Notes on Machine Learning & Deep Learning

C. Chan

Compiled on May 20, 2019

Contents

1 Introduction

In this section, we briefly introduce some useful aspects of machine learning (ML), some basic concepts in information theory, and two statistical estimators for ML models.

1.1 Machine learning

▷ Domingos, "A few useful things to know about machine learning" (2012).

$$\text{Learning} = \text{Representation} + \text{Evaluation} + \text{Optimization}$$

▷ Machine learning: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E.", by Tom M. Mitchell. Mainly concern the **opertional definition**, rather than defining in cognitive terms. This approach is **data driven**, instead of knowledge (logic) or statistics driven.
▷ Problems/Tasks: Three categories: Supervised, unsupervised, & reinforcement learning. Briefly divided into: Classification, regression, clustering, dimension reduction
▷ Hyperparameters: controls the representation capacity of the model (linear model, NN etc). In Bayesian statistics, hyperparameters are just the parameters in the prior belief. For NN, they include number of layers etc.

1.2 Information theory

Ref: [Goodfellow] Section 3.13. Read also [Goodfellow] Sections 3.1 for the reasons why **probability models** are used in statistical learning & deep learning.

1.2.1 Self-information

For only a single outcome
$$I(x) = -\log P(x)$$
(1) Likely events should have low information content
(2) Less likely events should have higher information content
(3) Independent events should have **additive** information

1.2.2 Shannon entropy

To quantify the amount of uncertainty in an entire probability distribution, use the Shannon entropy

$$H(x) = \mathbb{E}_{x\sim P}[I(x)] = -\mathbb{E}_{x\sim P}[\log P(x)]$$

which gives a lower bound on the number of bits needed on average to encode symbols drawn from a distribution P.

1.2.3 KL (Kullback-Leibler) divergence

Defined as

$$D_{\mathrm{KL}}(P||Q) = \mathbb{E}_{x\sim P}[\log P(x) - \log Q(x)]$$

which is the extra amount of information needed to send a message containing symbols drawn from P, when we use a code that was designed to minimize the length of messages drawn from Q. It is non-negative, and it is zero iff P and Q are identical distribution. Also $D_{\mathrm{KL}}(P||Q) \neq D_{\mathrm{KL}}(Q||P)$.

▶ *Proof* (Non-negativity of D_{KL}): Using Jensen's inequality $\mathbb{E}[f(x)] \geq f(\mathbb{E}[x])$ for a convex function $f(x)$,

$$\begin{aligned} D_{\mathrm{KL}}(P||Q) &= -\int dx P \log \frac{Q}{P} \\ &\geq -\log \left[\int dx P \cdot \frac{Q}{P} \right] = 0 \end{aligned}$$

where we take $f(x) = -\log x$. Note: A more rigirious derivation needs to concern about the set A such that $P(x) > 0, \forall x \in A$. See [Murphy] Theorem 2.8.1. ◀

▶ *Jensen's inequality*: For any convex function $f(x)$, we have

$$\sum_{i=1}^{n} \lambda_i f(x_i) \geq f\left(\sum_{i=1}^{n} \lambda_i x_i \right)$$

where $\lambda_i \geq 0$ and $\sum_i \lambda_i = 1$. This is clearly true for $n = 2$ by definition of convexity, and can be proved by induction for $n > 2$. ◀

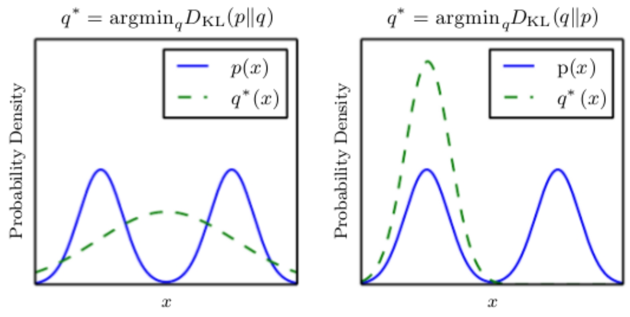


Figure 1: [Goodfellow] Fig. 3.6. (Left, **mass covering**) The effect of minimizing $D_{\mathrm{KL}}(p||q)$, such that a $q(x)$ with high probability overlap with $p(x)$ is selected. (Right, **mode collapse** or **mode seeking**) The effect of minimizing $D_{\mathrm{KL}}(q||p)$, such that a $q(x)$ that has low probability where $p(x)$ has low probability is selected. Reason: For some x, $q(x) \to 0$ and $p(x) \to 1$, then $q(x) \log \frac{q(x)}{p(x)} \to 0$, hence vanishing contribution to $D_{\mathrm{KL}}(q||p)$; while $q(x) \to 1$ and $p(x) \to 0$, then $q(x) \log \frac{q(x)}{p(x)} \to +\infty$, hence large contribution to $D_{\mathrm{KL}}(q||p)$.

1.3 Statistical estimators

Ref: [Goodfellow] Sections 3.11 & 5.6

1.3.1 Bayesian Statistics

▷ Conditional Probability: Defined as

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

▷ Bayes' Rule: Given by

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} = \frac{P(B|A)P(A)}{\sum_{a \in A} P(B|A)P(A)}$$

where the second equality utilizes $P(B) = \sum_{a \in A} P(B \cap A)$.

