Fundamentals of machine learning

Probability of error and decision boundaries

Training data

- The "standard" process for obtaining data for a task is as follows:
  - ↳ obtain $\underline{x}_i$ from an initial deployment / available data $\quad \underline{x}_i \sim p(\underline{x})$
  - ↳ obtain label y_i manually / from outcomes $\quad x_i \sim P(w|\underline{x}_i)$
  - → these are "draws" from the joint dist $p(w, \underline{x})$
- This yields supervised training data

$$D = \{\{\underline{x}_1, y_1\}, \ldots, \{\underline{x}_N, y_N\}\}$$

where $\underline{x}_i$ is the observation / feature vector

and $y_i \in \{w_1, \ldots, w_k\}$ is the class label for observation $\underline{x}_i$

Nature of classifier

- Classifiers can be broadly split as
  - ① Generative models $p(\underline{x}, w; \theta)$ ← we could use this to generate synthetic data
    - ↳ Model of the joint dist. of observations and classes is trained
    - ↳ Posterior dist. of class w_i obtained using Bayes rule:
  - ② Discriminative models $p(w|\underline{x}; \theta)$ ← cannot generate synthetic data unless given $\underline{x}$
    - ↳ Model the posterior dist. of the class given the observations is trained
  - ③ Discriminant functions
    - ↳ A mapping from an observation $\underline{x}^*$ to a class w is directly trained
    - ↳ No posterior probability generated, just class label.

Loss functions

- A loss function is a mathematical way to measure how good / bad the predictions of a ML method are compared to the actual values in the data.
- Common loss functions include:
  - ↳ squared error $\quad L(\hat{w}, w) = (\hat{w} - w)^2$
  - ↳ 0-1 loss $\quad L(\hat{w}, w) = \begin{cases} 0 & \text{if } \hat{w} = w \\ 1 & \text{o/w} \end{cases}$
  - ↳ cross entropy $\quad L(p, w) = -\sum_i w_i \log p_i$

p is probability vector, w is one-hot encoding vector

* Loss functions may be unequal for particular applications

↳ eg: $L(f(\underline{x}^*; \theta), w_1) = \begin{cases} 0 & f(\underline{x}^*; \theta) = w_1 \\ c_1 & f(\underline{x}^*; \theta) = w_2 \end{cases}$

$L(f(\underline{x}^*; \theta), w_2) = \begin{cases} c_2 & f(\underline{x}^*; \theta) = w_1 \\ 0 & f(\underline{x}^*; \theta) = w_2 \end{cases}$ where $c_1 \neq c_2$.

Fundamentals of machine learning

Probability of error and decision boundaries

Training data

- the "standard" process for obtaining data for a task is as follows:
  - ↳ obtain $\underline{x}_i$ from an initial deployment / available data $\quad \underline{x}_i \sim P(\underline{x})$
  - ↳ obtain label y_i manually / from outcomes $\quad x_i \sim P(w|\underline{x}_i)$
  - → these are "draws" from the joint dist $P(w, \underline{x})$
- this yields supervised training data

$$D = \{\{\underline{x}_1, y_1\}, \ldots, \{\underline{x}_N, y_N\}\}$$

where $\underline{x}_i$ is the observation / feature vector

and $y_i \in \{w_1, \ldots, w_k\}$ is the class label for observation $\underline{x}_i$

Nature of classifier

- classifiers can be broadly split as
  - ① Generative models $P(\underline{x}, w; \theta)$ ← we could use this to generate synthetic data
    - ↳ Model of the joint dist. of observations and classes is trained
    - ↳ posterior dist. of class w_i obtained using Bayes rule:
  - ② Discriminative models $P(w|\underline{x}; \theta)$ ← cannot generate synthetic data unless given $\underline{x}$
    - ↳ Model the posterior dist. of the class given the observations is trained
  - ③ Discriminant functions
    - ↳ A mapping from an observation $\underline{x}^*$ to a class w is directly trained
    - ↳ No posterior probability generated, just class label

Loss functions

- A loss function is a mathematical way to measure how good / bad the predictions of a ML method are compared to the actual values in the data.
- common loss functions include:
  - ↳ squared error $\quad L(\hat{w}, w) = (\hat{w} - w)^2$
  - ↳ 0-1 loss $\quad L(\hat{w}, w) = \begin{cases} 0 & \text{if } \hat{w} = w \\ 1 & \text{o/w} \end{cases}$
  - ↳ cross entropy $\quad L(p, w) = -\sum_i w_i \log p_i$

p is probability vector, w is one-hot encoding vector

* Loss functions may be unequal for particular applications

↳ ex: $L(f(\underline{x}^*; \theta), w_1) = \begin{cases} 0 & f(\underline{x}^*; \theta) = w_1 \\ c_1 & f(\underline{x}^*; \theta) = w_2 \end{cases}$

$L(f(\underline{x}^*; \theta), w_2) = \begin{cases} c_2 & f(\underline{x}^*; \theta) = w_1 \\ 0 & f(\underline{x}^*; \theta) = w_2 \end{cases}$ where $c_1 \neq c_2$

Expected loss, empirical loss and heldout empirical loss

- We can obtain a good predictor function $f(\underline{x};\theta)$ by training the model parameters θ to minimise the expected loss L_{act}, given by (average loss over K classes; average over observations $\underline{x}$)

$$L_{act} = \int \left[\sum_{i=1}^{K} L(f(\underline{x};\theta), w_i)\, p(w_i|\underline{x}) \right] p(\underline{x})\, d\underline{x}$$

where $p(w_i|\underline{x})$ is the "true" prob. of class given observation and $p(\underline{x})$ is the "true" prob. of an observation —] neither of these is usually known

- We can instead compute the empirical loss L_{emp} from the training data D, which are samples drawn from $p(w,\underline{x}) = p(w|\underline{x})\, p(\underline{x})$.

$$L_{emp} = \frac{1}{N} \sum_{i=1}^{N} L(f(\underline{x}_i,\theta), y_i)$$

by Monte Carlo, $\lim_{N\to\infty} L_{emp} = L_{act}$, but in practice N is finite so $L_{emp} \leq L_{act}$

- However, we only care about heldout data performance, so just consider the held-out evaluation set of N^* samples to compute the held-out empirical loss L_{eval}

$$L_{eval} = \frac{1}{N^*} \sum_{i=1}^{N^*} L(f(\underline{x}_i^*,\theta), y_i^*)$$

note loss function used for training can be diff. to that used to compute L_{eval}.

Bayes decision rule

- Given an i/p $\underline{x}^*$, we need to make a "decision" $\hat{w} \in \{w_1 \dots w_K\}$, and associated w/ any decision is a loss/risk $L(\hat{w}, y^*)$, where $y^* \in \{w_1, \dots w_K\}$ is the "correct" outcome.

* Due to uncertainty, $y^* \sim P(w|\underline{x}^*)$ ("true" prob. of class given observation, unknown)

- Bayes decision rule minimises the expected loss to give the optimal decision

$$\hat{w} = \underset{w}{\operatorname{argmin}} \left\{ \sum_{i=1}^{K} L(w, w_i)\, p(w_i|\underline{x}^*) \right\}$$

and we train a model $P(w|\underline{x}^*;\theta)$ to approx. unknown "true" prob. $p(w_i|\underline{x}^*)$

- For equal losses across classes, we select the class w/ highest posterior $P(w|\underline{x}^*;\theta)$,

$$\hat{w} = f(\underline{x}^*;\theta) = \underset{w}{\operatorname{argmax}}\, p(w|\underline{x}^*;\theta) \approx \underset{w}{\operatorname{argmax}}\, p(w|\underline{x}^*)$$

Probability of error

- For 0-1 loss function, we can estimate the prob. of error w/ the heldout empirical loss L_{eval},

$$p(\text{error}) \approx L_{eval} = \frac{1}{N^*} \sum_{i=1}^{N^*} L(f(\underline{x}_i^*,\theta), y_i)$$

- For a classifier $f(\underline{x};\theta)$ that yields two regions $\Omega_1 \to w_1$, $\Omega_2 \to w_2$, the prob. of error is

$$p(\text{error}) = p(w_1, \underline{x} \in \Omega_2) + p(w_2, \underline{x} \in \Omega_1) = \int_{\Omega_2} p(w_1, \underline{x})\, d\underline{x} + \int_{\Omega_1} p(w_2, \underline{x})\, d\underline{x}$$

This can be expressed in two forms

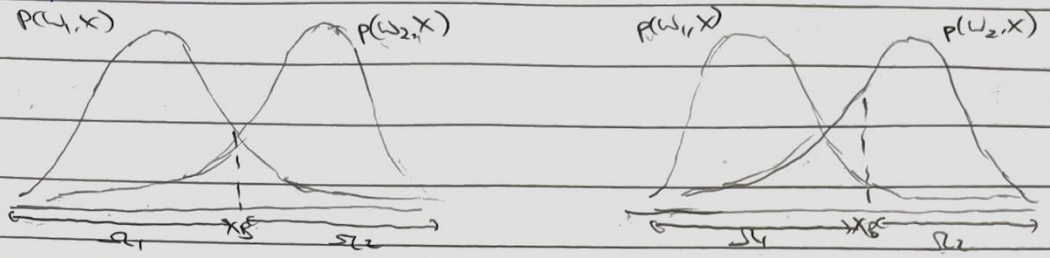
↳ Generative model: $p(\text{error}) = \int_{\Omega_2} p(\underline{x}|w_1)\, p(w_1)\, d\underline{x} + \int_{\Omega_1} p(\underline{x}|w_2)\, p(w_2)\, d\underline{x}$

↳ Discriminative model: $p(\text{error}) = \int_{\Omega_2} p(w_1|\underline{x})\, p(\underline{x})\, d\underline{x} + \int_{\Omega_1} p(w_2|\underline{x})\, p(\underline{x})\, d\underline{x}$

- The optimal decision boundary $\underline{x}_B$ is where the min. prob. of error is obtained (same decision boundary as Bayes decision rule)

Receiver operating characteristic (ROC) curve

- Consider the binary classification case, i.e. $\omega, y \in \{\omega_1, \omega_2\}$. For equal losses, the optimal decision boundary is where $p(\omega_1, x) = p(\omega_2, x) = 0.5$.
- However, there are cases where unequal losses are needed, and the optimal decision boundary is no longer where $p(\omega_1, x) = p(\omega_2, x)$



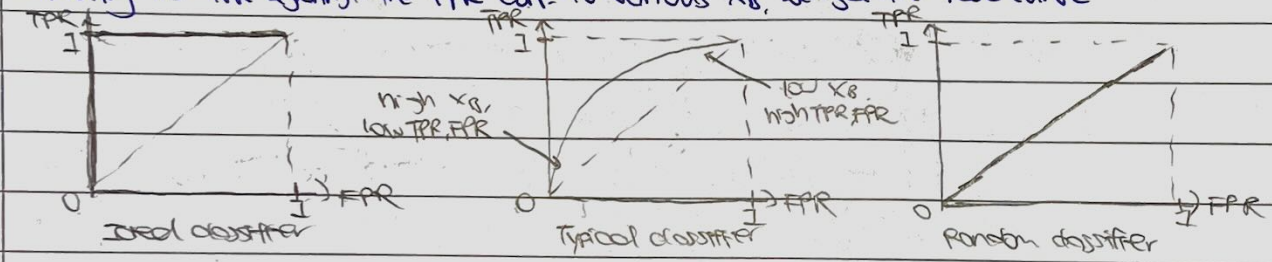
- For a given threshold x_θ, we can construct a confusion matrix based on the evaluation set

actual y \ predicted $\hat{y} = f(x;\theta)$	1	0
1	TP	FN
0	FP	TN

- The true positive rate (TPR) and false positive rate (FPR) are defined as follows:

$$TPR = \frac{TP}{TP + FN} \qquad FPR = \frac{FP}{FP + TN}$$

- Plotting the TPR against the FPR corr. to various x_θ, we get the ROC curve



- To compare diff. ROC curves, we can consider the metric area under curve (AUC)
  (loses information due to dimensionality reduction) ideal when AUC = 1

⇒ Useful for determining the req. (unequal) losses and the decision boundary x_θ.

Decision boundary (DB)

- The DB is a hyper-plane btwn class labels where the (log) class posteriors are equal. For the DB btwn two classes w/ labels ω_1, ω_2.

$$\log p(\omega_1 | x;\theta) = \log p(\omega_2 | x;\theta) \rightarrow \log p(\omega_1) p(x|\omega_1;\theta) = \log p(\omega_2) p(x|\omega_2;\theta)$$

- For a MoG model, we have multivariate Gaussian dist. as class-conditional pdfs i.e.

$$p(x|\omega_i;\theta_i) = N(x|\mu_i, \Sigma_i) = \frac{1}{(2\pi)^{D/2}|\Sigma_i|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_i)^T \Sigma_i^{-1} (x-\mu_i)\right)$$

At the DB btwn classes 1/2, $p(x|\omega_1;\theta_1) = p(x|\omega_2;\theta)$, which is hyper-quadratic

$$x^T A x + b^T x + c = 0$$

where $A = \Sigma_1^{-1} - \Sigma_2^{-1}$, $b = 2(\Sigma_2^{-1}\mu_2 - \Sigma_1^{-1}\mu_1)$, $c = \mu_1^T \Sigma_1^{-1} \mu_1 - \mu_2^T \Sigma_2^{-1} \mu_2 - \log\left(\frac{|\Sigma_2|}{|\Sigma_1|}\right) - 2\log\left(\frac{p(\omega_1)}{p(\omega_2)}\right)$

- If $\Sigma_1 = \Sigma_2 = \Sigma$, we have a linear DB, $b^T x + c = 0$

conditional independence and graphical models

Conditional independence (CI).

- If X and Y are conditionally independent given Z, i.e. $X \perp Y \mid Z$,

$$P(X,Y|Z) = P(X|Z)P(Y|Z)$$

- Modelling data often req. specifying a high-dim. dist. $p(X_1 \dots X_d)$, but working w/ fully flexible joint dist. is intractable.
- If we work w/ structured dist, we can use CI to rewrite $p(X_1 \dots X_d)$ as a product of simple factors evaluated only on a subset of $X_1 \dots X_d$.
  - ↳ This results in a compact representation of the dist.
  - ↳ This simplifies the fit of the dist. parameters to data
  - ↳ Allows us to sum out variables efficiently (compute normalisation const. in polytime)
- Factorisations can be encoded one-to-one as graphs (graphical models) s.t. nodes are RVs, and edges connect variables for which no CI exists.

Directed graphical models (Bayesian networks)

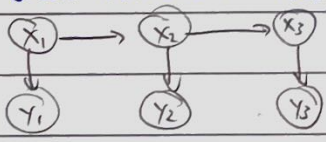
- A Bayesian network G is a directed acyclic graph whose nodes are RVs $X_1 \dots X_d$.

Let $PA^G_{X_i}$ be the parents of X_i in G, and $ND^G_{X_i}$ be the non-descendants of X_i in G.

