UBC CPSC 540: Machine Learning

UBC CPSC 540: Machine Learning

Linear Prediction

Linear regression in 1D - find Θ_1, Θ_2 such that

$$\hat{y_i}(x_i) = \Theta_1 + x_i\Theta_2$$

(where x_i is ith data point and $\hat{y_i}$ is the estimate for that point) minimises some loss function, say J:

$$J(\Theta) = \sum_{i=1}^{n}(y_i - \hat{y_i})^2$$

In general, the linear model can be expressed as follows:

$$\hat{y_i} = \sum_{j=1}^{d} x_{ij}\Theta_d$$

where d is the numer of dimensions/features and i represents ith sample. Note, that $x_{i1} = 1$ (i.e. all data points will have the value of 1 as the first component to account for the bias term Θ_1), in the matrix notation (see below) the first column of the matrix will have value of 1 everywhere.

I matrix notation:

$$\hat{\mathbf{y}} = \mathbf{X\Theta}$$

$\hat{y} \in R^{n\times 1}, X \in R^{n\times d}, \Theta \in R^{d\times 1}$
(Note data points are represented as rows)

Optimisation Approach

Our error/cost/energy function is a sqaure of difference between the truth and the prediction:

$$J(\Theta) = \sum_{i=1}^{n}(y_i - x_i^T\Theta)^2 = (y - X\Theta)^T(y - X\Theta)$$

To find optimal Θ we need to find $\frac{\partial J(\Theta)}{\partial \Theta} = 0$

Matrix Differentiation Properties

- $\frac{\partial A\Theta}{\partial \Theta} = A^T$
- $\frac{\partial \Theta^T A\Theta}{\partial \Theta} = 2A^T\Theta$

Least Squares

Finding $\frac{\partial J(\Theta)}{\partial \Theta} = 0$ yields:

$$\hat{\Theta} = (X^T X)^{-1} X^T y$$

Maximum Likelihood and Linear Regression

Univariate Gaussian Distribution

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x-\mu)^2}$$

where μ is mean and σ^2 is variance (distance from the mean to the inflection point)

Multivariae Guassian Distribution

Let $x \in R^{n \times 1}$, then pdf of an n-dimensional Gaussian is given by:

$$p(x) = (2\pi\Sigma)^{-1/2} e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)}$$

Where Σ is a covariance matrix (multidimensional equivalent of σ^2):

$$\Sigma = \begin{pmatrix} \sigma_{11} & \cdots & \sigma_{1n} \\ \vdots & & \vdots \\ \sigma_{n1} & \cdots & \sigma_{nn} \end{pmatrix}$$

Sampling from a Multivariate Gaussian Distribution

If you want to sample x from a univariate Gasussian, i.e. $x \sim N(\mu, \sigma^2)$ you can sample from $N(0,1)$ and scale it accrodingly, i.e. $x \sim \mu + \sigma N(0,1)$

In multivariate case you need to be able to find a "square root" of a covariance matrix using Cholesky decomposition, i.e.

$$B = \Sigma\Sigma^T$$

and then sample:

$$x \sim \mu + BN(0, I)$$

where I is an identity matrix

The Likelihood for Linear Regression

The Linear Regression can be solved for Θ also using Maximum Likelihood Estimation, i.e. trying to answer a question - what's the most likely Θ that produced our data? If we assume our data is idependent and Gaussian we can describe the likelihood of the data given some model Θ as

$$p(x_1, \ldots, x_n | \Theta) = p(x_1|\Theta) * p(x_2|\Theta) * \ldots * p(x_n|\Theta)$$

so finding Θ that maximizes likelihood means finding such a Gaussian distribution that this product is maximized.

In the Regression setting we assume that each data label y_i is Gaussian distributed with mean $x_i^T\Theta$ and variance σ^2. Thus we have:

$$p(y|X,\Theta,\sigma^2) = \prod_{i=1}^{n} p(y_i|x_i,\Theta,\sigma^2)$$

Intuitively we find such a line $x_i^T\Theta$ that maximises the joint likelihood of the data points by assuming that they are Gaussian distributed around the line (which defines a mean of the distribution for each x_i)

Maximising that probability is equivalent to minimising the (least squares) cost function

Maximum Likelihood

The maximum likelihood esitmate (MLE) of Θ is obtained by taking a derivative of the log-likelihood $l(\Theta) = log\ p(y|X,\Theta,\sigma^2)$ w.r.t Θ (we can do that because log is monotonic so does not change the position of the local extremum) and this trick allows us to get rid of the exponents in the expression (multivariate Gaussian). Finally, our ML estimate of Θ is:

$$\hat{\Theta_{ML}} = (X^TX)^{-1}X^Ty$$

We can also find an ML estimate of σ (we could not do this with least squares approach) which quantifies our uncertainty about the model. If we find $\frac{\partial l(\Theta)}{\partial \sigma} = 0$ we'll get:

$$\sigma^2 = \frac{1}{n}(y - X\Theta)^T(y - X\Theta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})^2$$

Frequentist Learning and Maximum Likelihood

Frequentist learning assumes there exists a true model Θ_0 which we're trying to find. Since we have limited data we find an estimation $\hat{\Theta}$

Bernoulli Distribution

Bernoulli random variable x takes values in $\{0,1\}$ such that $p(x=1) = \theta$ and $p(x=0) = 1-\theta$ (e.g. coin flip).

Entropy

In information theory, entropy H is a measure of uncertainty associated with a random variable defined as:

$$H(x) = -\sum_{x} p(x|\theta)\ log\ p(x|\theta)$$

Regularization and Regression

MLE - Properties

- One important property is that MLE, for i.i.d data from $p(x|\Theta_0)$ minimizes the KL divergence between the distributions Θ and Θ_0 (true distribution)
- MLE is asymptotically normal, i.e. as number of data points increases to infinity we have: $\hat{\Theta} - \Theta_0 \rightarrow N(0, I^{-1})$ where I^{-1} is the Fisher Information matrix
- It is asymptotically efficient, i.e. has the lowest variance

Bias and Variance

- Bias is the difference between what **the mean** $\bar{\Theta}$ of your multiple estimates (or, more formally, expected value over all possible estimates) $\hat{\Theta}$ and the true model Θ_0
- Variance is an expected value of the (squared) difference between the estimated $\hat{\Theta}$ and the mean estimate $\bar{\Theta}$, i.e. how much our different estimates can differ given different data

Regularization (Ridge)

Problem with Least Squares was that it requires you to invert a matrix, which might be problematic if a problem is poorly conditoned (see *Matrix condition number*, e.g. ratio of largest and the smallest eigenvalue). To solve this we can add a small element to the diagonal; δ (squared to make it positive):

$$\hat{\Theta} = (X^T X + \delta^2 I_d)^{-1} X^T y$$

This is called a Ridge regression estimate and is the solution to the following cost function:

$$J(\Theta) = (y - X\Theta)^T (y - X\Theta) + \delta^2 \Theta^T \Theta$$

(to prove it, calculate $\frac{\partial J(\Theta)}{\partial \Theta} = 0$)
If $\delta = 0$ we get the least sqaures solution $\hat{\Theta_{ML}}$, if $\delta \rightarrow \infty$ the solution is $\hat{\Theta_R} = \vec{0}$. For other deltas we move along the curve between those two solutions. Since it's not necessarily a line, different thetas get closer to zero quicker than others - thus *regulatization*.

