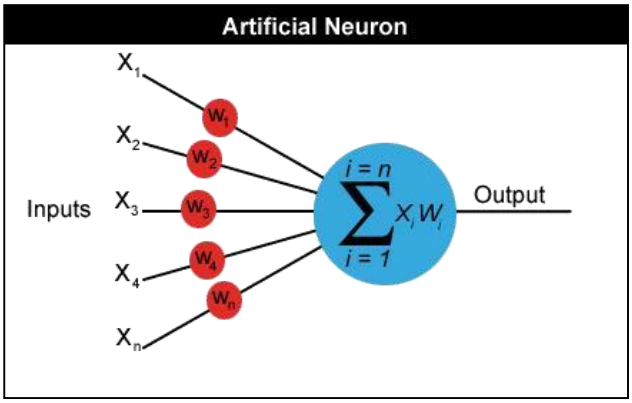
Unit II LINEAR MODELS

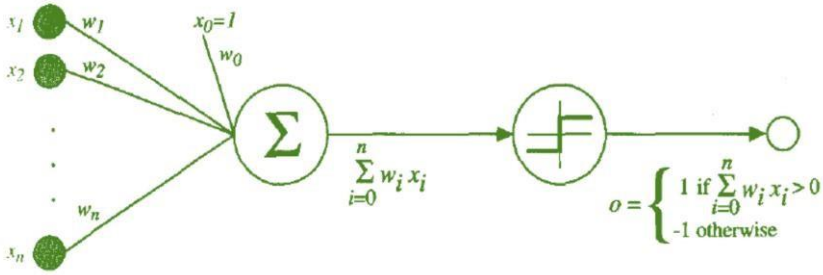
2.1 Multi-layer Perceptron

- Neural networks
  - Neural networks are made up of many artificial neurons.
  - Each input into the neuron has its own weight associated with it illustrated by the red circle.
  - A weight is simply a floating point number and it's these we adjust when we eventually come to train the network.
- A neuron can have any number of inputs from one to n, where n is the total number of inputs.
- The inputs may be represented therefore as x1, x2, x3… xn.
- And the corresponding weights for the inputs as w1, w2, w3… w_n.
- Output $a = x_1w_1+x_2w_2+x_3w_3... +x_nw_n$.



- Feedforward network: The neurons in each layer feed their output forward to the next layer until we get the final output from the neural network.
- There can be any number of hidden layers within a feedforward network.
- The number of neurons can be completely arbitrary.

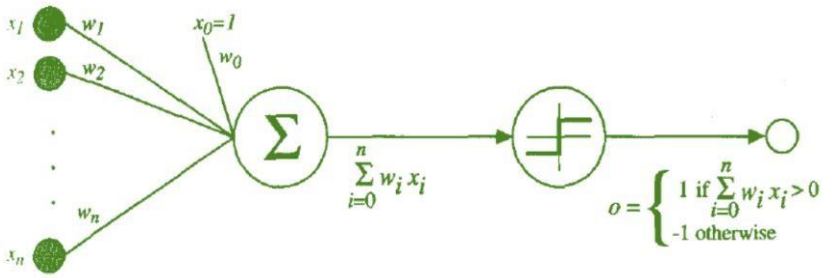
Perceptron



$$o(x_1, \ldots, x_n) = \begin{cases} 1 \text{ if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 \text{ otherwise.} \end{cases}$$

Sometimes we'll use simpler vector notation:

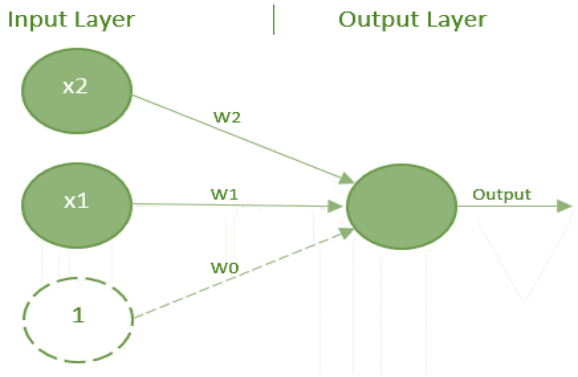
$$o(\vec{x}) = \begin{cases} 1 \text{ if } \vec{w} \cdot \vec{x} > 0 \\ -1 \text{ otherwise.} \end{cases}$$



Perceptron

- The perceptron is a type of feed-forward network, which means the process of generating an output
  - — known as forward propagation
  - — flows in one direction from the input layer to the output layer.
- There are no connections between units in the input layer. Instead, all units in the input layer are connected directly to the output unit.
- Input values X1 and X2, along with the bias value of 1, are multiplied by their respective weights W0..W2, and parsed to the output unit.
- The output unit takes the sum of those values and employs an activation function — typically the step function — to convert the resulting value to a 0 or 1, thus classifying the input values as 0 or 1.

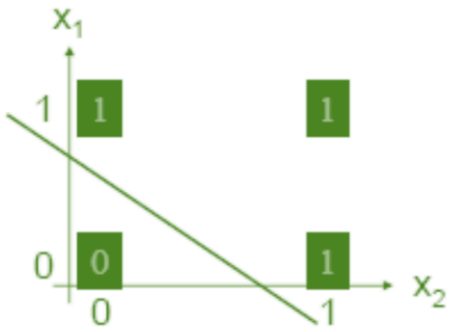
- A limitation of this architecture is that it is only capable of separating data points with a single line.



- A perceptron (threshold unit) can learn anything that it can represent (i.e. anything separable with a hyperplane)

OR function

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1



2.1 Multi-layer Perceptron

- Linear models are easy to understand and use, they come with the inherent cost that is implied by the word ‘linear’; that is, they can only identify straight lines, planes, or hyperplanes.
- And this is not usually enough, because the majority of interesting problems are not linearly separable.
- Learning in the neural network happens in the weights. So, to perform more computation it seems sensible to add more weights.
- There are two things that we can do:
  - Add some backwards connections, so that the output neurons connect to the inputs again, or add more neurons. This approach leads into recurrent networks. But are not that commonly used.

- In the second approach, We can add neurons between the input nodes and the outputs, and this will make more complex neural networks, such as the one shown in Figure 2.1.1

- We can check that a prepared network can solve the two-dimensional XOR problem, something that we have seen is not possible for a linear model like the Perceptron.
- A suitable network is shown in Figure 2.1.2 To check that it gives the correct answers, all that is required is to put in each input and work through the network, treating it as two different Perceptrons.

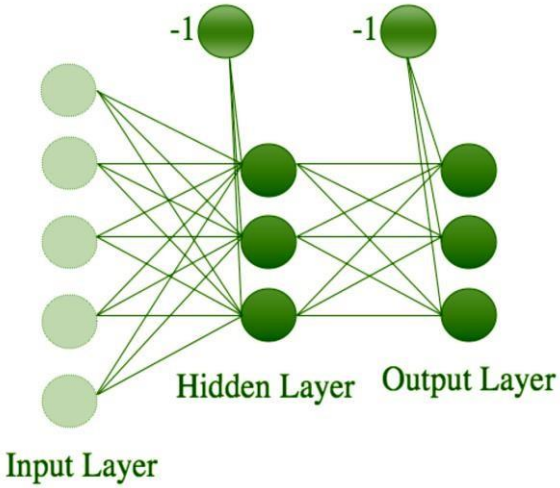


Fig. 2.1.1 The Multi-layer Perceptron network, consisting of multiple layers of connected neurons.