- G encodes the factorisation $p(X_1 \dots X_d) = \prod_{i=1}^{d} p(X_i | PA^G_{X_i})$

and the CI $(X_i \perp ND^G_{X_i} | PA^G_{X_i})$, $i = 1 \dots d$

- The BN expresses the joint dist. as a product of factors which depend only on a small no. of variables → we can exploit BN factorisation to avoid generating very large factors (prob. tables) during the summation process.
- Markov models are an example of BN, where we have
  - ↳ 1st order Markov : $W_{t+1} \perp W_1 \dots W_{t-1} | W_t$
  - ↳ 2nd order Markov : $W_{t+1} \perp W_1 \dots W_{t-2} | W_t, W_{t-1}$



$p(X_1,X_2,X_3,Y_1,Y_2,Y_3) = p(Y_1|X_1)p(Y_2|X_2)p(Y_3|X_3)p(X_3|X_2)p(X_2|X_1)p(X_1)$

CI: $Y_1 \perp X_2,X_3,Y_2,Y_3 | X_1$, $Y_2 \perp X_1,X_3,Y_3 | X_2$ ($X_2 \perp X_1,Y_1,Y_2 | X_2$), $Y_3 \perp X_1,X_2,Y_1,Y_2 | X_3$

say we want to compute $p(Y_1,Y_2,Y_3) = \sum_{X_1,X_2,X_3} p(X_1,X_2,X_3,Y_1,Y_2,Y_3)$, → $O(N^3)$.

using CI, $p(Y_1,Y_2,Y_3) = \sum_{X_3}\left(\sum_{X_2}\left(\sum_{X_1} p(X_1)p(X_2|X_1)p(Y_1|X_1)\right)p(X_3|X_2)p(Y_2|X_2)\right)p(Y_3|X_3)$ → $O(N^2)$

* $\sum_{X_1} p(X_1)p(X_2|X_1)p(Y_1|X_1) = \sum_{X_1} p(X_1,X_2,Y_1) = p(X_2,Y_1)$ N entries for X_1,X_2 → N^2 cost

$\sum_{X_2} p(X_2,Y_1)p(X_3|X_2)p(Y_2|X_2) = \sum_{X_2} p(X_2,X_3,Y_1,Y_2) = p(X_3,Y_1,Y_2)$ N entries for X_2,X_3 → N^2 cost

$\sum_{X_3} p(X_3,Y_1,Y_2)p(Y_3|X_3) = \sum_{X_3} p(X_3,Y_1,Y_2,Y_3) = p(Y_1,Y_2,Y_3)$ N entries for X_3 → N cost

Undirected graphical models (Markov networks)

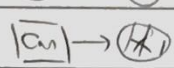
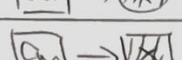
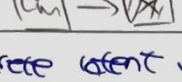
- A Markov network G is an undirected graph whose nodes are the RVs $X_1 \dots X_d$. Consider positive potential functions $\phi_1(D_1) \dots \phi_k(D_k)$, where $D_1 \dots D_k$ are sets of variables, each forming a clique of G (clique is a fully-connected subset of nodes).
- G encodes the factorisation $p(X_1 \dots X_d) = Z^{-1} \prod_{i=1}^{k} \phi_i(D_i)$ Z is the partition function, $Z = \sum_{X_1 \dots X_d} \prod_{i=1}^{k} \phi_i(D_i)$

  and the CI $(A \perp B \mid C)$ for any sets of nodes A, B, C s.t. C separates A from B in G
- MN are better options over BN when having to choose a dir. for the edges is rather unnatural (eg. symmetry in the graph).

Latent variable and sequence models

Latent variable models

Latent variable generative models

- common Bayesian networks for latent variable generative models are:
  - Factor analysis $p(x) = \int p(x|z)p(z)dz$ 
  - Finite mixture model $p(x) = \sum_{m=1}^{M} p(c_m)\, p(x|c_m)$ 
  - Discrete mixture model $p(x) = \sum_{m=1}^{M} p(c_m)\, p(x|c_m)$ 

where x is the observation and z/c_m are the continuous/discrete latent variables.

(Graphically, $\square$ = discrete, $\bigcirc$ = continuous // shaded = observed not shaded = latent)

Factor analysis (FA)

- In FA, the observations $\underline{x} \in \mathbb{R}^d$ are assumed to be generated by a process of the form

$$\underline{x} = \underline{\underline{C}}\,\underline{z} + \underline{v}$$

where $\underline{z} \in \mathbb{R}^p \sim N(\underline{0}, \underline{\underline{I}})$ is the latent variable ($p<d$), $\underline{\underline{C}} \in \mathbb{R}^{d \times p}$ is the loading matrix and $\underline{v} \in \mathbb{R}^d \sim N(\underline{0}, \underline{\underline{\Sigma}}_{diag})$ [$\underline{\underline{\Sigma}}_{diag} = \sigma^2 \underline{\underline{I}}$ is probabilistic PCA].

$\rightarrow p(\underline{x}|\underline{z}) = N(\underline{x}|\underline{\underline{C}}\,\underline{z}, \underline{\underline{\Sigma}}_{diag})$. $p(\underline{x}) = N(\underline{x}|\underline{0}, \underline{\underline{C}}\,\underline{\underline{C}}^T + \underline{\underline{\Sigma}}_{diag})$

Gaussian mixture model

- The Gaussian mixture model / mixture of Gaussians (MoG) model is given by

$$p(\underline{x}) = \sum_{m=1}^{M} \pi_m N(\underline{x}|\underline{\mu}_m, \underline{\underline{\Sigma}}_m)$$

where π_m is the cluster membership distribution,

$$\pi_m = p(c=m|\theta) \quad . \quad \sum_{m=1}^{M} \pi_m = 1 \, .$$

Jensen's inequality

- for a concave function f, and $\lambda \in [0,1]$.

$$\lambda f(x) + (1-\lambda) f(y) \leq f(\lambda x + (1-\lambda) y)$$

We can generalize the result to expectations

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

Alternatively, for a prob distribution w_k, i.e. $\sum_k w_k = 1$,

$$\sum_k w_k f(x_k) \leq f\left(\sum_k w_k x_k\right)$$

- setting $f = \log$, we have

$$\boxed{\sum_k w_k \log x_k \leq \log \sum_k w_k x_k}$$

Auxiliary function Q

- We would like to optimise the model parameters θ to max. the log-likelihood $L(\theta)$
  $L(\theta)$ is difficult to max. → consider a lower bound auxiliary function $Q(\theta)$ to max.
- We want to iterate the model parameters $\theta^t \to \theta^{t+1}$ to increase the log-likelihood $L(\theta^t) \to L(\theta^{t+1})$

$$L(\theta^{t+1}) - L(\theta^t) = \sum_{i=1}^{n} \log \frac{p(x^{(i)}|\theta^{t+1})}{p(x^{(i)}|\theta^t)} > 0$$

- We can rewrite $L(\theta^{t+1}) - L(\theta^t)$ as

$$L(\theta^{t+1}) - L(\theta^t) = \sum_{i=1}^{n} \log \frac{\sum_{m=1}^{M} p(x^{(i)}, c=m|\theta^{t+1})}{p(x^{(i)}|\theta^t)} = \sum_{i=1}^{n} \log \sum_{m=1}^{M} p(c=m|x^{(i)},\theta^t) \frac{p(x^{(i)}, c=m|\theta^{t+1})}{p(c=m|x^{(i)},\theta^t)p(x^{(i)}|\theta^t)}$$

Jensen $\geq \sum_{i=1}^{n}\sum_{m=1}^{M} p(c=m|x^{(i)},\theta^t) \log \frac{p(x^{(i)}, c=m|\theta^{t+1})}{p(c=m|x^{(i)},\theta^t)p(x^{(i)}|\theta^t)}$

$$= \sum_{i=1}^{n}\sum_{m=1}^{M} p(c=m|x^{(i)},\theta^t) \log \frac{p(x^{(i)}, c=m|\theta^{t+1})}{p(x^{(i)}, c=m|\theta^t)}$$

We therefore have the inequality

$$L(\theta^{t+1}) - L(\theta^t) \geq Q(\theta^t, \theta^{t+1}) - Q(\theta^t, \theta^t)$$

where $Q(\theta^t, \theta') = \sum_{i=1}^{n}\sum_{m=1}^{M} p(c=m|x^{(i)},\theta^t) \log p(x^{(i)}, c=m|\theta')$

→ When we max. $Q(\theta^t, \theta)$ wrt θ by setting $\theta^{t+1} = \arg\max_{\theta} Q(\theta^t, \theta)$,

$$Q(\theta^t, \theta^{t+1}) - Q(\theta^t, \theta^t) \geq 0 \Rightarrow L(\theta^{t+1}) - L(\theta^t) \geq 0$$

- For the continuous case, the auxiliary function Q becomes.

$$Q(\theta^t, \theta') = \int p(\underline{z}|\underline{x}, \theta^t) \log p(\underline{x}, \underline{z}|\theta') \, d\underline{z}$$

Negative variational free energy $F(q(\underline{z}), \theta)$

- The negative variational free energy / evidence lower bound (ELBO) is a lower bound for the likelihood $\log p(\underline{x}|\theta)$, and is defined to be

$$F(q(\underline{z}), \theta) = \log p(\underline{x}|\theta) - KL(q(\underline{z}) \| p(\underline{z}|\underline{x}, \theta)) = \log p(\underline{x}|\theta) - \int q(\underline{z}) \log \frac{q(\underline{z})}{p(\underline{z}|\underline{x}, \theta)} d\underline{z}$$

* KL-divergence is non-negative, so $F(q(\underline{z}), \theta) \leq \log p(\underline{x}|\theta)$, w/ equality iff $q(\underline{z}) = p(\underline{z}|\underline{x}, \theta)$

- The variational free energy $F(q(\underline{z}), \theta)$ can be expressed in another form. Consider $\log p(\underline{x}|\theta)$.

$$\log p(\underline{x}|\theta) = \log p(\underline{x}|\theta) \int q(\underline{z}) \, d\underline{z} = \int q(\underline{z}) \log \frac{p(\underline{x}|\theta) p(\underline{z}|\underline{x}, \theta)}{p(\underline{z}|\underline{x}, \theta)} d\underline{z} = E_{q(\underline{z})}\left[\log \frac{p(\underline{x}, \underline{z}|\theta)}{p(\underline{z}|\underline{x}, \theta)}\right]$$

$$= E_{q(\underline{z})}\left[\log \frac{p(\underline{x}, \underline{z}|\theta)}{q(\underline{z})}\right] + E_{q(\underline{z})}\left[\log \frac{q(\underline{z})}{p(\underline{z}|\underline{x}, \theta)}\right] = E_{q(\underline{z})}\left[\log \frac{p(\underline{x}, \underline{z}|\theta)}{q(\underline{z})}\right] + KL(q(\underline{z}) \| p(\underline{z}|\underline{x}, \theta))$$

$$\therefore F(q(\underline{z}), \theta) = E_{q(\underline{z})}[\log p(\underline{x}, \underline{z}|\theta)] - E_{q(\underline{z})}[q(\underline{z})] = \int q(\underline{z}) \log p(\underline{x}, \underline{z}|\theta) \, d\underline{z} - \int q(\underline{z}) \log q(\underline{z}) \, d\underline{z}$$

* $F(q(\underline{z}), \theta) = \log p(\underline{x}|\theta) - KL(q(\underline{z}) \| p(\underline{z}|\underline{x}, \theta))$ is useful for the E-step (one term const. wrt $q(\underline{z})$)

$F(q(\underline{z}), \theta) = \int q(\underline{z}) \log p(\underline{x}, \underline{z}|\theta) \, d\underline{z} - \int q(\underline{z}) \log q(\underline{z}) \, d\underline{z}$ is useful for the M-step (one term const. wrt θ)

Expectation maximisation (EM) algorithm

- The EM algorithm is an iterative method to find (local) MLE of parameters θ in statistical models which involve unobserved latent variables z
- Since the negative variational free energy $F(q(\underline{z}),\theta)$ is a lower bound of the log-likelihood $\log p(\underline{x}|\theta)$, optimising $F(q(\underline{z}),\theta)$ gives a better and better lower bound for $\log p(\underline{x}|\theta)$
- The $(t+1)$-th iteration of the EM algorithm is as follows:
  - (i) Initialise $q^t(\underline{z}) = p(\underline{z}|\underline{x},\theta^t) \rightarrow L(\theta^t) = F(q^t(\underline{z}),\theta^t)$
  - (ii) M-step: For fixed $q^t(\underline{z})$, max. the lower bound $F(q^t(\underline{z}),\theta)$ wrt θ

$$\theta^{t+1} = \underset{\theta}{\operatorname{argmax}}\, F(q^t(\underline{z}),\theta) = \underset{\theta}{\operatorname{argmax}} \int q^t(\underline{z}) \log p(\underline{x},\underline{z}|\theta)\, d\underline{z} - \int q^t(\underline{z}) \log q^t(\underline{z})\, d\underline{z} = \underset{\theta}{\operatorname{argmax}} \int q^t(\underline{z}) \log p(\underline{x},\underline{z}|\theta)\, d\underline{z}$$

  - (iii) E-step: For fixed θ^{t+1}, max the lower bound $F(q(\underline{z}),\theta^{t+1})$ wrt $q(\underline{z})$

$$q^{t+1}(\underline{z}) = \underset{q}{\operatorname{argmax}}\, F(q(\underline{z}),\theta^{t+1}) = \underset{q}{\operatorname{argmax}} \log p(\underline{x}|\theta^{t+1}) - KL(q(\underline{z}) \,||\, p(\underline{z}|\underline{x},\theta^{t+1})) = p(\underline{z}|\underline{x},\theta^{t+1})$$

  We iterate until convergence — find a local max. of likelihood (which depends on initial θ_0)
- Note that each iteration cannot decrease likelihood as

$$\log p(\underline{x}|\theta^t) \overset{\text{E-step}}{=} F(q^t(\underline{z}),\theta^t) \overset{\text{M-step}}{\leq} F(q^t(\underline{z}),\theta^{t+1}) \overset{\text{lower bound}}{\leq} \log p(\underline{x}|\theta^{t+1})$$

* If we are unable to set $q(\underline{z}) = p(\underline{z}|\underline{x},\theta)$ (e.g. $p(\underline{z}|\underline{x},\theta)$ is not a Gaussian), then

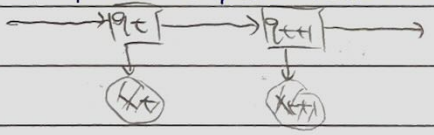
$$\log p(\underline{x}|\theta^t) \geq F(q^t(\underline{z}),\theta^t) \leq F(q^t(\underline{z}),\theta^{t+1}) \leq \log p(\underline{x}|\theta^{t+1})$$

i.e. there is no guarantees of not decreasing the log-likelihood

Sequence models

Hidden Markov models (HMM)

- HMMs are generative models w/ dynamic Bayesian networks of the following form



CIs: $p(q_t|q_{1:t-1}) = p(q_t|q_{t-1})$; $p(x_t|x_{1:t-1},q_{1:t}) = p(x_t|q_t)$

The joint distribution over all latent variables $q_{1:T}$ and observations $x_{1:T}$ is therefore

$$p(q_{1:T},x_{1:T}) = \prod_{t=1}^{T} p(q_t|q_{t-1})\, p(x_t|q_t) \quad \text{where } p(q_1|q_0) = p(q_1)$$

The likelihood of the data can be computed by marginalising over all possible state seq. $q \in Q^T$.

$$p(x_{1:T}) = \sum_{q \in Q^T} p(q)\, p(x_{1:T}|q) = \sum_{q \in Q^T} \prod_{t=1}^{T} p(q_t|q_{t-1})\, p(x_t|q_t)$$

- HMMs have emitting states that produce the observation seq. $x_{1:T}$ and non-emitting states used to define valid start and end states [N states, S_1, S_N non-emitting, $S_2,\ldots S_{N-1}$ emitting]
- HMMs have the following parameters θ:
  - transition matrix A — $a_{ij} = p(q_t = s_j | q_{t-1} = s_i)$
  - state obs prob. $\{b_2(x_t)\ldots b_{N-1}(x_t)\}$ — $b_j(x_t) = p(x_t|q_t = s_j)$

  ] usually trained using EM
- HMMs can be used as a generative classifier:

$$\hat{w} = \underset{w}{\operatorname{argmax}}\, p(q = w|x_{1:T}) = \underset{w}{\operatorname{argmax}}\, p(w)\, p(x_{1:T}|w)$$

↑ we need to compute this quickly

Viterbi algorithm

- The Viterbi algorithm is used to find an approx (lower bound) to the likelihood $p(x_{1:T})$ and the best state-seq (path) to a state / end of entire seq.
- We approx. the likelihood as the following using $\sum_q f(q) \approx \max_q f(q)$

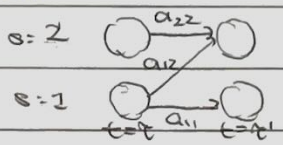
$$p(x_{1:T}) = \sum_{q \in Q^T} p(x_{1:T}, q) \approx p(x_{1:T}, \hat{q})$$

where $\hat{q}$ is the best state seq. through the discrete state space

$$\hat{q} = \{\hat{q}_0, \dots q_{T+1}\} = \arg\max_{q \in Q^T} p(x_{1:T}, q)$$

- There is an associated max log prob. for reaching state j at time t, $\phi_j(t)$.