Reparametrisation Trick

Note that in Ridge you need to invert $X^T X$ which is $d \times d$. However, if $d >> n$ you might use a reparametrisation trick. The solution to Ridge regression can be written as:

$$\Theta = X^T \alpha$$

where $\alpha = \delta^{-2}(y - X\theta)$, which can also be written as:

$$\alpha = (XX^T + \delta^2 I_n)^{-1} y$$

which requires inverting XX^T which is $n \times n$.

Ridge Regression as constrained optimization

You can visualise the optimal curve that minimises the Ridge cost by noting that:

- $\Theta^T \Theta$ describes circles centered at the origin (in 2D: $\Theta_1^2 + \Theta_2^2$)
- $(y - X\Theta)^T (y - X\Theta)$ represents ellipses cenetered at least squares solution $\hat{\Theta_{ML}}$

- The solution (curve) for the Ridge cost results from connecting the points at which the ellipses and circles are tangent.

Going Nonlinear via Basis Functions

You can still use the linear model to model nonlinearities by using basis functions, e.g. say the data looks parabolic, we can use $\phi(x)$ instead of x:

$$y(x) = \phi(x)\Theta + \epsilon$$

where (for example) $\phi(x) = [1, x, x^2]$ (Note this is a 1D problem and we're using 3 thetas here, so this approach scales badly with higher dimensions as we need to account for cross terms, e.g. $x_1 x_2$)

Kernel Regression and RBFs

We can use kernels or radial basis functions (RBFs) as features (instead of e.g. $[1, x, x^2]$):

$$\phi(x) = [\kappa(x, \mu_1, \lambda), \ldots, \kappa(x, \mu_d, \lambda)]$$

e.g. $\kappa(x, \mu_i, \lambda) = e^{-\frac{1}{\lambda}||x-\mu_i||^2}$. Ituitively this is putting multiple basis (Gaussian) functions (centered around their means with λ describing their width) and finding such thetas that, when all terms (scaled Gaussians) are added, we get reasonable regression line.

Note that in this setting we need to choose hyperparameters μ, λ. Very often people put bases at the data points, so with n data points we'd have n RBFs centered at each data point.

Regularization, Cross-validation and Data Size

Cross Validation

When evaluating a hyperaparameter you split the dataset into training and test dataset. Then evaluate the model using those datasets to get e_{train} and e_{test}. Then you can use the following approaches:

- MinMax - Choose hyperparameter which has the minimum $max(e_{train}, e_{test})$
- Average - Choose hyperaparameter which gives the best $avg(e_{train}, e_{test})$

K-fold Cross Validation

K-fold requires splitting the dataset into K folds, trainining on all but kth fold and testing on kth fold in a round-robin fashion.

For $K = N$ we call it Leave-one out Cross-Validation (LOOCV)

Bayesian Learning

Bayesian Learning is an alternative to Maximum Likelihood - it's a problem of integration as opposed to the problem of optimisation.

It gives us an automatic way to quantify uncertainty of data (unlike the previous assumption of all data points having the same variance)

Bayes Rule

Bayes rule enables us to reverse probabilities:

$$P(h|d) = \frac{P(d|h)P(h)}{P(d)} = \frac{P(d|h)P(h)}{\sum_{h \in H} P(d|h)P(h)}$$

where: h means a *hypothesis*, d means *data*, $P(h)$ is a prior, $P(d|h)$ is a likelihood (data given model/hypothesis) and $P(h|d)$ is a posterior. Note that the denominator normalizes the whole expression, i.e. includes all possibilities of the nominator.

Note few identities in continuous case:

$$\int P(A)dA = \int P(A|B)dA = 1$$

$$P(A) = \int P(AB)dB$$

Bayesian Linear Regression

The likelihood is Gaussian $N(y|X\theta, \sigma^2 I_n)$. The conjugate prior is also a Gaussian, which we will denote by $p(\theta) = N(\theta|\theta_0, V_0)$.

Conjugate analysis allows us to achieve the fact that prior and posterior are of the same shape, so if we select prior to belong to a certain family (e.g. Gaussian) we're guaranteed that posterior will also be Gaussian.

The idea is to derive an equation for the posterior distribution by manipulating the product of likelihood (Gaussian) and the prior (Gaussian) until we get something (the expression in a particular form), for which we know the integral from the multivariate Gaussian, i.e.

$$\int e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)} dx = |2\pi\Sigma|^{\frac{1}{2}}$$

So the equation for the Posterior is:

$$p(\theta|X, y, \sigma^2) \propto N(y|X\theta, \sigma^2 I_n)\ N(\theta|\theta_0, V_0) = N(\theta|\theta_n, V_n)$$

Where:

$$\theta_n = V_n V_0^{-1} \theta_0 - \frac{1}{\sigma^2} V_n X^T y$$

$$V_n^{-1} = V_0^{-1} - \frac{1}{\sigma^2} X^T X$$

Special cases:

- For $\theta_0 = 0$ and $V_0 = \tau_0^2 I_d$ where $\lambda := \frac{\sigma^2}{\tau_0^2}$ we get Ridge regression
- For a uniform prior, i.e. when $\lambda = 0$ we get Maximum Likelihood Estimation

Bayesian vs ML Prediction

Previously, in order to do prediction, we plugged in new $x_\star$ to the θ we found during learning. Bayesian approach, however, marginalizes over the posterior, i.e. it does not use one θ but integrates (weights) over all thetas:

$$P(y|x_\star, D, \sigma^2) = \int N(y|x_\star^T\theta, \sigma^2)N(\theta|\theta_n, V_n)d\theta = N(y|x_\star^T\theta_n, \sigma^2 + x_\star^T V_n x_\star)$$

This is in contrast to previous, frequentist predictor:

$$P(y|x_\star, D, \sigma^2) = N(y|x_\star^T\theta_{ML}, \sigma^2)$$

So they differ in the representation of variance, which is constant in frequentist solution but variable in Bayesian. Where there is data, the uncertainty is small. Where there is less data - the uncertainty is big.

Gaussian Processes

Covariance between x_1 and x_2 describes the dependence between x_1 and x_2 i.e. if we increase one, what do we expect to happen to the other one? Formally:

$$cov(X, Y) = E\left[(X - E[X])(Y - E[Y])\right]$$

If we scale covariance of two random variables by the product of their standard deviation we get **correlation**:

$$\rho_{XY} = \frac{cov(XY)}{\sigma(X)\sigma(Y)}$$

In 1D the (univariate) Gaussian has positive variance σ^2 (i.e. the curve does not *implode* onto itself, it has positive width). In the multivariate case this corresponds to the covariance matrix being **positive definite** (the bell cannot implode onto itself in any dimension) - all eigenvalues are positive.

Positive definite requires Σ that for any x:

$$x^T\Sigma x > 0$$

Note that this will ensure that our exponent in the Gaussian is positive.

Multivariate Gaussian Theorem (KMP Book)

Suppose $x = (x_1, x_2)$ is jointly Gassian with parameters μ, Σ. Then, the marginals are given by:

$$p(x_1) = N(x_1|\mu_1, \Sigma_{11})$$
$$p(x_2) = N(x_2|\mu_2, \Sigma_{22})$$

and the expressions for conditional mean, covariance (i.e. slice of the Gaussian for a given x_2):

$$\mu_{1|2} = \mu_1 + \Sigma_{12}\Sigma_{22}^{-1}(x_2 - \mu_2)$$
$$\Sigma_{1|2} = \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21}$$

Prerequisites to understand GPs

Say you have few data points $D = \{(x_1, f_1), (x_2, f_2), (x_3, f_3)\}$ (say they lie in a line) and you plot them on a 2D grid. Now you're given $x_\star$ and you need to find a corresponding $f_\star$

First, note that covariance matrix describes similarity/correlation between the points. So if we build a covariance matrix Σ for $D_{3\times3}$ its entries would describe the correlation between x_1, x_2, x_3. E.g. Σ_{12} is a similarity measure computed for x_1 and x_2.