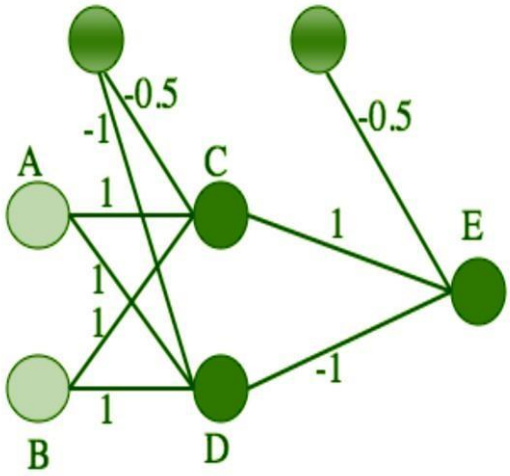


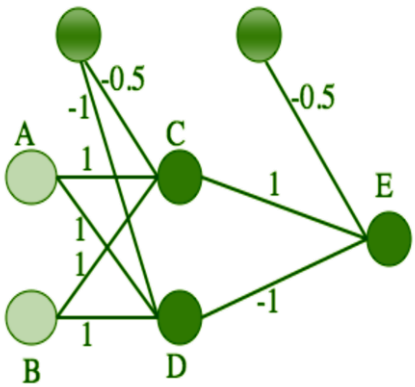
Fig. 2.1.2 A Multi-layer Perceptron network showing a set of weights that solve the XOR problem.

- First computing the activations of the neurons in the middle layer (labelled as C and D) and then using those activations as the inputs to the single neuron at the output.
- As an example, put (1, 0) as an input; Input (1, 0) corresponds to node A being 1 and B being 0. The input to neuron C is therefore

$$-1 \times 0.5 + 1 \times 1 + 0 \times 1 = -0.5 + 1 = 0.5.$$

- This is above the threshold of 0, and so neuron C fires, giving output 1.

- For neuron D the input is $-1 \times 1 + 1 \times 1 + 0 \times 1 = -1 + 1 = 0$, and so it does not fire, giving output 0.
- Therefore the input to neuron E is $-1 \times 0.5 + 1 \times 1 + 0 \times -1 = 0.5$, so neuron E fires.
- Checking the result of the inputs, neuron E fires when inputs A and B are different to each other, but does not fire when they are the same, which is exactly the XOR function.
- The MLP is one of the most common neural networks in use. It is often treated as a 'black box', in that people use it without understanding how it works, which often results in fairly poor results.



Let us consider

A= 1;

B= 0,

Default bias = -1

w11 =w12=w13=w14 = 1 w21 = 1, w22 = -1

C = Bias * learning rate + A * w11 + B* w12

= -1 * 0.5 + 1 * 1 + 0*1

= -0.5+1+0 = 0.5 > 0 Neuron fires the output as 1 D = Bias * learning rate + A * w13 + B* w14

= -1 * 1 + 1 * 1 + 0*1

= -1+1+0 = 0 <= 0 Neuron fires the output as 0 E

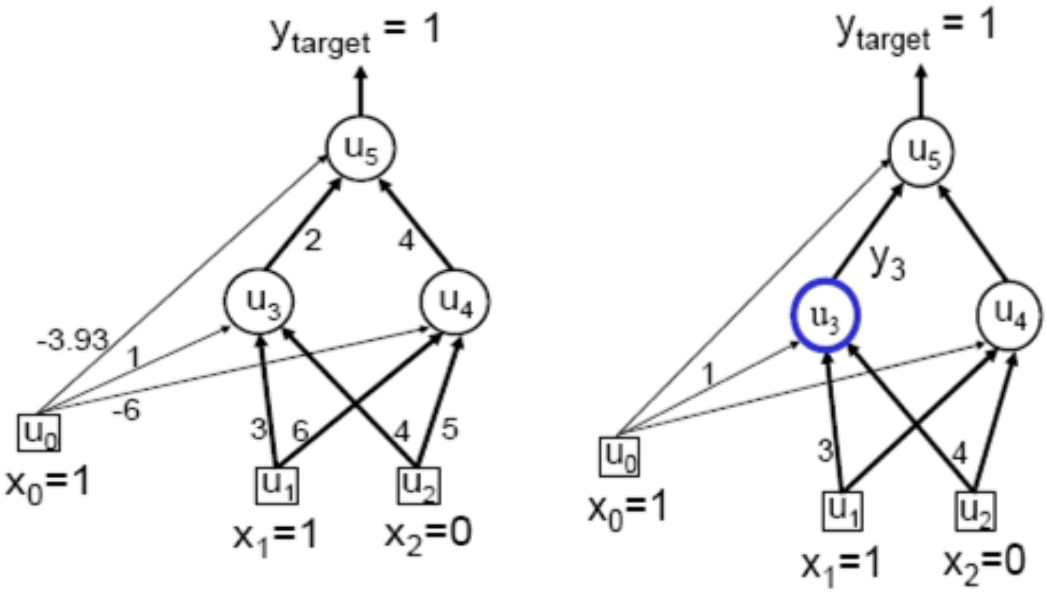
= Bias * learning rate + A * w21 + B* w22

= $-1 \times 0.5 + 1 \times 1 + 0 \times -1$

= -0.5 + 1+0 = 0.5 > 0 Neuron fires the output as 1

2.2 Going Forwards

- Training the MLP consists of two parts: working out what the outputs are for the given inputs and the current weights, and then updating the weights according to the error, which is a function of the difference between the outputs and the targets.
- These are generally known as going forwards and backwards through the network.
- It is same as the Perceptron, except that we have to do it twice, once for each set of neurons, and we need to do it layer by layer, because otherwise the input values to the second layer don't exist.
- This won't even change our recall (forward) algorithm much, since we just work forwards through the network computing the activations of one layer of neurons and using those as the inputs for the next layer.
- So looking at Figure 2.1.1, we start at the left by filling in the values for the inputs. We then use these inputs and the first level of weights to calculate the activations of the hidden layer, and then we use those activations and the next set of weights to calculate the activations of the output layer.
- Now that we've got the outputs of the network, we can compare them to the targets and compute the error. Example



2.2 Going Forwards

Example

Output for any neuron/unit j can be calculated from:

$$a_j = \sum_i w_{ij} x_i$$

$$y_j = f(a_j) = \frac{1}{1 + e^{-a_j}}$$

e.g Calculating output for Neuron/unit 3 in hidden layer:

$$a_3 = 1*1 + 3*1 + 4*0 = 4$$

$$y_3 = f(4) = \frac{1}{1+e^{-4}} = 0.982$$

$X_1 = 1$; $X_2 = 0$

- **X1 OR X2 = YTARGET**

  => 1 OR 0 = 1

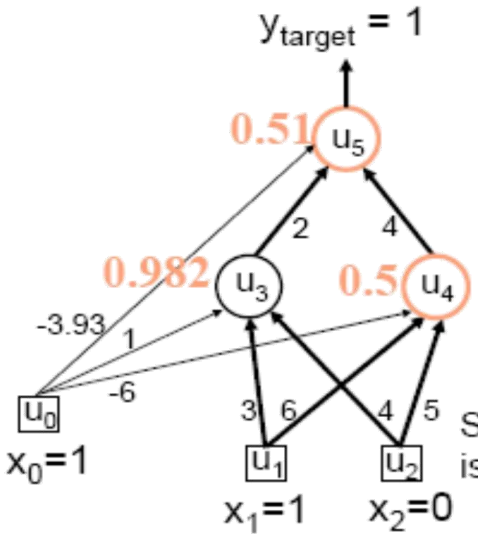
- Here $X_0 = 1$ (X_0 is an extra input with fixed value)
- Bias for unit 3,4,5 = 1

$a_3 = X_0 * \text{Bias} + X_1 * W_{13} + X_2 * W_{23}$

$= 1 * 1 + 1* 3 + 0 * 4$

$= 1+3+0 = 4$

y3 = f(4) = 1T1+e−4 → **0.98**



Unit	activation a_j	output y_j
u_3	4.00	0.982
u_4	0.00	0.500
u_5	0.04	**0.510** (network output)

So the error for this training example is: $(y_{target} - y_5) = (1 - 0.510) = 0.490$

Error = target output – predicted output

= YTarget - YPredicted

= 1 – 0.510 = 0.490

2.2.1 Biases

- We need to include a bias input to each neuron. As we did for the Perceptron by having an extra input that is permanently set to -1, and adjusting the weights to each neuron as part of the training.
- Thus, each neuron in the network (whether it is a hidden layer or the output) has 1 extra input, with fixed value.

