CS 281: Advanced Machine Learning

taught by Sasha Rush

Fall 2017

Contents

CS281: Advanced ML **September 6, 2017**

Lecture 2: Discrete Models

Lecturer: Sasha Rush *Scribes: Anna Sophie Hilgard, Diondra Peck*

- Discrete models take values from a countable set, e.g. {0,1}, {cold, flu, asthma} and are simpler than continuous models.
- We will use simple discrete models to develop our tactics such as marginalization and conditioning.
- Today, we will focus coins as a real-world example.

2.1 Bernoulli model

The likelihood is of the form $p(\text{heads}) = \theta$.

Easy Prior

Assume we know the coin came from one of 3 unknown manufacturers (later, we'll have mixture model estimation, but for now assume these probabilities come from an oracle).

1. $\theta = 0.4$ with probability .1
2. $\theta = 0.5$ with probability .8
3. $\theta = 0.6$ with probability .1

$$p(\theta) = 0.1 \cdot \delta(\theta = 0.4) + 0.8 \cdot \delta(\theta = 0.5) + 0.1 \cdot \delta(\theta = 0.6)$$

Likelihood

Likelihood = $p(\text{data}|\text{parameters})$. For the coin example,

$$p(\text{coin flips}|\theta) = \text{Bin}(N_1|N, \theta) = \binom{N}{N_1}\theta^{N_1}(1-\theta)^{N-N_1} \quad \text{where } N = N_0 + N_1 = \text{number of flips}$$

Note that the last two terms, the "score", is our focus since they are the only terms that depend on θ. The first term normalizes the distribution.

2.2 Inference

Inference 1: $p(\theta|x)$ ($x \in N_0, N_1$). How can we estimate θ?

Maximum Likelihood Estimation (MLE)

$\theta_{MLE} = \text{argmax}_\theta \; p(N_0, N_1|\theta) = \text{argmax}_\theta \log\left[p(N_0, N_1|\theta)\right]$

$\theta_{MLE} = \text{argmax}_\theta \log\binom{N}{N_1} + N_1 \log\theta + N_0 \log(1-\theta)$ Because the first term is not a function of θ, we can ignore it.

$\frac{d}{d\theta} = \frac{N_1}{\theta} + \frac{N_0}{1-\theta} \cdot (-1) \rightarrow \theta_{MLE} = \frac{N_1}{N_0 + N_1}$

Note that Inference $\neq$ Decision Making. If we asked you to make a bet on the coin, based on this you could either

1. Always take heads if $\theta > .5$. In this case, $p(\text{win}) = \theta$
2. Take heads with probability = θ. In this case, $p(\text{win}) = \theta^2 + (1-\theta)^2$ [p(is heads) * p(choose heads) + ...]

If θ = 0.6, for option 1, p(win) = θ = 0.6. For option 2, p(win) = $\theta^2 + (1-\theta)$ = 0.52. In this case, the additional information used in the calculation of option 2 does not result in a better decision.

Maximizing the Posterior (MAP)

Bayes Rule : $p(\theta|\text{data}) \propto p(\text{data}|\theta)p(\theta)$

- Posterior: $p(\theta|\text{x})$
- Likelihood: $p(\text{x}|\theta)$
- Prior: $p(\theta)$

$$\theta_{MAP} = \text{argmax}_\theta \; p(\theta|\text{x}) = \text{argmax}_\theta \log\left[p(\text{x}|\theta)p(\theta)\right] \quad \text{from Bayes' Rule: } p(\theta|\text{x}) \propto p(\text{x}|\theta)p(\theta)$$

Consider an example:

$$p(\theta = 0.4|N_0, N_1) \propto \binom{N}{N_1}(.4)^{N_1}(1-.4)^{N_0}(0.1)$$

$$p(\theta = 0.45|N_0, N_1) = 0 \quad \text{Due to the sparsity of the prior - similar result for } \theta = 0.5 \text{ and } 0.6$$

$\theta_{MAP} = \theta_{MLE}$ when we have a uniform prior since the MLE calculation does not explicitly factor a prior into its calculation.

Full Posterior

Partition or Marginal Likelihood: $p(N_0, N_1) = \int_\theta p(N_0, N_1, \theta)$.

$$p(\theta|N_0, N_1) = \frac{p(x|\theta)p(\theta)}{p(N_0, N_1)} \qquad \text{Note that } p(N_0, N_1) \text{ is a very difficult term to compute.}$$

Beta Prior

$$p(\theta|\alpha_0, \alpha_1) = \frac{\Gamma(\alpha_0 + \alpha_1)}{\Gamma(\alpha_0)\Gamma(\alpha_1)}\theta^{\alpha_1 - 1}(1-\theta)^{\alpha_0 - 1} \qquad \text{support} \in [0, 1]$$

From the image of the beta function for different parameters, we can see that it can ether be balanced, skewed to one side, or tend toward infinity on one side.
With a beta prior:

$$p(\theta|N_0, N_1) = \frac{\Gamma(\alpha_0 + \alpha_1)}{\Gamma(\alpha_0)\Gamma(\alpha_1)}\theta^{\alpha_1 - 1}(1-\theta)^{\alpha_0 - 1}\theta^{N_1}(1-\theta)^{N_0} \cdot (\text{constant normalization term w.r.t } \theta)$$

The key insight is that we get additive terms in the exponent and the resulting distribution looks like another beta. The prior "counts" (pseudocounts) from the hyperparameters can be interpreted as counts we have beforehand.

$$p(\theta|N_0, N_1) = \frac{\Gamma(\alpha_0 + \alpha_1)}{\Gamma(\alpha_0)\Gamma(\alpha_1)}\theta^{N_1 + \alpha_1 - 1}(1-\theta)^{N_0 + \alpha_0 - 1} \cdot (\text{constant normalization term w.r.t } \theta)$$

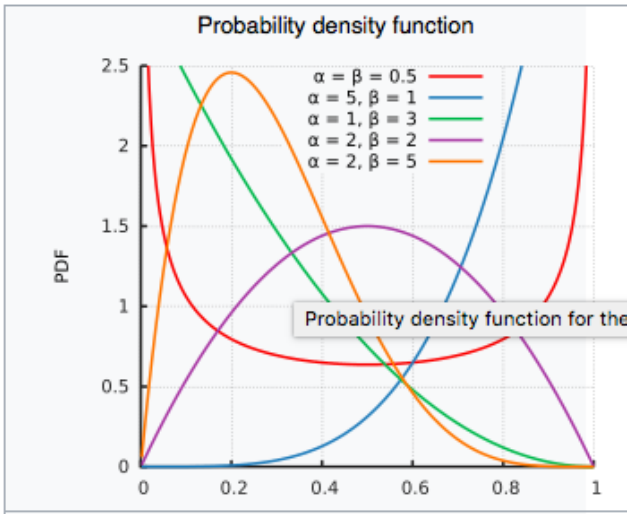


Figure 2.1: Beta Params

To make this distribution sum to 1, use the known beta normalizer

$$p(\theta|N_0, N_1) = \frac{\Gamma(\alpha_0 + \alpha_1 + N_0 + N_1)}{\Gamma(\alpha_0)\Gamma(\alpha_1)\Gamma(N_0)\Gamma(N_1)}\theta^{N_1+\alpha_1-1}(1-\theta)^{N_0+\alpha_0-1} \sim \text{Beta}(\theta|N_0+\alpha_0, N_1+\alpha_1) \quad \text{(posterior)}$$

The mode of the Beta gives us back θ_{MAP}, but with additional information about the shape of the distribution. What does the prior that tends to infinity at 1 imply? That in the absence of other information, the coin is definitely heads.

Predictive Distribution

$$\begin{aligned} p(\hat{x}|N_0, N_1) &= \int_\theta p(x|\theta, N_0, N_1)p(\theta|N_0, N_1)d\theta \\ &= \int_\theta \theta p(\theta|N_0, N_1)d\theta \\ &= \mathbb{E}_{\theta\sim p(\theta|N_0,N_1)}\theta \end{aligned}$$

This is the expectation under the posterior of θ which is the mean of the Beta distribution. Feel free to prove this as an exercise.

Marginal Likelihood

$$\begin{aligned} p(N_0, N_1) &= \int_\theta p(x_1, \dots x_n|\theta)p(\theta)d\theta \\ &= \int_\theta \frac{\Gamma(\alpha_1+\alpha_1)}{\Gamma(\alpha_0)\Gamma(\alpha_1)}\theta^{\alpha_1+N_1-1}(1-\theta)^{\alpha_0+N_0-1} \end{aligned}$$

The first term can be moved outside, as it does not depend on θ. After introducing our normalization term and making the distribution insidesum to 1,

$$p(N_0, N_1) = \frac{\Gamma(\alpha_1+\alpha_1)}{\Gamma(\alpha_0)\Gamma(\alpha_1)}\frac{\Gamma(N_0+\alpha_0)\Gamma(N_1+\alpha_1)}{\Gamma(N_0+N_1+\alpha_0+\alpha_1)}$$

2.3 Extensions on the Coin Flip Model: Super Coins

- Many correlated coins: models of binary data, important for discrete graphical models
- Many-sided coins aka dice: models of categorical data, generalization of Bernoulli

2.4 Other Distributions

$$\text{Bernoulli}(x|\theta) = \theta^x(1-\theta)^x$$

$$\text{Categorical}(x|\theta) = \prod_k \theta_k^{x_k} \qquad \text{generalization of Bernoulli}$$

$$\text{Multinomial}(x|\theta) = \frac{(\sum x_k)!}{\prod_k x_k!} \prod_k \theta_k^{x_k} \qquad \text{generalization of Binomial}$$

$$\text{Dirichlet}(x|\alpha) = \frac{\Gamma(\sum \alpha_k)}{\prod \Gamma(\alpha_k)} \prod_k \theta_k^{\alpha_k - 1} \qquad \text{generalization of Beta, often used as a prior}$$

Note that the Dirichlet distribution is the conjugate prior of the Categorical and Multinomial distributions.

2.5 Example notebook

See Beta.ipynb

CS281: Advanced ML **September 11, 2017**

Lecture 3: Multivariate Normal Distributions

Lecturer: Sasha Rush *Scribes: Christopher Mosch, Lindsey Brown, Ryan Lapcevic*

3.1 Examples

Multivariate gaussians are used for modeling in various applications, where knowing mean and variance is useful:

- radar: mean and variance of approaching objects (like invading aliens)
- weather forecasting: predicting the position of a hurricane, where the uncertainty in the storm's position increases for timepoints farther away
- tracking the likely outcome of a sports game: last year's superbowl is an example of a failure of modeling with multivariate gaussians as the Patriots still won after a large Falcons' lead

3.2 Review: Eigendecomposition

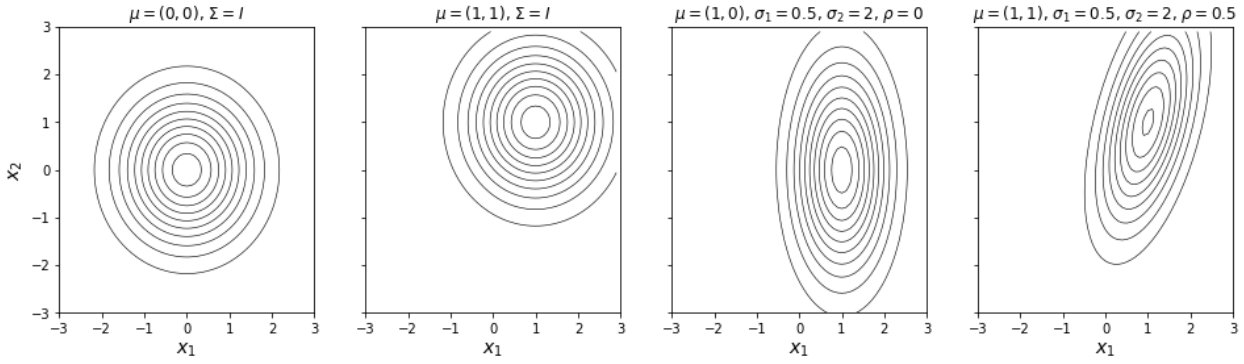
Let $\mathbf{\Sigma}$ be a square, symmetric matrix. Then its eigendecomposition is given by $\mathbf{\Sigma} = \boldsymbol{U}^T \mathbf{\Lambda} \boldsymbol{U}$, where $\boldsymbol{U}$ is an orthogonal matrix and $\mathbf{\Lambda}$ is a diagonal matrix. In the special case that $\mathbf{\Sigma}$ is positive semidefinite (as is the case for covariance matrices), denoted $\mathbf{\Sigma} \succeq 0$, all its eigenvalues are nonnegative, $\Lambda_{ii} \geq 0$, and we can decompose its inverse as $\mathbf{\Sigma}^{-1} = \boldsymbol{U}^T \mathbf{\Lambda}^{-1} \boldsymbol{U}$, where $\Lambda_{ii}^{-1} = 1/\Lambda_{ii}$.

3.3 Multivariate Normal Distributions (MVNs)

Let X be a D-dimensional MVN random vector with mean $\boldsymbol{\mu}$ and covariance matrix $\mathbf{\Sigma}$, denoted $X \sim \mathcal{N}(\boldsymbol{\mu}, \mathbf{\Sigma})$. Then the pdf of X is

$$p(x) = (2\pi)^{-D/2} |\mathbf{\Sigma}|^{-1/2} \exp\left[-\frac{1}{2}(x-\boldsymbol{\mu})^T \mathbf{\Sigma}^{-1}(x-\boldsymbol{\mu})\right],$$

where for many problems we focus on the quadratic form $(x-\boldsymbol{\mu})^T \mathbf{\Sigma}^{-1}(x-\boldsymbol{\mu})$ (which geometrically can be thought of as distance) and ignore the normalization factor $(2\pi)^{-D/2} |\mathbf{\Sigma}|^{-1/2}$. The figure below plots the contours of a bivariate Normal for various $\boldsymbol{\mu}$ and $\mathbf{\Sigma}$ (in the figure, ρ denotes the off-diagonal elements of $\mathbf{\Sigma}$, given by the covariance of x_1 and x_2).



Note that we can decompose $\mathbf{\Sigma}$ as

$$\begin{aligned}\mathbf{\Sigma} &= (x-\boldsymbol{\mu})^T\mathbf{\Sigma}^{-1}(x-\boldsymbol{\mu}) \\ &= (x-\boldsymbol{\mu})^T\left(\boldsymbol{U}^T\mathbf{\Lambda}^{-1}\boldsymbol{U}\right)(x-\boldsymbol{\mu}) \\ &= (x-\boldsymbol{\mu})^T\left(\sum_d \frac{1}{\lambda_d}U_d U^T\right)(x-\boldsymbol{\mu}) \\ &= \sum_d \frac{1}{\lambda_d}(x-\boldsymbol{\mu})^T U_d U_d^T(x-\boldsymbol{\mu}),\end{aligned}$$

where $(x-\boldsymbol{\mu})^T U_d$ can be interpreted as the projection of $(x-\boldsymbol{\mu})$ onto U_d (which can each be thought of as univariate gaussians), the eigenvector corresponding to the eigenvalue λ_d. Since $\mathbf{\Sigma}$ is the weighted sum of the dot product of such projections (with weights being given by $1/\lambda_d$, which can be thought of as the scale $1/\sigma^2$), we can describe the MVN as tiling of univariates.

3.3.1 Manipulating MVNs: Stretches, Rotations, and Shifts

Let $x \sim \mathcal{N}(0, \boldsymbol{I})$ and $y = \boldsymbol{A}x + \boldsymbol{b}$. We want to consider two ways of obtaining the complete distribution of y.

- 'Overkill': We can perform a change of variables[1]. Here, we have $x = \boldsymbol{A}^{-1}$ and $|dx/dy| = |A^{-1}|$, leading to

$$\begin{aligned}p(y) &= \mathcal{N}\left(\boldsymbol{A}^{-1}(y-\boldsymbol{b})|0,\boldsymbol{I}\right)|\boldsymbol{A}^{-1}| \\ &= \frac{1}{z}\exp\left[(\boldsymbol{A}^{-1}(y-\boldsymbol{b}))^T(\boldsymbol{A}^{-1}(y-\boldsymbol{b}))\right] \\ &= \frac{1}{z}\exp\left[(y-\boldsymbol{b})^T(\boldsymbol{A}^{-1})^T(\boldsymbol{A}^{-1})(y-\boldsymbol{b})\right] \\ &= \mathcal{N}(y|\boldsymbol{b}, \boldsymbol{A}\boldsymbol{A}^T),\end{aligned}$$

  where z is the normalizing constant.

- Using the properties of MVN, we know that y is also MVN, so is completely specified by its mean and covariance matrix which can easily be derived,

$$\mathbb{E}(y) = \mathbb{E}(\boldsymbol{A}x+\boldsymbol{b}) = \boldsymbol{A}\mathbb{E}(x)+\boldsymbol{b} \qquad \text{cov}(y) = \boldsymbol{A}\boldsymbol{A}^T.$$

Thus, we can generate MVN from $\mathcal{N}(0, \boldsymbol{I})$ via the transformation $y = \boldsymbol{A}x + \boldsymbol{b}$, where we set $\boldsymbol{A} = \boldsymbol{U}\mathbf{\Lambda}^{1/2}$, leading to $\mathbf{\Sigma}_Y = \boldsymbol{U}^T\mathbf{\Lambda}\boldsymbol{U}$. Then shifts are represented by $\boldsymbol{b}$, stretches by $\mathbf{\Lambda}$, and rotations by $\boldsymbol{U}$.

3.3.2 Detour: MVN in High-Dimensions ($D \gg 0$)

Let x be a D-dimensional random vector, distributed as $\mathcal{N}(0, I/D)$, where I is the identity. The expected length of x is given by

$$\mathbb{E}\left(\|x\|^2\right) = \mathbb{E}\left(\sum_d x_d^2\right) = D\sigma_d^2 = 1,$$

which means that x is expected to be on the boundary of a unit sphere centered at the origin. Moreover, the variance of the length is

$$\text{var}\left(\|x\|^2\right) = D\cdot\left(\mathbb{E}(x^4) - \mathbb{E}\left(x^2\right)^2\right) = D\cdot(3\sigma^4-\sigma^4) = 2D/D^2 = 2/D$$

[1] A change of variables can be done in the following way: Let $y = f(x)$ and assume f is invertible so that $x = f^{-1}(y)$. Then $p(y) = p(x)|dx/dy|$. This is a technique which will be used often in this course

Thus, it is not only expected that x lies on the boundary but as D increases most of its realizations will in fact fall on the boundary[2].

3.3.3 Key Formulas for MVN: Marginalization and Conditioning

Let $X \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ with

$$X = \begin{pmatrix} X_1 \\ X_2 \end{pmatrix}, \quad \boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \end{pmatrix}, \quad \boldsymbol{\Sigma} = \begin{pmatrix} \boldsymbol{\Sigma}_{11} & \boldsymbol{\Sigma}_{12} \\ \boldsymbol{\Sigma}_{21} & \boldsymbol{\Sigma}_{22} \end{pmatrix}.$$

Note that Σ is written in block matrix form, rather than scalar entries. It turns out that the marginals, X_1 and X_2, are also MVN, and their mean and covariance matrice are given by $\boldsymbol{\mu}_1$ and $\boldsymbol{\Sigma}_{11}$ and $\boldsymbol{\mu}_2$ and $\boldsymbol{\Sigma}_{22}$ respectively. A sketch of the proof is provided below.

$$p(x_1) = \int_{x_2} N(x|\boldsymbol{\mu}, \boldsymbol{\Sigma}) dx_2,$$

which can be written as

$$0.5 \int_{x_2} \exp\left[(x_1 - \boldsymbol{\mu}_1)^T \boldsymbol{\Sigma}_{11}^{-1}(x_1 - \mu_1) + 2(x_1 - \boldsymbol{\mu}_1)^T \boldsymbol{\Sigma}_{12}^{-1}(x_2 - \mu_2) + (x_2 - \boldsymbol{\mu}_2)^T \boldsymbol{\Sigma}_{22}^{-1}(x_2 - \mu_2)\right] dx_2.$$

Note that this equals

$$p(x_1) \int_{x_2} p(x_2|x_1) dx_2,$$

implying that $X_1 \sim \mathcal{N}(\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_{11})$.
While the marginals have a simple form, the conditionals are more complicated. (For a complete derivation, which requires matrix inversion lemmas, refer to Murphy.) It can be shown that $X_1|X_2 \sim \boldsymbol{\mu}_{\infty|\in}, \boldsymbol{\Sigma}_{\infty|\in}$ with

$$\boldsymbol{\mu}_{1|2} = \boldsymbol{\mu}_1 + \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}(x_2 - \boldsymbol{\mu}_2), \quad \boldsymbol{\Sigma}_{1|2} = \boldsymbol{\Sigma}_{11} - \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{21}.$$

3.3.4 Information Form

An alternative formulation, called information form, uses the precision matrix (inverse variance) $\Lambda = \Sigma^{-1}$. Partitioning Λ as

$$\boldsymbol{\Lambda} = \begin{pmatrix} \boldsymbol{\Lambda}_{11} & \boldsymbol{\Lambda}_{12} \\ \boldsymbol{\Lambda}_{21} & \boldsymbol{\Lambda}_{22} \end{pmatrix},$$

the covariance matrices of the conditional distributions have a simple form. For example, the covariance matrix of X_1 given X_2 is $\Lambda_{1|2} = \Lambda_{11}$. However, the simplicity of the conditional precision comes at the cost of marginalization (which was simple when using Σ) becoming a more complicated expression (see Murphy subsection 4.3 for more details).

[2]It is left as an exercise to show that this formula holds. Hint: Use the fact that we assumed no covariance.

CS281: Advanced ML **September 13, 2017**

Lecture 4: Linear Regression

Lecturer: Sasha Rush *Scribes: Kojin Oshiba, Michael Ge, Aditya Prasad*

4.1 Multivariate Normal (MVN)

The multivariate normal distribution of a D-dimensional random vector X is defined as:

$$N(X|\mu,\Sigma) \sim (2\pi)^{-\frac{D}{2}}|\Sigma|^{-\frac{1}{2}}\exp\left(-\frac{1}{2}(X-\mu)^T\Sigma^{-1}(X-\mu)\right)$$

Note:

- $(2\pi)^{-\frac{D}{2}}|\Sigma|^{-\frac{1}{2}}$ and $-\frac{1}{2}$ are constants we can ignore in MLE and MAP calculations.
- $(X-\mu)^T\Sigma^{-1}(X-\mu)$ is a quadratic term.

There are three types of inference we're interested in doing: MLE, MAP, and prediction.

4.2 Maximum Likelihood of MVN

Let $\theta = (\mu, \Sigma)$, where Σ can be approximated as a diagonal/low rank matrix. If there are $x_1, \ldots, x_n$ observations, the MLE estimate of μ is

$$\begin{aligned}\mu^* &= \arg\max_{\mu} -\sum_n logN(x_n|\mu,\Sigma) \\ &= \arg\max_{\mu} \quad log(constant) - \sum_n (x_n-\mu)^T\Sigma^{-1}(x_n-\mu) \\ &= \arg\max_{\mu} -\sum_n (x_n-\mu)^T\Sigma^{-1}(x_n-\mu)\end{aligned}$$

Let $L = \sum_n (x_n-\mu)^T\Sigma^{-1}(x_n-\mu)$.

$$\begin{aligned}\frac{dL}{d\mu} &= \Sigma_n\Sigma^{-1}(x_n-\mu) = 0 \\ &\Leftrightarrow \mu^*_{MLE} = \frac{\Sigma_{X_n}}{N}\end{aligned}$$

Similarly,

$$\begin{aligned}\frac{dL}{d\Sigma} &= (exercise) = 0 \\ &\Leftrightarrow \Sigma^*_{MLE} = \frac{1}{N}\sum_n x_n x_n^T = \frac{1}{N}X^\top X\end{aligned}$$

For calculating $\frac{dL}{d\Sigma}$ as an exercise , the following might be helpful:

- $\frac{d}{dA}ln|A| = A^{-1}$
- $\frac{d}{dA}tr(BA) = B^T$
- $tr(ABC) = tr(BCA)$

4.3 Linear-Gaussian Models

Let x be a vector of affine, noisy observations with a prior distribution:

$$x \sim N(m_0, S_0)$$

Let y be the outputs:

$$y|x \sim N(Ax + b, \Sigma_y)$$

4.3.1 $p(x|y)$

We are interested in calculating the posterior distribution: $p(x|y)$.

$$\begin{aligned} p(x|y) &\propto p(x)p(y|x) \\ &= \frac{1}{2}\exp\begin{cases}(x-m_0)^\top S_0^{-1}(x-m_0) \\ +(y-(Ax+b))^\top \Sigma_y^{-1}(y-(Ax+b))\end{cases} \\ &= \frac{1}{2}\exp\begin{cases}\underline{x^\top S_0^{-1}x^{\star\star}} - \underline{2x^\top S_0^{-1}m_0}^\star + \dots \\ \underline{+x^\top(A^\top\Sigma_y^{-1}A)x^{\star\star}} - \underline{2x^\top(A^\top\Sigma_y^{-1})y^\star} + \underline{2x^\top(A^\top\Sigma_y^{-1})b^\star} + \dots\end{cases} \end{aligned}$$

The terms containing x are underlined. Double-starred ($\star\star$) terms are quadratic in x, while single-starred ($\star$) terms are linear in x. The remaining terms are constants that are swallowed up by the proportionality. By Gaussian-Gaussian conjugacy, we know the resulting distribution should be Gaussian. To find the parameters, we'll modify $p(x|y)$ to fit the form of a Normal. This requires completing the square!

4.3.2 Completing the Square

$$ax^2 + bx + c \to a(x-h)^2 + k, h = \frac{-b}{2a}, k = c - \frac{b^2}{4a}$$

We ignore the k term since it too is swallowed up in the proportionality. In application to our problem, we group the quadratic and linear terms together to calculate our terms for completing the square.

- "a" is $S_N^{-1} = S_0^{-1} + A^\top\Sigma_y^{-1}A$
- "h" is $m_N = S_N\left[S_0^{-1}m_0 + A^\top\Sigma_y^{-1}(y-b)\right]$

In this more "intuitive" representation, we find that $p(x|y)$ has the form of $N(m_N, S_N)$. Murphy also has a more explicit representation:

- $\Sigma_{x|y} = \Sigma_x^{-1} + A^\top\Sigma_y^{-1}A$
- $\mu_{x|y} = \Sigma_{x|y}[\Sigma_x^{-1}\mu_x + A^\top\Sigma_x^{-1}(y-b)]$

4.3.3 p(y)

We now calculate the normalizer term, $p(y)$. Now, x is fixed. y follows the linear model:

$$y = Ax + b + \epsilon$$

The result is that y follows a Normal distribution with the following form:

$$p(y) = N(y|Am_0 + b, \Sigma_y + A\Sigma_x A^\top)$$

4.3.4 Prior (just for μ)

$$p(\mu) = N(\mu|m_0, S_0)$$

where m_0, S_0 are pseudo mean, pseudo variance. $p(\mu)$ is defined Gaussian because Gaussian is the conjugate prior of itself. A prior is called a conjugate prior if it has the same distribution as the posterior distribution.

4.3.5 Posterior (just for μ)

$$p(\mu|X) \propto p(\mu)p(X|\mu) = N(\mu|m_0, s_0)N(X|\mu, \Sigma)$$

This is a special case of linear regression. Recall,

- "a" is $S_N^{-1} = S_0^{-1} + A^\top \Sigma_y^{-1} A$
- "h" is $m_N = S_N \left[S_0^{-1} m_0 + A^\top \Sigma_y^{-1} (y - b) \right]$

We let $b = 0$ and $A = I$. Then we obtain,

$$S_N^{-1} = S_0^{-1} + \Sigma^{-1}$$

$$m_N = S_N[S_0^{-1} m_0 + \Sigma^{-1} X]$$

Hence,

$$p(\mu|X) = N(m|m_N, S_N)$$

4.3.6 Unknown Variance

Similar to μ, we can also define a conjugate prior on Σ, which is Inverse Wishart distribution. It is defined as:

$$IW(\Sigma|S, \nu) = \frac{1}{2}|\Sigma|^{-(\nu-(D+1)/2)} exp\{\frac{1}{2} tr(S^{-1}\Sigma^{-1})\}$$

- distribution over positive semi definite Σ with two parameters S, ν.
- pseudo info $S = \Sigma X X^T$ is a psuedo scatter matrix called the scale matrix. $\nu = n\mu$ is degrees of freedom where $\nu - (D + 1)$ is the number of observations.

4.4 Linear Regression

In an undergraduate version of the class, we might define the problem as follows: We are given "fixed" set of inputs, $\{\boldsymbol{x}_i\}$. We want to "predict" the outputs.

Here, we define the problem as attempting to compute $p(y \mid \boldsymbol{x}, \boldsymbol{\theta})$. Consider the following example. We assume that our data is generated as follows:

$$y = \boldsymbol{w}^T \boldsymbol{x} + \text{noise}$$

Further, we assume that the noise (denoted by ϵ) is distributed as Gaussian with mean 0; that is:

$$\epsilon \sim \mathcal{N}(0, \sigma^2)$$

Then, we have:

$$p(y \mid \boldsymbol{x}, \theta) = \mathcal{N}(y \mid \boldsymbol{w^T x}, \sigma^2)$$

Note that the bias term here is included as a dimension in $\boldsymbol{w}$, w_0.

4.4.1 Log Likelihood

Consider a data-set that looks like $\{(\boldsymbol{x_i}, y_i)\}_{i=1}^N$. The log-likelihood $\mathcal{L}(\boldsymbol{\theta})$ is given by:

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}) &= \log p(\text{data} \mid \boldsymbol{\theta}) \\ &= \sum_{n=1}^{N} \log p(y_n \mid \boldsymbol{x_n}, \boldsymbol{\theta}) \\ &= \sum_{n=1}^{N} \log(\text{constant}) - \frac{1}{2\sigma^2}(y_n - \boldsymbol{w}^T \boldsymbol{x_n})^2\end{aligned}$$

Note that data here refers to just the y_i's. The y_n's are called the target; the $\boldsymbol{w}$ represents the weights; and the x_n's are the observations. The term $(y_n - \boldsymbol{w}^T \boldsymbol{x_n})^2$ is essentially just the residual sum of squares.

