always close to its expected value $\Leftarrow \tilde{p}(x^l, x^{l'}) = \tilde{p}(x^l)\tilde{p}(x^{l'})$.
The last two properties are desirable but difficult to ensure!

2.7.2 Multivariate Sampling

Each sample x^l contains n values.

X	S_1	S_2	S_3	S_4	S_5
x^1	1	1	2	4	5
x^2	4	3	2	1	2
x^3	5	2	5	3	4
⋮	⋮	⋮	⋮	⋮	⋮
x^L	3	5	6	6	1

Naive Approach

We build an univariate distribution $p(S)$, where S is a discrete variable with C^n states (for exaple $p(1,1,1), p(1,1,2), ..., p(C,C,C)$). Now we can sample from $p(S)$ using the univariate schema (computationally infeasible).
Another way is using the **chain rule**, we can rewrite the joint distribution as:
$p(s_1, ..., s_n) = p(s_1)p(s_2|s_1)...p(s_n|s_1, ..., s_{n-1})$. Then we sample the variables in the following order: sample $\tilde{s}_1 \sim p(s_1)$; sample $\tilde{s}_2 \sim p(s_2|\tilde{s}_1)$; sample $\tilde{s}_n \sim p(s_n|\tilde{s}_1, ..., \tilde{s}_{n-1})$

2.7.3 Ancestral Sampling

The approach used previously is called Ancestral Sampling (AS). It is a method for generating a sample from a joint probability distribution over a set of random variables by sampling the variables in a specific order, based on the **structure of the graphical model** that represents the joint distribution. AS performs exact sampling since each sample x^l is drawn from $p(x)$. The samples are also independent, thus, AS has low variance. if $p(x)$ is a BN, we can use AS (valid and with low variance).

2.7.4 Sampling with evidence

We need an efficient method to sample under evidence (condition our sampling on the data), we can't use AS because we would need to compute the new structure which is complex as running exact inference.

Gibbs Sampling

The idea is to start from a sample $x_1 = s_1^1, ..., s_n^1$ and to update only one variable at a time generating x_2 etc. During the $(l+1)$-th iteration, we select a variable s_j and we sample its value according to $s_j^{l+1} \sim p(s_j|s_{\setminus j})$. It depends only on the Markov blanket of s_j, easy to sample! The Gibbs sampling draws a new sample x^l from $q(x^l|x^{l-1})$.
Properties: we are not sampling from p(x) but in the limit of infinite samples, the Gibbs sampler is valid. It does NOT have low variance, samples are highly dependent.

Markov Chain Monte Carlo Sampling Framework

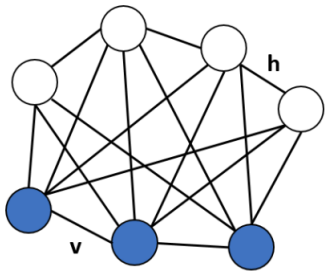
Gibbs sampling is a specialization of the MCMC sampling framework. The idea is to build a Markov Chain whose stationary distribution is $p(x)$. The Markov Chain has a unique stationary distribution if it is irreducible and aperiodic. The MC state-transition distribution in this case is $q(x^{l+1}|x^l)$ but there are many sampling procedures in the MCMC framework, defining different state-transition distribution $q(.)$.

2.8 Boltzmann Machines

Boltzmann Machines are an example of **Markov Random Field**, probabilistic graphical model that can learn and represent complex probability distributions, composed of visible RV $v \in \{0,1\}$ and latent RV $h \in \{0,1\}$; $\mathbf{s} = [\mathbf{vh}]$ (binary variables). The behaviour is regulated by a linear energy function.

$$E(\mathbf{s}) = -\frac{1}{2}\mathbf{s^T M s} - \mathbf{b^T s} = -\frac{1}{2}\sum_{ij} M_{ij} s_i s_j - \sum_j b_j s_j \tag{2.29}$$

where $\mathbf{M}$ is the weight matrix and $\mathbf{b}$ is the bias vector, they are the model parameters that encode the interactions between the variables. To obtain distribution we consider $P(\mathbf{s}) = exp(-E(\mathbf{s})/Z)$, where Z is the partition function for normalization.
Boltzmann Machines can be viewed as neural networks of units whose activation is determined by a stochastic function. The output of a neuron is not computed at a given timestep instead is sampled from a given probability distribution. Network activity is a sample from posterior probability given inputs (visible data).



2.8.1 Stochastic Binary Neurons

Spiking point neuron with binary output

$$s_j^{(t)} = \begin{cases} 1, & \text{with probability } p_j^{(t)} \\ 0, & \text{with probability } 1 - p_j^{(t)} \end{cases} \tag{2.30}$$

The local neuron potential x_j is defined as usual $x_j^{(t+1)} = \sum_i M_{ij} s_i^{(t)} + b_j$. A chosen neuron fires with spiking probability $p_j^{(t+1)} = P(s_j^{(t+1)} = \mathbf{s}^t) = \sigma(x_j^{(t+1)})$.
We can update the model state (activation of all neurons) either in parallel (**Parallel Dynamics**) or one neuron at random is chosen for update at each step (**Glauber Dynamics**)

2.8.2 Learning Considering Only Visible Variables

Boltzmann machines can be trained so that the equilibrium distribution (of the machines) tends towards any arbitrary distribution across binary vectors given samples from that distribution. Use probabilistic learning techniques to fit the parameters, i.e. maximizing the loglikelihood.

$$\mathcal{L}(M) = \frac{1}{L} \sum_l \log P(\mathbf{v^l}|\mathbf{M}) \tag{2.31}$$

To obtain distribution $P(\mathbf{v^l}|\mathbf{M}) = \frac{exp(-E(\mathbf{v^l})}{Z}$ where E is the energy function and Z is the partition function.

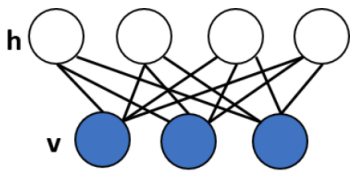
Gradient Approach

$$\frac{\partial \mathcal{L}}{\partial M_{ij}} = \mathbb{E}_{v\sim D}[v_i v_j] - \mathbb{E}_{v\sim model}[v_i v_j] = \langle v_i v_j \rangle_D - \langle v_i v_j \rangle \tag{2.32}$$

with free expectation (*dream*, from the model) $\langle v_i v_j \rangle = \sum_{\mathbf{v}} P(\mathbf{v}) v_i v_j$ and clamped expectation (*wake*, from the data) $\langle v_i v_j \rangle_D = 1/L \sum_l v_i^l v_j^l$.

2.8.3 Restricted Boltzmann Machines

To efficiently capture higher-order correlations we need to introduce hidden RV. Expectations again become intractable due to the partition function Z.
The term "restricted" in RBM refers to the connectivity pattern between its units. Unlike a fully connected Boltzmann machine, where all units are connected to each other, RBMs have a **restricted connectivity pattern**. Specifically, RBMs consist of two layers: a visible layer and a hidden layer. The visible layer represents the input data, while the hidden layer captures higher-level representations or features learned from the data. In an RBM, the connections are only between the visible and hidden units (**bipartite graph**). Learning (and inference) becomes tractable due to graph bipartition which factorizes distribution.



The energy function, highlights bipartition in hidden ($\mathbf{h}$) and visible ($\mathbf{v}$) units.

$$E(\mathbf{v}, \mathbf{h}) = -\mathbf{v^t M h} - \mathbf{b^T v} - \mathbf{c^T h} \tag{2.33}$$

In RBMs hidden units are conditionally independent given visible units, and viceversa.

$$P(h_j|\mathbf{v}) = \sigma\left(\sum_i M_{ij} v_i + c_j\right) \tag{2.34}$$

$$P(v_i|\mathbf{h}) = \sigma\left(\sum_j M_{ij} h_j + b_i\right) \tag{2.35}$$

Training RBM with Gibbs Sampling

A Gibbs sampling approach

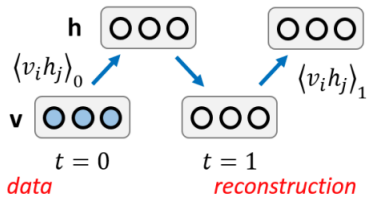
Wake
- Clamp data on v
- Sample $v_i h_j$ for all pairs of connected units
- Repeat for all elements of dataset

Dream
- Don't clamp units
- Let network reach equilibrium
- Sample $v_i h_j$ for all pairs of connected units
- Repeat many times to get a good estimate

Start with a training vector on the visible units, alternate between updating all the hidden units in parallel and updating all the visible units in parallel (iterate). Gibbs sampling can be painfully slow to converge (high variance).

Contrastive-Divergence Learning

A very crude approximation of the gradient of the log-likelihood



1. Clamp a training vector v^l on visible units
2. Update all hidden units in parallel
3. Update all the visible units in parallel to get a *reconstruction*
4. Update the hidden units again

$$\underbrace{\langle v_i h_j \rangle_0}_{data} - \underbrace{\langle v_i h_j \rangle_1}_{reconstruction}$$

RBMs can be very powerful if stacked (deep learning)

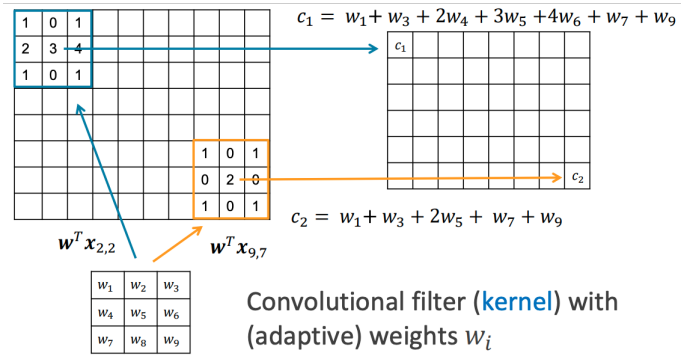
Chapter 3

Deep Learning

3.1 Convolutional Neural Networks

3.1.1 Convolution Concepts

Adaptive Convolution



Stride

Basic convolution slides the filter on the image one pixel at a time (stride = 1). Stride can define of different sizes (hyperparameter) and it can reduce the number of multiplications.
What is the size of the image (DIM) after application of a filter with a given size (K) and stride (S)?
General rule: $DIM' = \frac{DIM-K}{S} + 1$

Zero Padding

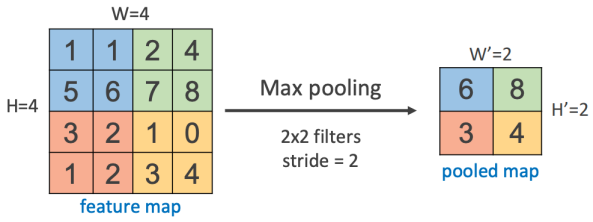
If the image cannot be scanned by the filter (e.g. 3x3 filter with stride 3 on a 7x7 image) we add **columns and rows of zeros** to the border of the image. When performing a convolution operation, the filter is usually smaller than the input data, which results in the output being smaller than the input. By padding the input with zeros, the output size can be adjusted to match the desired dimensions or to ensure compatibility with subsequent layers in a neural network.

Feature Map Transformation

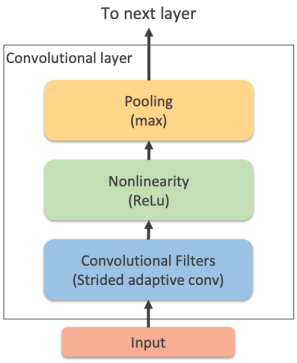
Convolution is a linear operator, apply an element-wise nonlinearity to obtain a transformed feature map.

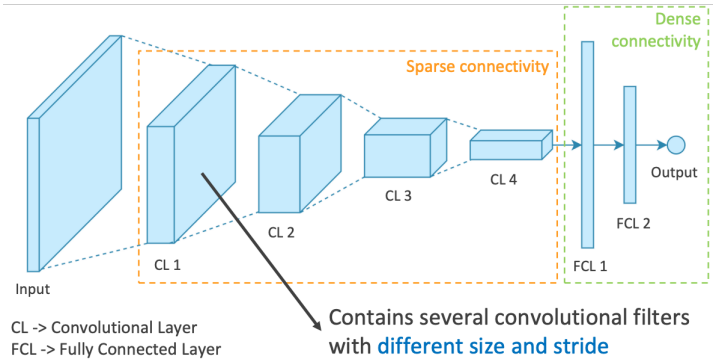
Pooling

Operates on the feature map to make the representation: smaller (subsampling) and robust to (some) transformations.



