Exploring relations between MLP, SVMs, and Kernel functions

Internship Report

Satwik Dudeja

Delhi Technological University

Research Intern, Centre for Artificial Intelligence and Robotics, DRDO

10th May 2021 – 30th July 2021

Chapter 1 - Introduction

1. Neural Networks

Artificial Neural Networks are computing systems that are meant to simulate the working of a human brain. With the help of experience and data, a neural network improves itself using artificial neurons. ANN consist of elementary units called as neurons which takes one or more inputs and produces an output. At each node, the following computations are carried out.

$$Z^{[i]} = W^{[i]} * A^{[i-1]} + b^{[i]}$$
$$A^{[i]} = f\left(Z^{[i]}\right)$$

$$Where - W^{[i]} - Weights\ of\ the\ connection$$
$$A^{[i-1]} - Output\ from\ the\ previous\ layer$$
$$b^{[i]} - bias\ of\ the\ connection$$
$$f(x) - Non\ linear\ activation\ function$$

A neural network comprises of input, hidden and output layers. A layer is a group of parallel neurons without any interaction between them. Neurons in the input layer are connected to the layer of hidden units, which are then connected to neurons in the output layer.

Gradient descent based back propagation technique is used to tune the weights and the bias to improve the accuracy of the model.

2. Backpropagation and Gradient Descent

Backpropagation is the algorithm in supervised learning of neural networks for backward propagation of errors. After every feed-forward epoch, the calculation of the gradient is done in a reverse manner, where weights of the final layer are calculated and used for subsequent calculations for the layers moving backwards. This enables efficient calculation of gradient, and optimisation of weights and biases.

The equations used in the backpropagation algorithm for gradient and loss computations is listed -

$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

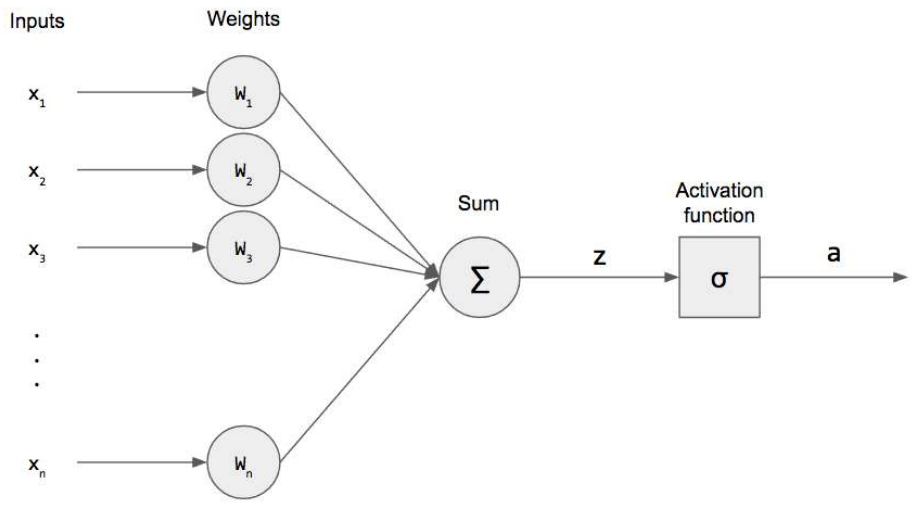
$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$$

3. Perceptron

In the perceptron model inputs can be real numbers unlike the Boolean inputs in MP Neuron Model. The output from the model will still be binary {0, 1}. The perceptron model takes the input **x** if the weighted sum of the inputs is greater than threshold **b** output will be 1 else output will be 0.



Algorithm: Perceptron Learning Algorithm

$P \leftarrow$ *inputs with label* 1;
$N \leftarrow$ *inputs with label* 0;
Initialize **w** randomly;
while !*convergence* **do**
  Pick random $\mathbf{x} \in P \cup N$;
  if $\mathbf{x} \in P$ *and* $\sum_{i=0}^{n} w_i * x_i < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$;
  end
  if $\mathbf{x} \in N$ *and* $\sum_{i=0}^{n} w_i * x_i \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$;
  end
end
//the algorithm converges when all the inputs are classified correctly

4. Python Implementation

The function model takes input values **x** as an argument and perform the weighted aggregation of inputs (dot product between **w.x**) and returns the value 1 if the aggregation is greater than the threshold **b** else 0. Next, we have the predict function that takes input values **x** as an argument and for every observation present in **x**, the function calculates the predicted outcome and returns a list of predictions.

Finally, we will implement fit function to learn the best possible weight vector **w** and threshold value **b** for the given data. The function takes input data(**x** & **y**), learning rate and the number of epochs as arguments.

```
class Perceptron:

    def __init__ (self):
        self.w = None
        self.b = None

    def model(self, x):
        return 1 if (np.dot(self.w, x) >= self.b) else 0

    def predict(self, X):
        Y = []
        for x in X:
            result = self.model(x)
            Y.append(result)
        return np.array(Y)

    def fit(self, X, Y, epochs = 1, lr = 1):

        self.w = np.ones(X.shape[1])
        self.b = 0

        accuracy = {}
        max_accuracy = 0

        wt_matrix = []

        for i in range(epochs):
            for x, y in zip(X, Y):
                y_pred = self.model(x)
                if y == 1 and y_pred == 0:
                    self.w = self.w + lr * x
                    self.b = self.b - lr * 1
                elif y == 0 and y_pred == 1:
                    self.w = self.w - lr * x
                    self.b = self.b + lr * 1

            wt_matrix.append(self.w)
            accuracy[i] = accuracy_score(self.predict(X), Y)
            if (accuracy[i] > max_accuracy):
                max_accuracy = accuracy[i]
                j = i
                chkptw = self.w
                chkptb = self.b

        self.w = chkptw
        self.b = chkptb
        return j+1
```

5. Limitations of a Perceptron

1. It is only a linear classifier, can never separate data that are not linearly separable.
2. The algorithm is used only for Binary Classification problems.