4.4.2 Computing MLE

We want the argmax of the log-likelihood. We therefore have:

$$\begin{aligned}\text{argmax}_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}) &= \text{argmax} - \sum_{n=1}^{N} \frac{1}{2\sigma^2}(y_n - \boldsymbol{w}^T \boldsymbol{x_n})^2 \\ &= \text{argmax}_{\boldsymbol{w}} - [\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}]^T [\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}] \\ &= \text{argmax}_{\boldsymbol{w}} \left[\boldsymbol{w}^T \boldsymbol{X^T X} \boldsymbol{w} - 2\boldsymbol{w}^T \boldsymbol{X^T y} + \text{constant}\right]\end{aligned}$$

There is an analytical solution to this, and we obtain it by simply computing the gradient and setting it to $\boldsymbol{0}$.

$$\partial_w \left[\boldsymbol{w}^T \boldsymbol{X^T X} \boldsymbol{w} - 2\boldsymbol{w}^T \boldsymbol{X^T y}\right] = 2\boldsymbol{X^T X}\boldsymbol{w} - 2\boldsymbol{X^T y}$$

Setting this to $\boldsymbol{0}$, we obtain:

$$\boldsymbol{w}_{MLE} = (\boldsymbol{X}^T \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{y}$$

As we will see in homework 1, $(\boldsymbol{X}^T \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{y}$ can be viewed as the projection of $\boldsymbol{y}$ onto the column space of $\boldsymbol{X}$.

4.5 Bayesian Linear Regression

In the Bayesian framework, we also introduce a probability distribution on the weights. Here, we choose:

$$p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w} \mid \boldsymbol{m_0}, \boldsymbol{S_0})$$

Thus, we have:

$$p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w}, \boldsymbol{\mu}, \sigma^2) = \mathcal{N}(\boldsymbol{y} \mid \boldsymbol{\mu} + \boldsymbol{X^T w}, \sigma^2 \boldsymbol{I})$$

We assume that $\boldsymbol{\mu} = 0$.
The posterior then is of the form:

$$p(\boldsymbol{w} \mid \ldots) \propto \mathcal{N}(\boldsymbol{w} \mid \boldsymbol{m_0}, \boldsymbol{S_0}) \mathcal{N}(\boldsymbol{y} \mid \boldsymbol{X}^T \boldsymbol{w}, \sigma^2 \boldsymbol{I})$$

Applying the results obtained above with the linear Gaussian results, with:

$$\begin{aligned} b &= 0 \\ \boldsymbol{A} &= \boldsymbol{X}^T \\ \boldsymbol{\Sigma_y} &= \sigma^2 \boldsymbol{I} \end{aligned}$$

Thus, we have:

$$\boldsymbol{S_N}^{-1} = \boldsymbol{S_0}^{-1} + \frac{1}{\sigma^2}\boldsymbol{X^T X}$$

$$\boldsymbol{m_N} = \boldsymbol{S_N}\left[\boldsymbol{S_0^{-1}}\boldsymbol{m_0} + \boldsymbol{X^T y}\frac{1}{\sigma^2}\right]$$

Now, we compute the posterior predictive:

$$p(y \mid \boldsymbol{x}, y) = \int \mathcal{N}(y \mid \boldsymbol{w}^T\boldsymbol{x}, \sigma^2)\mathcal{N}(\boldsymbol{w} \mid \boldsymbol{m_N}, \boldsymbol{S_N})d\boldsymbol{w}$$

Using the form for the marginal derived earlier, we have:

$$p(y \mid \boldsymbol{x}, y) = \mathcal{N}(y \mid \boldsymbol{X^T m_N}, \sigma^2 + \boldsymbol{X^T S_0 X})$$

The variance term is particularly interesting because now the variance has dependence on the actual data; thus, the Bayesian method has thus produced a different result. The mean, however, is the same as the MAP estimate ($\boldsymbol{x}^T\boldsymbol{m_N}$)

4.6 Non Linear Regression

All the examples done so far have been in linear space. To define an adaptive basis, we simply transform point x with the transormation of our choice:

$$x \to \phi(x)$$

Examples include:

- $\phi_1(x) = \sin(x)$
- $\phi_2(x) = \sin(\lambda x)$
- $\phi_3(x) = \max(0, x)$
- $\phi(x; w) = \max(0, w' \top x)$

The last example is the core of neural networks and deep learning where the weights are learned for each level of w.

CS281: Advanced ML **September 18, 2017**

Lecture 5: Linear Classification

Lecturer: Sasha Rush *Scribes: Demi Guo, Artidoro Pagnoni, Luke Melas-Kyriazi, Florian Berlinger*

5.1 Classification Introduction

Last time we saw linear regression. In linear regression we were predicting $y \in \mathbb{R}$, in classification instead we deal with a discrete set, for example $y \in \{0, 1\}$ or $y \in \{1, \ldots, C\}$. This distinction only matters for this lecture, starting from next class we will generalize the topics and treat them as the same thing.

Among the many applications, linear classification is used in sentiment analysis, spam detection, and facial and image recognition. We will use generative models of the data, which means that we will model both the x and the y explicitly, and we are not keeping x fixed. In the case of the spam filter earlier, x is the email body, and y is the label {spam, not spam}. A generative model of the email and labels, we would model the distribution of x, of the text in the email itself, and not only the distribution of the category y.

We will explore the basic method of Naïve Bayes in detail. Even with a very simple method like Naïve Bayes with basic features it is possible to perform extremely well on many classification tasks when large training data sets are available. For example, this simple model performs almost as well (one percent point difference) as very complex methods on spam detection.

5.2 Naïve Bayes

Note that the term "Bayes" in Naïve Bayes (NB) does not have to do with Bayesian modeling, or the presence of priors on parameters. We won't have any priors for the moment. General Naïve Bayes takes the following form:

$$\begin{aligned} y &\sim \text{Cat}(\pi) && \text{[class distribution]} \\ \boldsymbol{x}|y &\sim \prod_j p(x_j \mid y) && \text{[class conditional]} \end{aligned}$$

where y is the class label and comes from a categorical distribution, and x_j is a dimension of the input x.

In Naïve Bayes, the form of the class distribution is fixed and parametrized independently from the class conditional distribution. The "Naïve" term in "Naïve Bayes" precisely refers to the conditional independence between y and $x_j|y$. Depending of what the data looks like we can choose a different form for the class conditional distribution.

Here we present three possible choices for the class conditional distribution:

- **Multivariate Bernoulli Naïve Bayes**:

  $$x_j|y \sim \text{Bern}(\mu_{jc}) \quad \text{if } y = c$$

  Here y takes values in a set of classes, and μ_{jc} is a parameter associated with a specific feature (or dimension) in the input and a specific class. We use multivariate Bernoulli when we only allow two possible values for each feature, therefore $x_j|y$ follows a Bernoulli distribution.

  We can think of x as living in a hyper cube, with each dimension j having an associated μ for each class c. From here we get the name multivariate Bernoulli distribution.

- **Categorical Naïve Bayes**:

  $$x_j|y \sim \text{Cat}(\mu_{jc}) \quad \text{if } y = c$$

  We use the Categorical Naïve Bayes when we allow different classes for each feature j, so $x_j|y$ follows a Categorical distribution.

- **Multivariate Normal Naïve Bayes**

$$\boldsymbol{x}|y \sim \mathcal{N}(\boldsymbol{\mu}_c, \boldsymbol{\Sigma}^c_{diag})$$

Note that here we use $\boldsymbol{x}$ vector and not a specific feature. Since we impose that $\boldsymbol{\Sigma}^c$ is a diagonal matrix, we have no covariance between features, so this comes down to having an independent normal for each feature (or dimension) of the output. This is also required by the "Naïve" assumption of conditional independence. We would use MVN Naïve Bayes when the features take continuous values in $\mathbb{R}^n$.

5.3 General Naïve Bayes

We consider the data points $\{(\boldsymbol{x}_n, y_n)\}$, without specifying a particular generative model. The likelihood of each data point is:

$$p(\boldsymbol{x}_n, y_n|\text{param}) = p(y_n|\boldsymbol{\pi}) \prod_j p(x_{nj}|y_n, \text{param}) \tag{5.1}$$

$$= \prod_c \pi_c^{(y_n=c)} \prod_j \prod_c p(x_{nj}|y_n)^{(y_n=c)} \tag{5.2}$$

where in equation (5.2) we assume conditional independence (the "Naïve" assumption). The term $p(x_{nj}|y_n)$ depends on the generative model used for x and also on the class y_n.

We can then solve for the parameters maximizing the likelihood, which is equivalent to maximizing the log likelihood.

$$(\pi_{\text{MLE}}, \mu_{\text{MLE}}) = \underset{(\boldsymbol{\pi},\boldsymbol{\mu})}{\text{argmax}} \sum_n \log p(\boldsymbol{x}_n, y_n|\text{param}) \tag{5.3}$$

$$= \underset{(\boldsymbol{\pi},\boldsymbol{\mu})}{\text{argmax}} \sum_c N_c \log \pi_c + \sum_i \sum_c \sum_{n:y_n=c} \log p(x_{nj}|y_n) \tag{5.4}$$

$$= \left(\underset{(\boldsymbol{\pi},\boldsymbol{\mu})}{\text{argmax}} \sum_c N_c \log \pi_c \right) + \left(\underset{(\boldsymbol{\pi},\boldsymbol{\mu})}{\text{argmax}} \sum_i \sum_c \sum_{n:y_n=c} \log p(x_{nj}|y_n) \right) \tag{5.5}$$

Where $N_c = \sum_n \mathbb{1}(y_n = c)$, and $N =$ the number of data points.

This factors into two parts (5.10), the first only depending on $\boldsymbol{\pi}$ the other is the MLE for the class condition distribution on each feature or dimension of the input. This factorization allows to solve for the maximizing $\boldsymbol{\pi}$ and the maximizing parameters for the class conditional separately.

For example, if we use a Multivariate Bernoulli Naïve Bayes generative model we would get the following parameters from MLE:

$$\pi_c = \frac{N_c}{N} \tag{5.6}$$

$$\mu_{jc} = \frac{\sum_{n:y_n=c} x_{nj}}{N_c} = \frac{N_{cj}}{N_c} \tag{5.7}$$

Again, where $N_c = \sum_n \mathbb{1}(y_n = c)$, $N_{cj} = \sum_n \mathbb{1}(y_n = c)x_{nj}$ and $N =$ number of data points.

5.4 Bayesian Naive Bayes: Add a Prior

Here, instead of working with a single distribution, we are working with multiple distributions. For simplicity, let's use the following factored **prior**:

$$p(\pi, \mu) = p(\pi) \prod_j \prod_c p(\mu_{jc})$$

where $p(\pi)$ represents the prior on class distribution and $\prod_j \prod_c p(\mu_{jc})$ represents prior on class conditional distribution.

Now, **what prior should we use?**

1. π: Dirichlet (goes with Categorical)
2. μ_{jc}:
   (a) Beta (goes with Bernoulli)
   (b) Dirichlet (goes with Categorical)
   (c) Normal (goes with Normal)
   (d) Inverse-Wishart (Iw) (goes with Normal)

   Here, what distribution we choose depends on our choice of class conditional distribution.

Recall that we want to use conjugate priors to have a natural update (that's why we pair them up!). By using conjugate priors, we will have:

$$p(\pi|\text{data}) = Dir(N_1 + \alpha_1, \cdots, N_c + \alpha_c)$$
$$p(\mu_{jc}|\text{data}) = \beta((N_c - N_{jc}) + \beta_0, N_c + \beta_1)$$

5.4.1 Intuition

You can think of the α_i above as initial pseudocounts. Those pseudocounts give nonzero probability to features we haven't seen before, which is crucial for NLP. For unseen features, you could have a pseudocount of 1 or 0.5 (Laplace term) or something.
Because of this property, a Bayesian model helps prevents overfitting by introducing such priors: consider the spam email classification problem mentioned before. Say the word "subject" (call it feature j) always occurs in both classes ("spam" and "not spam"), so we estimate $\hat{\theta}_{jc} = 1$ (we overfit!) What will happen if we encounter a new email which does not have this word in it? Our algorithm will crash and burn! This is another manifestation of the black swan paradox discussed in Book Section 3.3.4.1. Note that this will not happen if we introduce pseudocounts to all features!

5.5 Posterior Predictive

$$p(\hat{y}, \hat{x} \mid \text{data}) = (\text{integrate over parameters})$$
$${\pi_c}^{\text{MAP}} = \frac{N_c + \alpha_c}{N + \sum_c \alpha_c} (\text{Dirichlet MAP})$$
$${\mu_{jc}}^{\text{MAP}} = \frac{N_{jc} + \beta_1}{N_c + \beta_1 + \beta_0} (\text{Beta MAP})$$

(How to derive this? Good exercise!)

5.6 More on Predictive

Now, let's consider a little bit more about what's happening in our predictive. Consider the email spam classification problem: given some features of an email, we want to predict if the email is a spam or not a spam. We have:

$$\begin{aligned} p(y = c|x, data) &\propto \pi_c \prod_j p(x_j|y) \text{ (try to generate observations from class)} \\ &= \pi_c \prod_j \mu_{jc}^{x_j} (1 - \mu_{jc})^{(1-x_j)} \text{ (informal parametrization)} \\ &= \exp(\log \pi_c + \sum_j x_j \log \mu_{jc} + (1 - x_j) \log(1 - \mu_{jc})) \text{ (take exp of log)} \\ &= \exp\left(\log \pi_c + \sum_j \log(1 - \mu_{jc}) + \sum_j x_j \log \frac{\mu_{jc}}{1 - \mu_{jc}}\right) \end{aligned}$$

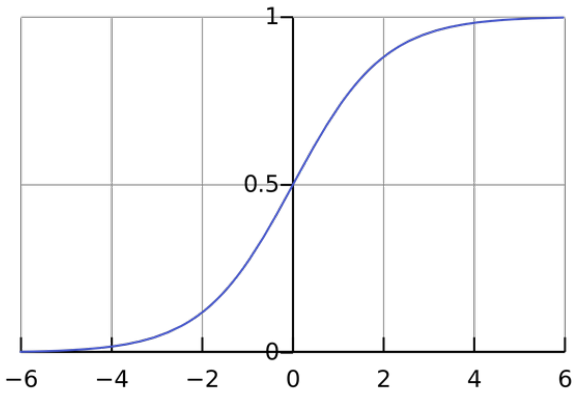


Figure 5.2: The Sigmoid Function

where the first two term $log\pi_c + \sum_j log(1 - \mu_{jc})$ is a constant (we call it b for bias), and the last term $\sum_j x_j log\frac{\mu_{jc}}{1-\mu_{jc}}$ is linear (we call it θ).

5.7 Multivariate Bernoulli Naive Bayes

For Multivariate Bernoulli NB, we will have:

$$\theta_{jc} = \log \frac{\mu_{jc}}{1 - \mu_{jc}}$$
$$b_c = \log \pi_c + \log(1 - \mu_{jc})$$

So, we have:

$$p(y = c \mid x) \propto \exp(\theta_c^T x + b_c)$$

Thus, in order to determine which class ("spam" or "not spam"), for each class we simply compute a linear function with respect to x, and compare the two. Our $\theta x + b$ is going to be associated with a linear separator of the data. Even better, for prediction, we can simply compute θ and β (as shown above) using closed form for both MAP and MLE cases.

5.8 The Sigmoid Function

Before proceeding, we should name our variables to speak about them more easily.
We call μ the "informal parameters" and θ the "scores." In the case of a Multivariate Bernouli model, we have the map $\theta_{jc} = \log \frac{\mu_{jc}}{1-\mu_{jc}}$, which we call the "log odds." We may also invert this relationship to find μ as a function of θ:

$$\theta = \log \frac{\mu}{1 - \mu} \implies \mu = \frac{e^\theta}{1 + e^\theta} = \frac{1}{1 + e^{-\theta}} = \sigma(\theta)$$

We denote this function $\sigma(\theta)$ as the sigmoid function. The sigmoid function is a map from the real line to the interval $[0, 1]$, so is useful as a representation of probability. It is also a common building block in constructing neural networks, as we will see later in the course.

5.9 The Softmax Function

We will now return to the predictive $p(y = c|x) \propto \exp(\theta_c^T x + c_c)$ to try to compute the normalizer Z:

$$p(y = c|x) = \frac{1}{Z} \exp(\theta_c^T x + b_c)$$

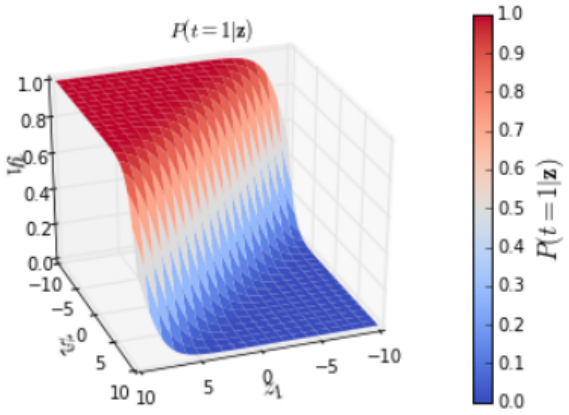


Figure 5.3: The Softmax Function

In general, we can compute the normal by summing over all our classes.

$$Z(\theta) = \sum_{c'} = \exp(\theta_{c'}^T x + b_{c'})$$

In practice, this summation is often computationally expensive. However, it is not necessary to compute this sum if we are only interested in the most likely class label given an input.
We call the resulting probability density function the *softmax*:

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_{i'} \exp(z_{i'})}$$

This function generalizes the sigmoid function to multiple classes/dimensions. We call it the "softmax" because we may think of it as a smooth, differentiable version of the function which simply returns 1 for the most likely class (or argmax).

5.10 Discriminative Classification

We may apply the mathematical tools developed in the generative classification setting discussed above to perform discriminative classification. In discriminative classification, we assume that our inputs x are fixed, rather than coming from some probability distribution.
We take the maximum likelihood estimate, as in linear regression, given that $p(y = c|x) \propto \exp(\theta_c^T x)$:

$$\text{MLE} : \underset{\theta}{\text{argmax}}\, p(y|x, \theta) = \underset{\theta}{\text{argmax}} \sum_n \log \text{softmax}(\theta_c^T x_n) c_n$$

What are the advantages and disadvantages of this approach? The primary disadvantage compared to methods we have seen earlier is that this maximum likelihood estimate has *no closed form*. It is also not clear how we might incorporate our prior (although there is recent work in this area). On the other hand, this equation is convex and it is easy (at least mathematically, not necessarily computationally) to compute gradients, so we may use gradient descent.

$$\frac{d(\cdot)}{d\theta_c} = \sum_n x_n \cdot \begin{cases} 1 - \text{softmax}(\theta_c^T x) & \text{if } y_n = c \\ \text{softmax}(\theta_c^T x) & \text{otherwise} \end{cases} \tag{5.8}$$

This model is known as **logistic regression** (even though it is used for classification, not regression) and is widely used in practice.

More Resources on Optimization

- Convex Optimization by Lieven Vandenberghe and Stephen P. Boyd

CS281: Advanced ML **September 20, 2017**

Lecture 6: Exponential Families

Lecturer: Sasha Rush *Scribes: Meena Jagadeesan, Yufeng Ling, Tomoka Kan, Wenting Cai*

6.1 Introduction

(Wainwright and Jordan (textbook) presents a more detailed coverage of the material in this lecture.)

This lecture, we will unify all of the fundamentals presented so far:

$p(\theta)$	$p(x)$	$p(y \mid x)/p(x,y)$
Beta, Dir	Discrete	Classification
MVN, IW	MVN	Linear Regression
	Exponential Families	**Generalized Linear Models**
	Undirected Graphic Models	Conditional UGM
	Variational Inference	

We will focus on coming up with a general form for Discrete and MVN through exponential families. We will also come up with a general form for classification and linear regression through generalized linear models.

6.2 Definition of Exponential Family

The definition is

$$\begin{aligned} p(x \mid \theta(\mu)) &= \frac{1}{Z(\theta)} h(x) \exp\{\theta^T \phi(x)\} \\ &= h(x) \exp \theta^T \phi(x) - A(\theta) \end{aligned}$$

where

μ	mean parameters
$\theta(\mu)$	natural / canonical / exponential parameters
$Z(\theta) A(\theta)$	also written as $Z(\theta(\mu))$ or $Z(\mu)$, the partition function and log partition
$\phi(x)$	sufficient statistics of x, potential functions, "features"
$h(x)$	scaling term, in most cases, we have $h(x) = 1$

Note that there is "minimal form" and "overcomplete form".

6.3 Examples of Exponential Families

6.3.1 Bernoulli/Categorical

First, we consider the Bernoulli as an exponential family. Like last lecture, we rewrite the distribution as an exp of log.

$$\begin{aligned} \text{Ber}(x|\mu) &= \mu^x (1-\mu)^{(1-x)} \\ &= \exp x \log \mu + (1-x) \log(1-\mu) \\ &= \underbrace{}_{h(x)} \exp \underbrace{\log\left(\frac{\mu}{1-\mu}\right)}_{\theta} \underbrace{x}_{\phi(x)} + \underbrace{\log(1-\mu)}_{-A(\mu)} \end{aligned}$$

For the **minimal form**, we have

$$\begin{aligned} h(x) &= 1 \\ \phi_1(x) &= x \\ \theta_1(\mu) &= \log \frac{\mu}{1-\mu} (\text{“log odds”}) \\ \mu &= \sigma(\theta) \\ A(\mu) &= -\log(1-\mu) \\ A(\theta) &= -\log(1-\sigma(\theta)) = \theta + \log(1+e^{-\theta}) \end{aligned}$$

For the **overcomplete form**, we have

$$\begin{aligned} \phi(x) &= \begin{bmatrix} x \\ 1-x \end{bmatrix} \\ \theta &= \begin{bmatrix} \log \mu \\ \log(1-\mu) \end{bmatrix} \end{aligned}$$

For the Categorical/Multinouilli distribution, we have

$$\theta = \begin{bmatrix} \log \mu_1 \\ \vdots \\ \log \mu_n \end{bmatrix}$$

where $\sum_c \mu_c = 1$.
Side note: Writing out in overcomplete form usually comes with some restraints.

6.3.2 Univariate Gaussians

$$\begin{aligned} \mathcal{N}(x \mid \mu, \sigma^2) &= (2\pi\sigma^2)^{1/2} \exp\{-\frac{1}{2\sigma^2}(x-\mu)^2\} \\ &= \underbrace{(2\pi\sigma^2)^{-\frac{1}{2}}}_{A(\mu,\sigma^2)} \exp\{\underbrace{-\frac{1}{2\sigma^2}x^2 + \frac{\mu}{\sigma^2}x}_{\theta^T\phi(x)} - \underbrace{\frac{1}{2\sigma^2}\mu^2}_{A(\mu,\theta^2)}\} \end{aligned}$$

$$\begin{aligned} \phi(x) &= \begin{bmatrix} x \\ x^2 \end{bmatrix} \\ \theta &= \begin{bmatrix} \frac{\mu}{\sigma^2} \\ -\frac{1}{2\sigma^2} \end{bmatrix} \\ A(\mu, \sigma^2) &= \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}\mu^2 \\ \mu &= -\frac{\theta_1}{2\theta_2} \\ \sigma^2 &= -\frac{1}{2\theta_2} \\ A(\theta) &= -\frac{1}{2}\log(-2\theta_2) - \frac{\theta_1^2}{4\theta_2} \end{aligned}$$

6.3.3 Bad distributions

Two simple distributions that do not fit this form are the uniform distribution Uniform$(0,1)$ (check this as an exercise), and the Student-T distribution.

6.4 Properties of Exponential Families

Most inference problems involve a mapping between natural parameters and mean parameters, so this is a natural framework.
Here are three properties of exponential families:

Property 1 Derivatives of $A(\theta)$ provide us the cumulants of the distribution $\mathbb{E}(\phi(x))$, $\text{var}(\phi(x))$:

Proof. For univariate, first order:

$$\begin{aligned}
\frac{dA}{d\theta} &= \frac{d}{d\theta}(\log Z(\theta)) \\
&= \frac{d}{d\theta}\log \underbrace{\left(\int \exp\{\theta\phi\}h(x)dx\right)}_{\text{needed to integrate to 1}} \\
&= \frac{\int \phi \exp\{\theta\phi\}h(x)dx}{\int \exp(\theta\phi)h(z)dx} \\
&= \frac{\int \phi \exp\{\theta\phi\}h(x)dx}{\exp(A(\theta))} \\
&= \int \phi(x)\underbrace{\exp(\theta\phi(x) - A(\theta))h(x)}_{p(x)}dx \\
&= \int \phi(x)p(x)dx \\
&= \mathbb{E}(\phi(x))
\end{aligned}$$

The same property holds for multivariates (refer to textbook for proof). □

Bernoulli:

$$A(\theta) = \theta + \log(1 + e^{-\theta})$$
$$\frac{dA}{d\theta} = 1 - \frac{e^{-\theta}}{1+e^{-\theta}} = \underbrace{\frac{1}{1+e^{-\theta}}}_{\text{sigmoid}} = \sigma(\theta) = \mu$$

Univariate Normal Left as exercise.

Property 2 MLE has a nice form (through "moment matching")

Proof.

$$\begin{aligned}
\underset{\theta}{\text{argmax}} \log p(\text{data} \mid \theta) &= \underset{\theta}{\text{argmax}} \left(\sum_d \theta^T \phi(x_d)\right) - NA(\theta) \\
&= \underset{\theta}{\text{argmax}} \theta^T \underbrace{\left(\sum_d \phi(x_d)\right)}_{\text{sum of sufficient statistics}} - \underbrace{NA(\theta)}_{\text{amount of points}}
\end{aligned}$$

We take a derivative to obtain:

$$\begin{aligned}
\frac{d(.)}{d\theta} &= \sum_d \phi(x_d) - N\frac{dA(\theta)}{d\theta} \\
&= \sum_d \phi(x_d) - N\mathbb{E}(\phi(x)) \\
&= 0
\end{aligned}$$

$$E(\phi(x)) = \underbrace{\frac{\sum \phi(x_d)}{N}}_{\text{set mean parameter to sample means that gives us MLE}}$$

□

Property 3 Exponential families have conjugate priors.

Proof. We first introduce some notations.

$$\begin{aligned}
&\eta \text{ - parameters} \\
&\bar{s} = \sum_d \phi(x_d)/N \\
&p(\text{data} \mid \eta) \propto \exp[(N\bar{s})\eta - NA(\eta)] \\
&p(\eta \mid N_0, s_0) \propto \exp[(N_0, \bar{s}_0)\eta - N_0 \underbrace{A(\eta)}_{\text{not log partition, which has to be a function strictly of parameters}}] \\
&p(\eta|\text{data}) \propto \exp((N\bar{s} + N_0\bar{s}_0)^T\eta - (N_0 + N)A(\eta))
\end{aligned}$$

The above two distributions have the same sufficient statistics – so we have a conjugate prior. It also tells us that it is not a coincidence that we kept obtaining pseudo counts. (More references will be put up to describe this).

□

6.5 Definition of Generalized Linear Models

While exponential families generalize $p(x)$, GLMs generalize $p(y|x)$.

$$p(y|x, w) = h(y)\exp\{\theta(\underbrace{\mu(x)}_{\text{predict mean}})^T\phi(y) - A(\theta)\}$$

where $\mu(x) = \underbrace{g^{-1}}_{\text{squashing const}}(w^Tx + b)$ where g is an appropriate linear transformation.
This can be summarized through the following sequence of transformations:

$$x \overset{g^{-1}(w^Tx+b)}{\longrightarrow} \mu \to \theta \to p(y \mid x).$$

6.6 Examples of Generalized Linear Models

We present three examples:

Example 1 Exponential family - Normal distribution with $\sigma^2 = 1$ and g^{-1} is the identity function. This gives us the linear regression

$$\mu = w^Tx + b \qquad \mathbb{R} \to \mathbb{R}.$$

Example 2 Exponential family - Bernoulli distribution and g^{-1} is the sigmoid function $\sigma : \mathbb{R} \to (0, 1)$. Now, $\mu = \sigma(w^Tx + b)$ and $\theta = \log\left(\frac{\mu}{1-\mu}\right)$. This is how we define logistic regression. This gives us

$$p(y \mid x) = \sigma(w^Tx + b)^y(1 - \sigma(w^Tx + b))^{1-y}$$

Example 3 Exponential family - Categorical distribution with g^{-1} as the softmax function.
$\mu_c = \text{softmax}(w_c^Tx + b_c)_c$
$\theta_c = \log \mu_c$

CS281: Advanced ML **September 25, 2017**

Lecture 7: Neural Networks

Lecturer: Sasha Rush *Scribes: Juntao Wang, Alexander Wei, Kevin Zhang, Aron Szanto*

7.1 Introduction

Neural networks have been a hot topic recently in machine learning. But everything we will cover today has essentially been known since '70s and '80s. Since then, there has been in increased focus on this subject due to its successes after improved computing power, larger datasets, better neural network architectures, and more careful study in academia. Neural networks have also seen wide adoption in industry in recent years. Lately, there has also been work trying to integrate other methods of inference into neural networks—we will take a look at this topic later in this course. We cover neural networks now as a tangentially-related introduction to graphical models and as an example of combining traditional inference with deep models.

7.1.1 Review of General Linear Models

In the last lecture, we saw that we can learn a mapping between mean parameters and natural parameters for many general linear models and use squashing functions to change natural parameters into mean parameters. In general, we describe these models using a transformation $\mu = g^{-1}(w^\top x + b)$ for some function g and a distribution $y \mid x$.

Example 1 (Linear Regression)**.** Here we have $g(x) = x$ (the identity function) with $y \mid x \sim \mathcal{N}(w^\top x + b, \sigma^2)$. This is the classic model we have already seen.

Example 2 (Linear Classification)**.** In this case, g^{-1} is the sigmoid function σ, so that $\mu = \sigma(w^\top x + b)$ with $y \mid x \sim \text{Bern}(\sigma(w^\top x + b))$. We can think of the sigmoid function as a smooth approximation to an indicator variable, so that $\sigma(w^\top x + b)$ is simply an estimation of the class of $w^\top x + b$.