Consider the following sub-graph



There are two paths of reaching state 2 at time τ':

↳ state 1 of time τ + transition a_{12} + observation $x_{\tau'} \rightarrow \phi_1(\tau) + \log a_{12} + \log b_2(x_{\tau'})$

↳ state 2 of time τ + transition a_{22} + observation $x_{\tau'} \rightarrow \phi_2(\tau) + \log a_{22} + \log b_2(x_{\tau'})$

We select the path w/ the max log prob: $\phi_2(\tau') = \max\{\phi_1(\tau) + \log a_{12}, \phi_2(\tau) \log a_{22}\} + \log b_2(x_{\tau'})$

- The Viterbi algorithm is as follows:

↳ (i) Initialization: $\phi_1(0) = 0$, $\phi_1(t) = \text{LZERO}$ $1 \le t \le T$, $\phi_j(0) = \text{LZERO}$. $1 < j < N$ LZERO = log 0

↳ (ii) Recursion: for $t = 1, \dots, T$ do

for $j = 2, \dots, N-1$ do

$$\phi_j(t) = \max_{1 < k < N} \{\phi_k(t-1) + \log a_{kj}\} + \log b_j(x_t)$$

end for

end for

↳ (iii) Termination: $\log p(x_{1:T}, \hat{q}) = \max_{1 < k < N} \{\phi_k(T) + \log a_{kN}\}$.

* We store all the previous best states to yield the best state seq. $\hat{q}$.

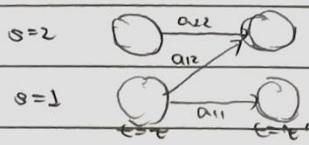
Forward algorithm

- The forward algorithm is used to find all state-seq (path) to a state
- Define the LAdd operator as follows:

$$LAdd(a,b) = \log(\exp(a)+\exp(b)) \rightarrow \exp(LAdd(a,b)) = \exp(a)+\exp(b)$$

- The total path to state j at time t is $\alpha_j(t) = \log p(x_1,\dots x_t, q_t = s_j)$

Consider the following sub-graph



There are two paths of reaching state 2 at time t':

↳ state 1 at time t + transition a_{12} + observation $x_{t'} \rightarrow \alpha_1(t) + \log a_{12} + \log b_2(x_{t'})$

↳ state 2 at time t + transition a_{22} + observation $x_{t'} \rightarrow \alpha_2(t) + \log a_{22} + \log b_2(x_{t'})$

The total path is the LAdd of all paths: $\alpha_2(t') = LAdd(\alpha_1(t)+\log a_{12}, \alpha_2(t)+\log a_{22}) + \log b_2(x_{t'})$

- The forward algorithm is as follows:

↳ (i) Initialization: $\alpha_1(0) = 0$, $\alpha_1(t) = LZERO$, $1 \le t \le T$, $\alpha_j(0) = LZERO$ $1 < j < N$ $LZERO = \log 0$

↳ (ii) Recursion: for $t = 1,\dots,T$ do

for $j = 2,\dots,N-1$ do

$$\alpha_j(t) = \log\left(\sum_k \exp(\alpha_j(t-1) + \log a_{kj})\right) + \log b_j(x_t)$$

end for

end for

Forward-backward algorithm.

- The forward-backward algorithm is used to find the likelihood $p(x_{1:T})$ and the posterior req. for EM (to train the parameters θ). $p(q_t = s_j \mid x_{1:T})$
- The forward probability $\alpha_j(t)$ is given by

$$\alpha_j(t) = \log p(x_1,\dots,x_t, q_t = s_j) = \log\left(\sum_{k=1}^{N} \exp(\alpha_k(t-1) + \log a_{kj})\right) + \log b_j(x_t)$$

The backward probability $\beta_j(t)$ is given by

$$\beta_j(t) = \log p(x_{t+1},\dots,x_T \mid q_t = s_j) = \log\left(\sum_{k=1}^{N} \exp(\beta_k(t+1) + \log a_{jk} + \log b_k(x_{t+1}))\right)$$

Adding the expressions give:

$$\alpha_j(t) + \beta_j(t) = \log p(x_1,\dots,x_t, q_t = s_j)$$

- The posterior req. for EM $p(q_t = s_j \mid x_{1:T})$ is given by

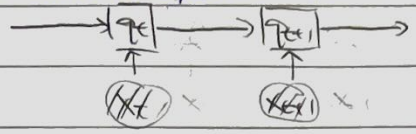
$$p(q_t = s_j \mid x_{1:T}) = \frac{1}{Z}\exp(\alpha_j(t) + \beta_j(t))$$

where the normalization const Z is the likelihood $p(x_{1:T})$ $Z = \sum_{j=1}^{N} \exp(\alpha_j(t) + \beta_j(t)) = p(x_{1:T})$

* we just need α_j to find Z, $Z = \sum_{j=1}^{N} \exp(\alpha_j(T))$

Maximum entropy Markov model (MEMM)

- MEMMs are discriminative models w/ dynamic Bayesian networks of the following form:



CIs: $p(q_t|q_{0:t-1}, x_{1:t}) = p(q_t|q_{t-1}, x_t)$

The posterior prob. of the seq. of latent variables $q_{0:T}$ (given observations $x_{1:T}$) is therefore

$$p(q_0:q_T|x_{1:T}) = \prod_{t=1}^{T} p(q_t|q_{t-1}, x_t)$$

where the posterior prob. of a single state q_t is given by

$$p(q_t|q_{t-1}, x_t) = \frac{1}{Z_t} \exp\left(\sum_{i=1}^{D} \lambda_i f_i(q_t, q_{t-1}, x_t, t)\right)$$

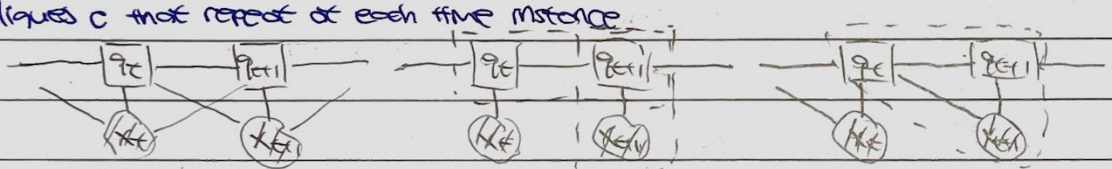
- The MEMM could be extended to the complete seq. w/ a global normalisation term

$$p(q_{0:T}|x_{1:T}) = \frac{1}{Z} \exp\left(\sum_{i=1}^{D} \lambda_i f_i(q_{0:T}, x_{1:T})\right)$$

* PROBLEM is it is diff. to handle large no. of possible features (of varying length) → many λ.

conditional random fields (CRF).

- CRFs are discriminative models w/ dynamic Markov networks that have sets of cliques C that repeat at each time instance.



- The posterior prob. of the seq. of latent variables $q_{0:T}$ (given observations $x_{1:T}$) is given by

$$p(q_{0:T}|x_{1:T}) = \frac{1}{Z} \exp\left(\sum_{t=1}^{T} \sum_{c \in C} \underline{\lambda}_c^T \underline{f}(q_{ct}, x_{1:T}, t)\right)$$

where $\underline{\lambda}_c$ are time-independent parameters associated w/ clique C

and $\underline{f}(q_{ct}, x_{1:T}, t)$ are time-dependent features extracted from clique C w/ time-dependent label seq. q_{ct}

- The simple linear chain CRF has two cliques, $C = \{C_1, C_2\}$. $C_1 = \{q_t, q_{t-1}\}$ (transition), $C_2 = \{q_t, x_t\}$ (acoustics).

$$p(q_{0:T}|x_{1:T}) = \frac{1}{Z} \exp\left(\sum_{t=1}^{T} \sum_{c \in C} \underline{\lambda}_c^T \underline{f}(q_{ct}, x_{1:T}, t)\right) = \frac{1}{Z} \exp\left(\sum_{t=1}^{T} \left(\sum_{i=1}^{D_e} \lambda_i^e f_i(q_t, q_{t-1}) + \sum_{i=1}^{D_a} \lambda_i^a f_i(q_t, x_t)\right)\right)$$

the linear chain CRF has one clique $C = \{C_1\}$, $C_1 = \{q_t, q_{t-1}, x_t\}$,

$$p(q_{0:T}|x_{1:T}) = \frac{1}{Z} \exp\left(\sum_{t=1}^{T} \sum_{c \in C} \underline{\lambda}_c^T \underline{f}(q_{ct}, x_{1:T}, t)\right) = \frac{1}{Z} \exp\left(\sum_{t=1}^{T} \left(\sum_{i=1}^{D} \lambda_i f_i(q_t, q_{t-1}, x_t)\right)\right)$$

- We can use an equivalent to the forward-backward algorithm to compute the normalisation term Z.

$$Z = \sum_{q \in Q^T} \exp\left(\sum_{t=1}^{T} \sum_{c \in C} \underline{\lambda}_c^T \underline{f}(q_{ct}, x_{1:T}, t)\right)$$

- We train CRFs via supervised learning, w/ training observation seq. $x_{1:T}$ and label seq. $y_{1:T}$.

The model parameters $\underline{\lambda}$ are found via gradient descent, s.t.

$$\hat{\lambda} = \arg\max_{\lambda} p(y_{1:T}|x_{1:T}, \lambda) = \arg\max_{\lambda} \frac{1}{Z} \exp\left(\sum_{i=1}^{D} \lambda_i f_i(x_{1:T}, y_{1:T})\right)$$

Support vector machines

Maximum margin classifier and soft margin classifier

Constrained optimisation

- Consider the constrained optimisation problem w/ equality constraints

$$\max \; f(x,y) \quad \text{subject to } g(x,y)=0$$

The soln (x^*,y^*) must satisfy conditions $\nabla_{x,y} f(x,y) = -\lambda \nabla_{x,y} g(x,y)$ and $g(x,y)=0$.

- We can find the soln (x^*,y^*) by optimising the Lagrangian function $L(x,y,\lambda)$ wrt x, y, λ.

$$L(x,y,\lambda) = f(x,y) + \lambda g(x,y)$$

where $\lambda \neq 0$ is the Lagrangian multiplier (could be +ve or -ve)

- Now consider the constrained optimisation problem w inequality constraints

$$\max. \; f(x,y) \quad \text{subject to } g(x,y) \geq 0$$

We have two possibilities:

↳ (i) constraint is not active at the soln, i.e. $g(x,y)>0$

As w/ the unconstrained case, $\nabla_{x,y} f(x,y)=0$, and we have $\nabla_{x,y} L(x,y,\lambda)=0$ iff $\lambda=0$.

↳ (ii) Constraint is active at the soln, i.e. $g(x,y)=0$.

We have $\nabla_{x,y} f(x,y) = -\lambda \nabla_{x,y} g(x,y)$ and we req. $\lambda > 0$ (o/w we would always set $\lambda = -\infty$)

⇒ the soln satisfies the Karush-Kuhn-Tucker (KKT) conditions.

$$\nabla_{x,y} f(x,y) = -\lambda \nabla_{x,y} g(x,y), \quad \lambda g(x,y) = 0, \quad \lambda \geq 0, \quad g(x,y) \geq 0.$$

Binary classification problem

- Given a dataset $D = \{(\underline{x}_n, t_n)\}_{n=1}^N$, formed by pairs of input features $\underline{x}_n \in \mathbb{R}^N$ and target variables $t_n \in \{-1, 1\}$, we want to learn a classifier $y(\underline{x})$ s.t.

$$y(\underline{x}) \geq 0 \text{ if } t_n = 1 \qquad y(\underline{x}) < 0 \text{ if } t_n = -1.$$

The classifier makes the correct prediction on a new i/p $\underline{x}_n$ when $y(\underline{x}_n)\, t_n > 0$.

- The decision boundary is the set of i/ps for which $y(\underline{x}) = 0$.

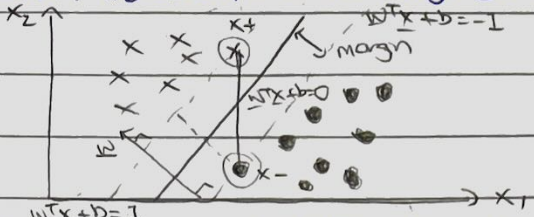
- A classification problem is linearly separable when a classifier w/ linear DB makes no mistakes on the training data. A linear classifier has the form

$$y(\underline{x}) = \underline{w}^T \underline{x} + b$$

where $\underline{w}$ is a vector orthogonal to the DB and b is the bias.

Maximum margin classifier (MMC)

- When the data is linearly separable, many possible $\underline{w}$ have zero training error
- The MMC is where we select a hyperplane whose distance to closest pt in each class, the margin, is maximal. The data pts on the margin are the support vectors.



* MMCs are fully determined by its support vectors (other data pts don't affect the classifier)

- Note that the DB of linear classifiers are scale invariant, i.e. $y(\underline{x})$ and $y(\underline{x})=cy(\underline{x})$ have identical DB → we can choose the scale s.t. for the +ve/-ve support vectors $\underline{x}_+/\underline{x}_-$,

$$y(\underline{x}_+)=\underline{w}^T\underline{x}_+ + b = +1 \qquad y(\underline{x}_-)=\underline{w}^T\underline{x}_- + b = -1 .$$

The magnitude of the margin is therefore

$$\frac{\underline{w}^T(\underline{x}_+ - \underline{x}_-)}{2\|\underline{w}\|} = \frac{(\underline{w}^T\underline{x}_+ + b - \underline{w}^T\underline{x}_- - b)}{2\|\underline{w}\|} = \frac{1}{\|\underline{w}\|}$$

→ The MMC has minimal $\|\underline{w}\|$ that has all data correctly classified, i.e. $t_n y(\underline{x}_n) \geq 1 \ \forall n$.

- Since maximising $\frac{1}{\|\underline{w}\|}$ is the same as minimising $\frac{1}{2}\|\underline{w}\|^2$, the optimisation problem is

$$\min_{\underline{w},b} \ \frac{1}{2}\|\underline{w}\|^2 \quad \text{subject to} \quad t_n(\underline{w}^T\underline{x}_n + b) \geq 1, \ n=1,\dots,N$$

i.e. we have a quadratic problem w/ linear constraints → there is a unique min.

Introducing the Lagrange multipliers $\{a_n \geq 0\}_{n=1}^N$ we have the objective.

$$L(\underline{w},b,\underline{a}) = \frac{1}{2}\|\underline{w}\|^2 - \sum_{n=1}^N a_n\{t_n(\underline{w}^T\underline{x}_n + b) - 1\}$$

(negative because we min.)

We can then consider the equivalent dual problem:

$$\max_{a_n \geq 0}\left[\min_{\underline{w},b} L(\underline{w},b,\underline{a})\right]$$ ← max. the soln to the inner min. problem

For the inner minimisation problem, we req.

$$\frac{\partial L}{\partial \underline{w}} L(\underline{w},b,\underline{a}) = \underline{w} - \sum_{n=1}^N a_n t_n \underline{x}_n = 0 \quad \rightarrow \quad \underline{w} = \sum_{n=1}^N a_n t_n \underline{x}_n$$

$$\frac{\partial L}{\partial b} L(\underline{w},b,\underline{a}) = -\sum_{n=1}^N a_n t_n = 0 \quad \rightarrow \quad \sum_{n=1}^N a_n t_n = 0 .$$

Substituting into the expression for $L(\underline{w},b,\underline{a})$, we have

$$\max_{a_n \geq 0} \sum_{n=1}^N a_n - \frac{1}{2}\sum_{n=1}^N\sum_{m=1}^N a_n a_m t_n t_m \underline{x}_n^T\underline{x}_m \quad \text{subject to} \quad \sum_{n=1}^N a_n t_n = 0$$

found via numerical methods, $O(N^2)$–$O(N^3)$ typically $N \leq 50000$

KKT conditions req. $a_n(t_n y(\underline{x}_n) - 1) = 0$, $a_n \geq 0$, $t_n y(\underline{x}_n) - 1 \geq 0 \ \forall n$.