▷ Bayesian Inference:

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)} \quad \text{or} \quad \text{posterior} = \frac{\text{likelihood} \times \text{prior}}{\text{evidence}}$$

Prior and *posterior* are probability distributions for our *beliefs* before and after revealing the *evidence*. The likelihood $P(D|\theta)$ is the probability of seeing the evidence as generated by a prior model $P(\theta)$ with parameter θ. Starting with the model θ, our posterior belief is *adjusted* by the evidence D to be $P(\theta|D)$.

1.3.2 MLE (Maximum likelihood estimation)

Maximum likelihood estimator $\hat{\theta}$ for θ is defined as

$$\begin{aligned}
\hat{\theta}_{\mathrm{ML}} &= \arg\max_\theta \, p_{\mathrm{model}}(\{x^{(i)}\};\theta) \\
&= \arg\max_\theta \prod_{i=1}^m p_{\mathrm{model}}(x^{(i)};\theta) \\
&= \arg\max_\theta \sum_{i=1}^m \log p_{\mathrm{model}}(x^{(i)};\theta) \\
&= \arg\max_\theta \mathbb{E}_{x\sim\hat{p}_{\mathrm{data}}} \log p_{\mathrm{model}}(x;\theta) \\
&= \arg\min_\theta D_{\mathrm{KL}}(\hat{p}_{\mathrm{data}}||p^{(\theta)}_{\mathrm{model}}) \\
&= \arg\min_\theta \mathbb{E}_{x\sim\hat{p}_{\mathrm{data}}}[\log \hat{p}_{\mathrm{data}}(x) - \log p_{\mathrm{model}}(x;\theta)]
\end{aligned}$$

▶ *Aside* [(Point) Estimator or Statistic]: Let $\{x^{(i)}\}$ be a set of m i.i.d. data points. An estimator $\hat{\theta}_m$ of the parameter θ is any function of the data:

$$\hat{\theta}_m = g(\{x^{(i)}\})$$

Bias: of an estimator is defined as

$$\mathrm{bias}(\hat{\theta}_m) = E(\hat{\theta}_m) - \theta$$

where the expectation is over the data (seen as samples from a random variable) and θ is the true underlying value of θ used to define the data generating distribution. An estimator is unbiased if $\mathrm{bias}(\hat{\theta}_m) = 0$, and is asymptotically unbiased if $\lim_{m\to\infty} \mathrm{bias}(\hat{\theta}_m) = 0$. ◀

1.3.3 MAP (maximize a posteriori) estimation

Beside the MLE, one can estimate the parameters by MAP

$$\begin{aligned}
\hat{\theta}_{\mathrm{MAP}} &= \arg\max_\theta \, p(\theta|\mathcal{D}) \\
&= \arg\max_\theta \, p(\mathcal{D}|\theta)p(\theta) \\
&= \arg\max_\theta \log p(\mathcal{D}|\theta) + \log p(\theta)
\end{aligned}$$

when given evidence or data $\mathcal{D}$. If we use the uniform prior $p(\theta) \propto 1$, then it reduces to a MLE.

2 Statistical Learning

In this section, we will introduce several methods in statistical learning, namely regression, classification, dimension reduction, and clustering. The first two belong to *supervised* learning, while the later two are *unsupervised.* We shall start with linear regression, which is just simple straight line fitting, to familiarize with the terminologies, as well as the optimization methods to train the regression model.

2.1 Linear regression

Ref: [CS229]
▷ Hypothesis function $h_\theta(x)$ on features x_i. For example, a multi-linear function $h_\theta(x) = \theta^T x$, where $\theta = (\theta_0, \theta_1, \ldots, \theta_n)$ are the training parameters, and $x = (1, x_1, \ldots, x_n)$ are the features.
The task is to *train* (or learn) a *model* (or hypothesis function) $h_\theta(x)$, provided the *training data* $\{(x^{(i)}, y^{(i)}), i = 1, \ldots\}$ are given. If we are then given test features x_{test}, we can make a prediction $h_\theta(x_{\mathrm{test}})$, where θ's are the parameters trained by the training data.

▷ Cost function $J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$. The problem is to perform

$$\min_\theta J(\theta)$$

in order to minimize the *least square error*.

2.2 Training by minimization of cost function

▷ Gradient descent method: Repeat

$$\theta_j \leftarrow \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta) \quad \text{for all } j$$

until convergence. Here α is the *learning rate*.
Practical tricks: (1) feature scaling, (2) plotting $\min_\theta J(\theta)$ against number of iteractions, to ensure convergence, (3) To choose an appropriate α with the trial sequence

$$\ldots, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, \ldots$$

▷ *Advanced* optimization algorithms: (1) conjugate gradient, (2) BFGS, (3) L-BFGS, (4) ADAM. These are all implemented in machine learning framework like TensorFlow. As practitioner, we may just apply the algorithms.

2.3 Regularization

If we have too few features, the learned model may be *underfitting*. If we have too many features, the learned hypothesis may fit the training set very well (overfitting) such that the training error $J(\theta) \approx 0$, but fail to generalize to new samples (make good predictions on new samples). Options: (1) reduce number of features: (i) manually select which features to keep, (ii) model selection algorithm; (2) regularization: keep all the features, but reduce magnitudes/values of parameters θ_j. This method works well when we have a lot of features, each of which contributes a bit to predicting y. This is particularly useful for NN.
Modified cost function with *regularization parameter* λ

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right]$$

The λ-term is to penalize the appearance of unnecessary features, and more importantly it controls the trade-off between the training θ in the first term and keeping θ "reasonable". (It is a convention not to include θ_0 in the regularization term, since it only shifts the fitting function $h_\theta(x)$.)
The gradient descent equation for regularized cost function suppresses θ_j in each iteration: $\theta_j \leftarrow (1 - \alpha\lambda/m)\theta_j + \cdots$ to control the overfitting problem.