Convolution Layers



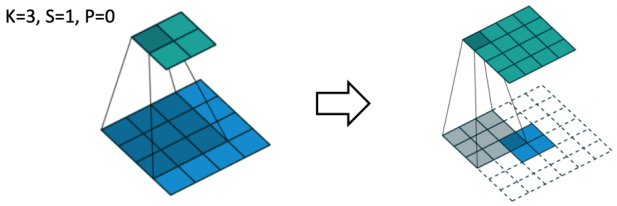


CNN as a Sparse NN

Convolution amounts to sparse connectivity (reduce parameters) with parameter sharing (enforces invariance)

3.1.2 CNN Training

Variants of the standard backpropagation that account for the fact that connections share weights (convolution parameters). The gradient is obtained by summing the contributions from all connections sharing the weight. Backpropagating gradients from convolutional layer N to N-1 is not as simple as transposing the weight matrix (need deconvolution with zero padding). For example input is a 4x4 image, output is a 2x2 image, backpropagation step requires going back from the 2x2 to the 4x4 representation. We can obtain the transposed convolution using the same logic of the forward convolution



If you had no padding in the forward convolution, you need to pad much when performing transposed convolution. If you have striding, you need to fill in the convolution map with zeroes to obtain a correctly sized deconvolution.

3.1.3 Convolutions Advanced Concepts

ReLU Nonlinearity

ReLu help counteract gradient vanish. Sigmod first derivative vanishes as we increase or decrease z. ReLu first derivative is 1 when unit is active and 0 elsewhere. ReLu second derivative is 0 (no second order effects). Easy to compute (zero thresholding). Favors sparsity.

1x1 Convolutions

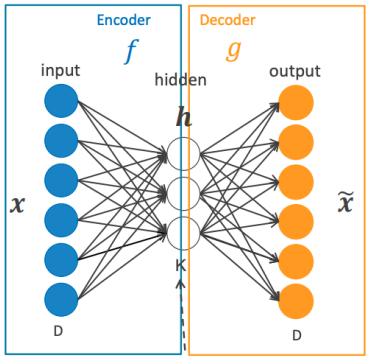
They are helpful because placing 1x1 convolutions before larger kernels reduces the number of input channels, saving computations and parameters.

Batch Normalization

It is a technique used to normalize the inputs of a layer by adjusting and scaling the activations. It aims to improve the stability and speed of training by reducing the internal covariate shift and providing a smoother optimization landscape. The internal covariate shift refers to the change in the distribution of the input values for different layers training with mini-batches. This can make it challenging for the network to learn effectively since the distribution of the input can vary as the parameters of the previous layer change. Batch normalization helps alleviate this problem by normalizing the inputs to each layer, thus reducing the impact of the internal covariate shift. The steps are: compute the mean and variance of the mini-batch, normalize the mini-batch by subtracting the mean and dividing by the standard deviation, scale and shift the normalized values using learned parameters (gamma and beta) to allow the network to learn the optimal scale and shift for each feature.

3.2 Autoencoders

3.2.1 Basic Autoencoder



Train a model to reconstruct the input by loss minimization $L(\mathbf{x}, \tilde{\mathbf{x}}) = L(\mathbf{x}, g(f(\mathbf{x})))$. Since $K << D$ there's some of information bottleneck.

Generally, we would like to train nonlinear AEs, with possibly $K > D$, that do not learn trivial identity.

3.2.2 Regularized Autoencoders

Sparse Autoencoders

In a sparse autoencoder, an additional sparsity constraint is introduced to encourage the learned representations to be sparse, meaning that only a few neurons are activated or have non-zero values for a given input. The sparsity constraint is typically enforced by adding a regularization term to the loss function during training. This can lead to more meaningful and efficient representations, especially in high-dimensional data where many of the input features may be irrelevant or redundant.

Denoising Autoencoder

A denoising autoencoder learns robust representations of data by removing noise or corruption from the input. The primary goal of a denoising autoencoder is to learn a mapping from a corrupted or noisy input to a clean output, while also learning a compressed representation of the input data. The denoising autoencoder achieves this by introducing a deliberate corruption process during training, where the input data is intentionally distorted or perturbed $C(\hat{\mathbf{x}}|\mathbf{x})$. The key intuition is that we want to learn a representations that is robust to partial destruction of the input.
In **manifold learning**, the assumption is that high-dimensional data often lie on or near a lower-dimensional manifold, which can be thought of as a curved surface embedded in the high-dimensional space. The goal is to learn a mapping or representation that captures the essential structure of this underlying manifold. Denoising autoencoders can be seen as a way to learn this underlying manifold

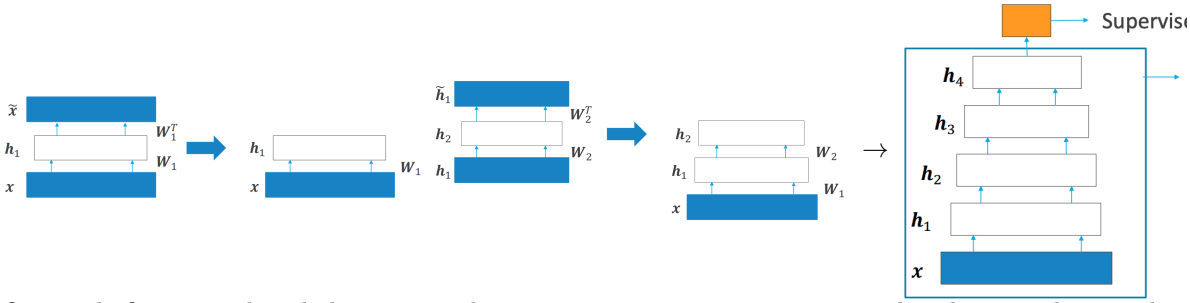
by introducing a corruption process and training the autoencoder to reconstruct clean versions of the input.

Contractive Autoencoder

The key idea behind a contractive autoencoder is to encourage the learned representation to be insensitive to small changes in the input space. This is achieved by adding a contractive penalty term to the loss function during training, which penalizes the sensitivity of the learned representation to small input perturbations. Penalty term: $\Omega(\mathbf{h}) = \omega(f(\mathbf{x})) = ||\frac{\partial f(\mathbf{x})}{\mathbf{x}}||_F$

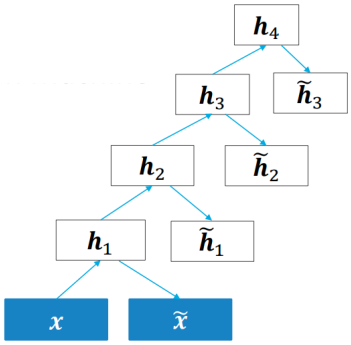
3.2.3 Deep Autoencoder

Backpropagating thru many layers can be difficult. We use **Unsupervised Layerwise Pretraining** incrementally to construct Deep AEs.



Optional: fine tune the whole autoencoder to optimize input reconstruction by plugging the encoder part to a decoder (transposing the weights). You can use backpropagation, but it remains an unsupervised task.

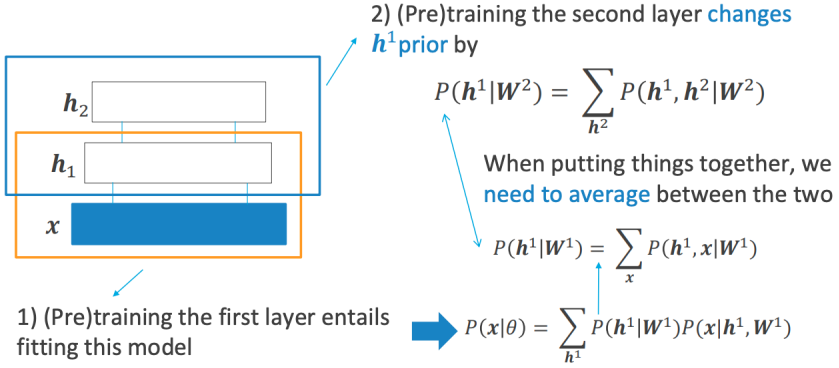
3.2.4 Deep Boltzmann Machines



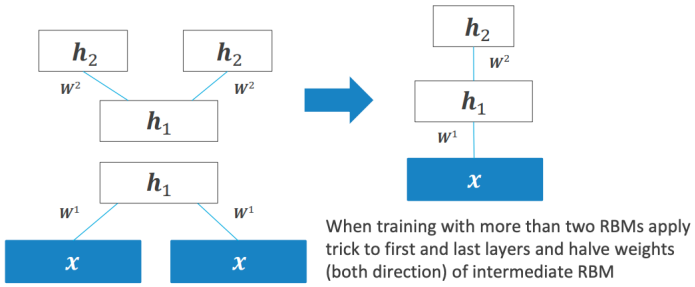
This structure that represents the layerwise pretraining can be seen as a layered Restricted Boltzmann Machine. We can use what we know from RBM to perform layerwise pretraining and learn the matrices $\mathbf{W}_i$

Pretraining

Training requires some attention because of the interactions from higher layers to the bottom, but also the interactions from the bottom to the top (undirected graph).



Trick: averaging the two models of $\mathbf{h}^1$ can be approximated by taking half contribution from $\mathbf{W}^1$ and half from $\mathbf{W}^2$.

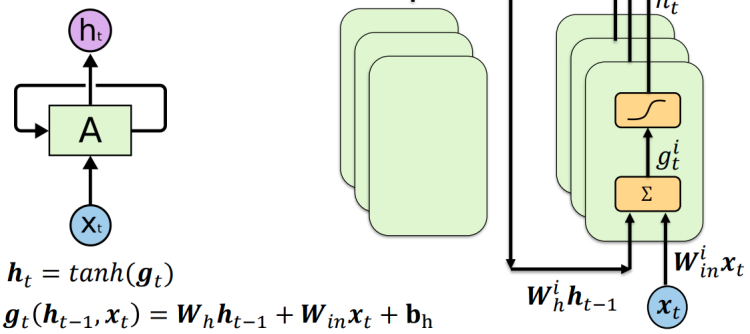


The pretrained DBM matrices can be used to initialize a deep autoencoder. Add output layer and fine tune the RBM matrices by backpropagation.

3.3 Gated Recurrent Neural Netoworks

3.3.1 Vanilla RNN

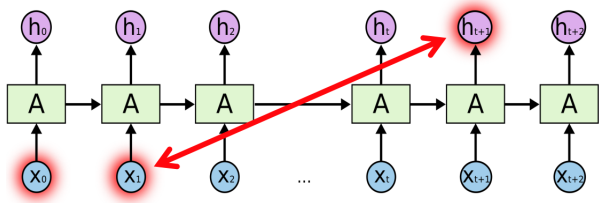
$$\boldsymbol{y}_t = f(\boldsymbol{W}_{out}\boldsymbol{h}_t + \boldsymbol{b}_{out})$$



$$\boldsymbol{h}_t = tanh(\boldsymbol{g}_t)$$

$$\boldsymbol{g}_t(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t) = \boldsymbol{W}_h\boldsymbol{h}_{t-1} + \boldsymbol{W}_{in}\boldsymbol{x}_t + \mathbf{b}_\mathrm{h}$$

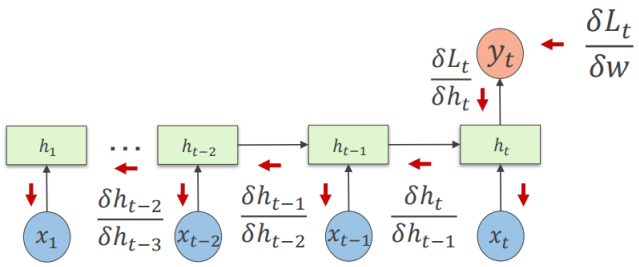
Hidden state $\mathbf{h}_t$ summarizes information on the history of the input signal up to time t. When the time gap between the observation and the state grows there is little residual information of the input inside of the memory. They are hard to train due to gradient propagation issues. They are relatively fast at inference time. However, the recurrence limits the parallelization opportunities.



3.3.2 Exploding/Vanishing Gradient

Gradients propagated over many stages tend to vanish (no learning) or explode (instability and oscillations). Weights are shared between time steps so we have to sum gradient contributions through time.

$$\frac{\partial L_t}{\partial \mathbf{W}} = \sum_{k=1}^{t} \frac{\partial L_t}{\partial h_t} \left(\prod_{l=k}^{t-1} \frac{\partial h_{l+1}}{\partial h_l} \right) \frac{\partial h_k}{\partial \mathbf{W}} \tag{3.1}$$



The gradient is a recursive product of hidden activation gradients (Jacobian).

Bounding the Gradient

$$\left\|\frac{\delta L_t}{\delta \boldsymbol{h}_k}\right\| = \left\|\frac{\delta L_t}{\delta \boldsymbol{h}_t}\prod_{l=k}^{t-1}\boldsymbol{D}_{l+1}\boldsymbol{W}_{hl}^T\right\| \leq \left\|\frac{\delta L_t}{\delta \boldsymbol{h}_t}\right\|\prod_{l=k}^{t-1}\|\boldsymbol{D}_{l+1}\boldsymbol{W}_{hl}^T\| = \left\|\frac{\delta L_t}{\delta \boldsymbol{h}_t}\right\|\prod_{l=k}^{t-1}\sigma(\boldsymbol{D}_{l+1})\sigma(\boldsymbol{W}_{hl}^T)$$

The gradient is bounded by the spectral radius σ, it can shrink to zero ($\sigma < 1$) or increase exponentially ($\sigma > 1$) depending on the spectral properties.