Chapter 2 – Comparing Neural Networks over MATLAB and the Keras API

Different Training and Learning functions from MATLAB, and Keras Optimizers

1. MATLAB

trainlm	*trainscg*
The Levenberg-Marquardt algorithm, also known as the damped least-squares method, has been designed to work specifically with loss functions, which take the form of a sum of squared errors. The Jacobian with the derivatives of the errors - $$\boldsymbol{J}_{i,j} = \frac{\partial e_i}{\partial \boldsymbol{w}_j}$$ This defines the algorithm – $$w^{i+1} = w^{i} - \left(J^{(i)T}.J^{(i)} + \lambda^{(i)}I\right)^{-1}.\left(2J^{(i)T}.e^{(i)}\right)$$ The parameter λ is initialized to be large so that the first updates are small steps in the gradient descent direction. If any iteration happens to result in a fail, then λ is increased by some factor. This process typically accelerates the convergence to the minimum.	Scaled Conjugate Gradient is fully-automated, includes no critical user-dependent parameters, and avoids a time consuming line search. SCG has been shown to be considerably faster than standard backpropagation and other conjugate gradient methods. A conjugate gradient method will proceed in a direction which is **conjugate** to the directions of the previous steps. Thus the minimization performed in one step is not partially undone by the next, as it is the case with standard backpropagation and other gradient descent methods.

trainbfg
trainbfg can train any network as long as its weight, net input, and transfer functions have derivative functions. Newtons method is an alternative to the conjugate gradient methods for fast optimization. The basic step of Newtons method is – $$x_{k+1} = x_k - A_k^{-1}g_k$$ where $A^{-1}{}_k$ is the Hessian matrix (second derivatives) of the performance index at the current values of the weights and biases. Newtons method often converges faster than conjugate gradient methods. This algorithm requires more computation in each iteration and more storage than the conjugate gradient methods, although it generally converges in fewer iterations.

2. Keras Optimizers

1. SGD

SGD, or stochastic gradient descent, is the "classical" optimization algorithm. In SGD we compute the gradient of the network loss function with respect to each individual weight in the network.

Given a small enough learning rate, SGD always simply follows the gradient on the cost surface. The new weights generated on each iteration will always be strictly better than old ones from the previous iteration.

SGD's simplicity makes it a good choice for shallow networks. However, it also means that SGD converges significantly more slowly than other algorithms. It is also less capable of escaping locally optimal traps in the cost surface. Hence SGD is not used or recommended for use on deep networks.

2. Adadelta

Adadelta uses momentum techniques to deal with the monotonically decreasing learning rate problem. Adadelta has each gradient update on each weight be a weighted sum of the current gradient and an exponentially decaying average of a limited number (a rolling window) of past gradient updates.

3. RMSprop

RMSprop is equivalent to Adadelta, with one difference: the learning rate is further divided by an exponentially decaying average of all squared gradients, e.g. a global tuning value.

As with the other adaptive learning rate optimization algorithms the recommendation is to leave the algorithm hyper parameters at their default settings.

4. Adam

Adam stands for Adaptive Moment Estimation. In addition to storing an exponentially decaying average of past squared gradients like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients, similar to momentum.

Adam is basically RMSprop plus momentum. The pathfinding of the Adam algorithm is essentially that of a ball with both momentum and friction. Adam biases the path the algorithm takes towards flat minima in the error surface, slowing down learning when moving along a large gradient.

Adam provides both the smart learning rate annealing and momentum behaviours of the different algorithms

Comparison of Neural Network Implementation of a 10 input AND Gate using MATLAB and Keras API

1. AND Gates

Logic gates are the basic building blocks of any digital system. It is an electronic circuit having one or more than one input and only one output. The relationship between the input and the output is based on a certain logic.
A high output results only if all the inputs to the AND gate are high. If none or not all inputs to the AND gate are high, low output results. Basic truth table –

A	*B*	*Output*
1	1	1
1	0	0
0	1	0
0	0	0

Because the Boolean expression for the logic AND function is defined as (.), which is a binary operation, AND gates can be cascaded together to form any number of individual inputs.

2. Dataset

Both models on MATLAB and Python use the same dataset for a 10 input AND gate truth table.

The dataset is generated on Python using the *ttg* package, and the training, validation and testing splits are made the exact same for both models. Dataset dimensions - 1024

3. Model Parameters and Training Configuration

The Sequential Model from the Keras API of Pythons Tensorflow library is used. For MATLAB, the Neural Net Pattern Recognition Application is used.

Adam Optimizer in Keras and *learnscg* in MATLAB work on the same principle of Stochastic Gradient Descent with Adaptive Learning.

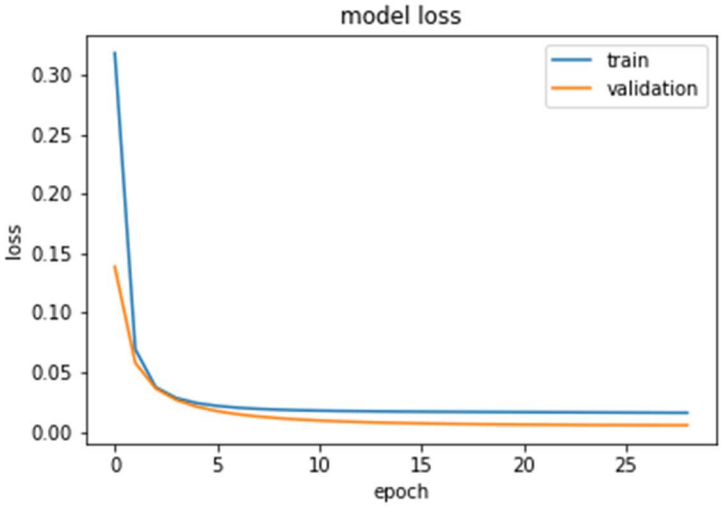
No shuffling of dataset ensures same weight initialization for both models.

