Chapter 1

4F10 Deep Learning and Structured Data

Please note that the margins of these notes can be used to check factual recall simply by covering up the right hand text.

1.1 Fundamentals

Consider a *classification task* where we need to classify an unseen observation, $\mathbf{x}^*$, having seen a dataset, $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots (\mathbf{x}_N, y_N)\}$ where $y_i \in \{1, \ldots, K\}$. Denote the decision made as ω, the 'correct' outcome as y^* and the *loss* as $\mathcal{L}(\omega, y^*)$.

Bayes' Decision Rule

Bayes' Decision Rule states that the optimal decision is the decision which minimises the expected loss, which quantifies the cost of making a particular incorrect decision.

$$\omega^* = \arg\min_{\omega} \sum_{i=1}^{K} \mathcal{L}(\omega, i)\ p(y^* = i|\mathbf{x}^*) \tag{1.1}$$

MoG Decision Boundary

In the situation where the loss is uniform, this corresponds to selecting the class with the highest posterior probability. The decision boundary is thus when the class posteriors have the same value. For the *Mixture of Gaussians* model, the decision boundary is hyper-quadratic, collapsing to linear if the covariance matrices are the same.

A model is trained in order to learn the probability distribution, $p(y^* = i|\mathbf{x}^*)$, the distribution required to make an optimal decision. Using machine learning, the classifier, $f(\mathbf{x}^*, \boldsymbol{\theta})$, which has model parameters $\boldsymbol{\theta}$, is used to make the decision.

$$\mathcal{L}_{\text{act}} = \int \left[\sum_{i=1}^{K} \mathcal{L}(f(\mathbf{x}^*, \boldsymbol{\theta}), i)\ p(y^* = i|\mathbf{x}^*) \right] p(\mathbf{x}^*)\ d\mathbf{x}^* \tag{1.2}$$

$$\simeq \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(f(\mathbf{x}^*, \boldsymbol{\theta}), y_i) = \mathcal{L}_{\text{emp}} \tag{1.3}$$

In the limit of large N, $\mathcal{L}_{\text{act}} = \mathcal{L}_{\text{emp}}$ by Monte Carlo.

Generalisation

In practice, we are not interested in the performance on the training data but rather the performance on unseen, held-out data. Optimising empirical loss may not yield good 'performance'; it is possible to achieve zero loss simply by memorising the training data sample.

Types of Models

Broadly, there are two types of model:

- *Generative models* model the joint distribution of observations and classes i.e. $p(\mathbf{x}, \omega|\boldsymbol{\theta})$. The posterior is straightforward to find using Bayes' rule. If the class conditions and priors are correct and an appropriate training algorithm is used in the limit of infinite data, these classifiers give the minimum error probability.

- *Discriminative models* directly train the posterior distribution of the class given an observation.

Error Probability

For the classification task, the probability of error can be written by marginalising over the joint distribution.

$$P[\text{error}] = \sum_{i=1}^{K} \sum_{j=1;\ j\neq i}^{K} \int_{\Omega_j} p(\mathbf{x}, i)\, d\mathbf{x} \tag{1.4}$$

where $\Omega_j = \{\mathbf{x} \in \mathbb{R}^n : f(\mathbf{x}, \boldsymbol{\theta}) = j\}$ i.e. the area over which the classifier predicts the point to be class j.

Maximum Likelihood

One method for training model parameters is by *maximum likelihood* i.e.

$$\boldsymbol{\theta}_{\text{MLE}} = \arg\max_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) \qquad \mathcal{L}(\boldsymbol{\theta}) = p(\mathcal{D}|\boldsymbol{\theta}) \tag{1.5}$$

Note that often the *log-likelihood* is maximised; the monotonicity of the log function means that the maximising value of $\boldsymbol{\theta}$ is the same in both cases.

Independence Notation

The notation $X \perp\!\!\!\perp Y|Z$ indicates that X and Y are conditionally independent given Z i.e. $p(X, Y|Z) = p(X|Z)p(Y|Z)$. Conditional independence structure is powerful, allowing large probability tables to be represented using smaller table sizes.

Language Models

A language model is a **high dimensional** probability distribution over sequences of words.

Problem with ML

Whilst we could use a maximum likelihood approach over sequences with no conditional independence assumptions, almost all table entries would be zero as only a small number of sequences are observed. Even so, working with fully flexible joint distributions is intractable.

Markov Models

Instead, *structured distributions* are utilised where the joint distribution is written as a product of simpler factors. For example, *Markov Models* use a conditional independence assumption with the product rule. Denote W_t as the word at time index t; for a first-order Markov model $W_{t+1} \perp\!\!\!\perp W_{t-1}, \ldots, W_1 | W_t$ i.e. the future is independent of the past given the present. The factorisation of a word sequence is now more simple e.g. for a second order Markov model

$$p(W_1, \ldots, W_T) = p(W_1)p(W_2|W_1)p(W_3|W_2, W_1) \ldots p(W_T|W_{T-1}, W_{T-2}) \tag{1.6}$$

Benefits of CI Assumptions

in effect, only substrings of length 3 need be considered, giving a larger number of observations. By introducing conditional independencies, a more compact representation of the distribution can be formed and more efficienct inference can be performed.

Graphical Models

Bayesian Networks

A *Bayesian network*, $\mathcal{G}$, is a directed acyclic graph whose nodes are random variables. Denote the parents of node X in $\mathcal{G}$ as $\text{PA}_X^{\mathcal{G}}$ and the non-descendants of X as $\text{ND}_X^{\mathcal{G}}$. Then,

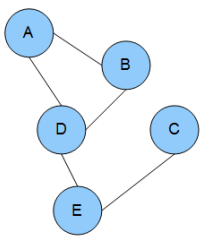
$$p(X_1, \ldots, X_d) = \prod_{i=1}^{d} p(X_i|\text{PA}_{X_i}^{\mathcal{G}}) \qquad X_i \perp\!\!\!\perp \text{ND}_{X_i}^{\mathcal{G}} | \text{PA}_{X_i}^{\mathcal{G}} \ \forall i \tag{1.7}$$

Efficient Inference

Note that if a node is shaded in the graph, it is observed. The Bayesian network expresses the joint as a product of factors, each of which depend on a small number of variables. Some expressions need only be calculated once and thus, large gains can be made by caching intermediate results.

Markov Networks
Clique

Choosing a direction for the edges can be awkward e.g. in cases where the factors of the joint pdf are symmetric. A *Markov Network* is an undirected graph $\mathcal{G}$ where each node, $\{X_1, \ldots, X_n\}$ is a random variable. Define a *clique* as a fully connected subset of nodes. There is a **positive** potential function for each clique of $\mathcal{G}$ i.e. $\phi_i(\mathcal{C}_i)$. Then



$$P(X_1,\ldots,X_n) = \frac{1}{Z}\prod_{i=1}^{|\mathcal{C}|}\phi_i(\mathcal{C}_i) \tag{1.8}$$

where Z is the normalisation constant

$$Z = \int \prod_{i=1}^{|\mathcal{C}|}\phi_i(\mathcal{C}_i)\; d\{X_j\}_{j=1}^{n} \tag{1.9}$$

Conditional Independencies

$\mathcal{G}$ encodes the conditional indepenencies $A \perp\!\!\!\perp B|C$ for any **sets** of nodes A, B and C such that C seperates A from B. For the example shown, $(A,B) \perp\!\!\!\perp (E,C)|D$. Whilst Bayesian Networks capture causal relationships, Markov Networks capture correlations between variables since the factors are symmetric. This makes sense in certain scenarios, for example, image modelling.

1.2 Latent Variable and Sequence Models

Latent variables are variables which are not observed. In graphical models, such models are represented by shading. Note that here, circular nodes represent continuous variables and square/rectangular nodes represent discrete variables.

Gaussian Mixture Models

In *Gaussian Mixture Models*, each component is modelled using a Gaussian distribution.

$$p(\mathbf{x}) = \sum_{m=1}^{M} p(c_m)p(\mathbf{x}|c_m) = \sum_{m=1}^{M} p(c_m)\,\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m) \tag{1.1}$$

$p(c_m)$ is the component prior. This is a flexible model, able to model a range of distributions.

Factor Analysis

Factor Analysis can be viewed as performing dimensionality reduction. The latent variable, $\mathbf{z}$, has lower dimensionality than the observed variable, $\mathbf{x}$.

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|\mathbf{0},\mathbf{I}) \tag{1.2}$$

$$p(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mathbf{x}|\mathbf{Cz},\boldsymbol{\Sigma}) \tag{1.3}$$

$\mathbf{C}$ is known as the loading matrix and represents the linear transformation between the latent and observed variable. If the covariance matrix is a scaled version of the identity matrix, this model is probabilistic PCA. Note that since everything is Gaussian, there is a close form expression for the likelihood.

Expectation Maximisation

Given data $\mathcal{D} = \{\mathbf{x}_i\}_{i=1}^{N}$ and some generative model with parameters $\boldsymbol{\theta}$, we wish to train the model. Calculating the maximum likelihood solution directly is intractable because it is difficult to marginalise over the latents. Whilst gradient descent could be used, this is not a parameter free algorithm. An alternative is the *Expectation Maximisation* algorithm which is now derived.

EM - Derivation via Jensen

$$\begin{aligned}
\mathcal{L}(\boldsymbol{\theta}^{k+1}) - \mathcal{L}(\boldsymbol{\theta}^{k}) &= \log \frac{p(\mathbf{X}|\boldsymbol{\theta}^{k+1})}{p(\mathbf{X}|\boldsymbol{\theta}^{k})} \\
&= \log \frac{\int p(\mathbf{X},\mathbf{Z}|\boldsymbol{\theta}^{k+1})\; d\mathbf{Z}}{p(\mathbf{X}|\boldsymbol{\theta}^{k})} && \text{(Marginalisation)} \\
&= \log \int \frac{p(\mathbf{Z}|\mathbf{X},\boldsymbol{\theta}^{k})\; p(\mathbf{X},\mathbf{Z}|\boldsymbol{\theta}^{k+1})}{p(\mathbf{Z}|\mathbf{X},\boldsymbol{\theta}^{k})\; p(\mathbf{X}|\boldsymbol{\theta}^{k})}\; d\mathbf{Z} \\
&\geq \int p(\mathbf{Z}|\mathbf{X},\boldsymbol{\theta}^{k}) \log \frac{p(\mathbf{X},\mathbf{Z}|\boldsymbol{\theta}^{k+1})}{p(\mathbf{X},\mathbf{Z}|\boldsymbol{\theta}^{k})} d\mathbf{Z} && \text{(Jensen's Inequality)} \\
&= \mathcal{Q}(\boldsymbol{\theta}^{k},\boldsymbol{\theta}^{k+1}) - \mathcal{Q}(\boldsymbol{\theta}^{k},\boldsymbol{\theta}^{k})
\end{aligned}$$

where *Auxilary Function*

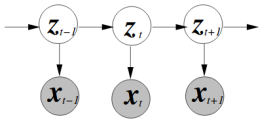
$$\mathcal{Q}(\boldsymbol{\theta}_i, \boldsymbol{\theta}_j) = \int p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}_i) \log p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}_j) d\mathbf{Z} \tag{1.4}$$

$\mathcal{Q}$ is known as the *Auxiliary Function*. Optimisating the log-likelihood requires an increase in the auxiliary function; the change of the auxiliary function acts as a lower bound for the change of the log likelihood. This forms the expectation maximisation algorithm and it can be seen by maximising the auxiliary function, the log-likelihood is **guaranteed to increase** at every iteration. Note: we have formed a parameter free algorithm but because the log-likelihood must always increase, the algorithm is very sensitive to the initial guess of the parameters. The algorithm also gets stuck in local optima.

Hidden Markov Models

Discrete Kalman Filters

For *Discrete Kalman Filter*, the latent and observed variables are continuous with a linear relationship between the variables.



Discrete Kalman Filter Graphical Model

$$\mathbf{z}_t = \mathbf{A}\mathbf{z}_{t-1} + \boldsymbol{\nu}_t \qquad \boldsymbol{\nu}_t \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma}_\nu) \tag{1.5}$$

$$\mathbf{x}_t = \mathbf{C}\mathbf{z}_t + \boldsymbol{\epsilon}_t \qquad \boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma}_\epsilon) \tag{1.6}$$

As usual, the linear Gaussian equations give rise to closed form solutions.