Second, note that we'd want Σ to be Symmetric Positive Definite (to give a Bell curve) so we need to define such a similarity measure for our x's that gives a SPD covariance matrix for all pairs of x's in our dataset. That's why we use **kernels**, e.g. Gaussian Kernel

$$K_{ij} = e^{||x_i - x_j||^2}$$

where K_{ij} describes similarity between x_i and x_j and builds our covariance matrix Σ (note that now we use K_{ij} instead of Σ_{ij}). Note that $K_{ij} = 0$ when $||x_i - x_j|| \to \infty$ and $K_{ij} = 1$ when $x_i = x_j$. Using all $x \in D$ we build our K.

Let's assume that f can be modeled by a Gaussian distribution with mean μ and covariance K:

$$f \sim N(\mu, K)$$

where $f = [f_1, f_2, f_3]$ in our case. Then, if you want to find $f_\star$, just stack it with f:

$$\begin{bmatrix} f \\ f_* \end{bmatrix} \sim N\left(\begin{bmatrix} \mu \\ \mu_* \end{bmatrix}, \begin{bmatrix} K & K_* \\ K_*^T & K_{**} \end{bmatrix}\right)$$

where $K_* = [K_{1*}, K_{2*}, K_{3*}]$

Using results from (KPM) we get expressions for μ_* and σ_*:

$$\mu_* = \mu(x_*) + K_*^T K^{-1}(f - \mu(x))$$
$$\sigma_* = K_{**} - K_*^T K^{-1} K_*$$

We've essentially arrived at the function that **takes an input x_* and outputs mean μ_* and a variance σ_***. This is a Gaussian Process.

Note: If your data is noisy, i.e. your dependent variable $y = f(x) + \epsilon, \epsilon \sim N(0, \sigma_y^2)$ all you need to do is to add a $\sigma_y^2 I$ term to the training covariance matrix K.

GP: A Distribution over Functions

A GP is a distribution over functions:

$$f(x) \sim GP(m(x), \kappa(x, x'))$$
$$m(x) = E[f(x)]$$
$$\kappa(x, x') = E[(f(x) - m(x))(f(x') - m(x'))^T]$$

It's a function because you pass x and you get mean and the variance.

Effect of Kernel *Width* Parameter

For a kernel κ:

$$\kappa(x, x') = \sigma_f^2 e^{-\frac{1}{2l^2}(x-x')^2}$$

l is a (kernel) width parameter. The bigger it is the smoother the function. If it's small, e.g. $l^2 = 0.1$ only points that are very close to each other are similiar and thus a *wiggly* shape.

Learning the Kernel Parameters

Marginal Likelihood is a likelihood function in which some parameter variables have been marginalized out (integrated out). Given a set of points $X = (x_1, \ldots, x_n)$ where $x_i \sim p(x_i|\theta)$ (probability distribution parametrised by θ) with θ also being a random variable $\theta \sim p(\theta|\alpha)$, marginal likelihood asks for a probability $p(X|\alpha)$ where θ has been marginalized:

$$p(X|a) = \int_\theta p(X|\theta)p(\theta|\alpha)d\theta$$

In order to learn kernel parameters (e.g. width l) you can use Maximum Likelihood Estimation.

GP Regression and Ridge

If you rewrite Ridge using reparametrisation trick (i.e. use α to get an expression w.r.t XX^T) - take note that XX^T is a matrix where each component is a dot product of (vector) components of X, i.e.

$$XX^T = \begin{bmatrix} x_1 x_1^T & \ldots & x_1 x_n^T \\ \vdots & \ddots & \vdots \\ x_n x_1^T & \ldots & x_n x_n^T \end{bmatrix}$$

where each $x_i \in R^{1\times d}$ is a (data) **row** vector (so x_i^T is a column). Since dot product is a measure of similarity between the vectors we can treat XX^T as **a kernel** K.

As a consequence, **A dual form of Ridge regression is a GP** (but using linear kernel instead of Gaussian).
Note: here we're comparing Ridge to a noisy GP, where we add $\sigma_y^2 I$ term to the training covariance matrix K.

Bayesian Optimisation and Multi-Armed Banits

Aquisition function is a function that tells how to aquire information (i.e. where to evaluate the objective function next) and thus trades-off exploration and exploitation. This forms a basis of *Bayesian Optimisation*:

1. For $t = 1, 2, \ldots n$ do:
    2. Find x_t maximising some aquisition function: $x_t = argmax_x a(x|D_{1:t-1})$
    3. Sample objective function $y_t = f(x_t) + \epsilon$
    4. Augment the dataset: $D_{1:t} = \{D_{1:t-1}, (x_t, y_t)\}$ and update the GP

Bayesian optimisation is used when we don't know the function and it's expensive to evaluate.

Aquisition Functions

The most common aquisition cireria are:

- Probability of Improvement
- Expected Improvement
- GP-UCB
- Thompson sampling

Probability of Improvement

$$PI(x) = P(f(x) \geq \mu^+ + \epsilon) = \Phi\left(\frac{\mu(x) - \mu^+ - \epsilon}{\sigma(x)}\right)$$

where μ^+ is best observed value so far and ϵ is just for numerical stability. Intuitively this is the probability of obtaining higher value then the one seen so far. Geometrically it is the area under the (Gaussian) curve (representing confidence interval for a given x, i.e. a vertical slice) Φ centered at the mean of a given x calculated from μ^+ (horizontal line) up.

Expected Improvement

$$x = argmax_x E(max\{0, f_{n+1}(x) - \mu^+\}|D_n)$$

For this aquisition we can obtain an analytical expression:

$$EI(x) = \begin{cases} (\mu(x) - \mu^+ - \epsilon)\Phi(Z) + \sigma(x)\phi(Z) & \text{if } \sigma(x) > 0 \\ 0 & \text{if } \sigma(x) = 0 \end{cases}$$

where $Z = \frac{\mu(x) - \mu^+ - \epsilon}{\sigma(x)}$, ϕ and Φ denote PDF and CDF of standard Gaussian.

Intuitively here we want to find x for which we expect to see the biggest improvement. In *Probability of Improvement* we wanted x which results in the highest probability of *any* improvement.

Bayes and Decision Theory

Utilitarian view: We choose an action that maximises expected utility, or equivalently, which minimises expected cost. For this, we need to build a cost/reward model (e.g. when deciding whether to treat a patient or not we need to define the cost matrix with costs of TP - treating sick patient, FP - treaing healthy patient, TN - not treating healthy patient and FN - not treating sick patient) and the probability distribution of various states:

$$EU(a) = \sum_x u(x, a)P(x|D)$$

where EU - expected utility, u - utility function, D - data, a - action (e.g. treatment vs no treatment), x - state (e.g. healthy vs sick).