2.4 Logistic regression

Also know as binary classification with two outcomes $y^{(i)} \in \{0, 1\}$. It is useful to restrict the hypothesis function $0 \le h_\theta(x) \le 1$, which serves as a *probability density function*.
▷ For linear logistic regression, the hypothesis $h_\theta(x) = P(y = 1|x; \theta)$ is

$$h_\theta(x) = g(\theta^T x)$$
$$g(z) = \frac{1}{1 + e^{-z}}$$

where $g(z)$ is the *sigmoid function* (or *logistic function*). In this case, $\theta^T x$ is the *decision boundary*.

▷ Note that the navie choice of cost function (for a particular data) $\mathrm{Cost}\left(h_\theta(x), y\right) = \frac{1}{2}\left(h_\theta(x) - y\right)^2$, this cost function is non-convex (multiple local minima). We want the cost function for logistic regression to be convex [intuitively, Hessian matrix $H_{ij} = \partial^2 f/\partial x_i \partial x_j > 0$]. The choice is

$$\mathrm{Cost}\left(h_\theta(x), y\right) = J(\theta) = -y \log h_\theta(x) - (1-y)\log\left(1 - h_\theta(x)\right)$$

Gradient descent formula is $\theta_j \leftarrow \theta_j - \alpha \sum_{i=1}^m \left(h_\theta(x^{(i)}) - y^{(i)}\right) x_j^{(i)}$, which is identical to linear regression.
▷ There exists a very similar approach called *support vector machines* (SVM) for binary classification tasks. One advantage of SVM is that the *kernel trick* can be applied. In this case, the decision boundary can be nonlinear. Here we shall not go into the details of SVM since it is now replaced by more general neural network approach[1]. Interested readers can consult [Goodfellow] Section 5.7.2, or [CS229] Lecture 6.

2.5 Multinomial logistic regression

Also known as *softmax regression*, or multi-category classification.
A generalization of logistic regression to the case where there are multiple classes. Here $y^{(i)} \in \{1, \ldots, K\}$, where K is the number of classes. Our aim is to estimate the probability that $P(y = k|x)$ for each value of $k \in \{1, \ldots, K\}$. Thus our hypothesis will output a K-dim vector, whose elements sum to 1, giving us our K estimated probabilities. Concretely, the hypothesis takes the form

$$h_\theta(x) = \begin{bmatrix} P(y=1|x;\theta) \\ P(y=2|x;\theta) \\ \vdots \\ P(y=K|x;\theta) \end{bmatrix} = \frac{1}{\sum_{k=1}^K \exp \theta^{(k)T} x} \begin{bmatrix} \exp \theta^{(1)T} x \\ \exp \theta^{(2)T} x \\ \vdots \\ \exp \theta^{(K)T} x \end{bmatrix} \equiv \mathrm{softmax}(x; \theta^{(i)})$$

where $\theta^{(i)}$ are the training parameters for the decision boundaries of the i-th class. The cost function is

$$J(\theta) = -\left[\sum_{i=1}^m \sum_{k=1}^K \mathbb{I}(y^{(i)} = k) \log \frac{\exp \theta^{(k)T} x^{(i)}}{\sum_{k=1}^K \exp \theta^{(k)T} x^{(i)}}\right]$$

which is a negative log likeklihood (NLL), see [Murphy] Sect 8.3.7. Here $\mathbb{I}(\text{condtition}) = 1$ if condition is true, and $= 0$ otherwise.

2.6 EM for Mixture Models

Ref: [Murphy] Ch. 11
In this section, we will discuss, EM (expectation maximization) and its variant K-means algorithm, which can be used for *clustering*.
The simplest form of LVM (latent variable model) is when the latent variables $z_i \in \{1, \ldots, K\}$ represents a discrete latent state. We use a discrete prior $p(z_i) = \mathrm{Cat}(\pi)$, where π_i is the probability for each class $z_i = k$ satisfying $\pi_k \in [0, 1]$ and $\sum_k \pi_k = 1$.

► *Aside* (categorical or multinoulli distribution):

$$\mathrm{Cat}(x|\theta) = \prod_{j=1}^K \theta_j^{\mathbb{I}(x_j = 1)}$$

◄

For the likelihood, we use $p(x_i|z_i = k) = p_k(x_i)$, where p_k is the k-th base distribution for the observations. The overall model is a mixture model for K base distributions,

$$p(x_i|\theta) = \sum_{k=1}^K \pi_k p_k(x_i|\theta)$$

[1]Though the use of kernels improves the representation power of the decision boundary, but it still cannot represent all possible functions. In the NN approach, the decision boundary is represented by the universal approximator – the NN, hence more general.