$\therefore a_n > 0 \rightarrow t_n y(\underline{x}_n) = 1$ i.e. they are support vectors.

- Let $S = \{n : a_n > 0\}$ denote the set of indices of support vectors.

The bias can be obtained from the set of constraints $\{t_n y(\underline{x}_n) - 1 : n \in S\}$.

$$b = \frac{1}{|S|}\sum_{n\in S}\left\{t_n - \sum_{m\in S} a_m t_m \underline{x}_m^T\underline{x}_n\right\}$$

where we average the soln for b given by each constraint for numerical stability.

- After training, predictions can be made using.

$$y(\underline{x}) = \sum_{n\in S} a_n t_n \underline{x}_n^T\underline{x} + b$$

($\underline{w}$; only the support vectors have non-zero contribution)

Soft margin classifier (SMC) / Support vector classifier (SVC)

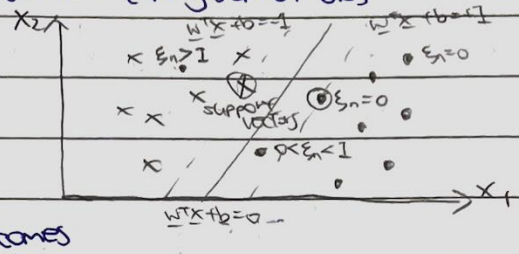
- Sometimes, allowing for mistakes in the training data produces better classifiers
  → trade-off btwn margin size and no. of training errors (allow training errors → ↑ margin size)
- Errors are allowed by introducing slack variables $\{\xi_n\}_{n=1}^N$ in the constraints.

$$t_n(\underline{w}^T\underline{x}_n + b) \geq 1 - \xi_n \,, \quad \xi_n \geq 0 \quad n = 1, \ldots, N$$

points w/ $\xi_n = 0$: correctly classified – on the margin or beyond

points w/ $0 \leq \xi_n \leq 1$: lie inside the margin, but on the correct side of DB

points w/ $\xi_n > 1$: misclassified (wrong side of DB)



- The optimisation now becomes

$$\min_{\underline{w},b} \tfrac{1}{2}\|\underline{w}\|^2 + C\sum_{n=1}^N \xi_n \quad \text{subject to } t_n(\underline{w}^T\underline{x}_n + b) \geq 1 - \xi_n \,, \ \xi_n \geq 0 \ \ n = 1, \ldots, N$$

↓C → ↑ size of margin
↑C → ↓ size of margin
C=∞ → MMC.

where $C > 0$ controls the trade-off btwn the slack variable penalty and the margin

i.e. we have a quadratic problem w/ linear constraints → there is a unique min.

Introducing Lagrangian multipliers $\boxed{\{a_n \geq 0\}_{n=1}^N, \{\mu_n \geq 0\}_{n=1}^N}$, we have the objective

$$L(\underline{w}, b, \underline{a}, \underline{\xi}, \underline{\mu}) = \tfrac{1}{2}\|\underline{w}\|^2 + C\sum_{n=1}^N \xi_n - \sum_{n=1}^N a_n\{t_n(\underline{w}^T\underline{x}_n + b) - 1 + \xi_n\} - \sum_{n=1}^N \mu_n\xi_n$$

We can then consider the equivalent dual problem

$$\max_{\underline{a} \geq 0} \left[\min_{\underline{w},b,\underline{\xi}} L(\underline{w}, b, \underline{a}, \underline{\xi}, \underline{\mu}) \right]$$

For the inner minimisation problem, we req.

$$\frac{\partial}{\partial \underline{w}} L(\underline{w}, b, \underline{a}, \underline{\xi}, \underline{\mu}) = \underline{w} - \sum_{n=1}^N a_n t_n \underline{x}_n = 0 \quad \rightarrow \quad \underline{w} = \sum_{n=1}^N a_n t_n \underline{x}_n$$

$$\frac{\partial}{\partial b} L(\underline{w}, b, \underline{a}, \underline{\xi}, \underline{\mu}) = -\sum_{n=1}^N a_n t_n = 0 \quad \rightarrow \quad \sum_{n=1}^N a_n t_n = 0$$

$$\frac{\partial}{\partial \xi_n} L(\underline{w}, b, \underline{a}, \underline{\xi}, \underline{\mu}) = C - a_n - \mu_n = 0 \quad \rightarrow \quad \boxed{\{a_n = C - \mu_n\}_{n=1}^N}$$

Substituting into the expression for $L(\underline{w}, b, \underline{a}, \underline{\xi}, \underline{\mu})$, we have

$$\boxed{\max_{0 \leq a_n \leq C}} \sum_{n=1}^N a_n - \frac{1}{2}\sum_{n=1}^N\sum_{m=1}^N a_n a_m t_n t_m \underline{x}_n^T \underline{x}_m \quad \text{subject to } \sum_{n=1}^N a_n t_n = 0$$

KKT conditions req. $a_n(t_n y(\underline{x}_n) - 1 + \xi_n) = 0$, $\mu_n \xi_n = 0$, $a_n \geq 0$, $\mu_n \geq 0$, $t_n(\underline{w}^T\underline{x}_n + b) \geq 1 - \xi_n \ \forall n$.

$\therefore$ $a_n > 0$ ($0 < a_n \leq C$), $\mu_n > 0$ → $\xi_n = 0$ → $t_n y(\underline{x}_n) = 1$ i.e. they are support vectors.

$a_n > 0$ ($a_n = C$), $\mu_n = 0$ → pt. lies within the margin boundaries

- Let $M = \{n : 0 < a_n < C\}$ denote the set of indices of support vectors (and $S = \{n : a_n > 0\}$ as before)

The bias can be obtained from the set of constraints $\{t_n y(\underline{x}_n) = 1 - \xi_n : n \in S\}$ w/ $\xi_n = 0$ (i.e. $0 < a_n < C$)

$$b = \frac{1}{|M|}\sum_{n \in M}\left\{t_n - \sum_{m \in S} a_m t_m \underline{x}_m^T \underline{x}_n\right\}$$

where we average the soln for b given by each constraint for numerical stability

- After training, predictions can be made using

$$y(\underline{x}) = \sum_{n \in S} a_n t_n \underline{x}_n^T \underline{x} + b$$

Non-linear max-margin classifiers and multi-class max-margin classifiers

Non-linear max-margin classifiers / support vector machines (SVM)

- For classification problems w/ non-linear DB, we replace each feature vector $\underline{x}_n$ w/ a new one formed by applying non-linear functions to the original vector $\phi(\underline{x}_n) = \begin{bmatrix} \phi_1(\underline{x}_n) \\ \vdots \\ \phi_M(\underline{x}_n) \end{bmatrix}$
- The new classifier has a non-linear DB and has the form

$$y(\underline{x}) = \underline{w}^T \phi(\underline{x}) + b$$

The optimization objective now becomes

$$\max_{a \geq 0} \sum_{n=1}^{N} a_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} a_n a_m t_n t_m \phi(\underline{x}_n)^T \phi(\underline{x}_m) \quad \text{subject to} \quad \sum_{n=1}^{N} a_n t_n = 0$$

- For training, we only need the gram matrix $\underline{\underline{K}}$, (matrix of dot products), given by:

$$k_{n,m} = \phi(\underline{x}_n)^T \phi(\underline{x}_m) \quad \Leftrightarrow \quad \underline{\underline{K}} = \underline{\underline{\Phi}} \underline{\underline{\Phi}}^T$$

where $\underline{\underline{\Phi}} = \begin{bmatrix} \phi(\underline{x}_1) \\ \vdots \\ \phi(\underline{x}_N) \end{bmatrix}$

After training, predictions can be made also using dot products

$$y(\underline{x}) = \sum_{n \in S} a_n t_n \phi(\underline{x})^T \phi(\underline{x}_n) + b$$

* Cost of computing the dot product scales linearly w/ o/p dim d of feature mapping $\phi(\cdot)$

Kernel functions

- So far, the gram matrix $\underline{\underline{K}}$ is computed using the following two steps:
  - ↳ (i) map i/ps $\underline{x}_n, \underline{x}_m$ to feature-space to obtain $\phi(\underline{x}_n)$ and $\phi(\underline{x}_m)$ (feature expansion)
  - ↳ (ii) compute dot product $\phi(\underline{x}_n)^T \phi(\underline{x}_m)$
- Instead, we can use kernel function $k(\cdot,\cdot)$ to implicitly map $\underline{x}_n$ and $\underline{x}_m$ to $\phi(\underline{x}_n)^T \phi(\underline{x}_m)$.

$$k_{n,m} = k(\underline{x}_n, \underline{x}_m) = \phi(\underline{x}_n)^T \phi(\underline{x}_m)$$

* $k(\cdot,\cdot)$ computes dot products in feature-space w/o explicitly performing the feature-expansion.

- Common kernel functions include:
  - ↳ Linear: $k(\underline{x}_n, \underline{x}_m) = \underline{x}_n^T \underline{x}_m$
  - ↳ Polynomial: $k(\underline{x}_n, \underline{x}_m) = (1 + \underline{x}_n^T \underline{x}_m)^d$
  - ↳ Gaussian: $k(\underline{x}_n, \underline{x}_m) = \exp(-\frac{1}{2\sigma^2} \|\underline{x}_n - \underline{x}_m\|^2)$ [σ controls smoothness of DB]

$\sigma \to 0$, $\underline{\underline{K}} \to$ diag (rank($\underline{\underline{K}}$)=N) → wiggly DB
$\sigma \to \infty$, $\underline{\underline{K}}$ has equal entries (rank($\underline{\underline{K}}$)=1) → smooth DB

- $k(\cdot,\cdot)$ is a valid kernel iff there is a map $\phi(\cdot)$ s.t. $k(\underline{x}_n, \underline{x}_m) = \phi(\underline{x}_n)^T \phi(\underline{x}_m)$ for any $\underline{x}_n, \underline{x}_m$.

In particular, iff for any data set, the gram matrix $\underline{\underline{K}}$ obtained w/ $k(\cdot,\cdot)$ satisfies $\underline{\underline{K}} = \underline{\underline{\Phi}} \underline{\underline{\Phi}}^T$.

- Mercer's condition states that $k(\cdot,\cdot)$ is a valid kernel iff the following are satisfied:
  - ↳ $k(\cdot,\cdot)$ is symmetric, i.e. $k(\underline{x}_n, \underline{x}_m) = k(\underline{x}_m, \underline{x}_n)$
  - ↳ Any gram matrix $\underline{\underline{K}}$ obtained w/ $k(\cdot,\cdot)$ is PSD, i.e. $\underline{g}^T \underline{\underline{K}} \underline{g} \geq 0$ for any $\underline{g}, \{\underline{x}_n\}_{n=1}^N$ (or for any subset of data pts)

- eg: $\underline{x}_n \in \mathbb{R}^2$, $\phi: \mathbb{R}^2 \to \mathbb{R}^5$, $\phi(\underline{x}_n) = [1, \sqrt{2}x_{n,1}, \sqrt{2}x_{n,2}, \sqrt{2}x_{n,1}x_{n,2}, x_{n,1}^2, x_{n,2}^2]^T$

$$k(\underline{x}_n, \underline{x}_m) = \phi(\underline{x}_n)^T \phi(\underline{x}_m) = [1, \sqrt{2}x_{n,1}, \sqrt{2}x_{n,2}, \sqrt{2}x_{n,1}x_{n,2}, x_{n,1}^2, x_{n,2}^2] [1, \sqrt{2}x_{m,1}, \sqrt{2}x_{m,2}, \sqrt{2}x_{m,1}x_{m,2}, x_{m,1}^2, x_{m,2}^2]^T$$

$$= 1 + 2x_{n,1}x_{m,1} + 2x_{n,2}x_{m,2} + 2x_{n,1}x_{n,2}x_{m,1}x_{m,2} + x_{n,1}^2 x_{m,1}^2 + x_{n,2}^2 x_{m,2}^2$$

$$= (1 + x_{n,1}x_{m,1} + x_{n,2}x_{m,2})^2 = (1 + \underline{x}_n^T \underline{x}_m)^2$$

The kernel trick

- The kernel trick is that any algorithm that operates on the i/ps $\{\underline{x}_n\}_{n=1}^N$ by only using dot products can be implemented using kernels

- e.g. Kernelised least squares regression

The regularised least squares objective is given by

$$\text{cost}(\underline{w}, \{(\phi(\underline{x}_n), t_n)\}_{n=1}^N) = \frac{1}{2}(\underline{\Phi}\underline{w} - \underline{t})^T(\underline{\Phi}\underline{w} - \underline{t}) + \frac{\lambda}{2}\underline{w}^T\underline{w} = \frac{1}{2}\underline{w}^T\underline{\Phi}^T\underline{\Phi}\underline{w} + \frac{1}{2}\underline{t}^T\underline{t} - \underline{t}^T\underline{\Phi}\underline{w} + \frac{\lambda}{2}\underline{w}^T\underline{w}$$

Differentiating wrt $\underline{w}$, we have

$$\frac{\partial}{\partial \underline{w}}\text{cost}(\underline{w}, \{(\phi(\underline{x}_n), t_n)\}_{n=1}^N) = \underline{\Phi}^T\underline{\Phi}\underline{w} - \underline{t}^T\underline{\Phi} + \lambda\underline{w} = 0 \rightarrow \underline{w} = \underline{\Phi}^T(\underline{t} - \underline{\Phi}\underline{w})/\lambda = \underline{\Phi}^T\underline{a}.$$

$$\Rightarrow \text{cost}(\underline{a}, \{(\phi(\underline{x}_n), t_n)\}_{n=1}^N) = \frac{1}{2}\underline{a}^T\underline{\Phi}\underline{\Phi}^T\underline{\Phi}\underline{\Phi}^T\underline{a} + \frac{1}{2}\underline{t}^T\underline{t} - \underline{t}^T\underline{\Phi}\underline{\Phi}^T\underline{a} + \frac{\lambda}{2}\underline{a}^T\underline{\Phi}\underline{\Phi}^T\underline{a}$$

Differentiating wrt $\underline{a}$, we have

$$\frac{\partial}{\partial \underline{a}}\text{cost}(\underline{a}, \{(\phi(\underline{x}_n), t_n)\}_{n=1}^N) = \underline{\Phi}\underline{\Phi}^T\underline{\Phi}\underline{\Phi}^T\underline{a} - \underline{\Phi}\underline{\Phi}^T\underline{t} + \lambda\underline{\Phi}\underline{\Phi}^T\underline{a} = 0 \rightarrow \underline{a} = (\underline{K} + \lambda\underline{I}_N)^{-1}\underline{t}$$

predictions are then given by

$$y(\underline{x}) = \underline{w}^T\phi(\underline{x}) = \underline{a}^T\underline{\Phi}\phi(\underline{x}) = \sum_{n=1}^N a_n k(\underline{x}_n, \underline{x})$$

Multiclass max-margin classifiers

- In one-vs-all classification, we train one classifier per class.

i.e. for K classes, we train class i vs class -i, w/ parameters $\underline{w}_i, b_i$, $i = 1, \ldots K$

→ We predict label for new data pt $\underline{x}^*$ using:

$$t^* = \underset{i \in \{1,\ldots,K\}}{\arg\max}\ \underline{w}_i^T\underline{x}_* + b_i$$

- there are a few issues w/ one-vs-all classification:

↳ classifiers trained independently → classifiers may have diff scales → comparing is meaningless.