Example 3 (Softmax Classification)**.** Here g^{-1} is the softmax function, so that $\mu_c = \text{softmax}(Wx + b)_c$ where W is some matrix rather than a vector. Remember that the softmax is defined by $\text{softmax}(z)_c = \frac{\exp(z_c)}{\sum_{c'} \exp(z_{c'})}$. Think of the softmax function as a sigmoid approximation to multi-dimension.

7.2 Basis Functions

It can be advantageous to apply models after modifying the data set using a transformation called a *basis*. We can have a huge variety of basis functions (some of which we have seen on the problem set), e.g., $\phi_j(x) = \sin(x), \tanh(x), \text{ReLU}(x)$, and so on. Vector examples (i.e., functions on vectors) include $\phi_j(x) = \max\{x_1, x_2\}$ and $\phi_j(x) = x_1 x_2 + x_1^2$. Figuring out a good basis within which to represent data is an important problem in machine learning.

Example 4. (Basis Function in Speech Recognition) A snippet of speech might be a waveform, and one way to extract features is to chunk the waveform by time, for each chunk applying a Fourier transform. Then we would take as features some values of each transformed chunk in the frequency domain. Typically this process gives 13 features per chunk. These features are then passed to a learning model.

We now consider general linear models in combination with basis functions. Suppose $y|x \sim \mathcal{N}(w^\top \phi(x) + b, \sigma^2)$, and $y|x \sim \text{Bern}(\sigma(w^\top \phi(x)) + b)$, where ϕ gives rise to a basis. We can do MLE just as before, i.e., compute $\text{argmax}_w \sum_n \log p(y_n|x_n, w)$. In general, these will be solvable just as before, e.g., with numerical optimization—iterative gradient calculation and updates. The form of the MLE depends on the distribution of $y \mid x$. When it is normal, the optimization becomes over sum of squares, when it is Bernoulli, the optimization becomes over cross-entropy (as discussed in previous classes).

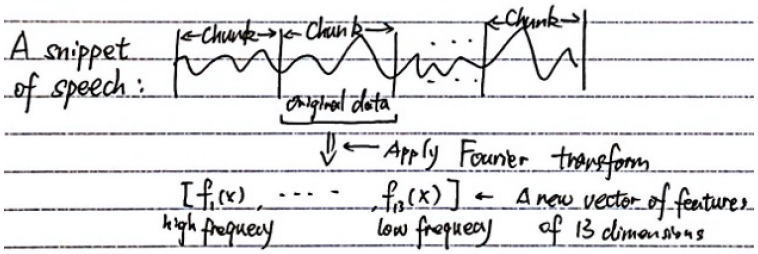


Figure 7.4: Fourier transform in speech recognition in Example 4

7.3 Now Let's Talk About Neural Networks

An adaptive basis function is a model with parameters in the basis functions. Neural networks are specific adaptive basis functions with particular structures, which we will describe below.
One can think of the function of a neural net as learning the correct basis function(s) for the data—having the computer come up with the best representation of the data over features of the form $\phi(x; w, b) = g^{-1}(Wx + b)$, where $g^{-1} = \text{relu}, \sigma, \text{softmax}$, or another nonlinear function. This procedure can be applied recursively, e.g., we can define the x that lives inside this basis function to also have its own basis function, e.g., $\phi(x; w, b) = g^{-1}(w\phi'(x; w', b') + b)$, and so on. This is also why neural networks are often referred to as "deep learning." This allows us to do non-linear regression and classification with parameters. Now, when we do regression and classification, we can have complex models such as

$$y|x \sim \mathcal{N}(w^\top \tanh(w'x + b') + b, \sigma^2)$$

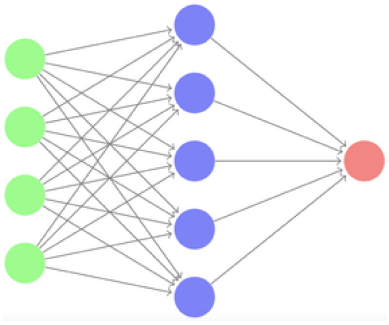
When we do MLE, we have to take the same argmax over the parameters w, w', b', b. All that's changing is that the function we are optimizing is non-linear, with many parameters, and non-convex. So when we optimize such functions, we might end up at a local optimum instead of the global optimum. We will see many techniques for combating the complexities of non-convex optimization.

7.4 Demo

See iPython notebook for demo.

7.5 Graphical Representation

Consider the adaptive basis $\sigma(w^\top \sigma(Wx + b') + b)$. We can represent this graphically with a two-layer, fully-connected network:



In the literature, the circles are called "neurons," matrices are "fully connected," each column is a "layer" with implied squashing, each line is a parameter. The goal of these networks is to find μ.
"Personally, I find this part—the 'it's like a brain!'—pretty silly. It's just linear algebra separated by nonlinear transformations." - S. Rush

7.5.1 Application Architectures for Neural Networks

In a typical neural network, we have $x \to$ Layer 1 $\to$ Layer 2 $\to \cdots \to$ Output, where each arrow is a linear map, and in each layer is a non-linear function. In the class before, we talked about classifying the MNIST data set—for this simple model, we had 8000 parameters(!). "But that's nothing—just yesterday I was working with a model with 1.2 billion parameters."
Although some of the power of neural networks comes from this flexibility in parameters, much of the interesting work is done in trying to find better neural network architectures that capture more of the essence of the data with fewer parameters. For example, the modern approaches to digit classification are done by convolutional neural networks, where the architecture captures some of the "local" information of images.

Example 5. [Speech Recognition] Suppose we want to map sounds into classes of saying the digits "one," "two," and so on. Recall that the typical approach is to split speech into chunks and perform Fourier transforms to extract features from each chunk. The problem here is that individual chunks don't necessarily map to single digits, since there's no guarantee the chunk even corresponds to an entire word in speech!
Instead, what is typically done in this case is convolution using a *kernel* (equivalently, a single weight vector called w^{tile}) that spans several chunks. Rather than applying learning on the full $\mathbb{R}^{n \cdot 13}$ data set (where n is the number of chunks), we multiply each k-chunk stretch of speech by the kernel to obtain $\approx n - k$ chunks. Based on our choice of kernel, we can take advantage of sparsity to improve structure in the data set. This is known as a one-dimensional convolution between the kernel, w^{tile}, and the input $\phi(x)$.

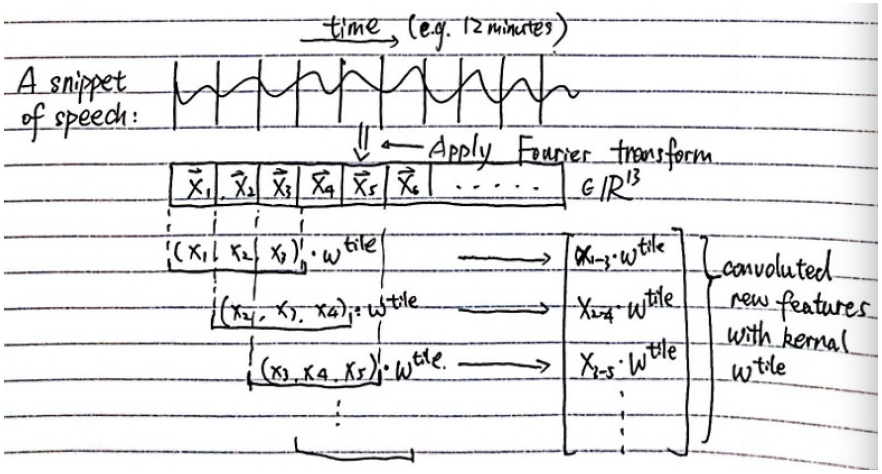


Figure 7.5: Convolution architecture in speech recognition in Example 5

Example 6 (Image Classification)**.** For the case of images, we can do the same as above with two-dimensional convolution, where have blocks in the image instead of tiles. This lets us pick up on information that is very local—e.g., or edges or corners in images, informatoin which can then be recombined in later layers with spectacular success.

Example 7 (Language Classification)**.** Suppose we want to determine whether a movie review was good or bad. Consider the review "The movie was not very good." One way to do this to convert words to vector representations (e.g., via word2vec or glove), since discrete words are difficult to deal with, but vectors let us have a more continuous approach while taking into account the meaning. We can do things like add these vectors up over the course of the review (e.g., a bag-of-words approach). An alternative is to take blocks of words and use a one-dimensional convolution. One advantage of the latter is that it allows you to pick up on structures such as "not very good," which wouldn't be observed in a bag-of-words model, which may pick up on the words "very" and "good" instead.

Remark. All of these convolution methods exist in PyTorch under `nn.conv`.

CS281: Advanced ML **September 27, 2017**

Lecture 8: Backpropagation & Directed Graphical Models

Lecturer: Sasha Rush *Scribes: Giridhar Anand, Michael Xueyuan Han, Ana-Roxana Pop*

8.1 Backpropagation in Neural Networks

8.1.1 Neural networks review

In the last lecture, we defined the mean parameter of a neural network as follows:

$$\mu = \sigma(w^T \text{ReLU}(Wx))$$

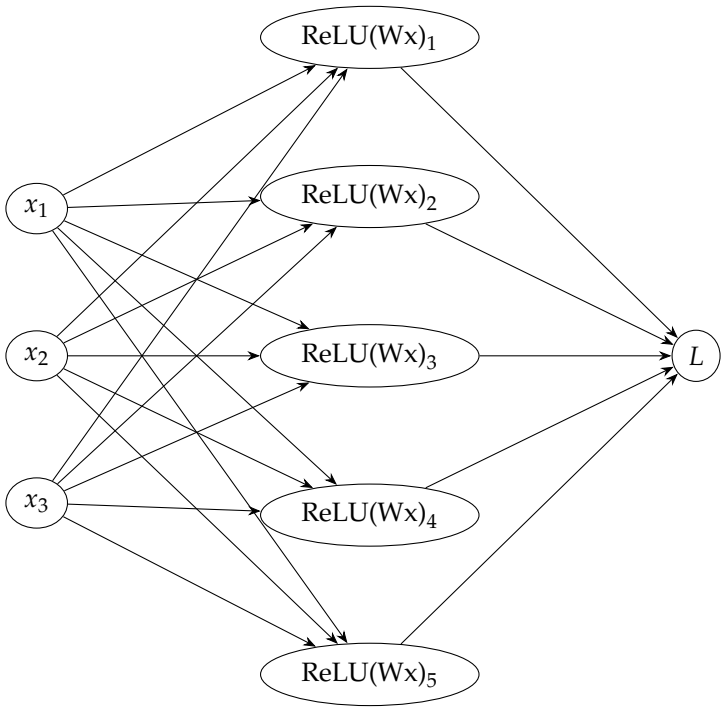
Here, μ parameterizes a Bernoulli distribution, $\text{Ber}(\mu)$. Suppose we want to find μ such that it maximizes the likelihood of a single data example (x, y). Then we compute

$$\mu = \underset{\mu}{\text{argmax}} \log p(y|x) = \arg\min_{\mu} (-\log p(y \mid x)) = \arg\min_{\mu} L$$

where L is the loss of the neural network.

8.1.2 Chain rule and backpropagation

We can represent the neural network graphically as follows:



In order to generate this graph, we must perform the following computational operations in order:

$$\begin{array}{ccccccccccc} v^{(0)} & \rightarrow & v^{(1)} & \rightarrow & v^{(2)} & \rightarrow & v^{(3)} & \rightarrow & v^{(4)} & \rightarrow & L \\ x & & Wv^{(0)} & & \text{ReLU}(v^{(1)}) & & w^T v^{(2)} & & \sigma(v^{(3)}) & & -\log v^{(4)} \end{array}$$

We would like to get the gradient terms $\dot{v}^{(i)} \equiv \frac{dL}{dv^{(i)}}$ for any i, which tell us how each part of the neural network affects our loss. We can do this by applying the chain rule (of calculus) to get a recursive solution (by convention, the derivative of a scalar with respect to a vector is represented as a column vector):

$$\frac{dL}{dv^{(i)}} = \left(\frac{dL}{dv^{(i+1)}}\right)^T \frac{dv^{(i+1)}}{dv^{(i)}}$$

$$\frac{\partial L}{\partial v_k^{(i)}} = \sum_j \frac{\partial L}{\partial v_j^{(i+1)}} \frac{\partial v_j^{(i+1)}}{\partial v^(i)}$$

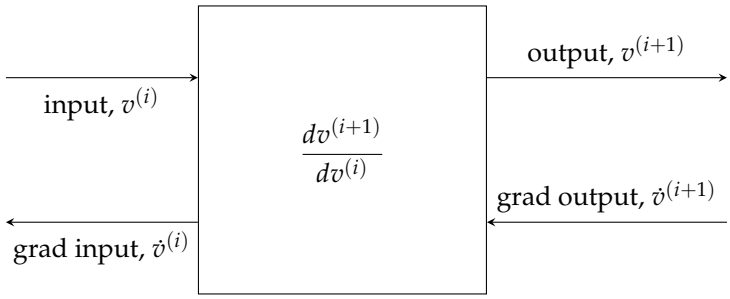
Since the gradient of each term depends on the gradient of the subsequent term, we can compute the gradients in reverse while applying the chain rule. This method is known as backpropagation. For each backward step, we need to remember everything that was computed in the corresponding forward step, namely $v^{(i)}$, $\dot{v}^{(i+1)}$, and $\frac{dv^{(i+1)}}{dv^{(i)}}$:

$v^{(0)}$	$\rightarrow$	$v^{(1)}$	$\rightarrow$	$v^{(2)}$	$\rightarrow$	$v^{(3)}$	$\rightarrow$	$v^{(4)}$	$\rightarrow$	L
x		$Wv^{(0)}$		$\text{ReLU}(v^{(1)})$		$w^T v^{(2)}$		$\sigma(v^{(3)})$		$-\log v^{(4)}$
$\dot{v}^{(0)}$	$\leftarrow$	$\dot{v}^{(1)}$	$\leftarrow$	$\dot{v}^{(2)}$	$\leftarrow$	$\dot{v}^{(3)}$	$\leftarrow$	$\dot{v}^{(4)}$	$\leftarrow$	
...		...		$(\dot{v}^{(2)})^T W$		$\dot{\sigma}(v^{(3)})\dot{v}^{(4)}$		$-\frac{1}{v^{(4)}}$		

8.1.3 Writing software for neural networks

- "blocks" style neural network (e.g. Torch)

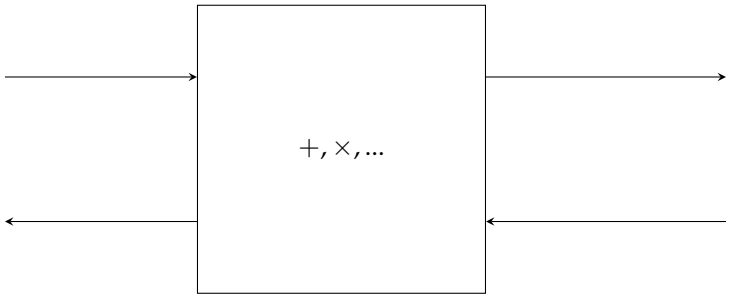
  Computation is done in "blocks" which are black boxes f that implement the following contract:



  We can also augment the black boxes. For instance, if we let f take in parameter W, we can also compute $\frac{dL}{dW}$ within this function.

- computational graph (e.g. Theano, TensorFlow)

  Everything is implemented in terms of primitives, so there are no black boxes:



  This allows us to optimize the neural network once and run it on many examples.

- imperative/autograd systems

  These are tape-based systems in which the computational graph can look different for different examples, but we can still compute gradients using backpropagation. Torch is built on an autograd core, but higher level functions like the Linear module take on a "blocks" style approach.

8.2 Graphical Models

8.2.1 Directed Graphical Models

The goal of using directed graphical models (DGMs) is to separate out two parts of a model:

1. Conditional Independence
2. Parameters and Parametrization

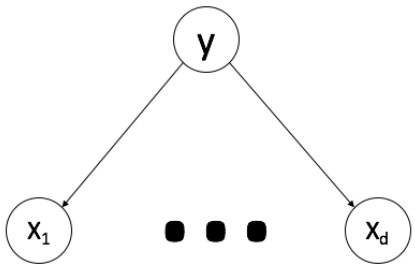


Figure 8.6: The graphical model of Naive Bayes

In the case of Naive Bayes (see Figure 8.6), we know from a previous lecture that:

- $p(y)$ is probably categorical.
- $p(x_j|y)$ could be one of many different distributions, including Categorical, Gaussian, Bernoulli, etc.

We are interested in the following distributions from the underlying data that we have:

- p(y, x): joint distribution
- $p(x_j)$: marginal distribution, or $p(y \mid x)$: conditional distribution

The structure of the model will often determine the difficulty of inference. This is the motivation of why we want to draw these graphs.
On a high level, given $p(A, B, C)$, we can always apply the chain rule (in probability):

$$p(A, B, C) = p(A \mid B, C)\, p(B \mid C)\, p(C)$$

However, if we write $p(A, B, C)$ in the way above, we basically assume that all variables depend on each other. In some cases, this is not necessarily true, and we want to find a factorization as below.

8.2.2 Factorization

If we have the case presented in Figure 8.7, we can rewrite $p(A \mid B, C) \rightarrow p(A \mid B)$. Having A only depend on one variable (B) is better than having it depend on two variables (B and C).

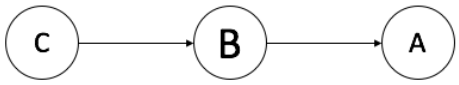


Figure 8.7: A graph where factorization is possible.

8.2.3 Formalism of DGMs

Formally, for directed graphical models (DGMs) (or *Bayes Nets*, or *causal graphs*) we have:

- A graph $G = (V, E)$ where $(s, t) \in E, s \neq t$ (V are vertices, E are edges)
- Each node is a random variable.
- Each edge is a conditioning decision.
- The graph is topologically ordered and it is a directed acyclic graph (DAG).
- Notation: $\text{pa}(x)$ represents x's parents.

8.2.4 Parents notation

Here, A and B are independent of each other, and they are C's parents (as in Figure 8.8). We can then write:

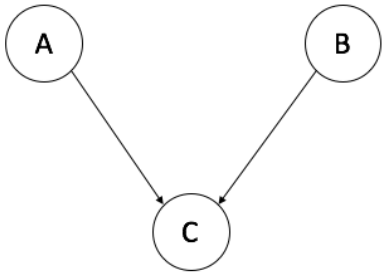


Figure 8.8: A graph to illustrate the use of parents notation.

$$p(A, B, C) = p(A)\, p(B)\, p(C \mid A, B) = p(A)\, p(B)\, p(C \mid \text{pa}(C))$$

8.2.5 Plate Notation

When we have lots of exchangeable variables (i.e., order is not important), we can use the *plate* notation. We want to graphically represent Naive Bayes on examples $(x_j^{(n)}, y^{(n)})$, in which

- y is parameterized on π: $p(y^{(n)} \mid \pi)$, and
- x_j depends on y and μ: $p(x_j^{(n)} \mid y, \mu)$

Since we have n samples of (x, y), we can use the plate notation, as shown in Figure 8.9.

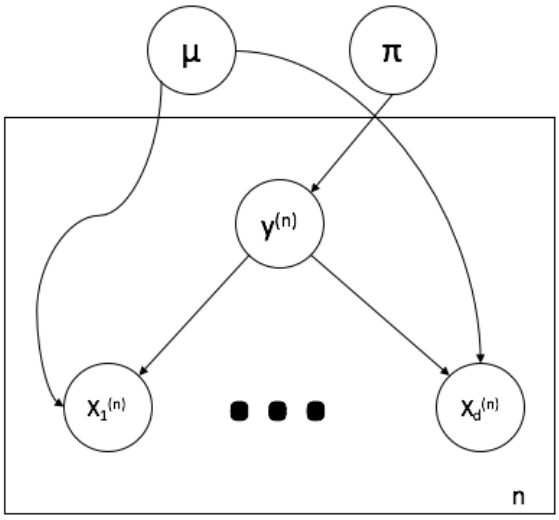


Figure 8.9: A graph to illustrate the use of plate notation.

8.2.6 Caching probabilities

You can save "probabilities" in the model, which is simply *caching* the values of probabilities with the graph. In this case (Figure 8.10), variables are discrete. We call C's probability table *conditional probability table (CPT)* since it is conditioned on A, B. Note that the values do not tell us anything about how the distribution is parameterized. Those are simply the probability values that you can read off from the graph.
Note also that the CPT of $p(x_i \mid x_1, \cdots, x_{i-1}) = O(\prod_i |x_i|)$ (i.e., exponential growth with the number of conditional terms).

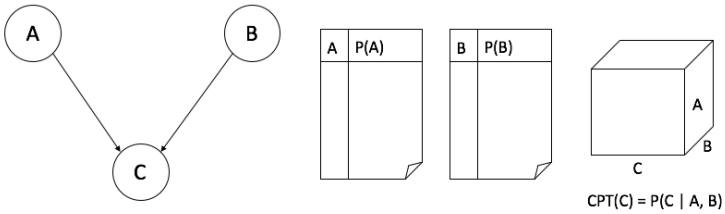


Figure 8.10: A model with probability tables. The CPT of C is a three-dimensional table.

8.2.7 Examples of Directed Graphical Models

Example 8 (Markov Chain)**.** Figure 8.11 shows an example of a Markov chain graphical model.

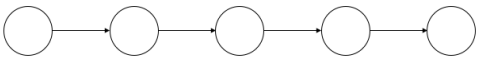


Figure 8.11: Markov Chain Graphical Model

Example 9 (Second Order Markov Chain)**.** Figure 8.12 shows an example of a second order Markov chain graphical model.

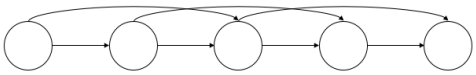


Figure 8.12: Second Order Markov Chain Graphical Model

Example 10 (Hidden Markov Model)**.** Figure 8.13 shows an example of a hidden Markov graphical model.

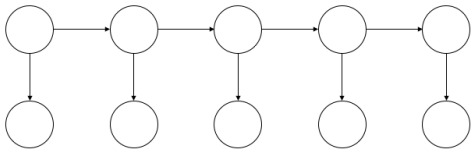


Figure 8.13: Hidden Markov Graphical Model

Example 11 (Navie Bayes)**.** Figure 8.14 shows an example of a Naive Bayes graphical model.

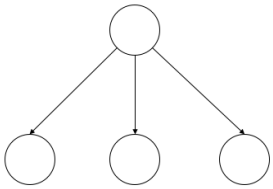


Figure 8.14: Naive Bayes Graphical Model

As we have seen in Figure 8.9, we can also incorporate parameters in the DGMs, as it is illustrated in the next example.

Example 12. In this case (Figure 8.15), we use the same Naive Bayes example with parameters. Here, we incorporate parameters $\alpha \sim$ Dirichlet. This is interesting because it combines two types of distributions: some of them are discrete, but in this example α and π are drawn from continuous distributions, as marked in the figure below.

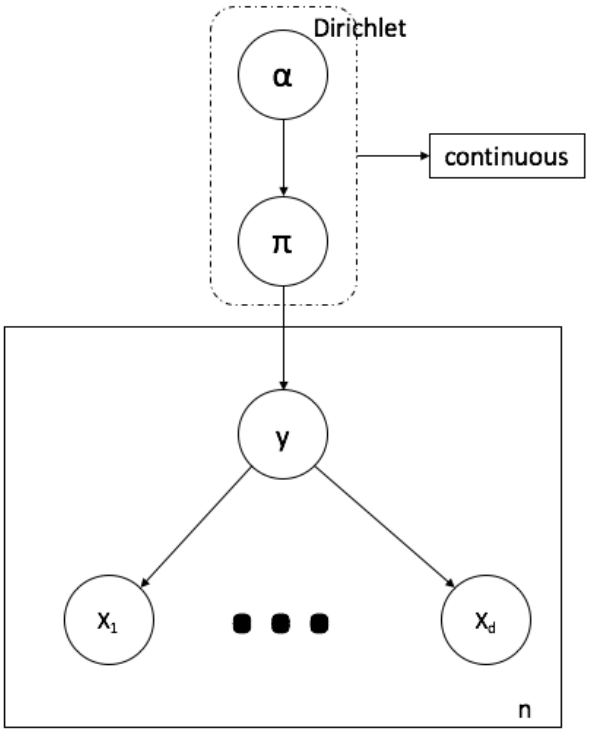


Figure 8.15: Naive Bayes Graphical Model with Parameters

Figure 8.15 corresponds to a single example. If we have multiple examples, we can use a plate-in-plate representation as in Figure 8.16.

8.3 Gaussian Directed Models

Gaussian directed models are a special case of DGMs where every one of the variables has the following distribution:

$$p(x_i \mid \text{pa}(x_i)) = \mathcal{N}(x_i \mid \mu_i + \sum_{j=\text{pa}(x_i)} W_{ij}(x_j - \mu_j),\ \ \sigma_i^2)$$

In the above equation, we transform each of the μ_i based on the starting mean plus a linear transformation of their parents (and to simplify things, we subtract the mean of each parent). We have an underlying generative process where each one of our random variables is a draw from a Gaussian and its children are a linear transformations of that draw.
This means that, we can rewrite x_i as:

$$x_i = \mu_i + \sum_j W_{ij}(x_j - \mu_j) + \sigma_i z_i \qquad \forall i \ \ z_i \sim \mathcal{N}(0,1)$$

Notice that $\sigma_i z_i$ is just Gaussian random noise.
If we define $S = \text{diag}(\sigma)$ (where each term will contribute a different corresponding σ_i), we can rewrite the above equation in a matrix form:

$$(x_i - \mu_i) = W(x - \mu) + Sz$$

where $\mu = [\mu_1, \cdots, \mu_d]$ is a vector containing each individual means.
By rearranging the terms, we find:

$$Sz = (I - W)(x - \mu)$$
$$x - \mu = (I - W)^{-1} Sz$$

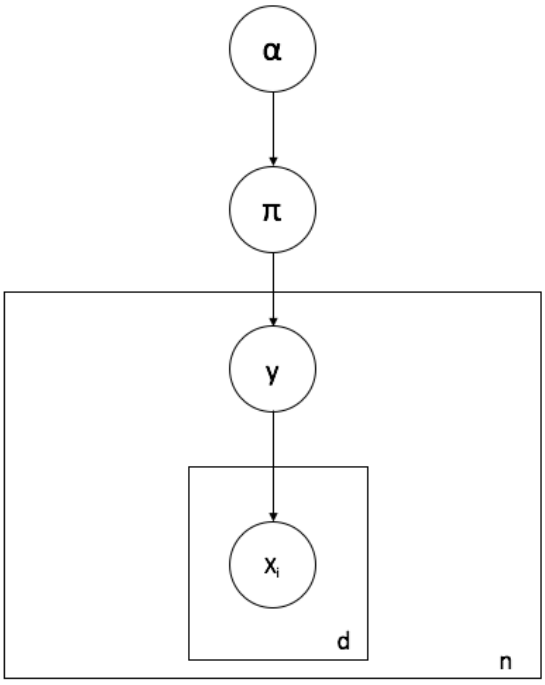


Figure 8.16: Naive Bayes Graphical Model with Parameters and Plate-in-Plate Notation

This tells us how the x random variable differs from the mean at each of the different positions. We know that the Σ term for our covariant matrix is defined as:

$$\begin{aligned}\Sigma \equiv \text{cov}\left[x-\mu\right] &= \text{cov}\left[(I-W)^{-1}S\,z\right] \\ &= (I-W)^{-1}S\,\text{cov}\left[z\right]\,S((I-W)^{-1})^T \\ &= (I-W)^{-1}\,S^2\,((I-W)^{-1})^T\end{aligned}$$

which means that, in general, for Gaussian DGMs we have:

$$\text{Gaussian DGM} \sim \mathcal{N}(\mu,\,(I-W)^{-1}\,S^2\,((I-W)^{-1})^T)$$

Remark. We will talk about *D-Separation* in the next lecture.

CS281: Advanced ML **October 2, 2017**

Lecture 9: Undirected Graphical Models

Lecturer: Sasha Rush *Scribes: Chris Hase, Denis Ellenrieder, Shuran Zheng, Rafi Small, Tim Menke*

9.1 Left from last lecture: Conditional independence properties of DGMs

When we have a directed graphical model (DGM), how can we "read" the graph and learn about the independence properties of the variables? To begin we note that conditional independence follows if the marginal probability factors in the following way:

$$p(A, B|C) = p(A|C)\, p(B|C) \Rightarrow A \perp B|C \tag{9.9}$$

Let (B) denote observing B, which informs our understanding on how A and C are conditional independent or not. Our cases are then:

$$(A) \rightarrow (B) \rightarrow (C),\ A \perp C \times \tag{9.10}$$

$$(A) \rightarrow \textcircled{B} \rightarrow (C),\ A \perp C|B \checkmark \tag{9.11}$$

$$(A) \leftarrow (B) \rightarrow (C),\ A \perp C \times \tag{9.12}$$

$$(A) \leftarrow \textcircled{B} \rightarrow (C),\ A \perp C|B \checkmark \tag{9.13}$$

$$(A) \rightarrow (B) \leftarrow (C),\ A \perp C \checkmark \tag{9.14}$$

$$(A) \rightarrow \textcircled{B} \leftarrow (C),\ A \perp C|B \times \text{ explaining away} \tag{9.15}$$

Information is being blocked in cases 2, 4, and 5 but flowing freely in all other cases. It's useful to think about the concept of "explaining away" to understand what is going on in the last case. '"Explaining away" is a common pattern of reasoning in which the confirmation of one cause of an observed or believed event reduces the need to invoke alternative causes.'
(http://strategicreasoning.org/wp-content/uploads/2010/03/pami93.pdf)

How would a directed graphical model be interesting in practice? One example is probabilistic programming: demonstrated in the IPython notebook "DGM.ipynb". The demonstration shows how we can convert programs from a directed to an undirected form. We can also specify priors on our features and visualize the flow of the model.

The purpose of creating a DGM is to specify the relationship between variables of interest, in order to to facilitate understanding of the independence properties.

