Theory

Bayes classifier

g is the *classifier*.

$$g : \mathcal{X} \to \mathcal{Y}$$
$$\mathbb{R}^d \to \{0, 1\}$$

To model the learning problem, we use the pair (X, Y) described by (μ, η) where μ is the probability measure:

$$\mu(A) = \mathbb{P}(X \in A)$$

And η is the regression of Y on X:

$$\eta(X) = \mathbb{P}(Y = 1 | X = x) = \mathbb{E}[Y | X = x]$$

η is also called the *a posteriori probability*.
The Bayes classifier is:

$$\begin{cases} 1 & \text{if } \eta(x) > 1/2 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Or, if $\mathcal{Y}$ is $\{-1, 1\}$, we write the classifier as such: $g(x) = 2\mathbb{1}\{\eta(x) > 1/2\} - 1$.

Theorem:

For any classifier g: $\mathbb{R}^d \to \{0, 1\}$,

$$\mathbb{P}(g^*(X) \neq Y) \leq \mathbb{P}(g(X) \neq Y)$$

In other words, the Bayes classifier is theorically **the best classifier**.

Proof: express $\mathbb{P}(g(X) \neq Y) - \mathbb{P}(g^*(X) \neq Y)$ in terms of dummies (use complementaries) and show that it is superior to 0.

Gradient descent

ℓ the loss function to minimize:
$\theta_{t+1} = \theta_t - \alpha \nabla \ell$

Algorithm 1 Global gradient descent

Loss function $\widehat{L}_n(\hat{f}_\omega(x)) = \Sigma_{i=1}^n \ell(\hat{f}_\omega(x), y)$
$E = 1000$
ϵ = small value
ω_0 = intial value in t_0
while $E > \epsilon$ **do**
    $\omega_{t+1} = \omega_t - \epsilon \Sigma_{i=1}^n \nabla_\omega \ell(\hat{f}_\omega(x), y)$
    Compute $E = L_n(\omega_{t+1})$
end while

Algorithm 2 Stochastic gradient descent

Loss function $\widehat{L}_n(\hat{f}_\omega(x)) = \Sigma_{i=1}^n \ell(\hat{f}_\omega(x), y)$
$E = 1000$
ϵ = small value
ω_0 = intial value in t_0
while $E > \epsilon$ **do**
    for i = 1,...,n **do**
        $\omega_{t+1} = \omega_t - \epsilon \nabla_\omega \ell(\hat{f}_\omega(x_i), y_i)$
        Compute $E = L_n(\omega_{t+1})$
    end for
end while

Algorithm 3 Stochastic and random gradient descent

Loss function $\widehat{L}_n(\hat{f}_\omega(x)) = \Sigma_{i=1}^n \ell(\hat{f}_\omega(x), y)$
$E = 1000$
ϵ = small value
ω_0 = intial value in t_0
while $E > \epsilon$ **do**
    for i = 1,...,n **do**
        Random draw of $i \in \{1, ..., n\}$
        $\omega_{t+1} = \omega_t - \epsilon \nabla_\omega \ell(\hat{f}_\omega(x_i), y_i)$
        Compute $E = L_n(\omega_{t+1})$
    end for
end while

The main advantage of stochastic gradient is that it avoid computing the gradient descent on all the observations (greedy). However, in doing so, the gradient descent is subject to noise and can take longer to reach the optimum.

Note: the gradient is the derivation w.r.t ω
Note (stochastic and random gradient descent): the draw can be done with or without replacement.

Proof of the gradient descent formula:

Taylor series by definition

$$C(\theta_{new}) = C(\theta_{old}) + \nabla C \cdot [\theta_{new} - \theta_{old}] + ..$$

Here $\Delta\theta = [\theta_{new} - \theta_{old}]$, $\nabla C = \frac{\partial C}{\partial \theta}$

$$C(\theta_{new}) \approx C(\theta_{old}) + \frac{\partial C}{\partial \theta} \cdot \Delta\theta \qquad (1)$$

If we set $\Delta\theta = -\eta \frac{\partial C}{\partial \theta}$ with η a small positive learning rate

equation (1) becomes:

$$C(\theta_{new}) \approx C(\theta_{old}) + \frac{\partial C}{\partial \theta} \cdot \left(-\eta \frac{\partial C}{\partial \theta}\right) = C(\theta_{old}) - \eta \left(\frac{\partial C}{\partial \theta}\right)\left(\frac{\partial C}{\partial \theta}\right)$$

$C(\theta_{new}) \leq C(\theta_{old})$, since $\eta \left(\frac{\partial C}{\partial \theta}\right)\left(\frac{\partial C}{\partial \theta}\right)$ is always positive

Conclusion: if we set $\Delta\theta = -\eta \frac{\partial C}{\partial \theta}$ it will decrease C

(Idemia courses "DeepLearningTPT2018S1S2.pdf")

Learning rate optimization

Note:
- one epoch = one forward pass and one backward pass of all the training examples.
- batch size = the number of training examples in one forward/backward pass. The higher the batch size, the more memory space needed.
- number of iterations = number of passes, each pass using [batch size] number of examples. To be clear, one pass = one forward pass + one backward pass.

Learning rate decay
Simple idea: reduce the learning rate progressively.
E.g. $1/t$ decay:

$$\alpha_t = \frac{1}{(t+1)}$$

Momentum
Momentum is a method that helps accelerate SGD in the relevant direction and reduce oscillations.
General idea:
$0 \leq \gamma \leq 1$

$M_{t_0} = x_0$
$M_{t_1} = \gamma M_{t_0} + x_1$
$M_{t_2} = \gamma M_{t_1} + x_2$
Let's develop M_{t_2} to have a better view of momentum effect:
$M_{t_2} = \gamma(\gamma M_{t_0} + x_1) + x_2 = \gamma^2 x_0 + \gamma x_1 + x_2$
We can see that more importance is given to the most recent value (x_2) and the least to the past values.

In practice, $\gamma = 0.9$ gives good results.
Advantage: less dependent on noise

Adagrad - Adaptive Gradient Algorithm (2011)
Divide the learning rate by "average" gradient:
$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{\Sigma_{i=0}^{t}(\nabla f_i^2)}} \nabla f$

RMSProp - Root Mean Squared Propagation
Same as AdaGrad, but with an exponentially decaying average of past squared gradients.
$\theta_t = \theta_{t-1} - \frac{\alpha}{\sigma_{t-1}} \nabla f$
Where:
$\sigma_t = \sqrt{\alpha(\sigma_{t-1})^2 + (1-\alpha)(\nabla f_t)^2}$

Adam - Adaptive moment estimation
Adam = RMS + momentum => use of a exponential decaying average of past squared gradients and past gradients

Bias-Complexity trade-off

ERM = Empiric Risk Minimization algorithm
$\mathcal{H}$ = hypothesis class = all the classifiers that are considered

We can decompose the error of an $ERM_{\mathcal{H}}$:

$$L_{\mathcal{D}}(h_s) = \epsilon_{app} + \epsilon_{est}$$

- *Approximation error*: $\epsilon_{app} = min_{h \in \mathcal{H}} L_{\mathcal{D}}(h)$. This is the error done by the best predictor among those considered.
- *Estimation error*: $\epsilon_{est} = L_{\mathcal{D}}(h) - \epsilon_{app}$. This is the error difference from a used predictor and the best one.
ϵ_{app} low => ϵ_{est} high => overfitting
ϵ_{est} low => underfitting

Supervised learning

Supervised learning aims at **finding a predictor when we have data with their labels**.