Hidden Markov Models

The *Hidden Markov Model* has discrete latent variables. There are two types of state; emitting states which produce the observation sequence and non-emitting states which define valid start and end states. These models have the following parameters:

1. N, the number of discrete hidden states. It is assumed that s_1 and s_N are non-emitting.
2. $\mathbf{A}$, the transition matrix, defined as $[A]_{ij} = p(q_t = s_j | q_{t-1} = s_i)$.
3. $\{b_2(\mathbf{x}_t), \ldots, b_{N-1}(\mathbf{x}_t)\}$ the set of state output distributions. $b_j(\mathbf{x}_t) = p(\mathbf{x}_t | q_t = s_j)$

Expression for HMM Likelihood

The likelihood of the data is written

$$\begin{aligned} p(\{\mathbf{x}_i\}_{i=1}^T) &= \sum_{\mathbf{q} \in \mathbf{Q}_T} p(\mathbf{q})\, p(\{\mathbf{x}_i\}_{i=1}^T | \mathbf{q}) \\ &= \sum_{\mathbf{q} \in \mathbf{Q}_T} p(q_0) \prod_{t=1}^{T} p(q_t | q_{t-1})\, p(\mathbf{x}_t | q_t) \end{aligned} \tag{1.7}$$

Viterbi Technique

where $\mathbf{Q}_T$ is the set of all possible hidden state sequences for T observations. An important technique for HMM models is to approximate the likelihood using a lower bound from the best state sequence through the hidden discrete space.

$$p(\{\mathbf{x}_i\}_{i=1}^T) \simeq p(\{\mathbf{x}_i\}_{i=1}^T, \hat{\mathbf{q}})$$

where

$$\hat{\mathbf{q}} = \arg\max_{\mathbf{q} \in \mathbf{Q}_T} p(\{\mathbf{x}_i\}_{i=1}^T, \mathbf{q}) \tag{1.8}$$

Viterbi Algorithm

The *Viterbi Algortihm* is a dynamic programming algorithm which allows us to compute the most probable path of hidden states which produced some output string. Consider a HMM with states s_j, output distributions $\{b_j(\mathbf{x}_t)\}_j$ and transition matrix $\mathbf{A}$ where $j \in \{1, \ldots, N\}$. Denote the most probable path to state s_j at time t as $\phi_j(t)$. Then the following recursion hold

Viterbi: Recursion

$$\phi_j(t) = b_j(\mathbf{x}_t) \cdot \max_{k \in \{1,\ldots,N\}} \phi_k(t-1) \cdot A_{kj} \tag{1.9}$$

The algorithm them constructs these probabilities forwards with the following initialisation.

$$\phi_1(0) = 1 \qquad \phi_j(0) = 0 \ \forall j \in \{2, \ldots, N-1\}$$

i.e. Eq. 1.9 is apply over increasing t for every single state. Finally, the maximum probability can be found as follows

$$p(\{\mathbf{x}_i\}_{i=1}^T|\hat{\mathbf{q}}) = \max_{k\in\{1,\ldots,N\}} \phi_k(T)\cdot A_{kN} \tag{1.10}$$

An extra transition probability is included at the end to transition into the final non-emitting state. When this algorithm is applied, there are often numerical issues with probabilities being very close to zero. In order to mitigate this, the logarithm of the probability may be taken with a special character used in placed of $\log 0$. Note that this technique can be modified in order to yield the best sequence simply by encoding previous state information in each ϕ.

Algorithm Modification

HMM probabilities have useful recursion properties. The *Forwards Probability* is

HMM Forwards Probability

$$\begin{aligned}\hat{\alpha}_j(t) &\triangleq p(\{\mathbf{x}_i\}_{i=1}^t, q_t = s_j) &(1.11)\\ &= \sum_{\hat{j}} \underbrace{p(\{\mathbf{x}_i\}_{i=1}^{t-1}, q_{t-1} = s_{\hat{j}})}_{\hat{\alpha}_{\hat{j}}(t-1)} p(q_t = j|q_{t-1} = s_{\hat{j}}) p(\mathbf{x}_t|q_t = s_j) &(1.12)\end{aligned}$$

note that usually logs are taken. Similarly, the *Backwards Probability* is

HMM Backwards Probability

$$\begin{aligned}\hat{\beta}_j(t) &\triangleq p(\{\mathbf{x}_i\}_{i=t+1}^T|q_t = s_j) &(1.13)\\ &= \sum_{\hat{j}} \underbrace{p(\{\mathbf{x}_i\}_{i=t+2}^T|q_{t+1} = s_{\hat{j}})}_{\hat{\beta}_{\hat{j}}(t+1)} p(\mathbf{x}_{t+1}|q_{t+1} = s_{\hat{j}}) p(q_{t+1} = s_{\hat{j}}|q_t = s_j) &(1.14)\end{aligned}$$

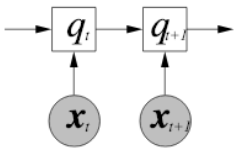
Note the asymmetry; the backwards probability is **given** the state at time t. The state posterior (as required for EM) can be written as follows

Rewriting the State Posterior

$$\begin{aligned}p(q_t = s_j|\{\mathbf{x}_i\}_{i=1}^T) &= p(q_t = s_j|\{\mathbf{x}_i\}_{i=1}^t, \{\mathbf{x}_i\}_{i=t+1}^T)\\ &\propto p(q_t = s_j|\{\mathbf{x}_i\}_{i=1}^t) p(\{\mathbf{x}_i\}_{i=t+1}^T|q_t = s_j, \cancel{\{\mathbf{x}_i\}_{i=1}^t})\\ &\propto p(q_t = s_j, \{\mathbf{x}_i\}_{i=1}^t) p(\{\mathbf{x}_i\}_{i=t+1}^T|q_t = s_j)\\ &\propto \hat{\alpha}_j(t)\cdot\hat{\beta}_j(t)\end{aligned}$$

This gives the *Forwards-Backwards Algorithm*.. In the first path, the forward probabilities are evaluated and the recursion is applied forwards in term. In the second pass, the backward probabilities are propagated through the HMM. These two sets of probability distributions are combined to obtain the state distribution posterior, as required for the EM algorithm.

Forwards-Backwards Algorithm



Maximum Entropy Markov Model

Discriminative Sequence Models

Whilst HMMs are generative models, there are discriminative sequence models. The diagram on the right shows a *Maximum Entropy Markov Model*, described by the following equation.

$$p(q_0, \ldots, q_T|\{\mathbf{x}_i\}_{i=1}^T) = \prod_{t=1}^T p(q_t|q_{t-1}, \mathbf{x_t}) \tag{1.15}$$

the product is of terms of the following form

State Posterior

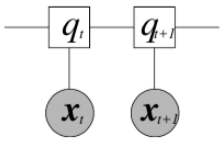
$$p(q_t|q_{t-1}, \mathbf{x_t}) = \frac{1}{\mathcal{Z}_t} \exp\left[\sum_{j=1}^D \lambda_i f_i(q_t, q_{t-1}, \mathbf{x}_t)\right] \tag{1.16}$$

where each $f_i(\cdot)$ corresponds to some *feature* and λ_i represents the weighting of that feature. Note that the above expression corresponds to a weighted soft-max. The above expression can be extended to the complete sequence i.e.

$$p(q_0,\ldots,q_T|\{\mathbf{x}_i\}_{i=1}^T) = \frac{1}{\mathcal{Z}}\exp\left[\sum_{j=1}^{D}\lambda_i f_i(q_0,\ldots,q_T|\{\mathbf{x}_i\}_{i=1}^T)\right]$$

An Obvious Problem...

Immediately, the effectiveness of such models depends on the features which are extracted; there are a vast number of possible features. Note that features must be able to handle variations in sequence length and the number of model parameters, $\boldsymbol{\lambda}$, should not be too high.

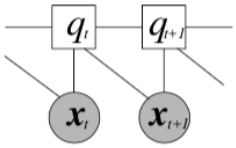


Simple Linear Chain CRF

Feature Extraction Techniques

Feature extraction can be performed using conditional random fields e.g. a simple linear chain conditional random field. This gives the following posterior model

$$p(q_0,\ldots,q_T|\{\mathbf{x}_i\}_{i=1}^T) = \frac{1}{\mathcal{Z}}\exp\left[\sum_{t=1}^{T}\left(\sum_{i=1}^{D_t}\lambda_i^t f_i^t(q_t,q_{t-1}) + \sum_{i=1}^{D_a}\lambda_i^a f_i^a(q_t,\mathbf{x}_t)\right)\right] \tag{1.17}$$



Linear Chain CRF

where $\boldsymbol{\lambda}^t$ represents the parameters for the *transition style* features and $\boldsymbol{\lambda}^a$ represents the parameters for the *acoustic style* features.

Alternatively, a more complex linear chain model can be used, as seen on the side. This corresponds to the following posterior

$$p(q_0,\ldots,q_T|\{\mathbf{x}_i\}_{i=1}^T) = \frac{1}{\mathcal{Z}}\exp\left[\sum_{t=1}^{T}\left(\sum_{i=1}^{D}\lambda_i f_i(q_t,q_{t-1},\mathbf{x}_t)\right)\right] \tag{1.18}$$

Normalisation Term

these sorts of model tend to give more complex and interesting features than HMMs but only have global normalisation properties. It is important to be able to compute the normalisation term effectively; this is computed using an algorithm equivalent to the forward-backward algorithm.

For a general CRF, the posterior probability is written as follows where $\mathcal{C}$ represents the set of cliques in the graph **which are repeated at each timestep**.

$$p(q_0,\ldots,q_T|\{\mathbf{x}_i\}_{i=1}^T) = \frac{1}{\mathcal{Z}}\exp\left[\sum_{t=1}^{T}\left(\sum_{c\in\mathcal{C}}\boldsymbol{\lambda}_c^T\mathbf{f}_c(c_t,t)\right)\right] \tag{1.19}$$

where $\boldsymbol{\lambda}_c^T$ are time-independent parameters associated with each repeating clique c and $\mathbf{f}_c(c_t,t)$ is a vector of time-dependent features. Note that c_t denotes the variables of the 'repeated clique' c at time index t and each c is the repeating clique itself.

Training CRFs

Training for conditional random fields is usually fully observed i.e. the latent label sequence is known. A maximum likelihood approach is used where the parameters $\boldsymbol{\lambda}$ are chosen to maximise $p(y_1,\ldots,y_T|\{\mathbf{x}_i\}_{i=1}^T)$ where y_i represents the true (or training) label. Thus a number of different features are posed, and those with large coefficients are chosen as the features to use.

1.3 Deep Learning

Preliminaries

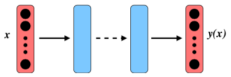
Deep Learning

Deep Learning is a branch of machine learning based on a set of algorithms that model data by using multiple processing layers with complex structures.

Deep Neural Networks

A *Deep Neural Network* maps an input $\mathbf{x}$ to output $\mathbf{y}$ through some non-linear function.

$$\mathbf{y}(\mathbf{x}) = \mathcal{F}(\mathbf{x})$$



Deep Neural Network Schematic

These networks are composed of a number of hidden layers; deep refers to the number of hidden layers. Denoting $\mathbf{x}^{(k)}$ as the input to layer k and noting that $\mathbf{x}^{(k+1)} = \mathbf{y}^{(k)}$ (i.e. the output of layer k), the general form of each layer is

$$y_i^{(k)} = \phi((\mathbf{w}_i^{(k)})^T \mathbf{x}^{(k)} + b_i^{(k)}) \tag{1.1}$$

where $\phi(\cdot)$ is some non-linear function.

Designing Neural Networks

In general, the inputs and outputs of the network are problem specific but the number of hidden layers and the number of nodes per hidden layer are free parameters. The generalisation of the network is heavily dependent on the number of parameters, N. Neglecting the bias term, the number of parameters can be written as

Number of Parameters

$$N = d \cdot N^{(1)} + K \cdot N^{(L)} + \sum_{k=1}^{L-1} N^{(k)} \cdot N^{(k+1)} \tag{1.2}$$

where L is the number of hidden layers, $N^{(k)}$ is the number of hidden nodes for layer k, d is the input vector size and K is the output vector size. The form of the summation is justified as each node in the $(k+1)$-th layer has $N^{(k)}$ parameters (as that is the input vector dimension).

There are a number of possible *activation functions*, $\phi(\cdot)$, which can be used. They must be non-linear to ensure that $\mathcal{F}(\cdot)$ is a non-linear function since the linear combination of linear functions is also linear.

Typical Activation Functions

- Heaviside step function.

$$\phi(z_i) = \begin{cases} 0 & z_i < 0 \\ 1 & z_i \geq 0 \end{cases} \tag{1.3}$$

- Sigmoid; $\phi(z) \in [0, 1]$

$$\phi(z) = \frac{1}{1 + \exp(-z)} \tag{1.4}$$

  This is often used for hidden layers.

- Softmax; $\phi_i(x) \in [0, 1]$ and $\sum_i \phi(z_i) = 1$.

$$\phi(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)} \tag{1.5}$$

  This gives a valid discrete probability distribution and is thus the typical layer for classification tasks.

- tanh; $\phi_i(x) \in [-1, 1]$

$$\phi(z_i) = \frac{\exp(z_i) - \exp(-z_i)}{\exp(z_i) + \exp(-z_i)} \tag{1.6}$$

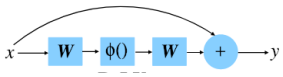
- *Rectified Linear Units (ReLU)*

ReLU

$$\phi(z_i) = \max(0, z_i) \tag{1.7}$$

  Empirically, these are found to converge rapidly when training. Lack of division makes ReLUs efficient. There are many variations on this theme, including noisy ReLUs: $\phi(z_i) = \max(0, z_i + \epsilon)$; $\epsilon \sim \mathcal{N}(0, \sigma^2)$ and leaky ReLus

$$\phi(z_i) = \begin{cases} z_i & z_i \geq 0 \\ \alpha z_i & z_i < 0 \end{cases} \tag{1.8}$$



Residual Networks

Residual Networks modify each layer to model the residual i.e.

$$\mathbf{y}(x) = \mathcal{F}(\mathbf{x}) + \mathbf{x}$$

This allows deeper networks to be built since layers which do not contribute can be skipped e.g. by setting the weight matrix to be $\mathbf{0}$, improving generalisation. This also aids the network by making feature recall easy.