9.2 Undirected Graphical Models (UGM)

Differences of UGMs as opposed to DGMs:

1. There are no arrows on lines
2. No longer model *local* probabilistic decisions (the term "local" is important and defined later)
3. UGMs are Markov random fields (similar to exponential families) or conditional random fields (similar to generalized linear models)
4. Nice part: The rules are much simpler, especially for conditional independence
5. Downside: The math is much more complicated
6. Sasha's personal bias: UGMs are much more useful

9.2.1 Independence properties

As stated above, we have conditional independence if the marginal probability factors in the following way:

$$p(A, B|C) = p(A|C)\, p(B|C) \Rightarrow A \perp B|C \tag{9.16}$$

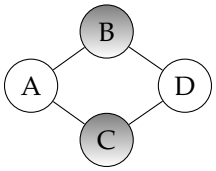
$$\text{(A)} - \text{(B)} - \text{(C)},\ A \perp C \times \tag{9.17}$$

$$\text{(A)} - \text{(B)} - \text{(C)},\ A \perp C|B \checkmark \tag{9.18}$$

$$\tag{9.19}$$

We can say that we have "conditional independence" between two nodes given some third node if all paths between the two nodes are blocked. For our simple example with three nodes, this is when the third node is in the evidence, that is, when the third node is "seen".

Here is an example of how independence works in the UGM.

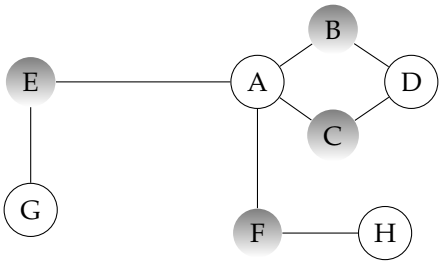


$$A \perp D|S \text{ if S blocks all the paths} \tag{9.20}$$

$$\text{Here: } A \perp D|B, C \checkmark \tag{9.21}$$

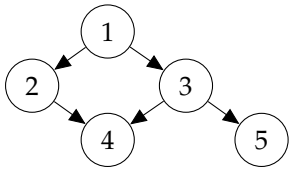
Fundamental consequence: Imagine we have a graph with a node A and some complicated connection of nodes around A. We can make A conditionally independent from all other nodes by conditioning on the "Markov blanket" of A. The Markov blanket is defined to be the *neighbors of A*. We will see later in this class that is is very nice if we can establish these independence properties. We will be able to look at a point in a graphical model and, if we can condition on the Markov blanket of the node of interest, we can ignore the rest of the graph.

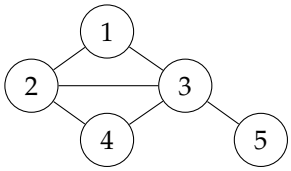
In the example below, conditioning on the Markov blanket of A means conditioning on B, C, E, and F.



9.2.2 Converting directed to undirected graphs

In order to convert directed to undirected graphs, we will use the (socially improperly termed) technique of *moralization*, i.e. "marry the parents".





To carry out this process, we take all the directed edges and make them undirected. We then create additional edges by "marrying" the parents of a node. In this case we gain an extra edge between 2 and 3, which comes from marrying the parents of 4.

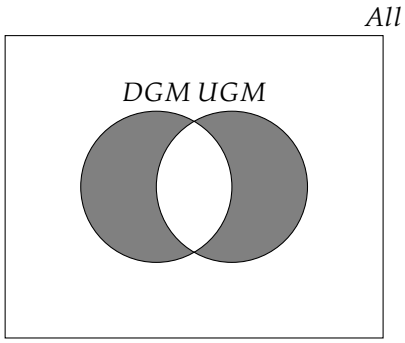
Let's write Naive Bayes out in a directed graphical model:

For this case, the UGM and DGM are the same. However, if we condition the other way, i.e. the features $x_1, ..., x_D$ are on top of the DGM with directed arrows towards y at the bottom, we would need to add connections in the UGM between all of the features. In the illustrated example, the joint probability having seen y is

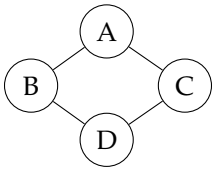
$$\prod_d p(x_d)p(y|x_1, ..., x_v). \tag{9.22}$$

9.3 Corner cases

Unfortunately, there are lots of corner cases in UGMs. DGMs and UGMs represent only a subset of all graphical models. There is some overlap between the UGM and DGM classes within the set of all independence structures. In the corner cases, we cannot convert between DGMs and UGMs and retain all of the independence information encoded within.



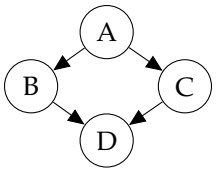
We will now consider an example of a UGM and attempt to convert it to a DGM.



Here we have

$$A \perp D|B, C \checkmark \tag{9.23}$$
$$B \perp C|A, D \checkmark \tag{9.24}$$
$$\tag{9.25}$$

One potential DGM we can create is

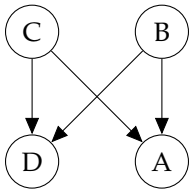


In contrast to the UGM, the DGM has the following independence properties.

$$A \perp D|B, C \checkmark \tag{9.26}$$

$$B \perp C|A, D \times \tag{9.27}$$

Another DGM may be

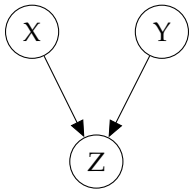


And now we see that

$$A \perp D|B, C \checkmark \tag{9.28}$$

$$B \perp C|A, D \times \tag{9.29}$$

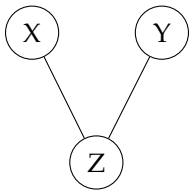
So there is no directed graphical model structure that gives us same properties that the original UGM had. How about another example of a DGM



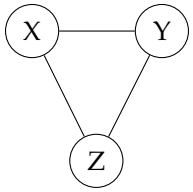
This graph has two properties:

$$(1)\ X \not\perp Y|Z \tag{9.30}$$

$$(2)\ X \perp Y \tag{9.31}$$



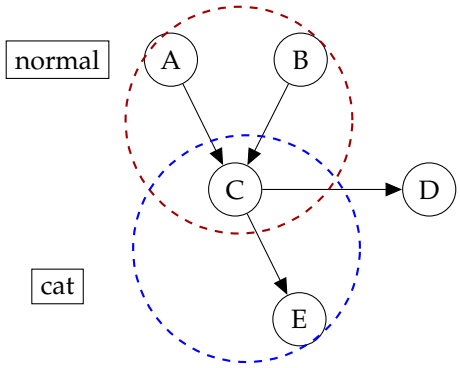
This graph has neither property 1 nor property 2.



Here we do have property 1 but not property 2.

9.4 Parametrization of UGMs

The following section is some of the harder material that we will cover. However, this is the last time we are introducing a new model. Thereafter, we will do inference on the models we have discussed.



Suppose that in this example the nodes within the red circle follow a local normal distribution, but that the nodes within the blue circle follow a local categorical distribution. The notation relating a node X to its parents is $p(x|\text{pa}(x))$, where $\text{pa}(c)$ refers to the parents, and the conditional probability table is "locally normalized", "sums to one", and is non-negative.

For UGMs we use "global normalization". All is fine locally as long as the whole global probability sums up to make whole joint distribution normalized. For this, we treat everything as an *exponential family*.

$$p(x_1, ..., x_d) = \text{multivariate exp. fam.} = \exp\left\{\theta^T \phi(x_1, ..., x_D) - A(\theta)\right\} \tag{9.32}$$

Here, θ are the parameters and $\phi(x_1, ..., x_D)$ the sufficient statistics. For every "clique" in the graph, we associate a set of sufficient statistics. A "clique" is defined as a set of nodes that are all connected to each other. Note: a set of one or two nodes forms a trivial clique.

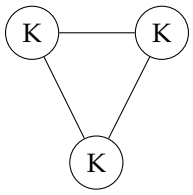
In the discrete case:

$$\phi(x_1, ..., x_D) = [\phi_c(x_c)...]^T \tag{9.33}$$

Each entry is an indicator of the value set for a clique.

$$\phi_c^v(x) = \begin{cases} 1 & \text{if clique } c \text{ has } x_c = v \\ 0 & \text{otherwise} \end{cases} \tag{9.34}$$

Suppose we have the following UGM in which each node can take on one of k possible discrete outcomes:



This means $v \in K^3$ in this example. We also have a Θ associated with each of these values. Since $\phi(x)$ is big (and awkward), we use the following notation:

$$\theta^T \phi(x) = \sum_c \theta^T [0, ..., 0, \phi_c(x), 0, ...,]^T = \underbrace{\sum_c \theta_c(x_c)}_{\text{convenient notation}} \tag{9.35}$$

Now we write out the whole thing:

$$p(x_1, ..., x_D) = \exp\left\{ \underbrace{\sum_c \theta_c(x_c)}_{\text{(neg) energy}} - A(\theta) \right\} \tag{9.36}$$

Finally, we can compute the value $A(\theta)$:

$$A(\theta) = \log \underbrace{\sum_{x'}}_{\text{"everything"}} \exp\{\sum_c \theta_c(x'_c)\} \tag{9.37}$$

The sum over "everything" is our nemesis because it is over something big. And in general, this is NP-Hard or even #P-Hard. But in practice, the structure of the graph determines the difficulty. Examples of the "everything" include: all possible images, social network graphs... (How we can exchange the sums over x' and c depends on the structure of the graph.)

9.5 Examples

9.5.1 Example 1: Naive Bayes model

$$\text{(A)} - \text{(B)} - \text{(C)}, \text{ all binary} \tag{9.38}$$

We have two cliques: (A) − (B) and (B) − (C)
Define features and parameters:

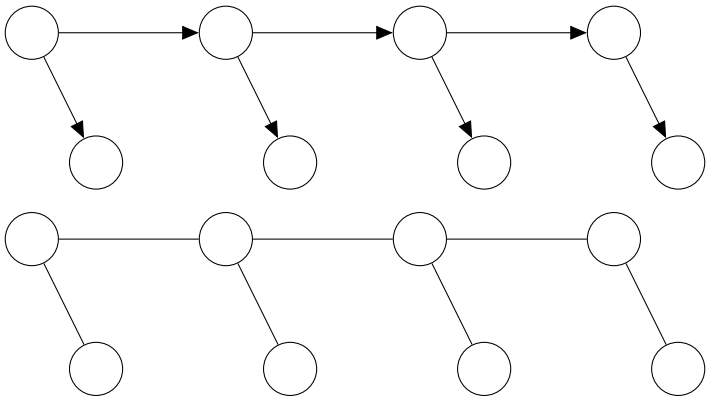
$$\phi(x) = \begin{bmatrix} \mathbb{1}(A=0,B=0) \\ \mathbb{1}(A=0,B=1) \\ \mathbb{1}(A=1,B=0) \\ \mathbb{1}(A=1,B=1) \\ \mathbb{1}(B=0,C=0) \\ \mathbb{1}(B=1,C=0) \\ \mathbb{1}(B=0,C=1) \\ \mathbb{1}(B=1,C=1) \end{bmatrix}, \ \theta = \begin{bmatrix} \theta_{AB}(0,0) \\ \theta_{AB}(1,0) \\ \theta_{AB}(0,1) \\ \theta_{AB}(1,1) \\ \theta_{BC}(0,0) \\ \theta_{BC}(1,0) \\ \theta_{BC}(0,1) \\ \theta_{BC}(1,1) \end{bmatrix} \tag{9.39}$$

Then we have

$$p(A=a, B=b, C=c) = \exp\{\theta_{AB}(a,b) + \theta_{BC}(b,c) - A(\theta)\}, \tag{9.40}$$

where $A(\theta) = \log\left(\sum_{a',b',c'} \exp\{\theta_{AB}(a',b') + \theta_{BC}(b',c')\}\right)$.

9.5.2 Example 2: Hidden Markov model



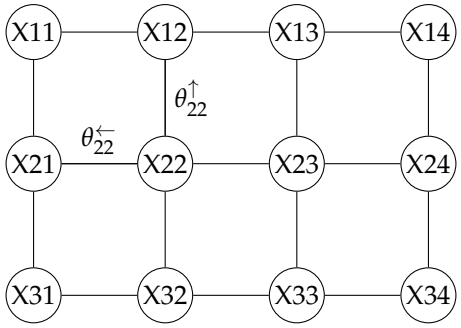
Any distribution obeying this structure has an exponential family parametrization. We convert the local distribution

$$p(y_1)\prod_i p(y_i|y_{i-1})p(x_i|y_i) \tag{9.41}$$

to the globally normalized distribution

$$p(x,y) = \exp\{\sum_i [\theta_{i,i-1}(y_i, y_{i-1}) + \theta_i(y_i, x_i)] - A(\theta)\}. \tag{9.42}$$

9.5.3 Example 3: Ising model



For this example, the shown grid could connect to other such grids. As an example, suppose we want to detect the foreground and background of an image. For this, we perform a binary classification of each pixel (0=pixel in background; 1=pixel in foreground). We want to obtain a probability that a pixel is in either class. The class should depend on the neighboring pixel so that there is consistency among neighboring pixels - it would be weird if every other pixel is in a different class.
This results in a binary model with neighbor scores. In order to force this to be a probability distribution, we treat the whole thing as an exponential family:

$$p(x) = \exp\left\{\sum_{ij}\left(\theta^{\uparrow}_{ij}(x_{ij}, x_{i-1,j}) + \theta^{\leftarrow}_{ij}(x_{ij}, x_{i,j-1}) + \theta_{ij}(x_{ij})\right) - A(\theta)\right\}, \tag{9.43}$$

where $A(\theta) = \log\left(\sum_{x'} \exp\sum_c \theta_c(x'_c)\right)$. Note that A is again very hard to calculate. Also note that the "missing" $\theta^{\downarrow}_{22}$ and $\theta^{\rightarrow}_{22}$ in the diagram are given by other $\theta^{\uparrow}$ and $\theta^{\leftarrow}$ so that when we sum over all i and j we are not double counting any of the connections.

CS281: Advanced ML **September 11, 2017**

Lecture 10: Exact Inference: Time Series

Lecturer: Sasha Rush *Scribes: Max Hopkins, Sebastian Wagner-Carena, Mien Wang, Jamila Pegues*

10.1 Prelude

10.1.1 Notation

Recall from Lecture 9 our notation for the joint probability distributions of *Undirected Graphical Models* (UGMs). In particular, we have

$$p(x_1, \dots, x_T) = exp\{\sum_c \theta_c(x_c) - A(\theta)\}$$

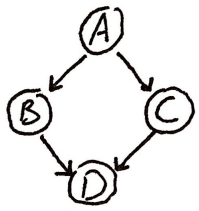
Here, $\theta_c(x_c)$ is the score associated with some clique c, and x_c is some value assignment on the clique. This notation is great because it corresponds to exponential families! However, there are a few other notations you may see around:

$$\begin{aligned} p(x_1, \dots, x_T) &\propto \prod_c exp(\theta_c(x_c)) \\ &= \prod_c \psi_c(x_c)) \end{aligned}$$

This latter notation is used by Murphy. $\psi_c(x_c)$ are the potentials, and they are simply $\exp(\theta_c(x_c)$. The score functions θ_c are then known as the log potentials.

10.1.2 Moralization

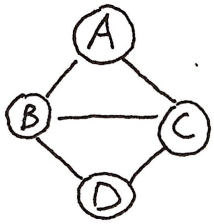
Recall from last lecture the process of conversion between a *Directed Graphical Model* (DGM) and UGM, known as moralization. Here we consider two diagrams:



The first has joint probability distribution

$$P(A, B, C, D) = P(A)P(B|A)P(C|A)P(D|B, C)$$

After moralization, we have



We see that the UGM has two cliques, the $c_0 = \{A, B, C\}$, and $c_1 = \{B, C, D\}$. Using the notation above, this gives us

$$P(A, B, C, D) = \psi_{c_0}(A, B, C)\psi_{c_1}(B, C, D)$$

This form gives a particularly nice parallelism where it is clear that the first three terms

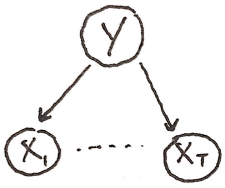
$$P(A)P(B|A)P(C|A) = \psi_{c_0}(A, B, C)$$

and

$$P(D|B, C) = \psi_{c_1}(B, C, D)$$

10.1.3 Conditional Independence vs. Parametrization

It is important to keep in mind that DGMs only specify conditional independence. The same DGM's could correspond to completely different parametrizations of the random variables. For example, consider Naive Bayes:



In Lecture 5, we discussed all kinds of Naive Bayes (NB), including Bernoulli, MVN, Categorical, and more. All of these follow the diagram above, which only specifies the conditional probability distribution

$$p(x|y) = p(y)\prod_i p(x_i|y)$$

Now consider a conditional model:

Here we are only interested in $p(y|x_1, \ldots, x_T)$. Similarly to NB, the parametrizations of this model can take many forms. For instance, one could use logistic or linear regression, most GLM's, Neural Networks, or even convolutional Neural Networks. In fact this is the model on which Alphago functions, where features x are the tile placements on the board, and y is the output move!
The main point here is that we can stick arbitrary parametrizations into graphical models. These diagrams just specify the conditional dependence–not the distributions. Further, as long as we specify the graphical model structure, we will be able to tell how hard inference will be.

10.2 Time Series

In this lecture, we only consider an informal definition of a time series. This structure is marked by collecting data over time, and predicting an output for each data input. We are going to consider a special case, labeling a time series. Here our we will have $x_{1:T}$ as our input sequence (features), and $y_{1:T}$ as our output labels. We will begin by providing examples of labeling time series:

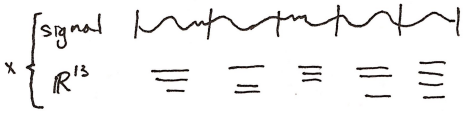
Example 13 (OCR)**.** Here we have blocks of pixels, discrete vectors, each which depict a letter. These blocks are our x_i, and each corresponding y_i the discrete symbol the x_i represents

y = {t,h,e, ,d,o,g}

Example 14 (NLP)**.** Here we are given discete words as input variables x, and we wish to predict discrete y, their parts of speech

x: the dog ate a carrot
y: DT NOUN VERB DT NOUN

Example 15 (Speech Recognition)**.** Recall from previous lectures that our signal may be divided up into time steps and translated to continues vectors in $\mathbb{R}^{13}$, these our are continuous input variables x. Our output for each vector is the phoneme corresponding to the sound.



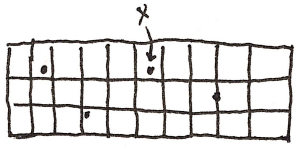
y = {d,d,d,o,o}

Example 16 (Tracking)**.** Here we are tracking the position of some object with presumed Gaussian noise. In this case both our input and output variables are continuous. The inputs are our position vectors, and the output is the predicted correction for noise.

Example 17 (Education)**.** This example is based upon the presentation at the beginning of the class. Our inputs are given by a kinect tracker and are continuous body positions at snapshots in time. The discrete output y is whether the body position corresponds to attentive or bored.

y = {attentive, attentive, bored, bored }

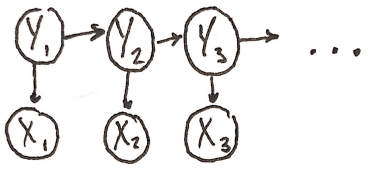
Example 18 (Touch-typing)**.** We consider typing on an iphone, where inputs are the continuous position of keystrokes on the phone. The outputs y are given by discrete letters.



y = {s,t,o,p}

10.3 Markov Models

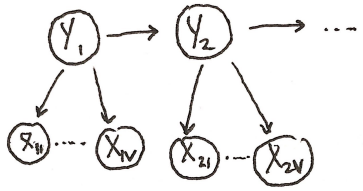
10.3.1 Hidden Markov Model



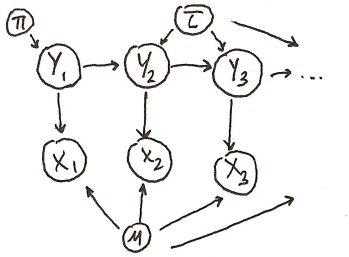
We discussed several *Markov Models* in previous lectures. We begin with the *Hidden Markov Model* (HMM), which actually predates DGMs. HMMs assume discrete y, though DGMs with the same structure may have continuous y. This has a fully joint parametrization $p(y_{1:T}, x_{1:T})$, where in most cases $p(y_t|y_{t-1})$ is categorical. We can choose the distribution of $p(x_i|y_i)$ to fit our given circumstances:

1. $p(x_i|y_i)$ is categorical [e.g. parts of speech]
2. $p(x_i|y_i)$ is MVN [typing, speech (this uses a mixed gaussian)]
3. $p(x_{i1}, \ldots, x_{iv}|y_i) = \prod_v p(x_{iv}|y_i)$ [OCR]

The last example here has an embedded Naive Bayes model for each feature x.



HMMs such as the above are often parametrized in the following manner:

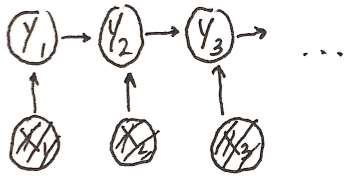


10.3.2 State Space Model

While continuous y has the same graphical model, it has completely different usage and was developed completely separately. This model is called the *State Space Model* (SSM). This model is often used when we have a continuous signal disturbed by gaussian noise–this becomes multivariate and we can compute inference nicely.

10.3.3 Maximum Entropy Markov Model

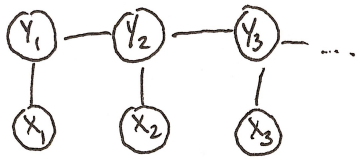
Yet another Markov Model is the *Maxent Markov Model* (MEMM). This model assumes we have observed all features, and flips the direction of the conditioning in the vertical direction.



Thus we are interested in $p(y_1, \ldots, y_T|x_1, \ldots, x_T)$, and $p(y_t|x_t, y_{t-1})$ is a GLM such as logistic or softmax regression.
MEMM comes with the distinct advantage that we no longer have to assume our features are independent. This is particularly useful in, say, tagging parts of speech where can pick an arbitrary feature basis without worrying about independence. However, the model comes with the downside that there is no closed form, so we must use SGD, i.e. we isolate each x_i and predict y_i with logistic regression.
Picking $p(y_t|x_t, y_{t-1})$ here as an arbitrary neural network is called a Neural Network Markov Model or NN-Markov Model.

10.4 Conditional Random Field Markov Model

While CRF is still a Markov Model, it gets its own subsection due to its importance. The CRF is simply the UGM of all the above Markov Models:



Recalling Subsection 10.1.1, we write

$$p(y_{1:T}|x_{1:T} = exp\{\sum_t(\theta_t^h(y_t, y_{t-1}) + \theta_t^o(y_t, x_t)) - A(\theta)\}$$

Here h and o refer to the labeled arrows in the diagram. This is the general form of any Markov Model. Note that because we have observed the features $x_{1:T}$, we may rewrite the $T_t^o(y_t, x_t)$ terms as $\theta_t^o(y_t; x_t)$. If we are trying to do inference, we can think of the above after conditioning as simply

This is a simple *Markov Chain* UGM. In fact after converting to UGM and conditioning on our features, all Markov models become the above!
Now while we can compute the closed form of the MLE on HMMs, CRF is a bit tricker. However, it is a member of the exponential family model, and we know the MLE for these families in general.

$$\mathbb{E}(\phi(x)) = \frac{\sum \phi(x_d)}{N}$$

Here, $\phi(y)$ are the sufficient statistics–but what do these look like? In fact we went over this in a previous lecture, it is simply a vector of indicator functions for every clique assignment (see Lecture 9). For inference in particular, we care about

$$\mathbb{E}(\mathbf{1}(x_c = v))$$

Then the clique marginals are given by

$$\frac{\sum_{x'=x'_c=v} p(x')}{\sum_{x''} p(x'')}$$

However this may be computationally intractable, as the x'' we sum over is the entire universe! However, this can be computed efficiently for some models such as chain models

10.5 Bonus: Lecture by Bertrand Schnieder

Bertrand Schnieder gave a snazzy guest lecture on his research, and advertised that the datasets from his research would be ideal as a base for CS 281 final projects. We take a moment to review his lecture below.
Schnieder was very interested in studying patterns across concepts of collaborative learning. For example, joint attention refers to when a group of individuals are focused on the same physical object, and is an important part of language development. Signs of joint attention can be seen in the physical movement of the individuals involved, such as their eye movements, gestures, and body postures.
Schnieder discussed studies that he and others carried out to explore joint attention. For example, one study gave forty-two pairs of two participants a task to solve, within a total of forty-five minutes of time. Over that time period, Schnieder et al. used high-frequency and multi-modal sensors to track different aspects of movement, including: video, audio, eye movement (tracked at 60Hz), physiological data (like heart rate;

tracked at 1-30Hz), and even body posing and coordination (using a Kinect; tracked at 30Hz). Schnieder pointed out that these sorts of studies provide massive amounts of data. During the lecture, he highlighted that there are a number of cool CS 281 projects that could arise from datasets like this one. Some projects he proposed are:

1. Unsupervised machine learning: modeling collaborative learning processes with probabilistic graphical models.

2. Supervised machine learning: training models to make predictions. For instance, training a model to predict the joint visual attention of two people based on their gestures, head orientation, and speech. Another example is training a model to predict physiological activity based on features like pupil size and body posture. With supervised machine learning, however, Schnieder noted that overfitting can be a dangerous pitfall.

3. Design your own!

Schnieder emphasized that he is open to ideas, and that he highly encourages anyone interested in working with this data in some way or form to email him at bertrand_schneider@gse.harvard.edu.

CS281: Advanced ML **October 11, 2017**

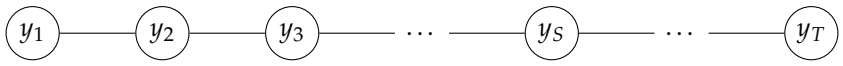
Lecture 11: Exact Inference: Belief Propagation

Lecturer: Sasha Rush *Scribes: Ismail Ben Atitallah, Hao Wu, Raphael Rovinov, Ziliang Che, Jiaoyang Huang*

11.1 Simple Marchov Chain

For a simple Markov Chain with time limit T and V classes per node:

$$\begin{aligned} p(y,t) &= \exp(\sum_t \theta_t^T(y_{t-1}, y_t) + \theta_t^o(y_t) - A(\theta)) \\ &\propto \prod_t \psi_t(y_{t-1}, y_t)\psi_t(y_t) \end{aligned}$$



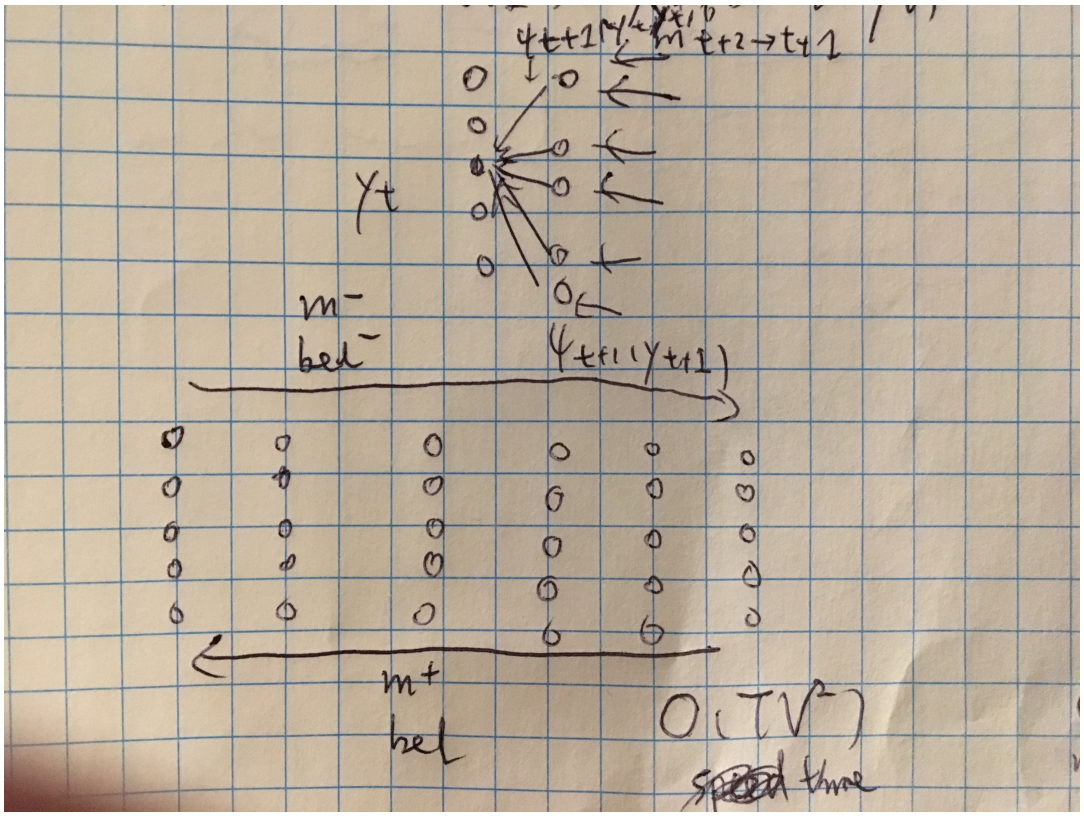
11.1.1 Distributive Property

Marginal:

$$\begin{aligned} p(y_s = v) &= \sum_{y_{1:T}, y'_s = v} \prod_t \psi_t(y'_{t-1}, y'_t)\psi(y'_t)/Z(\theta) \\ &= \sum_{y'_t} \psi_T(y'_T) \sum_{y_T - 1}{}' \psi_{T-1}(y'_{t-1})\psi_T(y_{T-1}, y'_T) \sum \cdots \sum_{y'_2} \psi_2(y'_2) \sum_{y'_1} \psi_1(y'_1, y'_2)\psi_1(y'_1) \end{aligned}$$