Perceptron

The Perceptron is the most basic algorithm for binary classification.

We want to estimate $f : \mathcal{X} \to \mathcal{Y}$ where $X \in \mathcal{X} \subset \mathbb{R}^p$ and $\mathcal{Y} = \{-1, 1\}$

Hyperplane

The classification is linear. We look for an hyperplane that separates the best the observations.

$$\mathcal{H} = \{x \in \mathbb{R}^p, \hat{f}_\omega(x) := \omega_0 + \Sigma_{i=1}^p \omega_i x_i = 0\}$$

We can also write:

$$\mathcal{H}_\omega : \omega^T x + \omega_0$$

Perceptron classifier: $x \mapsto sign(\hat{f}_\omega(x))$

Loss function

We want to find ω such that the loss function ℓ is minimised:
$\mathbb{E}[\ell(\hat{f}_\omega(x), y)]$ is the theoric risk to minimize.
Recall that ERM consists of minimizing the empiric risk.

Empiric risk

The following is an example of loss function that can be used for the Perceptron problem:

$$\widehat{L}_n(\hat{f}_\omega(x)) = -\Sigma Y_i(\omega_0 + \Sigma_{i=1}^p \omega_i x_i)$$

(when Y_i (target) and $\omega_0 + \Sigma_{i=1}^p \omega_i x_i$ (predicted) have different signs, the loss function increases)

Logistic regression

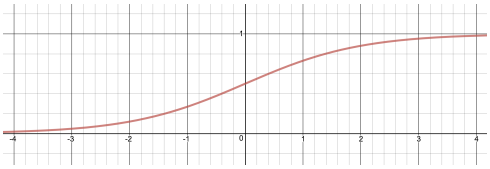
Logistic regression is used for binary classification.
It is quite similar to a simple linear regression in the sense that the objective is to find optimal weights ω to predict a variable. However, in the logistic regression we use a sigmoïd function.

Rem: "logistic" because the logistic law has a sigmoïd function as a repartition function.

Rationale behind the use of the sigmoïd function:
We look for the *à posteriori* probability $\mathbb{P}(x|y = 1) = \pi(x) = \hat{y}$.

The predicted variable $\hat{y}$ is thus a probability.

The sigmoïd function: $\sigma : z \to \frac{1}{1+e^{-z}}$ is well adapted because we want an output that is included in $[0, 1]$.



Classification function: $\widehat{f_\omega}(x) = \sigma(\omega x)$ with a threshold

Loss function
If $y = 1$, we want $\sigma(\omega x)$ to be high $=>$ $1 - \sigma(\omega x)$ should be low. The loss function should be increasing with $1 - \sigma(\omega x) = 1 - \frac{1}{1+e^{-\omega x}} = \frac{1}{1+e^{\omega x}}$. Equivalently, it should be increasing with $1 + e^{-\omega x}$.
More generally, the loss function is defined as such: $\ell(f_\omega, (x, y)) = log(1 + e^{-y\omega x})$ (adding y in the expression allows to take into account cases when $y = 1$ and $y = -1$).
Recall that the log is a monotonic function.

Estimation
The advantage of the logistic loss function is that it is a convex function. Hence the ERM problem can be solved efficiently using santard methods.
Estimation is done using maximum likelihood. Maximum likelihood is finding the parameter that maximizes the probability to have a specific event (x_i, y_i). We want to maximize the *à posteriori* probability that depends on x:

$L(\omega, b) = \prod_{i=1}^{n} \pi(x_i)^{y_i}(1 - \pi(x_i))^{1-y_i}$

This equation has no analytic solution. We use a numeric method to find the optimal parameters (see optimizaton algorithms).
See *Neural Network* section for more details on optimization.

Note: logistic regression is really a linear model since the objective is to find ω that is the slope of the line $\omega^T x + b$.

Linear discriminant analysis

We focus on the binary case, that is when $Y = +1$ or $Y = -1$, that is to sets of variables.
These two conditional laws need to be gaussians with same covariance:
$X|Y = +1 \sim \mathcal{N}(\mu_+, \Sigma)$ with density f_+
$X|Y = -1 \sim \mathcal{N}(\mu_-, \Sigma)$ with density f_-
Let π_+, π_- be the simple probabilities $P(Y = +1)$, $P(Y = -1)$

$\mathbb{P}\{Y = +1|X = x\} = \frac{\mathbb{P}\{Y=+1, X=x\}}{\mathbb{P}\{X=x\}}$
$\mathbb{P}\{Y = +1|X = x\} = \frac{\mathbb{P}\{X=x|Y=+1\}\mathbb{P}\{Y=+1\}}{\mathbb{P}\{X=x\}}$

$\mathbb{P}\{Y=+1|X=x\} = \frac{f_+\pi_+}{\mathbb{P}\{X=x\}}$
$\mathbb{P}\{Y=+1|X=x\} = \frac{f_+\pi_+}{(\mathbb{P}\{X=x|Y=+1\}\mathbb{P}\{Y=+1\}+\mathbb{P}\{X=x|Y=-1\}\mathbb{P}\{Y=-1\})}$
$\mathbb{P}\{Y=+1|X=x\} = \frac{f_+\pi_+}{(f_+\pi_++f_-\pi_-)}$
Similarly,
$\mathbb{P}\{Y=-1|X=x\} = \frac{f_-(1-\pi_+)}{\mathbb{P}\{X=x\}}$
$\mathbb{P}\{Y=-1|X=x\} = \frac{f_-(1-\pi_+)}{(f_+\pi_++f_-\pi_-)}$

The result shows us that we can express the two conditionnal probabilities in terms of conditionnal densities and "simple" probabilities (π_+, π_-).
Recall that multivariable gaussian density is: $f(x) = \frac{1}{\sqrt{2\pi|\Sigma|}}e^{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)}$
In practice, μ_+, μ_-, π_+ and Σ are unknown. Thus we use empiric values:
$\widehat{\pi}_+ = m/n$
$\widehat{\mu}_+ = \frac{1}{m}\Sigma\mathbb{1}_{\{y_i=+1\}}x_i$
$\widehat{\mu}_- = \frac{1}{n-m}\Sigma\mathbb{1}_{\{y_i=-1\}}x_i$
$\widehat{\Sigma} = \frac{1}{n-2}((m-1)\widehat{\Sigma}_+ + (n-m-1)\widehat{\Sigma}_-)$
$\widehat{\Sigma}_+ = \frac{1}{m-1}\Sigma\mathbb{1}_{\{y_i=+1\}}(x_i-\widehat{\mu}_+)(x_i-\widehat{\mu}_+)^T$
$\widehat{\Sigma}_- = \frac{1}{n-m-1}\Sigma\mathbb{1}_{\{y_i=-1\}}(x_i-\widehat{\mu}_-)(x_i-\widehat{\mu}_-)^T$

Classification
We predict class = 1 when $\mathbb{P}(Y=+1|X) > \mathbb{P}(Y=-1|X)$
$=> \frac{\mathbb{P}(Y=+1|X)}{\mathbb{P}(Y=-1|X)} > 1$
$=> \log(\frac{\mathbb{P}(Y=+1|X)}{\mathbb{P}(Y=-1|X)}) > 0$
=>
Using previous conditional probability expressions, we end up with the following prediction rule:

$$\begin{cases} 1 & \text{if } x^T\widehat{\Sigma}^{-1}(\widehat{\mu}_+ - \widehat{\mu}_-) > \frac{1}{2}\widehat{\mu}_+^T\widehat{\Sigma}\widehat{\mu}_+ - \frac{1}{2}\widehat{\mu}_-^T\widehat{\Sigma}\widehat{\mu}_- + \log(1-m/n) - \log(m/n) \\ -1 & \text{otherwise} \end{cases} \quad (2)$$

$\widehat{\mu}_+$, $\widehat{\mu}_-$, $\widehat{\pi}_+$ and $\widehat{\Sigma}$ will be computed with *train data*.
x is the *test data*.