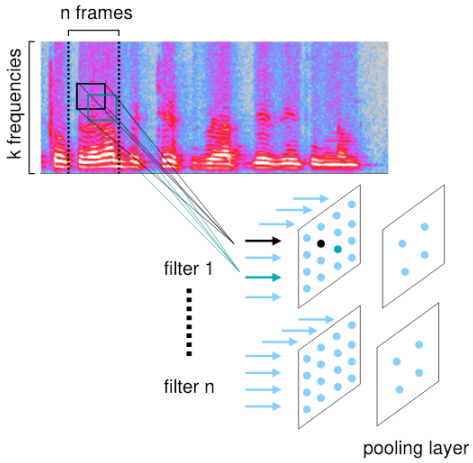
Pooling

Pooling reduces the number of parameters in a network by 'pooling' a set of nodes into a single node. A number of pooling methods can be used, including:

- Maxout: $\phi(\{y_i\}_{i=1}^k) = \max_k\{y_i\}_{i=1}^k$
- Soft-Maxout: $\phi(\{y_i\}_{i=1}^k) = \log(\sum_i \exp(y_i))$
- p-norm: $\phi(\{y_i\}_{i=1}^k) = ||\mathbf{y}||_p$

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are commonly applied to analysing image data. Convolutional layers are repeatedly applied to the input data, effectively filtering the input image. Pooling layers are also employed to reduce the number of parameters; typically, (both overlapping and non-overlapping) max-pooling is used. A CNN would typically consist of a number of convolution layers (which includes a non-linearity) each followed by a pooling layer, followed by some output activation function e.g. a soft-max.



For a 2D image, a layer can be written as

$$\phi(z_{ij}) = \phi\left[\sum_{k,l} w_{kl} x_{(i-k)(i-l)}\right] \tag{1.9}$$

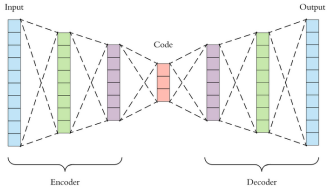
Design Choices

where w_{kl} represents the 'weights' which control which filter operation is performed and defines an image filter. A number of parameters control the form of CNN:

- Depth: how many filters to apply.
- Receptive Field: size of the filter applied i.e. height/width/depth.
- Stride: spacing between filters i.e. how i and j vary in Eq. 1.9
- Dilation: 'gaps' between filter elements. This allows a larger receptive field without an increase in parameters.
- Zero-Padding: edges of images could be padded with zeroes.

An *Autoencoder* is a type of feed-forward network which is trained to encode an input into a low dimension code layer and decode the low dimensional representation, reconstructing the original input. This can be used to de-noise data as well as perform non-linear feature extraction. The error function penalises the reconstruction error:

Autoencoder



$$E(\boldsymbol{\theta}) = \sum_{p=1}^{n} f(\mathbf{x}_p, \hat{\mathbf{x}_p}) \tag{1.10}$$

Training & Back-Propagation

For classification tasks, class labels are *one-hot* encoded into vectors. Denote the true class of data point i as $y_i \in \{\omega_1, \ldots, \omega_k\}$. For each datapoint, there is a target vector, $\mathbf{t}_i$, meaning that $\mathcal{D} = \{(\mathbf{x}_1, \mathbf{t}_1), \ldots, (\mathbf{x}_N, \mathbf{t}_N)\}$.

One-Hot Encoding

$$t_{ij} = \begin{cases} 1 & y_i = \omega_j \\ 0 & \text{otherwise} \end{cases} \tag{1.11}$$

Two typical training criteria for such tasks are:

Training Criteria

- Least squares error

$$E(\boldsymbol{\theta}) = \frac{1}{2} \sum_{i=1}^{N} ||\mathcal{F}(\mathbf{x}_i) - \mathbf{t}_i||_2^2 \tag{1.12}$$

  This training criteria can be applied directly to regression problems where $\mathbf{y}_i$ takes the place of $\mathbf{t}_i$. Minimisation of the least-squares error corresponds to the maximum-likelihood solution where the likelihood is Gaussian with a scaled identity covariance matrix.

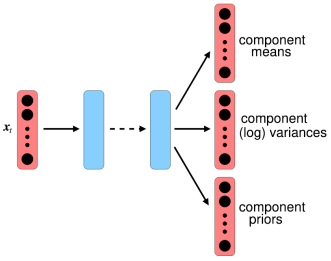
- Cross-Entropy:

$$E(\boldsymbol{\theta}) = -\sum_{i=1}^{N} \sum_{j=1}^{K} t_{ij} \log(\mathcal{F}(\mathbf{x}_i)_j) \tag{1.13}$$

  Recalling the one-hot encoding employed, the cross-entropy error maximises the average output value for each class. Note that this error is not strictly positive.

Note that the output layer activation function must be chosen to be appropriate e.g. when attempting to predict a non-negative quantity, the activation function must ensure that this is indeed the case. For example, when predicting the variance of a random variable, the activation layer could be exponential.

Mixture Density Neural Networks are a mix between probabilistic models and neural networks. A mixture of M Gaussians is predicted with a neural network used for the prior mixture proportions, the mixture means i.e. the mixture components. For each component m, the neural network outputs



Mixture Density Neural Networks

$$\mathbf{y}_m(\mathbf{x}_p) = \begin{bmatrix} \mathcal{F}_m^c(\mathbf{x}_p) \\ \mathcal{F}_m^\mu(\mathbf{x}_p) \\ \mathcal{F}_m^\sigma(\mathbf{x}_p) \end{bmatrix} \tag{1.14}$$

The model is then optimised using maximum likelihood. Note: this is a general technique for performing regression with an arbitrary output density for each input data-point.

$$p(\mathbf{y}_p|\mathbf{x}_p, \boldsymbol{\theta}) = \sum_{m=1}^{M} \mathcal{F}_m^c(\mathbf{x}_p) \mathcal{N}(\mathbf{y}_p|\mathcal{F}_m^\mu(\mathbf{x}_p), \mathcal{F}_m^\sigma(\mathbf{x}_p)) \tag{1.15}$$

Gradient Descent is a simplistic gradient descent algorithm used for optimisation if there

Gradient Descent

is no closed form solution.

$$\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \eta \frac{\partial E}{\partial \boldsymbol{\theta}}\bigg|_{\boldsymbol{\theta}^{(i)}} \tag{1.16}$$

η is known as the *learning rate* and is a free parameter which can be difficult to choose. It is desired to avoid local minima. Note that the gradient of the error function is required.

Single Layer Perceptron Training

For a single layer perceptron with some error function $E(\boldsymbol{\theta})$, the chain rule is used to compute derivatives throughout the network as follows

$$\frac{\partial E(\boldsymbol{\theta})}{\partial w_i} = \frac{\partial E(\boldsymbol{\theta})}{\partial y(\mathbf{x})} \frac{\partial y(\mathbf{x})}{\partial z} \frac{\partial z}{\partial w_i} \tag{1.17}$$

The gradient is composed of terms depending on the cost function (change of error with the network output), the activation function (change of network output with activation function inputs) and the network weights (change of activation function inputs with input). Note that this requires that the activation function is smooth and can be differentiated.

Deriving Back-Propagation - Notation

Consider a multi-layer perceptron with d-dimensional input data, L hidden layers (and so $L+1$ layers in total), $N^{(k)}$ nodes in level k and K dimensional output. Denote the input to the kth layer as $\mathbf{x}^{(k)}$ and the 'extended input' as $\tilde{\mathbf{x}}^{(k)} = \begin{bmatrix}\mathbf{x}^{(k)} & 1\end{bmatrix}^T$. $\mathbf{W}^{(k)}$ is the weight matrix of the kth layer and thus has dimension $N^{(k)} \times N^{(k+1)}$ by definition. Each row of the weight matrix defines a node in layer k. Similarly, the weight matrix can be extended to include the bias term $\tilde{\mathbf{W}}^{(k)} = \begin{bmatrix}\mathbf{W}^{(k)} & \mathbf{b}^{(k)}\end{bmatrix}$ where $\mathbf{b}^{(k)}$ is the $N^{(k)}$ length column vector of node biases. The target values for network training are denoted at $\mathbf{t}$. Then, the equations governing the behaviour of the multi-layer perception are

Extended Input

$$\mathbf{x}^{(1)} = \mathbf{x} \tag{1.18}$$

$$\mathbf{x}^{(k)} = \mathbf{y}^{(k-1)} \text{ for } k = \{2, \ldots, (L+1)\} \tag{1.19}$$

$$\mathbf{z}^{(k)} = \tilde{\mathbf{W}}^{(k)} \tilde{\mathbf{x}}^{(k)} \tag{1.20}$$

$$\mathbf{y}^{(k)} = \phi(\mathbf{z}^{(k)}) \tag{1.21}$$

$$\mathbf{y}(\mathbf{x}) = \mathbf{y}^{(L+1)} \tag{1.22}$$

Deriving Back Propagation

We need to calculate $\frac{\partial E}{\partial \tilde{W}_{ij}^{(k)}}$ for all layers. The chain rule gives

$$\frac{\partial E}{\partial \tilde{W}_{ij}^{(k)}} = \underbrace{\frac{\partial E}{\partial z_i^{(k)}}}_{\delta_i^{(k)}} \underbrace{\frac{\partial z_i^{(k)}}{\partial \tilde{w}_{ij}^{(k)}}}_{\tilde{x}_j^{(k)}} \tag{1.23}$$

Evaluating δ for hidden layers

For the output node, the evaluation of δ_i is straightforward (the same as the single layer perceptron). For hidden layers,

$$\begin{aligned}\delta_i^{(k)} &= \sum_m \frac{\partial E}{\partial z_m^{(k+1)}} \frac{\partial z_m^{(k+1)}}{\partial z_i^{(k)}} \\ &= \sum_m \frac{\partial E}{\partial z_m^{(k+1)}} \frac{\partial y_i^{(k)}}{\partial z_i^{(k)}} \frac{\partial z_m^{(k+1)}}{\partial y_i^{(k)}}\end{aligned} \tag{1.24}$$

but $z_m^{(k+1)} = \sum_j \tilde{W}_{mj}^{(k)} y_j^{(k)}$ and $y_j^{(k)} = \phi(z_j^{(k)})$ (see above). Therefore, **for the sigmoidal activation function**

$$\delta_i^{(k)} = y_i^{(k)}(1 - y_i^{(k)}) \sum_m \delta_m^{(k+1)} \tilde{W}_{mi}^{(k)} \tag{1.25}$$

In matrix notation, *Matrix Formulation*

$$\frac{\partial E}{\partial \tilde{\mathbf{W}}^{(k)}} = \boldsymbol{\delta}^k (\tilde{\mathbf{x}}^{(k)})^T \tag{1.26}$$

Define the activation derivative matrix for layer k as

$$\boldsymbol{\Lambda}^{(k)} = \begin{bmatrix} \frac{\partial y_1^{(k)}}{\partial z_1^{(k)}} & \cdots & \frac{\partial y_{N^{(k)}}^{(k)}}{\partial z_1^{(k)}} \\ \vdots & \cdots & \vdots \\ \frac{\partial y_1^{(k)}}{\partial z_{N^{(k)}}^{(k)}} & \cdots & \frac{\partial y_{N^{(k)}}^{(k)}}{\partial z_{N^{(k)}}^{(k)}} \end{bmatrix} \tag{1.27}$$

Note that this is diagonal for most layers. Eq. 1.24 can be modified to include non-diagonal terms of the activation derivative matrix as follows

$$\begin{aligned} \delta_i^{(k)} &= \sum_m \underbrace{\frac{\partial E}{\partial z_m^{(k+1)}}}_{\delta_m^{(k+1)}} \sum_j \underbrace{\frac{\partial y_j^{(k)}}{\partial z_i^{(k)}}}_{\Lambda_{ij}^{(k)}} \underbrace{\frac{\partial z_m^{(k+1)}}{\partial y_j^{(k)}}}_{\tilde{W}_{mj}^{(k+1)}} \\ &= \sum_{m,j} \Lambda_{ij}^{(k)} \tilde{W}_{mj}^{(k+1)} \delta_m^{(k+1)} \qquad (1.28) \\ \therefore\ \boldsymbol{\delta}^{(k)} &= \boldsymbol{\Lambda}^{(k)} (\mathbf{W}^{(k+1)})^T \boldsymbol{\delta}^{(k)+1} \qquad (1.29) \end{aligned}$$

Thus a matrix form for the back-propagation algorithm has been found. The initialisation of the backward recursion is from the output player. *Back-Propagation - Initialisation*

$$\begin{aligned} \boldsymbol{\delta}^{(L+1)} &= \frac{\partial E}{\partial \mathbf{z}^{(L+1)}} \\ &= \boldsymbol{\Lambda}^{(L+1)} \frac{\partial E}{\partial \mathbf{y}(\mathbf{x})} \end{aligned} \tag{1.30}$$

Thus, the process to get the derivative involves propagating forwards $\mathbf{y}^k$ and $\mathbf{z}^{(k)}$, computing the gradient at the output layer and then propagating back $\boldsymbol{\delta}$, using this value to compute the desired derivative.