↳ Imbalanced datasets → many more training data in a particular class → bias.

↳ Some problems cannot be solved w/ linear classifiers.

- In simultaneous learning of classifiers, we add constraints to enforce o/p of correct classifier to be larger than the others.

i.e. for K classes, for each training pt $(\underline{x}_n, t_n)$ we enforce $\underline{w}_{t_n}^T\underline{x}_n + b_{t_n} > \underline{w}_j^T\underline{x}_n + b_j$, $j \neq t_n$.

The new objective to optimise wrt $\{\underline{w}_i\}_{i=1}^K$ and $\{b_i\}_{i=1}^K$ are

$$\min_{\underline{w}, \underline{b}} \frac{1}{2}\sum_{i=1}^K \|\underline{w}_i\|^2 + C\sum_{n=1}^N \sum_{j \neq t_n} \xi_{nj} \text{ subject to } \underline{w}_{t_n}^T\underline{x}_n + b_{t_n} \geq \underline{w}_j^T\underline{x}_n + b_j + 1 - \xi_{nj}, \ \xi_{nj} \geq 0 \ \forall n, j \neq t_n.$$

→ we predict label for new data pt $\underline{x}^*$ using

$$t^* = \underset{k \in \{1,\ldots,K\}}{\arg\max}\ \underline{w}_i^T\underline{x}_* + b_i$$

- The main problem w/ simultaneous learning of classifiers is the very large computational cost.

→ In practice, one-vs-all classifier is more widely used.

Kernels for unstructured data

Kernels for strings

- Each data pt x is a string of characters from alphabet A, i.e. $x \in A^*$, where $*$ denotes the Kleene closure operation (set of concatenating zero or more characters)
- The main idea is as follows:
  - (i): Given a list of substrings $s_1, s_2 \ldots \in A^*$, each string x is encoded using the feature vector $\phi(x) = \begin{bmatrix} \phi_{s_1}(x) \\ \phi_{s_2}(x) \\ \vdots \end{bmatrix}$
  - (ii): For each substring s, the feature $\phi_s(x)$ is given by the no. of occurences of s in x ($\phi_s(x) = 0$ if s does not occur in x, o/w $\phi_s(x) > 0$)
  - (iii): The kernel btwn two strings x, x' is given by

$$k(x, x') = \phi(x)^T \phi(x') = \sum_{s \in \{s_1, s_2 \ldots\}} \phi_s(x)\, \phi_s(x')$$

- For the gap-weighted kernel, substrings s may not need to be continuous in x, but more gaps in the occurence → smaller value of $\phi_s(x)$. (e.g. $\phi_s(x) = \lambda^n$, $\lambda \in (0,1)$)
- The kernel can be computed efficiently for all possible substrings using DP
- e.g. $\phi_s(x) = \lambda^n$, $\lambda \in (0,1)$, $s = \{ca, ct, cr, ar, rt, ba, br\}$

	ϕ_{ca}	ϕ_{ct}	ϕ_{cr}	ϕ_{ar}	ϕ_{rt}	ϕ_{ba}	ϕ_{br}
ϕ(cat)	1	1	0	0	0	0	0
ϕ(cart)	1	λ^2	λ	1	1	0	0
ϕ(bar)	0	0	0	1	0	1	λ

→ $k(cart, cat) = 1 + \lambda^3$, $k(cat, bar) = 0$, $k(cart, bar) = 1$

- For the k-spectrum kernel, we consider all possible subsequences of length k
  - $\phi_s(x)$ is the no. of times that s exactly occurs in x
  - $k(x, x')$ computed fast using suffix tree to store all k-length seq. in x and x'.
- We have the bag-of-words kernel when $k = 1$, often applied to documents, (word strings)
- Co-occurrence of substrings is more informative for long substrings → k should be large
  Common occurences decrease as k increases → k should be small
  ⇒ we usually select intermediate values of k
- e.g. For DNA seq, we consider $s = \{AAA, AAG \ldots TTC, TTT\}$

Random-walk graph kernel

- For a graph $G = \{V, \mathcal{E}\}$, V is the node set and $\mathcal{E} = \{(i,j) : i,j \in V\}$ is the edge set.

The adjacency matrix $A \in \mathbb{R}^{|V| \times |V|}$ has entries $a_{ij} = \begin{cases} 1 & (i,j) \in \mathcal{E} \\ 0 & \text{o/w} \end{cases}$

- A k-length graph walk is defined as $w = \{v_1, \dots, v_{k+1}\}$ where $(v_i, v_{i+1}) \in \mathcal{E}$

The no. of k-length walks btwn nodes i and j is given by $[A^k]_{ij}$, since

$$[A^k]_{ij} = \sum_{s_1=1}^{|V|} \cdots \sum_{s_{k-1}=1}^{|V|} a_{i,s_1} a_{s_1,s_2} \dots a_{s_{k-1},j}$$

- For the random-walk graph kernel, we have

$\phi(G)$ where the r-th entry counts the no. k-length walks in G (sum of entries of A^k)

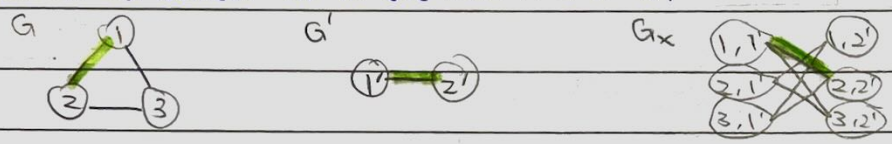
$k(G, G')$ counts the no. of common walks in graphs $G = (V, \mathcal{E})$ and $G' = (V', \mathcal{E}')$

- The kernel function can be computed efficiently using the direct product graph of G and G'.

$$G_\times = (V_\times, \mathcal{E}_\times)$$

where $V_\times = \{(a,b) : a \in V \text{ and } b \in V'\}$ and $\mathcal{E}_\times = \{((a,a'),(b,b')) : (a,b) \in \mathcal{E} \text{ and } (a',b') \in \mathcal{E}'\}$

- Each walk in $G_\times$ corresponds to one walk in G and G',



- The adjacency matrix of $G_\times$ is given by $A_\times = A \otimes A'$, where $\otimes$ is the Kronecker product

↳ $A \in \mathbb{R}^{n \times m}$, $B \in \mathbb{R}^{p \times q} \rightarrow A \otimes B \in \mathbb{R}^{np \times mq}$, $A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1m}B \\ \vdots & \ddots & \vdots \\ a_{n1}B & \cdots & a_{nm}B \end{bmatrix}$

For $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$, $B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$ $\rightarrow$ $A \otimes B = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12} \\ a_{11}b_{21} & a_{11}b_{22} & a_{12}b_{21} & a_{12}b_{22} \\ a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{11} & a_{22}b_{12} \\ a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{21} & a_{22}b_{22} \end{bmatrix}$

- Assuming that $\lambda > 0$ is sufficiently small to guarantee convergence, the kernel is given by

$$K(G, G') = \sum_{i,j=1}^{|V|} \left[\sum_{n=0}^{\infty} \lambda^n A_\times^n \right] = 1^T [I - \lambda A \otimes A']^{-1} 1 = 1^T x$$

where $x = [I - \lambda A \otimes A']^{-1} 1$ → we have $x = 1 + \lambda (A \otimes A') x$ → find x using the iteration.

$$x_{t+1} = 1 + \lambda (A \otimes A') x_t$$

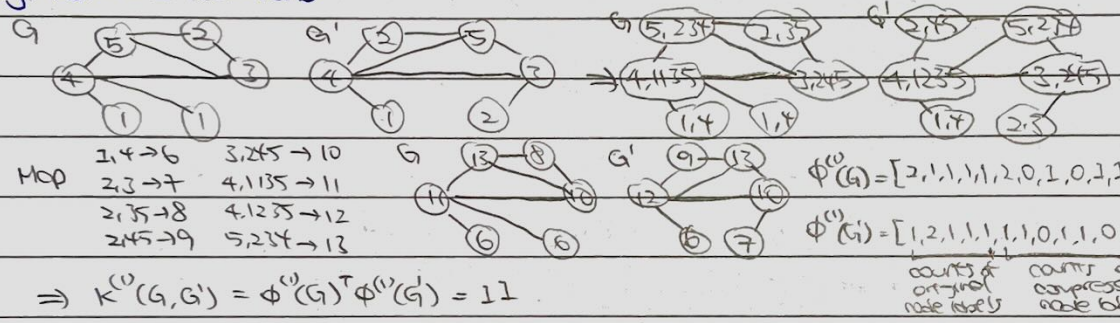
- The problem is that a walk can visit the same cycle again and again → small structural similarities can produce huge kernel values. Also it has high computational cost $O(|V|^3)$

Weisfeiler-Lehman graph kernel

- The Weisfeiler-Lehman graph kernel scales well w/ large graphs and allows for node labels.
- To compute the Weisfeiler-Lehman graph kernel, we repeat for M iterations for each graph in the dataset.
  - (i) Repeat for each node in the current graph
    1) create a set w/ labels of adjacent vertices
    2) sort the label set and add vertex label as a prefix
    3) compress resulting label rep. into a unique value
  - (ii) Assign to each graph node its resulting unique value as a new vertex label

Then we apply the bag-of-words kernel to the vertex labels obtained throughout all the iterations (inc. the initial labels).

- e.g.: For M=1 iterations



$\Rightarrow K^{(1)}(G,G') = \phi^{(1)}(G)^T \phi^{(1)}(G') = 11$

Fisher kernel

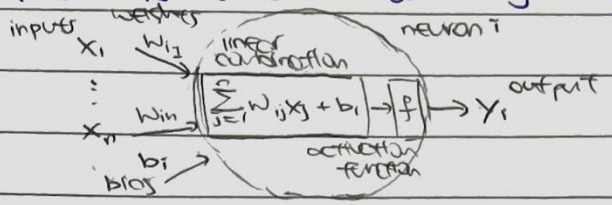
- For the Fisher kernel, we use a probabilistic generative model to obtain a fixed-length vector representation of complex structured data (structure data → fix-length score vectors)
- The Fisher kernel is computed as follows:
  - (i) Train generative model $p(x|\theta)$ on available i/p data $\{x_n\}_{n=1}^N$ e.g. by MLE
  - (ii) Define the Fisher score vector $\phi(x_n) = \nabla_\theta \log p(x_n|\theta)|_{\theta_{MLE}}$
  - (iii) The naive Fisher kernel is given by $K(x_n, x_m) = \phi(x_n)^T \phi(x_m)$

Deep learning

Neural network architecture

Perceptron

- The perceptron is the simplest NN architecture consisting of a single neuron

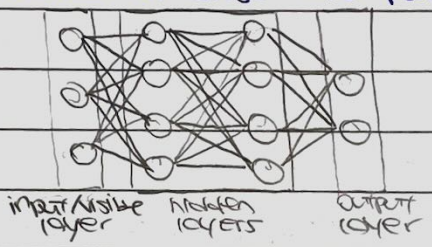


The mathematical model of the perceptron is given by

$$y_i = f\left(\sum_{j=1}^{n} w_{ij} x_j + b_i\right) = f(\underline{w}_i^T \underline{x} + b_i)$$

Multi-layer perceptron (MLP)

- MLPs (or FFNN) are a cascade of single-layer perceptrons



Depth of network: no. of hidden layers + output layer (1: shallow ; 2+: deep)

Width of layer: no. of nodes of the layer

- The width of the input layer depends on the dataset — no. of inputs
- The width of the output layer depends on the task:
  - ↳ N-class classification: N
  - ↳ Regression: 1.
- We typically set the width of the hidden layer to be between that of the input/output layers (this can be tuned using the validation set)
- The universal approx. theorem states that a FF network w/ a linear output layer and at least one hidden layer w/ any squashing activation function can approx. any cont. function

Activation functions.

- For hidden layers, we typically use the following activation functions:
  - ↳ ReLU (scale i/p to $[0,1]$)
  - ↳ Tanh (scale i/p to $[-1,1]$)
  - ↳ Sigmoid (scale i/p to $[0,1]$)
- For output layers, we typically use the following activation functions:
  - ↳ Linear (for regression)
  - ↳ Softmax (for multiclass classification)
  - ↳ Sigmoid (for binary classification)

* There are continuous equivalents to ReLU: SiLU, GeLU, ELU, SELU

Number of output regions,

- Consider a shallow neural network w/ D_i input nodes and D_h hidden nodes.

  Each output consists of D_h dimensional convex polytopes.

  The no. of output regions is given by

$$2^{D_i} < \sum_{j=0}^{D_i} \binom{D_h}{j} < 2^{D_h}$$

  For 1D input ($D_i = 1$), the no. of output regions is $1 < D_h$ (no. of kinks = D_h)

- Composing two shallow neural networks is equivalent to a deep neural network, and the no. of output regions is much greater

Neural network training

Loss function

- Learning the model parameters can be cast as an optimisation problem

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, L(\theta)$$

  where L is the loss function.

- Common loss functions for regression are:

  ↳ MSE: $\frac{1}{N}\sum_{n=1}^{N}(y_n - \hat{y}_n)^2$ ↳ MAE: $\frac{1}{N}\sum_{n=1}^{N}|y_n - \hat{y}_n|$

  ↳ RMSE: $\sqrt{\frac{1}{N}\sum_{n=1}^{N}(y_n - \hat{y}_n)^2}$

- Common loss functions for classification are:

  ↳ cross entropy: $-\sum_{c=1}^{C} p(y_c)\log q(\hat{y}_c)$ ↳ Accuracy: $\frac{\text{correct classifications}}{\text{total predictions}}$

  ↳ ROC-AUC

Gradient descent

- Batch size B is the no. of training samples to work through before the error is calculated and the model parameters are updated. (usually $B = 2^n$, $n \in \mathbb{Z}^+$)

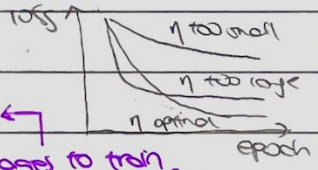
- The no. of epochs is the no. of complete passes through the training set.

  One epoch consists of $\frac{\lceil \text{training set size} \rceil}{\text{batch size}}$ batches / updates.

- We can update the model parameters iteratively via gradient descent

$$\theta^{(t+1)} = \theta^{(t)} - \eta \frac{1}{B}\sum_{i=1}^{B} \frac{dL_i(\theta^{(t)})}{d\theta^{(t)}}$$

  where η is the learning rate (most important hyperparameter!)



- $B = 1$: stochastic gradient descent SGD

  B = training set size: batch gradient descent

  $1 < B <$ training set size: mini-batch gradient descent

  η too small: take ages to train, may not even converge

  η too large: loss may oscillate over training epochs

* Having $B <$ training set size introduces randomness → add noise, can escape local minima, less computationally expensive.

Momentum

- We can incorporate momentum into gradient descent to smooth out the updates and allow the optimization process to maintain a direction based on past gradients.
- Instead of using the current gradient, we consider a weighted sum of the current gradient and previous gradient.

$$M^{(t+1)} = \beta M^{(t)} + (1-\beta)\frac{1}{B}\sum_{i=1}^{B}\frac{dL_i(\theta^{(t)})}{d\theta^{(t)}} \qquad \theta^{(t+1)} = \theta^{(t)} - \eta M^{(t+1)}$$

* $\beta = 0$: gradient descent w/o momentum.