2.3 Going Backwards: Back Propagation Error

- Computing the errors at the output is no more difficult than it was for the Perceptron, but working out what to do with those errors is more difficult.
- The method that we are going to look at is called back-propagation of error, which makes it clear that the errors are sent backwards through the network. It is a form of gradient descent.
- In Perceptron, we changed the weights so that the neurons fired when the targets said they should, and didn't fire when the targets said they shouldn't.

- An error function for each neuron k: $E_k = y_k - t_k$, and tried to make it as small as possible.
- Since there was only one set of weights in the network, this was sufficient to train the network.
- We still want to do the same thing—minimise the error, so that neurons fire only when they should—but, with the addition of extra layers of weights, this is harder to arrange. The problem is that when we try to adapt the weights of the Multi-layer Perceptron, we have to work out which weights caused the error.
- This could be the weights connecting the inputs to the hidden layer, or the weights connecting the hidden layer to the output layer.
- The error function that we used for the Perceptron was $\sum_{k=1}^{N} E_k = \sum_{k=1}^{N} y_k - t_k$
- Where N is the number of output nodes. However, suppose that we make two errors.
- In the first, the target is bigger than the output, while in the second the output is bigger than the target.
- If these two errors are the same size, then if we add them up we could get 0, which
- means that the error value suggests that no error was made.
- To get around this we need to make all errors have the same sign. We used sum-ofsquares error function, which calculates the difference between y and t for each node, squares them, and adds them all together:

$$E(\mathbf{t}, \mathbf{y}) = \frac{1}{2} \sum_{k=1}^{N} (y_k - t_k)^2.$$

- There are only three things in the network that change: the inputs, the activation function that decides whether or not the node fires, and the weights.
- The first and second are out of our control when the algorithm is running, so only the weights matter, and therefore they are what we differentiate with respect to.

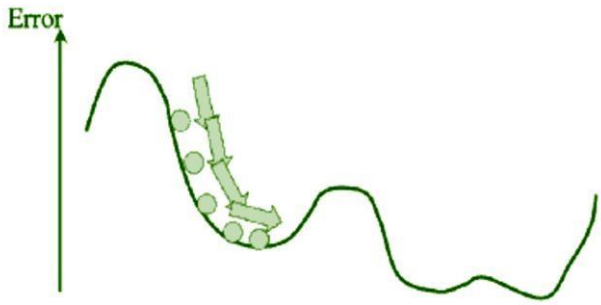


Fig 2.3.1 The weights of the network are trained so that the error goes downhill until it reaches a local minimum, just like a ball rolling under gravity.

- The threshold function that we have been using for our neurons so far, which is that it is discontinuous (see Figure 2.3.2; it has a sudden jump in the middle) and so differentiating it at that point isn't possible.
- The problem is that we need that jump between firing and not firing to make it act like a neuron.
- We can solve the problem if we can find an activation function that looks like a threshold function, but is differentiable so that we can compute the gradient.

- If you squint at a graph of the threshold function (for example, Figure 2.3.2) then it looks kind of S-shaped. There is a mathematical form of S-shaped functions, called sigmoid functions (see Figure 2.3.3).

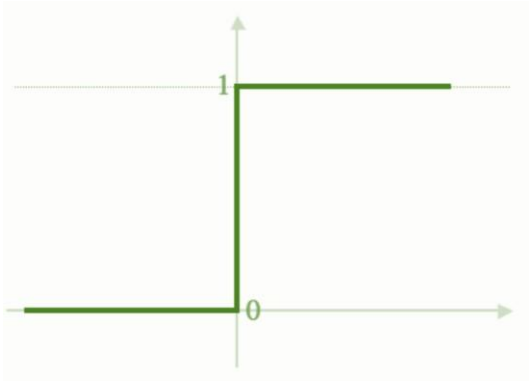


Fig 2.3.2 The threshold function that we used for the Perceptron. Note the discontinuity where the value changes from 0 to 1.

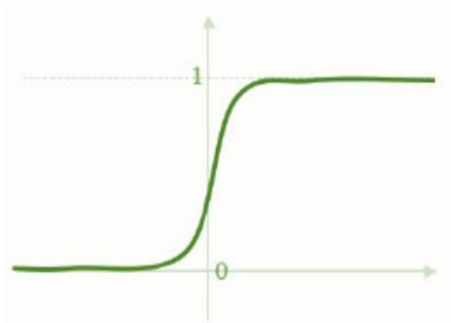


Fig 2.3.3 The sigmoid function, which looks qualitatively fairly similar, but varies smoothly and differentiably.

- The most commonly used form of this function (where is some positive parameter) is:

$$a = g(h) = \frac{1}{1 + \exp(-\beta h)}.$$

- As far as an algorithm goes, we've fed our inputs forward through the network and worked out which nodes are firing. Now, at the output, we've computed the errors as the sum-squared difference between the outputs and the targets.
- Next is to compute the gradient of these errors and use them to decide how much to update each weight in the network. We will do that first for the nodes connected to the output layer, and after we have updated those, we will work backwards through the network until we get back to the inputs again.
- There are just two problems:
  - ✓ for the output neurons, we don't know the inputs.

- ✓ for the hidden neurons, we don't know the targets; for extra hidden layers, we know neither the inputs nor the targets, but even this won't matter for the algorithm we derive.

- Here, the chain rule tells us that if we want to know how the error changes as we vary the weights, we can write the activations of the output nodes in terms of the activations of the hidden nodes and the output weights, and then we can send the error calculations back through the network to the hidden layer to decide what the target outputs were for those neurons.
- Note that we can do exactly the same computations if the network has extra hidden layers between the inputs and the outputs.
- We do this by differentiating the error function with respect to the weights, but we can't do this directly, so we have to apply the chain rule and differentiate with respect to things that we know.
- This leads to two different update functions, one for each of the sets of weights, and we just apply these backwards through the network, starting at the outputs and ending up back at the inputs.

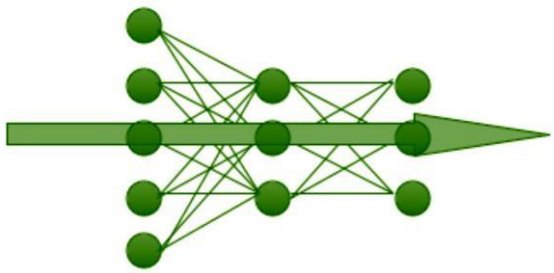


FIGURE 2.3.4 The forward direction in a Multi-layer Perceptron.

2.3.1 The Multi-layer Perceptron Algorithm

MLP training algorithm using back-propagation of error is described.

1. An input vector is put into the input nodes
2. The inputs are fed forward through the network
    - the inputs and the first-layer weights (here labelled as v) are used to decide whether the hidden nodes fire or not. The activation function g(·) is the sigmoid function given in Equation (4.2) above
    - the outputs of these neurons and the second-layer weights (labelled as w) are used to decide if the output neurons fire or not
3. the error is computed as the sum-of-squares difference between the network outputs and the targets
4. this error is fed backwards through the network in order to
    - first update the second-layer weights
    - and then afterwards, the first-layer weights

The Multi-layer Perceptron Algorithm

- Initialisation

– initialise all weights to small (positive and negative) random values

- Training

– repeat:

* for each input vector:

Forwards phase:

· compute the activation of each neuron j in the hidden layer(s) using:

$$h_\kappa = \sum_j a_j w_{j\kappa}$$

$$y_\kappa = g(h_\kappa) = \frac{1}{1+\exp(-\beta h_\kappa)}$$

· work through the network until you get to the output layer neurons, which have activations :

$$h_\zeta = \sum_{i=0}^{L} x_i v_{i\zeta}$$

$$a_\zeta = g(h_\zeta) = \frac{1}{1+\exp(-\beta h_\zeta)}$$

Backwards phase:

· compute the error at the output using:

$$\delta_o(\kappa) = (y_\kappa - t_\kappa)\, y_\kappa (1 - y_\kappa)$$

· compute the error in the hidden layer(s) using:

$$\delta_h(\zeta) = a_\zeta (1 - a_\zeta) \sum_{k=1}^{N} w_\zeta \delta_o(k)$$

· update the output layer weights using:

$$w_{\zeta\kappa} \leftarrow w_{\zeta\kappa} - \eta \delta_o(\kappa) a_\zeta^{\text{hidden}}$$

· update the hidden layer weights using:

$$v_\iota \leftarrow v_\iota - \eta \delta_h(\kappa) x_\iota$$

* (if using sequential updating) randomise the order of the input vectors so that you don't train in exactly the same order each iteration – until learning stops

- Recall

– use the Forwards phase in the training section above

2.3.2 Initialising the Weights

- The MLP algorithm suggests that the weights are initialized to small random numbers, both positive and negative.
- If the initial weight values are close to 1 or -1 (which is what we mean by large here) then the inputs to the sigmoid are also likely to be close to ±1 and so the output of the neuron is either 0 or 1 (the sigmoid has saturated, reached its maximum or minimum value).
- If the weights are very small (close to zero) then the input is still close to 0 and so the output of the neuron is just linear, so we get a linear model.
- If we view the values of these inputs as having uniform variance, then the typical input to the neuron will be $w \sqrt{n}$, where w is the initialization value of the weights. So a common trick is to set the weights in the range $-1/\sqrt{n} < w < 1/\sqrt{n}$, where n is the number of nodes in the input layer to those weights.
- This makes the total input to a neuron have a maximum size of about 1.
- Further, if the weights are large, then the activation of a neuron is likely to be at, or close to, 0 or 1 already, which means that the gradients are small, and so the learning is very slow.

2.3.3 Different Output Activation Functions

- In the algorithm, we used sigmoid neurons in the hidden layer and the output layer.
- This is fine for classification problems, since there we can make the classes be 0 and 1. However, we might also want to perform regression problems, where the output needs to be from a continuous range, not just 0 or 1.
- The sigmoid neurons at the output are not very useful in that case. We can replace the output neurons with linear nodes that just sum the inputs and give that as their activation (so g(h) = h in the notation of Equation). This does not mean that we change the hidden layer neurons; they stay exactly the same, and we only modify the output nodes.

- The soft-max function rescales the outputs by calculating the exponential of the inputs to that neuron, and dividing by the total sum of the inputs to all of the neurons, so that the activations sum to 1 and all lie between 0 and 1. As an activation function it can be written as:

$$y_\kappa = g(h_\kappa) = \frac{\exp(h_\kappa)}{\sum_{k=1}^{N} \exp(h_k)}.$$

- Of course, if we change the activation function, then the derivative of the activation function will also change, and so the learning rule will be different.
- For the linear activation function the first is replaced by:

$$y_\kappa = g(h_\kappa) = h_\kappa,$$

- while the second is replaced by:

$$\delta_o(\kappa) = (y_\kappa - t_\kappa)\,.$$

- For the soft-max activation, the update equation that replaces is

$$\delta_o(\kappa) = (y_\kappa - t_\kappa) y_\kappa (\delta_{\kappa K} - y_K),$$

2.3.4 Sequential and Batch Training

- The MLP is designed to be a batch algorithm. All of the training examples are presented to the neural network, the average sum-of-squares error is then computed, and this is used to update the weights.
- Thus there is only one set of weight updates for each epoch (pass through all the training examples). This means that we only update the weights once for each iteration of the algorithm, which means that the weights are moved in the direction that most of the inputs want them to move, rather than being pulled around by each input individually.
- The batch method performs a more accurate estimate of the error gradient, and will thus converge to the local minimum more quickly.

- The algorithm that was described earlier was the sequential version, where the errors are computed and the weights updated after each input. This is not guaranteed to be as efficient in learning, but it is simpler to program when using loops, and it is therefore much more common.
- Since it does not converge as well, it can also sometimes avoid local minima, thus potentially reaching better solutions.

2.3.5 Local Minima

- The driving force behind the learning rule is the minimization of the network error by gradient descent (using the derivative of the error function to make the error smaller).
- This means that we are performing an optimization: we are adapting the values of the weights in order to minimize the error function.
- This is achieved by approximating the gradient of the error and following it downhill so that we end up at the bottom of the slope. However, following the slope downhill only guarantees that we end up at a local minimum, a point that is lower than those close to it.

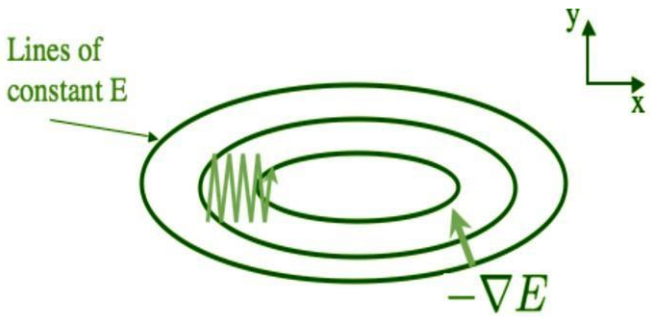


FIGURE 4.7 In 2D, downhill means at right angles to the lines of constant contour. Imagine walking down a hill with your eyes closed. If you find a direction that stays flat, then it is quite likely that perpendicular to that the ground goes uphill or downhill. However, this is not the direction that takes you directly towards the local minimum.

- If we imagine a ball rolling down a hill, it will settle at the bottom of a dip. However, there is no guarantee that it will have stopped at the lowest point—only the lowest point locally.
- There may be a much lower point over the next hill, but the ball can't see that, and it doesn't have enough energy to climb over the hill and find the global minimum
- Gradient descent works in the same way in two or more dimensions, and has similar (and worse) problems.
- The problem is that efficient downhill directions in two dimensions and higher are harder to compute locally.
- The first is that you find a nearby local minimum, while the second is that sometimes the steepest direction is effectively across the valley, not towards the global minimum.
- All of these things are true for most of our optimization problems, including the MLP.
- We don't know where the global minimum is because we don't know what the error landscape looks like; we can only compute local features of it for the place we are in at the moment.

2.3.6 Picking Up Momentum

Let's go back to the analogy of the ball rolling down the hill. The reason that the ball stops rolling is because it runs out of energy at the bottom of the dip. If we give the ball some weight, then it will generate momentum as it rolls, and so it is more likely to overcome a small hill on the other side of the local minimum, and so more likely to find the global minimum. We can implement this idea in our neural network learning by adding in some contribution from the previous weight change that we made to the current one. There is another benefit to momentum. It makes it possible to use a smaller learning rate, which means that the learning is more stable. The only change that we need to make to the MLP algorithm is we need to add a second term to the weight updates so that they have the form:

$$w_{\zeta\kappa}^{t} \leftarrow w_{\zeta\kappa}^{t-1} + \eta\delta_o(\kappa)a_{\zeta}^{\text{hidden}} + \alpha\Delta w_{\zeta\kappa}^{t-1},$$

2.3.7 Minibatches and Stochastic Gradient Descent

- The batch algorithm converges to a local minimum faster than the sequential algorithm, which computes the error for each input individually and then does a weight update, but that the latter is sometimes less likely to get stuck in local minima.
- The reason for both of these observations is that the batch algorithm makes a better estimate of the steepest descent direction, so that the direction it chooses to go is a good one, but this just leads to a local minimum.
- The idea of a minibatch method is to find some happy middle ground between the two, by splitting the training set into random batches, estimating the gradient based on one ofthe subsets of the training set, performing a weight update, and then using the next subset to estimate a new gradient and using that for the weight update, until all of the training set have been used.
- The training set are then randomly shuffled into new batches and the next iteration takes place.
- If the batches are small, then there is often a reasonable degree of error in the gradient estimate, and so the optimisation has the chance to escape from local minima, albeit at the cost of heading in the wrong direction.

