Machine Learning Review

Francesco Saverio Zuppichini

November 6, 2017

The aim of this report is to summarise the topics threaded in my Machine Learning class at USI.

1 History of Machine Learning

Write something about the history of ML

2 Gradient Descend

The gradient descend is an iterative optimisation algorithm that follows the direction of the negative gradient in order to minimised an objective function. It can be effectively used as Learning Algorithm because it reduces the error function, Equation **??**, and adjusts the weights properly. Equation **??** shows the generic update rule.

$$w_{k+1} = w_k - \eta \nabla E(w_k) \tag{1}$$

Where η is the step size, also called **learning rate** in Machine Learning. This parameter influences the behaviour of gradient descent, a small number can lead to local minimum, while a bigger learning rate could "over-shoot" and decreasing the converge ration. Later in this project you will see how a wrong η can strongly change the output of a Neural Network.

For this reasons, numerous improvements have been proposed to avoid local minima and increase its convergence ration, some of them are: Conjugate Gradient and Momentum.

2.1 Loss Functions

In order to apply gradient descend we need a loss function to minimise, usually they are divided by problem type;

- **For Regression**
  Mean Square Error:

$$E = \frac{1}{N} \sum_{i=1}^{N} (\underbrace{y_i}_{\text{predicted}} - \underbrace{t_1}_{\text{actual}})^2 \tag{2}$$

- **For Classification**
  SoftMax Function:

$$y_j = \frac{e^{x^T w_j}}{\sum_{k=1}^{K} e^{x^T w_k}} \tag{3}$$

with negative log likelihood:

$$-\sum_{i=1}^{K} t_i log(y_i) \tag{4}$$

3 Perceptron

3.1 Definition

The **Perceptron** is binary **linear classifier** algorithm used in **supervised learning**. It can be seen as the most basic form of Neural Network. Equation 5 defines the its output.

$$f(x) \begin{cases} 1 & \text{if w } \cdot x + b \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Given a training set $D = \{(x_1, t_1, ..., (x_n, t_i)\}$, $x_i \in X$ and $t_i \in Y$ denotes the input vector and the target vector respectively. We express $y = f(x)$ as the output of the algorithm, w the weight and b the bias. At each iteration the error is calculated using the Mean Square Error, defined in equation 2 The algorithm uses stochastic **Gradient Descent** in order to update the weight at each iteration using the formula defined in Equation 1.

$$\frac{\partial E}{\partial w_k} = y - t \tag{6}$$

4 Neural Network

A **Neural Network** is a universal function approximation. It is a nested composite functions like $f(g((h...)))$. In its simplest representation, an FeedForward Neural Network, it is composed by a **input layer**, an **hidden layer** and an **output layer**. The size of the hidden layer is usually refers as the **deepth** of the network.

4.1 Forward pass

In order to get the prediction out of our network we need to calculate the compute the activation at each layer l. Equation 7 shows the activation a of layer l for the j-th neuron on that layer.

$$a_j^l = \sigma(\sum_k w_{jk}^l a_k^{l-1} + b_j^l) \tag{7}$$

Where w_{jk}^l is the connection from neuron k in the $l-1$ layer to j, a^{l-1} is the activation of the previous layer and b_j^l is the bias of j-th neuron in the l layer. With this in mind, we can rewrite 8 in a efficient vectorised form

$$a^l = \sigma(W^l a^{l-1} + b^l) \tag{8}$$

4.2 Delta rules

In a Neural Network the weights are iteratively changed in order to decrease the cost function, called E. We want to find out how much they should be updated, in order to do so we need the output error at each layer. Equation 9 defines δ_j^l as the **output error** of neuron j in layer l

$$\delta_j^l = \frac{\partial E}{\partial z_j^l} \tag{9}$$

Strictly speaking, δ_j^l, is how much the error function changes by changing the weighted input on that layer. Applying the chain rule, Equation 9 becomes:

$$\delta_j^l = \frac{\partial E}{\partial a_j^l}\frac{\partial a_j^l}{\partial z_j^l} \tag{10}$$