GP-UCB (Upper Confidence Bound)

Define Regret and Cumulative regret:

$$r(x) = f(x^*) - f(x)$$
$$R_T = r(x_1) + \ldots + r(x_T)$$

The GP-UCB criterion gives:

$$GP - UCB(x) = \mu(x) + \sqrt{\nu\beta_t\sigma(x)}$$

With $\nu = 1$ and β_t being a function of $log(t)$ (complicated expression) can be shown with high probability to have zero regret $lim_{T\to\infty}\frac{R_T}{T} = 0$. This implies a lower bound on the convergence rate for the optimisation problem

Thompson Sampling

Instead of defining an aquisition function, draw a sample **function** from the GP posterior (remember, GP is a distribution over functions) and find its maximum to get the next evaluation point x_{t+1}.

Note on Aquisition Functions

The advantage of using BO is that we know we should not explore places where UCB is below the μ^+ because chances of getting anything greater than our current best are negligible, which, at each step, discards a solid portion of the search space

Decision Trees

Classification Trees

- Each node in the decision tree is associated with a decision (question that decides which child to visit) and, in general, contains a probability distribution over all classes (doesn't need to indicate one class only).
- At each node, the feature/attribute that should be used to make a decision should be selected such that it splits the dataset well, i.e. ideally after making a decision we only get data belonging to one class. This is equivalent to minimsing entropy of the data.
- Note we are defining each node/split greedily layer by layer instead of considering all possible combinations of splits we could make, so we are suboptimal.
- If a feature is continuous we might need to discretize it first and then calculate information gain (see below) based on those discretized thresholds
- Tree with three nodes (depth 1) is called a *stamp*

For this we use **Entropy**, which, for a given (probability) distribution (over classes) Π:

$$H(\pi) = -\sum \pi\ log(\pi)$$

i.e. for a n-class problem, $H(\pi)$ would have n terms - one for each class.

Formally

A chose attribute A with K distinct values divides the training set E into subsets $E_1, \dots E_k$. The **Expected Entropy (EH) remaining** after trying attribute A (with branches $1, \dots K$) is:

$$EH(A) = \sum_{i=1}^{K} \frac{p_i + n_i}{p + n} H(\frac{p_i}{p_i + n_i}, \frac{n_i}{p_i + n_i})$$

where p_i indicate positive samples, n_i - negative samples in child i, $p+n$ indicate number of samples in the parent.

Intuitively, for a given attribute/feature A, we want to weight the entropy of each child by the proportion of data that it has and then add all of that up.

Information Gain
Information gain (I) or **Reduction in Entropy** for an attribute A is:

$$I(A) = H\left(\frac{p}{p+n}, \frac{n}{p+n}\right) - EH(A)$$

Intuitively, for an attribute A at a given node we're measuring how much we reduce the overall entropy by choosing A.

- You can use *Gaussian information gain* to decide splits as well (i.e. for a given node you fit gaussians to the data)

Random Forests

Random forest is an **ensemble** method (other enseble technique is **Boosting**). It is consistent, i.e. the resulting estimator is convergent as the nubmer of data grows to infinity (unlike some estimators such as *Nearest Neighbour*). You can split the data into smaller chunks and distribute across a cluster and then combine the resulting trees and that forest is a consistent estimator.

Algorithm

Start with N data points.

1. For $b = 1$ to B:
   a) Draw a bootstrap sample Z^* of size N from the training data
   b) Grow a random-forest tree T_b to the bootstrapped data, by recursively repeating the following steps for each terminal node of the tree, until the minimum node size n_{min} is reached:
   - Select m variables at random from p variables/features
   - Pick the best variable/split-point among the m
   - Split the node into two daughter nodes
2. Output the ensemble of trees $\{T_b\}_1^B$

where B is a number of trees, **bootstrapping** denotes random sampling with replacement and **baggining** refers to averaging multiple estimators.

- Making inference using a forest simply requires passing a test data point through all of the trees and then averaging all the results (in classification - calculating an average over all distributions)
- When deciding the values for making a split at a given node for a given feature (e.g. in continuous case), the algorithm usually considers only the values of the data points. e.g. say we have three

data points with heights $h_1 = 1.82, h_2 = 1.76, h_3 = 1.55$ we will consider $1.82, 1.76, 1.55$ as split thresholds for this feature

- Trees in a forest have a **huge variance** because each tree is very different from other trees. However, since they are different, they are not correlated. That's why, when we average them, we get rid of variance, keeping bias low.

Text Classification Example

We can build a text classifier that recognizes spam vs. not spam using RFs. Let's consider a dataset made of texts (e.g. emails) being vectors of words and a given word dictionary (huge vector with all words). We want to build one tree using a (random) subset of words from a dictionary vector. Now, for each of those words we compute the information gain of our dataset (the decision at each tree node is *is this word in a given sample or not?*)

Hyperparameters

Tree Depth
If a tree is too shallow - risk of underfitting. If its too deep - overfitting

Bagging
RF with **no bagging** is a **max margin** classifier. i.e. it tends to find the extreme data points belonging to different classes and puts a decision boundary in the middle between them. *Margin* is a distance from the decision boundary to the closest point of a given class.

RF with bagging will usually not put the decision boundary in the middle between the extreme data points. This helps dealing with outliers, i.e. in max margin it is the outliers that define the boundary.

Boosting

When building normal RF, each tree is independent. **Boosting** is a technique where we build trees in sequence. After building a tree we evaluate it, add to the ensemble and then reassign dataset weights to favour misclassified samples.

> *Boosting is an ensemble meta-algorithm for primarily reducing bias, and also variance in supervised learning, and a family of machine learning algorithms that convert weak learners to strong ones. A weak learner is defined to be a classifier that is only slightly correlated with the true classification (it can label examples better than random guessing)*

Regression Trees

In regression trees you bild trees in exactly the same way with a difference that, in the leaf, you fit a (usually simple) regression model, e.g. linear model based on the data that fell to that leaf node.

Unconstrained Optimisation

First-order methods use first derivatives (gradient) and **Second-order methods** use second derivatives (hessian) to find a minimum.

Gradient Vector

Let θ be a $d-$dimensional vector and $f(\theta)$ a scalar-valued function. The gradient vector of f w.r.t θ is :

$$\nabla_\theta f(\theta) = \begin{bmatrix} \frac{\partial f(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial f(\theta)}{\partial \theta_d} \end{bmatrix}$$

- Gradient is perpendicular to the contour plot of a function

Hessian Matrix

The Hessian matrix of f w.r.t θ, written $\nabla^2_\theta f(\theta)$ denoted H is a $d \times d$ matrix of partial derivatives:

$$\nabla^2_\theta f(\theta) = \begin{bmatrix} \frac{\partial f(\theta)}{\partial \theta_1 \partial \theta_1} & \cdots & \frac{\partial f(\theta)}{\partial \theta_1 \theta_d} \\ \vdots & & \vdots \\ \frac{\partial f(\theta)}{\partial \theta_d \partial \theta_1} & \cdots & \frac{\partial f(\theta)}{\partial \theta_d \theta_d} \end{bmatrix}$$

- The Hessian matrix is a symmetric matrix, since the hypothesis of continuity of the second derivatives implies that the order of differentiation does not matter (Schwarz's theorem)
- Hessian measures the curvature of a function at a given point
- The Hessian matrix of a function f is the Jacobian matrix of the gradient of f; that is:

$$H(f(x)) = J(\nabla f(x))$$

Where Jacobian is a matrix of first-order partial derivatives of a vector-valued function $f : \mathbb{R}^n \to \mathbb{R}^m$ (i.e. f has n variables $x_1, \dots x_n$ and f consists of m components: $f_1, \dots f_m$):

$$J(f(x)) = \begin{bmatrix} \nabla^T f_1 \\ \vdots \\ \nabla^T f_m \end{bmatrix}$$

Steepest Gradient Descent Algorithm

Gradient descent/steepest descent can be written as follows:

$$\theta_{k+1} = \theta_k - \eta_k \nabla f(\theta_k)$$

where k indexes steps of the algorithm, θ is the parameter vector and $\eta_k > 0$ is the **learning curve** or **step size**.