2.4 Multi- layer Perceptron in Practice

2.4.1 Amount of Training Data

- For the MLP with one hidden layer there are (L + 1) ×M + (M + 1) × N weights, where L,M,N are the number of nodes in the input, hidden, and output layers, respectively.
- The extra +1s come from the bias nodes, which also have adjustable weights.

- This is a potentially huge number of adjustable parameters that we need to set during the training phase.
- Setting the values of these weights is the job of the back-propagation algorithm,
- which is driven by the errors coming from the training data. Clearly, the more training data there is, the better for learning, although the time that the algorithm takes to learn increases.
- Unfortunately, there is no way to compute what the minimum amount of data required is, since it depends on the problem. A rule of thumb that has been around for almost as long as the MLP itself is that you should use a number of training examples that is at least 10 times the number of weights.
- This is probably going to be a very large number of examples, so neural network training is a fairly computationally expensive operation, because we need to show the network all of these inputs lots of times.

2.4.2 Number of Hidden Layers

- The number of hidden nodes, and the number of hidden layers are the fundamental to the successful application of the algorithm.
- Two hidden layers are sufficient to compute these bump functions for different inputs, and so if the function that we want to learn (approximate) is continuous, the network can compute it. It can therefore approximate any decision boundary, not just the linear one that the Perceptron computed.

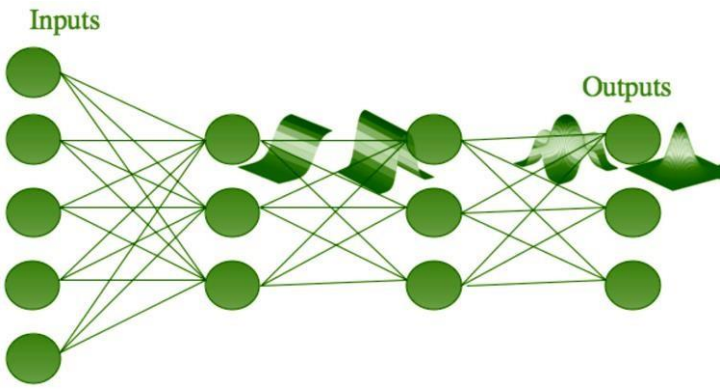


Fig 2.4.2 Schematic of the effective learning shape at each stage of the MLP.

2.4.3 When to Stop Learning

- The training of the MLP requires that the algorithm runs over the entire dataset many times, with the weights changing as the network makes errors in each iteration.
- How to decide when to stop learning ?
- setting some predefined number N of iterations, and running until that is reached runs the risk that the network has overfitted by then, or not learnt sufficiently, and only stopping when some predefined minimum error is reached might mean the algorithm never terminates, or that it overfits.
- If we plot the sum-of-squares error during training, it typically reduces fairly quickly during the first few training iterations, and then the reduction slows down as the learning algorithm performs small changes to find the exact local minimum.
- We don’t want to stop training until the local minimum has been found, but keeping on training too long leads to overfitting of the network. This is where the validation set comes in useful.
- We train the network for some predetermined amount of time, and then use the validation set to estimate how well the network is generalising. We then carry on training for a few more iterations, and repeat the whole process.
- At some stage the error on the validation set will start increasing again, because the network has stopped learning about the function that generated the data, and started to learn about the noise that is in the data itself (shown in Figure 2.4.3) At this stage we stop the training. This technique is called early stopping.

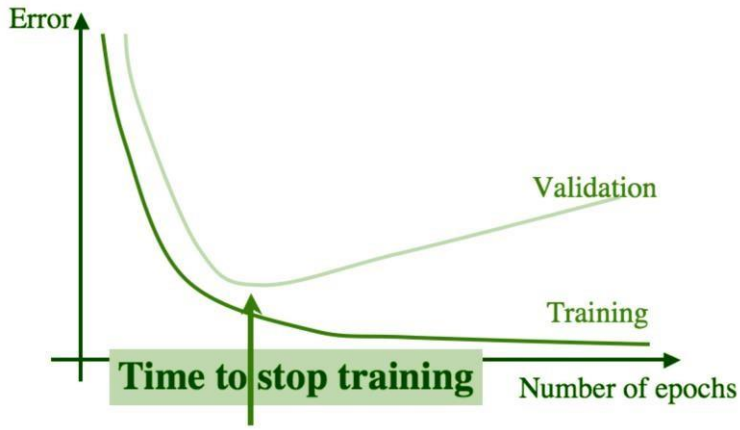


Fig 2.4.3 The effect of overfitting on the training and validation error curves, with the point at which early stopping will stop the learning marked.

2.5 Examples of using the MLP

2.5.1 A Regression Problem

- Take a set of samples generated by a simple mathematical function, and try to learn the generating function (that describes how the data was made) so that we can find the values of any inputs, not just the ones we have training data for.
- The function that we will use is a very simple one, just a bit of a sine wave.

```
x = np.ones((1,40))*np.linspace(0,1,40)
t = np.sin(2*np.pi*x) + np.cos(4*np.pi*x) +
np.random.randn(40)*0.2 x = x.T t = t.T
```

- The reason why we have to use the reshape() method is that NumPy defaults to lists for arrays that are N ×1; compare the results of the np.shape() calls below, and the effect of the transpose operator .T on the array:

```
>>> x = np.linspace(0,1,40)
>>> np.shape(x)
(40,)
>>> np.shape(x.T)
```

```
(40,)
>>>
>>> x = np.linspace(0,1,40).reshape((1,40))
>>> np.shape(x)
(1, 40)
>>> np.shape (x.T)
(40, 1)
```

- You can plot this data to see what it looks like using:>>> import pylab as pl
  >>> pl.plot(x,t,'.')
- Now train an MLP on the data. There is one input value, x and one output value t, so the neural network will have one input and one output.
- Before getting started, normalize the data using the method and then separate the data into training, testing, and validation sets.
- For this example there are only 40 datapoints, and we'll use half of them as the training set, although that isn't very many and might not be enough for the algorithm to learn effectively.
- We can split the data in the ratio 50:25:25 by using the odd-numbered elements as training data, the even-numbered ones that do not divide by 4 for testing, and the rest for validation: train = x[0::2,:] test = x[1::4,:] valid = x[3::4,:] traintarget = t[0::2,:] testtarget = t[1::4,:] validtarget = t[3::4,:]
- With that done, it is just a case of making and training the MLP. To start with, we will construct a network with three nodes in the hidden layer, and run it for 101 iterations with a learning rate of 0.25, just to see that it works:

```
>>> import mlp
>>> net = mlp.mlp(train,traintarget,3,outtype='linear')
>>> net.mlptrain(train,traintarget,0.25,101)
```