By knowing that $a_j^l = \sigma(z_j^l)$, Equation 10 can be expressed as:

$$\delta_j^l = \frac{\partial E}{\partial a_j^l}\sigma'(z_j^l) \tag{11}$$

Also, delta at layer l can be expressed by using the next $l+1$-th delta. Equation 12 shows the new rule.

$$\delta^l = (W^{l+1}\delta^{l+1}) * a^l \tag{12}$$

4.3 Back Propagation

The **Back Propagation** algorithm defines an efficient and interactive method to calculate the gradient at each layer. We want to compute $\frac{\partial E}{\partial w_{jk}^l}$, by applying the delta rule:

$$\frac{\partial E}{\partial w_{jk}^l} = \frac{\partial E}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} = \frac{\partial E}{\partial a_j^l}\frac{\partial a_j^l}{\partial z_j^l} \tag{13}$$

After some calculation we derive Equation 14 that shows the partial derivative of the error function with respect to the weight of the l-th layer for the j-th neuron using the error output in that layer.

$$\frac{\partial E}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \tag{14}$$

While 15 shows how to calculate the derivative of the error with respect to the bias.

$$\frac{\partial E}{\partial b_j^l} = \delta_j^l \tag{15}$$

We know from Equation 12, δ^l can be calculated used the $l+1$-th layer's information, this is why is called *back* propagation since we used the last layer information in order to, each time, find out the next layer's delta until we reach the input layer.

5 Convolutional Neural Network

Convolutional neural networks are neural network that uses a **convolutional** operation at place in at least one layer. They are used for processing grid-like data, such as time-series, a 1D grid and images, 2D grid. Figure 1 shows a the generic architecture for a CNN.

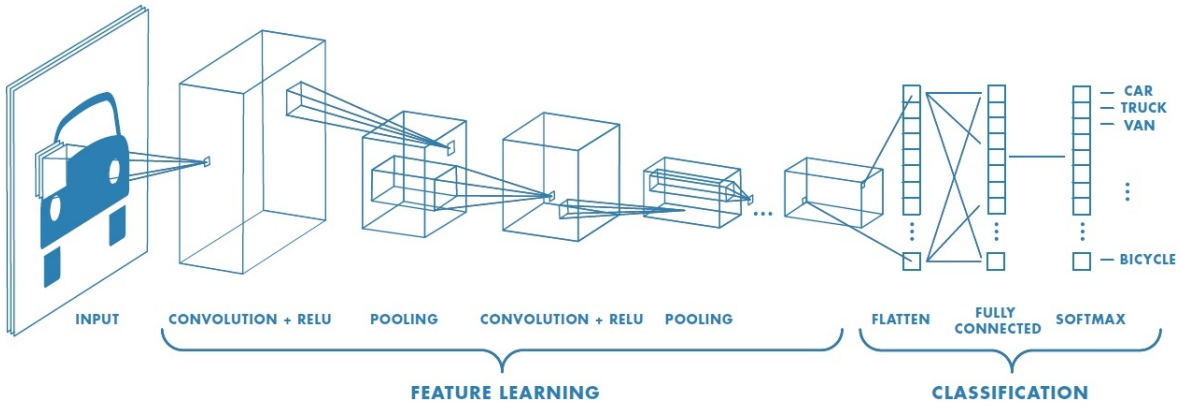


Figure 1

5.1 Notation

We express an Image as a function $f : Z^2 \rightarrow R^c$, where c is the number of channel, in a color image there are three (rgb). A Windows is a subset of the image domain, $W \in Z^2$, that corresponds to a rectangle inside the image.