2.6.1 Issue: non-convex MAP estimate

Consider the log-likelihood for an LVM:

$$\log p(\mathcal{D}|\theta) = \sum\nolimits_i \log \left[\sum\nolimits_{z_i} p(x_i, z_i|\theta)\right]$$

The sum over z_i is inside the log function since they are hidden variables.
Suppose we consider exponential family as the joint probability distribution $p(x_i, z_i|\theta)$, then

$$p(x_i, z_i|\theta) = \frac{1}{Z(\theta)} \exp\left[\theta^T \phi(x, z)\right]$$

For complete data, the problem is easily solved. But for latent variables, the **observed data log likelihood** is

$$\ell(\theta) = \sum\nolimits_i \log \sum\nolimits_{z_i} p(x_i, z_i|\theta) = \sum\nolimits_i \log \left[\sum\nolimits_{z_i} e^{\theta^T \phi(x_i, z_i)}\right] - N \log Z(\theta)$$

One can show that the log-sum-exp function is convex, and we know that $Z(\theta)$ is convex. However, the difference of two convex functions is not, in general convex. $\implies$ Complications when applying optimization algorithms for MLE or MAP would likely face the issue of local maxima. Here the solution is provided by the EM (expectation maximization) algorithm to iteratively optimize the mixture model.

2.6.2 Basic idea

The goal is to maximize the log-likelihood of the observed data

$$\ell(\theta) = \sum\nolimits_i \log p(x_i|\theta) = \sum\nolimits_i \log \sum\nolimits_{z_i} p(x_i, z_i|\theta)$$

EM gets around the issue as follows. Define the **complete data log likelihood**,

$$\ell_c(\theta) = \sum_{i=1}^{N} \log p(x_i, z_i|\theta)$$

and the **expected complete data log likelihood** or **auxiliary function**,

$$Q(\theta, \theta^{(t-1)}) = \mathbb{E}[\ell_c(\theta)|\mathcal{D}, \theta^{(t-1)}]$$

where t is the current iteration number. The expectation is taken w.r.t the old parameters $\theta^{(t-1)}$ and the observed data $\mathcal{D}$. The **E step** is to compute the auxiliary function or the terms inside of it. In the **M step**, we optimize $Q(\theta, \theta^{(t-1)})$ w.r.t θ, namely

$$\begin{aligned} &\text{MLE} \qquad \theta^{(t)} = \arg\max_\theta Q(\theta, \theta^{(t-1)}) \\ &\text{MAP} \qquad \theta^{(t)} = \arg\max_\theta Q(\theta, \theta^{(t-1)}) + \log p(\theta) \end{aligned}$$

2.6.3 Variational basis for EM

Ref: [Murphy] Section 11.4.7.
It can be shown that the observed data log likelihood respects the following relation

$$\ell(\theta^{(t+1)}) \geq Q(\theta^{(t+1)}, \theta^{(t)}) \geq Q(\theta^{(t)}, \theta^{(t)}) = \ell(\theta^{(t)})$$

Hence we conclude that the EM algorithm monotonically increases the auxiliary function until it reaches a local optimum.

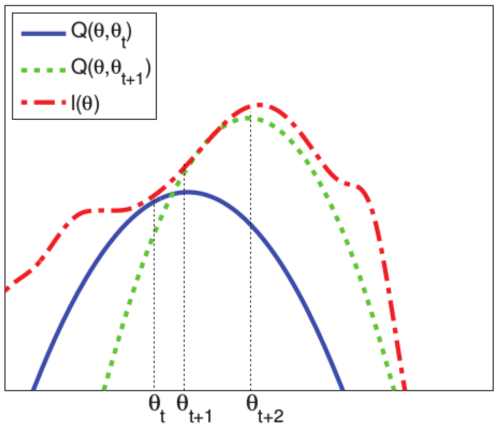


Figure 2: Illustration of EM algorithm. The parameter $\theta^{(t)}$ gives an auxiliary function $Q(\theta, \theta^{(t)})$ (E step), which is a lower bound of $\ell(\theta)$. Then we optimize $Q(\theta, \theta^{(t)})$ to give $\theta^{(t+1)}$ (M step), and then obtain the next auxiliary function $Q(\theta, \theta^{(t+1)})$. This alternating process of EM is continued until a local optimal is reached.

2.6.4 Mixtures of Gaussians

Multivariate Gaussian with mean μ_k and covariance matrix Σ_k:

$$p(x_i|\theta) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x_i|\mu_k, \Sigma_k)$$

where the model parameters θ are π_k, μ_k, Σ_k. The auxiliary function is

$$\begin{aligned} Q(\theta, \theta^{(t-1)}) &= \sum_i \mathbb{E}\left[\log \prod_{k=1}^{K} [\pi_k p(x_i|\theta_k)]^{\mathbb{I}(z_i=1)}\right] \\ &= \sum_{ik} \mathbb{E}[\mathbb{I}(z_i = 1)] \log [\pi_k p(x_i|\theta_k)] \\ &= \sum_{ik} \underbrace{p(z_i = k|x_i, \theta^{(t-1)})}_{\equiv r_{ik}} \log [\pi_k p(x_i|\theta_k)] \end{aligned}$$

where r_{ik} is the **responsibility** that cluster k takes for data point i.
E step:

$$r_{ik} = \frac{\pi_k p(x_i, \theta_k^{(t-1)})}{\sum_{k'} \pi_{k'} p(x_i, \theta_{k'}^{(t-1)})}$$

M step: Optimize Q w.r.t π and θ_k. For π, we have

$$\pi_k = \frac{1}{N} \sum_i r_{ik} \equiv \frac{r_k}{N}$$

where r_k is the weighted number of points assigned to cluster k. See eqs (11.31)-(11.32) for optimized μ_k and Σ_k. After computing the new estimates, we set $\theta^{(t)} \leftarrow (\pi_k, \mu_k, \Sigma_k)$ for $k = 1, \ldots, K$ and go to the next E step.