Optimisation

The gradient is calculated over all training samples; for large datasets, this is very slow. A simple modification is to use batch gradient descent where only a subset of the data is used to calculate the gradient. There is a compromise involved with choosing the size of the subset used as if too small, the gradient estimate is poor but the computation cost rises with subset size. A modification is made to this algorithm; the order of the data presented for training is randomised and mini-batch updates (with random subsets) are used. *Stochastic Gradient Descent*

These algorithms have issues with stopping at local minima and handling 'ravines'. A method of resolving this is by introducing *Momentum*: *Gradient Descent: Issues & Solutions*

$$\Delta \boldsymbol{\theta}^{(k)} = \eta \underbrace{\frac{\partial E}{\partial \boldsymbol{\theta}}\bigg|_{\boldsymbol{\theta}^{(k)}}}_{\nabla_{\boldsymbol{\theta}} E} + \alpha \Delta \boldsymbol{\theta}^{(k-1)} \tag{1.31}$$

This serves to smooth parameter changes over iterations. Additionally, the learning rate can be modified to be a function of the iteration, k. A simple approach is to have a larger learning rate when the previous iteration decreases the cost function.

Gradient descent only makes use of the first order derivative. A multi-dimensional *Second-Order Approximation*

Taylor expansion reads

$$E(\boldsymbol{\theta}) = E(\boldsymbol{\theta}^{(k)}) + (\boldsymbol{\theta} - \boldsymbol{\theta}^{(k)}) \frac{\partial E}{\partial \boldsymbol{\theta}}\bigg|_{\boldsymbol{\theta}^{(k)}} + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}^{(k)})^T \mathbf{H}_{\boldsymbol{\theta}^{(k)}} (\boldsymbol{\theta} - \boldsymbol{\theta}^{(k)}) + \mathcal{O}(\boldsymbol{\theta}^3) \tag{1.32}$$

where $\mathbf{H}_{\boldsymbol{\theta}^{(k)}}$ is the Hessian matrix of second derivatives. Taking derivatives and ignoring high order second order terms (valid when the error function is locally well approximated by a quadratic) gives

$$\nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta}) \simeq \frac{\partial E}{\partial \boldsymbol{\theta}}\bigg|_{\boldsymbol{\theta}^{(k)}} + \mathbf{H}_{\boldsymbol{\theta}^{(k)}} (\boldsymbol{\theta} - \boldsymbol{\theta}^{(k)}) \tag{1.33}$$

Equating this to zero gives

$$\Delta \boldsymbol{\theta}^{(k)} = -\mathbf{H}^{-1}_{\boldsymbol{\theta}^{(k)}} \frac{\partial E}{\partial \boldsymbol{\theta}}\bigg|_{\boldsymbol{\theta}^{(k)}} \tag{1.34}$$

Problems with Second-Order Approaches

Note that care must be taken to ensure that a minimum in the cost function is found (i.e. the Hessian matrix is positive semi-definite). Whilst in theory this works, there are a number of issues in practice:

1. The evaluation of the Hessian can be computationally challenging as $\mathcal{O}(N^2)$ parameters must be accumulated for each sample.
2. At each iteration, the Hessian must be inverted - this is an $\mathcal{O}(N^3)$ operation.
3. There is no guarantee that the step leads to a minimum; a maximum or saddle point could be reached instead.
4. If the surface is poorly locally approximated by a quadratic, the step sizes may be too large and the optimisation technique will be unstable.

QuickProp

A technique known as *QuickProp* assumes that the error surface is quadratic in nature with a diagonal Hessian, allowing each weight to be treated independently. Denote the estimate for θ as time-step τ as θ^τ

$$E(\theta, \hat{\theta}) \simeq E(\hat{\theta}) + b(\theta - \hat{\theta}) + a(\theta - \hat{\theta})^2 \tag{1.35}$$

$$g^{\tau'} = \frac{\partial E(\theta, \hat{\theta}^\tau)}{\partial \theta}\bigg|_{\theta = \theta^{\tau'}} \simeq b + 2a(\theta^{\tau'} - \theta^\tau) \tag{1.36}$$

It is desired that the gradient be zero after the next update. Thus:

$$g^{\tau-1} \simeq b + 2a(\theta^{\tau-1} - \theta^\tau) = b - 2a\Delta\theta^{\tau-1} \tag{1.37}$$

$$g^\tau \simeq b \tag{1.38}$$

$$g^{\tau+1} \simeq b + 2a\Delta\theta^\tau \tag{1.39}$$

Rearranging and solving for $g^{\tau+1} = 0$ yields:

$$\Delta\theta^\tau = \frac{g^\tau \cdot \Delta\theta^{\tau-1}}{g^{\tau-1} - g^\tau} \tag{1.40}$$

Regularisation

Generalisation performance is essential. Regularisation techniques prevent *overfitting* which is when training data performance increases whilst generalisation performance decreases (i.e. the network fits to the noise in the dataset). *Regularisation* is used to address this. Forms of regularisation include

- Early stopping - do not train the network to convergence but rather stop early.

- Constraining parameter size (effectively placing a zero 'prior' on the network parameters) i.e. modify the cost function as follows

$$\tilde{E}(\boldsymbol{\theta}) = E(\boldsymbol{\theta}) + \nu \cdot \frac{1}{2} \sum_{l,i,j} (w_{ij}^{(l)})^2 \tag{1.41}$$

  this is incredibly simple to include in gradient descent optimisation.

- *Dropout*: a random proportion of network nodes are 'deactivated' (relevant terms are excluded from the back-propagation algorithm) and model parameters are updated during each step of the training process. **This prevents a single node specialisation to a task** and thus improves generalisation.

Dropout

Weight Initialisation

Data Preprocessing

Data features have different dynamic ranges. For example, one feature could be measured in μm whilst another is measured in km. This affects the relative 'importance' of features during the training process. Data whitening is often employed to prevent gradients from being dominated by one term only and ensures that each feature as the same mean and variance.

$$\tilde{x}_{ni} = \frac{x_{ni} - \mu_i}{\sigma_i} \tag{1.42}$$

$$\text{where } \mu_i = \sum_{n=1}^{N} \frac{x_{ni}}{N} \tag{1.43}$$

$$\sigma_i^2 = \frac{1}{N} \sum_{n=1}^{N} (x_{ni} - \mu_i)^2 \tag{1.44}$$

Network Initialisation: Weight Parameters

The gradient descent algorithm requires an initialisation for the weights. We need to worry about both vanishing gradients (gradients going to zero resulting in no parameter updates) and exploding gradients (derivatives getting very large and causing saturation). Whilst it is not possible to guarantee a good starting point, a Gaussian random initialisation passed through a sigmoid non-linearity can work well. **Note:** we need to be careful about the initialisation of the weights - if we choose them too large, passing data through the sigmoid will end up with saturated sigmoids. In regions where the sigmoid has saturated, the gradient will be small because the gradients of the activation functions themselves are small in these regions; this the variance should be chosen to avoid these small gradients.

Xavier Initialisation

To avoid the problems of vanishing and exploding gradients, the input and output variance should be the same. This leads to the *Xavier Initialisation* where $\text{var}(w_{ij}^{(l)}) = \frac{1}{N^{(l-1)}}$. This can be justified by assuming linear activation functions $\mathbf{y} = \mathbf{W}\mathbf{x}$, that all weights are independent and everything has zero mean. Then

$$\text{var}(y_i^k) = \text{var}\left(\sum_j W_{ij} x_j \right) = N^{(k-1)} \text{var}(w_{ij}) \text{var}(x_j) \tag{1.45}$$

Batch Normalisation

Batch Normalisation processes the output of each layer with a normalisation term that varies as the system trains in an attempt to ensure values remain in the linear region. The output of each layer is normalised by considering only a batch of the data meaning that each normalisation term is only a noisy estimate of the true term. At test time, the normalisation constant is found by considering taking the expectation over a large number of different batches (from the training set).

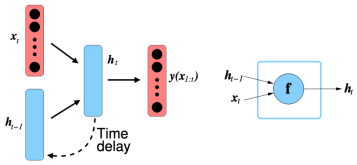
1.4 Deep Learning for Sequence Data

The previous deep learning techniques that we have seen have assumed that the input data is a fixed size. We will now focus on data which is of variable length, the key requirement being the sharing of model parameters to let us handle variable length inputs. There are a number of specific tasks that can be performed for such data, which will considered each in turn. Note that $\mathbf{x}_{1:T}$ denotes $\{\mathbf{x}_1, \ldots, \mathbf{x}_T\}$.

1.4.1 Sequence (Input, Output) Pairing

Sequence (input, output) pairing produces the sequence $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_T, \mathbf{y}_T)\}$ from the input sequence $\mathbf{x}_{1:T}$. An example task of this type is *part of speech tagging* where each word in the text sequence is tagged with a particular part of speech e.g. nouns, verbs, adverbs (though typically the tags are more advanced than this). This is a challenging problem as individual words are ambiguous and they function depends on their context within a sentence.

Example Task- PoS Tagging



Recurrent Neural Networks

Recurrent Neural Networks are a family of neural networks designed to process sequential data. Variable length sequences are dealt with by introducing an approximate *history vector*, $\mathbf{h}_t$, which models the history of observations.

$$\mathcal{F}(\mathbf{x}_{1:t}) = \mathcal{F}(\mathbf{x}_t, \mathbf{x}_{1:(t-1)}) \simeq \mathcal{F}(\mathbf{x}_t, \mathbf{h}_{t-1}) \simeq \mathcal{F}(\mathbf{h}_t) \tag{1.1}$$

The history vector is updated using a neural network.

$$\mathbf{h}_t = \mathcal{F}^{(h)}(\mathbf{W}_h^f \mathbf{x}_t + \mathbf{W}_h^r \mathbf{h}_{t-1} + \mathbf{b}_h) \tag{1.2}$$

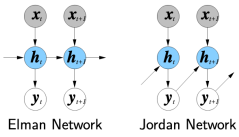
Note that the $\mathbf{W}$ matrices are simple linear transformations of the input and history vector at each point. Additionally, the output is determined by an additional neural network.

$$y(\mathbf{x}_{1:t}) = \mathcal{F}^{(y)}(\mathbf{W}_y \mathbf{h}_t + \mathbf{b}_y) \tag{1.3}$$

The network has causal memory encoding into the history vector. There are two main advantages of this structure:

1. Regardless of the sequence length, the learned model always has the same input size as it is specified in terms of the transition from state to state.
2. It is possible to use the same transition functions at every time step, corresponding to one shared set of weights.

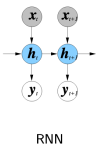
This means that a single model can be learnt which operates on all time-steps for variable sequence lengths.



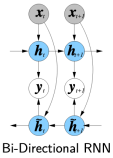
Elman Network

Jordan Network

This topology is known as the *Elman Network*. An alternative topology is known as the *Jordan Network* in which the history vector remembers the unobserved output sequence, $\mathbf{y}_{1:T}$.



RNN



Bi-Directional RNN

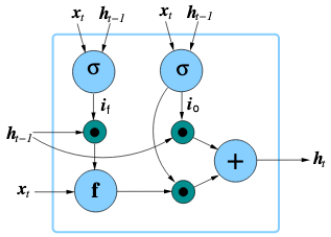
$$\mathcal{F}(\mathbf{x}_{1:t}, \mathbf{y}_{1:t-1}) \simeq \mathcal{F}(\mathbf{x}_t, \mathbf{y}_{1:t-1}) \simeq \mathcal{F}(\mathbf{x}_{1:t}, \mathbf{h}_{t-1}) \simeq \mathcal{F}(\mathbf{h}_t) \tag{1.4}$$

Alternatively, *Bi-Directional RNNs* may be used in which the complete observation sequence is used:

$$\mathcal{F}_t(\mathbf{x}_{1:T}) = \mathcal{F}_t(\mathbf{x}_{1:t}, \mathbf{x}_{t+1:T}) \simeq \mathcal{F}_t(\mathbf{h}_t, \tilde{\mathbf{h}}_t) \tag{1.5}$$

Network gating is a technique which creates paths through the network with derivatives that neither vanish nor explode.

The equations for the *Gated Recurrent Unit* are:



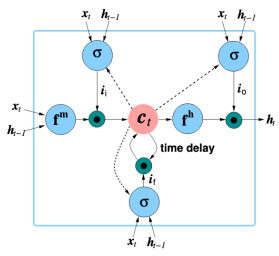
Gated Recurrent Unit

$$\mathbf{i}_f = \sigma(\mathbf{W}_f^f \mathbf{x}_t + \mathbf{W}_f^h \mathbf{h}_t + \mathbf{b}_f) \qquad \text{(forget gate)} \tag{1.6}$$

$$\mathbf{i}_o = \sigma(\mathbf{W}_o^f \mathbf{x}_t + \mathbf{W}_o^h \mathbf{h}_t + \mathbf{b}_o) \qquad \text{(output gate)} \tag{1.7}$$

$$\mathbf{h}_t = \mathbf{i}_0 \odot \mathbf{h}_{t-1} + (1 - \mathbf{i}_0) \odot \mathcal{F}\left(\mathbf{W}_y^f \mathbf{x}_\mathbf{t} + \mathbf{W}_y^h (\mathbf{i}_f \odot \mathbf{h}_{t-1}) + \mathbf{b}_y\right) \tag{1.8}$$

The forget and output gate can individually ignore parts of the history vector (given the state value). The output gate affects every dimension, allowing the new history to be purely the old history vector (at one extreme of the sigmoid) or replacing it with the new target state value. The forget gate controls which parts of the state are used to compute the next target state.