5.2 Convolutional Layer

As in a hidden layer, a Convolutional layer is formed by n neurons. The difference is that they are not necessarily connect to the activations of the neurons in the previous layer, but only in a particular window. The architecture is composed as following:

- A input $[w * h * c]$ holds the an image
- A convolutional layer computes the output of its neurons that are connected to some window in the input, each computes the dot product between its weight and the region they are connected to. Each neuron apply a filter, usually of size $3x3$ or $5x5$.
- A rectifier function, RELU $max(0, x)$, is applied in order set to zero the not discovered features.
- A max-pool layer perform a downsampling operation, storing only the most relevant feature (the biggest number) in a little window size (usually $2x2$). Figure 2 shows a visual representation of this operation

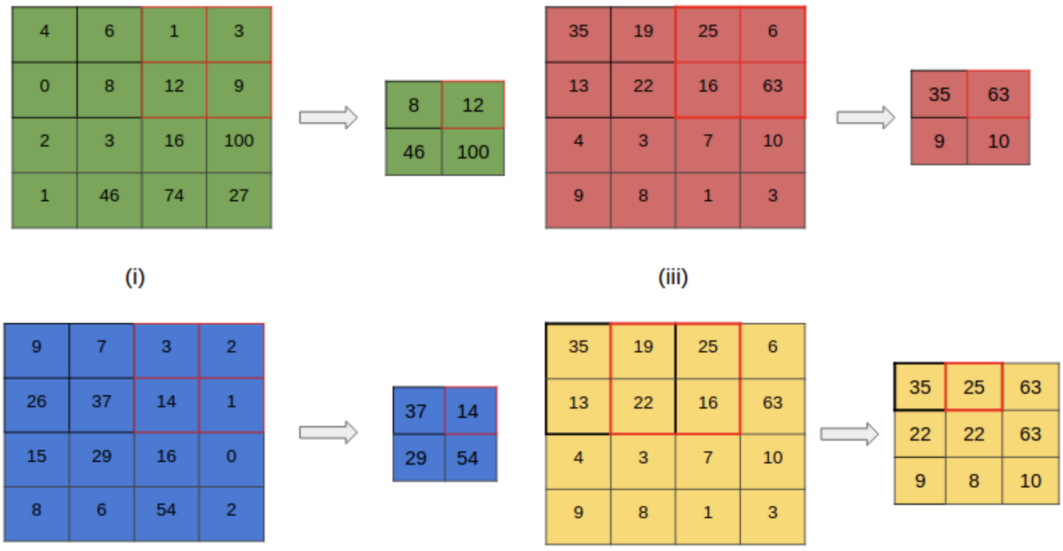


Figure 2

- A fully connected layer computes the class scores using softmax, Equation 3 in order to equally distribute the probabilities.

After each CONV $\rightarrow$ RELU $\rightarrow$ POOL layer the network learn more complex pattern each time, Figure 3 shows an example where in the first layer the network can only recognise little lines and in the last one it is able to merge all the previous information and classify very complex pattern such as a human face