Note: $\widehat{\Sigma}^{-1}(\widehat{\mu}_+ - \widehat{\mu}_-)$ is the **Fisher function** (Saporta).

Listing 1: LDA algorithm

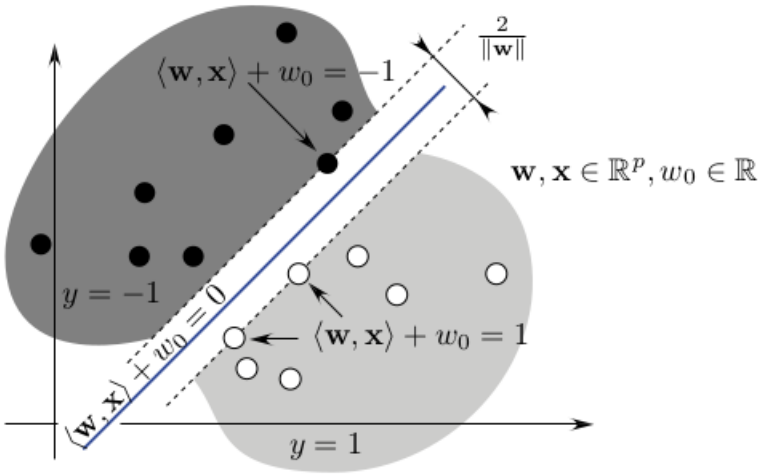
```
class LDAClassifier():

    def fit(self, X, y):

        X_p = X[y == 1, :]
        X_m = X[y == -1, :]

        X_p_x1 = X_p[:,0]
        X_p_x2 = X_p[:,1]
        X_m_x1 = X_m[:,0]
        X_m_x2 = X_m[:,1]

        n = len(X)
```

```
        m = len(X_p)

        mean_p_x1 = np.mean(X_p_x1)
        mean_p_x2 = np.mean(X_p_x2)
        mean_p = np.array([mean_p_x1,mean_p_x2]) # mu_plus (estimated)
        cov_p = np.cov(np.transpose(X_p))


        mean_m_x1 = np.mean(X_m_x1)
        mean_m_x2 = np.mean(X_m_x2)
        mean_m = np.array([mean_m_x1,mean_m_x2]) # mu_minus (estimated)
        cov_m = np.cov(np.transpose(X_m))

        cov_est = (1/(n-2))*( (m-1)* cov_p + (n-m-1)* cov_m) # sigma (estimated)
        inv_cov_est = np.linalg.inv(cov_est)

        a1 = np.dot(np.transpose(mean_p),inv_cov_est)
        a2 = np.dot(np.transpose(mean_m),inv_cov_est)

        # 2nd term in inequality
        self.alpha = 0.5*(np.dot(a1,mean_p)  - 0.5*np.dot(a2,mean_m)) + np.log(1- m/n) - np.log
        # 1st term in inequality
        self.beta =  np.dot(inv_cov_est,mean_p-mean_m)

        return self

    def predict(self, X):

        y_=[]

        for i in range(len(X)):
            X_pred = X[i]
            beta = np.dot(np.transpose(X_pred), self.beta)
            if (beta>self.alpha):
                Y_pred = 1
            else:
                Y_pred = -1
            y_.append(Y_pred)
        return np.array(y_)
```

SVM

Margin

This idea of SVM is to separate data as best as possible using a margin.



Three planes:
- $H_1 : \omega^T x + b = 1$
- $H : \omega^T x + b = 0$
- $H_{-1} : \omega^T x + b = -1$

Computing $H_1 - H_{-1}$ we get:
$(x_1 - x_{-1})\omega^T = 2$
$=> ||x_1 - x_{-1}|| = \frac{2}{||\omega||}$

The SVM problem starts with a **margin maximization**. We want to maximize the distance between x_1 and x_{-1} **(to be double checked)** and it is equivalent to minimizing $||\omega||$. Thus the optimization problem is written as such:

$$min_{\omega,b} \ \frac{1}{2}||\omega||^2$$
$$s.t. \ y_i(\omega^T x_i + b) \geq 1 \ i = 1, ..., n$$

Primal formulation

The above problem reflects the case where all data are perfectly separable, this is not true in practice. We thus add an error term ξ_i for each observation. This leads to the **primal formulation** of the problem:

$$min_{\omega,b,\xi} \ \frac{1}{2}||\omega||^2 + C\Sigma_{i=1}^n \xi_i$$
$$s.t. \ y_i(\omega^T x_i + b) \geq 1 - \xi_i \ i = 1, ..., n$$
$$\xi_i \geq 0 \ i = 1, ..., n$$

Note: the problem can be rewritten with the hinge loss function
$y_i(\omega^T x_i + b) \geq 1 - \xi_i => \xi_i \geq 1 - y_i(\omega^T x_i + b)$
$=> \xi_i \geq 1 - y_i(\omega^T x_i + b) \geq 0$ since $\xi_i \geq 0$
$=> \xi_i = max(0, 1 - y_i(\omega^T x_i + b))$
$=> \xi_i = (0, 1 - y_i(\omega^T x_i + b))_+$
$=> \xi_i = hinge(f(x))$ where *hinge* is the *Hinge loss* function
$hinge(f(x)) = (1 - yf(x))_+$

$$min_{\omega,b}\ \frac{1}{2}||\omega||^2 + C\Sigma_{i=1}^{n}(0, 1 - y_i(\omega^T x_i + b))_+$$

Lagrange

We can write this problem using Lagrange formulation, that is, integrating the constraints into the main formula:
$\mathcal{L}(\omega, b, \xi, \alpha, \mu) = \frac{1}{2}\omega^T\omega + C\Sigma\xi_i + \Sigma\alpha_i(1 - \xi_i - y_i(\omega^T x_i + b)) - \Sigma\mu_i\xi_i$
$\alpha_i, \mu_i \geq 0$

First order conditions:
$\frac{\partial\mathcal{L}}{\partial\omega} = 0 => \omega - \Sigma\alpha_i y_i x_i = 0 => \omega = \Sigma\alpha_i y_i x_i$
$\frac{\partial\mathcal{L}}{\partial b} = 0 => -\Sigma\alpha_i y_i = 0$
$\frac{\partial\mathcal{L}}{\partial\xi_i} = 0 => C - \alpha_i - \mu_i = 0 => \alpha_i = C - \mu_i$
Since $\alpha_i, \mu_i \geq 0$ we have $C \geq \alpha_i \geq 0$

Dual formulation

We can rewrite the problem using the first order conditions above:
$$\begin{aligned}\mathcal{L}(\omega, b, \xi, \alpha, \mu) &= \frac{1}{2}(\Sigma\alpha_i y_i x_i)^T(\Sigma\alpha_i y_i x_i) + C\Sigma\xi_i + \Sigma\alpha_i - \Sigma\alpha_i\xi_i - \Sigma\alpha_i y_i(\Sigma\alpha_i y_i x_i)^T x_i - \Sigma\alpha_i y_i b - \Sigma\mu_i\xi_i \\ &= -\frac{1}{2}(\Sigma\alpha_i y_i x_i)^T(\Sigma\alpha_i y_i x_i) + \Sigma(C - \alpha_i - \mu_i)\xi_i + \Sigma\alpha_i - \Sigma\alpha_i y_i b \\ &= -\frac{1}{2}\Sigma_{i,j}\alpha_i\alpha_j y_i y_j x_i^T x_j + \Sigma\alpha_i\end{aligned}$$