Gradient Clipping

Take $g = \frac{\partial L_t}{\partial \mathbf{W}}$, if $||g|| > \theta_0$ then $g = \frac{\theta_0}{||g||}g$. Rescaling works for exploding gradient but does not work for gradient vanish as total gradient is a sum of time dependent gradients (preserving relative contribution from each time makes it exponentially decay).

Error Propagation

Solution seems to be having the Jacobian with $\sigma = 0$. The **activations functions** sigmoid and tanh have the $\sigma < 1$ so they don't work, would be better to use modReLU or no activation function (identity). For the recurrent weights matrix it's possible to achieve $\sigma = 1$ with orthogonal matrices ($\mathbf{W^t}W = I$), unitary matrices and identity matrix ($\mathbf{W} = \mathbf{I}$).
So **orthogonal/identity matrix + linear activation** would solve the problem. Has the desired spectral properties but does not work in practice as it quickly saturates memory (e.g. with replicated/non-useful inputs and states). We want to be able to "control the forgetting".

3.3.3 Long-Short Term Memory Cell

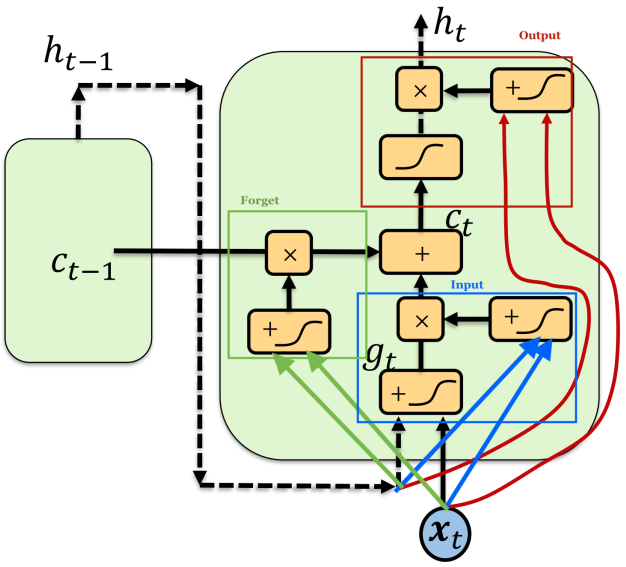
Forget Gate

Using identity activation function and identity weight matrix $h_t = h_{t-1} + c(\mathbf{t})$ is a mess because it accumulates input information and stores in $\mathbf{h} \Rightarrow$ no forgetting and hidden state saturation. A forget gate (a filter) to "soft reset" the units would help the saturation of the states.

$$\mathbf{f}_t = \sigma(\mathbf{W_{fh}h_{t-1}} + \mathbf{W_{fx}x_t} + \mathbf{b_f}) \text{ and use it in } \mathbf{h_t} = \mathbf{f_t} \odot \mathbf{h_{t-1}} + c(\mathbf{x_t}) \tag{3.2}$$

In this way it adaptively forgets the past, avoid saturation but it has no guarantess about constant propagation.

Design



Input gate: controls how inputs contribute to the internal state.
Forget gate: controls how past internal state c_{t-1} contributes to c_t.
Output gate: controls what part of the internal state is propagated out of the cell.

1) Compute activation of input and forget gates

$$\boldsymbol{I}_t = \sigma(\boldsymbol{W}_{Ih}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{Iin}\boldsymbol{x}_t + \mathbf{b}_\mathrm{I})$$
$$\boldsymbol{F}_t = \sigma(\boldsymbol{W}_{Fh}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{Fin}\boldsymbol{x}_t + \mathbf{b}_\mathrm{F})$$

2) Compute input potential and internal state

$$\boldsymbol{g}_t = tanh(\boldsymbol{W}_{h}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{in}\boldsymbol{x}_t + \mathbf{b}_\mathrm{h})$$
$$\boldsymbol{c}_t = \boldsymbol{F}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{I}_t \odot \boldsymbol{g}_t$$

3) Compute output gate and output state

$$\boldsymbol{O}_t = \sigma(\boldsymbol{W}_{Oh}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{Oin}\boldsymbol{x}_t + \mathbf{b}_\mathrm{O})$$
$$\boldsymbol{h}_t = \boldsymbol{O}_t \odot tanh(\boldsymbol{c}_t)$$

All current LSTM implementation use full Back-Propagation Thru Time training.

Regularizing LSTM

Via **Dropout**: randomly disconnect units from the network during training.

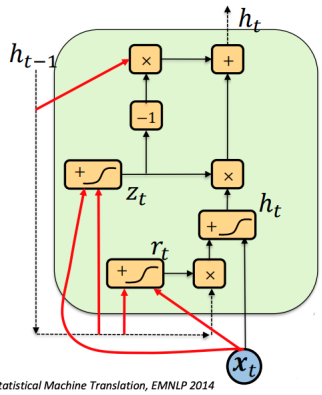
3.3.4 Gated Recurrent Unit

Reset acts directly on output state (no internal state and no output gate)

$$\boldsymbol{h}_t = (1 - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} + \boldsymbol{z}_t \odot \boldsymbol{h}_t$$
$$\boldsymbol{h}_t = tanh(\boldsymbol{W}_{hh}(\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}) + \boldsymbol{W}_{hin}\boldsymbol{x}_t + \mathbf{b}_\mathrm{h})$$

Reset and update gates when coupled act as input and forget gates

$$\boldsymbol{z}_t = \sigma(\boldsymbol{W}_{zh}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{zin}\boldsymbol{x}_t + \mathbf{b}_\mathrm{z})$$
$$\boldsymbol{r}_t = \sigma(\boldsymbol{W}_{rh}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{rin}\boldsymbol{x}_t + \mathbf{b}_\mathrm{r})$$



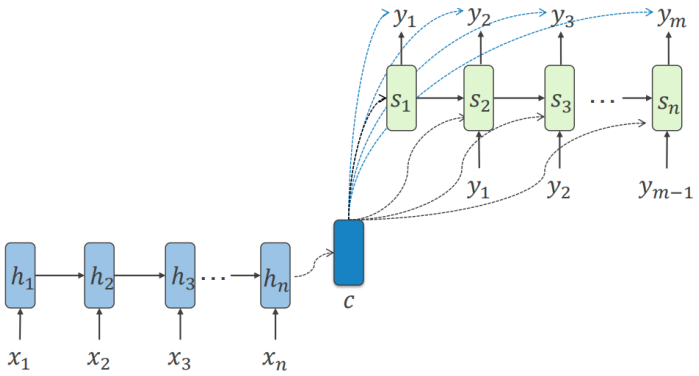
C. Kyunghyun et al, Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation, EMNLP 2014

3.4 Sequence-to-sequence and attention

3.4.1 Sequence Transduction

Input and output are both sequences and they may have different lengths (e.g. machine translation). The idea of an unfolded RNN with blank inputs-outputs does not really work well.

Encoder-Decoder



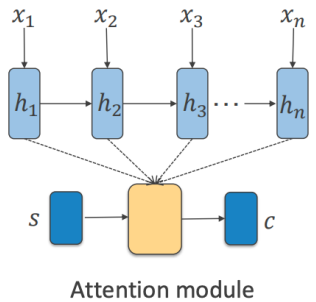
Encoder: produces a compressed and fixed length representation c of all the input sequence $x_1, ..., x_n$ (originally c = h_n).
Decoder: receives the context vector generated by the encoder and uses it to generate the output sequence.
Encoder-Decoder can share parameters (but it is uncommon). Encoder-Decoder can be trained end-to-end or independently.

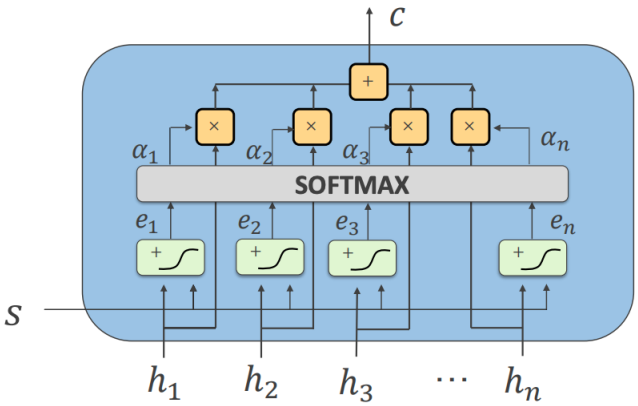
3.4.2 Attention

On the Need of Paying Attention

Encoder-Decoder scheme assumes that the hidden activation of the last input element summarizes sufficient information to generate the output (bias toward most recent past). Other parts of the input sequence might be very informative for the task, possibly elements appearing very far from sequence end. Attention mechanisms select which part of the sequence to focus on to obtain a good c.

Attention Module





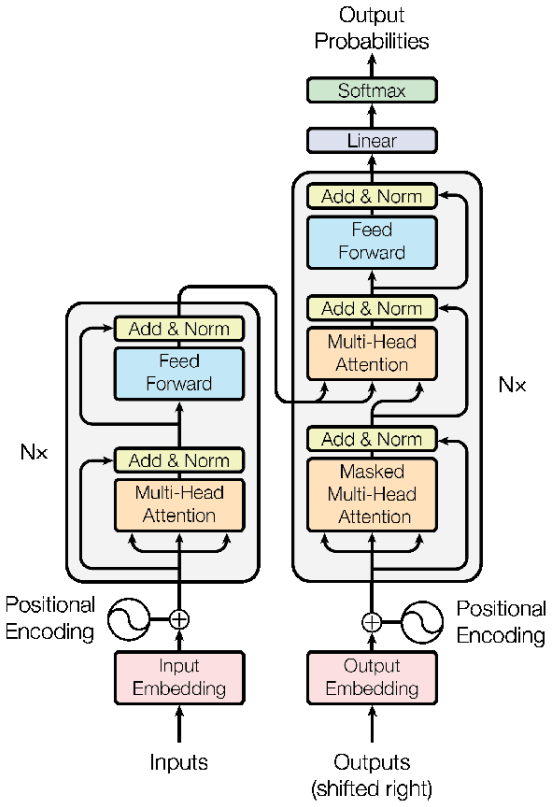
Relevance: tanh layer fusing each encoding with current context s, $e_i = a(s, h_i)$.
Softmax: a differentiable max selector operator, $\alpha_i = \frac{exp(e_i)}{\sum_j exp(e_j)}$.
Voting: aggregated seed by (soft) attention voting, $c = \sum_i \alpha_i h_i$.

Advanced Attention

Generalize Relevance: this component determines how much each h is correlated/associated with current context s.
Hard Attention: sample a single encoding using probability α_i.

3.4.3 Transformers

First pure attention-based model with self-attention and no recurrence. Encoder-decoder architecture: 6 layer encoder, 6 layer decoder.



Self Attention

Allows the model to capture dependencies between different positions in a sequence by assigning weights to the elements in the sequence based on their relevance to each other. Each element of an input sequence X_i projects into 3 vectors: **query, key** and **value**. These transformations are linear projections of the original input sequence and are learned during the training process. For each element compute attention over all other vectors.

$$SA(Q_i, \mathbf{K}, \mathbf{V}) = \sum_j softmax\left(\frac{Q_i \mathbf{K}^T}{\sqrt{d_k}}\right) V_j \tag{3.3}$$

By applying self-attention across all positions in the sequence, the Transformer model can capture long-range dependencies and relationships without relying on recurrent connections. Each position in the sequence attends to all other positions, allowing for parallel processing and efficient modeling of global dependencies.
In the original self-attention mechanism, a single set of queries, keys, and values is used to compute attention weights and perform the weighted sum. In **multi-head self-attention**, the self-attention operation is applied multiple times in parallel, with different learned linear projections for each attention "head". It enhances the modeling capability by allowing the model to attend to different aspects

capturing diverse dependencies and patterns.
Self-attention is orderindependent but in sequences we need ordering information. The purpose of **positional encoding** is to inject information about the position of each word in the input sequence into the model's embeddings. By incorporating positional information, the model can differentiate between words based on their position, which is essential for capturing the order of words in a sequence.

3.5 Hierarchical and Multiscale Recurrent Neural Networks

Gated RNN claim to solve the problem of learning long-range dependencies but, in practice, it is still difficult to learn on longer range. Exist different architectures trying to optimize dynamic memory usage. Skipping state updates mitigates vanishing gradients but how can we decide how to skip updates? To address the same problem we could use convolution instead of recurrence. This gives us better parallelization on GPUs and maller distance between long-range dependencies.