▶ *Derivation for* π_k: [Murphy] Sect 3.4.3. Add a Lagrange multiplier to the auxiliary function for the constraint $\sum_k \pi_k = 1$, we need to optimize

$$\ell(\theta, \lambda) = Q(\theta, \theta^{(t-1)}) + \lambda \left(1 - \sum\nolimits_k \pi_k\right)$$

Then $\partial \ell / \partial \lambda = 0$ gives the constraint. Taking derivatives w.r.t π_k yields

$$\pi_k \propto r_k$$

and the result follows. ◀

2.7 K-means algorithm

It is a popular variant of EM for Gaussian mixture. *Simplications*: (1) the covariance matrix $\Sigma_k = \sigma^2 \mathbb{I}$ is fixed, and (2) probability of classes $\pi_k = 1/K$ is fixed. Hence only the cluster centers μ_k have to be estimated. The algorithm is given below:

Algorithm 1 K-means algorithm.

```
1.  initialize the cluster centers $\mu_k$ randomly
2.  repeat
3.      Assign each data point to its closet cluster center: $z_i = \arg\min_k \|x_i - \mu_k\|^2$
4.      Update each cluster center by
            $$\mu_k = \frac{1}{N_k} \sum_{i:z_i=k} x_i$$
5.  until converged
```

2.8 Principal component analysis (PCA)

For dimension reduction.
Let $X = \begin{bmatrix} | & | & & | \\ x_1 & x_2 & \cdots & x_M \\ | & | & & | \end{bmatrix}$ be a $N \times M$ matrix, so each x_i is a N-dim column vector (data). Here M is the number of input data. We want to extract the key features in a k-dim ($k \leq N$) to represent the original N-dim data. We *assume* that the data are *centered*: $\bar{x} = \frac{1}{M}\sum_i x_i = 0$, then we can define the *covariance matrix*: $C = \frac{1}{M}\sum_i (x_i - \bar{x})(x_i - \bar{x})^T = \frac{1}{M} X X^T$. Then we solve the eigen-problem: $CU = U\Lambda$ with (orthonormal) eigenvector matrix $U = \begin{bmatrix} | & | & & | \\ u_1 & u_2 & \cdots & u_N \\ | & | & & | \end{bmatrix}$ ($u_i^T u_i = 1$) and diagonal eigenvalue matrix $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_N)$. We simply select the k largest eigenvalues and the corresponding eigenvectors as the projector $U_k = \begin{bmatrix} | & | & & | \\ u_1 & u_2 & \cdots & u_k \\ | & | & & | \end{bmatrix}$. The final aim of PCA is to find the projected data y_i in the reduced k-dim: $y_i = U_k^T x_i$. It is in these (linear) projection principal vectors, the data are most spread (with the highest covariance), and hence they are the *key features*.
Inner product formulation of PCA: Notice that the eigenvectors can be expressed linearly in terms of the data $u_\alpha = \sum_i a_i^\alpha x_i$ (or symbolically $U_k = aX$), where a_i^α are yet-determined coefficients. Next, the projected data is $y_i = U_k^T x_i = (aX)^T x_i$. Note that $X^T x_i$ is an **inner product**. Hence, once we know a_i^α, we can readily perform the projection. Indeed, we can show that a_i^α are determined by the eigen-problem: $Ka = \tilde{\lambda} a$, or explicitly

$$\underbrace{(x_i^T x_j)}_{\equiv K_{ij}} a_j^\alpha = \underbrace{N\lambda_\alpha}_{\equiv \tilde{\lambda}} a^\alpha$$

where K is a $N \times N$ matrix of elements of **inner products** $x_i^T x_j$.
Notice that the resulting u_α vectors (with $\|a^\alpha\| = 1$) are not normalized, we can simply rescale $a^\alpha \to \bar{a}^\alpha$ such that $\|\bar{a}^\alpha\| = 1/\sqrt{\tilde{\lambda}_\alpha}$. Then, we obtain the projector matrix $U_k = \bar{a}X$ such that each column vector u_α is orthonormal $u_\alpha^T u_\beta = \delta_{\alpha\beta}$.

2.8.1 Kernel trick

Ref: [NeuralComputation.10.1299] “Nonlinear Component Analysis as a Kernel Eigenvalue Problem”, by Schölkopf, Smola, & Müller.
In a higher dimension space, we define the mapping $\phi : X^N \to \mathcal{F}$. The input data is then $\phi(x_i)$. We want to perform PCA with the covariance matrix

$$\bar{C} = \frac{1}{M}\phi(x_i)\phi^T(x_i) = \frac{1}{M}\phi(X)\phi^T(X)$$

Naively, we can perform the mappings $\phi(x_i)$, then do PCA. But this is an inefficient method in terms of computational storage and time. We can actually find the final results, namely the projected data y_i, in a more efficient way without knowing the explicit form of ϕ and the eigenvector U by using a few *tricks*.
In the inner product formulation, the explicit knowledge of C and U is not needed, only the inner product is utilized. Hence, we simply generalize the inner product $x_i^T x_j$ in PCA to that of ϕ or the centered ψ in KPCA, namely

$$\bar{K}_{ij} = \psi^T(x_i)\psi(x_j)$$

Here $\psi(x_i) = \phi_i - \frac{1}{M}\sum_k \phi_k$ is the *centered* data after mapping ϕ, where we defined $\phi_i \equiv \phi(x_i)$. One can show that $\bar{K} = K - I_M K - K I_M + I_M K I_M$, or explicitly