The problem is convex with lineary inequality constraints, we can apply the saddle point theorem.
Note: a point is saddle if it's a maximum w.r.t. one axis and a minimium w.r.t. another axis
The saddle theorem allows us to solve the problem $min_\omega max_\alpha$ as $max_\alpha min_\omega$

This leads to the problem in its **dual formulation**:

$$\begin{gathered}max_\alpha \quad -\frac{1}{2}\Sigma_{i,j}\alpha_i\alpha_j y_i y_j x_i^T x_j + \Sigma\alpha_i \\ s.t.\ \ 0 \leq \alpha_i \leq C \quad i = 1, ..., n \\ \Sigma\alpha_i y_i = 0 \quad i = 1, ..., n\end{gathered}$$

Using the above expression of ω (optimal condition), the classification function is:

$$f(x) = sign(\Sigma\alpha_i y_i x_i^T x + b)$$

Kernel

Kernels are used when separation is non linear.
Recall the primal formulation:

$$\begin{gathered}min_{\omega,b,\xi}\ \frac{1}{2}||\omega||^2 + C\Sigma_{i=1}^{n}\xi_i \\ s.t.\ y_i(\omega^T x_i + b) \geq 1 - \xi_i\ i = 1, ..., n \\ \xi_i \geq 0\ i = 1, ..., n\end{gathered}$$

When separation is non linear, we set ϕ as a non linear transformation. The constraint becomes: $y_i(\omega^T\phi(x_i)+b) \geq 1-\xi_i \quad i=1,...,n$

Dual formulation is now:

$$\begin{array}{c} max_\alpha \quad -\frac{1}{2}\Sigma_{i,j}\alpha_i\alpha_j y_i y_j \phi(x_i)^T\phi(x_j) + \Sigma\alpha_i \\ s.t. \quad 0 \leq \alpha_i \leq C \quad i=1,...,n \\ \Sigma\alpha_i y_i = 0 \quad i=1,...,n \end{array}$$

The classification becomes:

$$f(x) = sign(\Sigma\alpha_i y_i \phi(x_i)^T\phi(x) + b)$$

To classify a new point, we thus need to be able to compute $\phi(x_i)^T\phi(x)$.

Kernel trick: there is no need to know an explicit expression of ϕ (i.e. knowing the coordinates of points in new set) since we are only looking at distances and angles, that is scalar product.

Kernel functions implement those scalar products: $K(x,x') = \phi(x)^T\phi(x')$ where ϕ is a transformation function into a Hilbertian set $\phi : \mathcal{X} \to \mathcal{F}$
Note: an Hilbertian is a set with scalar product: $\mathcal{F} = (\mathcal{H}, <.,.>)$

Most used kernels:
Linear kernel: $K(x,x') = x^Tx'$ (we often call this setup a "no-kernel SVM")
Polynomial kernel: $K(x,x') = (x^Tx'+c)^d$
Gaussian kernel: $K(x,x') = exp(-\gamma||x-x'||^2)$

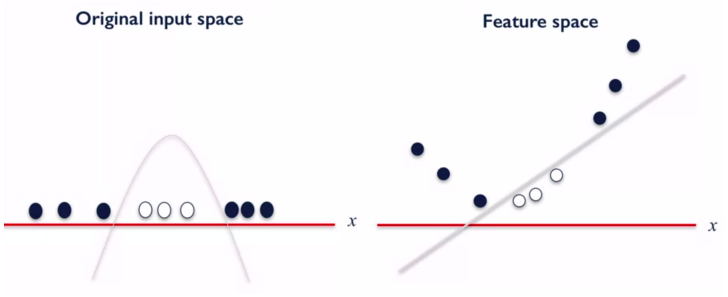
The optimisation problem can be written with kernel:

$$\begin{array}{c} max_\alpha \quad -\frac{1}{2}\Sigma_{i,j}\alpha_i\alpha_j y_i y_j K(x_i,x_j) + \Sigma\alpha_i \\ s.t. \quad 0 \leq \alpha_i \leq C \quad i=1,...,n \\ \Sigma\alpha_i y_i = 0 \quad i=1,...,n \end{array}$$

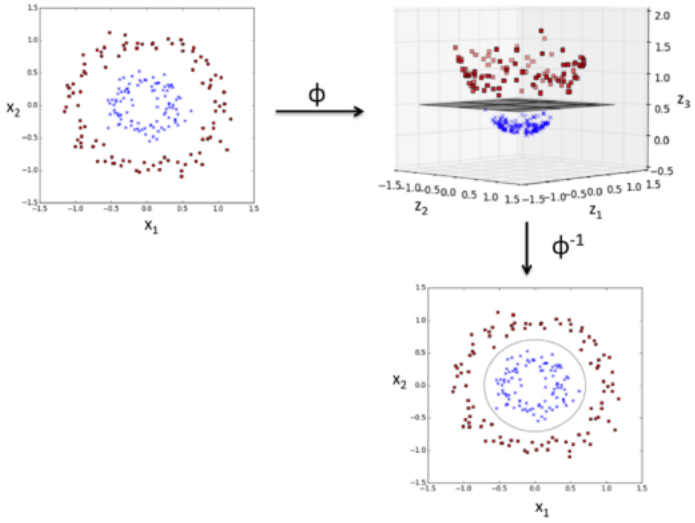
Summary

SVM allows to find complex non linear separations in transforming the problem into a higher dimension where data are linearly separable.

From 1D to 2D:



From 2D to 3D:



Neural network

Logistic regression with a neural network mindset

Based on the example from coursera (*deeplearning.ai*)

Let us take the example of an image that we want to classify in a **binary** way: man/woman

The picture is vectorized as a vector of pixels : $\begin{pmatrix} x_1 \\ ... \\ x_p \end{pmatrix}$

We use a regression to predict if it's a man/woman:

$y = \omega^T x + b$

Note: x are all the pixels of **one** image.

We want a probability in output (if it's ≥ 0.5 then we say it's a man).

We thus want the output to be $\widehat{y} = \sigma(\omega^T x + b) = \mathbb{P}(y|x) \in [0, 1]$

(see regression part to get more details on the sigmoid)

Now since it's a binary classification, we want the y (real value) to be 0 or 1.