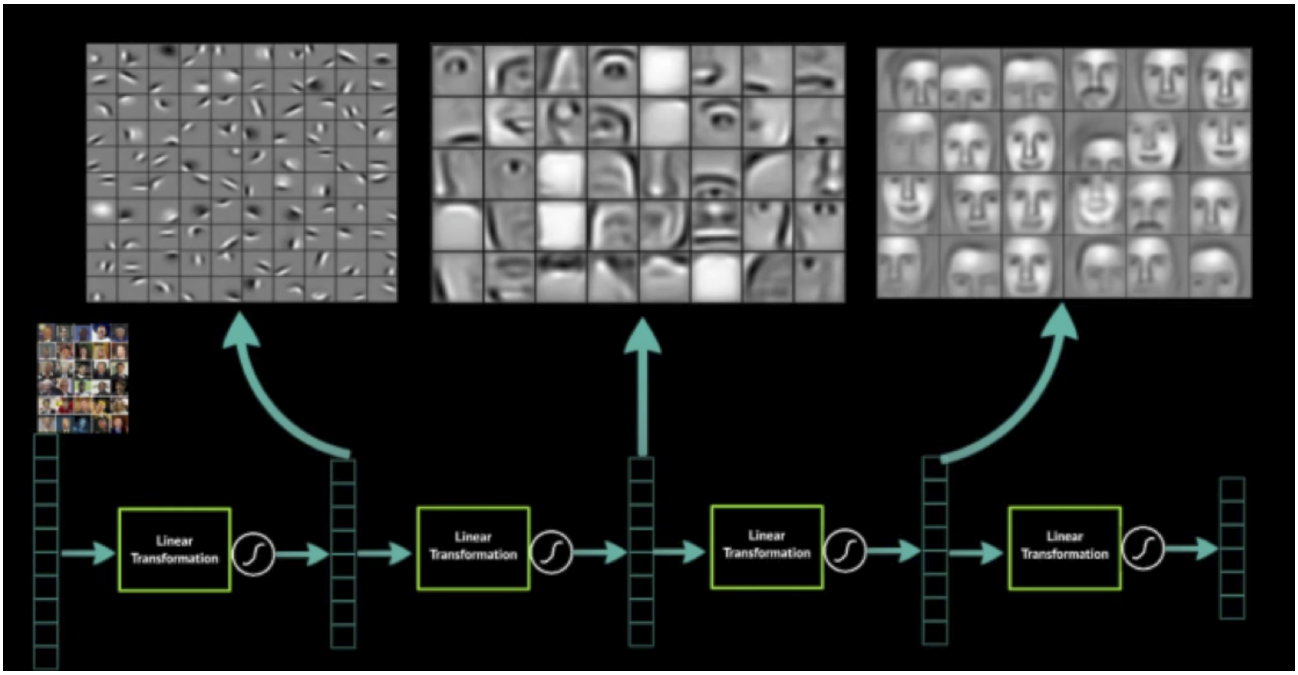


Figure 3

The **pooling** layer is used to get out from the convolutional layer only the most important features, in order to do so usually a $2X2$ matrix is

6 Recurrent Neural Network

6.1 Definition

A Recurrent Neural Networks can remember past decision by taking as input not only the current input but also the last time state. For this reason it is said that a RNN has **memory**. Figure 4 shows a classic representation. Usually, a RNN is represented unfolded to highlight the time dependencies. Due to its ability to remember it mostly used in text and speech recognition.

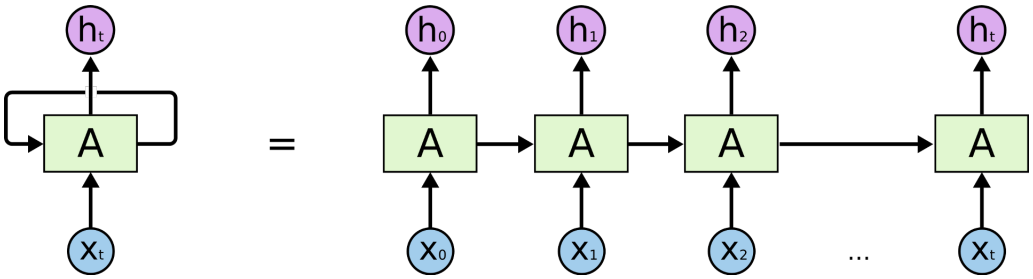


Figure 4: Fold and Unfold representation of a RNN

Very similar to a feedforward network, Equation 16 shows the weighted output at layer j. The first term on the right-hand is just the feedforward's weighted output, while the second term is the time-dependent term. Matrix ω is a hidden-state-to-hidden-state matrix. Basically, we are adding previous informations to our new state at time t.

$$z[t]^l = (W^l a[t]^{l-1} + b_j^l) + (\omega^l a[t-1]^{l-1}) \tag{16}$$

Equation 17 shows the activation of layer j at time t.