3.5.1 Hierarchical RNNs

A Hierarchical Recurrent Neural Network (HRNN) is a type of neural network architecture that combines the concepts of recurrent neural networks (RNNs) and hierarchical modeling. Many sequences have a latent hierchical structure, we want to model it.
Add skip connections to the model (static skip). Learn when to skip updates (adaptive skip). Skip units, blocks of units, or entire layer.
Zoneout at each time step decide which units to not update.

Clockwork RNNs

Introduces a hierarchical time structure to capture dependencies at different timescales. The key idea behind the CW-RNN is to divide the recurrent connections into modules or "clockwork" units, each operating at a different timescale. Each module is responsible for processing a subset of the hidden units in the RNN, and the activation of the modules is determined by clock signals.

Skip RNNs

Incorporate adaptive gates and learn to skip entire updates. Also saves computation.

Hierarchical Multiscale RNN

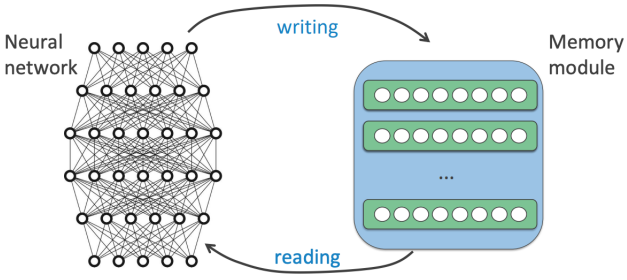
The HM-RNN architecture consists of multiple recurrent layers, each operating at a different timescale or resolution. Each layer models more abstract features by learning the boundaries (adaptive). This hierarchical structure allows the model to capture both local and global dependencies in the data. It has 3 operations: update, state update (LSTM cells) according to boundary detector; copy, copies cell and hidden states from the previous timestep to the current timestep; flush, sends summary to next layer and reinitializes current layer's state.

3.6 Neural Reasoning

Neural reasoning refers to the ability of neural networks to perform reasoning tasks, which involve understanding and manipulating information to arrive at logical conclusions or make decisions. View recurrent (sequential) models as general programs.

3.6.1 Memory Networks

In order to solve the task need to memorize: facts, question and answers. A bit too much for the dynamical RNN memory but we can try to address it through an external memory.



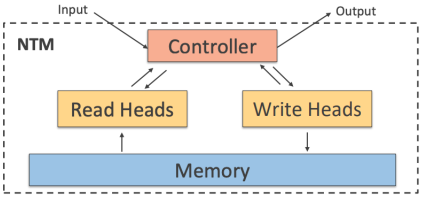
Components

The **components** of a memory network are the following:

- (I) Input feature map: Encodes the input in a feature vector
- (G) Generalization: decide what input (or function of it) to write to memory
- (O) Output feature map: reads the relevant memory slots
- (R) Response: returns the prediction given the retrieved memories

3.6.2 Neural Turing Machines

The key idea behind Neural Turing Machines is to incorporate an external memory component, similar to a computer's random-access memory (RAM), into a neural network architecture. This external memory allows the model to store and access information dynamically, enabling it to perform tasks that require memory, attention, and sequential processing.



Neural Controller

The controller is a recurrent neural network (RNN) or a similar architecture that processes input data and generates output. It interacts with the external memory by reading from and writing to it. The controller can perform computations, make decisions, and learn to access and manipulate the memory.

Operations

The controller can read from and write to the external memory based on the addressing mechanisms. During the read operation, the controller retrieves information from specific memory locations based on the addressing mechanism and uses it for computation. Retrieved memory is a weighted (with the attention distribution) mean of all memories. During the write operation, the controller writes new or modified information to specific memory locations. Write operation is actually performed by composing erasing and adding operations considering an attention distribution describing how much we change each memory.

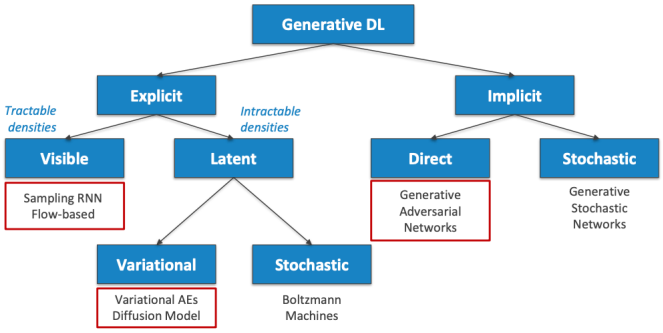
3.7 Explicit Density Learning

3.7.1 Deep Learning Overview

Labelled data is costly and difficult to obtain. Unsupervised is a sustainable future for deep learning: learning the latent structure of data, discover important features, learn task independent representations, introduce (if any) supervision only on few samples.

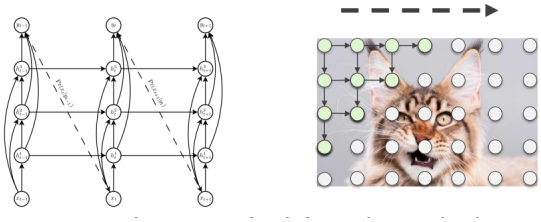
Focusing too much on discrimination rather than on characterizing data can cause issues. Generative models (try to) characterize data distribution. Given training data, learn a (deep) neural network that can generate new samples from (an approximation of) the data distribution.
Two approaches:

- Explicit: learn a model denisity $P_\theta(x)$
- Implict: learn a process that samples data from $P_\theta \approx P(x)$



Learning with Fully Visible Information

If all information is fully visible the joint distribution can be computed from the chain rule factorization $P(\mathbf{x}) = \prod_i P(x_i|x_1, ..., x_{i-1})$. For example we can represent the probability of a pixel having a certain intensity value, given the known intensity of its predecessor. So we need to be able to define a sensible ordering for the chain rule.



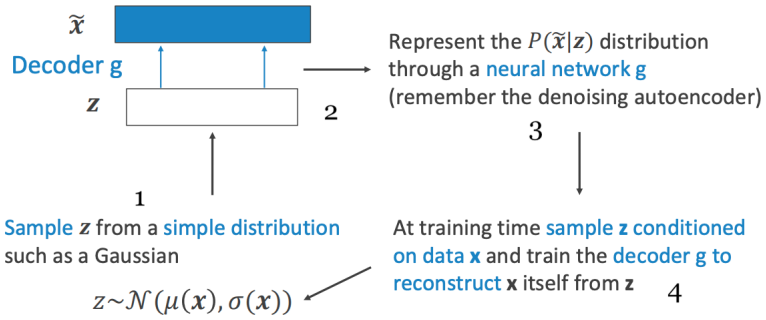
Scan the image according to a schedule and encode the dependency from previous pixels in the states of an RNN

Latent Information

Not always all the data is visible, so we introduce a latent process regulated by unobservable variables $\mathbf{z}$: $P_\theta(\mathbf{x}) = \int P_\theta(\mathbf{x}|\mathbf{z})P_\theta(\mathbf{z})d\mathbf{z}$ (calculated by marzinalization), which is typically intractable.

3.7.2 Variational Autoencoders

Since introducing latent variable makes the process intractable we want to approximate $\mathbb{E}_z[P(\mathbf{x}|\mathbf{z})]$ by sampling. We assume to be able to generate the reconstruction from a sampled latent representation $\tilde{\mathbf{x}} \sim P(\mathbf{x}|z)$ where $\mathbf{z} \sim P(\mathbf{z})$. Of course we don't have access to the true distributions, so how do we approximate them?



Sampling it is NOT backpropagatable.

Training

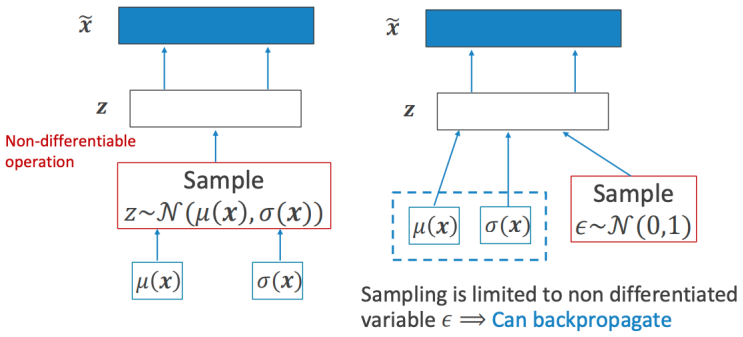
It's not that easy. Ideally we want to train maximizing.

$$L(D) = \prod_{i=1}^{N} P(\boldsymbol{x_i}) = \prod_{i=1}^{N} \int P(\boldsymbol{x_i}|\boldsymbol{z})P(\boldsymbol{z})d\boldsymbol{z}$$

Intractable → Variational approximation

Non differentiable → Reparameterization

Reparameterization



μ and σ are generated by deterministic rule (by a neural network - encoder part) and the noise is drawn from a normal distribution $\epsilon \sim \mathcal{N}(0,1)$ in order to make the "stochastic aspect" differentiable. **Variational Approximation**: maximizing the ELBO allows approximating from below the intractable log-likelihood $\log P(\mathbf{x})$

$$\mathcal{L}(x,\theta,\phi) = \mathbb{E}_Q[\log P(x|z)] + \mathbb{E}_Q\left[\log P(z)\right] - \mathbb{E}_Q[\log Q(z)]$$

Decoder estimate of the conditional, made possible and differentiable through the reparameterization trick

$KL(Q(z|\phi)||P(z|\theta))$

Need a Q(z) function to approximate P(z)

Training is performed by backpropagation on θ, ϕ to optimize the ELBO.

reconstruction

$$\mathcal{L}(x,\theta,\phi) = \mathbb{E}_Q\left[\log P(x|z = \mu(x) + \sigma^{1/2}(x) * \epsilon, \theta)\right]$$
$$-KL(Q(z|x,\phi)||P(z|\theta))$$ regularization

Can be computed in closed form when both Q(z) and P(z) are Gaussians

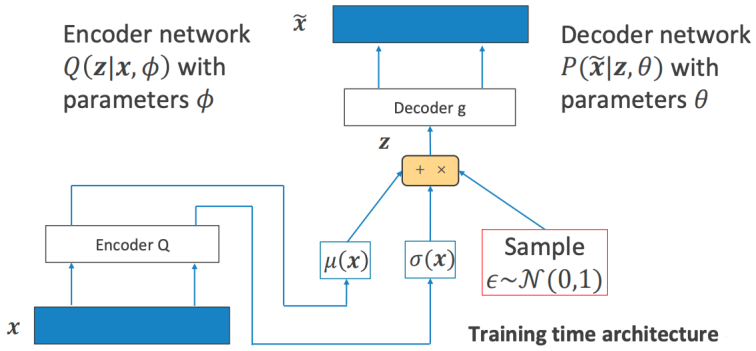
$KL(\mathcal{N}(\mu(x), \sigma(x)) \,||\mathcal{N}(0,1))$

Train the encoder to behave like a Gaussian prior with zero-mean and unit-variance

We would like to optimize the following loss by SGD:

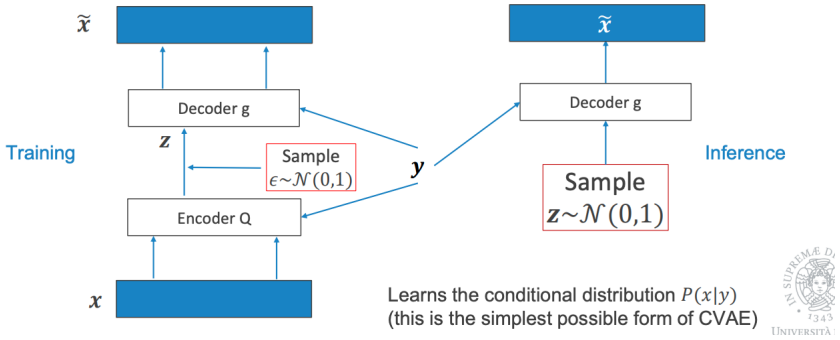
$$\mathbb{E}_{X\sim D}[\mathbb{E}_{\epsilon\sim\mathcal{N}(0,1)}\left[\log P(x|z = \mu(x) + \sigma^{1/2}(x) * \epsilon, \theta)\right] - KL(Q(z|x,\phi)||P(z))]$$

The Full Picture



Sampling (Testing): at test time detach the encoder, sample a random encoding and generate the sample as the corresponding reconstruction.