Steepest Gradient Descent for Linear Regression

$$f(\theta) = (y - X\theta)^T (y - X\theta)$$

The derivative w.r.t θ:

$$\nabla f(\theta) = \nabla (y^T y - 2y^T X\theta + \theta^T X^T X\theta) = -2X^T y + 2X^T X\theta$$

Note, this is equal to $-2\sum_i x_i^T(y - X\theta)$ in vector form.
The hessian:

$$\nabla^2 f(\theta) = 2X^T X$$

Newton's Algorithm

The most basic second-order optimization algorithm is **Newton's algorithm**:

$$\theta_{k+1} = \theta_k - H_k^{-1} g_k$$

Note: here, we use a Hessian matrix instead of η. The update is derived by making a second-order Taylor series approximation of $f(\theta)$ around θ_k:

$$f(\theta) \approx f(\theta_k) + g_k^T(\theta - \theta_k) + \frac{1}{2}(\theta - \theta_k)H_k^{-1}(\theta - \theta_k)$$

where g_k denotes a gradient and H_k denotes a hessian at step k and $\Delta\theta = \theta - \theta_k$In order to solve for θ_{k+1} differentiate and equate to zero - this is essentially finding a minimum of a quadratic.

- If you apply Newton's algorithm to Linear Regression you will find the least squares solution
- Hessian is very expensive to store so there is a big memory cost

Netwton CG (Conjugate Gradient) Algorithm

Instead of computing $d_k = -H_k^{-1} g_k$ directly (which might be expensive because of the matrix inversion), we can solve a system of linear equations $H_k d_k = -g_k$.

One efficient and popular way to do this, especially if H is sparse, is to use a **conjugate gradient method** to solve the linear system. Another advantage of this approach is that we don't need to iterate until convergence, especially during the first iterations. We only need to approximate d_k by running CG for few iterations.

Pseudocode

1. Initialise θ_0
2. for $k = 1, \ldots$ until convergence do
   2.1 Evaluate $g_k(\theta) = \nabla f(\theta_k)$
   2.2 Evaluate $H_k = \nabla^2 f(\theta_k)$
   2.3 Solve linear system $H_k d_k = -g_k$ for d_k
   2.4 Use line search to find stepsize η_k along d_k
   2.5 $\theta_{k+1} = \theta_k + \eta_k d_k$

In 2.3 you can use *minres* **(Minimum Residual Iteration)** algorithm to solve it, or **Conjugate Gradient** if it's symmetric and positive-definite. Both of the algorithms come from a family of methods called **Krylov methods**. The cost of those methods is $O(Kn^2)$ where K is the number of iterations.

Estimating Mean Recursively

We can represent the mean θ_N using a recursive relation:

$$\theta_N = \frac{1}{N}\sum_{i=1}^{N} x_i = \frac{1}{N}x_N + \frac{N-1}{N}\theta_{N-1}$$

where θ_N denotes an average of N x_i terms. The first expression is a **batch** way of computing mean and the second - an **online** way.

Online aka Stochastic Gradient Descent (SGD)

The word **stochastic** here means that we make an update after not seeing all data but only one sample (aka **Online Learning**).

> The key difference compared to standard (Batch) Gradient Descent is that only one piece of data from the dataset is used to calculate the step, and the piece of data is picked randomly at each step.

The difference between Batch and Stochastic Gradient Descent is essentially like computing a mean (above), i.e. batch vs. recursive (online) way. **Mini-batch** is a method inbetween, where we use a subset of the data (more than one sample at a time) to compute the update.

Logistic Regression and Neuron Models

Logstic Regression, unlike Linear Regression, is not analytic, so you can't find the closed-form expression for the optimum.

McCulloch-Pitts Neuron Model

For a given artificial Neuron k let there be $d+1$ inputs with signals $x_{k0}, \dots x_{kd}$ and weights $\theta_{k0}, \dots \theta_{kd}$ (where $x_0 = 1$ and $\theta_{k0} = b_k$ i.e. bias term). The output of of the neuron is:

$$y_k = x_k^T \theta_k$$

where $x_k \in \mathbb{R}^{d+1}$ is input data vector for neuron k and $\theta_k \in \mathbb{R}^{d \times 1}$ is a vector of weights associated with that neuron.

Sigmoid Function

$sigm(x)$ refers to **sigmoid function**, aka **logistic** or **logit** function:

$$sigm(\eta) = \frac{1}{1+e^{-\eta}} = \frac{1}{1+e^{-x_k^T\theta_k}}$$

Where $\eta = x_k^T \theta_k$ The reason we use sigmoid function instead of e.g. gate function (0 for $x < 0$ and 1 for $x > 0$) is that sigmoid is differentiable.

Linear Separating Hyper-plane

Since $sigm(x) = 0.5$ for $x = 0$ we can treat the sigmoid as the probability of obtaining given class (in a binary classification problem). The plane $x^T\theta = 0$ would be a class separator (aka discriminator), i.e. when $x^T\theta > 0$ we have class 1 and class 0 otherwise.

Logistic Regression

The logistic regression model specifies the probability of a binary output $y_i \in \{0, 1\}$ given the input x_i. The goal is to model the probability of a random variable y_i being 0 or 1 given training data. The likelihood function, assuming all the observations are idependently Bernoulli distributed is as follows:

$$p(y|X, \theta) = \prod_i^n Ber(y_i|sigm(x_i\theta)) = \prod_i^n sigm(x_i\theta)^{y_i} \times (1 - sigm(x_i\theta))^{1-y_i}$$

where Ber denotes Bernoulli (binary) distribution.

(Negative) Log-likelihood for Logistic Regression

As before, if we want to maximise the likelihood, we need to minimize the negative log likelihood, which is as follows:

$$J(\theta) = -log(p(y|X, \theta)) = \sum_i^n y_i \log(sigm(x_i\theta)) + (1 - y_i)\log(1 - sigm(x_i\theta))$$

which is a **Cross-Entropy** between y and $sigm(x_i\theta)$ (because it's across two quantities). Another view to the MLE is that, since entropy is uncertainty, we want to minimise the uncertainty.

Gradient and Hessian of Binary Logistic Regression

The gradient of $J(\theta)$ is:

$$\frac{d}{d\theta}J(\theta) = \sum_i^n x_i^T(\pi_i - y_i) = \mathbf{X^T}(\pi - \mathbf{y})$$

and the Hessian:

$$H = \frac{d}{d\theta}g(\theta)^T = \sum_i \pi_i(1 - \pi_i)x_i x_i^T = \mathbf{X^T}\text{diag}(\pi_\mathbf{i}(\mathbf{1} - \pi_\mathbf{i}))\mathbf{X}$$

One can show that H is positive definite; hence the problem is convex and has a unique global minimum.