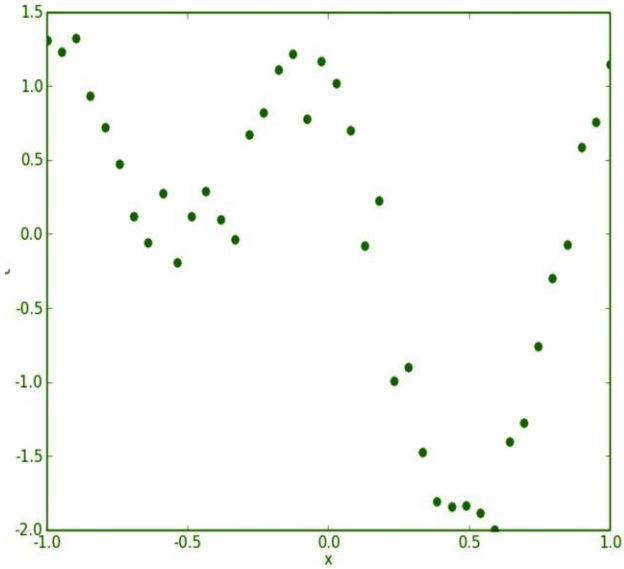
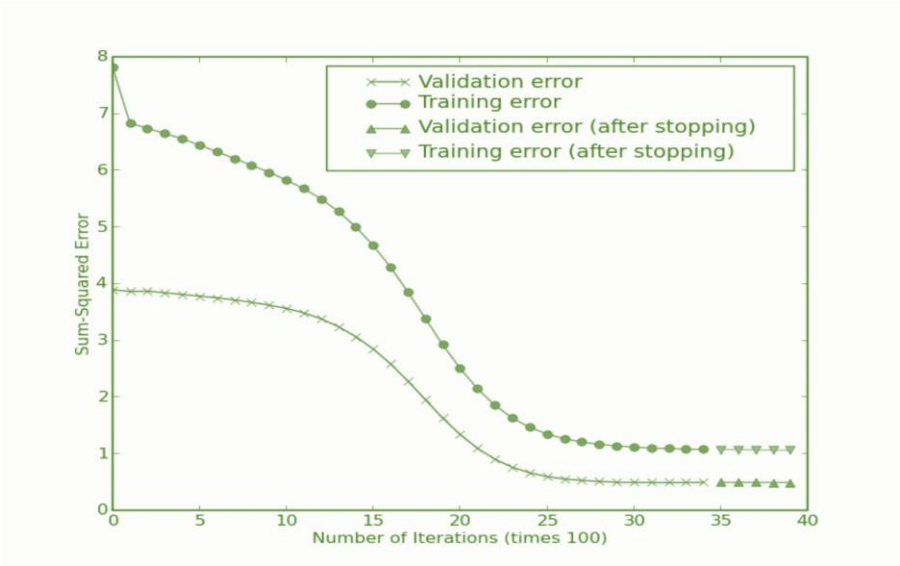


FIGURE 2.5.1 The data that we will learn using an MLP, consisting of some samples from a sine wave with Gaussian noise added.

- The output from this will look something like: Iteration: 0 Error: 12.3704163654 Iteration: 100 Error: 8.2075961385
- Since the error is decreasing. We need to do two things: • work out how many hidden nodes we need,
- decide how long to train the network for.
- In order to solve the first problem, we need to test out different networks and see which get lower errors, but to do that properly we need to know when to stop training.
- So solve the second problem first, which is to implement early stopping.
- Train the network for a few iterations (let's make it 10 for now), then evaluate the validation set error by running the network forward (i.e., the recall phase).
- Learning should stop when the validation set error starts to increase.



- Figure 4.13 gives an example of the output of running the function. It plots the training and validation errors.
- The point at which early stopping makes the learning finish is the point where there is a missing validation datapoint.
- Based on these numbers, we would select a network with a small number of hidden nodes, certainly between 2 and 10 (and the smaller the better, in general), since their maximum error is much smaller than a network with just 1 hidden node.
- Note also that the error increases once too many hidden nodes are used, since the network has too much variation for the problem.

No. of hidden nodes	1	2	3	5	10	25	50
Mean error	2.21	0.52	0.52	0.52	0.55	1.35	2.56
Standard deviation	0.17	0.00	0.00	0.02	0.00	1.20	1.27
Max error	2.31	0.53	0.54	0.54	0.60	3.230	3.66
Min error	2.10	0.51	0.50	0.50	0.47	0.42	0.52

2.5.2 Classification with the MLP

- Using the MLP for classification problems is not radically different once the output encoding has been worked out.
- The inputs are easy: they are just the values of the feature measurements (suitably normalised).

- There are a couple of choices for the outputs. The first is to use a single linear node for the output, y, and put some thresholds on the activation value of that node.

For example, for a four-class problem, we could use:

$$\text{Class is: } \begin{cases} C_1 & \text{if } y \leq -0.5 \\ C_2 & \text{if } -0.5 < y \leq 0 \\ C_3 & \text{if } 0 < y \leq 0.5 \\ C_4 & \text{if } y > 0.5 \end{cases}$$

- A more suitable output encoding is called 1-of-N encoding. A separate node is used to represent each possible class, and the target vectors consist of zeros everywhere except for in the one element that corresponds to the correct class, e.g., (0, 0, 0, 1, 0, 0) means that the correct result is the 4th class out of 6.
- Once the network has been trained, performing the classification is easy: simply choose the element y_k of the output vector that is the largest element of y.
- This generates an unambiguous decision, since it is very unlikely that two output neurons will have identical largest output values. This is known as the hard-max activation function.
- Suppose that we are doing two-class classification, and 90% of our data belongs to class 1. In that case, the algorithm can learn to always return the negative class, since it will be right 90% of the time, but still a completely useless classifier!
- So you should generally make sure that you have approximately the same number of each class in your training set.

• A Classification Example: The Iris Dataset

- Classifying examples of three types of iris (flower) by the length and width of the sepals and petals and is called iris.
- In the dataset, the last column is the class ID, and the others are the four measurements. normalizing the inputs, but using the maximum rather than the variance, and leaving the class IDs alone for now.

- convert the targets into 1-of-N encoding, from their current encoding s class 1, 2, or 3. make a new matrix that is initially all zeroes,
- And simply set one of the entries to be 1:
- Separate the data into training, testing, and validation sets.
- There are 150 examples in the dataset, and they are split evenly amongst the three classes, so the three classes are the same size and we don't need to worry about discarding any datapoints.
- We'll split them into half training, and one quarter each testing and validation. If you look at the file, you will notice that the first 50 are class 1, the second 50 class 2, etc.
- Finally set up and train the network.

```
>>> import mlp
>>> net = mlp.mlp(train,traint,5,outtype='softmax')
>>> net.earlystopping(train,traint,valid,validt,0.1)
>>>                net.confmat(test,testt)
Confusion matrix is:
[[ 16. 0. 0.]
[ 0. 12. 2.]
[ 0. 1. 6.]]
Percentage Correct: 91.8918918919
```

- The algorithm got nearly all of the test data correct, misclassifying just two examples of class 2 and one of class 3.

2.5.4 Time-Series Prediction

There is a common data analysis task known as time-series prediction, where we have a set of data that show how something varies over time, and we want to predict how the data will vary in the future.

It is useful in any field where there is data that appears over time, which is to say almost any field.

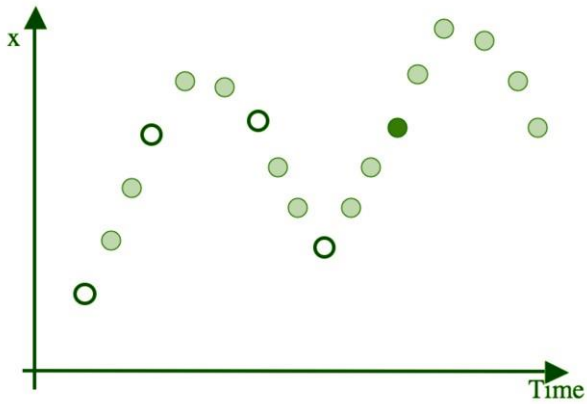


Fig 2.5.4.1 Part of a time-series plot, showing the datapoints and the meanings of T and k.

$$y = x(t+\tau) = f(x(t), x(t-\tau), \ldots, x(t-k\tau)),$$

The target data for training the neural network is simple, because it comes from further up the time-series, and so training is easy. Suppose that T=2 and k = 3.

Then the first input data are elements 1, 3, 5 of the dataset, and the target is element 7. The next input vector is elements 2, 4, 6, with target 8, and then 3, 5, 7 with target 9. Train the network by passing through the time-series and then press on into the future making predictions.

Figure 4.14 shows an example of a time-series with T=3 and k = 4, with a set of datapoints that make up an input vector marked as white circles, and the target coloured green.

2.5.5 Data Compression: The Auto-Associative Network

- Train the network to reproduce the inputs at the output layer (called auto-associative learning; sometimes the network is known as an autoencoder). The network is trained so that whatever you show it at the input is reproduced at the output, which doesn't seem very useful at first, but suppose that we use a hidden layer that has fewer neurons than the input layer.

- This bottleneck hidden layer has to represent all of the information in the input, so that it can be reproduced at the output.