Conditional Generation



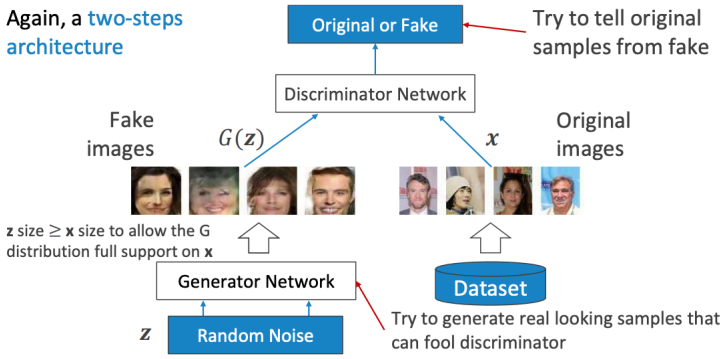
3.8 Generative Adversarial Networks

3.8.1 The Problem

We want to characterize the data (distribution, variances) to allow generating new observation. And we are talking about an **implicit** approach: learn a process that samples data from $P_\theta(x) \approx P(x)$. So we want to learn to sample (**generate samples**), not like in variational AEs that learn to approximate an intractable distribution.

3.8.2 Overview

The catch: sample from a simple distribution (random noise) train a differentiable function (neural network) to transform random noise to the training distribution.



3.8.3 Alternate Optimization

Discriminator output is likelihood of input being real. Discriminator tries to maximize C s.t: $D_{\theta_D}(x) \rightarrow 1$ and $D_{\theta_D}(G_{\theta_G}(z)) \rightarrow 0$ (D correctly distinguish fake from real). Generator tries to minimize C s.t. $D_{\theta_D}(G_{\theta_G}(z)) \rightarrow 1$ (fake the D).

$$C = \min_{\theta_G} \max_{\theta_D} \left[\mathbb{E}_x \left[\log D_{\theta_D}(x) \right] - \mathbb{E}_z \left[\log(1 - D_{\theta_D}(G_{\theta_G}(z))) \right] \right]$$

1. Discriminator gradient ascent

$$C_D = \max_{\theta_D} \left[\mathbb{E}_x \left[\log D_{\theta_D}(x) \right] - \mathbb{E}_z \left[\log(1 - D_{\theta_D}(G_{\theta_G}(z))) \right] \right]$$

2. Generator gradient descent

$$C_G = \min_{\theta_G} \left[\mathbb{E}_z \left[\log(1 - D_{\theta_D}(G_{\theta_G}(z))) \right] \right]$$

Optimizing this doesn't really work

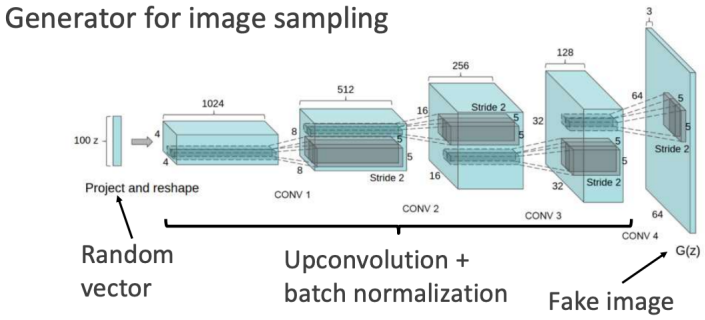
Issues

The cost that the Generator receives in response to generate $G(z)$ depends only on the Discriminator response. At the beginning the D learns quicker than G resulting in a flat gradient (because initially generated data is clearly fake). The optimal solution of the min-max problem is a saddle point. It has little stability, initially lot of heuristic work, now converged to more principled solutions.

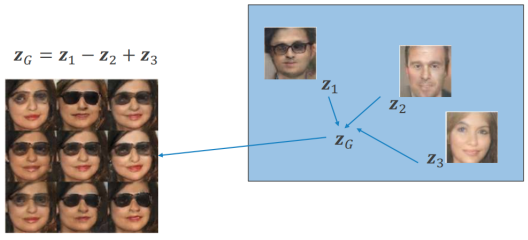
Wasserstein Distance Models

Attempts to solve the hardness of training adversarial generators by optimizing the Wasserstein distance (EMD) between the generator and empirical distribution filtered trough the discriminator function D. It quantifies the minimum amount of "work" or effort required to transform one distribution into another. WGAN provide gradients across all the range of training conditions

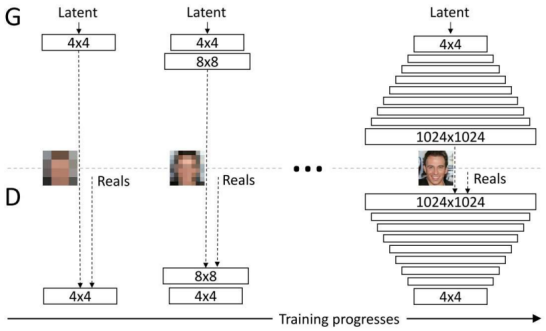
3.8.4 Deep Convolutional GANs



Latent Space Arithmetic



Progressive GAN

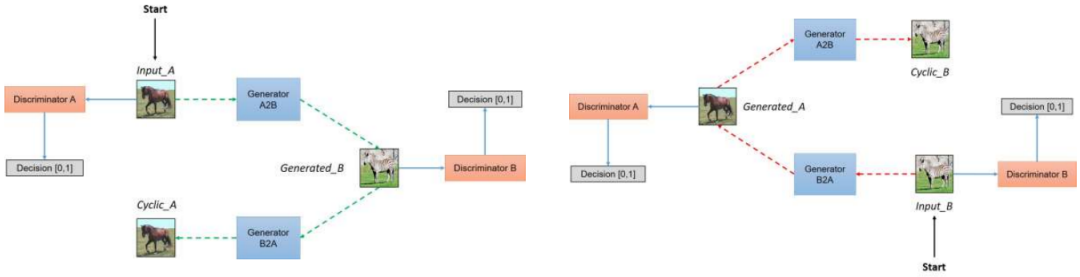


Conditional Generation

Learn a mapping from an observed side information x and a random noise vector z to the fooling samples y, $G : \{x, z\} \rightarrow y$.

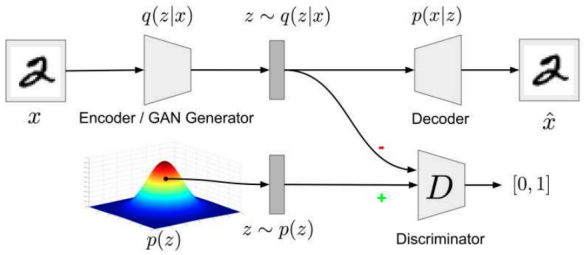
CycleGAN

Unsupervised style transfer. Enforce alignment by ensuring that generated images in domain B can lead to good fakes in domain A and vice versa.



3.8.5 Adversarial Autoencoders

It incorporates elements from both generative adversarial networks (GANs) and autoencoders to learn a latent space representation of the input data that captures meaningful features and allows for generation of new samples.



Training

$$\mathcal{L}(x) = \mathbb{E}_Q[\log P(x|z)] - KL(Q(z|x)||P(z))$$

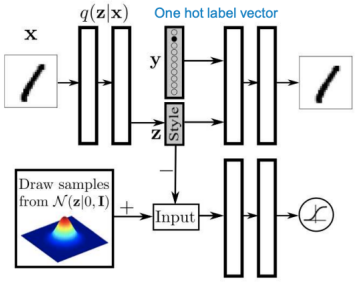
Replaced by an adversarial loss

Reconstruction phase: update the encoder and decoder to minimize reconstruction error.
Regularization phase: update discriminator to distinguish true prior samples from generated samples; update generator to fool the discriminator.

Adversarial regularization allows to impose priors for which we cannot compute the KL divergence. AAE yields a smoother coverage of the latent space, they are useful to impose "complex" or empirical priors.

AAE Style Transfer

Incorporate label information explicitly to force z to capture class-independent information (e.g. style).

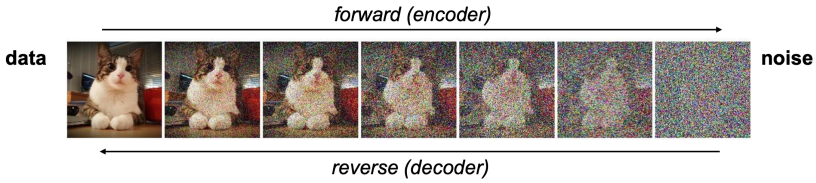


3.9 Diffusion Models

A diffusion model refers to a specific type of generative model that learns the underlying probability distribution of the data. The essential idea is to systematically and slowly destroy structure in a data distribution through an iterative forward diffusion process. We then learn a reverse diffusion process that restores structure in data yielding a highly flexible and tractable generative model of the data.

3.9.1 Learning to Generate with Noise

Two process: forward diffusion that gradually adds noise to input, reverse process reconstructs data from noise (generation). The key is how to do this efficiently.



Forward Diffusion

A fixed (i.e. non-adaptive) noise process in T steps mapping original data x into a same-sized latent variables z_t using simple additive noise.

$$\mathbf{z}_1 = \sqrt{1-\beta_1}\mathbf{x} - \sqrt{\beta_1}\epsilon_1 \qquad (3.4) \qquad\qquad \mathbf{z}_t = \sqrt{1-\beta_t}\mathbf{z}_{t-1} - \sqrt{\beta_t}\epsilon_t \qquad (3.5)$$

where $\epsilon_t \sim \mathcal{N}(0,1)$ and $\beta \in [0,1]$ is the noise schedule.

Diffusion Kernel

Generating $\mathbf{z}_t$ sequentially is time-consuming so we use a closed form solution for $q(\mathbf{z}_t|.)$.

$$q(\boldsymbol{z}_t|\boldsymbol{x}) = \mathcal{N}(\sqrt{\alpha_t}\boldsymbol{x}, (1-\alpha_t)\boldsymbol{I}) \quad \text{(diffusion kernel)}$$

$$\alpha_t = \prod_{s=1}^{t}(1-\beta_s)$$

Which allows writing the marginal as

$$q(\boldsymbol{z}_t) = \int q(\boldsymbol{z}_t, \boldsymbol{x})d\boldsymbol{x} = \int q(\boldsymbol{x})q(\boldsymbol{z}_t|\boldsymbol{x})d\boldsymbol{x}$$

data distribution

3.9.2 Denoising

Sample $\mathbf{z}_T \sim \mathbf{N}(0, 1)$, iterate $\mathbf{z}_{t-1} \sim q(\mathbf{z_{t-1}}|\mathbf{z_t})$. The true denoising distribution is intractable. We cannot de-mix noise if we don't know the starting point x. If we do, then we can show that $q(\mathbf{z}_{t-1}|\mathbf{z_t}, \mathbf{x})$ is Normal.

Reverse process learns an approximated denoising distribution (decoder). Assuming reverse distributions are approximately Normal (reasonable if β_t are small and T large).

$$P(\mathbf{z}_T) = \mathcal{N}(0, I) \tag{3.6}$$

$$P_\theta(\mathbf{z_{t-1}}|\mathbf{z_t}) = \mathcal{N}(\mu_\theta(\mathbf{z_t}, t), \sigma_t^2\mathbf{I}) \tag{3.7}$$

3.9.3 Training

Training follows the classical log-likelihood maximization view which is s, of course, intractable. ELBO at the rescue.

$$\underbrace{\mathbb{E}_{q(\mathbf{z}_1|\mathbf{x})}[\log P_\theta(\mathbf{x}|\mathbf{z}_1)]}_{\text{Reconstruction term}} - \sum_{t=2}^{T} \underbrace{KL\big(P_\theta(\mathbf{z}_{t-1}|\mathbf{z}_t)||q(\mathbf{z}_{t-1}|\mathbf{z}_t, \mathbf{x})\big)}_{\text{Aligns predicted and original (input-conditional) denoising densities}}$$

Distributions in KL are all Gaussians so can write the full form of the loss.

$$\sum_{\mathbf{x}}\Bigg(\underbrace{(-\log \mathcal{N}(\mu_\theta(\mathbf{z}_1, t), \sigma_1^2\boldsymbol{I}))}_{\text{Reconstruction}} + \sum_{t=2}^{T} \frac{1}{2\sigma_t^2} \Bigg\| \underbrace{\left(\frac{(1-\alpha_{t-1})}{(1-\alpha_t)}\sqrt{1-\beta_t}\mathbf{z}_t + \frac{\sqrt{\alpha_{t-1}}\beta_t}{(1-\alpha_t)}\boldsymbol{x}\right)}_{\text{Target mean of } q(\mathbf{z}_{t-1}|\mathbf{z}_t, \boldsymbol{x})} - \underbrace{\mu_\theta(\mathbf{z}_t, t), \sigma_t^2\boldsymbol{I})}_{\text{predicted } \mathbf{z}_{t-1}}\Bigg\|^2\Bigg)$$

Loss can be heavily simplified by reparameterizing so that the model predicts the noise $\epsilon(\cdot)$ that was mixed with the original data, rewriting $\mathbf{x}$ as:

$$\boldsymbol{x} = \frac{1}{\sqrt{\alpha_t}}\mathbf{z}_t - \frac{\sqrt{1-\alpha_t}}{\sqrt{\alpha_t}}\boldsymbol{\epsilon}$$

A network which predicts the unit noise given current noised input z_t

Inserting $\boldsymbol{x}$ above into the ELBO yields (after a while)

$$Loss(\theta) = \sum_{\boldsymbol{x}}\sum_{t=1}^{T}\Big\|\epsilon_\theta\big(\underbrace{\sqrt{\alpha_t}\boldsymbol{x} + \sqrt{1-\alpha_t}\boldsymbol{\epsilon}_t}_{\mathbf{z}_t}, t\big) - \boldsymbol{\epsilon}_t\Big\|^2$$

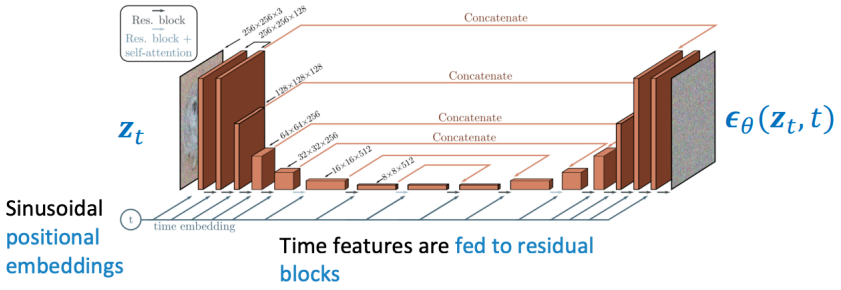
During forward we add noise to image. During reverse we predict that noise with a NN and then subtract it from the image to denoise it.