$$\bar{K}_{ij} = K_{ij} - \frac{1}{M}\sum_k K_{kj} - \frac{1}{M}\sum_l K_{il} + \frac{1}{M^2}\sum_{kl} K_{kl}$$

where I_M is a $M \times M$ matrix with all elements $= \frac{1}{M}$. We need to solve the eigen-problem

$$\bar{K}\bar{a} = \tilde{\lambda}\bar{a}$$

Finally, given the new data t, the projected data after KPCA is $t' = \sum_i \bar{a}_i \bar{K}(x_i, t)$.
Kernel examples:
▷ Linear: $K(x, y) = x^T y + c$
▷ Polynomial: $K(x, y) = (ax^T y + c)^d$
▷ RBF (radial basis function) or Gaussian: $K(x, y) = e^{-\|x-y\|^2/2\sigma^2}$

3 Deep Learning

[Goodfellow] quotes: The solution for computer to learn is to allow them to learn from experience and understand the world in terms of a hierarchy of concepts, with each concept defined in terms of its relation to simpler concepts. If we draw a graph showing how these concepts are built on top of each other, the graph is deep, with many layers $\implies$ deep learning.

3.1 Universal approximation theorem

Let $\varphi(\cdot)$ be a non-constant, bounded, and monotonically-increasing continuous function. Let I_m denote the m-dimension unit hypercube $[0, 1]^m$. The space of continuous functions on I_m is denoted by $C(I_m)$. Then, given any $\varepsilon > 0$ and any function $f \in C(I_m)$, there exists an integer N, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$, where $i = 1, \dots, N$, such that we may define

$$F(x) = \sum_{i=1}^{N} v_i \varphi(w_i^T x + b_i)$$

as an approximate realization of the function f, where f is independent of φ; that is

$$|F(x) - f(x)| < \varepsilon$$

for all $x \in I_m$. In other words, functions of the form $F(x)$ are dense in $C(I_m)$. This still holds when replacing I_m with any compact subset of $\mathbb{R}^m$.

Ref: [Cybenko, 1989] with sigmoid activation functions
Ref: [Hornik etal., 1989] With generic activation functions, it is the multilayer feed forward architecture itself gives neural networks the potential of being universal approximators.
Ref: a visual "proof" by Michael Nelson. Basically, the function f can be discretized over I_m, and each discretized value $f(x_i)$ can be approximated by several summing step functions of various coefficients at $x_i \in I_m$.

3.1.1 Implication

In the previous section, we considered many examples of statistical learning with *linear* model $f_\theta(x) = \theta^T x$. Given the above theorem, we can replace the linear model with a NN that can represent basically any functions:

$$f_\theta(x) = \text{neural network}$$

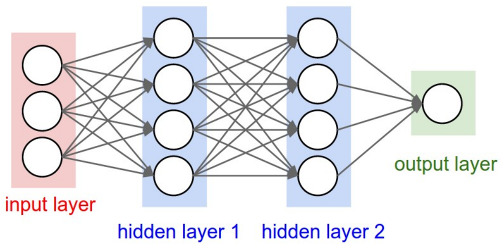
to perform the tasks of regression, classification as in statistical learning.

3.2 Neural networks

Feed-forward NN: Keywords: *input units* x, *hidden units* a.

$$x_i \to a_i^{(\ell)} \to h_\Theta(x)$$

where $a_i^{(\ell)}$ is the "activation" of unit i in layer j, and $\Theta^{(\ell)}$ is matrix of weights controlling function mapping from layer ℓ to layer $\ell + 1$



Cost function: K output nodes

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log h_\Theta(x^{(i)})_k + (1 - y_k^{(i)}) \log(1 - h_\Theta(x^{(i)})_k)\right] + \frac{\lambda}{2m}\sum_{\ell=1}^{L-1}\sum_{i=1}^{s_\ell}\sum_{j=1}^{s_{\ell+1}}(\Theta_{ij}^{(\ell)})^2$$

Forward propagation: To obtain the activation $a_j^{(\ell)}$,

$$\begin{aligned} a_i^{(\ell=1)} &= x_i \\ z_i^{(\ell)} &= \sum\nolimits_j \Theta_{ij}^{(\ell-1)} a_j^{(\ell-1)} \\ a_i^{(\ell)} &= g(z_i^{(\ell)}) \quad \text{with } g(z) = \frac{1}{1+e^{-z}} \\ h_\Theta(x)_i &\equiv a_i^{(L)} \end{aligned}$$

3.3 Back-propagation (Backprop)

BackProp is a more efficient way to obtain the gradients $\partial J(\Theta)/\partial\Theta_{ij}^{(\ell)}$ for gradient descent by computing the "error of node j in layer ℓ",

$$\begin{aligned} \delta_j^{(L)} &= a_j^{(L)} - y_j \\ \delta_i^{(\ell)} &= \sum\nolimits_j (\Theta_{ij}^{(\ell)})^T \delta_j^{(\ell+1)} \odot g'(z_i^{(\ell)}) \end{aligned}$$

Algorithm 2 Back-propagation (with regularization λ)