$$a[t]^l = \sigma(z[t]^l) \tag{17}$$

While Equation 18 shows the updating rule for the weight w^l

$$\begin{aligned} \frac{\partial E}{\partial w[T] l_{jk}} &= \sum_{t=0}^{T} \frac{\partial E}{\partial z[t]_j^l} \\ \frac{\partial E}{\partial w[T] l_{jk}} &= \sum_{t=0}^{T} a[t]_k^{l-1} \delta[t]_j^l \end{aligned} \tag{18}$$

6.2 Loss

Usually in a RNN the loss used function is the **cross-entropy loss**. For example, suppose we have K samples with each sample labeled by $i, ...K$. The loss function is then given by Equation 4

6.3 Vanishing Gradient Problem

Since the network layers and time steps are related to each other through multiplication, the gradient becomes smaller and smaller at each t. Figure 5 shows the effect of applying *sigmoid* function over time. The data is flatted more and more at each step and therefore the slope will $\to 0$.

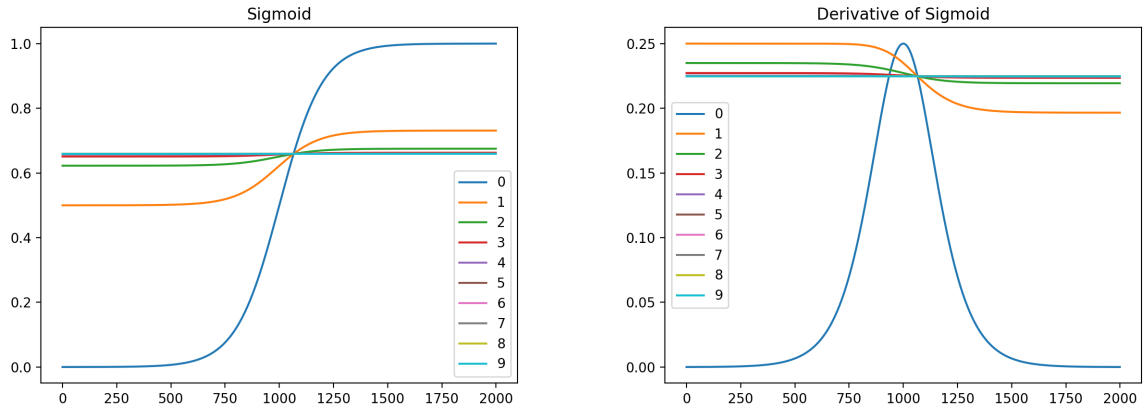


Figure 5: Sigmoid and its derivative over time

Since gradient expresses the change in all weights with respected to the error, without nothing it we cannot adjust properly the network.

7 Long Short Term Memory

7.1 Definition

The Long Short Term Memory networks, or just **LSTM**, are a special type of RNN capable of learning long-term dependencies. They were introduced by Juergen Schmidhuber in order to solve the vanish gradient problem. They are composed by LSTM cell, Figure 6 shows a unrolled representation.