Iteratively Reweighted Least Squares (IRLS)

IRLS is the name for a Newton method applied to logitic regression. Simply substitute the expressions for the gradient and the hessian to:

$$\theta_{k+1} = \theta_k - H_k^{-1}g_k$$

Note that the method is iterative (we don't necessarily find the optimum in the first iteration as in Linear Regression)

Bayesian Logistic Regression

Say we want to find θ using Bayesian approach, we need to find the posterior distribution $P(\theta|D)$ given the likelihood $p(y|X, \theta)$ (see expression above) and a prior $p(\theta)$ (say a Gaussian):

$$P(\theta|D) = \frac{p(y|X,\theta)p(\theta)}{p(y|X)} = \frac{p(y|X,\theta)p(\theta)}{\int p(y|X,\theta)p(\theta)}$$

Where D is the dataset. Note that putting a Gaussian prior $p(\theta)$ with mean of zero is essentially equivalent to solving the optimisation problem using MLE by putting a regularizer - that's because a zero-centered Gaussian enforces θ to be small.

- Note that $p(y|X,\theta)$ and $p(D|\theta)$ mean the same thing (in Logistic Regression - likelihood, i.e. Bernoulli model)
- Note that in Bayesian approach we don't only want to find a mode of the posterior (i.e. θ_{MAP}) but the whole distribution. This is because we care about the uncertainty (variance) of the distribution. That's why we need to find a normalization factor.

The problem is that the normalisation factor $p(y|X)$ for the posterior requires itegrating out theta from the nominator expression $p(y|X,\theta)p(\theta)$:

$$p(y|X) = \int p(y|X,\theta)p(\theta)d\theta$$

which is intractable. However, we can approximate this integral using **Monte Carlo**

Bayesian Logistic Regression: Predictive Distribution

Assuming we have found a posterior distribution $p(\theta|D)$ and a new test point x_{n+1} we want to predict y_{n+1}:

$$p(y_{n+1}|x_{n+1},D) = \int p(y_{n+1},\theta|x_{n+1},D,\theta)d\theta = \int p(y_{n+1}|x_{n+1},D,\theta)p(\theta|x_{n+1},D)d\theta$$
$$= \int p(y_{n+1}|x_{n+1},\theta)p(\theta|D)d\theta$$

- Note that, in Bayesian Statistics we don't want to find the maximum a posteriori θ and use it to make predictions. We want to integrate over the entire posterior probability distribution (as we also care about the uncertainty) and thus we need to sum over all possible values of θ.
- Note, in the last integral we dropped D from the $p(y_{n+1}|x_{n+1},D,\theta)$ - this is because once we know θ, D isn't needed anymore - θ encapsulates our knowledge about the data.

Monte Carlo

The rationale behind Monte Carlo is that integration in high-dimensional spaces is expensive. Say we want to compute an area of a 2D shape. We need to put a mesh/grid and then count the number of boxes that fit within the shape, whic requires n^2 boxes. To compute a volume in 3D we need n^3, so for a d-dimensional problem, the integral will require n^d (exponential size) of such boxes using such discretization approach. This is called a **curse of dimensionality**.

Suppose we want to compute

$$I = \int f(x)p(x)dx$$

In order to approximate the integral, we need to be able to sample from $p(x)$. By sampling $x \sim p(x)$ we can approximate the distribution with histograms for x (by counting x's in each range), by the Strong Law of Large Numbers (SLLN): *A sample average converges to the expectation*. Then the integral becomes:

$$I \approx \frac{1}{N} \sum_{i=1}^{N} f(x^{(i)})$$

The question is: Howe do we draw samples from an arbitrary distribution?

Importance Sampling

The setup is: let's assume we want to solve the predictive distribution for Bayesian Logistic Regression. We don't know how to sample from $p(\theta|D)$ (but we can evaluate it, i.e. have an expression). All we need to do is to introduce another distribution from which we can sample easily (e.g. Gaussian), say $q(\theta) = N(0, \Sigma)$:

$$I = \int f(\theta) p(\theta|D) d\theta = \int f(\theta) \frac{p(\theta|D)}{q(\theta)} q(\theta) d\theta$$

which can be approximated using MC by sampling $\theta \sim q(\theta)$:

$$I \approx \frac{1}{N} \sum_{i=1}^{N} f(\theta^{(i)}) \frac{p(\theta^{(i)}|D)}{q(\theta^{(i)})}$$

- Importance Sampling works well in low-dimensional spaces but fails in high-dimensional spaces. This is because Importance Sampling requires $q(\theta)$ to have a support for the whole posterior (i.e. the distribution must be positive where the posterior is positive) in all dimensions, which is very hard to ensure in high dimensions (i.e. there will be some dimensions in which $q(\theta)$ is very small but the posterior, which we're trying to find, will have high probability) because we don't have enough probability density to provide support everywhere. The alternative is a **Markov Chain Monte Carlo** which helps navigating in the high-dimensional space
- Sequential Monte Carlo (SMC) or Sequential Importance Sampling is called a **Particle Filter**

Neural Networks

Multi-Layer Perceptron - 1 Neuron

- Logistic Regression is just a Neural Network with one neuron. The problem with 1 Neuron was that it could only represent a linear separator, which did not allow to model e.g. XOR function. The solution was to introduce hidden layers and nonlinearities into the network.

MLP - Regression

- In the regression case you should change the output layer activation function to sigmoid and this should result in a continuous outocome.

- There is a theoretical result that one-layer NN can approximate any smooth function arbitrarily well (depending on the number of neurons).
- The resulting function is a weighted sum of (input) sigmoids. Since you essentially combine multiple "bumps" (sigmoids) you can appproximate any nonlinear smooth function. Note that here the network parameters θ control the shape of those sigmoids (in particular, when θ is negative we flip the shape of the sigmoid). In nature it's similar to composing multiple RBFs.

MLP - Multiclass

In a multilcass setting (say with n classes) it's useful to use one-hot encode the output data and then build an MLP with n output neurons, where each output neuron n_i outputs a real value (i.e. identity/lack of activation function) associated with class y_i. In order to get a probability distribution we use a **Softmax** function. This allows us not to pass each output-neuron's outcome by a sigmoid function thus saving some compute.

Softmax

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

where $z \in \mathbb{R}^K$ is an input vector with K real numbers

Overall, NNs are essentially the same and they differ by hyperparameters and the probability model that we use to define the probability of an output $P(y_i|X_i, \theta)$, which depends on the nature of the data. If the data is binary - we use a Bernoulli distribution (Bernoulli distribution is a special case of Binomial Distribution for a single trial), if it's continuous - we use Gaussian (i.e. regression case), if it's multiclass - we use Multinomial distribution. But depending on the nature of the data we could also use Poisson, Inverse-Wishart etc.

The probabilistic model is needed to describe the likelihood function and then derive the cost by calculating a negative log-likelihood. Finally, we need some optimisation method to find the optimum for that cost function.

Backpropagation

Backpropagation is essentially a chain rule that allows us to formulate the gradient of the output $\hat{y_i}$ w.r.t. some θ_n (i.e. $\frac{\partial \hat{y_i}}{\partial \theta_n}$) using partial derivatives of outputs along the path.