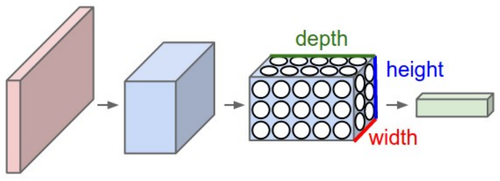
```
Training set $\{(x^{(i)}, y^{(i)}), i = 1, \dots, m\}$
Set $\Delta_{ij}^{(\ell)} = 0$ for all $i, j, \ell$
For $i = 1$ to $m$ {
    Set $a^{(1)} = x^{(i)}$
    Perform forward propagation to compute $a^{(\ell)}$ for $\ell = 2, 3, \dots, L$
    Using $y^{(i)}$ to compute $\delta^{(L)} = a^{(L)} - y$
    Compute $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$
    $\Delta_{ij}^{(\ell)} \leftarrow \Delta_{ij}^{(\ell)} + \delta_i^{(\ell+1)}(a_j^{(\ell)})^T$
}
$D_{ij}^{(\ell)} \leftarrow \begin{cases} \frac{1}{m}\Delta_{ij}^{(\ell)} + \lambda\Theta_{ij}^{(\ell)} & \texttt{for } j \neq 0 \\ \frac{1}{m}\Delta_{ij}^{(\ell)} & \texttt{for } j = 0 \end{cases}$
```

where $g'(z^{(\ell)}) = a^{(\ell)} \odot (1_\text{v} - a^{(\ell)})$ (1_v is a vector filled with 1's), and $\odot$ is the element-wise multiplication such that $(a \odot b)_i = a_i b_i$. If ignore regularization or $\lambda = 0$, then

$$\frac{\partial}{\partial \Theta_{ij}^{(\ell)}} J(\Theta) = \delta_i^{(\ell+1)}(a_j^{(\ell)})^T$$

A brief derivation is given in Appendix A.
Implementation notes: (1) unrolling parameters into vector, for better CPU optimization with vectorization; (2) gradient checking: $\partial_{\Theta_{ij}^{(\ell)}} J(\Theta) \approx [J(\Theta + \epsilon) - J(\Theta - \epsilon)]/2\epsilon$; (3) random initialization
Diagnostic: (1) Evaluating hypothesis by using random 70% of data as training set, and the rest as *test set*. Compute the test set error $J_\text{test}(\Theta) = \frac{1}{2m_t}\sum_{i=1}^{m_t}\left(h_\theta(x_t^{(i)}) - y_t^{(i)}\right)^2$. (2) Model selection with training set, *cross validation set*, and test set. (3) Balance between bias & variance error by plotting $J_\text{train}(\Theta)$ and $J_\text{cv}(\Theta)$ against number of features. High bias (underfit) *iff* $J_\text{cv}(\Theta) \sim J_\text{train}(\Theta)$; high variance (overfit) *iff* $J_\text{cv}(\Theta) \gg J_\text{train}(\Theta)$. (4) Learning curves against training set size m. Ref: [CS229] Lecture 7.

3.4 CNN/ConvNet



[CS231n notes]
▷ Inspired by brain experiments on visual cortex
▷ Usually used for image $\Longrightarrow$ the 1st NN layer consists of **width**, **height**, and **depth**. Here depth refers to the 3 color channels (RGB)
▷ Architecture
– `CONV`: Convolution layer. Filters are used to extract the visual features in an image. The parameters in each filter can be trained. The filters gather the *local* spatial information of a 2D image by computing a dot product between their weights and a small region they are connected to in the input volume. The number of filters become the depth of the output of this CONV layer.
Keywords: **stride**, **zero-padding** (at boundary) to maintain spatial size, useful formula $(W - F)/S + 1$
– `RELU`: element-wise activation function, such as the ReLU (rectified linear unit) $\max(0, x)$
– `POOL`: perform a downsampling operation along the spatial dimensions (width, height), such that *max pooling*.
– `NORM`: [Ioffe-Szegedy] Fallen out of favor since in practice their contribution has been shown to be minimal
– `FC`: Fully-connected layer

▷ Computational considerations: *Memory size* $\sim$ number of weights; *Computational complex* $\sim$ neuron connectivity

3.5 Recurrent NN

Ref: [colah blog] "understanding LSTM networks"
RNN: $h_t = \tanh W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$. During backprop, the weight matrix W multiplied n times, namely W^n, is needed. If the leading singular value of W is > 1 (< 1), then we have exploding (vanishing) gradient problem.
LSTM (Long short-term memory):

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$
$$c_t = f \odot c_{t-1} + i \odot g$$
$$h_t = o \odot \tanh c_t$$

where σ is the sigmoid function, and $\odot$ denotes the element-wise product. LSTM is used to tackle the vanishing and exploding gradient problem.
Here f is the forget gate, whether to erase cell; i is the input gate, whether to write to cell; g is a gate to control how much to write to cell; and o is the output gate, how much to reveal cell. h_t and c_t are the hidden state and the cell state at time step t.

3.6 Restricted Boltzmann machines (RBM)

RBM can be used to learn a probability distribution. The NN consists of only 2 layers – visible v and hidden h. The energy in matrix notation is

$$E(v,h) = -\sum_i a^T v - b^T h - v^T W h$$

where a, b are vectors and W is a matrix, and they are the model parameters to be learned. The probability distribution has a Boltzmann form

$$P(v,h) = \frac{e^{-E(v,h)}}{Z}$$

The marginalized probability by marginalize (integrate out) the hidden layer is

$$P(v) = \frac{1}{Z}\sum_h e^{-E(v,h)}$$

4 ML in Physics

4.1 PCA – Extracting the order parameter

Ref: [PRB.94.195105] & later developments
Consider classical Ising model $H = -J\sum_{\langle ij\rangle}\sigma_i\sigma_j$. Then use MC to generate M samples for multiple number of temperatures T for a lattice of N. These date is used to construct a $M \times N$ matrices X. PCA is then performed on the data to extract the principal component. It turns out that the principal component vector (with the largest principal eigenvalue) corresponds to the order parameter in the Ising system. This result can be easily understood since it is only in this principal vector the data is "most spread" (the highest covariance) in low temperature $T < T_c$. Later development shows that the second largest principal vector corresponds to the susceptibility of the spins.