Thus, the loss function is:

$$\mathcal{L}(y, \widehat{y}) = -y \log(\widehat{y}) + (1 - y) \log(1 - \widehat{y})$$

Proof: TODO from binary law (Bernoulli) to cross-entropy

The cost function is the empiric loss on all examples:

$$J(\omega, b) = \frac{1}{m} \Sigma_{i=1}^{m} \mathcal{L}(\widehat{y}^{(i)}, y^i)$$

Forward propagation

$$x_1, x_2, \omega_1, \omega_p, b \rightarrow z = \omega_1 x_1 + \omega_2 x_2 + b \rightarrow \widehat{y} = a = \sigma(z) \rightarrow \mathcal{L}(a, y)$$

- First arrow: regression
- Second arrow: probability
- Third arrow: error

Backward propagation
The idea is: with the error computed on the last step, we go backward in order to correct the parameters ω and b.

$$x_1, x_2, \omega_1, \omega_p, b \leftarrow z = \omega_1 x_1 + \omega_2 x_2 + b \leftarrow \widehat{y} = a = \sigma(z) \leftarrow \mathcal{L}(a, y)$$

Example: we want to find ω_1 that minimizes the cost function:
$\frac{d\mathcal{L}}{d\omega_1} = "d\omega_1" = \frac{d\mathcal{L}}{da}\frac{da}{dz}\frac{dz}{d\omega_1} = ... = (a-y)x_1 = dzx_1$
Steps:
We compute all the derivatives, then we apply the gradient descent

Listing 2: Gradient descent (logistic regression with a NN mindset)

```
for i in range(num_iterations):

    # Cost and gradient calculation
    grads, cost = propagate(w, b, X_train, Y_train) # propagation on ALL the training sample

    # Retrieve derivatives from grads
    dw = grads["dw"]
    db = grads["db"]

    # update parameters
    w = w - learning_rate * dw
    b = b - learning_rate * db

    # Record the costs
    costs.append(cost)
```

Listing 3: Propagation (logistic regression with a NN mindset)

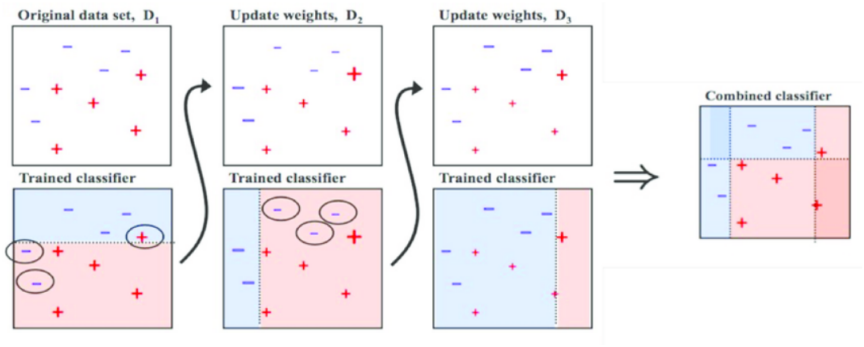
```
def propagate(w, b, X, Y):

    m = X.shape[1]

    # FORWARD PROPAGATION (FROM X TO COST)
    A = sigmoid(np.dot(w.T,X)+b)
    cost = (- 1 / m) * np.sum(Y * np.log(A) + (1 - Y) * (np.log(1 - A))

    # BACKWARD PROPAGATION (TO FIND GRAD)
    dw = (1/m)*np.dot(X,(A-Y).T)
    db = (1/m)*np.sum(A-Y)
```

AdaBoost

Boosting is an algorithmic paradigm addressing two major issues in machine learning:

- It optimizes the bias-complexity trade-off. The learning starts with a basic class (large approximation error) and as it progresses the hypothesis class becomes more complex.
- It allows to find predictors that are usually computationally infeasible to find.

Main idea: weak learners are "boosted" to become stronger altogether.



Weak learner (or γ-weak-learner): it's an **algorithm** returning a function h such that $L_{\mathcal{D}}(h) \leq 1/2 - \gamma$. In other words, it returns a simple binary predictor that does slightly better than a random guess.

Algorithm 4 AdaBoost

Input: training set $S = (x_1, y_1), ..., (x_m, y_m)$, weak learner WL, number of rounds T
Initialize $D^{(1)} = (1/m, ..., 1/m)$ // same weights for all observations
for t = 1,...,T **do**
  invoke weak learner $h_t = WL(D^{(t)}, S)$
  compute $\epsilon_t = \Sigma_{i=1}^{m} D_i^{(t)} \mathbb{1}_{[y_i \neq h_t(x_i)]}$ // misclassification error
  let $\omega_t = \frac{1}{2} log(\frac{1}{\epsilon_t} - 1)$ // $\epsilon_t < 1$
  update $D_i^{(t+1)} = \frac{D_i^{(t)} exp(-\omega_t y_i h_t(x_i))}{\Sigma_{j=1}^{m} D_j^{(t)} exp(-\omega_t y_j h_t(x_j))}$ for all $i = 1, ..., m$
end for
Output: the hypothesis $h_s(x) = sign(\Sigma_{t=1}^{T} \omega_t h_t(x))$

We note:
- Final predictor = weighted sum of weak predictors
- More weights are given to observations that gave wrong prediction. In doing so, the classifier of the next round will focus on these observations. Warning: to see this, focus on the variation of $D_i^{(t+1)}$ and not just w_t.

Theorem: the training error of the output hypothesis decreases **exponentially fast** with the number of boosting rounds.

Unsupervised learning

Unsupervised learning aims at **learning some underlying hidden structure of the data**

when we don't have the labels.
Unsupervised models can be used as a pre step for supervised learning, e.g.:
- reduce the training sample (dimensionality reduction: forward selection, PCA, autoencoders)
- give output for unlabeled data (clustering, autoencoders)
- grow the training sample (generative models? autoencoders? variational encoders?)

Expectation-Maximization (EM) in the case of GMM (Gaussian Mixture Model)

(for more details, see document *gmm.pdf* in Cloud folder)

GMM problem aims at estimating parameters of a sample distribution. ("Generative models" p. 295 Understanding Machine Learning).

A GMM sample is composed of j Gaussian variables (*clusters*) distributed with proportions $(\pi_1, ..., \pi_k)$ $(\Sigma\pi_i = 1)$
We can write:

$$X \sim \mathcal{N}(\mu_Z, \Sigma_Z) \qquad with \;\; Z \sim \pi$$

π is not really a law but more the proportions of each Gaussian categories.
Thus, X has a density which is a weighted-average of all Gaussian densities:

$$p_\theta(x) = \Sigma_{j=1}^k \pi_j f_j(x) \qquad (*)$$

Estimation
We want to estimate $\theta = (\pi, \mu, \Sigma)$ where:
$\pi = (\pi_1, ..., \pi_k)$, $\mu = (\mu_1, ..., \mu_k)$, $\Sigma = (\Sigma_1, ..., \Sigma_k)$

To do so, we use the maximum likelihood method (product of densities across all samples):

$$p_\theta(x) = \Pi_{i=1}^n p_\theta(x_i)$$

$$l(\theta) = log(\Pi_{i=1}^n p_\theta(x_i)) = \Sigma_{i=1}^n log(p_\theta(x_i))$$

We thus need to find $argmax(l(\theta))$
Problem: the likelihood function is not convex!

The expectation-maximization problem is used when we have *latent variables* (= variables for which we don't know their associated distribution).
Let $z = (z_1, ..., z_k)$ be the vector of latent variables. We can express the density $(*)$ as a joint function with respect to z:

$$p_\theta(x, z) = p_\theta(z) p_\theta(x|z)$$

$$l(\theta, z) = ... = \Sigma(log\pi_{z_i}) + \Sigma(log f_{z_i}(x_i))$$

A classic optimization (in case of Gaussians) give us empirical values as solutions e.g. $\hat{\pi_j} = \frac{n_j}{n}$
Problem: we don't know j!