Algorithm

Algorithm 18.1: Diffusion model training

Input: Training data $\mathbf{x}$
Output: Model parameters θ
repeat
  for $i \in \mathcal{B}$ **do** // For every training example index in batch
    $t \sim$ Uniform$[1, \ldots T]$ // Sample random timestep
    $\boldsymbol{\epsilon} \sim$ Norm$[\mathbf{0}, \mathbf{I}]$ // Sample noise
    $\ell_i = \left\| \epsilon_\theta\big(\sqrt{\alpha_t}\boldsymbol{x_i} + \sqrt{1-\alpha_t}\boldsymbol{\epsilon}, t\big) - \boldsymbol{\epsilon} \right\|^2$ // Compute individual loss
  Accumulate losses for batch and take gradient step
until converged

3.9.4 Diffusion Model for Images



3.9.5 Guided Generation

Guide diffusion process using auxiliary data c, using the gradient of a trained classifier as guidance. Train the diffusion model unconditionally. Train a classifier $P(c|\mathbf{z}_t)$ where c are conditioning labels. Add an extra term when sampling the diffusion model that modifies the reconstruction in the direction given by the gradient of a classifier.

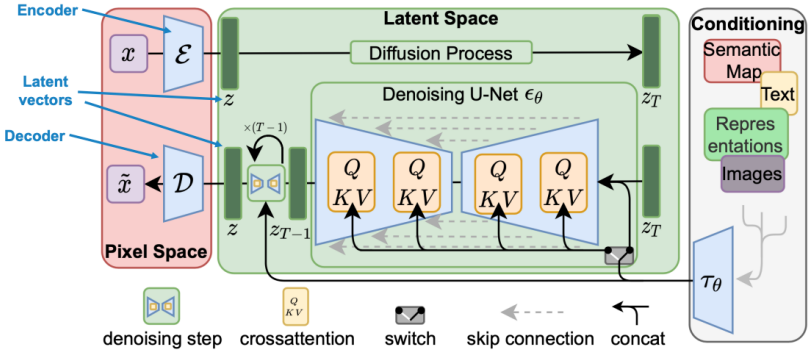
$$\mathbf{z}_{t-1} = \underbrace{\hat{\mathbf{z}}_{t-1} + \sigma_t \boldsymbol{\epsilon}}_{\text{Reversed diffusion}} + \underbrace{\sigma_t^2 \frac{\partial P(c|\mathbf{z}_t)}{\partial \mathbf{z}_t}}_{\text{Classifier guidance}}$$

3.9.6 Conditional Generation

We can condition the generating process in different way. **Scalar**: vector embedding + spatial addition (or adaptive group normalization), **image**: channel-wise concatenation of the conditional image, **text**: vector embedding + spatial addition or crossattention.

Latent Space Diffusion

Run diffusion in the latent space instead of pixel space for cost saving. It allows also to introduce semantic structuring.



Chapter 4

Reinforcement Learning

4.1 Introduction to Reinforcement Learning

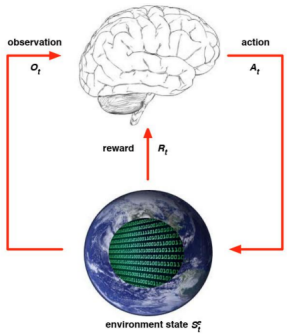
4.1.1 Fundamental Concepts

What characterizes Reinforcement Learning: no supervisor (only a reward signal); delayed asynchronous feedback; time matters (sequential data, continual learning); agent's actions affect the subsequent data it receives(inherent non-stationarity).

Reward

A reward R_t is a scalar feedback signal indicating how well the agent is doing at step t. The agent's primary objective is to maximize the cumulative reward, or more precisely, the expected cumulative reward, as reinforcement learning is founded on the reward hypothesis.
Actions may have long term consequences, reward may be delayed, it may be better to sacrifice immediate reward to gain more longterm reward.

Agents and Environments



S_t^e is the environment e private representation at time t.
S_t^a is the internal representation owned by agent a.

Full observability means that the agent directly observes the environment state $O_t = S_t^a = S_t^e$. **Partial observability** when the agent indirectly observes the environment $S_a^t \neq S_t^e$. The agent in this case needs to build its own state representation (history, beliefs, memory).

4.1.2 Components of a Reinforcement Learning Agent

Policy

A policy π is the agent's behaviour, a map from state s to action a. It is a distribution over actions a given the states.

Value Function

The value function v is a predictor of future reward, evaluates the goodness/badness of states.

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...|S_t = s] \tag{4.1}$$

It is an expectation of the discounted future rewards following policy π from state s.

Model

A model predicts the environment behaviour. Given an action a and a state s it predict next state s' and the next reward R_{t+1}.

4.1.3 Markov Decision Processes

A Markov Decision Process (MDP) is a Markov chain with rewards and actions. It is an environment in which all states are Markov.

A Markov Decision Process is a tuple $\langle \mathcal{S}, \mathcal{A}, \boldsymbol{P}, \mathcal{R}, \gamma \rangle$
- $\mathcal{S}$ is a finite set of states
- $\mathcal{A}$ is a finite set of actions a
- $\boldsymbol{P}$ is a state transition matrix, s.t. $P^a_{ss'} = P(S_{t+1} = s' | S_t = s, A_t = a)$
- $\mathcal{R}$ is a reward function, s.t. $\mathcal{R}^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$
- γ is a discount factor, $\gamma \in [0,1]$

The return G_t is the total discounted reward from time-step t, $G_t = \sum_{k=0} \gamma^k R_{t+k+1}$ where γ controls immediate reward vs delayed reward.

Bellman Equation

The state-value function $v(s)$ of a Markov Decision Process is the expected return starting from state s, $v(s) = \mathbb{E}[G_t | S_t = s]$. The value function can be decomposed into two parts: immediate reward and discounted value of successor state.

$$v(s) = \mathbb{E}[G_t | S_t] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t\right] = \tag{4.2}$$

$$= \mathbb{E}[R_{t+1} | S_t = s] + \gamma \mathbb{E}[v(S_{t+1}) | S_t = s] = \tag{4.3}$$

$$= \mathcal{R}_s + \gamma \sum_{s'} P_{ss'} v(s') \tag{4.4}$$

Adding the policy: the state-value function $v_\pi(s)$ of an MDP is the expected return starting from state s and following policy π, $v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$. The **action-value** function is the same but given also the action a,
$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$. The state-value and action-value function can again be decomposed into immediate reward plus discounted value of successor state and formulate them recursively.

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s, a) \tag{4.5}$$

$$q_\pi(s, a) = \mathcal{R}^a_s + \gamma \sum_{s' \in \mathcal{S}} P^a_{ss'} v_\pi(s') \tag{4.6}$$

One more step of nesting

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}^a_s + \gamma \sum_{s' \in \mathcal{S}} P^a_{ss'} v_\pi(s') \right)$$

The expected return of being in a state reachable from s through action a and then continue following policy

$$q_\pi(s, a) = \mathcal{R}^a_s + \gamma \sum_{s' \in \mathcal{S}} P^a_{ss'} \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s', a')$$

The expected return of any action a' taken from states reachable from s through action a (and then follow policy)

Finding an Optimal Policy

An optimal policy can be found by maximising over the action-value function $q_*(s,a)$. We want to find the optimal policy $\pi_*(a|s)$ that gives us 1 if $a = \arg\max_a q_*(s,a)$ and 0 otherwise. So if we know $q_*(s,a)$, we we straightforwardly find the optimal policy. Optimal value functions are recursively related Bellman-style.

$$v_*(s) = \max_{a \in \mathcal{A}} q_*(s,a) = \max_{a \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a v_*(s')$$

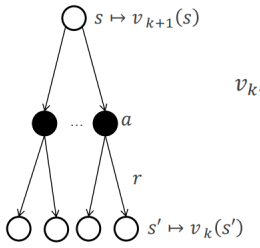
$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a v_*(s') = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a \max_{a' \in \mathcal{A}} q_*(s',a')$$

Bellman Optimality Equation is non-linear. No closed form solution (in general) but many iterative solution methods.

4.2 Model-Based RL

4.2.1 Iterative Policy Evaluation

We want to evaluate a given policy π by iterative application of Bellman expectation backup $v_1 \to v_2 \to ... \to v_\pi$. Using synchronous backups: at each iteration $k+1$, for all states $s \in \mathcal{S}$, update $v_{k+1}(s)$ from $v_k(s')$ where s' is a successor state of s.



$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a v_k(s') \right)$$

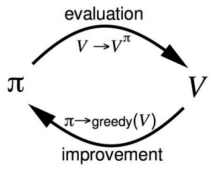
$$v_{k+1} = \mathcal{R}^\pi + \gamma \boldsymbol{P}^\pi v_k$$

Improve Policy

Given policy π:

- evaluate policy π: $v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + ... | S_t = s]$
- improve the policy by acting greedily with respect to v_π, $\pi' = greedy(\pi) \Rightarrow \pi'(s) = \arg\max_a q_\pi(s,a)$

This process of policy iteration always converges to π_*.

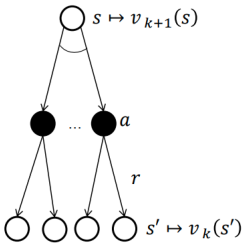


Generalized Policy Iteration

Do I need to reach ∞ steps? No.
Does policy evaluation need to converge to $v_{\pi*}$? Introduce stopping condition, stop after k iteration $\Rightarrow$ estimate v_k with any policy evaluation (e.g. less steps).
Why update policy every iteration? Value iteration $\Rightarrow$ generate $\pi' \geq \pi$ with any policy improvement algorithm.

4.2.2 Value Iteration

Unlike policy iteration, there is no explicit policy



$$v_{k+1}(s) = \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a v_k(s') \right)$$

$$v_{k+1} = \max_{a \in \mathcal{A}} (\boldsymbol{\mathcal{R}}^a + \gamma \boldsymbol{P}^a v_k)$$

4.3 Model-Free Reinforcement Learning

In a model-free setting, the learning agent focuses on learning an optimal policy, which is a mapping from states to actions, without explicitly modeling the underlying dynamics of the environment. The agent explores the environment, takes actions, and receives feedback in the form of rewards. It uses this feedback to update its policy or value function to improve its decision-making capabilities.
Model-Free prediction: estimate the value function of an unknown MDP.
Model-Free control: optimise the value function of an unknown MDP.

4.3.1 Prediction

Monte-Carlo RL

MC methods learn directly from full episodes of experience.
Policy evaluation: learn v_π from episodes of experience following policy π. Monte-Carlo policy evaluation uses empirical mean return instead of expected return.

Temporal-Difference RL

TD methods learn directly from episodes of experience but they can be incomplete. Every time an action is taken it can learn something, TD updates a guess towards a guess (bootstrapping).