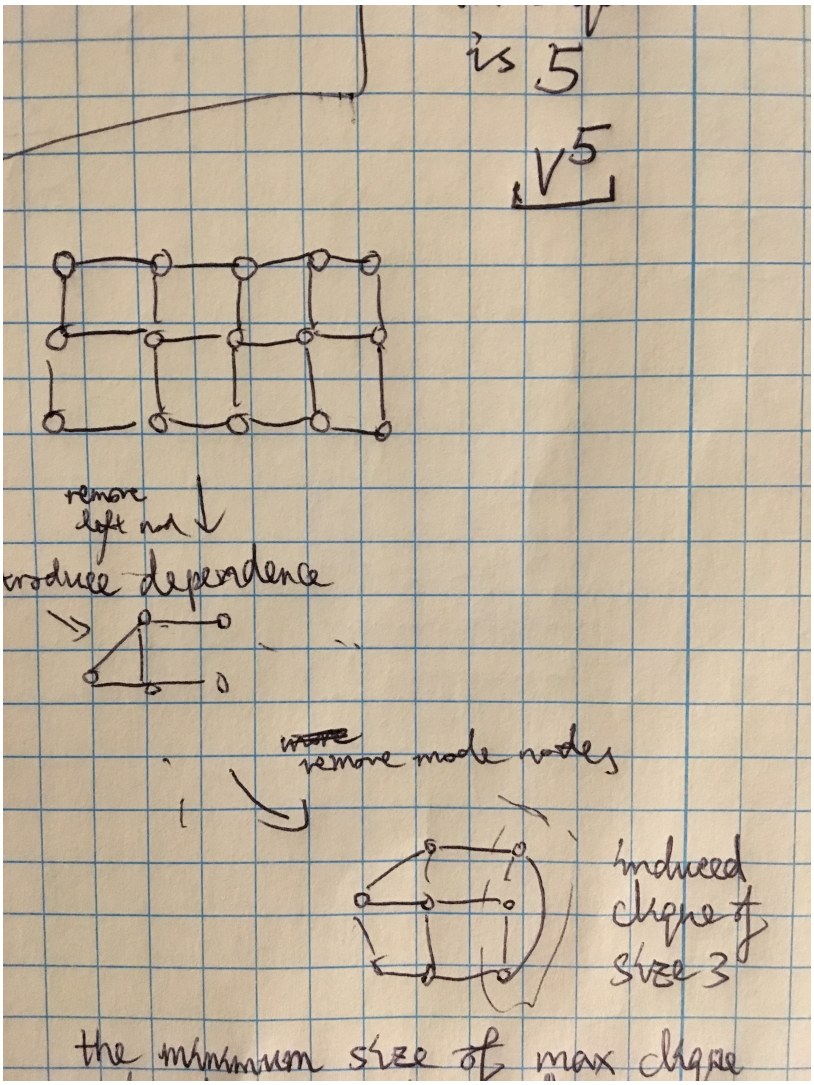
11.1.2 Compute All Marginals with Dynamic Programming

$$\begin{aligned} \text{bel}_t^-(y_t) &\propto \psi_t(y_t)\text{m}_{t-1\to t}^-(y_t) \\ \text{m}_{t-1\to t}^- &= \sum_{y_{t-1}} \psi_{t-1}(y_{t-1}, y_t)\text{bel}_{t-1}^-(y_{t-1}) \\ \text{m}_{t+1\to t}^+ &= \sum_{y_{t+1}} \psi_{t+1}(y_{t+1}, y_t)\psi_{t+1}(y_{t+1})\text{m}_{t+2\to t+1}^+(y_{t+1}) \\ p(y_t) &= \text{bel}_t(y_t) \propto \text{m}_{t+1\to t}(y_t)\text{bel}_t^-(y_t) \end{aligned}$$



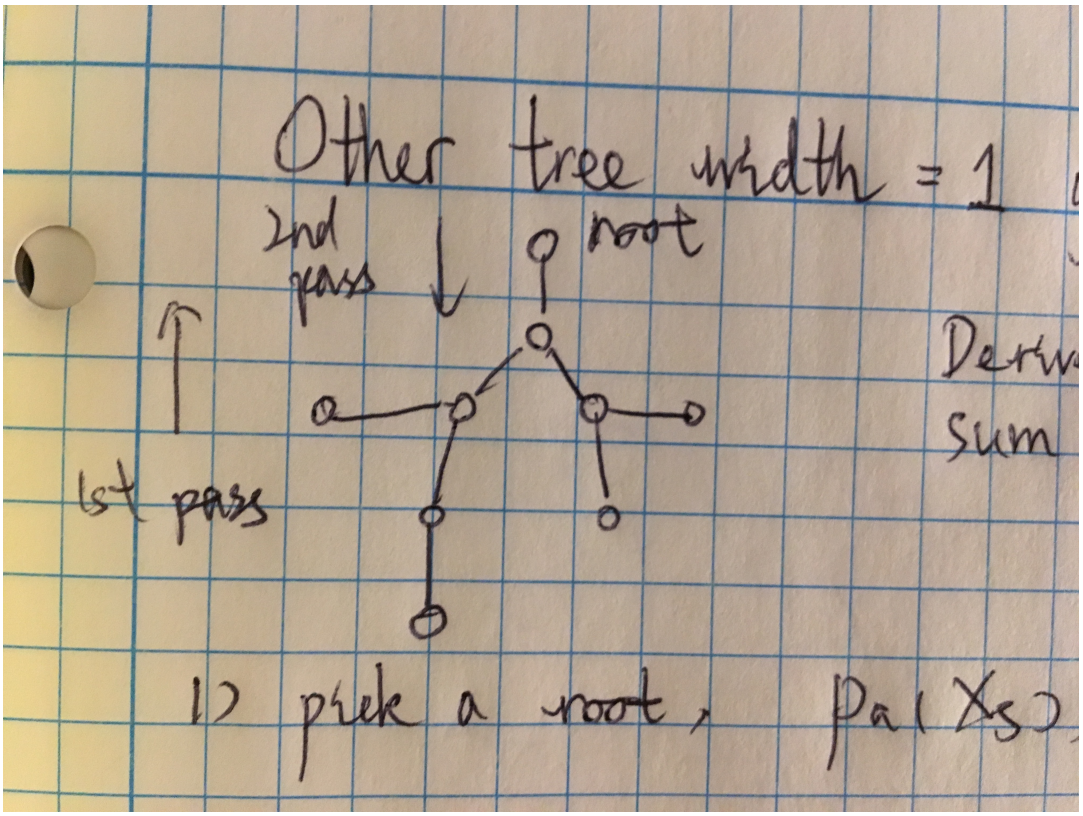
11.2 Other Graphs

$$p(y_s = v) = \sum_{y', y'_s = v} \prod_c \psi_c(y'_c)$$



Computing this sum product might be hard. Here we define the minimum size of maximum clique indcued -1 to be the *treewidth* of the graph.

11.2.1 Compute Marginals for treewidh = 1 Graph



Here we derive the generalization of the forward-backward sum-product algorithm

1. Pick a root s, $\text{pa}(x_s)$,$\text{ch}(x_s)$
2. Upward pass:

$$\text{m}_{s\to t}(x_t) = \sum_{x_s} \psi_{s-t}(x_s, x_t)\text{bel}_s^-(x_s)$$
$$\text{bel}_t^-(x_t) \propto \psi_t(x_t) \prod_{s\in\text{ch}(t)} m_{s-t}^-(x_t)$$

3. Downward pass:

$$\text{bel}_s(x_s) \propto \text{bel}_s^-(x_s) \prod_{t\in\text{pa}(s)} \text{m}_{t\to s}^+(x_s)$$
$$\text{m}_{t\to s}^+(x_s) = \sum_{x_t} \psi s - t(x_s, x_t)\psi_t(x_t) \prod_{c\in\text{ch}(t),c\neq s} = \text{m}_t^-(x_t)$$

11.3 Parallel Protocol for Sum Product

$$\text{bel}_s(x_s) \propto \psi_s(x_s) \prod_{t\in\text{nbr}(s)} \text{m}_{t\to s}(x_s)$$
$$\text{m}_(s \to t) = \sum_{x_s} \psi_s(x_s)\psi_{s-t} \prod_{u\in\text{nbr}(s),u\neq t} (x_s)$$

11.4 Final Notes

For this lecture we have been ultilizing + and $\times$ and distributive property

11.4.1 Commutative Semi-ring

$+$	max	$\cap$
$\times$	$\times$	$\cup$
$\vee$	$\vee$	$\vee$
marginal	argmax	satisfying assignment

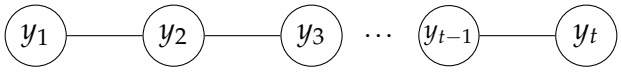
CS281: Advanced ML **October 16, 2017**

Lecture 12: Recurrent Neural Networks

Lecturer: Sasha Rush *Scribes: Alexander Goldberg, Daniel Eaton, George Han, Moritz Graule, Kristo Ment*

12.1 Introduction

Recall the basic structure of a time series model:



In previous lectures we discussed interpreting such a model as a UGM, with log-potentials given by:

$$\theta(y_t) + \theta(y_{t-1}, y_t)$$

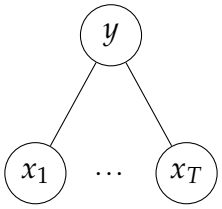
What did we gain from this abstraction? Because all of these models have the same comditional independence structure, we are able to run the same algorithm for inference on all of these parameterizations (sum-product). Thus, conditioned on observed data, we may find the exact marginals: $p(y_s = v)$

Perhaps these structures are not necessary? Is exact inference required in cases where there is alot of data?

12.2 RNNs

12.2.1 What is an RNN?

Recall our discussion of using neural networks for classification. The UGM describing this setting is as follows:



We could then choose to parameterize $p(y|x_{1:T})$ as a neural network:

$$p(y|x_{1:T}) = Softmax(\mathbf{w}^T \phi(x_{1:T}; \theta))$$

where ϕ was just some linear combination of $x_{1:T}$ passed through a link function. If we wish to apply this scheme to cases in which $x_{1:T}$ is a *sequence* we might think to use a ϕ of the following form:

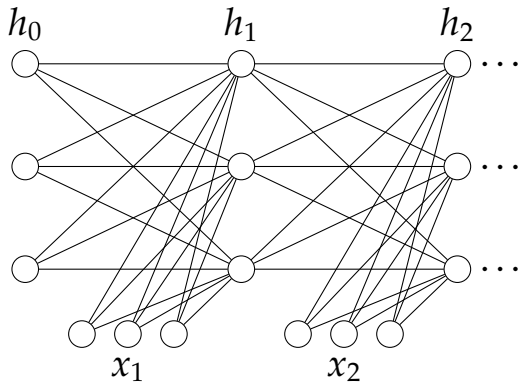
$$\phi(x_{1:T}; \theta) = tanh(\mathbf{w}\mathbf{x}) = tanh(\sum_{t=1}^{T} w^{(t)} x_t)$$

However, the problem with this type of approach is that it is "time invarient," that is the same weights are shared by all of the x_t. To see why this is problematic, consider using a bag of words representation for the X_t and encountering the two sentences: "The man ate the hot dog." and "The hot dog ate the man." While these two sentences are saying completely different things, they result in the same value generated by ϕ. Recurrent neural networks get around this problem by implementing the following choice of ϕ:

$$\phi(x_{1:T}; \theta) = tanh(\mathbf{w}^{(1)} x_t + \mathbf{w}^{(2)} \phi(x_{1:t-1}; \theta) + b)$$

where $\mathbf{w}^{(1)} x_t$ incorperates the current positional input, $\mathbf{w}^{(2)} \phi(x_{1:t-1}; \theta)$ carries information from the previous inputs, b is the bias and $tanh$ is the chosen nonlinear transformation.

Representing this RNN as a computational graph:



where we call h_t the RNN hidden state. As you can see, each h_t is a nonlinear function of $x_{1:t}$.

12.2.2 RNN Training

To understand how backpropagation works in RNNs, consider the functions that the NN is composed of:

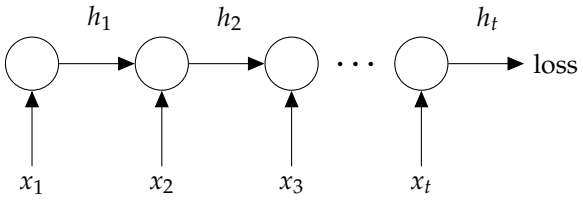
$$h_1 = tanh(\mathbf{w}^{(1)}\mathbf{x}_1 + \mathbf{w}^{(2)}\mathbf{h}_0 + b)$$

$$h_2 = tanh(\mathbf{w}^{(1)}\mathbf{x}_2 + \mathbf{w}^{(2)}\mathbf{h}_1 + b)$$

$$\dots$$

$$h_t = tanh(\mathbf{w}^{(1)}\mathbf{x}_t + \mathbf{w}^{(2)}\mathbf{h}_{t-1} + b)$$

Notice that the only parameters to optimize in these expressions are $\mathbf{w}^{(1)}$ and $\mathbf{w}^{(2)}$, which are shared among all of the equations. In our normal representation of backpropagation we have:



Thus the size of the computational graph and the amount of backpropagation necessary will scale with the length of the input, T. Now, thinking of this situation like a simple feed-forward NN, how many layers does this network have?

12.2.3 Issues: Network Layers

This network will have T layers, where T is the length of the input, which may be very long. Thus, when performing back-propagation it is very likely that there will be problems of gradient instability – very high (exploding) or low (vanishing) values of the gradient somewhere in the back propagation, making it hard to learn the parameters for low layers.

Consider using tanh as the activation function at each layer. Then, the gradient is close to 0 for very large or very negative values, which is quite likely to happen somewhere in a network with many layers, so multiplying these small gradients together in back-propagation will make the contributions of the beginning of the sequence to the loss very small. Thus, it could take prohibitively long to learn weights for the beginning of the sequence. This problem is known as the problem of *vanishing gradients*.

12.2.4 Main idea/trick/hack for vanishing gradients

In order to deal with vanishing gradients, we want to try to pass on more info from low layers while taking gradients, so we add connections variously called:

- residual connections
- gated connections
- highway connections
- adaptive connections

The idea of residual connections is that we let

$$\mathbf{h_t} = \mathbf{h_{t-1}} + \tanh\left(\mathbf{w}^{(1)}\mathbf{x_t} + \mathbf{w}^{(2)}\mathbf{h_{t-1}} + b\right)$$

so that taking the gradient at layer t we get more information passed on from the the linear term $\mathbf{h}_{t-1}$ outside of the tanh.
In fact, we can adaptively learn how much the gradient at each time step should be taken from the previous time step directly from the cata. Thus, we weight the contributions of the $\mathbf{h_t}$ and $\tanh\left(\mathbf{w}^{(2)}\mathbf{h_{t-1}} + ...\right)$ by a factor λ that is also learned from the data:

$$\mathbf{h_t} = \lambda \odot \mathbf{h}_{t-1} + (1-\lambda) \odot \tanh\left(\mathbf{w}^{(1)}\mathbf{x_t} + \mathbf{w}^{(2)}\mathbf{h}_{t-1} + b\right)$$

$$\lambda = \sigma\left(\mathbf{w}^{(4)}h_{t-1} + \mathbf{w}^{(3)}\mathbf{x}_t + b\right)$$

By passing on information directly from previous timesteps, we can prevent vanishing gradients, since the linear terms pass on more information from previous timesteps. In this sense, the λs function as "memory" of the previous timesteps. Important RNN variants using this idea are:

- LSTSM (Long short-term memory networks)
- GRU
- ResNet

12.3 Using RNNs

12.3.1 Classification

Our classification algorithm has three stages:

1. Run LSTM
2. Compute Softmax
3. Find maximizing class

This means that in order to make a prediction, we only need to compute

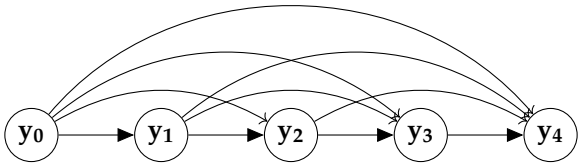
$$p(y_i|x_{1:T}) = \text{Softmax}(\mathbf{w}^{(2)}\mathbf{h}_i)$$

and multiply across each y_i to obtain

$$p(y_{1:T}|x_{1:T}) = \prod_{i=1}^{T} p(y_i|x_{1:T}) = \prod_{i=1}^{T} \text{Softmax}(\mathbf{w}^{(2)}\mathbf{h}_i)$$

Critically, this means that we do not attempt to model the relationship between the y_i at all, and thus we do not need to assume any distribution over y!

Lets compare this to our alternative approach, which requires full generation. Imagine that any y_i is conditional on all of $y_{1:i-1}$. Then our DGM (for five nodes) is a fully connected K_5:



How can we then compute $p(y_s = v)$? A naive approach would be to literally enumerate all possible sequences and sum across all possibilities. However, this is very computationally expensive (exponential in T). Instead, we can speed up this approach by employing a greedy search. Let

$$\hat{y}_1 = \text{argmax}_v \, p(y_1 = v)$$

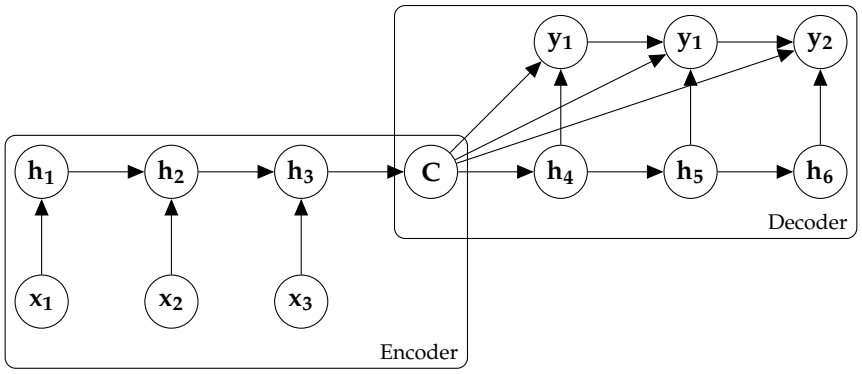
Now for each subsequent point y_i, we can compute

$$\hat{y}_i = \text{argmax}_v \, p(y_1 = v | \hat{y}_{i-1})$$

which is now linear in T, instead of exponential in T.

12.3.2 Applications

RNNs are commonly used for machine language translation and speech recognition. A simple machine translation model is that of the Encoder-Decoder model.



In this model, a sentence from the first language is fed in word by word (each x_i) into the Encoder. This then runs through a normal RNN setup before being fed into C, which stores the final result of the encoder.

Now in the decoder, another RNN is run in reverse that spits out words in the second, translated, language. Each translated word y_i depends on the current layer in the decoder RNN, h_i, C and the last translated word (to prevent the same word from being generated multiple times).

Remark. We will talk about *Information Theory* in the next lecture.

12.4 Practical Exercise: Speech Generator

RNNs have been used successfully as speech generators, taking in a sequence of letters (or words) and predicting subsequent letters (words). Implement and train a character-level RNN with PyTorch, and use it to sample a sentence.

12.4.1 Solution

The architecture of the RNN can be conveniently implemented with torch.nn. As our training data, we downloaded all of Donald Trump's 2016 campaign speeches from http://www.presidency.ucsb.edu/2016_election.php. The concatenated data is available in `concat_speeches.txt` (size: 1.3

MB). To facilitate the training process, we only included lowercase letters and a few additional symbols `.:?[]` from the raw data, for a total of `n_letters=32` characters. We then assembled a training set by randomly drawing 10,000 30-character sequences from the data, and used each of these to predict the next (31st) characters. We also generated a validation set by similarly drawing 500 additional character sequences. All of the input data must be converted to one-hot vectors: the entire training set was stored as an 30x10000x32 array, as required by `torch.nn.LSTM` below.
We then proceeded to set up an RNN with long-short term memory (LSTM), 2 layers, and `n_hidden=200` as follows:

```
model = torch.nn.Sequential()
model.add_module("lstm", torch.nn.LSTM(n_letters, n_hidden, 2))
model.add_module("encoder", torch.nn.Linear(n_hidden, n_letters))
```

The linear "encoder" layer is necessary to convert the output of the final LSTM layer to 32 probabilities, each for a single character. A `dropout` parameter can be added to the LSTM call in order to prevent overfitting (this will randomly overwrite some input nodes with zeros). We then implement the loss function via a cross-entropy layer which includes a `LogSoftmax()` function:

```
loss = torch.nn.CrossEntropyLoss()
```

Finally, we choose Adam from `torch.optim` as our optimizer:

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.005)
```

The model is then trained over a significant number of epochs: it took us a few hours to get through 300 epochs on a single CPU. This performance can likely be improved by utilizing the `torch.cuda` library to run the training on a GPU instead. The training function for a single epoch can be implemented as follows:

```
# Train the RNN and return the loss
def train(model, optimizer, data, target):

    optimizer.zero_grad()
    output, hc = model.lstm(data)
    new_loss = loss(model.encoder(output[-1]), target)
    new_loss.backward()
    optimizer.step()
    return new_loss.data[0]
```

Note that we are only using the last step of the output to compare against the target: we are interested in predicting the 31st character after seeing the entire 30-character sequence. We can monitor the training and validation losses to check for convergence and overfitting.
Finally, we can feed the trained model any input sequence and let it predict the next (say, 140) characters. For example, after 175 epochs of training, simply feeding the network "Trump:" yields the following sentence:

```
Trump: i an presisent of the sime and i wat the worlica tore and the spaising our
    componitien and i wat of the spared the worled the worle and the spaiting our
    communitien
```

This already begins to sound like real speech, considering the fact that the RNN had to learn the language from scratch! In any case, humans take years to learn to speak whereas we have only been training for a few hours. At 300 epochs, we obtained samples like this:

```
Trump:ey southon fac cort the whale getton. buoller campouse tilathad sumplee ours
    dousting to beokew athing the semerica ince forette. [applause]
```

However, both training and validation losses were still, at that point, declining rapidly. Thus, further training will likely result in an improved performance.
This solution is merely an outline to a plausible more successful algorithm. In particular, we can expand the size of the model, the number of training sequences, or the sequence length, or tweak other relevant model parameters to train a smarter speech generator. The Python code used for this write-up is included as `trump_generator.py`.

CS281: Advanced ML **October 18, 2017**

Lecture 13: Information Theory

Lecturer: Sasha Rush Scribes: Peter Chang, Ruiqi Chen, Joonhee Choi, Joshua Meier, Raphael Rouvinov, Hyungmok Son

13.1 Announcements

The Midterm is next Monday. The list of topics is on the website. It's open note but not open computer. You can bring your textbook. They'll try to bring copies of the textbook for people who don't have it so they don't have to print out copies.

13.2 Introduction

Interestingly, almost all of information theory is laid out in a single paper, written by Claude Shannon in 1948. We often quote Alan Turing as the 'father' of CS, but for the area we are discussing, that 'father' is Shannon. (Aside: Video of device Shannon built called the Ultimate Machine)

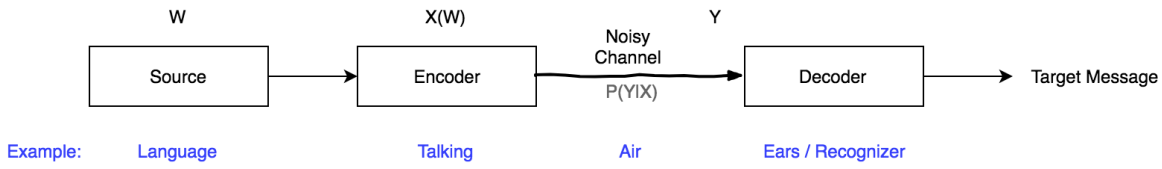
Today, we'll cover the core aspects of information theory and discuss how we'll use it in this class.

Earlier in this class, we've focused on exact methods for inference: MLE, MAP, etc. The one exception was neural networks, which are not convex.

Exact inference is only tractable in a small subset of models. Given most models are intractable, what do we do? We use approximate inference. Here, there are two main types of approximation: optimization-based (which includes coordinate ascent, SGD, and Linear Programming methods) and sample-based. You can use them together, but they have different histories, so we'll discuss them separately.

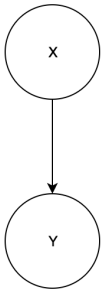
Information theory brings together many of the expectation-maximization methods we've seen earlier into a unified language. In particular, information theory lets us study relative entropy.

13.3 Information theory (Murphy 2.8)



The *Information Source* spits out bits (call this W), which is passed into the *encoder* (giving X(W)). The encoded bits pass through a *noisy channel* giving Y, which is passed into a *decoder* that gives us our target message.

We don't know how the noisy channel will act in a deterministic way. But we have a probability distribution $p(Y|X)$ describing its behavior. The target message also has a distribution $p(x|y) \propto p(x)p(y|x)$. Where $p(x)$ represents the source model, and $p(y|x)$ represents the channel model. We have a simple graphical model here:



Several fields of research are contained in the first diagram. One area of information theory is called channel coding (Noisy Channel in diagram). It studies how to best encode the data so it is most robust to the noisy channel. A second area is called source coding - data compression (Source in diagram). It discusses how we can exploit the way information is naturally represented in the world. For example, compression falls here. (Aside: Hutter prize).

Example: people in speech recognition use this as an analogy for what they are trying to do. The model of what a person wants to say is W. The encoding path is the sound the person makes $X(w)$. The decoder is what we (or the microphone) hear. And the goal, of course, is to uncover the target message.

We can write this analogy:

1. Source: Language
2. Encoder: Talking
3. Noisy Channel: Air
4. Decoder: ears/recognizer

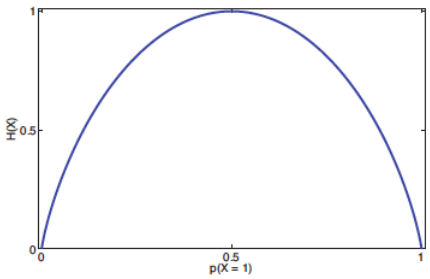
Then, $p(X)$ represents the source model. If you know a person well, you can anticipate what is going on in their head, and you know what they are going to say This makes your $p(X|Y)$ stronger. We multiply this quantity by $p(Y|X)$ which represents the noisiness of the channel.

13.4 Definitions

Entropy:

$$H(X) \stackrel{\Delta}{=} -\sum_{k=1}^{K} p(x=k)\log_2 p(x=k) = -E_k[\log_2 p(x=k)]$$

For a given random variable X, entropy measures the "uncertainty" in the distribution. Entropy maxes out when we have full uncertainty in the distribution.



This comes back to the idea of source coding. If there is full certainty in what can and is sent, then it does not matter what the encoder/decoder does. The answer is trivial. In contrast, if there is full uncertainty, we have to work much harder.

The unit of measure of this is called "bits"($\log_2$) or "shannons" or "nats" ($\log_e$). The average number of bits needed to represent a message is less than

$$H(x) + 1$$

We won't prove this, but this is the fundamental link between information and coding.

Shannon Game In his paper, Shannon proposes the "Shannon Game," which tries to quantify human source coding: given a sequence of text, give a probability distribution over the next letter/word.

THE ROOM WAS NOT VERY LIGHT A SMALL OBLONG READING LAMP ON THE DESK SHED GLOW ON POLISHED ___.

In English, it takes roughly 80 guesses to get the right answer. We write that the perplexity or "how confusing the next prediction is" is

$$2^{H(x)}$$

Minimizing perplexity is where the 'power'/advances of RNNs comes from.

Cross Entropy:

1. p - is a true underlying distribution
2. q - another distribution that approximates p

$$H(p, q) = -\sum_k p(x = k) \log q(x = k) = E_p[\log q(x = k)]$$

Cross entropy tells us expected number of bits to encode true distribution p with q.

We can sample from p to approximate

$$\tilde{x_1}, \ldots, \tilde{x_N} \sim p(x)$$

Then, we compute the minimization of the cross entropy.

$$min_q - \frac{1}{n} \sum_n^N \log q(\tilde{x_n})$$

But this is just the negative log likelihood for categoricals. Last class, we talked about RNNs. We learn q and compare to p. If we can make it more and more like p, this gives us the ability to do compression and source modeling better.

Relative entropy (KL-Divergence):

$$KL(p||q) \triangleq E_p \log \frac{p_(x = k)}{p(q = k)} = \sum_k p_k \log \frac{p_k}{q_k} = \sum_k p_k \log p_k - \sum_k p_k \log q_k = -H(p) + H(p, q)$$

So the relative entropy is the negative entropy of p by itself plus the relative entropy of p and q. It's a way of comparing two distributions, but is not a metric - $KL(p||q) \neq KL(q||p)$

Theorem. $KL(p||q) \geq 0$
Proof. We have that

$$-KL(p||q) = E_p[\log \frac{q_k}{p_k}] \leq \log E_p[\frac{q_k}{p_k}] = \log \sum_k q_k(x) = \log 1 = 0$$

This is only $KL(p||q) = 0$ when $p = q$.

Jensen's Inequality.

$$f(E[x]) \leq E[f(x)]$$

if f is convex. Mostly using when $f = \log$.

Information Geometry (Working with asymmetrical divergence).
We have some p and want $q \in Q$ (set of distributions). There are two options:

1. Forward (moment projection):

$$argmin_{q \in Q} KL(p||q) = -H(p) + H(p, q)$$

Note that $H(p)$ falls out when minimizing q. So instead, we minimize at $H(p, q)$, which is the negative log likelihood. So essentially, we are matching the moments. This equals

$$-E_p \log q_k$$

Issues arise when $q_k \to 0$ and $p_k > 0$. Forward projection will try to avoid zeros.

2. Reverse (information projection):

$$argmin_{q \in Q} KL(q||p) = -H(q) + H(q, p) = -H(q) + E_q \log p_k$$

This method "matches the modes".

Issues arises when $p_k = 0$ and $q_k = 0$. Reverse projection will over predict zeros.

These two approaches fall under a field called Information Geometry. The forward approach is called moment projection. The backward approach is called information projection. "Essentially, these are both methods for computing closeness among different distributions."

Jensen-Shannon divergence – combine the above: add half of the Forward projection value to half of the reverse projection value. This approach is popular right now.

This work is highly related to Generative Adversarial Networks (GANs). This metric is important in producing non-blurry images. Combining two images looks bad, using a method with mode finding yields a better image.

13.5 Demo

We did an example iPython notebook (KL.ipynb) in class.

Next class, we will use the reverse projection to get Q. This is called "variational inference".

CS281: Advanced ML **October 25, 2017**

Lecture 14: Mixture Models

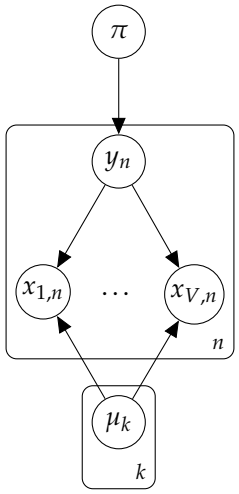
Lecturer: Sasha Rush *Scribes: Eric Dunipace, Andrew Fai, Isaac Xia*

14.1 Administrative Announcements

Overview: Second half of class is on Inference, especially on harder problems. There is only one problem set left, but the bulk of the time will be on the project.
There is some extra time left in the schedule, for which Prof. Rush will poll topics from the class.

14.2 Mixture Models: Introduction

Up until now, we've been considering mostly supervised learning, where we had an input x, an output y, and parameters θ.
Previously when doing MLE, MAP, marginalizing over nodes in a graphical model, we weren't trying to find a distribution, we were finding a **point-estimation** rather than a **distribution**.
For example, suppose we want to think about models from Naive Bayes:

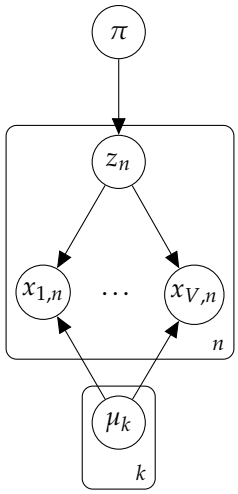


with joint probability

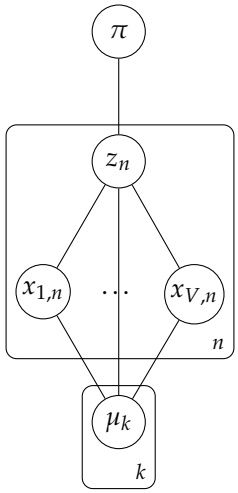
$$p(y|\pi)\prod_{j=1}^{V} p(x_j|y,\mu),$$

where the priors could be Categorical on $p(y|\pi)$ and Bernoulli on each $p(x_j|y,\mu)$.
In this setting, with data $\{x_i, y_i\}$, we can perform log-likelihood maximization with $p(\{x\},\{y\})$ to infer the parameters μ and π.

In a new setting, instead suppose we have the same graphical plate model, but we now do not observe the categories; instead of y_n, we denote these latent variables as z_n.



which also can be represented with the following undirected graphical model:



What does this graphical model look like, and what dependencies do we need to consider?

- There's clearly a dependency on π, but we can't observe z_n.
- There are the μ_k from the x_{in}.

All of these Naive Bayes models without observed y_n are used for **different types of clustering**. These models are referred to as **mixture models**, and can handle multimodality well.

14.3 Types of Mixture Models

Up until now, most models we've used have unimodal distributions, e.g. normal distributions.

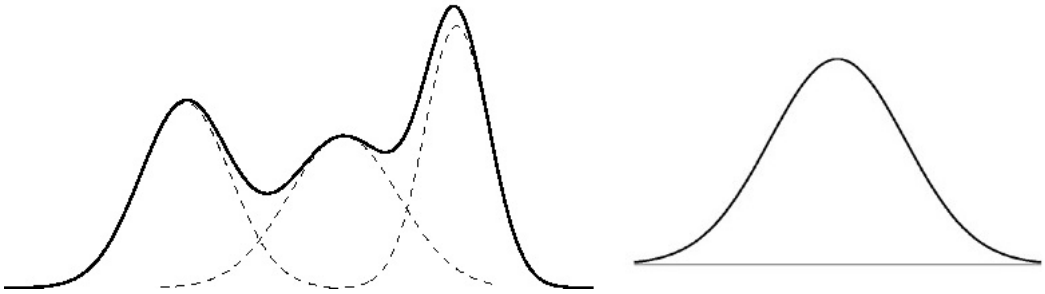


Figure 14.17: Multimodality of Multiple Gaussians vs Unimodality of One Gaussian

Although we won't limit ourselves to only using continuous mixture models, we'll be primarily working with mixtures of Gaussians. Below are examples of mixture models we'll see more of.

14.3.1 Mixture of Gaussian Modes

We can have a mixture of Gaussian models, by simply mixing different Gaussian conditional distributions together:

$$p(x|y) \sim \mathcal{N}(\mu, \Sigma)$$

An application example is speech recognition, which must deal with different accents (e.g. British vs American) for pronunciations of the same word. In this case, Gaussian mixture models are able to consolidate the seemingly different pronunciations.

14.3.2 Mixture of Bernoulli/Multinomial Models

We can instead mix together various Bernoulli or Multinomial conditionals.

$$p(x_j|y) \sim \text{Bern}(\mu_j) \text{ or } \text{Multinom}(\alpha)$$

14.4 Math behind Mixture Models

Now, we delve more deeply into the math behind using such mixture models. To start, the models involve x (data), z (latent variables), π (prior of latent variable), and $\{\mu_k\}$ (parameters for conditional distribution). We will use x and z to denote all n observations of the data and latent variables.

The **complete data likelihood** of the model is then

$$p(x, z) = \prod_n \prod_k \pi_k p(x_n|\mu_k)^{z_{nk}},$$

where z_{nk} is shorthand for $\mathbf{1}(Z_n = k)$.
To get the **marginal likelihood** of the data, we can marginalize and sum over all possible values of z_n:

$$p(x) = \sum_z \prod_n \prod_k \pi_k p(x_n|\mu_k)^{z_{nk}} = \prod_n \underbrace{\sum_z}_{\text{Hard to compute}} \prod_k \pi_k p(x_n|\mu_k)^{z_{nk}}$$

Although we pushed the sum into the product, the sum over all possible values of z is still hard to compute. Our main alternative strategy is thus to utilize the complete data likelihood in order to find $p(x)$.
Below is a heuristic algorithm to find $p(x)$:

1. Initialize π and μ randomly.

2. Compute the log-likelihood $p(x, z|\pi, \mu) = \sum_n \sum_k z_{nk} \log \pi_k + z_{nk} \log p(x_n|\mu_k)$ using fixed π and μ.

3. Denote $q_{nk} = p(z_n = k|x_n; ...)$ as a "hallucinated" distribution over z_{nk}; the origin of q_{nk} is unclear, but it satisfies the given requirements. We can use this distribution of $z \sim q_{nk}$ to compute the expectation of the log-likelihood as:

$$\mathbb{E}_{z\sim q_{nk}} [\log p(x_n, z_n|\pi, \mu)] = \sum_n \sum_k q_{nk} \log \pi_k + q_{nk} \log p(x_n|\mu_k).$$

   We can then "hallucinate" parameters through the distribution of z_{nk}! Returning to the graphical model, recall that it was difficult to actually compute the parameters. Howevere, given some parameters π, μ, we can compute a marginal distribution over the z: $p(z|x, \pi, \mu)$.

4. We thus can perform a coordinate ascent algorithm; that is, we optimize one set of parameters, then the other, then continue to alternate optimization.

This heuristic is summarized below:

1: **procedure** EM_NB(x, z)
2: Initialize π and μ randomly.
3: **while** π and μ not converged **do**
4: (*Expectation*) Compute $q_{nk} = p(z|x, \pi, \mu)$ using fixed π and μ
5: (*Maximization*) Compute MLE of π and μ using q through $\mathbb{E}_{z\sim q} [\log p(x, z)]$

The above is called the **Expectation Maximization** (EM) Algorithm, where step 4 computes the actual expectation maximization, and step 5 actually maximizes through the MLE. π will be Categorical and μ will be the means of the Gaussians themselves here.

14.5 Expectation Maximization Algorithm

The above algorithm can be generalized, and is known as the **Expectation Maximization** (EM) algorithm.

1: **procedure** EM(x,z)
2: Initialize π and μ randomly.
3: **while** π and μ not converged **do**
4: (*Expectation*) $q_{nk} \leftarrow \dfrac{\pi_k p(x_n|\mu_c)}{\sum_{k'} \pi_{k'} p(x_n|\mu_{k'})}$
5: (*Maximization*) $\mathbb{E}_{z\sim q} [\log p(x, z|\pi, \mu)] = \sum_n \sum_k q_{nk} \log \pi_k + q_{nk} \log p(x_n|\mu_k)$

We'll thinking about this algorithm in the context of information theory. In the *maximization* step, we have the first term $\sum_n \sum_k q_{nk} \log \pi_k$ can be thought about as maximizing the π_k under the model, with the assumption they were sampled from distribution of q_{nk}.

$$\sum_n \sum_k \underbrace{q_{nk}}_{\text{Came from } q_{nk} \text{ guess}} \log \left(\underbrace{\pi_k}_{\text{Want to find } \pi_k} \right)$$

So, we can think of q_{nk} as "expected counts" from a distribution. So, the MLE is similar to standard counts of observations. In particular,

$$\pi_k = \frac{\sum_n q_{nk}}{\sum_{n,k} q_{nk}}.$$

Alternatively, we could also use the MAP estimate. The main point is that the prior is embedded in the max step.
A similar update happens for μ, but will be specific to the actual form of the model $p(x|\mu)$.

14.5.1 Assumptions on EM

One key point is that the model yields an easily computable $p(z|x, \pi, \mu)$. If we imagine that $p(x|z)$ is an arbitrary neural net with continuous output, this could be very difficult. A specific example is a Variational Autoencoder (VAE), which could be found from Kingsma and Welling.

14.6 Demo on EM

We did a demonstration on Expectation Maximization.
In general, we saw it was pretty hard to even infer the clusters of data especially as the number of true clusters increased.
Q: *How to infer number of true clusters?* It's actually a hard problem, no really good standout way to do this.
Q: *Difference between k-means and EM here?* k-means will choose q as the highest probability, instead of assigning a distribution to q_{nk} as EM does.

14.7 Theory behind Expectation Maximization

We'll discuss why the EM algorithm works; specifically, why q_{nk} eventually converges to $p(z_n|x_n, \pi, \mu)$.
First, consider the marginal distribution of the data $p(x) = \prod_n \sum_{z_n} p(x_n, z_n|\pi, \mu)$. We have that the log-marginal is (in terms of entropy)

$$\begin{aligned}
\log p(x) &= \sum_n \log \left[\sum_{z_n} p(x_n, z_n, |\pi, \mu) \right] \\
&= \sum_n \log \left[\sum_{z_n} q(z_n = k) \frac{p(x_n, z_n|\pi, \mu)}{q(z_n = k)} \right] \\
&= \sum_n \log \mathbb{E}_q \left[\frac{p(x_n, z_n|\pi, \mu)}{q(z_n = k)} \right] \\
&\geq \sum_n \sum_k q(z_{nk}) \log \frac{p(x_n, z_n|\pi, \mu_k)}{q(z_n)} \\
&= \left[\sum_n \sum_k q(z_{nk}) \log p(x_n, z_n|\pi, \mu_k) \right] - \sum_n \left[\sum_k q(z_{nk}) \log q(z_n) \right] \\
&= \sum_n \sum_k q_{nk} \log p(x_n, z_n = k|\pi, \mu_k) + \sum_n H\left[q(z_n)\right]
\end{aligned}$$

In the second line we only multiplied by $1 = \frac{q(z_n = k)}{q(z_n = k)}$. In the third line, we use the trick of applying Jensen's inequality when we see the log of an expectation.
In now analyzing the M-step, we are computing

$$\pi, \mu \leftarrow \underset{\pi,\mu}{\operatorname{argmax}} \sum_n \sum_k q_{nk} \log \pi_k + q_{nk} \log p(x_n|\mu_k)$$

where $\sum_k q_{nk} \log \pi_k$ is the cross-entropy between parameters and expected counts.

The E-step, we find q_{nk}.

$$\begin{aligned}\arg\min_q \sum_n \left[\sum_k q(z) \log\left(\frac{p(z|x)p(x)}{q(z)}\right)\right] &= \arg\min_q \sum_n \left[\sum_k q(z) \log \frac{p(z|x)}{q(z)} + \text{const}\right] \\ &= \arg\min_q \sum_n KL\left(q(z) \,||\, p(z|x)\right),\end{aligned}$$

and we can minimize the KL divergence term by setting $q_{nk} = p(z_n|x_n, \pi, \mu)$ as the same distribution.

CS281: Advanced ML **October 30, 2017**

Lecture 15: Mean Field

Lecturer: Sasha Rush *Scribes: Alex Lin, Wei Zhang, Daniel Giebisch, Richard Zhang, Fahad Alhasoun*

15.1 Variational Inference

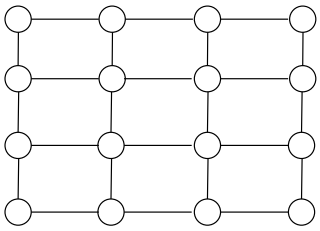
In inference, we're interested in computing $p(z|D)$, the marginal distribution of a node z, given D, where D is all of the evidence in the model. However, very often, direct computation with exact methods is too difficult. As a result the most feasible approach may be to find a $q(z)$ that reasonably approximates $p(z)$ where q is easy to compute. Thus, we define

$$q^* = argmin_{q \in EASY} d(q, p)$$

where EASY is the set of functions that are easy to compute and find the q* such that by some metric d, q* is the closest to p (or smallest gap). Note that the function p can be anything.

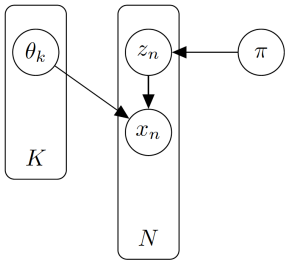
15.2 Hard models

- Ising model $P(y)$



Recall the Ising model from a few lectures ago, a grid of nodes connected horizontally and vertically to adjacent nodes on 4 sides (or 3 on the edges and 2 on the corners). Things in which we may be interested include the marginal distributions $p(y_{ij} = 1)$ and the partition function. Unlike for a tree structure, direct computation using loopy belief propagation may not converge well for this graph with cycles, so we will look for a different method.

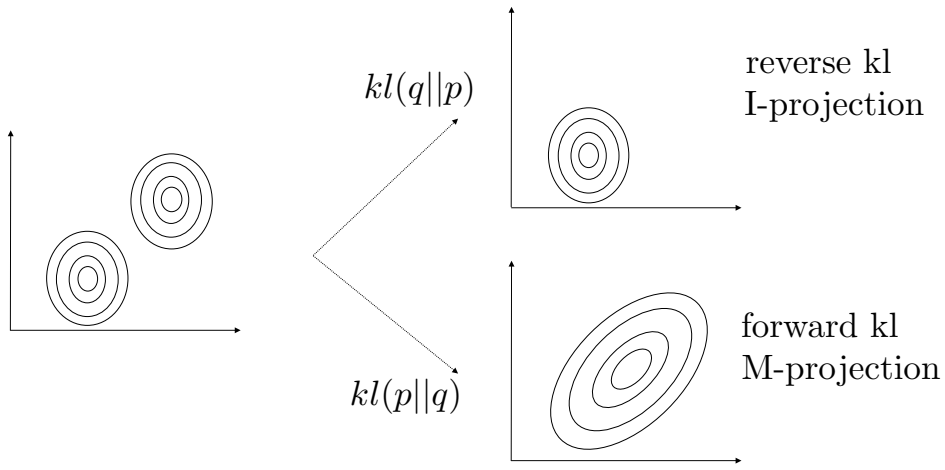
- Gaussian Mixture Model



$$\pi \sim dir(\alpha)$$
$$\Theta_k = (\mu_k, \Sigma_k)$$
$$Z_n \sim \pi$$
$$X_n|(Z_n, \Theta) = N(\mu_k, \Sigma_k)$$

Recall the Gaussian Mixture Model from last lecture, an unsupervised learning method that attempts to categorize data points into clusters. This is nearly the same graphical model as Naive Bayes classification, except that the classes are not observed, but rather inferred. Again, computation of $p(x_n)$ is rather difficult because z_n are not observed. Therefore, one must find a method that infers the z_n in order to infer $p(x_n)$.

15.3 Variational Idea



$$min_q d(p,q) = KL(q\|p) = \int q(z) log \frac{q(z)}{p(z|D)}$$

In trying find the gap function $d(p,q)$ that measures our gap from p to q, one good choice is the KL divergence, but because it is asymmetric, we essentially have 2 options, both of which lead to valid methods:

- $KL(q\|p)$
  The reverse KL, I-projection
- $KL(p\|q)$
  The forward KL, M-projection

The forward KL method is also known as the Expectation Propagation (EP) method.

15.4 Relationship to EM

Recall the Expectation-Maximization (EM) method from last lecture. Much of the mathematics we will use resembles that which we have already studied.

$$\begin{aligned}
\log p(D) &= \log \int_z p(D,z)dz && \text{(D is the observed data set)} \\
&= \log \int_z q(z)p(D,z)/q(z)dz \\
&= \log E_{z\sim q}\left[\frac{p(D,z)}{q(D)}\right] \\
&\geq E_{z\sim q}\left[\log \frac{p(D,z)}{q(D)}\right] && \text{(the lower bound, by Jensen's inequality)}
\end{aligned}$$

$$\begin{aligned}
\log p(D) - E_q \log \frac{p(D,z)}{q(z)} &= E_q\left[\log p(D) - \log \frac{p(D,z)}{q(z)}\right] \\
&= E_q\left[\log \frac{q(z)}{p(z|D)}\right] \\
&= KL(q\|p)
\end{aligned}$$

One more remaining issue is that

$$p(z|D) = \frac{p(z,D)}{p(D)}$$

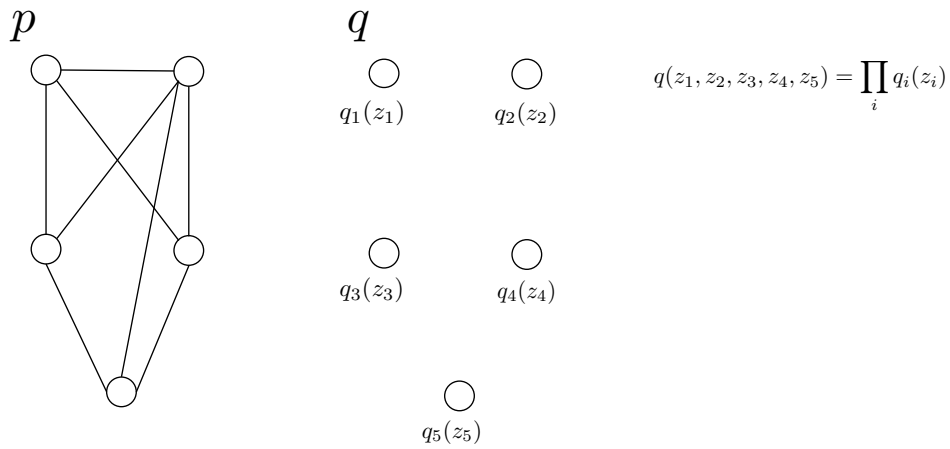
where $p(D)$ is generally very difficult to compute and the main reason why computing p exactly becomes an intractable problem. However, for the purposes of finding q, we can ignore $p(D)$ as just a constant, since it is independent of z. Thus,

$$\min_q KL(q\|p) = \min_q E\left[\log \frac{q(z)}{p(z,D)}\right]$$

Comparison to EM

- In variational inference, we have coordinate ascent just like in EM, except we try to optimize a lower bound.
- In variational inference, we pick q from the EASY set. As a result, we don't have to select point estimates, but rather we can use entire distributions.
- This is very useful in Bayesian setups.
- We can also combine this with EM and sampling to obtain more sophisticated techniques as well.

15.5 Mean Field



This is an algorithm that optimizes one particular q_i, via an expectation over the adjacent variables, while fixing all of the other q and then iterates this procedure over all of the q. It proceeds as follows:

- We assume we have all q except q_i.
- Select $q(z) = \prod q_i(z_i)$ from our EASY set.
- Recall that the goal is to reduce the gap $min_q KL(q\|p)$. We do this by fitting each q_i as

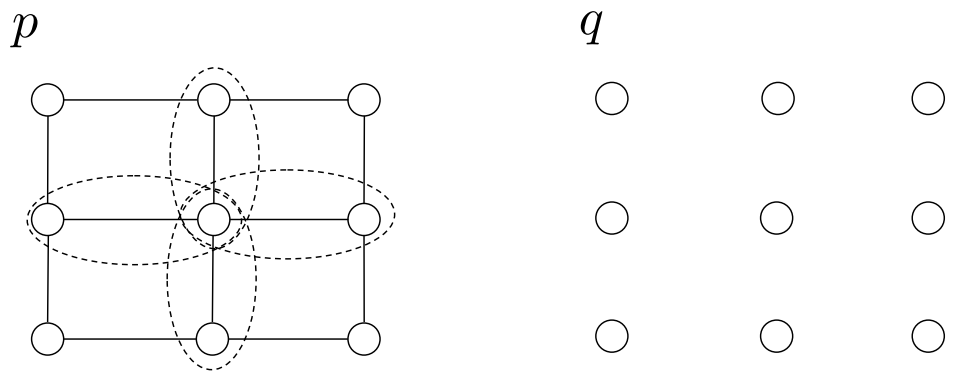
$$q_i^* \leftarrow argmin_q KL(q\|p)$$

where

$$\begin{aligned}
argmin_q KL(q\|p) &= argmin_{q_i} - H(q_i) - E_q \log(p(z)) \\
&= argmin_{q_i} - H(q_i) - \int_z (\prod_i q_i(z_i)) \log p(z) + other(j \neq i) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\text{q(z)=}\textstyle\prod q_i(z_i)) \\
&= argmin_{q_i} - H(q_i) - \int_{z_i} q_i(z_i) \int_{z_j : j \neq i} \prod_{j \neq i} q_i(z_i) \log p(z) + ... \\
&= argmin_{q_i} - H(q_i) - \int_{z_i} q_i(z_i) \log f_i \qquad\qquad (\log f_i = E_{q_i}[\log p(z)]) \\
&= argmin_{q_i} KL(q_i\|f_i) \qquad\qquad (f_i \text{ is not distribution , but still OK}) \\
q_i &\propto exp\{E_{-q_i}(log(p(z)))\}
\end{aligned}$$

Note that E_{-q_i} denotes an expectation taken over all the variables except z_i.

15.6 Ising Model



We now apply Mean Field Variational Inference to the Ising Model. Note that θ_{v_i} denotes the log-potential of vertex i while $\theta_{E_{i-n}}$ denotes the log-potential of the edge from vertex i to vertex n. We compute q_i as an expectation over the 4 neighboring nodes, the Markov blanket. This is then

$$\begin{aligned}
q_i &\propto exp[E_{qi} \log p(z)] \\
p(z) &\propto exp[\theta_v^T z + z^T \theta_E z] \\
\log q_i(x_i) &\propto E_{-q_i}[\log p(z)] \\
&= E_{-q_i}[\theta_v^T z + z^T \theta_E z] \\