We will thus use the *expected* log-likelihood method.
Let us find another expression of the likelihood:

$$p_\theta(x, z) = p_\theta(x)p_\theta(z|x)$$

As seen previously: $p_\theta(x, z) = \Pi\pi_{z_i} f_{z_i}(x_i)$
$p_\theta(z|x) = \Pi p_\theta(z_i|x_i) = \frac{\Pi\pi_{z_i} f z_i(x_i)}{p_\theta(x_i)} \propto \Pi\pi_{z_i} f z_i(x_i)$

Given an initial parameter θ_0, the *expected* log-likelihood is written as such:

$$\mathbb{E}_{\theta_0}[l(\theta; z)] = \Sigma p_{\theta_0}(z|x) l(\theta; z)$$
$$\mathbb{E}_{\theta_0}[l(\theta; z)] = \Sigma_j \Sigma_i p_{ij}(log\pi_j + log f_j(x_i))$$

We now have an expression that doesn't depend on z but only on p_{ij} and we know that $n_j = \Sigma_i p_{ij}$

K-means

(see kmeans.pdf from OneDrive folders for more details)

Objective: group data into k clusters so that samples in the same cluster are close to each other w.r.t. the Euclidean distance.

The cost function minimization is written as such:

$$\underset{C_1,...,C_k;\mu_1,...,\mu_k}{\operatorname{argmin}} \Sigma_{j=1}^{k} \Sigma_{i \in C_j} ||x_i - \mu_j||^2$$

Where μ_j is the mean, also called gravity center or cluster center:

$$\mu_j = \frac{1}{|C_j|} \Sigma_{i \in C_j} x_i$$

The quality of the clustering strongly depends on the initial center values. This is why the algorithm is generally run multiple times for different initial values. The best clustering (i.e., that of minimum cost) is returned.

K-means++
To improve the quality of the clustering, we choose the initial cluster centers far from each other:
- select the first cluster center uniformly at random among the n data samples
- select the following cluster centers at random, with a probability proportional to the square distance to the closest current cluster center

Listing 4: K-means++ initial centers selection

```
centers = []
centers.append(X[np.random.randint(X.shape[0])]) # inital center = one random sample
distance = np.full(X.shape[0], np.inf) # a vector (n,1) with only infinity terms
for j in range(1, self.n_clusters):
    distance = np.minimum(np.linalg.norm(X - centers[-1], axis=1), distance) # size (n,1);
# distance = the smallest distance associated with
# the last added center
```

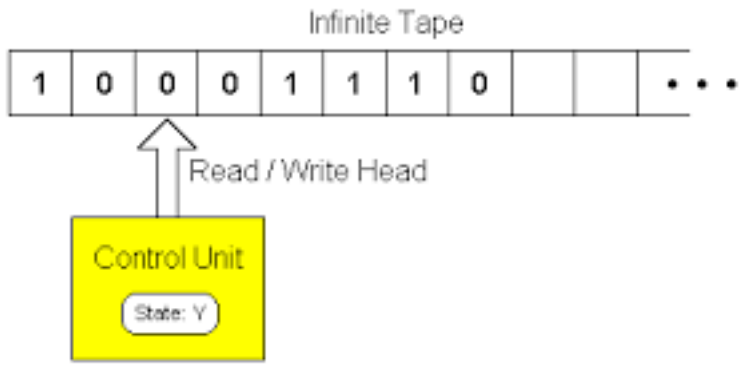
Algorithm 5 K-means

Input: $x_1, ..., x_n$
Output: clusters $C_1, ..., C_k$
Random values for $\mu_1, ..., \mu_k$
while no convergence **do**
  // Step 1: Update clusters
  $C_1, ..., C_k \leftarrow \emptyset$
  for $i = 1$ to n **do**
    $j \leftarrow argmin_l ||x_i - \mu_l||$
    $C_j \leftarrow C_j + \{i\}$ // We add observation i to the cluster C_j
  end for
  // Step 2: Update cluster centers
  for $j = 1$ to k **do**
    $\mu_j \leftarrow 0$
    $n_j \leftarrow 0$
    for i in C_j **do** // We loop on all observations of each cluster
      $\mu_j \leftarrow \mu_j + x_i$
      $n_j \leftarrow n_j + 1$
    end for
    $\mu_j \leftarrow \mu_j / n_j$
  end for
end while

```
    p = np.square(distance) / np.sum(np.square(distance)) # probability vector [p1,...,pn]
# the highest probability in p is associated
# with the biggest distance w.r.t the last added center
    sample = np.random.choice(X.shape[0], p = p) # one sample is
                                                 # selected according to probabilities
    centers.append(X[sample])
```

Note: this problem is called *NP-hard problem*. It means that its complexity is at least equal to the complexity of an NP-problem
NP-problem: a problem is NP if it can be determined by a non-deterministic Turing machine in polynomial time. Intuitively, a problem is NP if we can quickly verify if one candidate is a solution of the problem. E.g. "travelling salesman problem" = let d be a distance and n be a number of cities. Is there an itinerary with distance $\geq d$ stopping by every city? -> easy to check...

Turing machine (1936)



"non-deterministic turing machine": itinerary can be represented by a tree...

Local Outlier Factor

Local Outlier Factor is an unsupervised method used in anomaly detection. It consists of comparing local density of train observations VS local density of test observations.

Reachability distance

$reachability\text{-}distance_k(A, B) = max\{k\text{-}distance(B), d(A, B)\}$ = reachability of A *from* B.

$k\text{-}distance(B)$: distance from B to its kth nearest neighbor.
The reachability distance of A from B is *at least* the distance between A and B or *at least* the distance of B's neighbor.
When A is very far from B, it's simply the distance between the two points.
When A is very close to B, it's the distance between B and its neighbor.

The distance can be computed using different metrics: Euclidean distance, Mahalanobis distance, etc.

Local reachability density

$$lrd_k(A) = \frac{1}{\Sigma_{B \in N_k(A)} reachability\text{-}distance_k(A,B)/|N_k(A)|}$$

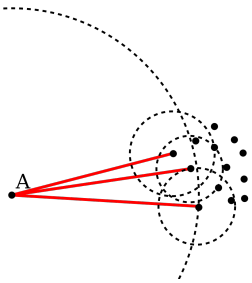
It's the inverse of the average of reachability-distances of A from B.
When A is very far from its neighbors: sum of reachability distances is high => local reachability density is small.

Local Outlier Factor

LOF computation consists of comparing the local densities of a point VS its neighbors.

$$LOF_k(A) := \frac{\frac{\Sigma_{B \in N_k(A)} lrd_k(B)}{lrd_k(A)}}{|N_k(A)|}$$

$LOF_k(A) > 1$: A is an outlier. Local reachability density of A is small compared to its neighbors.
$LOF_k(A) < 1$: A is an inlier.



On this figure, $k = 3$. We can see that the reachability distance of A from its neighbors is high (red segments). The local reachability density of A will thus be **low**.
On the contrary, the local reachability densities of its neighors is **high** because each neighbor can be easily reached from their own neighbors.
As a result, LOF would be high so A is an outlier.

sklearn algorithm

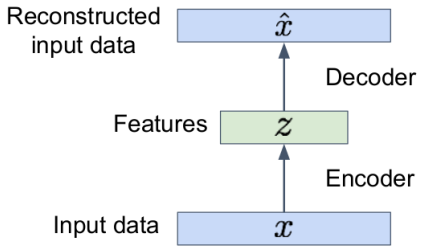
To score an observation, *fit* simply memorizes the train observations (same as in knn).
score_samples first finds the k-nearest neighbors from the train set thanks to the given distance metric. It then computes the local outlier factor for each test observation comparing the test observation local density with its closest k-neighbors local densities in the train set.

Variational Auto-Encoder

Variational autoencoders are a combination of three things:
1. Autoencoders
2. Variational Approximation & Variational Lower Bound
3. "Reparameterization" Trick

1. Autoencoders
Autoencoders are used to extract features from unlabeled training data. They are new methods for **dimensionality reduction** and part of neural networks branch.