- A variation of this is Nesterov accelerated momentum, which computes the gradient a step ahead.

$$M^{(t+1)} = \beta M^{(t)} + (1-\beta)\frac{1}{B}\sum_{i=1}^{B}\frac{dL_i(\theta^{(t)} - \eta M^{(t)})}{d\theta^{(t)}} \qquad \theta^{(t+1)} = \theta^{(t)} - \eta M^{(t+1)}$$

Adaptive moment estimation (Adam)

- We can do gradient descent using normalized gradients as follows.

$$M^{(t+1)} = \frac{dL(\theta^{(t)})}{d\theta^{(t)}}, \quad V^{(t+1)} = \left(\frac{dL(\theta^{(t)})}{d\theta^{(t)}}\right)^2 \qquad \theta^{(t+1)} = \theta^{(t)} - \eta\frac{M^{(t+1)}}{\sqrt{V^{(t+1)}}+\varepsilon}$$

* Note that $\frac{M^{(t+1)}}{\sqrt{V^{(t+1)}}+\varepsilon} = \text{sign}(M^{(t+1)})$ ← $\varepsilon = 10^{-6}$ for stability

- In Adam, we update the mean and of the squared gradients using momentum, then moderate/bias using an exponential decay → learning rate changes over time.

$$M^{(t+1)} = \beta_1 M^{(t)} + (1-\beta_1)\frac{1}{B}\sum_{i=1}^{B}\frac{dL_i(\theta^{(t)})}{d\theta^{(t)}} \qquad \tilde{M}^{(t+1)} = \frac{M^{(t+1)}}{1-\beta_1^{t+1}}$$

$$V^{(t+1)} = \beta_2 V^{(t)} + (1-\beta_2)\frac{1}{B}\sum_{i=1}^{B}\left(\frac{dL_i(\theta^{(t)})}{d\theta^{(t)}}\right)^2 \qquad \tilde{V}^{(t+1)} = \frac{V^{(t+1)}}{1-\beta_2^{t+1}}$$

$$\theta^{(t+1)} = \theta^{(t)} - \eta\frac{\tilde{M}^{(t+1)}}{\sqrt{\tilde{V}^{(t+1)}}+\varepsilon}$$

- A variation of this is AMSgrad, which uses the max. of past squared gradients rather than exponential average to update the parameters. → step size always decrease.

$$M^{(t+1)} = \beta_1 M^{(t)} + (1-\beta_1)\frac{1}{B}\sum_{i=1}^{B}\frac{dL(\theta^{(t)})}{d\theta^{(t)}}$$

$$V^{(t+1)} = \beta_2 V^{(t)} + (1-\beta_2)\frac{1}{B}\sum_{i=1}^{B}\left(\frac{dL_i(\theta^{(t)})}{d\theta^{(t)}}\right)^2 \qquad \hat{V}^{(t+1)} = \max(\hat{V}^{(t)}, V^{(t)})$$

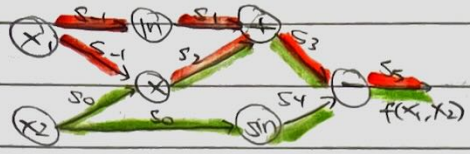
$$\theta^{(t+1)} = \theta^{(t)} - \eta\frac{M^{(t+1)}}{\sqrt{\hat{V}^{(t+1)}}+\varepsilon}$$

* Typically we choose $\beta_1 = 0.9$, $\beta_2 = 0.999$.

Automatic differentiation (AutoDiff)

- We can use reverse-mode AutoDiff (backpropagation) to compute the gradients req. for gradient descent.
- In AutoDiff, the function is decomposed into a seq. of elementary arithmetic operation and functions. This can be visualised using a computational graph.

eg: $f(x_1,x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$, $x_1=2$, $x_2=4$



$s_{-1} = x_1$ $\quad s_0 = x_2$
$s_1 = \ln(s_{-1})$ $\quad s_2 = s_{-1} s_0$
$s_3 = s_1 + s_2$ $\quad s_4 = \sin(s_0)$
$s_5 = s_3 - s_4$

Forward pass: $s_{-1}=2$, $s_0=4$, $s_1=0.69$, $s_2=8$, $s_3=8.69$, $s_4=-0.76$, $s_5=9.45$

Backward pass: $\frac{\partial s_5}{\partial s_3}=1$, $\frac{\partial s_5}{\partial s_4}=-1$, $\frac{\partial s_3}{\partial s_1}=1$, $\frac{\partial s_3}{\partial s_2}=1$, $\frac{\partial s_4}{\partial s_0}=\cos(s_0)=-0.65$,

$\frac{\partial s_1}{\partial s_{-1}}=\frac{1}{s_{-1}}=\frac{1}{2}$, $\frac{\partial s_2}{\partial s_{-1}}=s_0=4$, $\frac{\partial s_2}{\partial s_0}=s_{-1}=2$

$\rightarrow \frac{\partial f}{\partial x_1} = \frac{\partial s_5}{\partial s_3}\left(\frac{\partial s_3}{\partial s_1}\frac{\partial s_1}{\partial s_{-1}} + \frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{\partial s_{-1}}\right) = 4.5$, $\frac{\partial f}{\partial x_2} = \frac{\partial s_5}{\partial s_3}\frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{\partial s_0} + \frac{\partial s_5}{\partial s_4}\frac{\partial s_4}{\partial s_0} = 2.65$

* The forward pass propagates the function inputs down the computational graph
The backward pass computes the partial derivative of each node wrt its immediate inputs.

Feature scaling.

- Features (inputs) have varying magnitude, min-max range and units. We bring them to the same scale to improve model convergence by preventing certain features to overshadow others based solely on magnitude
- We can bring features to the same scale using normalisation or standardisation
  - ↳ Normalisation (min-max scaling): $x_{scaled,j} = \frac{x_j - \min(x)}{\max(x)-\min(x)} \in [0,1]$
  - ↳ Standardisation: $x_{standard,j} = \frac{x_j - \text{mean}(x)}{\text{std}(x)} \in (-\infty,\infty)$

He initialisation (weight initialisation)

- Consider a linear layer $y = \phi(\underline{\underline{A}}\underline{x} + \underline{b})$, w/ $\underline{x}\in\mathbb{R}^N$, $\underline{\underline{A}}\in\mathbb{R}^{M\times N}$, $\underline{b}\in\mathbb{R}^M$
We loosely have that $\sigma_y^2 \leq \frac{N}{2}\sigma_A^2\sigma_x^2$, so we initialise weights w/ $N(0,\sigma_A^2=\frac{2}{N})$
* For $\underline{z} = \underline{\underline{A}}\underline{x} + \underline{b}$, $\sigma_z^2 = N\sigma_A^2\sigma_x^2$; $z=\phi(x)$, $\sigma_z^2 \leq \frac{1}{2}\sigma_x^2$

Softmax stability

- Softmax is not numerically stable since exp(number) = very large number.
- We consider log softmax, $\log\text{softmax}(y)_k = y_k - \log\left(\sum_k \exp(y_k)\right) = y_k - \text{logsumexp}(y)$
where $\text{logsumexp}(y) = \text{logsumexp}(y-\max(y)) + \max(y)$ for stability.
* $\text{logsumexp}(y-C) = \log\left(\sum_k \exp(y_k - C)\right) = \log\left(\sum_k \frac{\exp(y_k)}{\exp(C)}\right) = \log\left(\frac{1}{\exp(C)}\sum_k \exp(y_k)\right) = \text{logsumexp}(y) - C$

Batch normalisation (BN)

- BN standardises the input of each layer within a NN across the mini batch.
  - ↳ 1) Compute the mean and variance: $\mu_B = \frac{1}{B}\sum_{i=1}^{B} x_i$, $\sigma_B^2 = \frac{1}{B}\sum_{i=1}^{B}(x_i - \mu_B)^2$
  - ↳ 2) standardise the input: $\hat{x} = \frac{x - \mu_B}{\sqrt{\sigma_B + \varepsilon}}$
  - ↳ 3) scale and shift: $z = \gamma\hat{x} + \beta$

* we introduce two learnable parameters: scale γ and shift β.

- BN has a regularising effect and allows for higher learning rates → faster training.

BN doesn't work well w/ small batch sizes or w/ seq. data (RNN)

BN/LN can be applied before/after linear layer

Layer normalisation (LN)

- LN standardises the input of each layer within a NN across features for each data pt.
  - ↳ 1) Compute the mean and variance: $\mu_L = \frac{1}{D_H}\sum_{i=1}^{D_H} x_i$, $\sigma_L^2 = \frac{1}{D_H}\sum_{i=1}^{D_H}(x_i - \mu_L)^2$
  - ↳ 2) Standardise the input: $\hat{x} = \frac{x - \mu_L}{\sqrt{\sigma_L + \varepsilon}}$
  - ↳ 3) Scale and shift: $z = \gamma\hat{x} + \beta$.

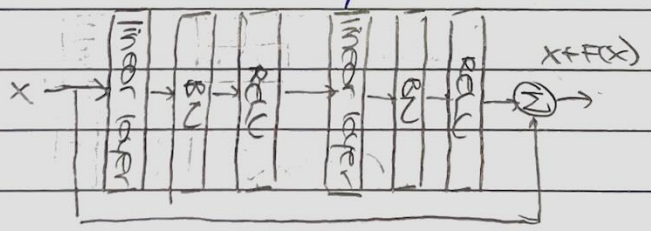
* we introduce two learnable parameters: scale γ and shift β.

- LN is indep. of batch size and is effective in seq. modelling.

LN can be computationally expensive and less effective in FCNN/CNN.

Vanishing gradients

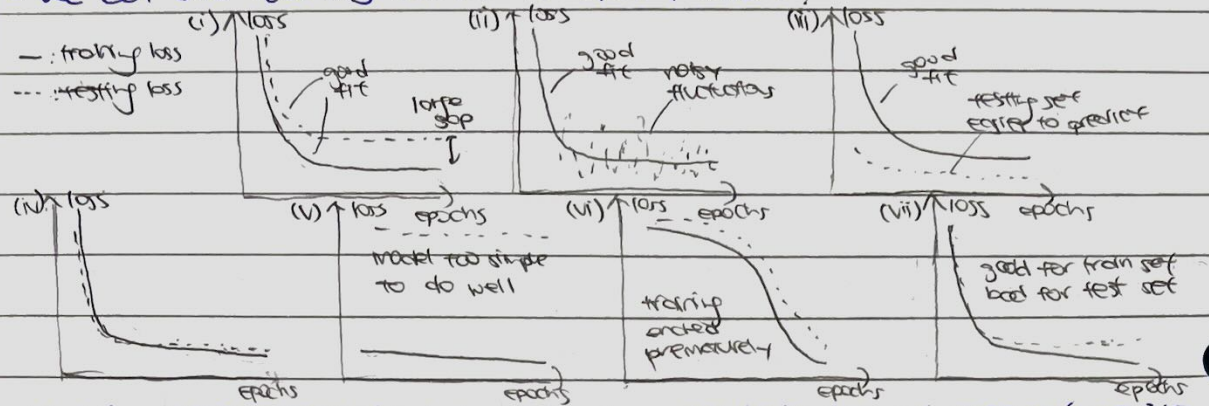
- As we add more hidden layers to a NN (i.e. increase depth), the gradients of the loss function wrt the weights become very small → weights not updated
- To mitigate the vanishing gradient problem, we can
  - ↳ use ReLU as the activation function — does not saturate.
  - ↳ use residual/skip connections — gradients flow more easily, won't become too small.
- A residual block has two branches: an identity branch in addition to the main branch



Regularisation

Learning curves

- The learning curve is a plot that shows time, steps or experience (iterations, model complexity) on the x axis and loss or accuracy on the y axis.
- We can use learning curves to spot problems.



↳ (i): unrepresentative train dataset — the train dataset does not provide sufficient info to learn the problem rel. to test dataset (e.g. train dataset too small)

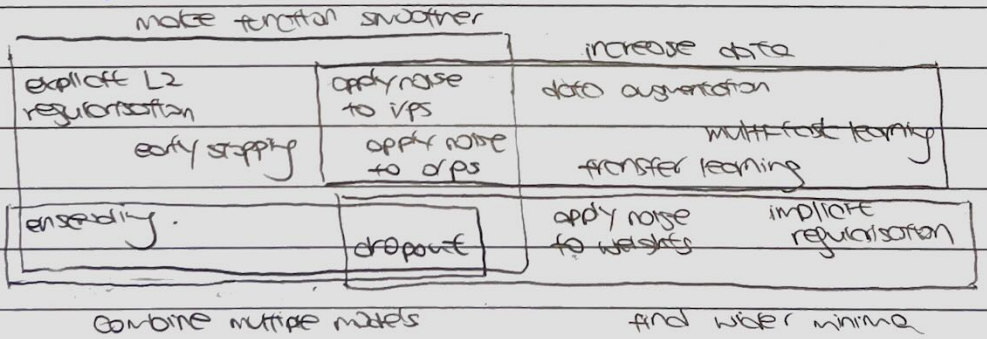
↳ (ii), (iii): unrepresentative test dataset — the test dataset does not provide sufficient info to evaluate the models ability to generalise.

↳ (v), (vi): model complexity too low / underfitting → ↑model complexity, ↑epochs

↳ (vii): model complexity too high / overfitting → regularisation / generalisation

Regularisation

- There may be a generalisation gap btwn the test and training loss curves due to the model overfitting or model constrained in case w/ no training data.
- Regularisation are methods to reduce this generalisation gap. (though technically, this would be a subset of methods that add an extra term to the loss)
- Common regularisation methods include:



Explicit regularisation

- In explicit regularisation, we add an extra term to the loss function,

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_i L(y_i, \hat{y}_i) - \log p(\theta)$$

where $p(\theta)$ is a prior — gives preference to certain parameter values.

- The most common choice is L2 regularisation / Tikhonov regularisation, where smaller parameters (in magnitude) are preferred. The prior is $p(\theta) \propto \exp(-\|\theta\|^2)$, so we have

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_i L(y_i, \hat{y}_i) + \lambda \|\theta\|^2$$

where the hyperparameter λ determines how much to penalise large $\|\theta\|$

- L2 regularisation effectively smoothens the loss surface → improved gradient descent

* In NN, we typically only penalise the weights (not bias) — weight decay.

Implicit regularisation.

- Analytical gradient descent is equivalent to solving the ODE

$$\dot{\theta} = -\eta \nabla L(\theta)$$

where $L(\theta)$ is the loss function.

- Iterative batch gradient descent results in a modified loss function

$$L_{GD}(\theta) = L(\theta) + \frac{\eta}{4} \|\nabla L(\theta)\|^2$$

encourage batches to have similar gradients

Iterative mini-batch gradient descent results in a modified loss function

$$L_{SGD}(\theta) = L(\theta) + \frac{\eta}{4m} \sum_{k=1}^{m} \|\nabla \hat{L}_k(\theta)\|^2 = L_{GD}(\theta) + \frac{\eta}{4m} \sum_{k=1}^{m} \|\nabla \hat{L}_k(\theta) - \nabla L(\theta)\|^2$$

where we have m mini batches in total and $\hat{L}_k(\theta)$ is the loss function of the kth mini batch

- By doing gradient descent iteratively, we effectively add a regularisation term (which depends on the learning rate η)

Early stopping

- During training, the weights start small and increase in magnitude. If we stop training early, the weights don't have time to overfit to noise → reduce model complexity.

- In practice, this is implemented using a patience parameter. If the training loss does not decrease by a certain amount for more epochs than the patience parameter, training is stopped early.

Data augmentation

- Data augmentation is a technique that enhances the diversity of training datasets w/o the need for collecting more data by creating modified versions of existing data pts.
- For image data, common data augmentation techniques include:
  - ↳ RandAugment: Flip, rotate, crop, stretch, blur, vignette, colour balance
  - ↳ Mixup: Mix two or more images via a weighted sum
- In general, we can also simply add noise to the i/p or o/p.
  - ↳ classification: Add noise to i/p
  - ↳ regression: Add noise to both i/p and o/p.

req. some physical intuition on the rough scale of i/ps or o/ps o/w SNR too low.

Ensembling

- Ensembling is a technique that combines multiple models (taking mean/median)
- We train for a few diff. times but modify something each time (could be diff. model, diff. initialisation, diff. subsets of data resampled w/ replacement (bagging))
- Ensembling works well on parallel computers / GPUs but not great on phones as inference time is increased significantly...