The following settings were kept the same –

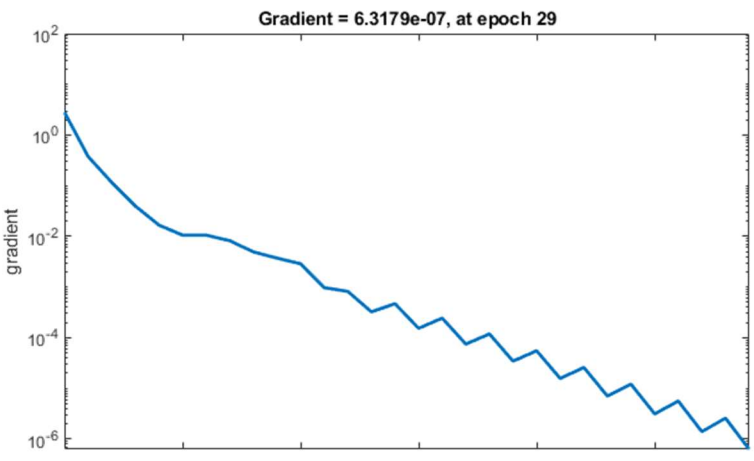
Parameter	*Keras*	*MATLAB*
Train Split	70%	70%
Validation Split	15%	15%
Test Split	15%	15%
Network Type	Feed-Forward Backpropagation	Feed-Forward Backpropagation
Number of hidden layers	1	1
1st Activation Function	*tanh*	*tanh*
2nd Activation Function	*sigmoid*	*sigmoid*
Optimizer	*Adam*	*learnscg*
Loss Function	*binary_crossentropy*	*crossentropy*
System Architecture	x64 Windows	x64 Windows
Shuffle	False	False

4. Computations and Results

Keras



MATLAB



Confusion Matrix

The shown figures are the visualisation of the gradient descent of both algorithms. Adam Optimizer works on Stochastic Gradient Descent whereas the MATLAB model is trained on Scaled Conjugate Gradient which performs faster. Also shown, is the confusion matrix showing 100% accuracy.

5. Conclusion

Both MATLAB and Keras are able to classify the AND Gate output into the linearly separable classes with perfect accuracy (100% in MATLAB and 99.51% in Keras). The gradient descent is different for both models due to minor differences in training methods, but the outcome is correct for both.

Chapter 3 – Epoch Sequencing

Identifying equations of separation from the neural network for AND and XOR logic gates

1. AND Gate

The summation of that our perceptron uses to determine its output is the **dot product** of the inputs and weights vectors, plus the bias.

$$w.x + b$$

When our inputs and weights vectors of are of 2-dimensions, the long form of our dot product summation looks like this:

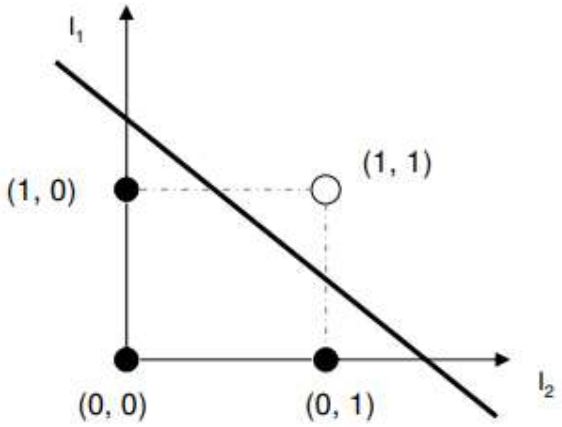
$$w_1x_1 + w_2x_2 + b$$

Since we consider x1 to be the x and x2 to be the y, we can rewrite it:

$$w_1x + w_2y + b = 0$$

This is the equation of the line.

AND		
I_1	I_2	out
0	0	0
0	1	0
1	0	0
1	1	1

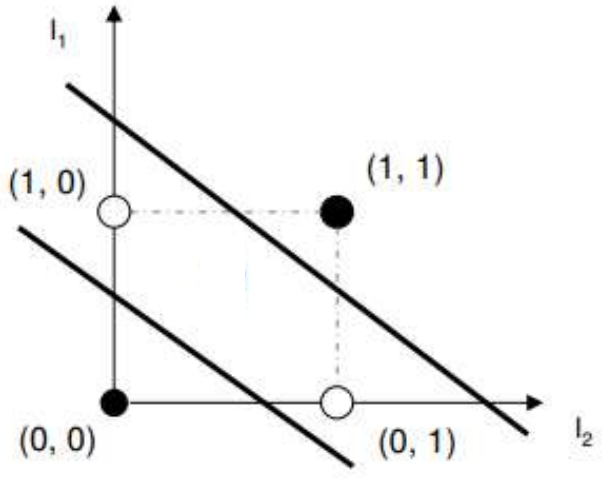


2. XOR Gate

In case of the XOR gate, since the network being used is a Multilayer Perceptron, weights between the two layers form the equations of lines.

$$w_{11}x_{11} + w_{12}x_{12} + b_1 = 0$$
$$w_{21}x_{21} + w_{22}x_{22} + b_2 = 0$$

XOR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	0



Epoch Sequencing for AND, OR, NAND gates to identify significant inputs

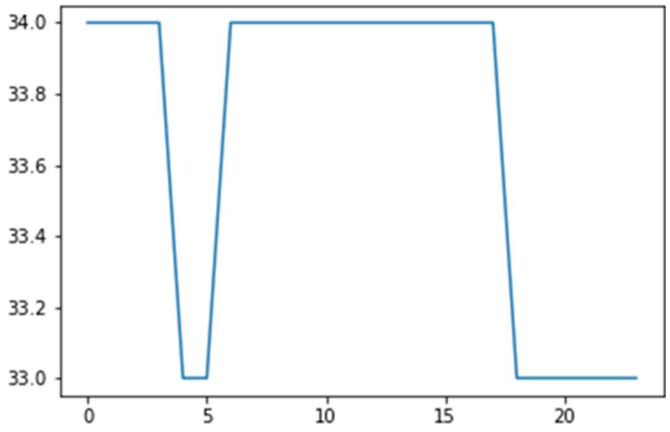
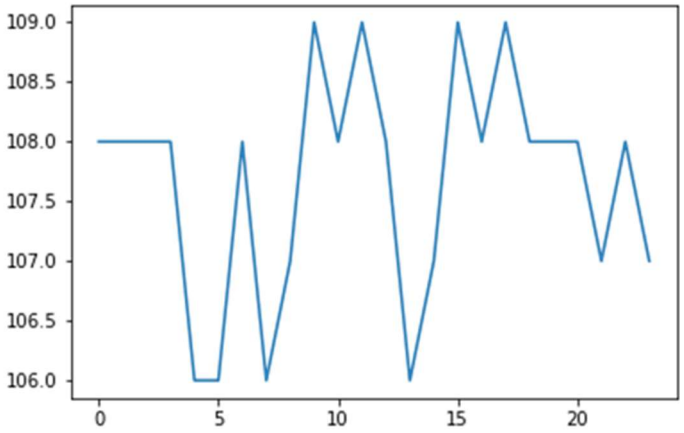
1. Weight Initialization