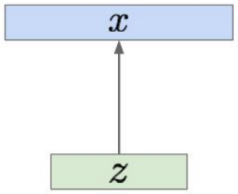
Note: autoencoders can be used to replace older dimensionality reduction methods such as PCA for several reasons:
- For very large data sets that can't be stored in memory, PCA will not be able to be performed. The autoencoder construction using keras can easily be batched resolving memory limitations.
- PCA is restricted to linear separation while autoencoders are capable of modelling complex non linear functions.



Learning can be done using a loss function such as $||x - \hat{x}||^2$. In a similar way than neural network, optimization is typically done with backpropagation.

2. Variational Approximation & Variational Lower Bound

We assume x is generated from unobserved (latent) z:



Practical example: x can be seen as images and z as the main attributes (orientation, colors, etc.)

$x \sim p_{\theta^*}(x|z)$ where $p_{\theta^*}(x|z)$ is called *true conditional*
$z \sim p_{\theta^*}(z)$ where $p_{\theta^*}(z)$ is called *true prior*

Objective: estimating $p_\theta(x)$. We thus need to estimate θ^*.

We can do it through maximum likelihood. The marginal density is $p_\theta(x) = \int p_\theta(x|z)p_\theta(z)dz$

Note: a marginal likelihood function is a likelihood function in which some parameter variables have been marginalized. Marginalization consists in summing over the possible values of one variable in order to determine the contribution of another. E.g., $\mathbb{P}(X) = \Sigma_y \mathbb{P}(X, Y = y)$ or in continuous probabilities $p(x) = \int p(x, y)dy$. Also, if we don't know the joint probability, we can express this using conditional probabilities: $p(x) = \int p(x|y)p(y)dy$

Problem: impossible to compute $p(x|z)$ for every z (computationally too expensive) => problem is said **intractable**

Solution: use another encoder learning $q_\phi(z|x)$ that approximates $p_\theta(z|x)$

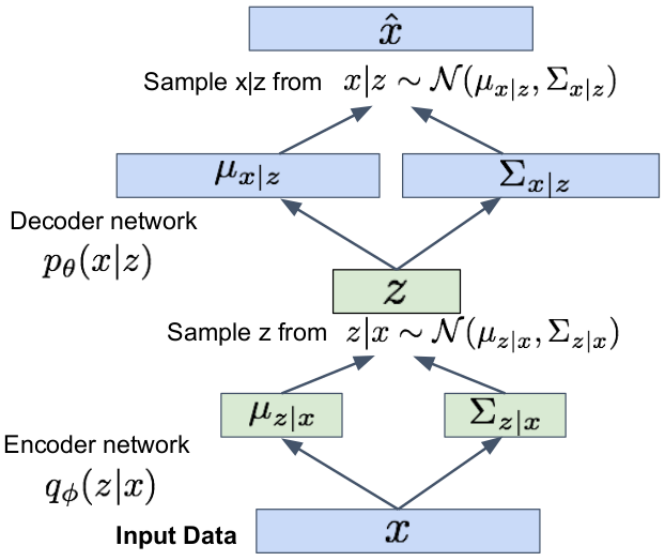
$$
\begin{aligned}
\log p_\theta(x^{(i)}) &= \mathbf{E}_{z\sim q_\phi(z|x^{(i)})}\left[\log p_\theta(x^{(i)})\right] \quad (p_\theta(x^{(i)}) \text{ Does not depend on } z) \\
&= \mathbf{E}_z\left[\log \frac{p_\theta(x^{(i)} \mid z)p_\theta(z)}{p_\theta(z \mid x^{(i)})}\right] \quad \text{(Bayes' Rule)} \\
&= \mathbf{E}_z\left[\log \frac{p_\theta(x^{(i)} \mid z)p_\theta(z)}{p_\theta(z \mid x^{(i)})}\frac{q_\phi(z \mid x^{(i)})}{q_\phi(z \mid x^{(i)})}\right] \quad \text{(Multiply by constant)} \\
&= \mathbf{E}_z\left[\log p_\theta(x^{(i)} \mid z)\right] - \mathbf{E}_z\left[\log \frac{q_\phi(z \mid x^{(i)})}{p_\theta(z)}\right] + \mathbf{E}_z\left[\log \frac{q_\phi(z \mid x^{(i)})}{p_\theta(z \mid x^{(i)})}\right] \quad \text{(Logarithms)} \\
&= \mathbf{E}_z\left[\log p_\theta(x^{(i)} \mid z)\right] - D_{KL}(q_\phi(z \mid x^{(i)}) \,||\, p_\theta(z)) + D_{KL}(q_\phi(z \mid x^{(i)}) \,||\, p_\theta(z \mid x^{(i)}))
\end{aligned}
$$

$\mathbb{E}_z[\log p_\theta(x^{(i)}|z]$: we can estimate this term through sampling
$D_{KL}(q_\phi(z|x^{(i)})||p_\theta(z))$: differentiable term
$D_{KL}(q_\phi(z|x^{(i)})||p_\theta(z|x^{(i)}))$: $p(z|x)$ intractable but we know that $D_{KL} \geq 0$

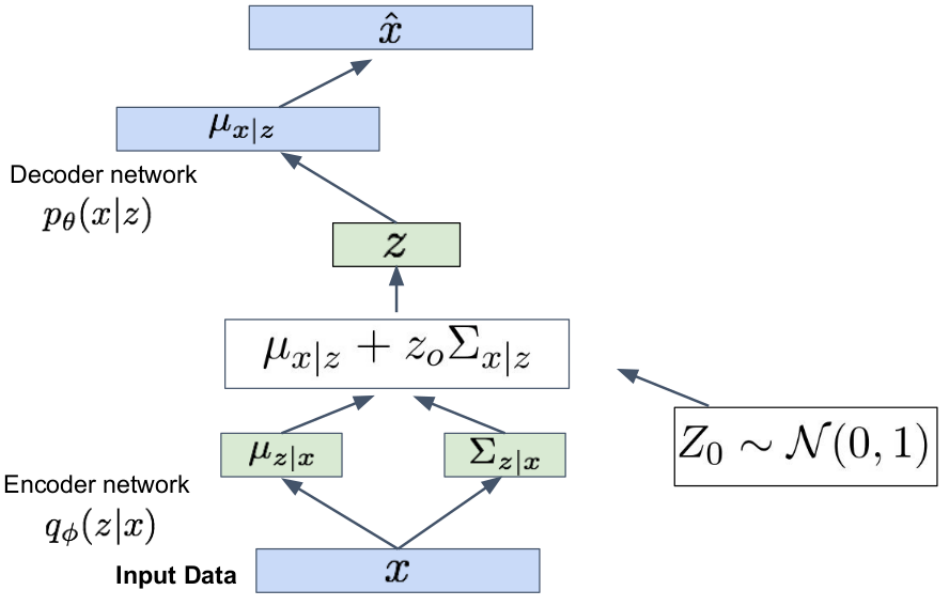
Let $\mathcal{L}(x^{(i)}, \theta, \phi) = \mathbb{E}_z[\log p_\theta(x^{(i)}|z] - D_{KL}(q_\phi(z|x^{(i)})||p_\theta(z))$ = **tractable lower bound** that we can optimize

We know that $p_\theta(x^{(i)}) \geq \mathcal{L}(x^{(i)}, \theta, \phi)$ since $D_{KL}(q_\phi(z|x^{(i)})||p_\theta(z|x^{(i)})) \geq 0$
Thus the maximum likelihood problem becomes: $\theta^*, \phi^* = argmax_{\theta,\phi}\Sigma_{i=1}^{N}\mathcal{L}(x^{(i)}, \theta, \phi)$

We can minimize $D_{KL}(q_\phi(z|x^{(i)})||p_\theta(z))$ making posterior distribution close to prior. To do so, we make encoder network predicting $\mu_{z|x}$ and $\Sigma_{z|x}$ and then we sample $z|x \sim \mathcal{N}(\mu_{z|x}, \Sigma_{z|x})$



Problem: sampling $z|x \sim \mathcal{N}(\mu_{z|x}, \Sigma_{z|x})$ and $x|z \sim \mathcal{N}(\mu_{x|z}, \Sigma_{x|z})$ is not differentiable (why?).
=> we use **reparametrization trick**: we sample $z_0 \sim \mathcal{N}(0, 1)$ to have $z = \mu_{x|z} + z_0\Sigma_{x|z} \sim \mathcal{N}(\mu_{x|z}, \Sigma_{x|z})$



Optimization through forward and backward propagation!