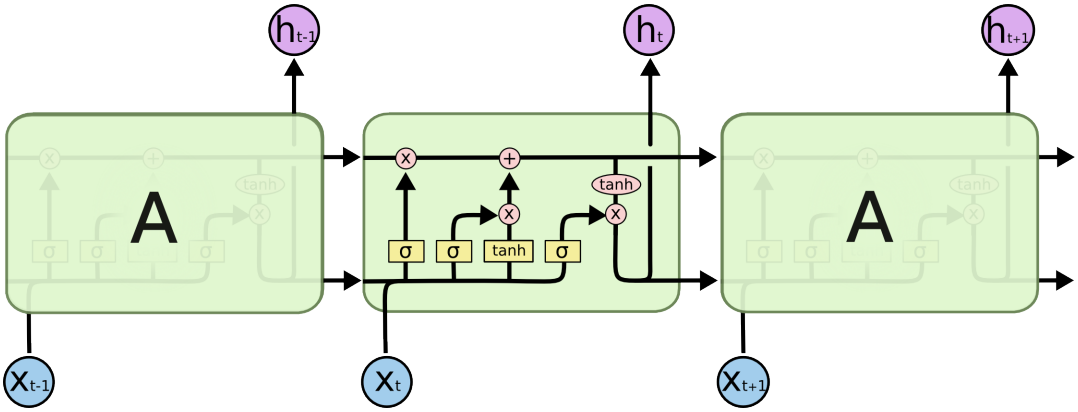
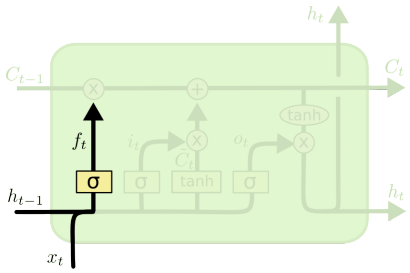


Figure 6: An unrolled LSTM

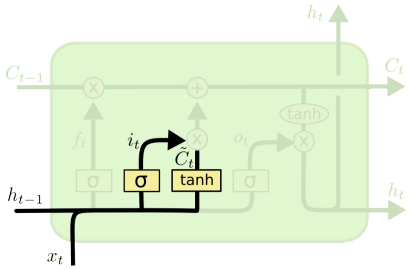
Each cell is composed by 3 gate: **forget** gate (f_t), **input** gate i_t and **output** gate (o_t). It takes as input the previous output h_{t-1} and the old cell state C_{t-1}, it outputs the next prediction and state, h_t and C_t. The cell computes four basic operations:

1. Forget Gate



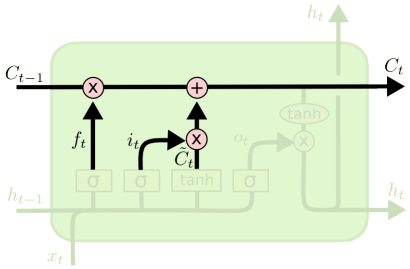
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

2. Input Gate



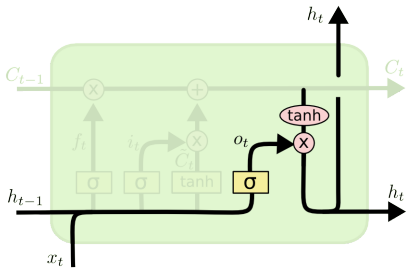
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

3. Update Cell State



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

4. Output Gate



$$o_t = \sigma\left(W_o \; [\, h_{t-1}, x_t] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

8 Support Vector Machine

A Support Vector Machine maps the input space to a high dimensional space, called *feature space*, using a set of non linear function, called the *feature vector*, defined as $\phi(x)$. Then, it constructs an optimal hyperplane in order to separate the features discovered in the *feature set* in order to classify the points that belong to the positive or negative class. Equation 19 shows the decision surface.

$$w^T \phi(x) = 0 \tag{19}$$

Where $\phi(x) = \{\varphi_j(x)\}_j^{\infty}$, is the set of non-linear function, called *feature vector*. It tries to find the **support vectors**, a subset of the data-set, in order to maximise the margin between them using Quadratic Programming.

8.1 Linear separable data

In the simple linear case, where the data is linearly separable, Equation 20 can be re-express by Equation 20, notice that we do not need the non-linear map anymore:

$$w^T x + b = 0 \tag{20}$$

Equation **??** shows the optimal hyperplane for the linear case

$$\begin{array}{lll} w_0^T x + b_0 \geq +1 & \text{for} & d_i = +1 \\ w_0^T x + b_0 \leq -1 & \text{for} & d_i = -1 \end{array} \tag{21}$$

In this case, the **support vectors** is a small subset of points (x_i, d_i) of the dataset for which Equation 21 holds. Figure 7 shows a graphic representation.