Weights were randomly assigned previously at the start of the algorithm. However, to maintain uniformity across all sequences of epochs, this randomness was removed. Also, the train test split was fixed to prevent shuffling of the data.

2. Epoch Sequencing

All possible permutations were stored in an array and were used with the model to observe number of epochs it takes to complete training. Total possible sequences possible are 24 for two input gates.

```
output = []
for i in tqdm(range(len(hi)), desc="Loading...", ascii=False, ncols=75):
    X = np.array(hi[i][:, 0:2], dtype='int32')
    Y = np.array(hi[i][:, 2], dtype='int32')
    perceptron = Perceptron()
    no_of_epochs = perceptron.fit(X, Y, 100, 0.01)
    output.append(no_of_epochs)
```

2 input AND	*2 input NAND*
	
Most of the permutations were trained in 34 epochs, but about 35% of the sequences trained faster due to change in position of the significant input.	Some of the permutations were trained in 109 epochs, but others trained faster, in 108 or even 106 epochs due to change in position of the significant input.

3. Conclusion

Epoch Sequencing concluded that the position of the significant input in the training iterations has an impact on the speed of training, which can be greater than observed for larger datasets.

Chapter 4 – Need for a hidden layer – The MLP

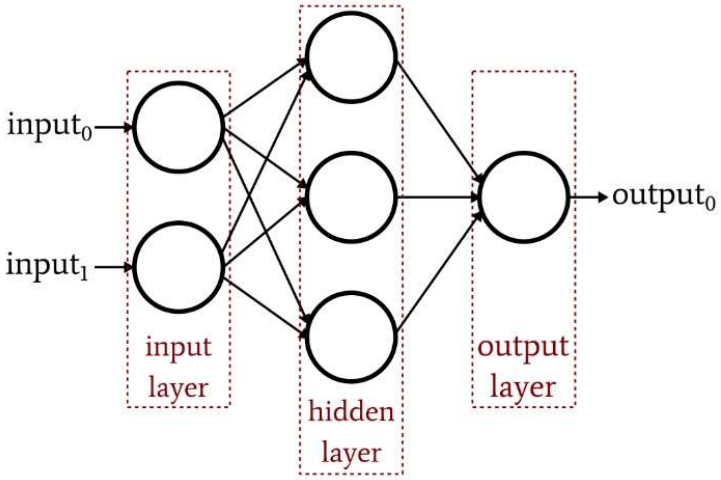
Identifying the XOR Problem and solving using a Multi-Layer Perceptron

1. XOR Gate

An XOR (exclusive OR gate) is a digital logic gate that gives a true output only when both its inputs differ from each other. A limitation of the perceptron architecture is that it is only capable of separating data points with a single line. This is unfortunate because the XOR inputs are not linearly separable.

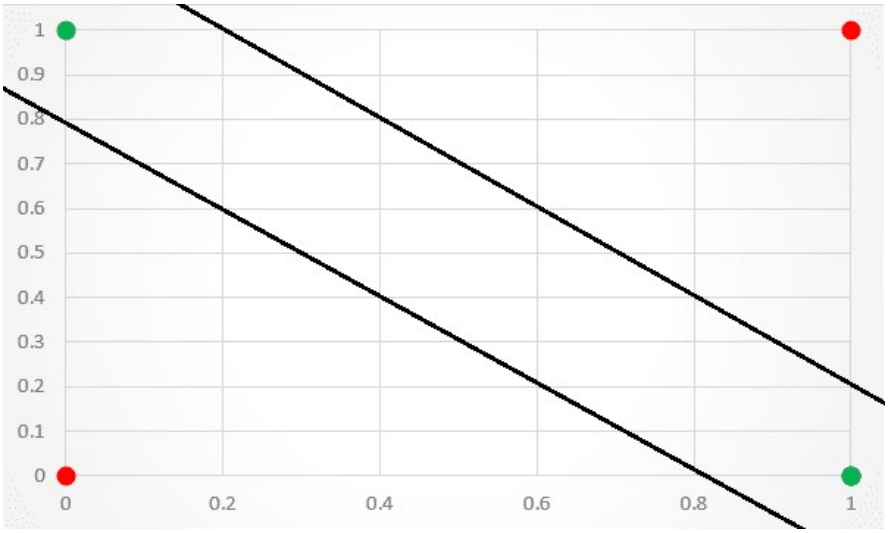
2. Multilayer Perceptron

The solution to this problem is to expand beyond the single-layer architecture by adding an additional layer of units without any direct access to the outside world, known as a hidden layer. This kind of architecture is another feed-forward network known as a multilayer perceptron (MLP).



Similar to the classic perceptron, forward propagation begins with the input values and bias unit from the input layer being multiplied by their respective weights, however, in this case there is a weight for each combination of input (including the input layer bias unit) and hidden unit (excluding the hidden layer bias unit). The products of the input layer values and their respective weights are parsed as input to the non-bias units in the hidden layer.

This architecture, while more complex than that of the classic perceptron network, is capable of achieving non-linear separation. Thus, with the right set of weight values, it can provide the necessary separation to accurately classify the XOR inputs.