Long-Short Term Memory Units

Long-Short Term Memory Units (LSTMs) are more complex than GRUs. The equations underpinning the LSTM are:

$$\mathbf{i}_f = \sigma\left(\mathbf{W}_f^f \mathbf{x}_t + \mathbf{W}_f^h \mathbf{h}_{t-1} + \mathbf{W}_f^m \mathbf{c}_{t-1} + \mathbf{b}_f\right) \tag{1.9}$$

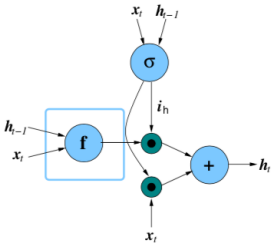
$$\mathbf{i}_i = \sigma\left(\mathbf{W}_f^i \mathbf{x}_t + \mathbf{W}_i^h \mathbf{h}_{t-1} + \mathbf{W}_i^m \mathbf{c}_{t-1} + \mathbf{b}_i\right) \tag{1.10}$$

$$\mathbf{i}_o = \sigma\left(\mathbf{W}_f^o \mathbf{x}_t + \mathbf{W}_o^h \mathbf{h}_{t-1} + \mathbf{W}_o^m \mathbf{c}_t + \mathbf{b}_o\right) \tag{1.11}$$

$$\mathbf{c}_t = \mathbf{i}_f \odot \mathbf{c}_{t-1} + \mathbf{i}_i \odot \mathcal{F}^m \left(\mathbf{W}_c^f \mathbf{x}_t + \mathbf{W}_c^h \mathbf{h}_{t-1} + \mathbf{b}_c\right) \tag{1.12}$$

$$\mathbf{h}_t = \mathbf{i}_o \odot \mathcal{F}^h(\mathbf{c}_t) \tag{1.13}$$

The LSTM includes a memory cell with diagonal weight matrices. The self-loop of the memory cell is controlled by a forget gate unit and the update is also dependent on an input gate unit. The output gate unit allows the output of the LSTM to be shut-off. The LSTM is able to learn long-term dependencies more easily than simply recurrent architectures.



Highway Connections

Highway Connections gate the output of the node and combine them with the output from the previous layer:

$$\mathbf{i}_h = \sigma(\mathbf{W}_i^f \mathbf{x}_t + \mathbf{W}_i^h \mathbf{h}_{t-1} + \mathbf{b}_i) \tag{1.14}$$

$$\mathbf{h}_t = \mathbf{i}_h \odot \mathcal{F}(\mathbf{W}_h^f \mathbf{x}_t + \mathbf{W}_h^h \mathbf{h}_{t-1} + \mathbf{b}_h) + (1 - \mathbf{i}_h) \odot \mathbf{x}_t \tag{1.15}$$

Note that the node could be anything e.g. an LSTM, a GRU, etc.

1.4.2 Input Sequence to Target

A typical example of this sort of task is sentiment analysis given a sequence of words.

A simple method of performing this sort of task would be averaging as follows:

Averaging Approach

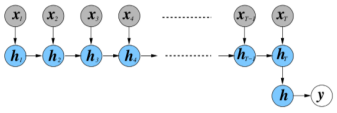
$$\mathbf{h}_t = \mathcal{F}_1(\mathbf{x}_t) \tag{1.1}$$

$$\mathbf{c} = \frac{1}{T} \sum_t \mathbf{h}_t \tag{1.2}$$

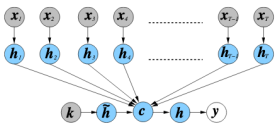
$$\tilde{\mathbf{h}} = \mathcal{F}_2(\mathbf{c}) \tag{1.3}$$

$$\mathbf{y} = \mathcal{F}_3(\tilde{\mathbf{h}}) \tag{1.4}$$

Note that several transformations are required; the first transformation is into a space where averages can be taken meaningfully. The average itself, $\mathbf{c}$, is then transformed into a space in which classification can occur. The network can then simply be trained


Sequence Embedding

An alternative is *Sequence Embedding* in which the final history vector is used to perform the task. This is able to handle variable length input sequences, but tends to be biased towards later inputs. A simple remedy is to used bi-directional information, and the final output is produced using both the forwards and reverse history vector.


Attention Mechanism

The *Attention Mechanism* is the extension of the averaging approach, altering the average to be a function of some key, $\mathbf{k}$.

$$c = \sum_t \alpha_t \mathbf{h}_t \tag{1.5}$$

$$\alpha_t = \frac{\exp(e_t)}{\sum_i \exp(e_i)} \tag{1.6}$$

$$\tilde{\mathbf{h}} = \mathcal{F}_1(\mathbf{k}) \tag{1.7}$$

$$e_t = \mathcal{F}_2(\mathbf{h}_t, \tilde{\mathbf{h}}) \tag{1.8}$$

α_t is a probability mass function over the input sequence. Note that it is vital to choose appropriate $\mathbf{k}$ and $\mathcal{F}_2(\cdot,\cdot)$ which relates the key to the observation. Typical forms of attention include:

Typical Attention Forms

1. Dot-product attention:

$$e_t = \mathbf{h}_t^T \mathbf{W}_{xk} \tilde{\mathbf{h}} \tag{1.9}$$

   This has parameters $\mathbf{W}_{xk}$.

2. Additive attention:

$$e_t = \mathbf{w}^T \tanh(\mathbf{W}_x \mathbf{h}_t + \mathbf{W}_k \tilde{\mathbf{h}}) \tag{1.10}$$

   This has parameters $\mathbf{w}, \mathbf{W}_x$ and $\mathbf{W}_k$.

For example, a question and response task could be solved by encoding both the question and document both forwards and backwards, using a suitable the attention mechanism with the key chosen to be the question encoding. Attention mechanisms are interpretable and the most relevant word in a document can clearly be examined.

1.4.3 Sequence to Sequence

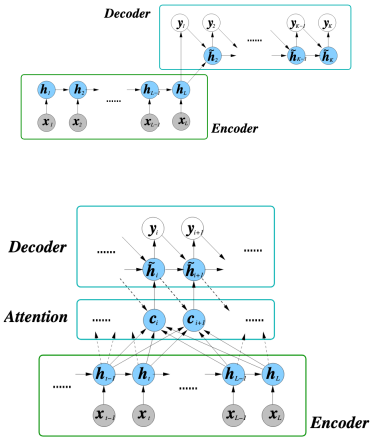
A typical sequence-to-sequence machine learning task would be translating a word sequence into another language.

Encoder-Decoder Sequence Models

An encoder-decoder sequence models trains a model from $\mathbf{x}_{1:L}$ (e.g. source language) and $\mathbf{y}_{1:K}$ (e.g. target language):

$$p(\mathbf{y}_{1:k}|\mathbf{x}_{1:L}) = \prod_{i=1}^{K} p(\mathbf{y}_i|\mathbf{y}_{1:i-1}, \mathbf{x}_{1:L}) \tag{1.1}$$

$$\simeq \prod_{i=1}^{K} p(\mathbf{y}_i|\mathbf{y}_{i-1}, \tilde{\mathbf{h}}_{i-1}, \mathbf{c}) \tag{1.2}$$



Attention-Based Model

i.e. $\mathbf{x}_{1:L}$ must be mapped to a **fixed-length** vector, $\mathbf{c} = \phi(\mathbf{x}_{1:L})$. One form is to use the hidden unit from an RNN/LSTM/GRU i.e. $\mathbf{c} = \mathbf{h}_L$. Depending on the context is global through $\mathbf{c}$, which may be limiting.

An attention based model introduces dependence on locality:

$$p(\mathbf{y}_{1:K}|\mathbf{x}_{1:L}) \simeq \prod_{i=1}^{K} p(\mathbf{y}_i|\mathbf{y}_{i-1}, \tilde{\mathbf{h}}_{i-1}, \mathbf{c}_i) \simeq \prod_{i=1}^{K} p(\mathbf{y}_i|\tilde{\mathbf{h}}_i) \tag{1.3}$$

$$\mathbf{c}_i = \sum_{\tau=1}^{L} \alpha_{i,\tau} \mathbf{h}_\tau \tag{1.4}$$

$$\alpha_{i\tau} = \frac{\exp(e_{i\tau})}{\sum_{j=1}^{L} \exp(e_{ij})} \tag{1.5}$$

$$e_{i\tau} = \mathcal{F}(\tilde{\mathbf{h}}_{i-1}, \mathbf{h}_\tau) \tag{1.6}$$

The decoder part of the model is similar to the Jordan network, and the history vector encodes the outputs **and** context histories i.e.

$$p(w_i|\hat{w}_{0:i-1}, w_{1:L}^{(s)}) \simeq p(w_i|\hat{\mathbf{y}}_{0:i-1}, \mathbf{c}_{1:i}^{(s)}) \simeq p(w_i|\tilde{\mathbf{h}}_i) \tag{1.7}$$

where each $\mathbf{x}$ represents embedding of a source language word, $\mathbf{c}_i$ is the context information for word i, $\hat{w}_i$ is the generated target language word at time i and $\hat{\mathbf{y}}_i$ is the embedding of the generated word $\hat{w}_i$.

Performing Inference

First, the source language word sequence is encoded i.e. $w_{1:L}^{(s)} \rightarrow \mathbf{x}_{1:L}$. Then, to generate word $\hat{w}_i$:

1. Embed the previous word: $\hat{w}_{i-1} \rightarrow \mathbf{y}_{i-1}$.

2. Compute context information using key.

3. Generate new history vector: $\tilde{\mathbf{h}}_i = \mathcal{F}(\tilde{\mathbf{h}}_{i-1}, \hat{\mathbf{y}}_{i-1}, \mathbf{c}_i)$.

4. Use $\tilde{\mathbf{h}}_i$ to compute $P(w_i|\tilde{\mathbf{h}}_i)$, which is then sampled from.

An issue with this approach is that the system can go off-track, and the system isn't very robust to model history errors.

1.4.4 Single Input to Sequence

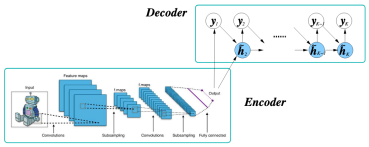


Image Captioning

An example task would be to caption images. In this case, the image would be encoded as a vector using e.g. a deep convolutional network, generating an output vector. This is fed into a recurrent network which generates a caption. Typically trained on example image captions. This system is trained end-to-end, minimising the need for human expertise. The encoder output need not be interpretable.

1.5 Variational Autoencoders

A *Variational Autoencoder* is a rich, latent variable model of the form: *VAE*

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|0, \mathbf{I}) \tag{1.1}$$
$$p(\mathbf{x}|\mathbf{z}, \boldsymbol{\lambda}) = \mathcal{N}(\mathbf{x})|\mathcal{F}(\mathbf{z}; \boldsymbol{\lambda}), \sigma^2\mathbf{I})) \tag{1.2}$$

The idea is to use a deep neural network to learn the mapping between the latent and mean observation. However, the log-likelihood cannot be computed for inference, and it's gradients can also not be computed. Thus, variational approaches are applied.

1.5.1 Variational EM

In the EM algorithm, an auxiliary function is maximised rather than the log-likelihood:

$$\boldsymbol{\lambda}^{(k+1)} = \arg\max_{\boldsymbol{\lambda}} \int p(\mathbf{z}|\mathbf{x}, \boldsymbol{\lambda}^{(k)}) \log p(\mathbf{x}, \mathbf{z}|\boldsymbol{\lambda}) d\mathbf{z} \tag{1.1}$$

This workers because in many models (e.g. factor analysis), the expected value of the log joint is simple, and often a function of the posterior moments of the latent.