Dropout

- The core idea of drop out is the randomly setting the o/p of a fraction of neurons to zero during training.
- Dropout can eliminate kinks in functions that are far from data and doesn't contribute to training loss (remove unnecessary kinks → reduce model complexity)

Transfer learning and multitask learning

- Transfer learning is a technique where a model developed for one task is reused or adapted for a different but related task. The key steps are:
  - ↳ 1) Pre-training: Model first trained on a large dataset for a related task.
  - ↳ 2) Fine-tuning: Pre-trained model then fine-tuned on a smaller dataset of the target task (usually modify the head and train w/ low learning rate)
- Multitask learning is an approach where a model is trained to perform multiple related tasks simultaneously — we have a trunk model and multiple head models.

* Multitask learning is diff. from transfer learning as we train the head models at the same time. (rather than have a large pretrained model, then fine-tune the head later)

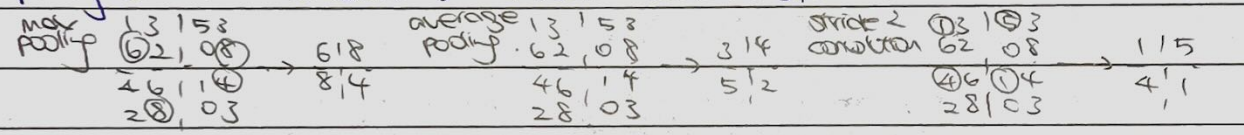
Convolutional neural network (CNN)

Convolutional layer (CONV)

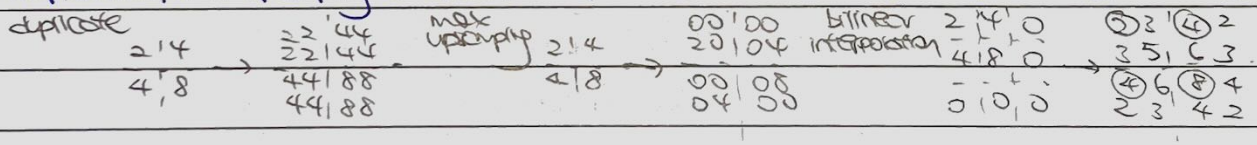
- The CONV layer applies a sliding window of filters to perform a correlation operation on the input data. (convolution is a linear operator).
- A CONV layer has the following hyperparameters:
  - ↳ Depth: no. of channels of the o/p (no. of channels of the i/p is usually fixed)
  - ↳ Kernel size: dimensions of the kernel (eg. 3×3)
  - ↳ Stride: no. of pixels the filter moves across the i/p after each convolution operation
  - ↳ Dilation: spacing btwn kernel elements (increase receptive field w/o increasing no. parameters)
  - ↳ Zero padding: adding zeros to the edge of the image to preserve spatial dim.
- Convolution + sliding window enables translation invariance in the model.
- A CONV layer is a special case of a fully-connected layer (w/ many zero weights)
- The o/ps of the CONV layer (feature map) are typically passed through a nonlinear activation function (usually RELU) → CONV_RELU block — we can even linearly combine feature maps to get a single channel.
- ReLU applies the nonlinearity element-wise → does not change the dimensions.

Downsampling and upsampling

- Downsampling is the process of reducing the spatial dimensions of data, typically through pooling or strided convolutions → introduce bottlenecks, extract dominant features

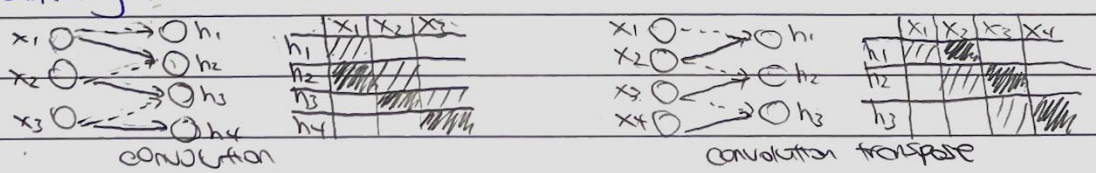


- Upsampling is the process of increasing the spatial dimensions of data, typically through duplication, max-upsampling or bilinear interpolation → restore features to original size



Convolution transpose layer

- convolution transpose (deconvolution) is a layer that learns to map low-resolution feature maps back to higher-dim. o/ps.
- This reverses the effects of regular convolution, effectively increasing the spatial dim. while maintaining learned features



Residual network (ResNet)

- Deeper networks perform better but have the vanishing gradient problem. This can be fixed using residual / skip connections.
- The residual unit idea is rather than having a layer that transform x, we consider a layer that adds something (residual) to the previous value

$$y = x + F(x)$$

where F is the residual function (diff. btwn input x and output y)

- Each residual block layer is now computing a separate residual function w/ separate trainable parameters. (Each of them are shallow → mitigate VGP)

* $F(x)$ and x must have the same size to perform the addition operation. (We could reduce dim. within $F(x)$, as long as we restore the dim. of the o/p).

Recurrent neural network (RNN)

Memory and hidden variables

- Intuitively, we can model memory by taking all the past information as input, i.e. we learn the map $\{x(\tau)_{\tau \le t}\} \mapsto y(t)$. [Complexity $O(t^2)$]
- Instead, we can encode past information using a set of internal variables $\{h_i\}_{i=1}^n$ that represents the memory. We now learn the map $\{x(t), h_1(t) .. h_n(t)\} \mapsto y(t)$.
- We assume n is finite, and there exists first order dynamics to describe the evolution of the internal variables. $\dot{h}_j(t) = \phi_j(x(t), h_1(t) .. h_n(t)) \ \forall j \in \{1, .. n\}$. [Complexity $O(t)$]

Recurrent neural network (cont'd)

- Consider the case $n=1$. For a seq. of i/p $\{x(\tau)_{\tau \le t}\}$ and o/p $\{y(\tau)_{\tau \le t}\}$, where we know $y(t) = \psi^t(x(\tau)_{\tau \le t})$, we can approximate ψ^t by

$$y(t) = g(x(t), h(t); \theta_g)$$

where the internal variable $h(t)$ encodes the history $\{x(\tau)_{\tau \le t}\}$ and follows update rule

$$\dot{h}(t) = \phi(x(t), h(t))$$

Discretising the update rule w/ forward Euler, and assuming uniform Δt.

$$\frac{h(t) - h(t-1)}{\Delta t} = \phi(x(t-1), h(t-1)) \quad \rightarrow \quad h(t) = f(x(t), h(t-1); \theta_f)$$

We also typically have the initial condition $h(0) = 0$.

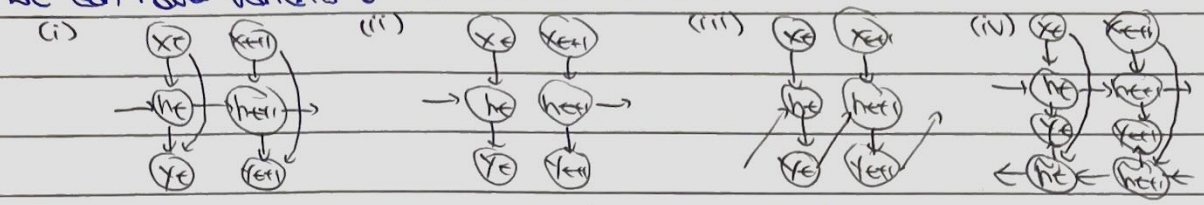
- We define the loss of RNN to be the average loss over timespan of the i/p.

$$L(\{y(t)\}_{t \le t}, \{\hat{y}(t)\}_{t \le t}) = \frac{1}{t} \sum_{\tau=1}^{t} L(y(\tau), \hat{y}(\tau))$$

* A defining feature for RNNs is that θ_f and θ_g is the same for all time τ → "recurrent"

RNN variants

- We can have variants of the recurrent network



(i) original: $h_t = g(x_t, h_{t-1})$, $y_t = f(x_t, h_{t-1})$

(ii) Elman: $h_t = g(x_t, h_{t-1})$, $y_t = f(h_t)$

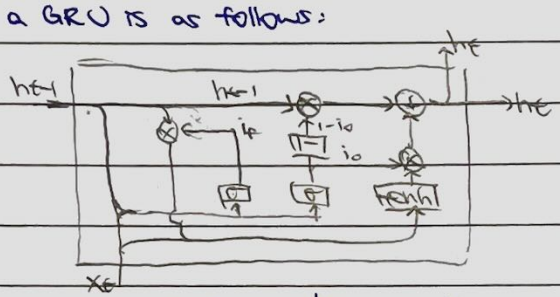
(iii) Jordan: $h_t = g(x_t, y_{t-1})$, $y_t = f(h_t)$ ← non-causal

(iv) Bi-directional: $h_t = g(x_t, h_{t-1})$, $\tilde{h}_t = \tilde{g}(x_t, h_{t+1})$, $y_t = f(h_t, \tilde{h}_t)$

- We can also add residual/skip connections, $h_t = g(x_t, h_{t-1}) + h_{t-1}$

Gated recurrent unit (GRU)

- The architecture of a GRU is as follows:

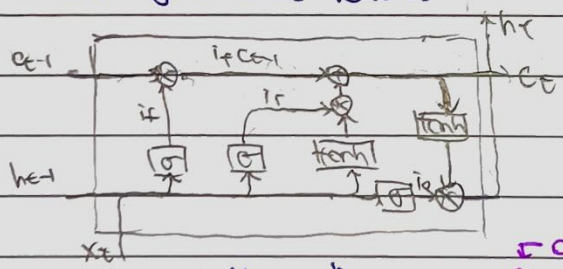


Forget gate	$i_f = \sigma(W_f^x x_t + W_f^h h_{t-1} + b_f)$ [gate over time]
output gate	$i_o = \sigma(W_o^x x_t + W_o^h h_{t-1} + b_o)$ [gate over feature (+time)]
Potential hidden state	$\tilde{h}_t = \tanh(W_h^x x_t + W_h^h (i_f h_{t-1}) + b_h)$
Hidden state update	$h_t = i_o \cdot h_{t-1} + (1 - i_o)\tilde{h}_t$

* sigmoid o/ps a value btwn 0 & 1.
→ useful for gating.

Long short term memory (LSTM)

- The architecture for a LSTM gate is as follows:

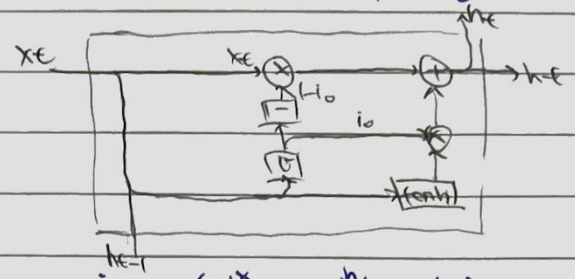


← optional peephole connections

Forget gate	$i_f = \sigma(W_f^x x_t + W_f^h h_{t-1} + b_f + W_f^c c_{t-1})$
Input gate	$i_i = \sigma(W_i^x x_t + W_i^h h_{t-1} + b_i + W_i^c c_{t-1})$
output gate	$i_o = \sigma(W_o^x x_t + W_o^h h_{t-1} + b_o + W_o^c c_t)$
Potential cell state	$\tilde{c}_t = \tanh(W_c^x x_t + W_c^h h_{t-1} + b_c)$
cell state update	$c_t = i_f c_{t-1} + i_i \tilde{c}_t$
Potential hidden state	$\tilde{h}_t = \tanh(c_t)$
Hidden state update	$h_t = i_o \tilde{h}_t$

Highway connection

- Highway connections are similar to residual / skip connections but we have a gated contribution from the passed-through input x_t



Output gate $i_o = \sigma(W_o^x x_t + W_o^h h_{t-1} + b_o)$

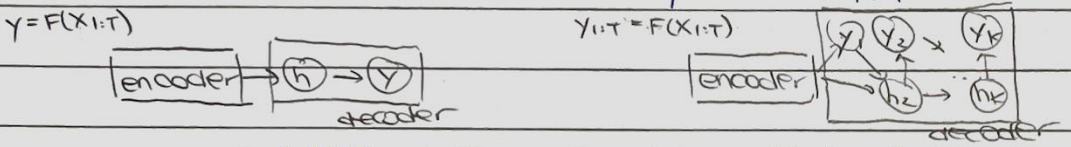
Potential hidden state $\tilde{h}_t = \tanh(W_h^x x_t + W_h^h h_{t-1} + b_h)$

Hidden state update $h_t = i_o \tilde{h}_t + (1-i_o) x_t$

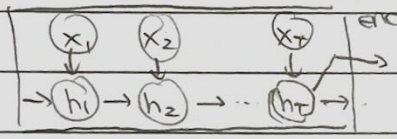
Attention mechanism and transformers

Sequence embedding.

- Consider regression / classification of the form $y = F(X_{1:T})$ / $Y_{1:T} = F(X_{1:T})$

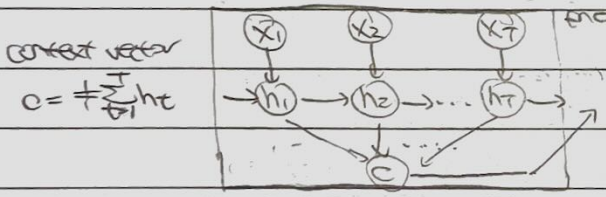


- We can use the final hidden state as information but this biases later i/ps.



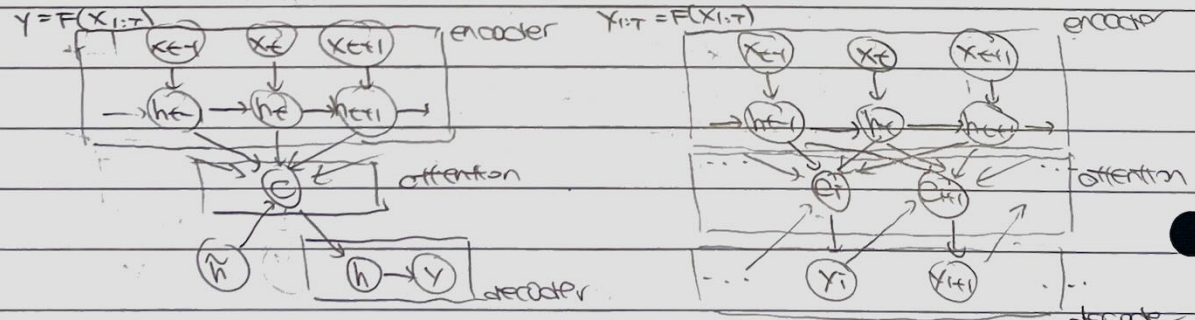
(We could use a bi-directional network, but this biases i/ps near the start/end)

- We can take the mean of all hidden states as information but not too effective



Instead, use weighted average which takes an input query $\tilde{h}$ — attention mechanism

(For the form $Y_{1:T} = F(X_{1:T})$, we feed the context vector c to each of the hidden states of the decoder → we req. a different decoder architecture)



* The output y should be formed by a fixed length vector (size independent to i/p seq. length)

Attention

- The attention mechanism addresses the bottleneck problem that arises w/ the use of fixed length encoding vector where the decoder has limited access to input info
- Suppose we have an i/p-o/p seq. $(\{x(t)\}_{t=1}^{T}, \{y(t)\}_{t=1}^{L})$ where $x(t), y(t) \in \mathbb{R}^D$

The attention layer has three main parts: (low dim. proj. $d_k, d_v \ll n$, usually $d_k = d_v$)

	learnable weights
1) Query function $W^Q \in \mathbb{R}^{d_k \times D}$	query vector $q_j = W^Q y(t=j) \in \mathbb{R}^{d_k}$, query matrix $Q = [q_1 \dots q_L]^T$
2) Key function $W^K \in \mathbb{R}^{d_k \times D}$	key vector $k_j = W^K x(t=j) \in \mathbb{R}^{d_k}$, key matrix $K = [k_1 \dots k_T]^T$
3) Value function $W^V \in \mathbb{R}^{d_v \times D}$	value vector $v_j = W^V x(t=j) \in \mathbb{R}^{d_v}$, value matrix $V = [v_1 \dots v_T]^T$

- We compute the similarity b/tw query Q and key K using a scaled dot product.

$$\tilde{\alpha}_{\tau j} = \frac{q_\tau^T k_j}{\sqrt{d_k}}$$ ← maintain appropriate variance after initialisation

Then normalise using softmax (for a given τ, normalise over j)

$$\alpha_{\tau j} = \text{softmax}(\tilde{\alpha}_{\tau j}) = e^{\tilde{\alpha}_{\tau j}} / \sum e^{\tilde{\alpha}_{\tau l}}$$

Finally, multiply w/ the value vector to get the attention

$$y(\tau) = \sum_{j=1}^{T} \alpha_{\tau j} v_j$$

- In summary, we have

$$\text{attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V$$

* In the context of seq. embedding, we have $q = \hat{h}$, $K = h$, $V = h$, $Y = C$ → we would not req. defining W^Q, W^K, W^V.