Exploring alternative solutions like curves to the XOR problem, and linear separability in multi-dimensional space

1. The need for a Multi-layer Perceptron

If both points are on same side of line —

$$(px_1 + qy_1 + r)(px_2 + qy_2 + r) > 0$$

If opposite —

$$(px_1 + qy_1 + r)(px_2 + qy_2 + r) < 0$$

$(0,0)(1,1)$ $\quad c \cdot (a+b+c) > 0$

$(1,0)(1,1)$ $\quad (a+c)(a+b+c) < 0$

$(0,1)(1,1)$ $\quad (b+c)(a+b+c) < 0$

Adding,

$$(a+b+2c)(a+b+c) < 0$$
$$(a+b+c)^2 + c(a+b+c) < 0$$

$$\Rightarrow c(a+b+c) < 0$$

but since $c(a+b+c) > 0$ already,

$\Rightarrow$ XOR is not linearly separable

Equation of curve $\Rightarrow ax^2 + 2hxy + by^2 + 2fy + 2gx + c = 0$

This satisfies for the points of a XOR Gate.

2. Increasing dimensions

Exploring equation of curve → (higher order than line)

To find a relation between h^2 and ab to determine a specific type of curve.

$$ax^2 + 2hxy + by^2 + 2fy + 2gx + c = 0$$

$(0,0)(1,1)$ $\quad c(a+2h+b+2f+2g+c) > 0$

$(1,0)(1,1)$ $\quad (a+2g+c)(a+2h+b+2f+2g+c) < 0$

$(0,1)(1,1)$ $\quad (b+2f+c)(a+2h+b+2f+2g+c) < 0$

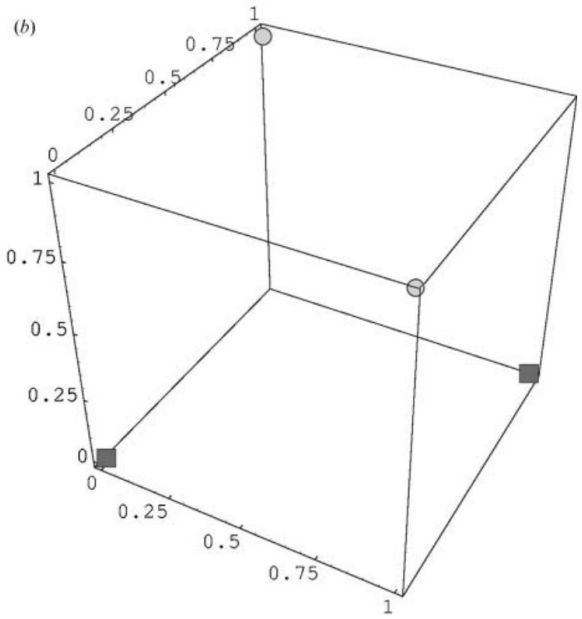
$(1,0)(0,1)$ $\quad (a+2g+c)(b+2f+c) > 0$

$(0,0)(0,1)$ $\quad c(b+2f+c) < 0$

$(0,0)(1,0)$ $\quad c(a+2g+c) < 0$

None of these contradict each other, so the curve satisfies the points of a XOR Gate.

Radial basis functions (RBFs) move the problem to a higher dimension by calculating distances between the centres and the points for classification. Plotting these distances makes the problem linearly separable in 3 dimensions.



Chapter 5 – Alternatives to the MLP

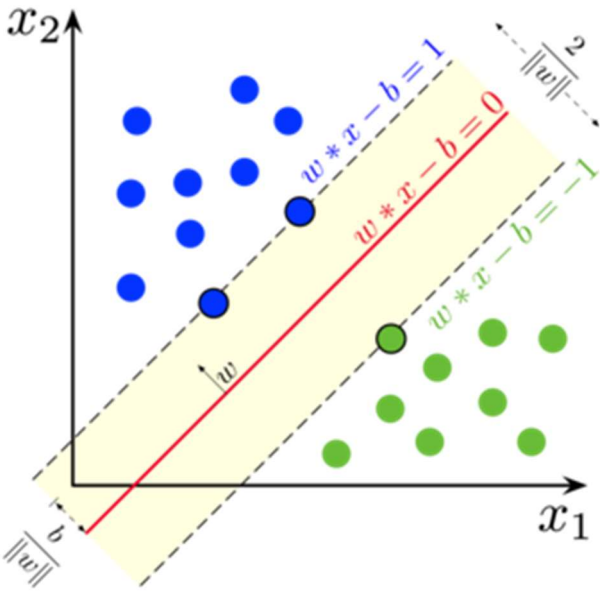
Exploring Support Vector Machines and Kernel Functions

1. Support Vector Machines

SVMs are based on the idea of finding a hyperplane that best divides a dataset into two classes.

Support vectors are the data points nearest to the hyperplane, the points of a data set that, if removed, would alter the position of the dividing hyperplane. Because of this, they can be considered the critical elements of a data set.

The distance between the hyperplane and the nearest data point from either set is known as the margin. The goal is to choose a hyperplane with the greatest possible margin between the hyperplane and any point within the training set, giving a greater chance of new data being classified correctly.

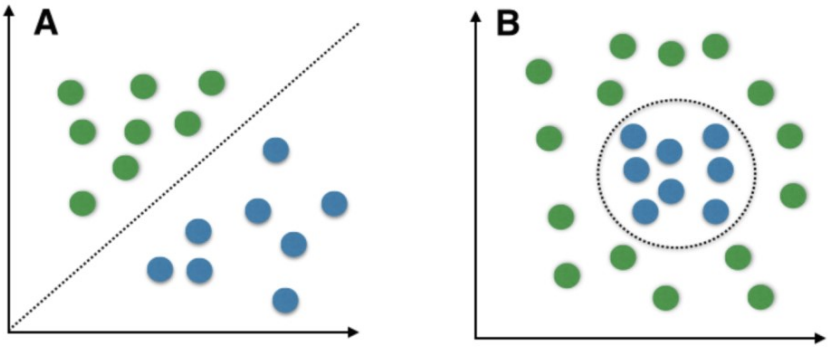


Pros

1. Accuracy
2. Works well on smaller cleaner datasets
3. It can be more efficient because it uses a subset of training points

Cons

- Isn’t suited to larger datasets as the training time with SVMs can be high
- Less effective on noisier datasets with overlapping classes



A hyperplane as shown in second figure can be achieved by moving the data points to a higher dimension where they can be separated by a plane. This is achieved by kernelling.