However, we now return to the log-likelihood: *Log Likelihood*

$$\begin{aligned}
\mathcal{L}(\boldsymbol{\lambda}) &= \log p(\mathbf{x}|\boldsymbol{\lambda}) \\
&= \int p(\mathbf{z}|\mathbf{x}, \boldsymbol{\lambda}) \log p(\mathbf{x}|\boldsymbol{\lambda}) d\mathbf{z} \\
&= \int p(\mathbf{z}|\mathbf{x}, \boldsymbol{\lambda}) \log \frac{p(\mathbf{x}|\boldsymbol{\lambda}) p(\mathbf{z}|\mathbf{x}, \boldsymbol{\lambda})}{p(\mathbf{z}|\mathbf{x}, \boldsymbol{\lambda})} d\mathbf{z} \\
&= \mathbb{E}_{p(\mathbf{z}|\mathbf{x}, \boldsymbol{\lambda})}\Big[\log \frac{p(\mathbf{z}, \mathbf{x}|\boldsymbol{\lambda})}{p(\mathbf{z}|\mathbf{x}, \boldsymbol{\lambda})} \Big]
\end{aligned} \tag{1.2}$$

In order to evaluate this, the posterior over the latent variables needs to be calculated. *Why is this difficult?*

Consider some proposal distribution, $q(\mathbf{z}|\boldsymbol{\lambda})$:

$$\begin{aligned}\mathcal{L}(\boldsymbol{\lambda}) &= \log p(\mathbf{x}|\boldsymbol{\lambda}) \\ &= \int q(\mathbf{z}|\boldsymbol{\lambda}))\log p(\mathbf{x}|\boldsymbol{\lambda})d\mathbf{z} \\ &= \int q(\mathbf{z}|\boldsymbol{\lambda})\log\frac{p(\mathbf{x}|\boldsymbol{\lambda})p(\mathbf{z}|\mathbf{x},\boldsymbol{\lambda})}{p(\mathbf{z}|\mathbf{x},\boldsymbol{\lambda})}d\mathbf{z} \\ &= \mathbb{E}_{q(\mathbf{z}|\boldsymbol{\lambda})}\Big[\log\frac{p(\mathbf{z},\mathbf{x},\boldsymbol{\lambda})}{p(\mathbf{z}|\mathbf{x},\boldsymbol{\lambda})}\Big] \\ &= \underbrace{\mathbb{E}_{q(\mathbf{z}|\boldsymbol{\lambda})}\Big[\log\frac{p(\mathbf{x},\mathbf{z}|\boldsymbol{\lambda})}{q(\mathbf{z}|\boldsymbol{\lambda})}\Big]}_{\mathcal{F}(q,\boldsymbol{\lambda})} + \underbrace{\mathcal{KL}(q(\mathbf{z}|\boldsymbol{\lambda})||p(\mathbf{z}|\mathbf{x},\boldsymbol{\lambda}))}_{\geq 0}\end{aligned} \tag{1.3}$$

Steps

$\mathcal{F}(q,\boldsymbol{\lambda})$ is a lower bound on the free-energy, and equality is reached when $q(\mathbf{z}|\boldsymbol{\lambda}) = p(\mathbf{z}|\mathbf{x},\boldsymbol{\lambda})$. The EM algorithm initialises q as the posterior for some parameter settings, $q(\mathbf{z}|\boldsymbol{\lambda}) = p(\mathbf{z}|\mathbf{x},\hat{\boldsymbol{\lambda}})$, and then repeatedly:

1. (M-Step): Maximise $\mathcal{F}(q,\boldsymbol{\lambda})$ with respect to $\boldsymbol{\lambda}$, fixing the current proposal distribution. This effective maximises a lower bound of the log-likelihood.
2. (E-Step): Maximise $\mathcal{F}(q,\boldsymbol{\lambda})$ with respect to q i.e. set $q(\mathbf{z}|\boldsymbol{\lambda}) = p(\mathbf{z}|\mathbf{x},\boldsymbol{\lambda})$.

Lyapunov Function for EM

The log-likelihood is a *Lyapunov* function:

$$\mathcal{L}(\boldsymbol{\lambda}^{(k)}) \overset{\text{E Step}}{=} \mathcal{F}(q(\mathbf{z}|\boldsymbol{\lambda}^{(k)}),\boldsymbol{\lambda}^{(k)}) \overset{\text{M Step}}{\leq} \mathcal{F}(q(\mathbf{z}|\boldsymbol{\lambda}^{(k)}),\boldsymbol{\lambda}^{(k+1)}) \leq \mathcal{L}(\boldsymbol{\lambda}^{(k+1)}) \tag{1.4}$$

Each iteration is thus guaranteed to not decrease the likelihood and a local maximum of the likelihood is found. The exact solution found depends on the initial parameters.

General EM

However, we may not be able to calculate the posterior depending on our model, and thus **in general**,

$$\mathcal{L}(\boldsymbol{\lambda}^{(k)}) \overset{\text{in general}}{\geq} \mathcal{F}(q(\mathbf{z}|\boldsymbol{\lambda}^{(k)}),\boldsymbol{\lambda}^{(k)}) \overset{\text{M Step}}{\leq} \mathcal{F}(q(\mathbf{z}|\boldsymbol{\lambda}^{(k)}),\boldsymbol{\lambda}^{(k+1)}) \leq \mathcal{L}(\boldsymbol{\lambda}^{(k+1)}) \tag{1.5}$$

Mean Field

meaning that there is no guarantee of improving the likelihood after each iteration. A standard approximation used is the mean-field approximation:

$$q(\mathbf{z}|\boldsymbol{\lambda}) = \prod_{i=1}^{N} q_i(z_i|\boldsymbol{\lambda}) \tag{1.6}$$

1.5.2 Variational Autoencoders

Recall the equations governing the VAES (Equations 1.1 and 1.2). The likehood of a datapoint is given by:

$$\begin{aligned}p(\mathbf{x}|\boldsymbol{\lambda}) &= \int p(\mathbf{x}|\mathbf{z},\boldsymbol{\lambda})p(\mathbf{z})\,d\mathbf{z} \\ &= \int \mathcal{N}(\mathbf{x}|\mathbf{f}(\mathbf{z}|\boldsymbol{\lambda}),\sigma^2\mathbf{I})\mathcal{N}(\mathbf{z}|\mathbf{0},\mathbf{I})\,d\mathbf{z}\end{aligned} \tag{1.1}$$

This integral cannot be computed; $\mathbf{f}(\mathbf{z}|\boldsymbol{\lambda})$ is a non-linear function.

Variational EM

We could use a variational EM maximisation, maximising the free energy, $\mathcal{F}(q,\boldsymbol{\lambda})$ by alternating maximisations over $\boldsymbol{\lambda}$ and q. Appropriate forms of the proposal distribution are important, and they influence how tight the lower bound is for the optimisation. However, this is not usually possible for deep learning. Please note that $\tilde{\boldsymbol{\lambda}}$ represents the parameters of the proposal distribution (i.e. the encoder) and $\boldsymbol{\lambda}$ represents the parameters of the decoder.

Applying Gradient Descent

The gradient of the likelihood is approximated using the gradient of the lower bound:

$$\begin{aligned}\nabla\mathcal{L}(\boldsymbol{\lambda}) &\simeq \nabla\left(\mathbb{E}_{q(\mathbf{z}|\tilde{\boldsymbol{\lambda}})}\Big[\log\frac{p(\mathbf{x},\mathbf{z}|\boldsymbol{\lambda})}{q(\mathbf{z}|\tilde{\boldsymbol{\lambda}})}\Big]\right)\\ &= \nabla\left(\mathbb{E}_{q(\mathbf{z}|\tilde{\boldsymbol{\lambda}})}[\log p(\mathbf{x}|\mathbf{z},\boldsymbol{\lambda})] + \mathbb{E}_{q(\mathbf{z}|\tilde{\boldsymbol{\lambda}})}\Big[\log\frac{p(\mathbf{z})}{q(\mathbf{z}|\tilde{\boldsymbol{\lambda}})}\Big]\right)\end{aligned} \tag{1.2}$$

A density network is used as the variational approximation, introducing a dependence on $\mathbf{x}$:

$$q(\mathbf{z}|\tilde{\boldsymbol{\lambda}}) \to q(\mathbf{z}|\mathbf{x},\tilde{\boldsymbol{\lambda}}) = \mathcal{N}(\mathbf{z}|\mathcal{F}_\mu(\mathbf{x}|\tilde{\boldsymbol{\lambda}}),\mathcal{F}_\Sigma(\mathbf{x}|\tilde{\boldsymbol{\lambda}})) \tag{1.3}$$

This is the encoder part of the network. The likelihood can now be re-written as:

$$\begin{aligned}\mathcal{L}(\boldsymbol{\lambda}) &= \mathbb{E}_{q(\mathbf{z}|\mathbf{x},\tilde{\boldsymbol{\lambda}})}\Big[\log\frac{p(\mathbf{z},\mathbf{x},\boldsymbol{\lambda})}{p(\mathbf{z}|\mathbf{x},\boldsymbol{\lambda})}\Big]\\ &= \mathbb{E}_{q(\mathbf{z}|\mathbf{x},\tilde{\boldsymbol{\lambda}})}\Big\{\underbrace{\log\frac{q(\mathbf{z}|\mathbf{x},\tilde{\boldsymbol{\lambda}})}{p(\mathbf{z}|\mathbf{x},\boldsymbol{\lambda})}}_{\text{error}} + \underbrace{\log p(\mathbf{x}|\mathbf{z},\boldsymbol{\lambda})}_{\text{decoding}} + \underbrace{\log\frac{p(\mathbf{z})}{q(\mathbf{z}|\mathbf{x},\tilde{\boldsymbol{\lambda}})}}_{\text{encoding}}\Big\}\end{aligned} \tag{1.4}$$

The error terms captures the difference between the likelihood and the free energy. Returning to the gradient of the lower bound, the dependence on $\mathbf{x}$ is now included:

Rewriting and interpreting the likelihood

$$\nabla\mathcal{L}(\boldsymbol{\lambda}) = \nabla\left(\mathbb{E}_{q(\mathbf{z}|\mathbf{x},\tilde{\boldsymbol{\lambda}})}[\log p(\mathbf{x}|\mathbf{z},\boldsymbol{\lambda})] + \mathbb{E}_{q(\mathbf{z}|\mathbf{x},\tilde{\boldsymbol{\lambda}})}\Big[\log\frac{p(\mathbf{z})}{q(\mathbf{z}|\mathbf{x},\tilde{\boldsymbol{\lambda}})}\Big]\right) \tag{1.5}$$

The first term is difficult as the expectations of highly non-linear functions must be computed. Note that this term is a function of both $\boldsymbol{\lambda}$ and $\tilde{\boldsymbol{\lambda}}$ The second term is however easy to calculate as it is the KL divergence between Gaussians which has a simple closed-form solution.

Using Monte-Carlo

Monte Carlo is used to evaluate the first term:

$$\mathbb{E}_{q(\mathbf{z}|\mathbf{x},\tilde{\boldsymbol{\lambda}})}[\log p(\mathbf{x}|\mathbf{z},\boldsymbol{\lambda})] \simeq \frac{1}{N}\sum_{i=1}^{N}\log p(\mathbf{x}|\mathbf{z}^{(i)},\boldsymbol{\lambda}) \text{ with } \mathbf{z}^{(i)} \sim q(\mathbf{z}|\mathbf{x},\tilde{\boldsymbol{\lambda}}) \tag{1.6}$$

Samples can be drawn easily since $q(\mathbf{z}|\mathbf{x},\tilde{\boldsymbol{\lambda}})$ is Gaussian. However, this is directly using samples from $q(\mathbf{z}|\mathbf{x},\tilde{\boldsymbol{\lambda}})$ and it does not allow gradients for $\tilde{\boldsymbol{\lambda}}$ to be calculated.

Reparameterisation Trick

The reparameterisation trick solves this by noting that:

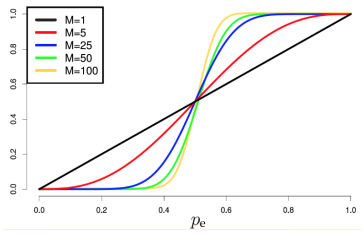
$$\mathbf{z}^{(i)} = \mathcal{F}_\mu(\mathbf{x}|\tilde{\boldsymbol{\lambda}}) + \mathcal{F}_\Sigma(\mathbf{x}|\tilde{\boldsymbol{\lambda}})^{1/2}\mathbf{e}^{(i)} \text{ with } \mathbf{e}^{(i)} \sim \mathcal{N}(\mathbf{0},\mathbf{I}) \tag{1.7}$$

1.6 Ensemble Methods

Simple Binary Classification

Consider an ensemble of binary classifiers with **independent** error probability, p_e, with overall classification performing using majority voting. The probability that such a classifier makes an error is:

$$\mathbb{P}(\text{error}) = \sum_{i=\frac{N+1}{2}}^{N}\binom{N}{i}p_e^i(1-p_e)^{N-1} \tag{1.1}$$



Consider an ensemble of discriminative, probabilistic classifiers. Extended the ensemble method approach, the decision rule becomes:

Decision Rule

$$\hat{\omega} = \arg\max_{\omega}\left\{\sum_{\mathcal{M}}\int \underbrace{P(\omega|\mathbf{x}^*,\boldsymbol{\theta},\mathcal{M})}_{\text{predictive | params}}\overbrace{p(\boldsymbol{\theta}|\mathcal{M},\mathcal{D})}^{\text{posterior}}\underbrace{P(\mathcal{M}|\mathcal{D})}_{\text{posterior over models}}\,d\boldsymbol{\theta}\right\} \tag{1.2}$$

$\mathcal{M}$ specifies a model topology, and $\boldsymbol{\theta}$ are the parameters given that topology. This is a highly complex distribution, so we resort to Monte Carlo.

$$\hat{\omega} = \arg\max_{\omega} \left\{ \frac{1}{N} \sum_{j=1}^{N} P(\omega|\mathbf{x}^*, \mathcal{M}^{(j)}) \right\} \text{ with } \mathcal{M}^{(i)} \sim p(\boldsymbol{\theta}, \mathcal{M}|\mathcal{D})$$
$$\mathcal{E} = \left\{ \mathcal{M}^{(1)}, \ldots, \mathcal{M}^{(N)} \right\} \tag{1.3}$$

However, the posterior $p(\boldsymbol{\theta}, \mathcal{M}|\mathcal{D})$ is incredibly complicated, and it is difficult to efficiently sample over it, especially over model topologies. Most of the approaches for ensemble methods are practical without significant links to theoretical sampling.

Practical ensemble methods:

Bagging

- One such procedure is *Bagging* in which a subset of the data, $\tilde{\mathcal{D}} \subset \mathcal{D}$, $|\tilde{\mathcal{D}}| = N$, is sampled (with or without replacement). The model is trained on $\tilde{\mathcal{D}}$, and this is repeated until the desired ensemble size is generated.