Example
Say the NN has some (input) weight θ_3 in the first layer that goes to node n_{12} (note first lower index indicates the layer and the second - the node within the layer) that outputs some u_{12}, which after activation gives o_{12}. Then, o_{12} gets fed into node n_{21} which outputs u_{21}, which, after passing through sigmoid, gives o_{21}, which is our prediction $\hat{y_i}$. The backpropagation formula gives us:

$$\frac{\partial \hat{y_i}}{\partial \theta_3} = \frac{\partial \hat{y_i}}{\partial u_{21}} \times \frac{\partial u_{21}}{\partial o_{12}} \times \frac{\partial o_{12}}{\partial u_{12}} \times \frac{\partial u_{12}}{\partial \theta_3}$$

Vanishing Gradient Problem

The problem with sigmoid function is that its derivative is equal to:

$$\frac{d}{dx}sigm(x) = \frac{d}{dx}\frac{1}{1+e^{-x}} = \ldots = sigm(x)(1-sigm(x))$$

and since $sigm(x) \in (0,1)$ if there are many layers, the backpropagation formula makes many such products of small numbers which makes the gradient vanish to zero. To avoid that we might use different activation functions such as **hyperbolic tangent or ReLU**.

Deep Learning I

In Computer Vision each neuron is thought to be reacting to a certain pattern (e.g. a vertical line). Patch is a 2D area with one such pattern (it's called a **local receptive field** in the brain) and each of our neurons gets activated (or not) if, in the input image, a pattern of a given patch (e.g. a vertical line) is present. For a given image, many of those local receptive fields can fire, depending on the patterns present there.

Each patch can be represented as a weight matrix of size $\theta \in \mathbb{R}^{n \times n}$ assuming a patch is of size $n \times n$.

Convolutional Architecutre is all about shifting patches through an image (i.e. sliding window) thus detecting the presence of certain patterns in certain image locations.

Autoencoders

The idea behind autoencoders is that, once we've learned something (e.g. a pattern/patch) we should be able to imagine the image representing this patch. The aim is to create a NN that takes as input an image and outputs the same image (i.e. a huge vector after flattening).

Note that for an image/patch of size n we're encoding it using $m < n$ neurons only (where each neuron is associated with a weight matrix/vector that represents one patch). This layer is called a **bottleneck** and is meant to capture important information about the input and thus it's by design smaller than the input (i.e. $m < n$). You can treat this bottleneck layer as a kind of hash for an image.

More formally, we'd like our neuron layer to produce predictions, $\hat{x}$ that are as similar to x as possible. The objective is to minimise:

$$\arg\min_W \frac{\lambda}{m}\sum_{i=1}^{m}||W^T W x^{(i)} - x^{(i)}||_2^2 + \sum_{i=1}^{m}\sum_{j=1}^{k} g(W_j x^{(i)})$$

where m is the number of neurons/activations/patches, $x^{(i)} \in \mathbb{R}^{n \times 1}$ (i.e. a column vector), $W \in \mathbb{R}^{m \times n}$ is a weight matrix (such that $Wx^{(i)} \in \mathbb{R}^{m \times 1}$ is a vector of activations a) and $W^T W x^{(i)} \in \mathbb{R}^{n \times 1}$ being our prediction (encoded image) $\hat{x}$.

The second term containing g is a regularizer wth g being a penalty $log(cosh(x))$, i.e. a smooth L1 norm; $g(x) = log(cosh(x))$ looks like an absolute value function $|x|$ but is differentiable at $x = 0$. Note that the nested sums (one over i and one over j) have nice interpretation. For a given i (image patch) you want to minimise weights over all k features/activations, which means that you only want to

activate certain neurons (it is inefficient to fire all of them all the time). Alternatively, for a given j (feature) you want to minimise the weight over all patches so you want it to fire only for some specific patterns.

The input weight W is called an **encoder** and the output weight (here - also W, but this doesn't need to be the case) is called a **decoder**.

- Note that in the approach above, no activation function is being used. If we used a nonlinear activation we'd call it a **nonlinear autoencoder**
- Autoencoder achieves dimensionality reduction by using bottleneck layers. In fact, the objective function to minimise without the regularization term is equivalent to PCA

Self-taught Learning / Transfer Learning

The idea is that once the NN has learned something, it can be transferred to other, similar task, and continue to recognize features. It should be sufficient to retrain only shallower layers of the network and not the entire network.

Autoencoders with Pooling (Simple and Complex Cells)

Adding pooling to autoencoders means adding a pooling layer after the activation layer that takes e.g. a max over given activation neurons. So if a given pooling unit is connected to few activation units (note the layers don't need to be fully connected) it will take a max of activations of activation neurons that it's connected to.

Pooling allows us to achieve some invariance to transformations, e.g. translations. This is because if an image is shifted, a nearby activation neuron might be activated (e.g. neuron $i+1$ instead of neuron i which was activated before the image shift), but to the pooling layer this image translation should not really make any difference.

L2 Pooling (alternative to max pooling) is essentially taking activation a_k and weighting it with a pooling weight H_{ik} (and taking a square root):

$$P_i = \sqrt{\sum_{k=1}^{m} H_{ik} a_k^2}$$

where P_i is a pooling activation from pooling unit i and m is the number of activation units a. Note weight matrix H is not learned - it is being designed (i.e. defines the connections between the pooling layer and the activation layer). Those layers are called **simple and complex cells**.

- Simple cell is also known as a **filter**

Greedy Layer-wise Training

We can learn complex features by iteratively learning one feature (hidden) layer at a time using autodecoder. In the first iteration we learn how to create an image $\hat{x}$ from input x by learning parameters of activation layer a (let's call it h_1, for *hidden*). In second iteration we get rid of the output layer and put a h_2 activation layer and try to, given h_1 as input learn to generate $\hat{h_1}$. This way we

stack one layer at a time, where each layer is smaller and is able to generate more and more complex features (e.g. eyes).

Deep Learning II

L2 Pooling and Local Contrast Normalization (LCN)

- In L2 Pooling, the activation in layer $i+1$ is an L2 norm of an input patch/image in layer i. E.g. if we have an image of size $n \times n$ in layer i we compute a square root of the sum of squares of all pixels.
- Local Contracts Normalization refers to scaling given patch by subtracting its mean and dividing by its standard deviation
- Pooling allows for some transformation invariance (e.g. rotation, translation)
- LCN allows for some scale invariance, i.e. we can scale (multiply) each patch by some value but the activation from LCN unit will be the same (CV ituition: light intensity)

Reverse Engineering a Neuron

Let's say we want to know what's the input that causes the highest excitation (activation) in a given neuron. What we need to do is to find x^* such that:

$$x^* = \text{argmax}_x f(x|W,H) \quad s.t. ||x||_2 = 1$$

this gives us an idea of what feature a given neuron is representing (e.g. a face), which allows us to reverse-engineer a NN.

Dropout

The premise behind dropout is that ensemble techniques like RFs proved to be very powerful. Dropout allows us to average many large neural networks by randomly omitting each hidden unit with some probability. In theory, since there is a probability p of selecting any given unit, there are p^H achitectures (where H is the number of hidden units) that share weights.

- At each time, only a small subset of those architectures get trained and they get only one training sample, which is an extreme form of bagging

Importance Sampling and MCMC I

Maximum a Posteriori (MAP)

The MAP estimate, $\hat{\theta}_{MAP}$ is the mode of the posterior distribution, i.e. its the estimate of the unknown parameter θ resulting from multiplication of a prior distribution $p(\theta)$ and a likelihood distribution $P(D|\theta)$.

Probability vs Density (Notation)

The posterior $p(\theta|D)$ is a probability **density** function, so the probabilities are areas under that curve:

$$P(d\theta|D) = P(\theta|D)d\theta$$

where $P(\theta|D)$ is a *height* of a density and $d\theta$ is the width (aka a *measure*). $P(d\theta|D)$ can be also written as $dP(\theta|D)$. Note that this notation is sometimes used by authors to indicate the distinction between the probability and density. That's why some itegrals might not necessarily have trailing $d\theta$. $p(d\theta)$ is a Lebesgue integral.