2. Kernel Methods

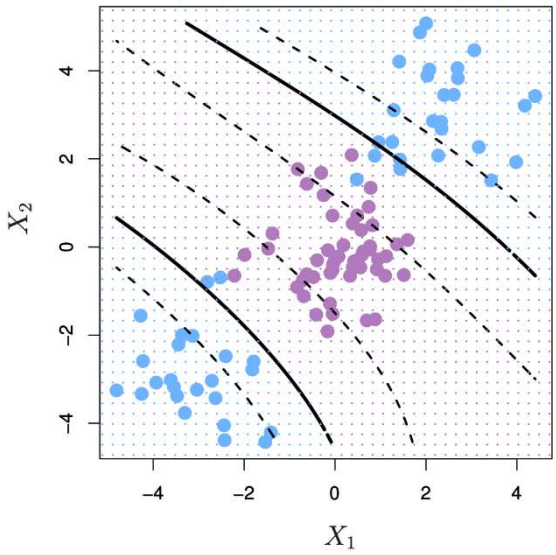
The linear, polynomial and RBF or Gaussian kernel are simply different in case of making the hyperplane decision boundary between the classes.

The kernel functions are used to map the original dataset (linear/nonlinear) into a higher dimensional space with view to making it linear dataset.

Usually linear and polynomial kernels are less time consuming and provides less accuracy than the RBF or Gaussian kernels.

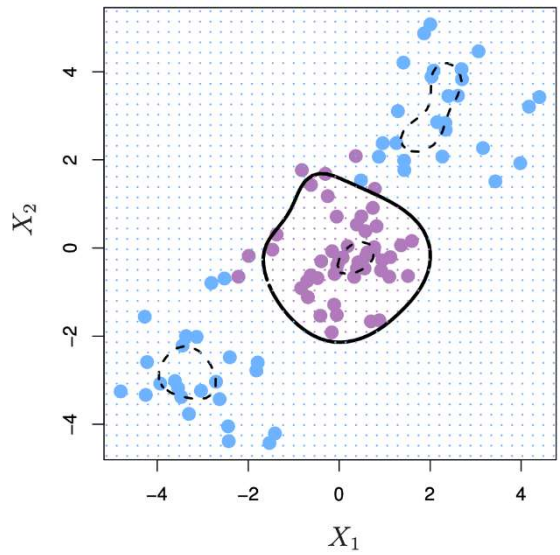
Polynomial Kernel - Support vector machine with a polynomial kernel can generate a non-linear decision boundary using those polynomial features. It is a transformer/processor to generate new features by applying the polynomial combination of all the existing features.

$$k(\mathbf{x_i}, \mathbf{x_j}) = (\mathbf{x_i} \cdot \mathbf{x_j} + 1)^d$$



Radial Basis Function - RBF is a transformer/processor to generate new features by measuring the distance between all other dots to a specific dot/dots — centres.

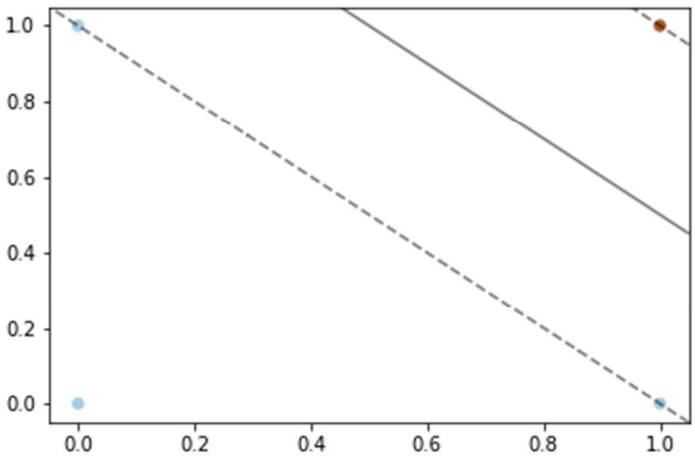
$$k(\mathbf{x_i}, \mathbf{x_j}) = \exp(-\gamma \|\mathbf{x_i} - \mathbf{x_j}\|^2)$$

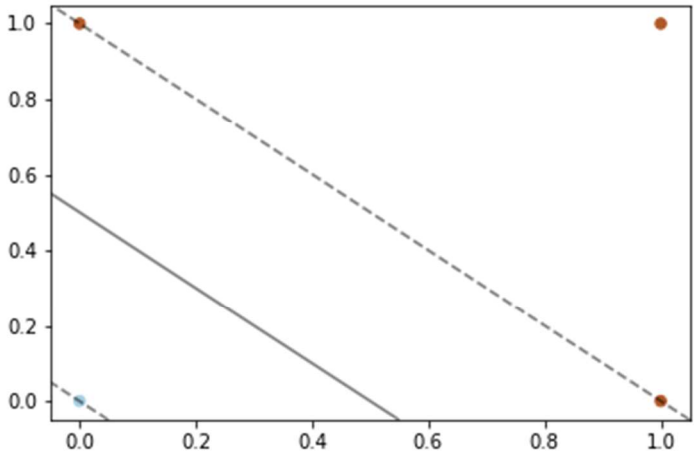


Implementing SVMs and Kernel Functions on Logic Gates, Exploring Adaptive Resonance Theory and Version Spaces

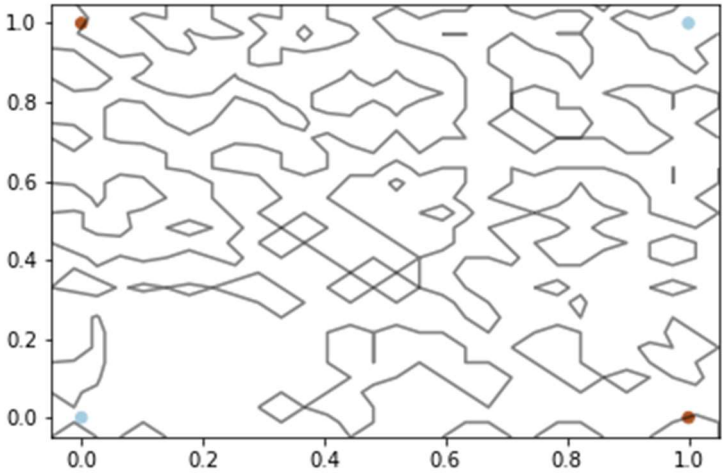
1. Linear Kernel

Linear hyperplanes are useful for linearly separable classes as demonstrated in the AND and OR gates.