Dropout

- With dropout, random nodes are deactivated during each step of training according to some probability, which acts as regularisation, preventing one path becoming too specialised. In effect, we are sharing weights between a large number of models, and to compute the overall output, we de-weight weights by the inverse of the dropout rate. This is approximately equivalent to computing the geometric mean of each sub-network. **Using this method, we can train an ensemble using one network only**.

Random Network Initialisation

- A simple alternative is to randomly initialise network parameters for a given network topology according to some prior distribution. Each member of the ensemble is then found by training the network from this initialisation, meaning that each ensemble method will be at a local optimum.

Adhoc Models

- Another simple approach is to select different configurations (e.g. numbers and sizes of layers, nature of activation function and nature of the targets), then manually sampling from the 'configuration posterior', with each system trained to a local optimum. This uses expert knowledge to improve synergies between different models in the ensemble.

Posterior Combination

The simplest approach to posterior combination is to average the class posteriors from the ensemble:

$$P(\omega|\mathbf{x}^*, \mathcal{E}) = \frac{1}{N} \sum_{i} P(\omega|\mathbf{x}^*, \mathcal{M}^{(i)}) \tag{1.4}$$

However, this requires all models in the ensemble to be evaluated and stored for each test point.

Teacher Student Training

Recall the standard cross entropy training criterion for classification with $\mathcal{D} = \{(x_i, y_i)\}$ and $y_i \in \{\omega_1, \ldots, \omega_K\}$.

$$\mathcal{L}_{\text{ce}} = -\sum_{i=1}^{n} \sum_{\omega} \delta(y_i, \omega) \log P(\omega|\mathbf{x_i}, \mathcal{M}) \tag{1.5}$$

This can be modified to use soft targets, obtained using some teacher network, $\mathcal{M}_T$:

$$\mathcal{L}_{\text{ts}} = -\sum_{i=1}^{n} \sum_{\omega} P(\omega|\mathbf{x_i}, \mathcal{M}_T) \log P(\omega|\mathbf{x_i}, \mathcal{M}_S) \tag{1.6}$$

This provides a way of performing model compression, replacing the single teacher with the ensemble posterior and attempting to compress this ensemble into a single model (which could be a simpler or more complex model).

1.7 Support Vector Machines

Binary Classification

Given a dataset, $\mathcal{D} = \{(\mathbf{x}_n, t_n)\}_{n=1}^N$ with $\mathbf{x}_n \in \mathcal{R}^d$ and $t_n \in \{+1, -1\}$, our objective is to learn a classifier, $y(\mathbf{x})$, such that:

$$t_n = +1 \Rightarrow y(\mathbf{x}) \geq 0 \qquad t_n = -1 \Rightarrow y(\mathbf{x}) < 0$$

The classifier makes a correct prediction on a new input, $\mathbf{x}^*$, when $y(\mathbf{x}^*)t^* > 0$. Note that the decision boundary is the set of inputs with $y(\mathbf{x}) = 0$.

Linear Separability

A problem is linearly separable when a classifier with a linear decision boundary makes no mistakes on the training data with a linear classifier.

$$y(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b \tag{1.1}$$

$\mathbf{w}$ is orthogonal to the decision boundary and b is known as the bias.

Maximum Margin Classifier

When the data is linearly separable, many possible $\mathbf{w}$ have zero training error. The maximum margin idea is to choose the plane who distance to the closest point in each class (margin) is maximal. Data points on the margin are called support vectors, and the max-margin classifier is fully determined by it's support vectors.

Computation of Max-Margin Classifier

Note that the scale of the classifier is arbitrary i.e. $y(\mathbf{x})$ and $cy(\mathbf{x})$ have identical decision boundaries. The scale is chosen that $y(\mathbf{x}_+) = +1$ for any positive support vector and $y(\mathbf{x}_-) = -1$ for any negative support vector. Then, the magnitude of the margin is:

$$\frac{\mathbf{w}^T(\mathbf{x}_+ - \mathbf{x}_-)}{2||\mathbf{w}||} = \frac{1}{||\mathbf{w}||} \tag{1.2}$$

The minimum value of $|y(\mathbf{x})|$ is achieved by the support vectors, so $t_n y(\mathbf{x}_n) \geq 1 \forall n$.

Optimisation Formulation

Therefore, with the intention to minimise the margin, the optimisation problem can be formulated as:

$$\begin{aligned} &\min_{||\mathbf{w}||} \frac{1}{2}||\mathbf{w}||^2 \\ &\text{s.t. } t_n(\mathbf{w}^T\mathbf{x}_n + b) \geq 1 \ \forall n \end{aligned} \tag{1.3}$$

Note that the constraint enforces that every point is classified correctly. Also note that this is a quadratic problem with linear constraints, meaning that there exists a unique minimum.

SVM Lagrangian

The *Lagrangian* is thus written as follows:

$$\mathcal{L}(\mathbf{w}, b, \boldsymbol{\lambda}) = \frac{1}{2}||\mathbf{w}||_2^2 - \sum_{n=1}^N \lambda_n(t_n(\mathbf{w}^T\mathbf{x}_n + b) - 1) \tag{1.4}$$

Dual Problem

where $\boldsymbol{\lambda}$ is a vector of Lagrange multipliers. Minimising with respect to $\mathbf{w}$ and b gives the *Langrangian Dual Problem*:

$$\begin{aligned} \max_{\boldsymbol{\lambda}} \ \mathcal{F}(\boldsymbol{\lambda}) &= \sum_{n=1}^N \lambda_n - \frac{1}{2}\sum_{n,m} \lambda_n\lambda_m t_n t_m \mathbf{x}_n^T\mathbf{x}_m \\ \text{s.t. } &\sum_n t_n\lambda_n = 0 \\ &a_n \geq 0 \ \forall n \end{aligned} \tag{1.5}$$

Practical Solution

This is a quadratic function subject to inequality constraints. There is no analytic solution, but quadratic programming techniques can be used for a cost between $\mathcal{O}(N^2)$ and $\mathcal{O}(N^3)$. These techniques perform incredibly well for $N \leq 50000$, but are otherwise too expensive.

Predictive Equation

Note that at convergence, from the KKT conditions points with $\lambda_n > 0$ satisfy $t_n y(\mathbf{x}_n) = 1$ and are thus support vectors. Predictions can be made according to:

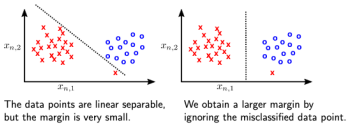
$$y(\mathbf{x}^*) = \sum_{n \in \mathcal{S}} \lambda_n t_n \mathbf{x}_n^T\mathbf{x}^* + b \tag{1.6}$$

where $\mathcal{S}$ denotes the set of support vectors, $\mathcal{S} = \{n : \lambda_n > 0\}$. The bias is obtained from the scale constraint (i.e. the value of the classifier for support vectors is ± 1), averaging to improve numerical stability:

$$b = \frac{1}{|\mathcal{S}|} \sum_{n \in \mathcal{S}} \left\{ t_n - \sum_{m \in \mathcal{S}} \lambda_m t_m \mathbf{x}_m^T \mathbf{x}_n \right\} \tag{1.7}$$

Constraint Violation and Soft Margins

Motivation



Sometimes allowing for misclassification can produce better classifiers e.g. when there may be mistakes in the training data. In general, there is a trade-off between margin size and the number of training errors.

Introducing Errors

Slack variables, $\{\xi_i\}$, are introduced to allow errors:

$$t_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1 - \xi_n \text{ where } \xi_n \geq 0 \ \forall n \tag{1.8}$$

Points with $\xi_n = 0$ are classified correctly, and lie either on the margin or beyond. Points with $\xi_n \in [0, 1]$ lie inside the margin, but on the correct side. Points with $\xi_n > 1$ are misclassified.

Modified Optimisation

The new optimisation problem is:

$$\begin{aligned} \min_{||\mathbf{w}||} \ & \frac{1}{2}||\mathbf{w}||^2 + C \sum_n \xi_n \\ \text{s.t. } & t_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1 - \xi_n \ \forall n \\ & \xi_n \geq 0 \ \forall n \end{aligned} \tag{1.9}$$

Interpret C

C controls the the trade-off between the slack variable penalty and margin. Small C does not penalise errors largely, meaning soft constraint and a wide margin (and vice versa). The Lagrangian is now:

$$\mathcal{L}(\mathbf{w}, b, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \frac{1}{2}||\mathbf{w}||_2^2 + C \sum_n \xi_n - \sum_{n=1}^N \lambda_n (t_n(\mathbf{w}^T \mathbf{x}_n + b) - 1 + \xi_n) - \sum_{n=1}^N \mu_n \xi_n \tag{1.10}$$

Dual

The dual can now be written:

$$\begin{aligned} \max_{\boldsymbol{\lambda}} \ & \mathcal{F}(\boldsymbol{\lambda}) = \sum_{n=1}^N \lambda_n - \frac{1}{2} \sum_{n,m} \lambda_n \lambda_m t_n t_m \mathbf{x}_n^T \mathbf{x}_m \\ \text{s.t. } & \sum_n t_n \lambda_n = 0 \\ & 0 \leq \ a_n \leq C \ 0 \ \forall n \end{aligned} \tag{1.11}$$

The additional constraint is due the constraint that **both** sets of Lagrange multipliers must be non-negative, and during the derivation we find $\lambda_n = C - \mu_n$. This is again solved using quadratic programming, but the support vectors are defined by the points with $\lambda_n > 0$ i.e. those points that satisfy $t_n y(\mathbf{x}_n) = 1 - \xi_n$ since if $0 < \lambda_n < C$, then $\mu_n > 0$ and so $\xi_n = 0$, meaning that these points lie inside the margin.

Obtaining b

b can be found from the constraint on the points on the margin i.e. $\{t_n y(\mathbf{x}_n) = 1\}$ which have $\xi_n = 0$, so $\mu_n > 0$ (and thus $\lambda_n < C$), again averaging to improve stability.

$$b = \frac{1}{|\mathcal{M}|} \sum_{n \in \mathcal{M}} \left\{ t_n - \sum_{m \in \mathcal{S}} \lambda_m t_m \mathbf{x}_m^T \mathbf{x}_n \right\} \tag{1.12}$$

where $\mathcal{M} = \{\mathbf{x}_n : 0 < \lambda_n < C\}$.

Advanced SVM Topics

Non-Linear Classifiers

Support Vector Machines can be expanded to give non-linear decision boundaries by replacing each feature vector, $\mathbf{x}_n$, with a new one, formed by applying non-linear functions to the original vector. i.e. $\boldsymbol{\phi}(\mathbf{x}) = (\phi_1(\mathbf{x}), \phi_2(\mathbf{x}), \ldots, \phi_M(\mathbf{x}))^T$.

Gram Matrix

Recalling the dual problem for the SVM, note that we do not require the expanded features but only their dot product in some space i.e. we need only the *Gram Matrix*, $\mathbf{K}$:

$$[\mathbf{K}]_{n,m} = \boldsymbol{\phi}(\mathbf{x}_n)^T \boldsymbol{\phi}(\mathbf{x}_m) \tag{1.13}$$

Similarly to make predictions, we need only consider the dot products with respect to the support vectors. However, the cost scales linearly with the output dimension of the feature mapping, which could be very large.

Kernel Functions

Instead of performing a feature expansion and computing a dot product, a *Kernel Function* is used to implicitly map $\mathbf{x}_n$, $\mathbf{x}_m$ to its dot product in some high dimensional space **without explicitly performing the feature expansion**.

$$[\mathbf{K}]_{n,m} = \underbrace{k(\mathbf{x}_n, \mathbf{x}_m)}_{\text{Kernel Function}} = \boldsymbol{\phi}(\mathbf{x}_n)^T \boldsymbol{\phi}(\mathbf{x}_m) \tag{1.14}$$

The function $k(\cdot,\cdot)$ is a valid kernel function iff there is a map $\phi(\cdot)$ such that $k(\mathbf{x}_n, \mathbf{x}_m) = \boldsymbol{\phi}(\mathbf{x}_n)^T \boldsymbol{\phi}(\mathbf{x}_m)$. Additionally, for any dataset, the gram matrix $\mathbf{K}$ satisfies $\mathbf{K} = \boldsymbol{\Phi}\boldsymbol{\Phi}^T$ with $\boldsymbol{\Phi} = (\boldsymbol{\phi}(\mathbf{x}_1), \ldots, \boldsymbol{\phi}(\mathbf{x}_N))^T$.

Mercer's Condition

$k(\cdot,\cdot)$ is a valid kernel iff:

1. $k(\cdot,\cdot)$ is symmetric i.e. $k(\mathbf{x}, \mathbf{y}) = k(\mathbf{y}, \mathbf{x})$.
2. **Any** gram matrix obtained with $k(\cdot,\cdot)$ is positive semi-definite.

$$\mathbf{g}^T \mathbf{K} \mathbf{g} = \sum_{i,j} g_i K_{i,j} g_j \geq 0 \tag{1.15}$$

for any vector $\mathbf{g}$ and any data-point values used to produce the gram matrix,

Gaussian Kernel

The Gaussian kernel is defined as follows:

$$k(\mathbf{x}_n, \mathbf{x}_m) = \exp\Big[\frac{-1}{2\sigma}||\mathbf{x}_n - \mathbf{x}_m||^2\Big] \tag{1.16}$$

Effective Dimension

σ is the scale parameter, and controls the smoothness of the decision border. Note that the rank of $\mathbf{K}$ determines the effective dimension of the feature space; $\sigma \to 0$ gives a diagonal matrix with rank K, and a very 'wiggly' decision border. As $\sigma \to \infty$, all entries in $\mathbf{K}$ become the same, giving rank 1 with a smooth decision border.