Multi-headed attention

- To extend on single-headed attention, we can have multiple heads, where each head has its own query, key, value functions.

$$\text{multihead}(Q, K, V) = \text{concat}(\text{head}_1, \dots \text{head}_H) W^O$$ ← project the results

where each head_i, $i = 1 \dots H$ is given by

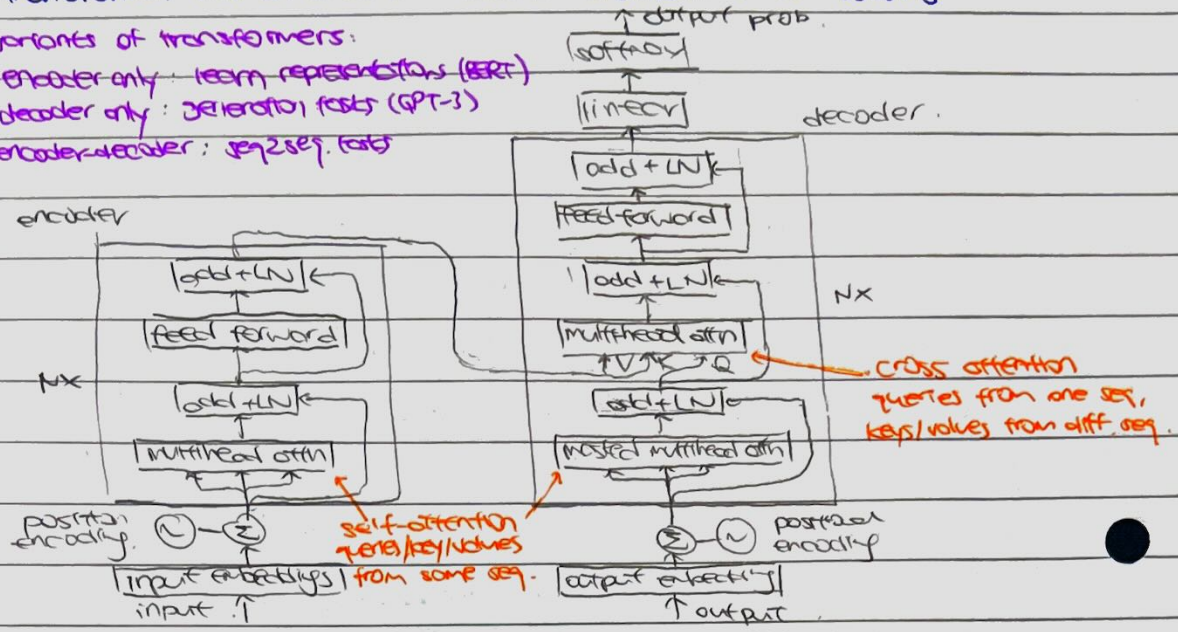
$$\text{head}_i = \text{attention}(Q_i, K_i, V_i)$$

* The projection matrices W_i^Q, W_i^K, W_i^V and output matrix W^O are learnable parameters.
* For purposes of 4F10, we simply add the contributions of each head

$$y(\tau) = \sum_{m=1}^{H} \sum_{j=1}^{T} \alpha_{\tau j}^m V_j^m$$

Transformer

- The transformer has an encoder-decoder architecture as follows:



- We stack N transformer blocks on both the encoder and decoder → N layers.
  (Note that within each layer, there are residual connections and layer norm)

Masked attention

- In certain applications, we would like the transformer to attend only to words in the past (i.e. cannot look ahead to know the answer). → "causal" attention
- This can be achieved by applying a zero mask to the attention layer.

$$(QK^T)_{ij} = 0 \quad \text{if } i > j$$

The masking pattern would look like the following:

- To ensure the rows still sum to 1, we set the top right cells to $-\infty$ before applying the softmax function.
- This is not needed for applications where it does not matter if the model can look ahead or not (eg. translating english to french)

Word embeddings.

[not necessarily each word = token but this is usually the case]

- A sequence of words is tokenized into a seq. of integers. These can be converted into vectors using one-hot encoding
- To obtain word embeddings, we apply an embedding matrix $A_0 \in \mathbb{R}^{|V| \times D}$ to the one-hot encoded vectors (practically, this is computationally wasteful → use array indexing).
- Mathematically, for an incoming sentence $w = [w_1, \dots w_T]$, w/ one-hot encoded matrix $\tilde{W} \in \mathbb{R}^{|V| \times T}$, the word embeddings $X \in \mathbb{R}^{D \times T}$ are given by

$$X = A_0^T \tilde{W} = A_0[w]$$

- The embedding matrix A_0 is typically trained on general text data (e.g. word2vec, BERT), then fine tuned on the specific target training data (transfer learning).

Positional encoding

- Attention has no representation of position, but predictions should be sensitive to location.
- We can incorporate positional encoding to our word embeddings by adding sine and cosine of diff. freq. (alternatively, this can be learned from scratch, as in ViT)

$$PE_{t,2i} = \sin(t/10000^{2i/D}) \qquad PE_{t,2i+1} = \cos(t/10000^{2i/D})$$

where t is the position of the word, D is the no. of dim. of the word embedding.
and $2i/2i+1$ denote the index of the PE vector.

* embedding = raw embedding + PE

Unembedding

[$A_{out} = A_0$ → weight tying.]

- We apply the unembedding matrix $A_{out} \in \mathbb{R}^{D \times |V|}$ to the final embeddings of the output, so we get a vector w/ a value for each token in the vocabulary. $X' \in \mathbb{R}^{|V|}$
- Then, we apply a softmax w/ hyperparameter temp. τ to get a prob. distribution. Including temp. allows us to control the randomness of predictions by scaling the logits before softmax.

$$\text{softmax}_\tau(X')_i = \frac{e^{X'_i/\tau}}{\sum_j e^{X'_j/\tau}}$$

* Increasing the temp. increases the randomness — flatter distribution
- To get an output word / seq. of words, we can either draw a single token each time or use Viterbi-style beam search to get the most likely seq.

<cls> token

- For classification tasks, we add a <cls> token at the start of the input seq. We then only take the output corr. to the <cls> token
(usually we have a transformer pretrained on general text as the trunk and have a randomly initialised MLP as the head).

Vision transformer (ViT)

- We can use ViT model for image classification tasks. Instead of word embeddings, we have patch embeddings. (w/ the first patch embedding corr. to the <cls> token).
- To generate patch tokens from an image, we divide the image into patches (eg. 4x4), then apply a linear layer on the flattened patches.
- At the output of the transformer, we have a MLP + softmax to get class probabilities.
- We can improve the performance of a ViT by distillation of a large CNN model.
  - ↳ step 1: Train a teacher CNN on ImageNet GT labels (cross entropy loss)
  - ↳ step 2: Train a student ViT to match ImageNet predictions from the teacher CNN (KL divergence loss) in addition to GT labels (cross entropy loss)
- Distillation allows the smaller student model to mimic a larger, pretrained teacher model → faster inference and reduced resource consumption on the simpler model

Image generation

Pixel RNN.

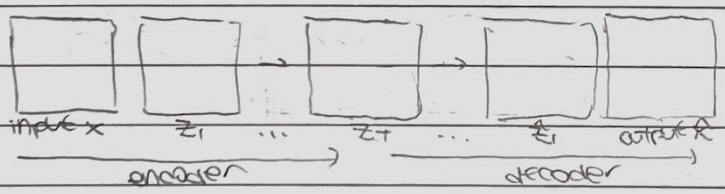
- Pixel RNN generates image pixel one at a time, starting at the top left corner.
- Each pixel depends implicitly on all pixel above and to the left. Each pixel has a hidden state that depends on the hidden state + RGB values from above + left.

$$h_{x,y} = f(h_{x-1,y}, h_{x,y-1}, W)$$

We then predict the RGB value of the pixel from its hidden state.

Diffusion models

- Consider an encoder that repeatedly adds noise to an image input (forward, diffusion process). We want to train a NN as the decoder (reverse process)



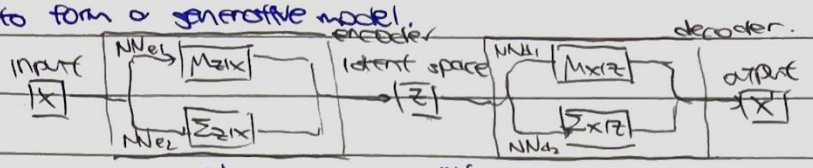
- The NN takes in a noisy image and the time step it is at, and outputs the slightly less noisy image from the previous step. (i.e. predict slightly denoised image)

$$\hat{z}_{t-1} = NN(z_t, t; \theta)$$

- To generate an image w/ a diffusion model, we input a completely noisy image, then get a high resolution image slowly over time.

Variational autoencoders (VAEs).

- VAEs use Bayesian variational inference w/ an autoencoder (encoder-decoder) architecture to form a generative model.



- Given training images $D = \{x_i\}_{i=1}^N$, where $x_i \in \mathbb{R}^{w \times h}$, we would like to learn the prob density $p(x)$ so we can draw from it (i.e. generate new image samples).
- We parameterise the prob density by θ, and we would like to max. the log likelihood

$$\sum_{x \in D} \log p_\theta(x).$$

Consider a latent space $z \in \mathbb{R}^{d_z}$. We can rewrite the log-likelihood of a data pt using Bayes rule.

$$p_\theta(x) = \frac{p_\theta(x|z)\,p(z)}{p_\theta(z|x)}$$

* $p_\theta(z|x)$ is intractable → $p_\theta(x)$ also intractable (cont. integrals)

- We set $p(z)$ to be an easy dist. to sample from, usually, the standard Gaussian $p(z) = \mathcal{N}(z|0,I)$
- Set $p_\theta(x|z)$ to have the form

$$p_\theta(x|z) = \mathcal{N}(x|\,\mu(z;\theta), \mathrm{diag}(\Sigma(z;\theta)))$$

where the mappings $\mu(z;\theta): \mathbb{R}^{d_z} \mapsto \mathbb{R}^{w \times h}$, $\Sigma(z;\theta): \mathbb{R}^{d_z} \mapsto \mathbb{R}^{w \times h}$ are NNs.

- Approx. $p_\theta(z|x)$ using $q_\phi(z|x)$ of the form

$$p_\theta(z|x) \approx q_\phi(z|x) = \mathcal{N}(z|\,\mu(x;\phi), \mathrm{diag}(\Sigma(x;\phi)))$$

where the mappings $\mu(x;\phi): \mathbb{R}^{w \times h} \mapsto \mathbb{R}^{d_z}$, $\Sigma(x;\phi): \mathbb{R}^{w \times h} \mapsto \mathbb{R}^{d_z}$ are NNs.

- The log likelihood $p_\theta(x)$ is lower bounded by the evidence lower bound (ELBO).

$$\log p_\theta(x) \geq \mathrm{ELBO}(x;\theta,\phi) = \mathbb{E}_{z \sim q_\phi(z|x)}[\log p_\theta(x|z)] - KL(q_\phi(z|x) \,||\, p(z))$$

Starting from $\log p_\theta(x)$,

$$\log p_\theta(x) = \log \frac{p_\theta(x|z)p(z)}{p(z|x)} = \log \frac{p_\theta(x|z)p(z)q_\phi(z|x)}{p(z|x)q_\phi(z|x)} = \log p_\theta(x|z) - \log \frac{q_\phi(z|x)}{p(z)} + \log \frac{q_\phi(z|x)}{p_\theta(z|x)}$$

Noting that $\mathbb{E}_{z \sim q_\phi(z|x)}[\log p_\theta(x)] = \int q_\phi(z|x) \log p_\theta(x)\,dz = \log p_\theta(x) \int q_\phi(z|x)\,dz = \log p_\theta(x)$,

$$\log p_\theta(x) = \mathbb{E}_{z \sim q_\phi}[\log p_\theta(x|z)] - KL(q_\phi(z|x) \,||\, p(z)) + KL(q_\phi(z|x) \,||\, p_\theta(z|x)) \geq \mathrm{ELBO}(x;\theta,\phi)$$

(dist. btwn prior and samples; how well $q_\phi(z|x)$ approx $p_\theta(z|x)$.)

→ We can max. $p_\theta(x)$ by max. $\mathrm{ELBO}(x;\theta,\phi)$

- During training, we can evaluate the ELBO as follows:

① $\mathbb{E}_{z \sim q_\phi(z|x)}[\log p_\theta(x|z)]$ term.

↳ Compute $\mu_{z|x}, \Sigma_{z|x}$ from the encoder, then sample zs from $q_\phi(z|x)$.

↳ Compute $\mu_{x|z}, \Sigma_{x|z}$ from the decoder, then compute the average of $\log p_\theta(x|z)$

② $KL(q_\phi(z|x) \,||\, p(z))$ term

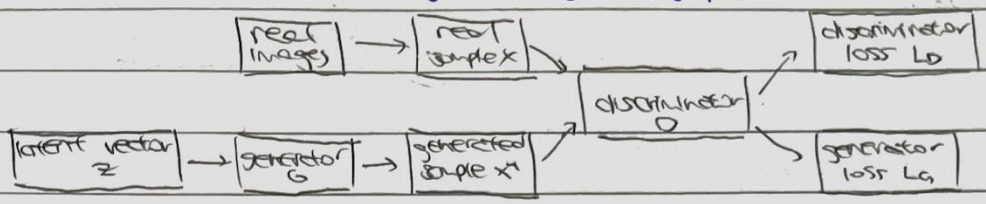
↳ $KL(q_\phi(z|x) \,||\, p(z)) = \int q_\phi(z|x) \log \frac{q_\phi(z|x)}{p(z)}\,dz = -\frac{1}{2} \sum_{j=1}^{d_z} \left(1 + \log((\Sigma_{z|x})_j^2) - (\mu_{z|x})_j^2 - (\Sigma_{z|x})_j^2\right)$

- After training the VAE, we can generate data by sampling z from the prior $p(z)$, then run the sampled z through the decoder to get the dist. of x. Then we sample from the dist. of x.

* Conditional sampling in VAEs enables more controlled (targeted) data generation.

Generative adversarial networks (GANs).

- GANs use two NNs, a generator and a discriminator, trained simultaneously through adversarial training to form a generative model.
  - ↳ the generator learns to generate plausible data. — used as -ve training data for discriminator
  - ↳ the discriminator learns to distinguish the generator's fake data from real data.



- The generative network $G: \mathbb{R}^{d_z} \mapsto \mathbb{R}^{w \times h}$ takes in a latent vector $z \in \mathbb{R}^{d_z}$ (sampled from a simple dist.) and transforms it into the data space $x^* \in \mathbb{R}^{w \times h}$.
- The discriminative network $D: \mathbb{R}^{w \times h} \mapsto \{0,1\}$ is a classifier that determines whether the input is a real sample x (gts 1) or a generated sample $x^* = D(z)$ (gts 0)
- The generator and discriminator loss functions L_G and L_D are given by.

$$L_G = -L_D = \min_G \max_D E_x[\log D(x)] + E_z[\log(1 - D(x^*))]$$

→ the generator and discriminator are playing a zero sum game.