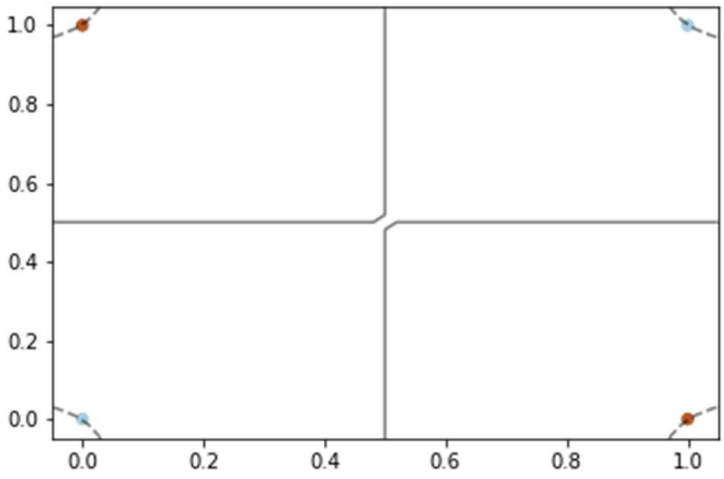
However, when used on a XOR gate, the algorithm malfunctions as it is non-separable linearly.



2. RBF Kernel

Radial Basis Function considers the distances of other points taking one as the centre. This creates the hyperplanes as shown -

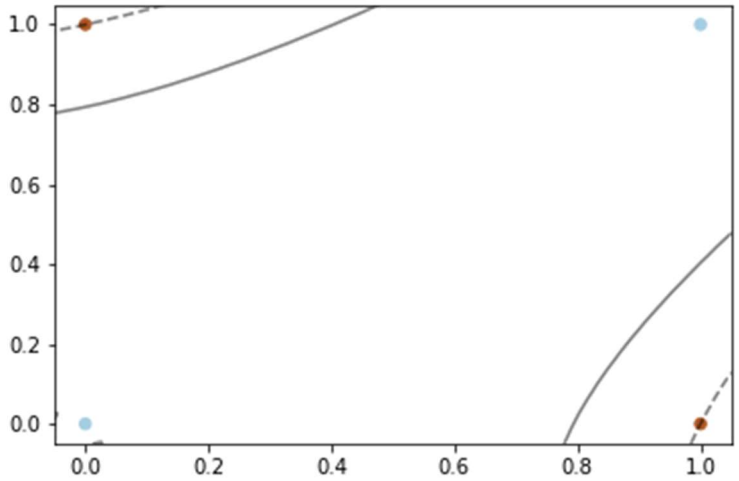
$$k(\mathbf{x_i}, \mathbf{x_j}) = \exp(-\gamma \|\mathbf{x_i} - \mathbf{x_j}\|^2)$$



3. Polynomial Kernel

Polynomial kernel maps curves instead of straight lines, with the data points as the support vectors. This makes the problem easily separable.

$$k(\mathbf{x_i}, \mathbf{x_j}) = (\mathbf{x_i} \cdot \mathbf{x_j} + 1)^d$$



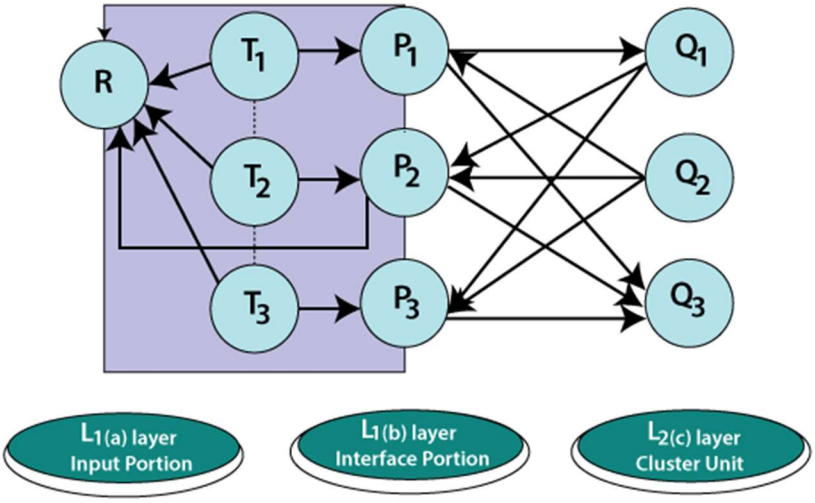
In all of the plots, dotted lines mark the margin planes passing through the support vectors, and the solid lines are the separating planes.

4. Adaptive Resonance Theory

The adaptive resonant theory is a type of neural network that is self-organizing and competitive.

The basic ART model is unsupervised in nature and consists of:

- F1 layer or the comparison field (where the inputs are processed)
- F2 layer or the recognition field (which consists of the clustering units)
- The Reset Module (that acts as a control mechanism)



The **F1 layer** accepts the inputs and performs some processing and transfers it to the F2 layer that best matches with the classification factor.

There exist **two sets of weighted interconnections** for controlling the degree of similarity between the units in the F1 and the F2 layer.

The **F2 layer** is a competitive layer. The cluster unit with the large net input becomes the candidate to learn the input pattern first and the rest F2 units are ignored.