Reinforcement learning

Reinforcement learning is inspired on human logic: we learn which strategy to take thanks to rewards we receive.

Reinforcement learning uses Markov Decision Processes (MDP).
A Markov Decision Process is defined by:
- an initial state s_0
- the reward distribution $r_t \sim p(r|s_t, a_t)$ (stochastic)
- the transition probabilities, $s_{t+1} \sim p(s|s_t, a_t)$ (stochastic)
=> **MDP: we know the state and the reward from the previous state only**.

A *policy* is an action for each state, for a given MDP.
=> **a policy can be seen as a strategy: we know where to go at each state**. In practice the policy is actually the **transition probability matrix**.

The *value function* is the gain we earn at a state, for a specific policy: $\forall s, V_\pi(s) = \mathbb{E}_\pi[G|s_0 = s]$
=> **The expected gain takes into account the transition probability.**
The gain is calculated as such: $G = r_0 + \gamma r_1 + \gamma^2 r_2 + ... = \Sigma_t \gamma^t r_t$ where γ is the *discount factor*. V can also be written $V_\pi(s) = \mathbb{E}_\pi[r_0 + \gamma V(s_1)|s_0 = s]$. This expression is called **Bellman equation**.

We can find the best policy:

$\forall s, \pi^*(s) = a^* \in argmax_a \mathbb{E}_\pi[r_0 + \gamma V_*(s_1)|s_0 = s, a_0 = a]$
Where V_* is solution of the **Bellman optimality equation**:
$\forall s, V(s) = max_a \mathbb{E}[r_0 + \gamma V(s_1)|s_0 = s]$

Online estimation

The expected value function is approached using empiric estimator (sum).
Two ways to do it:
- Monte-Carlo update: if we can memorize all the paths, we update the sum at each step $S \leftarrow x_t$. At the end we compute the mean $X \leftarrow \frac{S}{t}$
- TD-learning: we update the value function using temporal differences $\forall s, V(s_t) \overset{\alpha}{\leftarrow} r_t + \gamma V(s_{t+1})$
Where $X \overset{\alpha}{\leftarrow} x_t <=> X = X + \alpha(x_t - X)$ (α is usually $1/t$)

Online control

Recall that value function is $\forall s, V_\pi(s) = \mathbb{E}[G|s_0 = s]$
Unknown model <=> "unknown strategy" <=> we don't know the rewards in advance (we need to learn *online*).
=> we cannot compute the expectation $\mathbb{E}$
=> instead of the value function, we will estimate the **value-action function** thanks to the Bellman equation:

$$\forall s, a, \qquad Q_\pi(s, a) = \mathbb{E}_\pi[r_0 + \gamma Q(s_1, a_1)|s_0 = s, a_0 = a]$$

=> for an initial state and a fixed action, we can compute the Q function
=> the value-action function can be seen as the expected (=> use of transition matrix (probabilities)) gain we get at a state when doing a specific action
Note: the policy π takes account only from next state s_1

We want to *control* the policy and find the optimal one: $\pi^*(a) = a^* \in argmax_a Q_*(s, a)$

The optimal Bellman equation becomes:

$$\forall s, a, \qquad Q_\pi(s, a) = \mathbb{E}_\pi[r_0 + \gamma max_{a'} Q(s_1, a')|s_0 = s, a_0 = a]$$

How to choose initial state a_0?
–> Pure exploitation: we find the best action knowing the current system $\pi(s) \leftarrow argmax_a Q(s, a)$
=> problem: Q is estimated, thus we have a chance to miss good actions
–> Pure exploration: $\pi(s) \leftarrow random$
=> problem: we can waste time on bad actions and thus have bad estimation quality
Solution: trade-off exploitation/exploitation (SARSA, Q-learning, ...).

SARSA

This algorithm is based on ϵ-greedy algorithm.

$$\pi(s) \leftarrow \begin{cases} a^* \in argmax_a Q(s, a) \text{ with probability } 1 - \epsilon \\ random \text{ with probability } \epsilon \end{cases} \tag{3}$$

=> with a small probability (ϵ is usually lower than 5%), we do random actions. Otherwise we take the best action of the known ones.

Estimation is done through *TD-learning* (temporal differences - see Online Estimation):

$$\forall t, Q(s_t, a_t) \overset{\alpha}{\leftarrow} r_t + \gamma Q(s_{t+1}, a_{t+1})$$

Listing 5: SARSA

```
# action is identified with new state (after move) except teleportation (action = same state)

def sarsa(Q, model = model, alpha = 0.1, eps = 0.1, n_iter = 100):
    states = model.states
    terminals = model.terminals
    rewards = model.rewards
    gamma = model.gamma
    # random state (not terminal)
    state = np.random.choice(np.setdiff1d(np.arange(len(states)), terminals))
    # random action
    action = np.random.choice(Q[state].indices)
    new_state = action
    for t in range(n_iter):
        state_prev = state
        action_prev = action
        state = new_state
        if state in terminals: # if the new action gives terminal state
                # we start again from a new random state
                    state = np.random.choice(np.setdiff1d(np.arange(len(model.states)),
                                                          terminals))
                    action = np.random.choice(Q[state].indices)
                    Q[state_prev, action_prev] =
                (1 - alpha) * Q[state_prev, action_prev] + alpha * rewards[action_prev]
                  # we removed the second term since state_prev was the last step
        else:
                  # we choose the best action that has the highest expected value-action
                    best_action = Q[state].indices[np.argmax(Q[state].data)]
                    if np.random.random() < eps:
                # with probability of epsilon, we choose a random action
                        action = np.random.choice(Q[state].indices)
                    else:
                        action = best_action
                    Q[state_prev, action_prev] =
                        (1 - alpha) * Q[state_prev, action_prev]
                        + alpha * (rewards[action_prev] + gamma * Q[state, action])
        new_state = action
    return Q
```

Q-learning

This algorithm is also based on ϵ-greedy algorithm.

$$\pi(s) \leftarrow \begin{cases} a^* \in argmax_a Q(s,a) \text{ with probability } 1-\epsilon \\ random \text{ with probability } \epsilon \end{cases} \tag{4}$$

Estimation: unlike SARSA, Q-learning aims at updating the estimator using the best action at each iteration:

$$\forall t, Q(s_t, a_t) \overset{\alpha}{\leftarrow} r_t + \gamma max_a Q(s_{t+1}, a)$$

The only modification from SARSA is the following line:

Listing 6: Q-Learning

```
Q[state_prev, action_prev] = (1 - alpha) *  Q[state_prev, action_prev]
                        + alpha * (rewards[action_prev] + gamma * np.max(Q[state].data))
```