&= E_{-q_i}[z_i \theta_{v_i} + \Sigma_n \theta_{E_{i-n}} Z_i Z_n] \\
&= Z_i \theta_{v_i} + \Sigma_{n \in neighbor} \theta_{E_{i-n}} Z_i E_{q_n}[Z_n] \\
&= Z_i \theta_{v_i} + \Sigma_n \theta_{E_{i-n}} Z_i q_n(1)
\end{aligned}$$

Therefore

$$q_i \propto exp[Z_i \theta_{v_i} + \Sigma_n \theta_{E_{i-n}} Z_i q_n(1)]$$

We can then repeat this procedure over all nodes to update the entire graph. Then we can repeat that over several epochs until we find good convergence of q.

CS281: Advanced ML **November 1, 2017**

Lecture 16: Variational Inference Part 2

Lecturer: Sasha Rush *Scribes: Denis Turcu,Xu Si,Jiayu Yao*

16.1 Announcements

- T4 out, due 9/13 at 5pm
- Exams available in office
- OH - today 2:30-4pm (Wednesdays)
- Follow formatting for the abstract of the final project. There were many inconsistencies with the formatting requirements for the initial proposal.

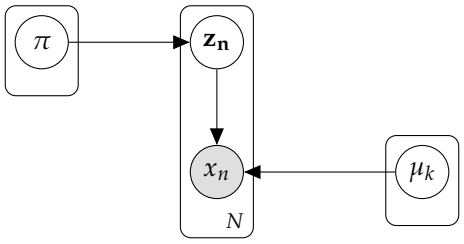
16.2 Introduction

Last class, we talked about variational inference. This class, we gonna talk about a very different type of VI. We also will talk about some other types of VI but will not go too much into the details.

Murphy's book, especially Chapter 22, covers many details on the theory side. The other text, Murphy referred as "The Monster", we put online as a textbook written by Michael Jordan.

16.3 Bayesian GMM

We are going to talk more about variational inference. We also put another reference online called VI: A Review for Statisticians. It covers in great detail of Bayesian GMM, so let's write down that model:



We assume:

$$\mu_k \sim \mathcal{N}(0, \sigma^2) \; \forall \; k$$
$$z_n \sim Cat(\frac{1}{k}, \ldots, \frac{1}{k}) \; \forall \; k$$
$$x_n | z_n, \mu \sim \mathcal{N}(\mu_{z_n}, 1) \; \forall \; n.$$

Then we write:

$$p(\{x_n\}, \{z_n\}, \mu) = p(\mu) \prod_n p(z_n) p(x_n | z_n, \mu).$$

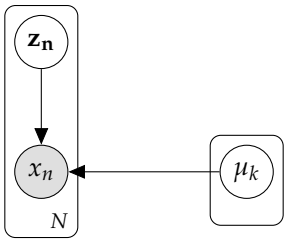
And we get:

$$p(x) = \int_{z,\mu} p(x, z, \mu) = \int p(\mu) \prod_n \sum_{z_n} p(z_n) p(x_n | z_n, \mu) \mathrm{d}\mu.$$

Variation setup. Goal:

$$\min_{q \in EASY} KL(q||p)$$

reverse KL.
We pick $EASY$ as mean field.



p

q

Variational parametrization:

$$q(\mu, z) = \prod_k q_k(\mu_k) \prod_n q_n(z_n).$$

$$q_n(z_n; \lambda_n^z) \qquad Cat(\lambda_n^z).$$

$$q_k(\mu_k; \lambda_k^\mu, \lambda_k^{\sigma^2}) \qquad \mathcal{N}(\lambda_k^\mu, \lambda_k^{\sigma^2}).$$

$$\arg\min_{q \in EASY} KL(q||p) = \arg\min_\lambda KL\left(\prod_k q_k(\mu_k; \lambda_k^\mu, \lambda_k^{\sigma^2}) \prod_n q_n(z_n; \lambda_n^z) \;||\; p\right)$$

"When we do *mean field*?"

$$q_i \sim \exp[\mathbb{E}_{-q_i} \log(p(z, x))]$$

Brief interlude: coordinate ascent $\rightarrow$ CAVI (coordinates ascent variational inference). Doing each individual one at a time.

- Bound we are optimizing is non-convex.
- This method is monotonically increasing.
- Sensitive to initialization $\rightarrow$ common for random restarts.

Example, deriving the math for GMM can be useful to understand how it works and how we will do mean field updates. Start from the above, setup the problem:

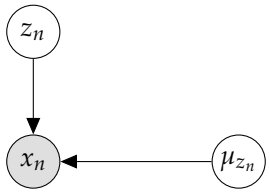
$$\mu_k \sim \mathcal{N}(0, \sigma^2) \; \forall \; k$$

$$z_n \sim Cat(\frac{1}{k}, \ldots, \frac{1}{k}) \; \forall \; k$$

$$x_n | z_n, \mu \sim \mathcal{N}(\mu_{z_n}, 1) \; \forall \; n$$

where we also have λ_n^z for hidden switch variable, and $\lambda_k^m, \lambda_k^{s^2}$ for the Gaussians.

$$
\begin{aligned}
q_n(z_n; \lambda_n^z) &\propto exp[\mathbb{E}_{-q_n} \log(p(\mu, z, x))] \\
&\propto exp[\mathbb{E}_{-q_n} \log(p(x_n|z_n, \mu_{z_n}))] \\
&\propto exp[\mathbb{E}_{-q_n} - (x_n - \mu_{z_n})^2/z] \\
&\propto exp[\mathbb{E}_{-q_n}(x_n \mu_{z_n} - {\mu_{z_n}}^2/2)] \\
&\propto exp[x_n \mathbb{E}_{-q_n}(\mu_{z_n}) - \mathbb{E}_{-q_n}({\mu_{z_n}}^2)/2]
\end{aligned}
$$



So we identify $\mathbb{E}_{-q_n}(\mu_{z_n})$ with $\lambda^m_{k=z_n}$ and $\mathbb{E}_{-q_n}({\mu_{z_n}}^2)$ with $\lambda_k^{s^2}$, and then we can write:

$$
\begin{aligned}
q_k(\mu_k; \lambda^m_{k=z_n}, \lambda_k^{s^2}) &\propto exp[\mathbb{E}_{-q_n}(\log p(\mu_k) + \sum_n \log p(x_n|z_n, \mu)] \\
&= -{\mu_k}^2/(2\sigma^2) + \sum_n \mathbb{E}(\log p(x_n|z_n, \mu)) \\
&= -{\mu_k}^2/(2\sigma^2) + \sum_n \mathbb{E}(z_{nk}(\log p(x_n|\mu_k)) \\
&= -{\mu_k}^2/(2\sigma^2) + \sum_n \mathbb{E}_{-q_n}(z_{nk})(\log p(x_n|\mu_k)) \\
&= -{\mu_k}^2/(2\sigma^2) + \sum_n \lambda^z_{nk}(-(x_n - \mu_{z_n}))^2/2 + const. \\
&= (\sum_k \lambda^z_{nk} x_n)\mu_k - (\frac{\sigma^2}{2} + \sum \lambda^z_{nk}/2){\mu_k}^2 + const.
\end{aligned}
$$

Then:

$$q_k(\mu_k) = exp[\theta^T \phi - A + \dots],$$

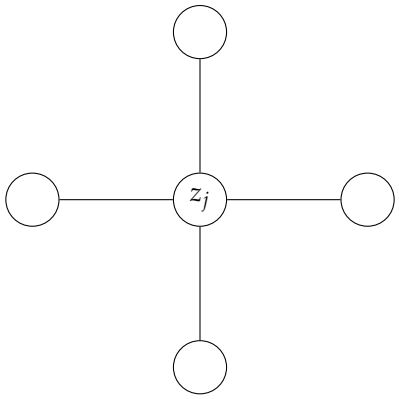
where $\theta_1 = \sum_n \lambda^z_{nk} x_n,\ \theta_2 = -(\frac{\sigma^2}{2} + \sum \lambda^z_{nk}/2)$ and $\phi_1 = \mu_k,\ \phi_2 = {\mu_k}^2$, as in GLM.
For normal distribution, we have:

$$\lambda^m_k = \frac{\sum_n \lambda^z_{nk} x_n}{1/\sigma^2 + \sum_n \lambda^z_{nk}}, \text{ and } \lambda_k^{s^2} = \frac{1}{1/\sigma^2 + \sum_n \lambda^z_{nk}}.$$

16.4 Exponential Family

$$p(z_j|z_{-j}, x) = h(z_j) exp(\theta^T \phi(z_j) - A(\theta)),$$

where θ are function of z_{-j}, x. One nice case is *UGM*:



blanket

$$q(z) = \prod_j q(z_j)$$

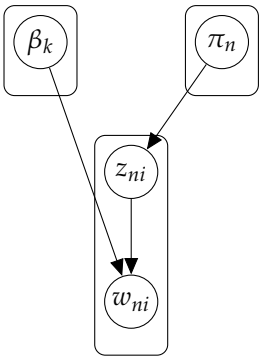
$$\begin{aligned} q(z) &\propto exp[\mathbb{E}_{-q_j} \log p(z_j | z_{-j}, x)] \\ &\propto exp[\log(h) + \mathbb{E}(\theta)_{z_j}^T - \mathbb{E}(A(\theta))] \\ &\propto h(z_j) exp[\mathbb{E}(\theta)^T z_j] \end{aligned}$$

where $\mathbb{E}(\theta)^T$ are the natural parameters of the variational approximation

$$\lambda_j = \mathbb{E}[\theta(z_{-j}, x)]$$

16.5 Latent Dirichlet Allocation

- Widely used generative latent variable model
- generative model set up



  where $\beta_k \sim Dir(\eta)$, $\pi_n \sim Dir(\alpha)$, $z_{ni} \sim Cat(\pi_n)$, $w_{ni} \sim Cat(\beta_{z_{ni}})$

- Topic molding story:
  - n - documents
  - i - words
  - π_n - document topic distribution
  - β_k - topic-word distribution

- z_{ni} - topic selected for word i of document n
- w_{ni} - word selected for ni.
- λ^{z}_{ni} - probability for the topic of word i in document n

16.6 Demo

We did an example iPython notebook (`TopicModeling.ipynb`) in class.

CS281: Advanced ML **November 6, 2017**

Lecture 17: Loopy BP, Gibbs Sampling, and VI with Gradients

Lecturer: Sasha Rush *Scribes: Patrick Varin, Lillian Pentecost, André Snoeck, Baojia Tong, Hsuan Lee*

17.1 Loopy BP

17.1.1 History

The history of Loopy Believe Propagation (Loopy-BP) began in 1988 with Judea Pearl who tried to analyze the behavior of the BP algorithm (which gives exact marginal inference on tree) on graphs that are not trees, like the Ising model. Remember the message passing algorithms is

$$m_{s\to t}(x_t) = \sum_{x_s} \psi(x_s)\psi(x_s, x_t) \prod_{u\in \text{NBR}(s)-t} m_{u\to s}(x_s)$$
$$bel_s(x_s) \propto \psi(x_s) \prod_{t\in \text{NBR}(s)} m_{t\to s}(x_s)$$

Note that this algorithm doesn't require an ordering on the nodes, so it can naturally extend to cyclic graphs. Around 1998, a decade after Pearl raised the question of the BP algorithm on general graphs, papers studying *Turbo coding* or *Low density parity check* (LDPC) codes used the BP algorithm on cyclical graphs with empirical success, motivating more research. This research drew parallels between loopy-BP and variational inference.

17.1.2 Implementations

We can implement loopy belief propagation either *synchronously* or *asynchronously*

- Synchronous Updates: All of the nodes are updated togethers, so that every node at iteration t depends on it's Markov blanket at $t-1$
  - parallelizable
  - usually takes more updates
- Asynchronous Updates: The nodes are given an order and updated sequentially
  - sequential
  - usually converges in fewer updates

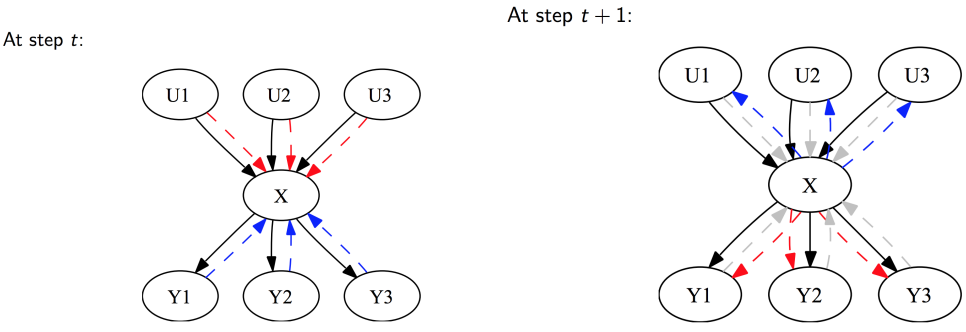


Figure 17.18: Message passing example: left (incoming), right (outgoing)

These updates look similar to mean field variational inference (VI) with some differences in performance.

- Loopy BP is exact on trees, mean-field variational inference is not.
- Loopy BP is not guaranteed to converge, where VI will.
- If Loopy BP converges it is not guaranteed to converge to a local optimum.

Loopy BP is generally more costly than mean field VI, because it stores node beliefs in addition to edge messages, whereas mean field VI only stores node parameters. If the graph is large and dense, then Loopy BP might be very costly.
In practice, because Loopy BP contains more information about the problem than mean field VI it tends to work better.

17.2 Gibbs Sampling (Preview)

Gibbs Sampling is a different method for approximate inference. It is a special case of Markov Chain Monte Carlo (MCMC), which will be covered in more detail in coming lectures.
The main idea of Gibbs Sampling is to re-estimate the posteriors for x_i assuming that we have we have explicit point estimates for all the other nodes in the graph (x_{-i}):

$$\tilde{x}_i \sim P(x_i|\tilde{x}_{-i})$$

Naturally, we only care about the Markov blanket when estimating a particular node. Gibbs sampling is not a parallel algorithm, the samples have to be generated sequentially. However, we can go through our graph in any particular order and update each node accordingly.

17.2.1 Example: Gibbs Sampling Topic Modeling

In the following example, we present the Gibbs update associated to the topic modeling that we defined in Lecture 16. Remember that

$$\begin{aligned}
\beta_k &\sim Dir(\eta) && k \text{ topics} \\
\pi_n &\sim Dir(\alpha) && n \text{ documents} \\
z_{ni} &\sim Cat(\pi_n) && \text{Draw topic for each word and each document} \\
w_{ni} &\sim Dir(\beta_{z_{ni}}) && \text{Draw a word from a topic}
\end{aligned}$$

and the full joint distribution is given by

$$p(\beta_k, \pi_n, \{z\}, \{w\}) = \prod_k p(\beta_k) \prod_n p(\pi_n) \prod_{n,i} p(z_{ni}|\pi_n) p(w_{ni}|z_{ni})$$

See Figure 17.19 for reference.
The updates for Gibbs sampling are similar to those of Mean Field (see Lecture 16), albeit much simpler. We need to compute the conditional probability for each of the following

$$\begin{aligned}
z_{ni} = k|\pi_n, w_{ni} &\propto \pi_{nk}\beta_{k,w_{ni}} \\
\pi_n|\alpha, z_{ni} &= Dir(a_k + \sum_i \mathbb{1}(z_{ni} = k)) \\
\beta_k = d|\eta, z_{ni} &= Dir(\eta_d + \sum_i \mathbb{1}(z_{ni} = k, w_{ni} = d))
\end{aligned}$$

The last two terms are essentially the full posterior that we saw in lecture 2. Although these updates are rather simple, it is used in many applications given it works with a wide variety of models. Typically, one would draw one sample per update. It is not clear whether drawing multiple samples or running more epochs helps the algorithm converge faster.

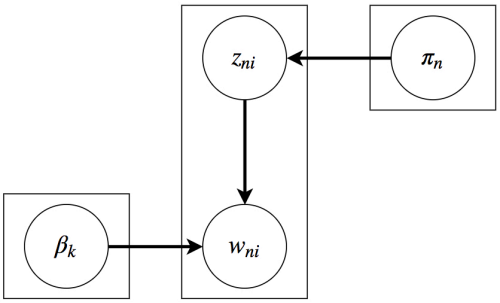


Figure 17.19: Graphical model associated to topic modeling example

17.3 Variational Inference with Gradients

17.3.1 Introduction

Variational inference with gradients is also referred to as black box variational inference or stochastic variational inference (SVI) and is actively gaining traction in the ML commnity today. For example, UberAI just released a new framework on top of pytorch called *pyro* to support SVI. One reason why this is attractive is because SVI integrates well with autograd and deep learning.

17.3.2 Aside on Monte Carlo Sampling

If we consider some expectation over a distribution p of some function $f(z)$, as given below for discrete z, we often find this is intractable to compute.

$$\mathbb{E}_{z\sim p}[f(z)] = \sum_{z'} p(z')f(z')$$

We can approximate this expectation by sampling some N number of $\tilde{z}$, with each sample represented by $\tilde{z}_n$, which results in the following expression for the expectation:

$$\mathbb{E}_{z\sim p}[f(z)] \approx \frac{1}{N}\sum_{n=1}^{N} f(\tilde{z}_n)$$

This is much simpler to compute, and it can be shown that the resulting $\tilde{\mu}$ satisfies that $(\tilde{\mu} - \mu) \sim \mathcal{N}(0, \frac{\sigma^2}{N})$, which implies that the variance of the approximation with respect to the true mean decreases with the number of samples N.

17.3.3 Rough Idea

We will apply gradient optimization to the lower bound on $p(x)$ that we use to derive the variational objective. We use the formulation of the Gaussian Mixture Model (GMM) and Mean Field (MF) approximation with variational parameters, developed in previous lectures, as an example. This is potentially more attractive than coordinate ascent because for a very large dataset compared to the number of classes (i.e. $n >> k$ in the GMM MF configuration) coordinate ascent requires (1) iterating through all data to perform updates to the variational parameters and (2) storing variational parameters per-example (the number of the parameters grows with the size of the dataset).
The two wishes/ideas of this approach will be to (1) update global parameters (λ_k) on mini-batches to avoid the need to iterate over all samples (all λ_n) to perform update and (2) avoid storage of a dedicated λ_n for each n. The remainder of the lecture will focus on idea (1) above.

17.3.4 Deriving the Variational Objective

Letting $\theta = z_n, \mu_k$, we would like $p(x)$ for the GMM MF configuration previously developed. We aim to maximize the following lower bound (also from previous lecture) in order to approach $p(x)$, where q_λ is any q with variational parameters λ.

$$p(x) \geq \mathbb{E}_q[\log \frac{p(x,\theta)}{q_\lambda(\theta)}]$$

$$\max_q \mathbb{E}_q[\log \frac{p(x,\theta)}{q_\lambda(\theta)}]$$

Expressing $p(x,\theta) = p(\theta)p(x|\theta)$ and distributing the log, we find the variational objective, and we can interpret each of the three terms:

$$\max_q -\mathbb{E}_q[\log q_\lambda(\theta)] + \mathbb{E}_q[\log p(\theta)] + \mathbb{E}_q[\log p(x|\theta)]$$

In the expression above, we can recognize that the first term is just an entropy term, the second term includes the prior to effectively find a q close to $p(\theta)$ based on cross-entropy, and the third term contains the likelihood to reflect how well θ predicts the data. Furthermore, we see that the first two terms together form the negative inverse KL divergence of p and q, so we further simplify to the expression below:

$$L(\lambda) = \max_q -KL(q(\theta)||p(\theta)) + \mathbb{E}_q(\log p(x|\theta))$$

In the expression above, the KL divergence will act to tend q towards the prior $p(\theta)$ and the expectation term will give weight in q to parameters that tend to explain x.

17.3.5 Optimization via Stochastic Gradient Descent (SGD)

Next, we aim to apply optimization via SGD, and the $L(\lambda)$ given above becomes a loss function; we will maximize the value of this function by computing the gradient with respect to the parameters we care about (λ). Here, we will take a mini-batch of x values and compute $\nabla_\lambda L(\lambda)$. However, we notice that the expectation term in $L(\lambda)$ over q must be approximated, and the two methods to compute an approximation are (1) reinforcement and (2) reparameterization. This lecture only covers method (2), reparameterization, which also appears on HW4.
Reparameterization requires known q of a certain type and removes q from the expectation. Beginning with the form of the expectation, we can reformulate in terms of a change of variable from distribution q to the standard normal (letting $\theta = Az + b$) and then apply the Monte Carlo approximation to simplify the expression and make it computationally tractable:

$$\mathbb{E}_{q_\lambda}[p(x|\theta)] = \mathbb{E}_{z\sim\mathcal{N}(0,1)}[p(x|f_\lambda(z))]$$

$$\nabla_\lambda \mathbb{E}_{z\sim\mathcal{N}(0,1)}[p(x|f_\lambda(z))] = \mathbb{E}_{z\sim\mathcal{N}(0,1)}[\nabla_\lambda p(x|f_\lambda(z))]$$

$$\nabla_\lambda \mathbb{E}_{q_\lambda}[p(x|\theta)] \approx \frac{1}{N}\sum_{n=1}^{N} \nabla_\lambda p(x|f_\lambda(\tilde{z}_n))$$

CS281: Advanced ML **November 8, 2017**

Lecture 18: Variational Auto-encoders and GANs

Lecturer: Sasha Rush Scribes: Jeff Chang, Michael Els, Zhixian Lei, Yuting Sun, Xincheng You, Srivatsan Srinivasan

Today we are going to go a step forward in Variational Inference. We are going to combine the variational inference and neural networks.

18.1 Autoencoder

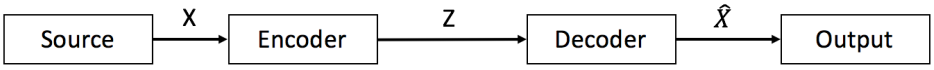
Suppose we receive a signal x from a source. Then we encode x with encoding function Enc

$$z = Enc(x)$$

so that z is simpler than x, for example, z can be lower dimension than x or the learned encoding function itself can be a simple function. The goal is to limit the capacity of the model by parameterising enc and $p()$, and to learn them from the data. This leads to the model learning the optimal compression for the data which only saves the most important information and discards the rest. This encoding is often nonlinear in practice, for example, people often use neural networks. Then z might be changed to z' by noise. After that, we want to recover x by inference:

$$\hat{x} = \text{argmax}_{x'} \Pr(x = x'|z')$$

This is basically what autoencoder does. For variational autoencoder, we need to parameterize function Enc and $\Pr(x = x'|z')$ usually with neural networks. We might want to learn these functions from data.



18.2 Back to Variational Inference

$$\begin{aligned}
\log p(x) &= \log \int_\theta p(\theta)p(x|\theta)d\theta && (18.44)\\
&= \log \int_\theta q(\theta)(p(\theta)/q(\theta))p(x|\theta)d\theta && (18.45)\\
&\geq E_q\left[\log \frac{p(\theta)}{q(\theta)}p(x|\theta)\right] && (18.46)\\
&= -\text{KL}(q(\theta)|p(\theta)) + E_q[\log p(x|\theta)] && (18.47)
\end{aligned}$$

where the first term makes q close to prior p and the second term is the likelihood based on q. There are two methods to optimization above expression

- Coordinates Ascent (CAVI)
- Stochastic Gradient Descent (SGD)

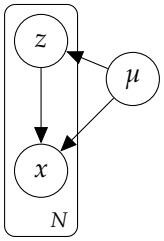
where CAVI is the method we have introduced in previous lectures and SGD is just directly optimize this function.
About the choice of q-function, we have following several methods:

- We can make mean field assumption (which requires the full Markov blanket), then $q(\theta) = \prod_i q_i(\theta)$

- We can also do structured mean field. Basically we can divide this graph into several simple subgraphs (they can overlap each other). Inside each subgraphs we can do belief propagation using for example forward-backward algorithm and outside of the subgraphs we use mean field approximation. This q is closer to the original and has a better bound.
- More generally we can use neural networks, where λ are neural network parameters and we use q_λ to approximate the posterior.

18.3 Variational Autoencoder

We have learned many models in this class. The Ising model is an example that is difficult to do inference on. There are also models that we can't use EM on. An example of such a model is illustrated here:



Since the parameters are not fixed, the EM fails in this case. Using this as a motivating example, let's examine Variational Autoencoders. In our previous examples for inference, we always consider the conditional distribution as discrete distribution or just linear Gaussian distribution, and that we can calculate our posterior in a straightforward manner. But in general $p(\text{Data}|\text{Label})$ can be non-linear, for example, neural network. Then $p(\text{Label}|\text{Data})$ becomes intractable. In this case, we should make q-function as a neural network. Suppose z are labels, x are data and μ are parameters. And we assume

$$\Pr(x|z,\mu) = \text{softmax}(w\phi(x;\mu))$$

where ϕ is a neural network. We use neural networks to compute q functions

$$q_\lambda(z) = \mathcal{N}(w^1\phi_1(x;\lambda), w^2\phi_2(x;\lambda))$$

where ϕ_1 and ϕ_2 are neural networks. w^1 and w^2 are transformation parameters.
We can interpret this process as autoencoder. We first encode x into a distribution

$$q_\lambda(z) \sim \mathcal{N}(w^1\phi_1(x;\lambda), w^2\phi_2(x;\lambda))$$

Then upon getting some sample z from $q_\lambda(z)$, we decode x as $\hat{x}$ where

$$\hat{x} = \text{argmax}_x \log p(x|z)$$

To make $\hat{x}$ be close to x, we should optimize the parameter λ by

$$\max_\lambda -\text{KL}(q(\theta)||p(\theta)) + E_q \log p(x|\theta) + E_q \log p(x|\theta)$$

We use gradient method to optimize λ where the gradient can be written as

$$\nabla_\lambda - KL(q(\theta)||p(\theta)) + \nabla_\lambda E_q \log p(x|\theta)$$

where the latter term can be approximated via reparameterization

$$\nabla_\lambda E_q \log p(x|\theta) \sim \frac{1}{N}\sum_{n=1}^{N} \nabla_\lambda \log p(x|f_\lambda(s_n))$$

after we obtain optimized parameter λ from large amount to data x, we will obtain an $q_\lambda(x)$ as encoding function for all x and we can use it for future encoding and recovery and even generate new x from encoding and decoding processes.

18.4 Sampling from other distribution

One typical problem when we want to do optimization is that we don't know how to compute

$$\nabla_\lambda E_{q_\lambda}[\log p(x|\theta)]$$

There are basically two ways to treat this problem

- Reparameterization. We can transform E_{q_λ} to other fixed distribution and sampling on that distribution by translating the parameter. Then we can move ∇ inside the expectation.
- Reinforce. We can make use of following identity:

$$\nabla_\lambda \log q_\lambda(\theta) = \frac{\nabla_\lambda q_\lambda(\theta)}{q_\lambda(\theta)}$$

$$\nabla_\lambda q_\lambda(\theta) = q_\lambda(\theta)\nabla_\lambda \log q_\lambda(\theta)$$

Then we have

$$\begin{aligned} \nabla_\lambda E_{q_\lambda}[\log p(x|\theta)] &= \nabla_\lambda \int q_\lambda(\theta)p(x|\theta)d\theta && (18.48)\\ &= \int q_\lambda(\theta)\nabla_\lambda \log q_\lambda(\theta)p(x|\theta)d\theta && (18.49)\\ &= E_{q_\lambda}[p(x|\theta)\nabla_\lambda \log[q_\lambda(\theta)]] && (18.50) \end{aligned}$$

Then we succeed to move ∇ inside expectation.

Finally, a few links to learn more on VAEs.

- https://arxiv.org/pdf/1606.05908.pdf
- https://arxiv.org/abs/1312.6114

CS281: Advanced ML **November 13, 2017**

Lecture 19: Monte Carlo Basics

Lecturer: Sasha Rush *Scribes: Chang Liu, Jiafeng Chen, Alexander Lin*

19.1 History

We started with Monte Carlo in the past few lectures. The main method is to use draw samples from a proposal distribution and take sample average to approximate expectations.

$$\begin{aligned}\mathbb{E}_{y\sim p(y|x)}[f(y)] &= \int p(y|x)f(y)dy \\ &\approx \frac{1}{N}\sum_{n=1}^{N} f(\tilde{y}^{(n)})\end{aligned}$$

where $\tilde{y}^{(n)} \sim p(y|x)$. This approach requires the ability to sample $y \sim p(y|x)$.

19.2 Univariate Case

Let's start at the beginning: We know $F(x) = p(y \leq x)$ and it is univariate.
Assume we can compute $F^{-1}(u)$, then

$$p(F^u \leq x) = p(u \leq F(x)) = F(x)u$$

where F is the CDF of $u \sim \text{Unif}(0,1)$. However, this works only for univariate case. What this means is that, is we wish to sample $y \sim F$ for some known CDF F, we need only to sample $U \sim \text{Unif}$ and transform the sample of uniform random variables through F^{-1} to get a sample of y. Formally,

$$U \sim \text{Unif} \implies F^{-1}(U) \sim F$$

(This is known as the probability integral transform).

19.3 Normal Samples

We pursue the same strategy as in the univariate case. Given Uniform samples, we wish to apply some transform to obtain a sample of multivariate random variables of some distribution we are interested in. We execute this strategy for Normal random variables (this is known as the Box-Muller transformation). The upshot here is that to sample Normal random variables, we need only to sample Uniform random variables, which is much easier to do.