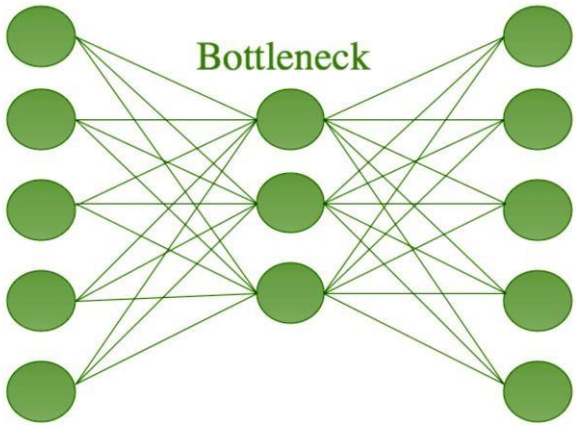


Fig 2.5.5.1 The auto-associative network. The network is trained to reproduce the inputs at the outputs, passing them through the bottleneck hidden layer that compresses the data.

- It therefore performs some compression of the data, representing it using fewer dimensions than were used in the input.
- This gives us some idea of what the hidden layers of the MLP are doing: they are finding a different (often lower dimensional) representation of the input data that extracts important components of the data, and ignores the noise.
- This auto-associative network can be used to compress images and other data.
- A schematic of this is shown in Figure 2.5.5.2: the 2D image is turned into a 1D vector of inputs by cutting the image into strips and sticking the strips into a long line.
- The values of this vector are the intensity (colour) values of the image, and these are the input values.
- The network learns to reproduce the same image at the output, and the activations of the hidden nodes are recorded for each image.

- After training, we can throw away the input nodes and first set of weights of the network. If we insert some values in the hidden nodes (their activations for a particular image; see Figure 2.5.5.3), then by feeding these activations forward through the second set of weights, the correct image will be reproduced on the output. S
- Auto-associative networks can also be used to denoise images, since, after training, the network will reproduce the trained image that best matches the current (noisy) input.

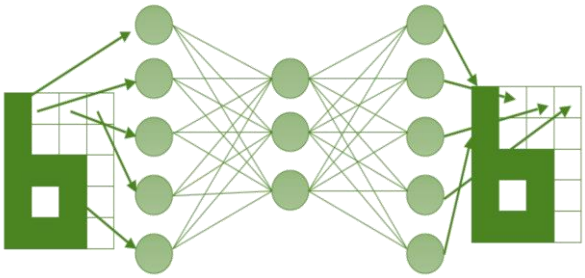


FIGURE 2.5.5.2 Schematic showing how images are fed into the auto-associative network for compression.

FIGURE 2.5.5.3 Schematic showing how the hidden nodes and second layer of weights can be used to regain the compressed images after the network has been trained.

2.6 Overview – Deriving Back Propagation

• Three things need to recall when deriving back propagation.

1. derivative (with respect to x) of $\frac{1}{2}x^2$
2. Chain rule, which says that $\frac{dy}{dx} = \frac{dy}{dt}\frac{dt}{dx}$
3. $\frac{dy}{dx} = 0$ if y is not a function of x.

2.6.1 The Network Output and the Error

The output of the neural network (the end of the forward phase of the algorithm) is a function of three things:

- the current input (x)
- the activation function g(·) of the nodes of the network
- the weights of the network (v for the first layer and w for the second)

2.6.2 The Error of the Network

- The Perceptron learning rule is minimizing the error function $E = \sum_{k=1}^{N} y_k - t_k.$
- But there are some problems with this: if $t_k > y_k$, then the sign of the error is different to when $y_k > t_k$, so if we have lots of output nodes that are all wrong, but some have positive sign and some have negative sign, then they might cancel out.
- Instead, choose the sum-of-squares error function, which calculates the difference between y_k and t_k for each node k, squares them,and adds them together.

$$\begin{aligned} E(\mathbf{w}) &= \frac{1}{2}\sum_{k=1}^{N}(y_k - t_k)^2 \\ &= \frac{1}{2}\sum_{k=1}^{N}\left[g\left(\sum_{j=0}^{M} w_{jk}a_j\right) - t_k\right]^2 \end{aligned}$$

- The Perceptron and index the input nodes by i and the output nodes by k, so above Equation will be replaced by:
- Use a gradient descent algorithm that adjusts each weight w_{ik} for fixed values of i and k, in the direction of the negative gradient of E(w).
- The notation δ means the partial derivative. It isUsed because there are lots of different functions that we can differentiate E with respect to: all of the different weights.

$$\frac{1}{2}\sum_{k=1}^{N}\left[g\left(\sum_{i=0}^{L} w_{ik}x_i\right) - t_k\right]^2$$

$$\begin{aligned}\frac{\partial E}{\partial w_{\iota\kappa}} &= \frac{\partial}{\partial w_{\iota\kappa}}\left(\frac{1}{2}\sum_{k=1}^{N}(y_k - t_k)^2\right) \\ &= \frac{1}{2}\sum_{k=1}^{N} 2(y_k - t_k)\frac{\partial}{\partial w_{\iota\kappa}}\left(y_k - \sum_{i=0}^{L} w_{i\kappa}x_i\right)\end{aligned}$$

$$\frac{\partial E}{\partial w_{\iota\kappa}} = \sum_{k=1}^{N}(t_k - y_k)(-x_\iota).$$

So the weight update rule (when we include the learning rate η) is:

$$w_{\iota\kappa} \leftarrow w_{\iota\kappa} + \eta(t_\kappa - y_\kappa)x_\iota,$$

2.6.3 Requirements of an Activation Function

In order to model a neuron we want an activation function that has the following properties:

- it must be differentiable so that we can compute the gradient
- it should saturate (become constant) at both ends of the range, so that the neuron either fires or does not fire
- it should change between the saturation values fairly quickly in the middle There is a family of functions called sigmoid functions because they are S-shaped .The form in which it is generally used is:

$$a = g(h) = \frac{1}{1+\exp(-\beta h)},$$

$$\begin{aligned} g'(h) = \frac{dg}{dh} &= \frac{d}{dh}(1+e^{-\beta h})^{-1} \\ &= -1(1+e^{-\beta h})^{-2}\frac{de^{-\beta h}}{dh} \\ &= -1(1+e^{-\beta h})^{-2}(-\beta e^{-\beta h}) \\ &= \frac{\beta e^{-\beta h}}{(1+e^{-\beta h})^2} \\ &= \beta g(h)(1-g(h)) \\ &= \beta a(1-a) \end{aligned}$$

2.6.4 The Output Activation Functions

- The sigmoidal activation function that we have created is aimed at making the nodes act a bit like neurons, either firing or not firing.
- This is very important in the hidden layer, but earlier in the chapter we have observed two cases where it is not suitable for the output neurons.
- One was regression, where we want the output to be continuous, and one was multiclass classification, where we want only one of the output neurons to fire.
- We identified possible activation functions for these cases, and here we will derive the delta term δo for them. As a reminder, the three functions are:

$$\textbf{Linear} \quad y_\kappa = g(h_\kappa) = h_\kappa$$

$$\textbf{Sigmoidal} \quad y_\kappa = g(h_\kappa) = 1/(1+\exp(-\beta h_\kappa)$$

$$\textbf{Soft-max} \quad y_\kappa = g(h_\kappa) = \exp(h_\kappa)/\sum_{k=1}^{N}\exp(h_k)$$

2.6.5 An Alternative Error Function

- In this probabilistic interpretation of the outputs, we can ask how likely we are to see each target given the set of weights that we are using.