MC vs TD learning

Goal: learn v_π from episodes of experience under policy π

Incremental every-visit MC

- Update value $V(S_t)$ toward actual return G_t

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))$$

Simplest temporal-difference learning algorithm (TD(0))

- Update value $V(S_t)$ toward estimated return $R_t + \gamma V(S_{t+1})$

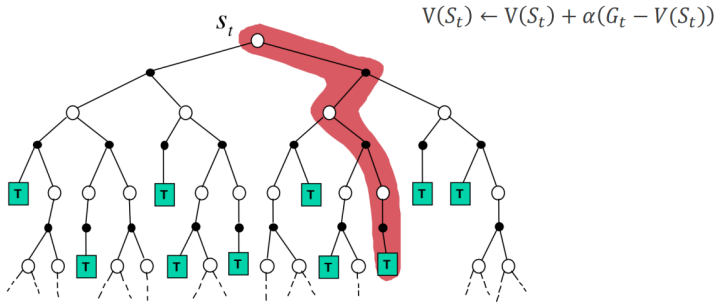
$$V(S_t) \leftarrow V(S_t) + \alpha(\underbrace{R_t + \gamma V(S_{t+1})}_{\text{TD target}} - V(S_t))$$

$$\underbrace{\qquad\qquad\qquad\qquad}_{\text{TD error } \delta_t}$$

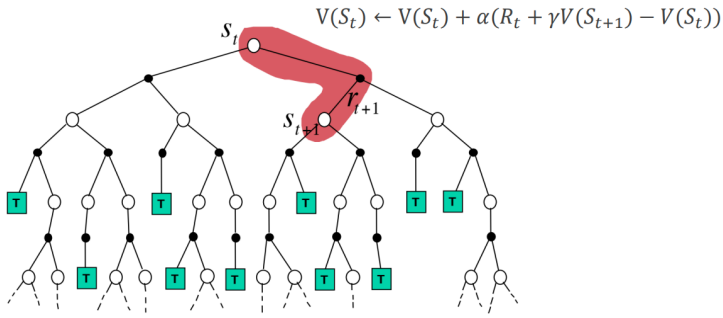
MC has high variance, zero bias, good convergence properties (even with function approximation), not very sensitive to initial value and is very simple to understand and use.
TD has low variance, some bias: usually more (sample) efficient than MC, $TD(0)$ converges to $v_\pi(s)$ (but not always with function approximation), more sensitive to initial value.

4.3.2 Unifyng and Generalizing

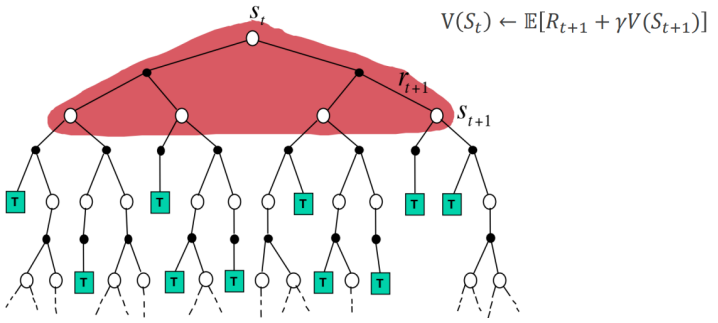
MC Update



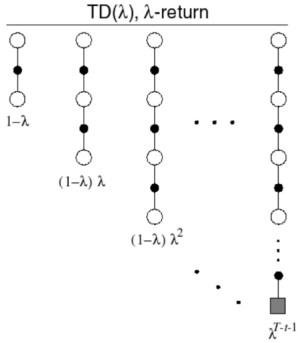
TD Update



Model-Based - Dynamic Programming



TD(λ)



Have TD look and target n steps in the future. The λ-return G_t^λ combines all $n-step$ returns $G_t^{(n)}$ using weight $(1-\lambda)\lambda^{n-1}$ (decaying factor behaving like the discount factor)

$$G_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_t^{(n)} \tag{4.7}$$

We can't wait possibly ∞ steps before updating so how do we implement the update in practice? With **Backward View TD(λ)**. Update online, every step, from incomplete sequences using **eligibility traces** for solving the credit assignment problem by trading between frequency heuristic (assign credit to most frequent states) and recency heuristic (assign credit to most recent states). Eligibility traces

help TD learning algorithms to learn more efficiently by propagating credit or blame backward through time, allowing the agent to update the value estimates of relevant states and actions. The eligibility trace vector is initialized to zero at the begin of the episode and it's updated state-by-state, with a fade part given by $\gamma\lambda$. TD(λ) algorithm updates the value $V(S_t)$ for every state s in proportion to the TD error δ_t and the eligibility trace $E_t(s)$.

$$\text{TD error: } \delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{4.8}$$
$$\text{Eligibility traces: } E_t(s) = \gamma\lambda E_{t-1}(s) + \mathbf{1}(S_t; s) \tag{4.9}$$
$$\text{Update: } V(s) = V(s) + \alpha\delta_t E_t(s) \tag{4.10}$$

4.3.3 Control

There's two approaches to optimise the value function of an unknown MDP:

- On-policy learning: learn on the job, learn about policy π from experience sampled from π. Using a policy to behave and optimizing that one.
- Off-policy learning: "look over someone's shoulder", learn about policy π from experience sampled from μ. Using a policy to behave while optimizing another one.

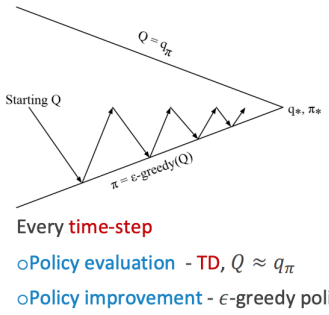
Greedy policy improvement over $Q(s, a)$ is model-free $\pi'(s) = \arg\max_a Q(s, a)$. As long as we operate with the Q function, the action-value function that estimates the expected cumulative reward an agent will receive by taking a specific action in a given state, we are model-free.

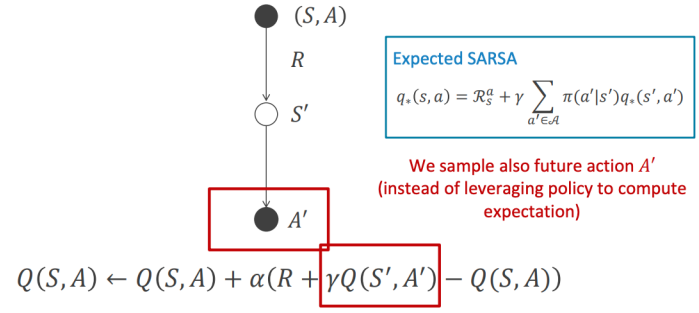
ϵ-greedy Exploration

We introduce non-determinism to sample the world in direction that greedy policy would not choose. All m actions are tried with non-zero probability: with probability $1 - \epsilon$ choose the greedy action ($\arg\max_a Q(s, a)$), with probability ϵ choose an action at random.

On-Policy Control with SARSA

SARSA is an on-policy TD control algorithm for estimating $Q \sim q_*$. Given a state s and an action a the $Q(s, a)$ is updated adding to the old value the difference between the TD error and the actual actual $Q(s, a)$.





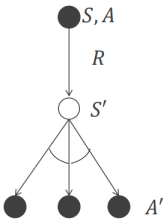
Off-Policy Control with Q-Learning

Evaluate target policy $\pi(a|s)$ to compute $v_\pi(s)$ or $q_\pi(s, a)$ while following behaviour policy $\mu(a|s)$ (learn from imitation, learn optimal policy while following exploration policy).
In **Q-Learning** the next action is chosen using behaviour policy $A_{t+1} \sim \mu(\cdot|S_t)$ but we consider alternative successor action $A' \sim \pi(\cdot|S_t)$. And update Q towards value of alternative action.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(\underbrace{R_{t+1} + \gamma Q(S_{t+1}, A')}_{target} - Q(S_t, A_t)) \tag{4.11}$$

Allow both behaviour and target policies to improve, the target policy π is greedy and the behaviour policy μ is ϵ-greedy. Then the Q-learning target simplifies to

$$R_{t+1} + \gamma Q(S_{t+1}, A') = R_{t+1} + \max_{a'} \gamma Q(S_{t+1}, a') \tag{4.12}$$



Theorem

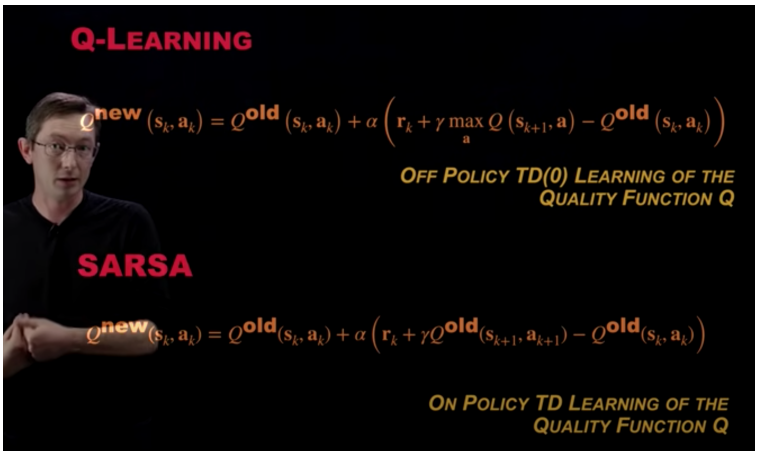
Q-learning control converges to the optimal action-value function, $Q(s,a) \rightarrow q_*(s,a)$

$$Q(S,A) \leftarrow Q(S,A) + \alpha\left(R + \max_{a'} \gamma Q(S',a') - Q(S,A)\right)$$

4.3.4 Dynamic Programming Vs Temporal Difference Learning

	Full Backup (DP)	*Sample Backup (TD)*
Bellman Expectation Equation for $v_\pi(s)$	Iterative Policy Evaluation	TD Learning
Bellman Expectation Equation for $q_\pi(s,a)$	Q-Policy Iteration	Sarsa
Bellman Optimality Equation for $q_*(s,a)$	Q-Value Iteration	Q-Learning

Recap: https://www.youtube.com/watch?v=0iqz4tcKN58



4.4 Value Function Approximation

We would like to be able to use reinforcement learning in problems with non-trivial state spaces. So far we've been working with assessing the value function for a policy where $V(s)/Q(s,a)$ is a lookup table. An entry for every state s or state-action pair s,a (bad at generalization). We want to scale up model-free methods for prediction and control. The new approach is to estimate value function with function approximation $\hat{v}(s;\mathbf{w}) \approx v_\pi(s)$, $\hat{q}(s,a;\mathbf{w}) \approx q_\pi(s,a)$.Update parameters $\mathbf{w}$ using MC or TD learning.

4.4.1 Incremental Methods

Learn as we go (online learning). We can use the approximator that we want but let's focus on differentiable methods.

Stochastic Gradient Descent in Value Approximation

Find parameter vector $\mathbf{w}$ minimising mean-squared error between approximate value $\hat{v}(s;\mathbf{w})$ and true value function $v_\pi(s)$.

$$J(\mathbf{w}) = \mathbb{E}_\pi[(v_\pi(S) - \hat{v}(S;\mathbf{w}))^2] \tag{4.13}$$

Stocastic approximation (sampling e.g. via MC), update rule for the linear model:

$$\Delta\mathbf{w} = \alpha(v_\pi(S) - \hat{v}(S;\mathbf{w}))\nabla_\mathbf{w}\hat{v}(S;\mathbf{w}) = \alpha(v_\pi(S) - \hat{v}(S;\mathbf{w}))\mathbf{x}(S) \tag{4.14}$$

Value as a linear combination of state features $\hat{v}(S;\mathbf{w}) = \mathbf{x}(S)^T\mathbf{w} \Rightarrow \nabla_\mathbf{w}\hat{v}(S;\mathbf{w}) = \mathbf{x}(S)$.

4.4.2 Incremental Prediction

So far have assumed access to true $v_\pi(s)$, but in RL there is no supervisor, only rewards. In practice, we substitute a target for $v_\pi(s)$

MC - Target is the return G_t

$$\Delta\boldsymbol{w} = \alpha\big(G_t - \hat{v}(S_t;\boldsymbol{w})\big)\nabla_w\hat{v}(S_t;\boldsymbol{w})$$

TD(0) - Target is the TD target $R_{t+1} + \gamma\hat{v}(S_{t+1};\boldsymbol{w})$

$$\Delta\boldsymbol{w} = \alpha\big(R_{t+1} + \gamma\hat{v}(S_{t+1};\boldsymbol{w}) - \hat{v}(S_t;\boldsymbol{w})\big)\nabla_w\hat{v}(S_t;\boldsymbol{w})$$

TD(λ) - Target is the λ-return G_t^λ

$$\Delta\boldsymbol{w} = \alpha\left(G_t^\lambda - \hat{v}(S_t;\boldsymbol{w})\right)\nabla_w\hat{v}(S_t;\boldsymbol{w})$$

Backward view linear TD(λ) (where E is the eligibility traces)

$$\begin{aligned}\delta_t &= R_{t+1} + \gamma\hat{v}(S_{t+1};\boldsymbol{w}) - \hat{v}(S_t;\boldsymbol{w})\\ E_t &= \lambda\gamma E_{t-1} + \boldsymbol{x}(S_t)\\ \Delta\boldsymbol{w} &= \alpha\delta_t E_t\end{aligned}$$

4.4.3 Incremental Control

It's pretty much the same as before changing variable, instead of $\hat{v}$ we have $\hat{q}$.