Box-Muller Sample $z_1, z_2 \sim \text{Unif}[-1,1]$. Discard the points outside of the unit circle, so our sample of $\{(z_1, z_2)\}$ is uniform on the unit disk. We would like to transform each (z_1, z_2) into some $(x_1, x_2) \sim \mathcal{N}(0, I)$. We want to find the right transform such that the Jacobian makes the following hold

$$\overbrace{p(x_1, x_2)}^{\text{Normal PDF}} = \overbrace{p(z_1, z_2)}^{\text{Unif disk PDF}} \left|\frac{\partial(z_1, z_2)}{\partial(x_1, x_2)}\right|.$$

We may check that

$$\begin{aligned}x_1 &= z_1\left(\frac{-2\log r^2}{r^2}\right)^{1/2} \\ x_2 &= z_2\left(\frac{-2\log r^2}{r^2}\right)^{1/2},\end{aligned}$$

where $r^2 = z_1^2 + z_2^2$, is the desired transform.

19.4 Rejection Sampling

Assume that we have access to the PDF $p(x)$ or the unnormalized PDF $\tilde{p}(x)$. The idea is to pick a **guide function** (valid PDF) $q(x)$ that is similar to p and easy to compute. We also pick a **scale** M. We require that

$$Mq(x) > p(x)$$

for all x and that we have access to $p(x)$ or $\widetilde{p}(x)$. The algorithm is as follows:

1. Sample $x_n \sim q(x)$
2. Draw $u \sim \text{Unif}[0, 1]$
3. If $u < \frac{p(x_n)}{Mq(x_n)}$, then keep x_n; otherwise, rerun from 1.

The interpretation is simple. The algorithm "graphs" $p(x)$ and the bounding $Mq(x)$ on a board, then proceeds to throw darts at the board and accepting those darts that hit below $p(x)$. The same algorithm works even if $\tilde{p}$ is unnormalized, since we have the degree of freedom to choose M and thus absorb the normalizing constant.

This method works with $\frac{\tilde{p}(x)}{Z} = p(x)$. But why should it work?
ANS This works because for whatever guide function we pick, we can write that guide function to be:

$$\widetilde{M} = ZM$$

19.4.1 Proof of Rejection Sampling

$$\begin{aligned}
p(x \le x_0 | x \text{is accepted}) &= \frac{\int_{-\infty}^{x_0} \int_0^1 q(x)\, 1(u \le \frac{\tilde{p}(x)}{Mq(x)}) du dx}{\int_{-\infty}^{\infty} \int_0^1 q(x)\, 1(u \le \frac{\tilde{p}(x)}{Mq(x)}) du dx} \\
&= \frac{\frac{1}{M}\int_{-\infty}^{x_0} \tilde{p}(x) dx}{\frac{1}{M}\int_{-\infty}^{\infty} \tilde{p}(x) dx} \qquad \text{the denominator is probability the acceptance} \\
&= \int_{-\infty}^{x_0} p(x) dx \\
&= p(x \le x_0)
\end{aligned}$$

19.5 Examples for Rejection Sampling

1. In Bayesian statistics, we often encounter the following problem. We are given $p(\theta)$, $p(x|\theta)$, and we wish to sample from the posterior $p(\theta|x)$. We can compute the unnormalized posterior $\tilde{p}(\theta|x) = p(x|\theta)p(\theta)$. Set $q(\theta) = p(\theta)$. Choose $M = p(x|\hat{\theta})$, where $\hat{\theta}$ is the maximum likelihood estimator. Then $Mq \ge p$. Thus, rejection sampling says the followng. Sample from the prior $q(\theta) = p(\theta)$, roll a uniform u, and keep those $\theta \sim p(\theta)$ with

$$u \le \frac{p}{Mq} = \frac{p(x|\theta)}{p(x|\hat{\theta})} \le 1.$$

2. Let $p \sim \mathcal{N}(0, \sigma_p^2 I)$ and $q \sim \mathcal{N}(0, \sigma_q^2 I)$ where $\sigma_q^2 > \sigma_p^2$. Pick

$$M = (\sigma_q / \sigma_p)^D,$$

where D is the dimension of the Multivariate Normal. Note that M becomes very large when D becomes large, and so rejection sampling may be inefficient when D is large. If we imagine M as the metric of which to boost the Gaussians to make random sampling work, due to the known geometry of Gaussian distributions we can imagine as D increase there will be more and more "space" between p and q to fill, thus making Random Sampling quite difficult.

19.6 Importance Sampling

We want to approximate the expectation

$$\mathbb{E}_{x\sim p}(f(x)) = \int f(x)p(x)\,dx.$$

So far, we can sample a bunch of points—via, say, rejection sampling—from p and calculate a sample mean to approximate the true expectation. If the structure of p and the structure of f are very different, so Monte Carlo methods so far might be inefficient, since it samples from high density areas in p, which may have very low values of f, and the Monte Carlo may miss areas with high values of f but low probability of happening.

Consider the integral

$$\int q(x)\frac{p(x)}{q(x)}f(x)\,dx = \mathbb{E}_q\left(f(x)\frac{p(x)}{q(x)}\right) = \mathbb{E}_p(f(x)).$$

We may now apply the same Monte Carlo trick to sample from q and take the sample mean of $f(x)p(x)/q(x)$.

What is the benefit of using q? Since we can choose q to be closer to f, then more of the sample we choose would be around high values of f. Here we don't need to wait for some low-probability tail event in p to happen in order to get reliable estimates of $E_p(f(x))$. Instead, we can directly look at the tail events via q and weight the data appropriately using p/q to still maintain asymptotic convergence.
How exactly do we choose q? We want to minimize the variance of $f(x)p(x)/q(x)$ when $x \sim q$, since this allows for faster convergence. Then

$$\text{Var}\left(\frac{f(x)p(x)}{q(x)}\right) = \mathbb{E}\left[\left(\frac{f(x)p(x)}{q(x)}\right)^2\right] - \underbrace{\left(\mathbb{E}\left(\frac{f(x)p(x)}{q(x)}\right)\right)^2}_{\text{constant eventually}}$$

$$\begin{aligned}\mathbb{E}\left(\frac{f(x)p(x)}{q(x)}\right)^2 &\geq \left(\mathbb{E}_q\left(\frac{p(x)|f(x)|}{q(x)}\right)\right)^2 && \text{(Jensen's inequality)}\\ &= \left(\int p(x)|f(x)|\,dx\right)^2\end{aligned}$$

We minimize the lower bound via Jensen's inequality (similar to what we did in variational inference). The optimal q is chosen via

$$q^* = \frac{|f(x)|p(x)}{\int |f(x)|p(x)\,dx}.$$

It may be difficult to normalize q^* in practice, however.

$$\begin{aligned}\mathbb{E}_p[f(x)] &= \int p(x)f(x)dx\\ &\approx \frac{1}{N}\sum_{n=1}^{N} f(\tilde{x}^{(n)}\end{aligned}$$

where $\tilde{x}^{(n)} \sim p(x)$
Exercise: Approximate $\int_0^\pi x\sin(x)dx$ using importance sampling.

Solution: See importancesampling.ipynb in the same directory.

CS281: Advanced ML **November 15, 2017**

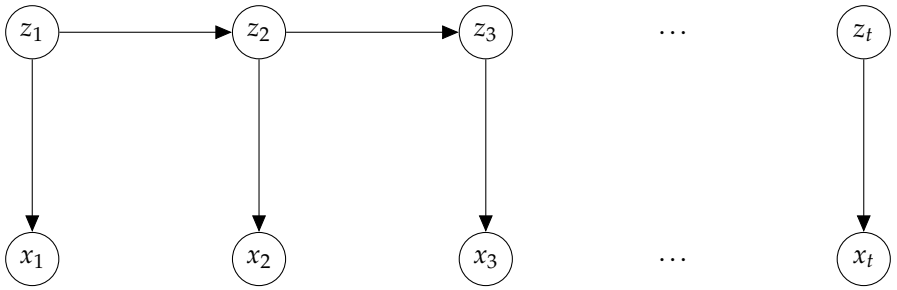
Lecture 20: Importance Sampling and Particle Filtering

Lecturer: Sasha Rush Scribes: Shangyan Li, Alex Wu, Nicolò Foppiani, Kevin Liu, Daniel Merchan, Milan Ravenell

20.1 State space models and Kalman Filter

20.1.1 Introduction on state space models

The first topic which has been covered during this lecture is a review about state space models and Kalman Filter. A state space model (SSM) is a Hidden Markov Model (HMM) with continuous hidden states.



The model is specified by

$$z_t = g_t(z_{t-1}, \epsilon_t)$$
$$x_t = h_t(z_t, \delta_t)$$

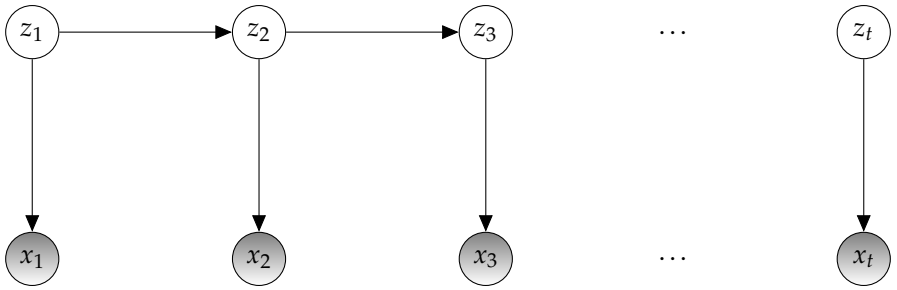
where ϵ and δ represent a noise which is added in the transition between two states. z_t is usually referred as the transition model and x_t as the observation model.
An important case is where the transition functions are linear-Gaussian, which means:

$$z_t = A_t(z_{t-1} + \epsilon_t) \text{ where } \epsilon_t \sim \mathcal{N}(0, Q_t)$$
$$x_t = C_t z_t + \delta_t \text{ where } \delta_t \sim \mathcal{N}(0, R_t)$$

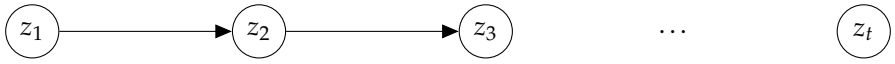
This case is called linear-Gaussian SSM (LG-SSM) or linear dynamical system (LDS).
In most situations we can assume that the process is stationary and considering the model functions independent of time label.
In principle this problem can be solved with the architecture developed for graphical models: this is a directed graphical model in which each variables is Gaussian distributed. Thus the whole model can be solved raising to a multivariate Gaussian for the whole model.
However this model is historically relevant and thus deserves a proper study.

20.1.2 Kalman Filter

In fact, if we know that $p(z_1)$ is MVN, then $p(z_t|x_{1:t})$ will be MVN. Suppose now to have observed $x_{1:t}$ and we want to compute $p(z_t|x_{1:t})$.



We can marginalize, obtaining a tree, on which we can apply exact belief propagation (BP).



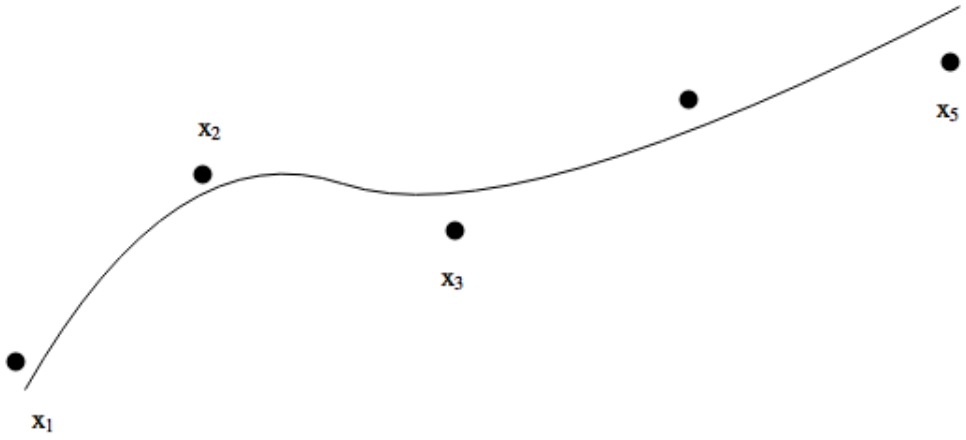
There is a difference with exact belief propagation in the discrete case:

- discrete: Bel(z_t) is an array
- MVN: Bel(z_t) is MVN with $\vec{\mu}$ and Σ

The inference algorithm based on MVN BP in this case is called **Kalman filter**. This is a crucial and popular algorithm, central to multiple technologies. However, at its core, it just runs the LG formula multiple times as part of BP.

20.1.3 Example: LG-SSM for tracking application

We have an object moving in the 2-D plane: the hidden state represents the position and velocity at every timestep, and the observed state is the observed position at the every timestep.



$$\vec{z_t} = [z_{1,t},\ z_{2,t},\ \dot{z}_{1,t},\ \dot{z}_{2,t}]$$

A_t updates $z_t \to z_{t+1}$
C_t generates x_t given z_t: in this case it sets $x_{1,t} = z_{1,t}$ and $x_{2,t} = z_{2,t}$
ϵ_t represents the noise related to physics
δ_t represents the noise generated by the sensors in the observations

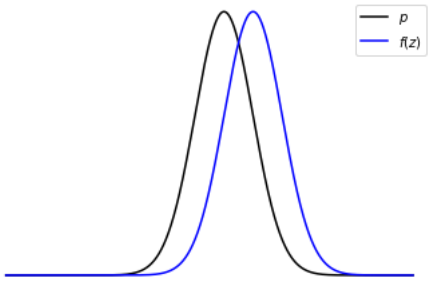
20.1.4 Kalman smoothing

Suppose we observed $x_{1:T}$ and we want to infer z_t. We can use another variant of BP in which we combine the messages from the past and from the future: this is called **Kalman smoothing**.

20.2 Review on importance sampling

20.2.1 Basic Idea

We aim to sample form p to approximate $f(z)$ using MonteCarlo sampling:



To approximate integrals of the form:

$$\mathbb{E}_p[f(z)] = \int q(z)\frac{p(z)}{q(z)}f(z)dz \approx \frac{1}{S}\sum_{i=1}^{S}\frac{p(z^s)}{q(z^s)}f(z^s) = \sum_{i=1}^{S} w^s f(z^s)$$

$w^s = \frac{p(z^s)}{q(z^s)}$ are importance weights associated with each sample.

20.2.2 Unnormalized Distributions

Frequently will have unnormalized target distribution $\tilde{p}(z)$ but not its normalization constant Z_p. Similarly, may have unnormalized proposal $\tilde{q}(z)$ and its unknown normalization constant Z_q.

$$p(z) = \frac{\tilde{p}(z)}{Z_p} \text{ and } q(z) = \frac{\tilde{q}(z)}{Z_q}$$

Substituting into the desired expectation:

$$\mathbb{E}_p[f(z)] = \frac{Z_q}{Z_p}\int q(z)\frac{\tilde{p}(z)}{\tilde{q}(z)}f(z)dz \approx \frac{Z_q}{Z_p}\frac{1}{S}\sum_{i=1}^{S}\tilde{w}^s f(z^s)$$

where $\tilde{w}_s = \frac{\tilde{p}(z^s)}{\tilde{q}(z^s)}$.

Now, how to compute $\frac{Z_q}{Z_p}$? Can also importance sample from q.

$$\frac{Z_p}{Z_q} = \frac{1}{Z_q}\int \tilde{p}(z)dz = \frac{1}{Z_q}\int q(z)\frac{\tilde{p}(z)}{q(z)}dz = \int q(z)\frac{\tilde{p}(z)}{\tilde{q}(z)}dz \approx \frac{1}{S}\sum_{i=1}^{S}\tilde{w}^s$$

Substituting the sampled $\frac{Z_p}{Z_q}$ back:

$$\mathbb{E}_p[f(z)] \approx \sum_{s=1}^{S}\frac{\tilde{w}^s}{\sum_{s'}\tilde{w}^{s'}}f(z^s) = \sum_{s=1}^{S} w^s f(z^s)$$

Note: This form of importance sampling is convenient as it lets us work with unnormalized distributions. However, in many cases is much less convenient than, for instance, rejection sampling or MonterCarlo sampling because it is not fully clear how to use the weights.
Now consider getting samples from p. We need to define $p(x)$ in terms of $\mathbb{E}_p[f(x)]$ using a δ function that returns 1 if it's the value we want, 0 otherwise:

$$p(x = x') = \mathbb{E}_p[\delta_{x'}(x)]$$

Then, our estimate in terms of importance sampling is:

$$p(x) = \sum_{s=1}^{S} w_s \delta_{x^s}(x)$$

But we actually want unweighted samples, which can be obtained through **re-sampling**: pick x^s with probability w^s. In other words, we create a discrete set that approximates the original distribution and then we draw samples from that discrete set based on a categorical distribution weighted by the w_s.

20.2.3 Example

Recall from last class, we have:

$$p(\theta|Data) = \frac{p(Data|\theta)p(\theta)}{p(Data)}$$
$$\tilde{p}(\theta|Data) = p(Data|\theta)p(\theta)$$

Notice we drop the normalization term for $\tilde{p}$. Then $q(\theta) = p(\theta)$, which implies that we importance-sample based on the prior. This gives normalized w_s:

$$w_s = \frac{\tilde{p}(\theta_s)/q(\theta_s)}{\sum_{s'} \tilde{p}(\theta_s)/q(\theta_s)} = \frac{p(Data|\theta_s)}{\sum_{s'} p(Data|\theta_{s'})}$$

In summary, we run importance-sampling to compute the posterior of the parameters by sampling different parameters from the priors and then by assigning weights to each of those different parameters based on likelihood in the data. The resulting samples are unbisased samples from the distribution of interest.

20.3 Particle Filtering/Sequential Monte Carlo

There might be cases of space-state models in which it might not be possible to derive closed-form expressions based on LG for the update equations. In those cases, a sampling-based approaches offer an alternative method.
In order to approximate the belief state of an entire sequence, we can use a weighted set of particles as follows:

$$p(z_{1:t} = z'_{1:t}|x_{1:t}) = \int p(z_1, ..., z_t|x_{1:t})\delta_{z'}(z)dz \approx \sum_{s=1}^{S} \hat{w}_t^s \delta_{z'}(z_{1:t}^s)$$

where $\hat{w}_t^s$ represents the normalized weight of sample s at time t.

We update the belief state using importance sampling, with the following importance weights.

$$w_t^s \sim \frac{p(z_{1:t}|x_{1:t})}{q(z_{1:t}|x_{1:t})}$$

The numerator can be expressed as follows:

$$p(z_{1:t}|x_{1:t}) \propto p(z_{1:t}|x_{t-1})p(x|t|z_{1:t}, y_{t-1}) \propto p(x_t|z_t)p(z_t|z_{t-1})p(z_{1:t-1}|x_{1:t})$$

where $p(x_t|z_t)$, $p(z_t|z_{t-1})$, and $p(z_{1:t-1}|x_{1:t})$ correspond to the observation, transition, and recursion of the sequence, respectively.

Meanwhile, we can express the denominator as

$$q(z_{1:t}|x_{1:t}) \propto q(z_t|z_{1:t-1}, y_{1:t})q(z_{1:t-1}|x_{1:t-1})$$

Our importance weights, therefore, simplify to

$$w_t^s \propto \frac{p(x_t|z_t)p(z_t|z_{t-1})p(z_{1:t-1}|x_{1:t})}{q(z_t|z_{1:t-1}, y_{1:t})q(z_{1:t-1}|y_{1:t-1})} = \frac{p(x_t|z_t)p(z_t|z_{t-1})}{q(z_t|z_{1:t-1}, y_{1:t})} w_{t-1}^s$$

Using this expression, we have the following algorithm to approximate the belief state.
1. Start with particles $(w_t, z_t)^{(s)}$.
2. At each time step t for particle x, calculate $w_t^s = \frac{p(x_t|z_t)p(z_t|z_{t-1})}{q(z_t|z_{1:t-1}, y_{1:t})} w_{t-1}^s$ and sample $z_t^s \sim q(z_t|z_{1:t-1}^s, x_t)$.
3. Compute $p(z_t, x_{1:t})$

CS281: Advanced ML **November 20, 2017**

Lecture 21: Deep Learning in Health Care

Lecturer: Sasha Rush *Scribes: Raghu Dhara, Rui Fang*

We had a guest lecturer today: Sumit Chopra, head of AI Research at Imagen Technologies, a stealth-phase medical imaging startup.

21.1 Past Work

Before his work at Imagen, Dr. Chopra was a research scientist at Facebook AI Research. There, Dr. Chopra worked primarily in natural language processing (with Dr. Rush!). He completed his doctoral thesis under Yann LeCun at NYU. Here, he worked on Siamese Networks for picture distance metrics, DrLIM (dimensionality reduction by learning an invariant mapping) , and factor graphs for relational regression, a topic he then later applied at a startup to predict residential real estate prices. He has also worked at AT&T labs as a research scientist.

21.2 Computer Vision

21.2.1 Traditional Methods

Traditional methods in computer vision perform image classification by transforming an input image into features through a hand-crafted feature extractor (SIFT, HOG, ...) and then classifying the features using a classifier (SVMs, Neural Network (rarely), ...). The drawback of such methods is that hand crating features become challenging when input is not visually perceptible: depth map, etc.

21.2.2 Deep Learning: Convolutional Neural Networks (CNNs)

Unlike traditional methods, deep learning proposes to learn the features from scratch - learn everything end to end. The intuition behind deep learning is to learn a highly complex function composed of lots of simple functions, of which the parameters will be learned. The main class of deep learning architectures used in computer vision is the convolutional neural networks.
There are three basic types of hidden layers in a CNN: convolutional layer, non-linearity layer, and pooling layer.

- The convolutional layer is built upon convolution operation, which applies a kernel matrix to an input matrix and returns the dot products between the kernel and each receptive field. For example, Figure 21.20 shows applying a 3×3 kernal with stride $= 1$ (move one pixel at a time) over a 7×7 input matrix results in a 5×5 output matrix.
- The non-linearity layer applies a non-linear activation function to the output of the convolutional layer.
- The pooling layer works as subsampling: it applies a kernel (usually a max function) to an input matrix without overlapping, as shown in Figure 21.21. The pooling layer reduces spatial dimension of the input and results in spatial invariance.

A typical CNN architecture is shown in Figure 21.22. This CNN performs an image level classification - it takes the entire image and output predictions of the major theme of this image. The input image is convoluted and subsampled a few times before getting classified through the fully connected layers.

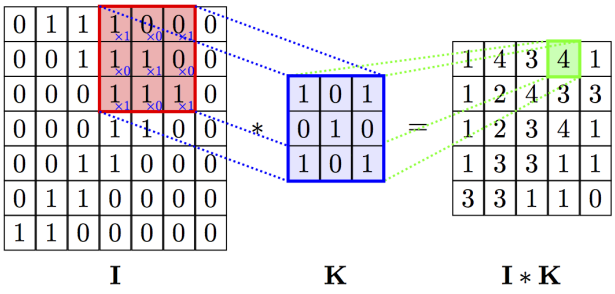


Figure 21.20: Convolution

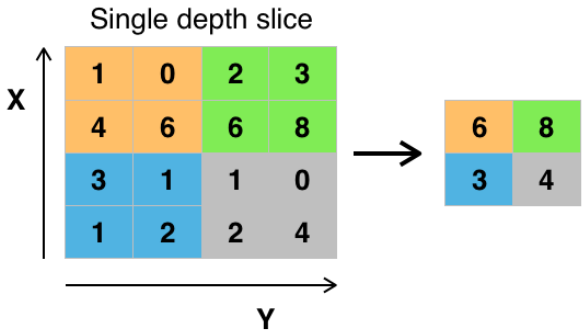


Figure 21.21: Pooling

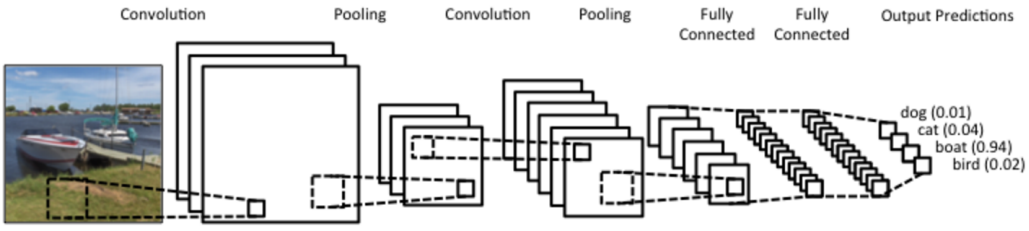


Figure 21.22: Typical CNN architecture

21.2.3 Pixel Level Classification CNNs

For image recognition, the tasks range from image level classification to object level classification to pixel level classification. The focus here is pixel level classification: the convolutional neural networks have a loss function associated with every pixel. Applications of pixel level classification include scene understanding, semantic segmentation, depth map prediction, medical imaging, etc.
We introduce three CNN based architectures in this category:

1. Fully Convolutional Network (FCN) [3]

   Fully convolutional networks take input of arbitrary size and produce correspondingly sized output (see Figure 21.23). Different from typical classification CNN architectures where outputs are non-spatial, in FCN the classification layers (fully-connected layers) are viewed as convolutions with kernels that cover their entire input regions, therefore generating 2D classification maps as outputs (see Figure 21.24). While the output is 2D, it is still coarse due to subsampling. Hence, upsampling is needed to generate outputs of the same size as inputs. In addition, the networks are designed to combine predictions from both the final layer and the intermediate layer to provide finer details (see Figure 21.25 and Figure 21.26).

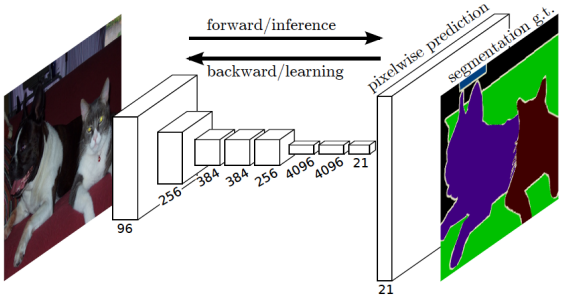


Figure 21.23: Fully convolutional networks can efficiently learn to make dense predictions for per-pixel tasks like semantic segmentation.

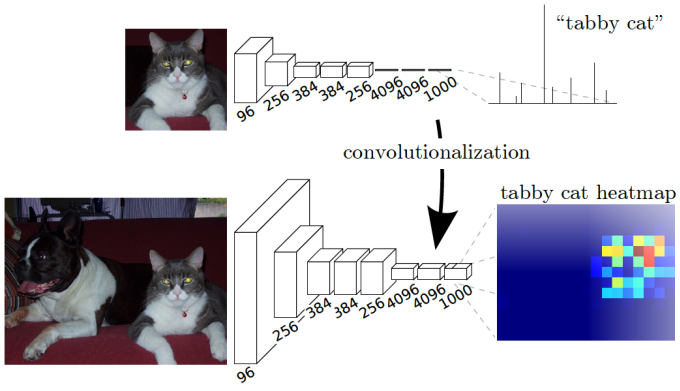


Figure 21.24: Transforming fully connected layers into convolution layers enables a classification net to output a heatmap.

2. U-Net [4]

[3] Long et al., *Fully Convolutional Networks for Semantic Segmentation*
[4] Ronneberger et al., *U-Net: Convolutional Networks for Biomedical Image Segmentation*

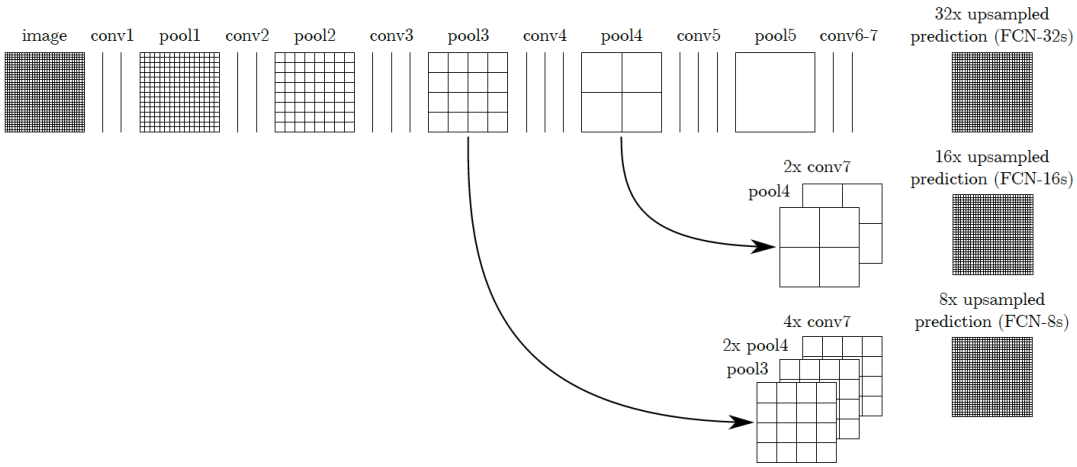


Figure 21.25: A fully convolutional net (FCN) for segmentation that combines layers of the feature hierarchy and refines the spatial precision of the output.

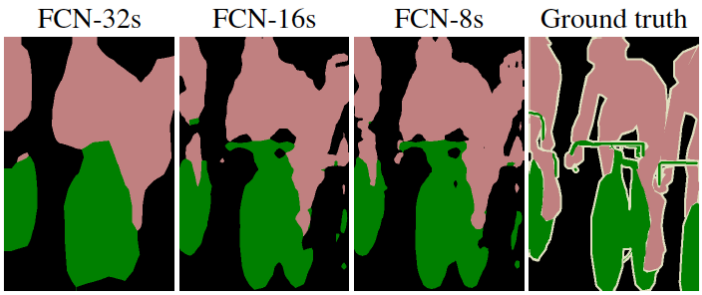


Figure 21.26: Refining fully convolutional nets by fusing information from layers with different strides improves segmentation detail.

Since annotated examples are not as easily obtainable for biomedical tasks due to the significant and specialized effort required to procure them, U-Nets tries to use the provided samples efficiently. The architecture consists of a contracting path to capture context and a symmetric expanding path that enables precise localization (see Figure 21.27). The contracting path involves repeatedly applying the following sequence: a 3x3 convolution, a ReLU, and a 2x2 max pool. The expansive step involves repeatedly applying the following sequence: upsampling the feature map followed by a 2x2 convolution (""up-convolution"), a concatenation of the corresponding feature map from the contracting path, another 3x3 convolution, and a ReLU. At the final layer a 1x1 convolution is used to map each 64- component feature vector to the desired number of classes, for a total of 23 convolutional layers.

For training, the energy function is computed by a pixel-wise soft-max over the final feature map combined with the cross entropy loss function that penalizes deviation from the ground truth pixel labels (see Figure 21.28). These U-Nets train and run faster than the previous state-of-the art deep convolutional networks that came before it (10 hours to train, under a second to run), making them a compelling option. They are invariant to elastic deformations to the input image, a strategy that was leveraged to augment the data.

3. Atrous (Dilated) Convolution [5]

   Atrous convolution is convolution with upsampled filters. This allows us to explicitly control the resolution at which feature responses are computed within deep CNNs. Such an ability gives us a

[5] Chen et al., *DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs*

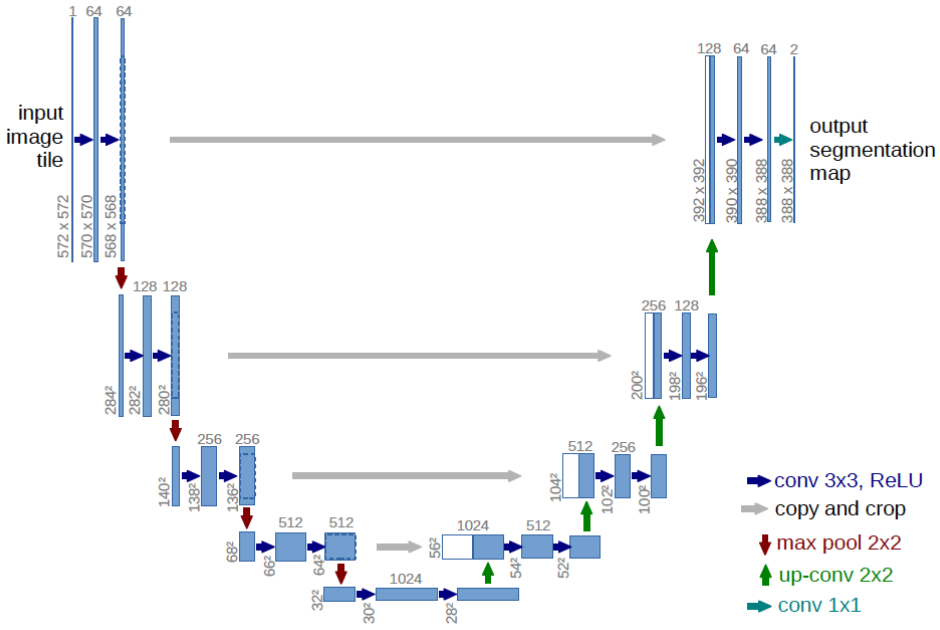


Figure 21.27: U-net architecture. Notice the namesake U shape.

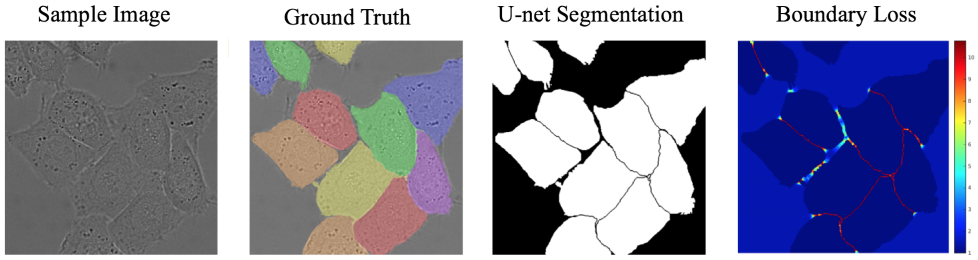


Figure 21.28: U-Net producing a pixel-wise loss map to force the network to learn border pixels

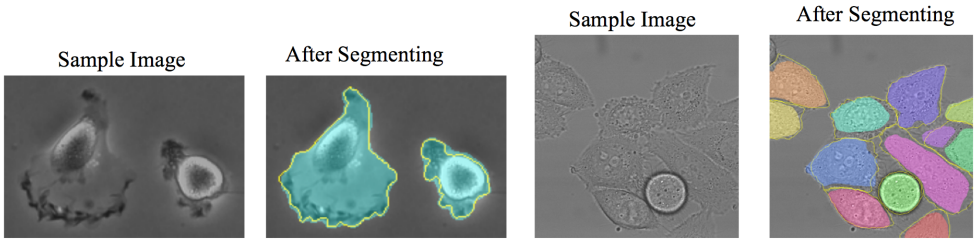


Figure 21.29: Example segmentation results with U-Net. The yellow border is the ground truth.

clear advantage: the output image will be closer in size to the input image, enabling sharper decision boundaries at useful scales. Traditional methods often involve downsampling steps that produce outputs significantly smaller than the inputs, and the corrective upsampling introduces pixel declocalization. To mitigate this, atrous convolutional networks combine the results at the final layer of a deep convolutional neural network with a fully connected Conditional Random Field (CRF), which is shown both qualitatively and quantitatively to improve localization performance.

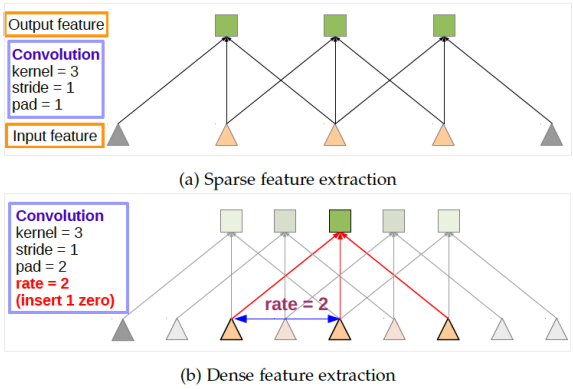


Figure 21.30: Atrous convolution in 1D

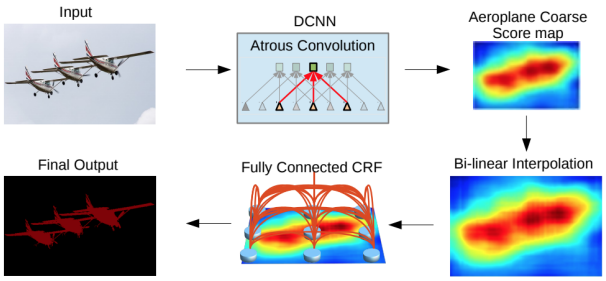


Figure 21.31: Overall procedure for atrous convolution

21.3 Healthcare Data Landscape and Diagnostic Radiology

21.3.1 Providers and Payers

In general, medical data is difficult to acquire. There are two primary repositories: healthcare providers and payers. Providers such as hospitals are often secretive about their datasets, which are often incomplete. Payers such as insurance companies have more thorough datasets so long as the patient does not change insurers. Medicare data is somewhat perpetual as well but is biased towards older people.

21.3.2 Why Diagnostic Radiology is Important

1. Prevalence of studies: 600 million per year in the US, over 5 billion per year in the world
2. Shortage of skilled radiologists: 11,000 fewer than needed in the US, some countries only have a couple in total
3. Prevalence of errors: 5-15% error rate in the US, which has a very sophisticated medical system - this translates to 30-90 million misdiagnoses per year, a number of which are potentially fatal

21.3.3 AI in Diagnostic Radiology

Given an image along with possibly other modalities of data, the principal goals are to

1. Identify the region(s) of interest/anomaly
2. Infer whether the image has a clinically relevant medical condition
3. Infer the underlying cause of the condition (causal inference)

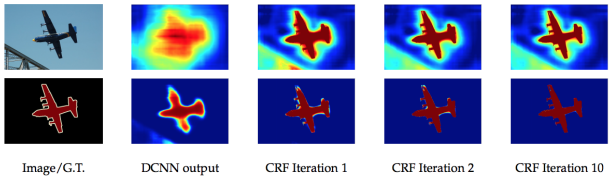


Figure 21.32: Repeated mean-field iterations further refine the score maps (top row) and belief maps (bottom row)

4. Write the full diagnostic report