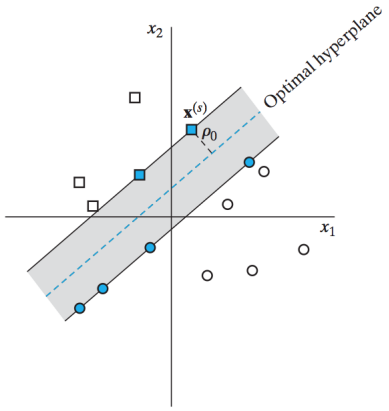


Figure 7: Support Vectors

Therefore we want to find the maximum margin possible, defined in Equation 23.

$$g(x) = wx + b = \pm 1 \tag{22}$$

$$r = \frac{g(x^{(2)})}{||w_o||}$$

$$r = \begin{cases} \frac{1}{||w_o||} & \text{if} \quad d^{(s)} = 1 \\ -\frac{1}{||w_o||} & \text{if} \quad d^{(s)} = -1 \end{cases} \tag{23}$$

$$r = \frac{2}{||w_o||}$$

Where $g(x)$ is the discriminant function, Equation 22, $x^(s)$ are the support vectors and w_0 is the weight. Therefore, maximising the margin is equal to minimising w_0, so we can state our object function and the constrains, defined in Equation 24. This is call **hard-margin**.

$$\begin{array}{ll} \min & \Phi(w) = \frac{1}{2} w^T w \\ \text{subject to} & d_i(w_i^T w_i + b_i) \geq 1 \qquad \text{for} \quad 1 = 1, 2, ..., N \end{array} \tag{24}$$

This basic linear classifier can still be used to classify non-linear separable data by adding a set of non negative variables, *slacks*, $\{\epsilon\}_i^N$ in order to measure the deviation of the data from the ideal

condition of pattern separability. Equation 25 shows the updated equation, this method is called **soft margin**

$$\begin{aligned} \min \quad & \Phi(w) = \tfrac{1}{2} w^T w + c \textstyle\sum_{k=0}^{R} \epsilon_k \\ \text{subject to} \quad & d_i(w_i^T w_i + b_i) \geq 1 - \epsilon_i \qquad \epsilon_i > 0 \quad \text{for} \quad \text{all} \quad i \end{aligned} \tag{25}$$

We skip the dual problems for simplicity, they are very simular to the non linear case only there is no *kernel* and for the case of *soft margin* the Lagrange multipliers must be also less than C. Thus we can express the optimal weight using this new information as follow:

$$w_o = \sum_{i=1}^{N(s)} \lambda_{0,j} d_i x_i^T x^{(s)} \tag{26}$$

There $\lambda_{0,j}$ is a Lagrange multiplier.

8.2 Non linear separable data

As we stated before, a SVM maps from a linear input set to a non-linear *feature set* using the *feature map* to infinite dimensional space, called feature space, then it perform a linear map to the *output state* . Figure 8 shows a graphical representation of a generic SVM