MC - Target is return return G_t

$$\Delta\boldsymbol{w} = \alpha\big(G_t - \hat{q}(S_t,A_t;\boldsymbol{w})\big)\nabla_w\hat{q}(S_t,A_t;\boldsymbol{w})$$

TD(0) - Target is the TD target $R_{t+1} + \gamma\hat{q}(S_{t+1},A_{t+1};\boldsymbol{w})$

$$\Delta\boldsymbol{w} = \alpha\big(R_{t+1} + \gamma\hat{q}(S_{t+1},A_{t+1};\boldsymbol{w}) - \hat{q}(S_t,A_t;\boldsymbol{w})\big)\nabla_w\hat{q}(S_t,A_t;\boldsymbol{w})$$

Forward TD(λ) - Target is the action-value λ-return

$$\Delta\boldsymbol{w} = \alpha\left(q_t^\lambda - \hat{q}(S_t,A_t;\boldsymbol{w})\right)\nabla_w\hat{q}(S_t,A_t;\boldsymbol{w})$$

Backward TD(λ) - Equivalent target

$$\begin{aligned}\delta_t &= R_{t+1} + \gamma\hat{q}(S_{t+1},A_{t+1};\boldsymbol{w}) - \hat{q}(S_t,A_t;\boldsymbol{w})\\ E_t &= \lambda\gamma E_{t-1} + \nabla_w\hat{q}(S_t,A_t;\boldsymbol{w})\\ \Delta\boldsymbol{w} &= \alpha\delta_t E_t\end{aligned}$$

4.4.4 Convergence of Prediction Algorithms

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD(0)	✓	✓	✗
	TD(λ)	✓	✓	✗
Off-Policy	MC	✓	✓	✓
	TD(0)	✓	✗	✗
	TD(λ)	✓	✗	✗

4.4.5 Batch Methods

In reinforcement learning, batch methods refer to a class of algorithms that learn from a fixed dataset or batch of experiences. Unlike online methods, which learn and update the policy or value function on the fly as the agent interacts with the environment, batch methods use a pre-collected dataset to perform offline learning.
Gradient descent is simple and appealing but it is not sample efficient. Batch methods seek to find the best fitting value function given the agent's experience (training data). Closer to supervised learning.

SGD with Experience Replay

Instead of learning from "real life" data we learn sampling from a collected data D (experience from the past).

Given value function approximation $\hat{v}(s;\boldsymbol{w}) \approx v_\pi(s)$ and experience $\mathcal{D}$ consisting of $\langle state, value \rangle$ pairs

$$\mathcal{D} = \{\langle S_1, v_1^\pi \rangle, \langle S_2, v_2^\pi \rangle, \dots, \langle S_T, v_T^\pi \rangle\}$$

Repeat

1. Sample state, value from experience
$$\langle s, v^\pi \rangle \sim \mathcal{D}$$
2. Apply stochastic gradient descent update
$$\Delta \boldsymbol{w} = \alpha(v^\pi - \hat{v}(s;\boldsymbol{w}))\nabla_{\boldsymbol{w}}\hat{v}(s;\boldsymbol{w})$$

Converges to least squares solution

$$\boldsymbol{w} = \arg\min_{\boldsymbol{w}} LS(\boldsymbol{w})$$

Deep Q-Networks

DQN uses experience replay and fixed Q-targets. Take action a_t according to ϵ-greedy policy. Sample random mini-batch of transitions (s, a, r, s') from D (previously stored). Compute Q-learning targets with respect to old fixed parameters $\mathbf{w}^-$ (set of parameters from the past).
Q function is not tabular but it's approximated with a Deep Neural Network. Optimise MSE between Q-network and Q-learning targets.

$$\mathcal{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i}\left[\left(r + \gamma \max_{a'} Q(s', a'; w_i^-) - Q(s, a; w_i)\right)^2\right]$$

4.5 Policy-based RL

We've been trying to approximate value or action-value function using parameters θ to generate policy (e.g. using ϵ-greedy). Now we want to parametrise the policy $\pi_\theta(s, a) = P(a|s, \theta)$ while maintaining the focus on model-free RL.
The **advantages** for this approach are: better convergence properties, effective in high-dimensional or continuos action space, can learn stochatstic policies.
While the **cons** are: typically converges to a local optimum, evalutating the policy is typically inefficient and high variance.

4.5.1 Policy Gradient

Policy Objective Functions

We need a function for evaluating how good the policy π_θ is. Then we can maximize that function manipulating θ.
There are two types of environments. **Episodic environments**: use start value $J_1(\theta) = V^{\pi_\theta} = \mathbb{E}_{\pi_\theta}[v_1]$ (MC approach). **Continuing environment** "learn while living": average value $J_{\bar{V}}(\theta) = \sum_s d^{\pi_\theta}(s)V^{\pi_\theta}(s)$, average reward (per time-step) $J_{\bar{R}}(\theta) = \sum_s d^{\pi_\theta}(s)\sum_a \pi_\theta(s, a)\mathcal{R}_s^a$, where $d^{\pi_\theta}(s)$ is stationary distribution of Markov chain for π.
Policy gradient algorithms search for a local maximum in $J(\theta)$ (policy objective function) by ascending

the gradient of the policy w.r.t. θ. For any differentiable policy $\pi_\theta(s,a)$ for any policy objective functions the policy gradient is

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla \pi_\theta \log \pi(s,a) Q^{\pi_\theta}(s,a)] \tag{4.15}$$

Thanks to this theorem we can keep the model-free approach ⇒ we don't need to know the environment (there's no d nor state transition function and we are using the Q function).

Score Function

So far we're interested into learning a policy. How do we learn a policy? We find some function approximator with parameters θ and we parameterize the policy. We have a function $J(\theta)$ to optimize, a function that tells us how good a specific policy is, but rather than optimizing that function, the policy gradient theorem tells us that we can optimize a slightly different function which is model-free. Now how do we parameterise the policy? The **score function** is $\nabla_\theta \log \pi(s,a)$.
We use the **Softmax Policy** when dealing with discrete actions: weight actions using linear combination of features $\phi(s,a)^T\theta$, probability of action is proportional to exponentiated weight.

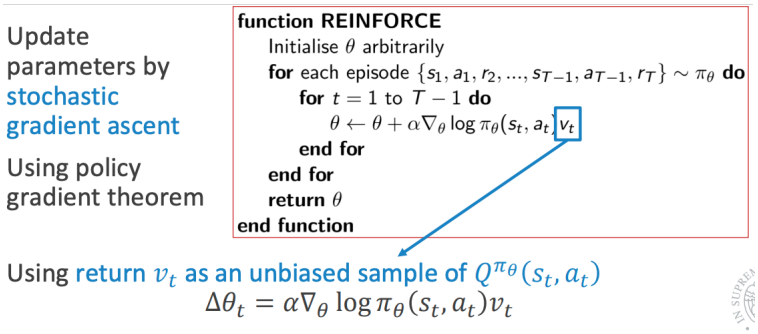
$$\pi_\theta(s,a) \approx e^{\phi(s,a)^T\theta} \Rightarrow \nabla_\theta \log \pi(s,a) = \phi(s,a) - \mathbb{E}_{\pi_\theta}[\phi(s,\cdot)] \tag{4.16}$$

Gaussian Policy: natural choice in continuous action spaces. Mean is a linear combination of state features $\mu(s) = \phi(s)^T\theta$, variance σ^2 may be fixed or parametrised. Then the policy is $a \sim \mathcal{N}(\mu(s), \sigma^2)$ and the score function is

$$\nabla_\theta \log \pi(s,a) = \frac{(a - \mu(s))\phi(s)}{\sigma^2} \tag{4.17}$$

REINFORCE Algorithm

MC Policy Gradient



4.5.2 Policy Gradient vs Maximum Likelihood

Policy gradient

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(s_t^i, a_t^i)\right)\left(\sum_{t=1}^{T} v_t^i\right)$$

Maximum Likelihood

$$\nabla_\theta J_{ML}(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(s_t^i, a_t^i)\right)$$

Thanks to the blue term good things are made more likely, bad things are made less likely (if a truck crushes that influences negatively in the maximum likelihood so that action won't be taken again even if it was the most probable one). Policy gradient is on policy so it needs to learn from actual experience. Neural networks change only slightly with each gradient step so naïve on-policy learning can be extremely inefficient!

4.5.3 Actor Critic

If we go with the MC approach we are inefficent and we have noise gradient (high variance). Rather than using the actual return we use TD learning return (low variance, high bias). We use a critic to estimate the action-value function $Q_w(s,a) \approx Q^{\pi_\theta}(s,a)$.
In reinforcement learning, an actor-critic algorithm is a hybrid approach that combines the benefits of both value-based and policy-based methods. It consists of two main components: an actor and a critic, each with a specific role in the learning process. The key idea behind the actor-critic algorithm is that the actor learns from the guidance and feedback provided by the critic. The critic's value function estimates are used to evaluate the actions taken by the actor, and this evaluation signal is used to update the actor's policy. In turn, the updated policy of the actor influences the actions taken and the rewards received, which are used to update the critic's value function estimates. So the critic updates action-value function parameters w and the actor updates policy parameters θ, in direction suggested by critic.
Actor-critic algorithms follow an approximate policy gradient

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) Q_w(s,a)]$$
$$\Delta\theta = \alpha \nabla_\theta \log \pi_\theta(s,a) Q_w(s,a)$$

Action-Value

The critic is solving a familiar problem: policy evaluation.

Simple actor-critic algorithm based on action-value critic

Using linear approximation $Q_w(s,a) = \phi(s,a)^T w$

- Critic - Updates w by linear TD(0)
- Actor - Updates θ by policy gradient

function QAC
  Initialise s, θ
  Sample $a \sim \pi_\theta$
  for each step **do**
    Sample reward $r = \mathcal{R}_s^a$; sample transition $s' \sim \mathcal{P}_{s,\cdot}^a$
    Sample action $a' \sim \pi_\theta(s', a')$
    $\delta = r + \gamma Q_w(s', a') - Q_w(s, a)$
    $\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(s, a) Q_w(s, a)$
    $w \leftarrow w + \beta \delta \phi(s, a)$
    $a \leftarrow a', s \leftarrow s'$
  end for
end function

Bias and Variance in Actor-Critic Algorithms

Approximating the policy gradient introduces **bias**. Luckily, if we choose value function approximation carefully, we can avoid introducing any bias. For **reducing variance** We subtract a baseline function $B(s)$ from the policy gradient, a good baseline is the state value function $B(s) = V^{\pi_\theta}(s)$. So we can rewrite the policy gradient using the advantage function A.

$$A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)$$
$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) A^{\pi_\theta}(s,a)]$$

The advantage function can significantly reduce variance of policy gradient so the critic should really estimate the advantage function. For example, by estimating both Q and V using two function approximators (e.g. Neural Networks)and two parameter vectors. I don't need to do that approximating the TD error.

Given true value function $V^{\pi_\theta}(s)$ the TD error δ^{π_θ}

$$\delta^{\pi_\theta} = r + \gamma V^{\pi_\theta}(s') - V^{\pi_\theta}(s)$$

is an unbiased estimate of the advantage function

$$\mathbb{E}[\delta^{\pi_\theta} | s, a] = \mathbb{E}[r + \gamma V^{\pi_\theta}(s') | s, a] - V^{\pi_\theta}(s)$$
$$= Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s) = A^{\pi_\theta}(s,a)$$

We can use the TD error to compute the policy gradient

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) \delta^{\pi_\theta}]$$

In practice need to approximate TD error

$$\delta_v = r + \gamma V_v(s') - V_v(s)$$

Only one set of critic parameters v is needed

4.5.4 Natural Policy Gradient

We need a methods that tell us how much should we be moving in the directions of the gradient, how much can we trust the gradient (control the α for each possible direction).
The **natural gradient** refers to an optimization technique that takes into account the geometry or curvature of the parameter space when updating model parameters. Traditional gradient descent algorithms update the parameters based solely on the gradient, which treats all parameters equally. In contrast, the natural gradient incorporates information about the structure of the parameter space, allowing for more efficient and effective updates.
The main idea behind **Trust Region Policy** (TRPO) is to perform updates that are constrained within a trust region, limiting the magnitude of policy changes.
The main idea behind **Proximal Policy Optimization** (PPO) is to iteratively update the policy by taking steps that are constrained to be close to the previous policy, preventing large policy updates that could lead to unstable learning.