MCMC

- Monte Carlo - approximate an integral by a sum (empirical average)
- Markov Chain - mechanism by which we generate samples

Transition matrix in Markov Chain, $T = p(x_t|x_{t-1})$ describes the probability distribution over the set of states given a current state. Usually the rows of the matrix are normalized (such that all the entries sum to 1). Markov property ensures that x_t only depends on x_{t-1}.

Note: $T \in \mathbb{R}^{N \times N}$ is a stochastic matrix (it describes probability distribution over states). As long as the graph (state space) is **aperiodic** and **irreducible**, we have that for any initial vector of probabilities v:

$$\lim_{t \to \infty} v^T T^t = \pi^T$$

where $T \in \mathbb{R}^{N \times N}$ is a stochastic transition matrix, $v^T \in \mathbb{R}^{1 \times N}$ is a (row) vector of probabilities, and π is the **invariant** or **stationary** distribution of the chain. It is unique. Google uses such distribution as a **PageRank**, i.e. an importance of each page (when web-crawling).

In the limit we have:

$$\pi^T T = \pi^T$$

so π is a left eigenvector of T. This means that for e.g. PageRank, the page rank is simply an eigenvector of a web graph. Componentwise, we have:

$$\sum_{i=1}^{N} \pi_i T_{ij} = \pi_j$$

and in the continuous case:

$$\pi(x_{t+1}) = \int \pi(x_t) p(x_{t+1}|x_t) dx$$

where $p(x_{t+1}|x_t)$ is a Markov Chain Kernel (transition probability distribution) - think of it as a infinitely big transition matrix.

Notation note: Kernel $k(x, B)$ describes the probability of going from x to interval B. It is an interval, because in the continuous space the probability of a point is 0.

Irreducibility

Irreducibility means that the graph needs to be connected (it cannot be reduced to subgraphs)

Aperiodicity

The term periodicity describes whether something (an event, or here: the visit of a particular state) is happening at a regular time interval. Here time is measured in the number of states you visit.
e.g.

$$T = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

is a periodic transition matrix - for any $v \in \mathbb{R}^2$ the transition matrix will swap its components. The solution to that would be adding/subtracting ϵ in each row to achieve aperiodicity.

MCMC: Metropolis-Hastings

Metropolis-Hastings algorithm arises if we reverse the MCMC problem, i.e. assuming we have the stationary distribution π and we're trying to construct a T that gives us samples from π. MCMC MH essentially gives a way to create new samples iteratively as if they were coming from this transition kernel T. The algorithm needs to satisfy the property called **Detailed Balance**. The requirement is that we know π up to the constant and we're in high-dimensional space (otherwise other methods such as importance sampling would do).

In case of Logistic Regression, we wanted to find $p(\theta|D)$, or to be able to sample $\theta \sim p(\theta|D)$. However, we did not know the normalisation constant. Metropolis-Hastings gives us a way to sample from that distribution without knowing that constant.

Notation note: $p(\theta|D)$ is the distribution that we want to find. Here it's denoted as $\pi(x)$ (pi - posterior).

Pseudocode

1. Initialise x_0
2. For $t = 0 \dots N-1$ do:
   2.1 Sample $u \sim U_{[0,1]}$
   2.2 Sample a candidate $x^* \sim q(x^*|x_t)$
   2.3 if $u < \frac{p(x^*)q(x_t|x^*)}{p(x_t)q(x^*|x_t)}$: $x_{t+1} = x^*$
   2.4 else: $x_{t+1} = x_t$

where $q(x^*|x_t)$ is a proposal distribution (e.g. Gaussian centered at x_t) that proposes new samples in the vicinity of x_t, u is just a random number $u \in [0, 1]$ that allows us to accept or reject a sample with some probability. The algorithm returns the samples $\{x_0, \dots x_N\}$ that should approximate $\pi(x)$.

The intuition behind MCMC MH is that we iteratively sample from a target distribution $\pi(x)$ by drawing new samples x_{t+1}. At each iteration the procedure proposes a new x^* (from the proposal distribution) and we either accept it or reject it. If we accept it, x^* becomes our new sample x_{t+1}, otherwise our new sample x_{t+1} is our old sample x_t. We will accept a new sample if it has higher probability according to the target (posterior) distribution (that's where $\pi(x)$ comes into picture - this is because we evaluate $p(x)$ according to the target distribution, which we know up to the value of constant). The trick is to compute the ratio $\frac{p(x^*)}{p(x_t)}$ (where x^* is a proposal sample under consideration),

because the normalization constants cancel each other out. If this ratio is high (i.e. > 1) we accept a new sample: $x_{t+1} = x^*$, otherwise we accept it with some probability proportional to this ratio. Overall, the histogram of all of those samples together should approximate our target distribution.

Note, the weighting ratio $\frac{q(x_t|x^*)}{q(x^*|x_t)}$ reflects the fact that our proposal distribution might not be symmetric in a sense that it might be easier to get from $x^* \rightarrow x_t$ than $x_t \rightarrow x^*$. If it is indeed easier to get $x_t \rightarrow x^*$ (than the other way round) we want to penalise the ratio a bit. For a Gaussian q the distribution is symmetric so the ratio of q's cancels out.

Detailed Balance

Detailed Balance requires the process to be *reversible*, i.e.

$$\pi(x_t)p(x_{t+1}|x_t) = \pi(x_{t+1})p(x_t|x_{t+1})$$

A Markov Process is called a **reversible Markov Process** or **reversible Markov Chain** if it satisfied Detailed Balance.

> A good analogy comes from thinking about traffic flow in New York City and surroundings: let each
> borough or suburb be represented by a node of a Markov chain, and join two nodes if there is a bridge or tunnel connecting the boroughs, or if there is a road going directly from one to the other. For example, the node corresponding to Manhattan would be connected to Jersey City (via the Holland tunnel), to Weekawken (via the Lincoln tunnel), to Fort Lee (via the George Washington bridge), to Queens (via the Queensboro bridge), etc. Suppose that cars driving around represent little elements of probability. The city is in global balance, or the stationary distribution, if the number of cars in Manhattan, and in all other nodes, doesn't change with time.

> This is possible even when cars are constantly driving around in circles, such as if all cars leave Manhattan across the Holland tunnel, drive up to Fort Lee, and enter Manhattan via the George Washington bridge. As long as the number of cars per unit time leaving Manhattan across the Holland tunnel, equals the number per unit time entering Manhattan via the George Washington bridge, the number of cars in Manhattan doesn't change, and the system can be in global balance.

> The city is in detailed balance, only if the number of cars that leaves each bridge or each tunnel per unit time, equals the number that enter that same bridge or tunnel. For example, the flux of cars entering Manhattan through the Holland tunnel, must equal the flux of cars exiting Manhattan through the Holland tunnel, and similarly for every other bridge or tunnel. Not only does the number of cars in Manhattan remain constant with time, but the fluxes of cars across each single bridge or tunnel separately must be equal in each direction.

Variations of Metropolis-Hastings

Simulated Annealing

If we take the ratio to the power of $\frac{1}{T}$: $\frac{p^{1/T}(x^*)}{p^{1/T}(x_t)}$ we will get samples concentrated on the modes of the distribution $p(x)$ (i.e. we explore less). This is called a **Simmulated Annealing**.