Choosing Hyperparameters

Note that SVM hyperparameters, such as C, or those in the kernel function are typically found by grid-search, maximising performance on held-out validation data.

The Kernel Trick

In general, any algorithm that operates only on dot-products of inputs can be implemented by using Kernel functions instead. SVMs are very suitable for kernel methods as they can generate high dimensional spaces, but the maximum margin technique provides good generalisation. Additionally, the computational cost scales only with the number of support vectors, meaning that these techniques are not expensive.

One-vs-All Classification

In the situation when there are K classes, *One-vs-All Classifiers* train one classifier (e.g. an SVM) for each class. Considering the case where each classifier is an SVM, labeling occurs as follows:

$$\hat{k}^* = \arg\max_k \mathbf{w}_k^T \mathbf{x} + b_k \tag{1.17}$$

However, the classifiers are not guaranteed to have similar output scales, and there are many more training examples for one class ('other') for each individual classifier.

Simultaneous Learning of Classifiers

In order to remedy this, and train multiple classifiers at once, an additional set of

constraints is introduced for each data-point, $(\mathbf{x}_n, t_n)$:

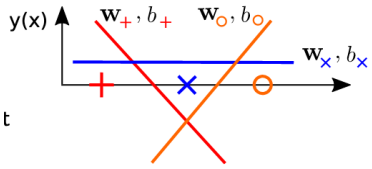
$$\mathbf{w}_{t_n}^T \mathbf{x} + b_{t_n} > \mathbf{w}_j^T \mathbf{x} + b_j \ \forall j \neq t_n \tag{1.18}$$

New Objective Function

The new optimisation problem is:

$$\begin{aligned} \min_{\{\mathbf{w}_i, b_i\}_{i=1}^K} \ & \frac{1}{2} \sum_k ||\mathbf{w}_k||^2 + C \sum_{n, j \neq t_n} \xi_{n,j} \\ \text{s.t. } & t_n(\mathbf{w}^T \mathbf{x}_n + b) \geq \mathbf{w}_j^T \mathbf{x} + b_j + 1 - \xi_{n,j} \\ & \xi_{n,j} \geq 0 \qquad (\forall n, j \neq t_n) \end{aligned} \tag{1.19}$$

Limitations



Predictions are evaluated as before. However, this approach has a very large computation cost; there are far more variables in the dual problem. Often in practice, simple one-vs-all classification is used.

Note that one-vs-all classification can often not work, and may not provide the right answer depending on the distribution of classes in the input space. An example of this is the simple, 1D case. However, the multi-class simultaneous learning approach can solve this problem.

1.8 Kernels for Structured Data

Motivation

The kernels that we have seen so far only work with fixed length input vectors, but many real world data sets are non-vectorial, for example, biological sequences and text documents. One approach is to extract features from input objects and compute kernels based on those features.

String Kernel

Consider the situation where each data point, x, is a string of characters from $\mathcal{A}$. Given a list of sub-strings, $\{s_1, s_2, \ldots\}$, x can be encoded using the following feature vector:

$$\phi(x) = \begin{bmatrix} \phi_{s_1}(x) & \phi_{s_2}(x) & \cdots \end{bmatrix}^T \tag{1.1}$$

Each feature indicates the occurrence of the sub-string s in x. This could be, for example, gap weighted e.g. for n gaps, $\phi_s(x) = \lambda^n$ with $\lambda \in [0, 1]$ where n is the number of gaps in the occurrence. This results in a larger number of gaps giving a weaker value.

k-spectrum kernel

The k-spectrum kernel considers all possible sub-sequences of length k. $\phi_s(x)$ is then defined as the number of occurrences of s in x. The kernel function can be computed quickly using a suffix tree . Note that co-occurrence of sub-strings is more informative for longer sub-strings, but the number of common occurrences decreases as k increases meaning that there is an optimal, balanced value of k. $k = 1$ gives the bag-of-words kernel, and is often applied to documents.

Tree Kernels

Consider the situation where each data-point is a tree $\mathcal{T}$. A subtree is defined as tree formed by selecting one node and all of its descendants from another tree. A subset tree is similar, but either includes all children of a node or none of them. Given a list of possible subset trees, $t_1, t_2, \ldots$, each tree $\mathcal{T}$ is encoded as :

$$\phi(\mathcal{T}) = \begin{bmatrix} \phi_{t_1}(\mathcal{T}) & \phi_{t_2}(\mathcal{T}) & \ldots \end{bmatrix} \tag{1.2}$$

where each $\phi_t(\mathcal{T})$ counts the occurrences of t in $\mathcal{T}$. The kernel is then simply defined as the dot-product.

Efficient Computation of Tree Kernels

The kernel can be computed efficiently using a recursive algorithm:

$$k(\mathcal{T}_a, \mathcal{T}_b) = \sum_{n_a} \sum_{n_b} f(n_a, n_b) \tag{1.3}$$

$f(n_a, n_b)$ counts the number of common subset trees starting at n_a, n_b. Denote the children of node n as children(n) and the ith child node of n as children$(n)_i$. Then, there are three situations:

(a): If $n_a \neq n_b$ or children$(n_a) \neq$ children(n_b), then $f(n_a, n_b) = 0$.

(b): Else if n_a and n_b are leaf nodes then $f(n_a, n_b) = 1$.

(c): Otherwise:

$$f(n_a, n_b) = \prod_{i}^{|\text{children}(n_a)|} g(\text{children}(n_a)_i, \text{children}(n_b)_i) \tag{1.4}$$

$$g(n_1, n_2) = \begin{cases} 1 & \text{if either is a leaf node} \\ 1 + f(n_1, n_2) & \text{otherwise} \end{cases} \tag{1.5}$$

The form of this expression can be justified by noting that for each sub-tree possible at a node, there are an additional $f(n_1, n_2)$ sub-trees by including the valid parent and neighbouring nodes and for each possible sub-tree at each child, the sub-tree formed by the valid parent and another sub-tree from another child is another valid sub-tree.

Graphs

A graph is defined as $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$. $\mathcal{V}$ is the node set and $\mathcal{E}$ is the edge set with $\mathcal{E} = \{(i, j) : i, j \in \mathcal{V}\}$. The $|\mathcal{V}| \times |\mathcal{V}|$ adjacency matrix, $\mathbf{A}$, satisfies $[A]_{i,j} = 1$ if $(i, j) \in \mathcal{E}$, otherwise 0. A k-length walk is defined as $w = \{v_1, \ldots, v_{k+1}\}$ where $(v_i, v_{i+1}) \in \mathcal{E}$. The number of k-length walks between nodes i and j is given by $[\mathbf{A}^k]_{ij}$ since:

k-length walks

$$[\mathbf{A}^k]_{i,j} = \sum_{s_1} \cdots \sum_{s_{k-1}} [\mathbf{A}]_{i,s_1} [\mathbf{A}]_{s_1,s_2} \cdots [\mathbf{A}]_{s_{k-1},j} \tag{1.6}$$

Random-Walk Graph Kernel

For the case where each data-point is a graph, a kernel function, $k(\mathcal{G}, \mathcal{G}')$ counts the number of common walks in $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ and $\mathcal{G}' = \{\mathcal{V}', \mathcal{E}'\}$. The direct product graph of $\mathcal{G}$ and $\mathcal{G}'$ is defined as :

$$\begin{aligned} \mathcal{G}_\times &= \{\mathcal{V}_\times, \mathcal{E}_\times\} \\ \mathcal{V}_\times &= \{(a, b) : a \in \mathcal{V} \text{ and } b \in \mathcal{V}'\} \\ \mathcal{E}_\times &= \{((a, b), (a', b') : (a, b)) \in \mathcal{E} \text{ and } (a', b') \in \mathcal{E}')\} \end{aligned} \tag{1.7}$$

Kronecker Product

There is an edge between notes in $\mathcal{G}_\times$ is there a path between both corresponding notes in the original graphs. Thus the adjacency matrix, $\mathbf{A}_\times$ is the *Kronecker Product* of the original adjacency matrices:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} A_{1,1}\mathbf{B} & \cdots & A_{1,N}\mathbf{B} \\ \cdots & \ddots & \cdots \\ A_{N,1}\mathbf{B} & \cdots & A_{N,N}\mathbf{B} \end{bmatrix} \tag{1.8}$$

Each walk in $\mathcal{G}_\times$ corresponds to a shared walk in $\mathcal{G}$ and $\mathcal{G}'$. $\mathbf{A}_\times$ is the *Kronecker Product* of $\mathbf{A}$ and $\mathbf{A}'$. The kernel is then computed as:

$$\begin{aligned} k(\mathcal{G}, \mathcal{G}') &= \sum_{i,j}^{|\mathcal{V}_\times|} \left[\sum_{n=0}^{\infty} \lambda^n \mathbf{A}_\times^n \right]_{i,j} \\ &= \mathbf{1}^T \underbrace{[\mathbf{I} - \lambda \mathbf{A}_\times]^{-1} \mathbf{1}}_{\mathbf{x}} \end{aligned} \tag{1.9}$$

which yields the equation:

$$\mathbf{x} = 1 + \lambda \mathbf{A}_\times \mathbf{x} \tag{1.10}$$

Problems

where it has been assumed that $\lambda > 0$ is small enough to guarantee convergence, and the formula for a geometric series has been used. Thus $\mathbf{x}$ can be found using an efficient, iterative scheme, but a walk can visit the same cycle repeatedly meaning that small structural similarities can produce large kernel values. Additionally, there is a high cost: $\mathcal{O}(n^3)$ for $n \times n$ matrices.

Weisfeiler-Lehman Graph Kernel

The *Weisfeilher-Lehman* graph kernel, for every single graph in the dataset, repeats the following steps:

1. For each node in the current graph:
   (a): Create a set with labels of adjacent vertices.
   (b): Sort the label set, append the vertex label as a prefix.
   (c): Compress the label into a unique value.
2. Assign to each node its unique value as the new vertex label.

The kernel is obtained by applying the bag-of-words kernel to the vertex labels obtained throughout **all** of the iterations, including the counts of the original labels. This is a low cost kernel, and can be applied to large graphs.

Fisher Kernel

A *Fisher Kernel* trains a probabilistic generative model to obtain a fixed-length vector representation of structure data. The kernel is obtained as follows:

(a): Train a generative model, $p(\mathbf{x}|\boldsymbol{\theta})$ on the available input data e.g. by maximum likelihood.

(b): Define the *Fisher Score* vector as

$$\phi(\mathbf{x}_n) = \nabla_\theta \log p(\mathbf{x}_n|\boldsymbol{\theta})|_{\boldsymbol{\theta}_{\text{MLE}}} \tag{1.11}$$

(c): The naive Fisher kernel is then given by:

$$k(\mathbf{x}_n, \mathbf{x}_m) = \phi(\mathbf{x}_n)^T \phi(\mathbf{x}_m) \tag{1.12}$$

1.9 Karush-Kuhn-Tucker (KKT) Conditions

Consider the constrained optimisation problem:

$$\begin{aligned} &\min_x f(x) \\ &\text{s.t. } g(x) \leq 0 \end{aligned} \tag{1.1}$$

Denote a unique local optimum of this solution as x^*. There are two situations:

1. The constraint is not active at the solution. Therefore:
$$\begin{aligned} \nabla f(x^*) &= 0 \\ \nabla^2 f(x^*) &> 0 \\ g(x^*) &\leq 0 \end{aligned}$$
2. The constraint is active at the solution.
$$\begin{aligned} \nabla f(x^*) &= -\mu \nabla h(x^*) \\ g(x^*) &= 0 \end{aligned}$$
Noting that μ **must be positive**, otherwise x^* cannot be a minimum. This can be verified by sketching this situation.

Langrangian

The constrained optimisation problem can now be written using the *Langrangian*.

$$\mathcal{L}(x, \mu) = f(x) + \mu g(x) \tag{1.2}$$

and any optimum, x^*, satisfies the *Karush-Kuhn-Tucker Conditions*:

$$\nabla \mathcal{L}(x^*, \mu^*) = 0 \tag{1.3}$$
$$\mu^* g(x^*) = 0 \tag{1.4}$$
$$\mu^* \geq 0 \tag{1.5}$$
$$g(x^*) \leq 0 \tag{1.6}$$

The KKT conditions encode the observations made before. Define the function: *Duality*

$$\mathcal{F}(\mu) = \min_x f(x) + \mu g(x) \tag{1.7}$$

Note that $\mathcal{F}(\mu)$ provides a lower-bound of the optimal cost for each value of $\mu \geq 0$ since $g(x) \leq 0$. This gives the *Dual Optimisation Problem*, which is to maximise this lower bound, obtaining the best lower bound of the *Primal*, problem.

$$\begin{aligned} &\max_\mu \mathcal{F}(\mu) \\ &\text{s.t. } \mu \geq 0 \end{aligned} \tag{1.8}$$