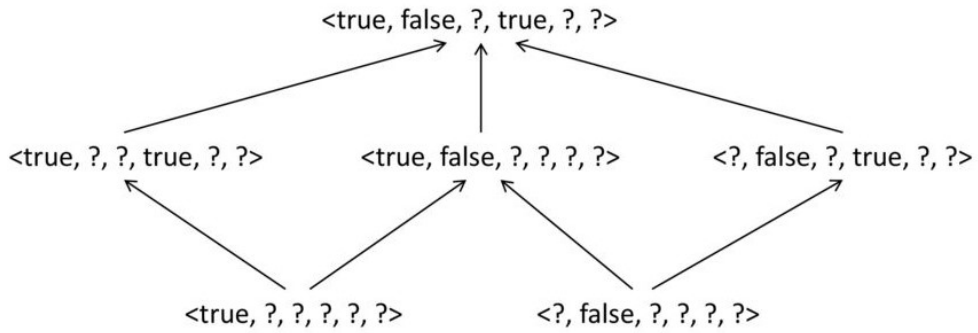
The **reset unit** makes the decision whether or not the cluster unit is allowed to learn the input pattern depending on how similar its top-down weight vector is to the input vector and to the decision. This is called the vigilance test.

Thus the **vigilance parameter** helps to incorporate new memories or new information. Higher vigilance produces more detailed memories, lower vigilance produces more general memories.

- It exhibits stability and is not disturbed by a wide variety of inputs provided to its network.
- It can be integrated and used with various other techniques to give more good results.
- It can be used for various fields such as mobile robot control, face recognition, land cover classification, target recognition, medical diagnosis, signature verification, clustering web users, etc.
- It has got advantages over competitive learning. The competitive learning lacks the capability to add new clusters when deemed necessary.
- It does not guarantee stability in forming clusters.

5. Version Spaces

A *version space* for a learning algorithm can be defined as the subset of hypotheses consistent with the training examples. That is, the hypothesis language is capable of describing a large, possibly infinite, number of concepts. When searching for the target concept, we are only interested in the subset of sentences in the hypothesis language that are consistent with the training examples, where consistent means that the examples are correctly classified (assuming deterministic concepts and no noise in the data).



Given a set of training examples, any concept consistent with them must −

- include every positive instance
- exclude every negative instance

Chapter 6 – Experimenting with a Spam Classification Dataset

Implementing SVMs and Kernel Functions for Spam Mail Classification

1. Dataset

The dataset extracted from Kaggle, consists of words that make up mails. Presence of these words in mails makes up the input parameter space. The labels depict if the mail is spam or not.

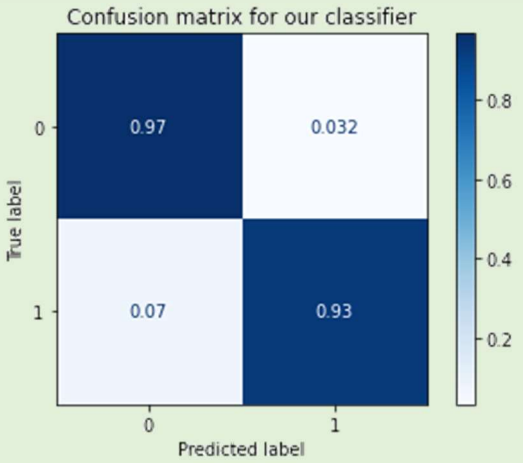
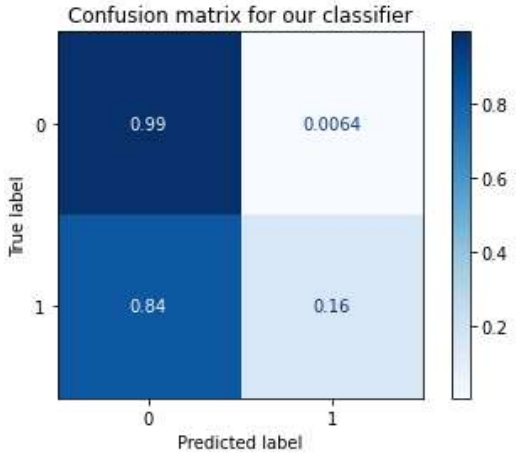
No. of samples = 5172

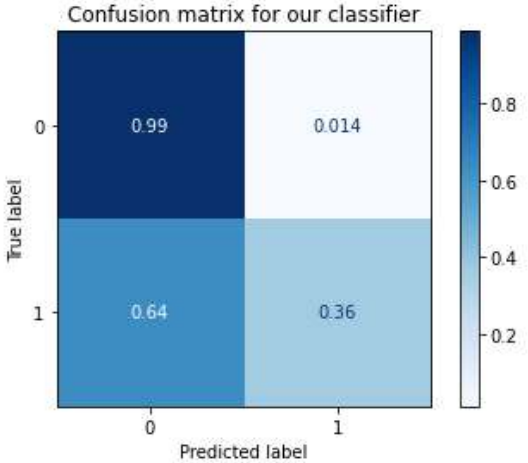
Input Parameters = 3000

The train-test split is 70-30 for a generalized model. The dataset was clean and ready to use.

2. Results from different kernels

The linear kernel gave the best accuracy for the classification, implying that the datapoints were linearly separable.

Linear Kernel	*Polynomial Kernel*
	
Precision Score – 0.923 Recall Score – 0.929 F1 Score – 0.926	Precision Score – 0.912 Recall Score – 0.160 F1 Score – 0.27

RBF Kernel

Precision Score – 0.916 Recall Score – 0.362 F1 Score – 0.519

Conclusion

During this project, it was concluded that a perceptron, a building block of data science and learning algorithms, can only be used to classify linearly separable classes. Epoch Sequencing shows us the importance of the placement of the significant input in training. Adding another layer of computing neurons to a perceptron model makes them capable of exploring relations beyond linearity. A multi-layer perceptron creates a hyperplane to separate two classes. Alternatives to MLPs do exist, in the form of Support Vector Machines and Kernel Functions. One removes bias and explores non-linear functions to separate the classes, while the other maps the data points to a higher dimension, to make them separable. SVMs with appropriate kernel methods generate good classification results, as portrayed in the case of Spam Mail Classification.