Similar methods are also applied to other physical systems but the results are not desirable due to a number of reasons: (1) For continuous symmetry breaking, enhanced flucutations of the physical degrees of freedom invalid the PCA method or its kernel variants; (2) for quantum system, the quantum fluctuations lead to similar problem. The unsupervised ML problem of order parameters seems not quite generalizable, and ML is usually used to perform phase transition detection (see below).

4.2 Classification – Phase transition detection

Ref: [nphys4035] & later developments
A NN is used to learn 2 distinct phases from MC samples. The phase transition point is determined by the maximum confusion point, that is the probability of classifying to each phase is 50%.

4.3 Regression – Acceleration for MC proposals

Ref: [PRB.95.041101] & later developments
State proposals of MC (Monte Carlo) simulations for some models can be inefficient. In this paper, they consider an effective model

$$H = \sum_i \alpha_i \tau_i + \sum_{ij} \alpha_{ij} \tau_i \tau_j + \sum_{ijk} \alpha_{ijk} \tau_i \tau_j \tau_k + \cdots$$

where τ_i are spins and α's are the parameters to be learned from the original model. This is essentially a regression problem. MC samples are first generated from the original model, then α's are fitted accordingly. Finally, only the effective model H is used to propose new states for MC simulations.

4.4 Representation power of NN – ANN for quantum many-body functions

Ref: [ncomms.8.662] "Efficient representation of quantum many-body states with RBM"
The paper proves that quantum many-body states with exponentially large Hilbert space can be *efficiently* represented by RBM (restricted Boltzmann machine) and DBM (deep Boltzmann machine) with only polynomial number of parameters. The representation power of RBM is restricted to a few classes of wavefunctions, but that of DBM is not.
Hence the Boltzmann machines can be useful for variational studies. See for instance [science.355.602], [PRB.96.205152].

4.5 Updates

– [1807.09422] Solving frustrated quantum many-particle models with CNN
— To avoid trapping in local minima, they optimize the CNN via the replica-exchange molecular dynamics method
— Minimize the total energy by SGD (stochastic gradient descent), where the energy and the gradients are calculated via MC sampling over spin configurations.

A Derivations for BackProp

Note: The derivation is **notational**. Readers should themselves fill in the details like index summations and element-wise multiplications.
Suppose the cost function is the square error function (regression)

$$J(\Theta) = \frac{1}{2}(a^{(L)} - y)^2$$

or the cross entropy (classification)

$$J(\Theta) = -y \log a^{(L)} - (1 - y) \log(1 - a^{(L)})$$

one can easily show that

$$\frac{\partial J}{\partial \Theta^{(L-1)}} = \delta^{(L)} a^{(L-1)}$$

where we defined

$$\delta^{(L)} = (a^{(L)} - y)$$

For the remaining layers ℓ, the **chain rule** implies

$$\begin{aligned}
\frac{\partial J}{\partial \Theta^{(\ell-1)}} &= \left(\frac{\partial J}{\partial a^{(\ell)}} \right) \frac{\partial a^{(\ell)}}{\partial z^{(\ell)}} \frac{\partial z^{(\ell)}}{\partial \Theta^{(\ell-1)}} \\
&= \left(\frac{\partial J}{\partial a^{(\ell+1)}} \frac{\partial a^{(\ell+1)}}{\partial z^{(\ell+1)}} \frac{\partial z^{(\ell+1)}}{\partial a^{(\ell)}} \right) \frac{\partial a^{(\ell)}}{\partial z^{(\ell)}} \frac{\partial z^{(\ell)}}{\partial \Theta^{(\ell-1)}} \\
&= \left(\frac{\partial J}{\partial a^{(\ell+1)}} g'(z^{(\ell+1)}) \Theta^{(\ell)} \right) g'(z^{(\ell)}) a^{(\ell-1)}
\end{aligned}$$

Note that $\frac{\partial J}{\partial \Theta^{(\ell-1)}}$ recursively depends on $\frac{\partial J}{\partial a^{(\ell)}}, \frac{\partial J}{\partial a^{(\ell+1)}}, \cdots$ of higher layers. Hence we can find $\frac{\partial J}{\partial \Theta^{(\ell-1)}}$ recursively by

$$\begin{aligned}
\frac{\partial J}{\partial \Theta^{(\ell-1)}} &= \delta^{(\ell)} a^{(\ell-1)} \\
\delta^{(\ell)} &= \Theta^{(\ell)} \delta^{(\ell+1)} g'(z^{(\ell)})
\end{aligned}$$

References

[1] Murphy, Machine Learning – A Probabilistic Perspective (2012)

[2] Goodfellow et al., Deep Learning (2016).

[3] Andrew Ng, CS229 Machine Learning, Stanford.

[4] Fei-Fei Li, CS231n Convolutional Neural Network for Visual Recognition, Stanford.

[5] arXiv:1803.08823. A high-bias, low-variance introduction to Machine Learning for physicists.