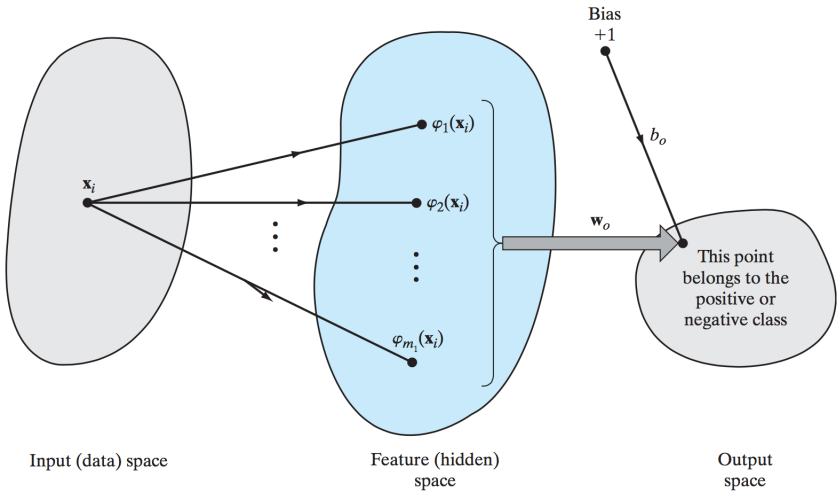


Figure 8: Mapping in a SVM

Specifically, the *feature map* is made by multiplying the inputs point with the *feature vector*, a set of non linear function that transform , defined as follow:

$$\phi(x) = \{\varphi_j(x)\}_{j=1}^{\infty} \tag{27}$$

Keeping in mind the Equation 21, the optimal decise surface, we can re-state the optimal weight Equation 26 as follow:

$$w = \sum_{i=1}^{N^{(s)}} a_i d_i \phi(x_i) \tag{28}$$

Hence, substituting Equation 21, we obtain

$$\sum_{i=1}^{N^{(s)}} a_i d_i \phi(x_i)^T \phi(x) = 0 \tag{29}$$

The term $\phi(x_i)^T \phi(x)$ represent a *inner product*, let us denote it as the scalar:

$$k(x, x_i) = \phi(x_i)^T \phi(x) = \sum_{j=1}^{\infty} \varphi_j(x_i)\varphi_j(x) \tag{30}$$

The function $k(x, x_i)$ is called the **kernel**. According to this, we can re-express the optimal decise surface, Equation 31, by:

$$\sum_{i=1}^{N^{(s)}} a_i d_i k(x_i, x) = 0 \tag{31}$$

Basically, the **kernel** computes the inner product of the images produced in the feature spaces under embedding ϕ of two data points in the input space. Equation 32 states the dual problem.

$$\begin{aligned} \text{Maximixe } \lambda \quad & \sum_{i=1}^{N} \lambda_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \lambda_i \lambda_j d_i d_j k(x_i, x_j) \\ \text{Subject to} \quad & \\ & \sum_{i=1}^{N} \lambda_i d_i = 0 \\ & 0 \leq \lambda_i \leq C \end{aligned} \tag{32}$$

The kernel must be decided *a priori* from the user.

9 Deep Learning

9.1 Supervised Learning

In **supervised** learning there is a fixed training set consists on a set of example patterns and their targets. The objective is to learn a map between the inputs and the targets. 'Supervised' comes from the likeness between a student that tries to answer a question, and a teacher that knows the solution.

Therefore, at each step the algorithm can know how much wrong is prediction was, using a cost function, and it can improve it by applying a learning algorithm such as gradient descent in order to reduce the error.

9.2 Reinforcement Learning

In **reinforcement** learning the *agent* takes *actions* in an *environment* in an to maximise a *reward*. Figure 9 shows the actions flow.

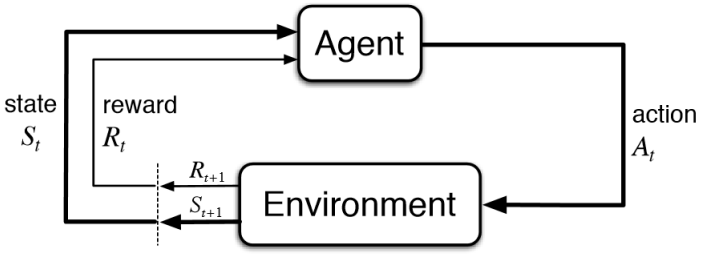


Figure 9: Reinforcement Learning Flow

9.3 Unsupervised Learning

9.4 Training Techniques

9.4.1 Mini-batch

9.5 Regularisation

9.5.1 L1 Regularisation

9.5.2 Dropout Regularisation

9.6 Activations Functions