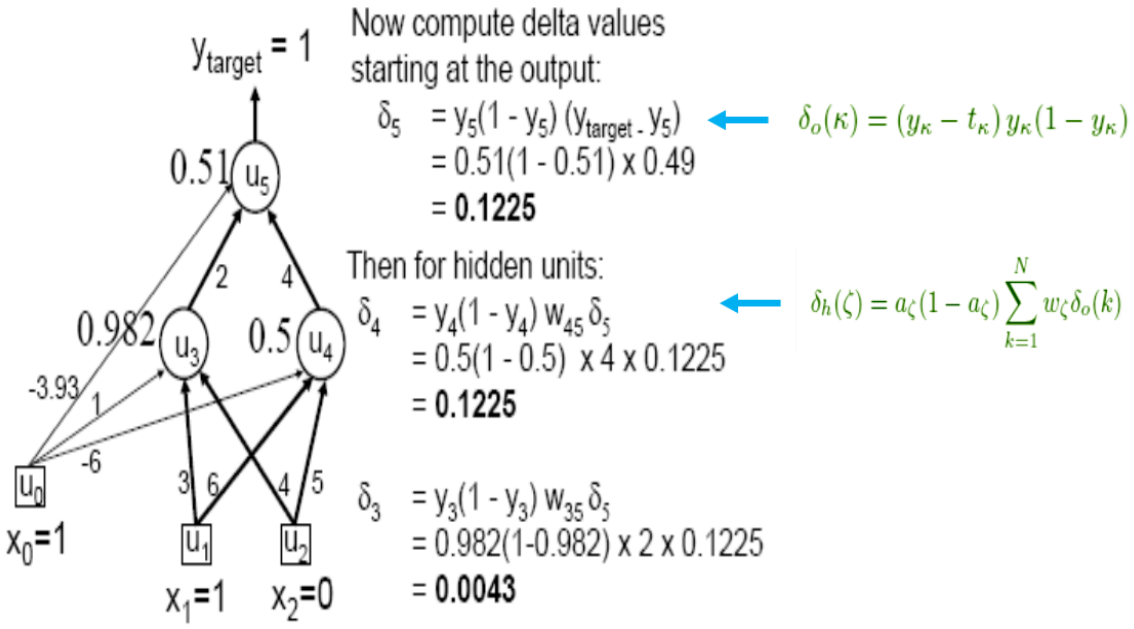
- This is known as the likelihood and the aim is to maximise it, so that we predict the targets as well as possible.
- If we have a 1 output node, taking values 0 or 1, then the likelihood is:

$$p(t|\mathbf{w}) = y_k^{t_k} \left(1 - y_k\right)^{1-t_k}.$$

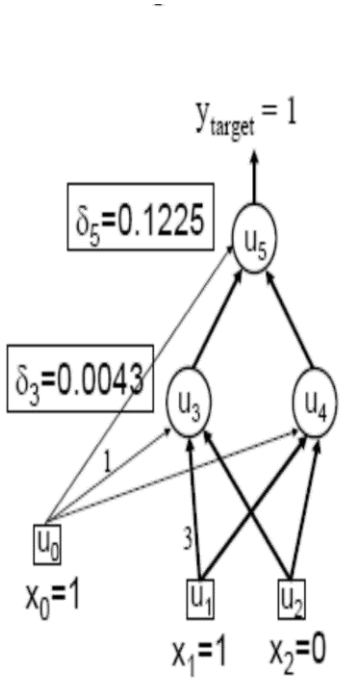
- In order to turn this into a minimisation function we put a minus sign in front, and it will turn out to be useful to take the logarithm of it as well, which produces the crossentropy error function, which is (for N output nodes):

$$E_{\mathrm{ce}} = -\sum_{k=1}^{N} t_k \ln(y_k),$$

Worked example: Backward Pass



Update Weights Using Generalized Delta Rule (BP)



♦ Set learning rate $\eta = 0.1$

Change weights by:

$$\Delta w_{ij} = \eta \delta_j y_i$$

♦ e.g.bias weight on u_3:

$\Delta w_{03} = \eta \delta_3 x_0$

$=0.1*0.0043*1$

$= 0.0004$

So, new w_{03} ⌧ $w_{03}(old)+\Delta w_{03}$

$=1+0.0004=1.0004$

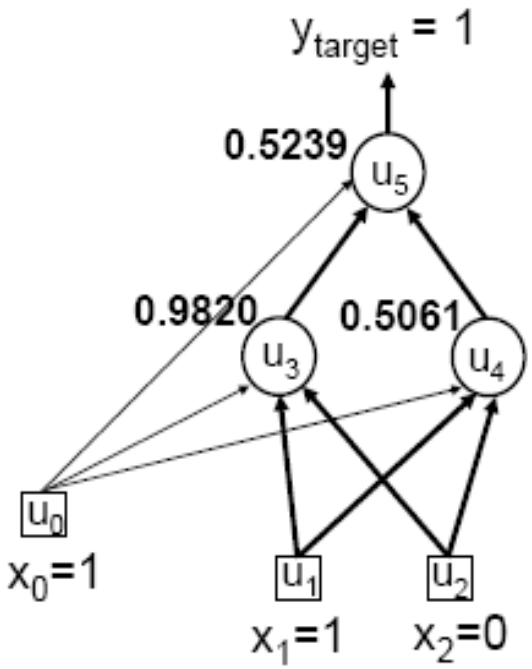
♦ and likewise:

w_{13} ⌧ 3 + 0.0004

= 3.0004

i	j	w_{ij}	δ_j	y_i	Updated w_{ij}
0	3	**1**	0.0043	1.0	**1.0004**
1	3	**3**	0.0043	1.0	**3.0004**
2	3	**4**	0.0043	0.0	**4.0000**
0	4	**-6**	0.1225	1.0	**-5.9878**
1	4	**6**	0.1225	1.0	**6.0123**
2	4	**5**	0.1225	0.0	**5.0000**
0	5	**-3.92**	0.1225	1.0	**-3.9078**
3	5	**2**	0.1225	0.9820	**2.0120**
4	5	**4**	0.1225	0.5	**4.0061**

Verification that it works



On next forward pass:

The new activations are:

$y_3 = f(4.0008) = 0.9820$

$y_4 = f(0.0245) = 0.5061$

$y_5 = f(0.0955) = \mathbf{0.5239}$

Thus the new error

$(y_{target} - y_5)=(1 - 0.5239)=0.476$

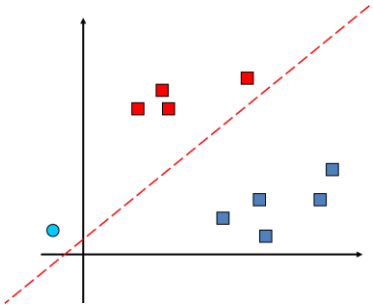
has been reduced by **0.014**

(from **0.490** to **0.476**)

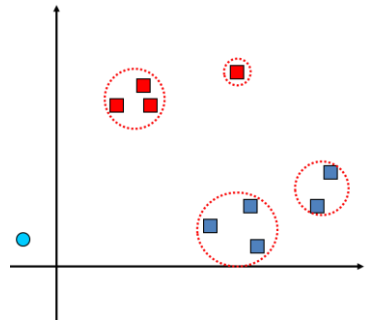
Ref: "Neural Network Learning & Expert Systems" by Stephen Gallant

2.7 Radial Basis Functions and Splines

MLP



RBF



- RBF is Completely different approach by viewing the design of a neural network as a **curve-fitting** (approximation) problem in high-dimensional space.
- In RBF → Local neurons are used.
    - → neuron only responds to inputs in one particular part of the **input space**.
    - → if inputs are similar, then the responses to those inputs should also be similar, and so the **same neuron** should respond.
    - → Eg. classification task- since if **two input vectors are similar** -> they should belong to the **same class**
    - → i) Weight space ii) Receptive fields

2.7 Radial Basis Functions and Splines

2.7.1 RECEPTIVE FIELDS

Set of 'nodes are imagined to be sitting in weight space, and we can change their locations by adjusting the weights. We want to decide how strongly a node matches the current input, so we pretend that input space and weight space are the same, and measure the distance between the input vector position and the position of each node.

The activation of these nodes can then be computed according to their distance to the current input, To put this idea of nodes firing when they are 'close' to the input into some sort of context, We can extend this to particular sensory neurons as well,

so that the response of particular neurons may depend on the location of the stimulus.

We might want to know what shape these receptive fields are, and how the response of the rod (or neuron) changes as the stimulus moves away from the area that matches the rod.

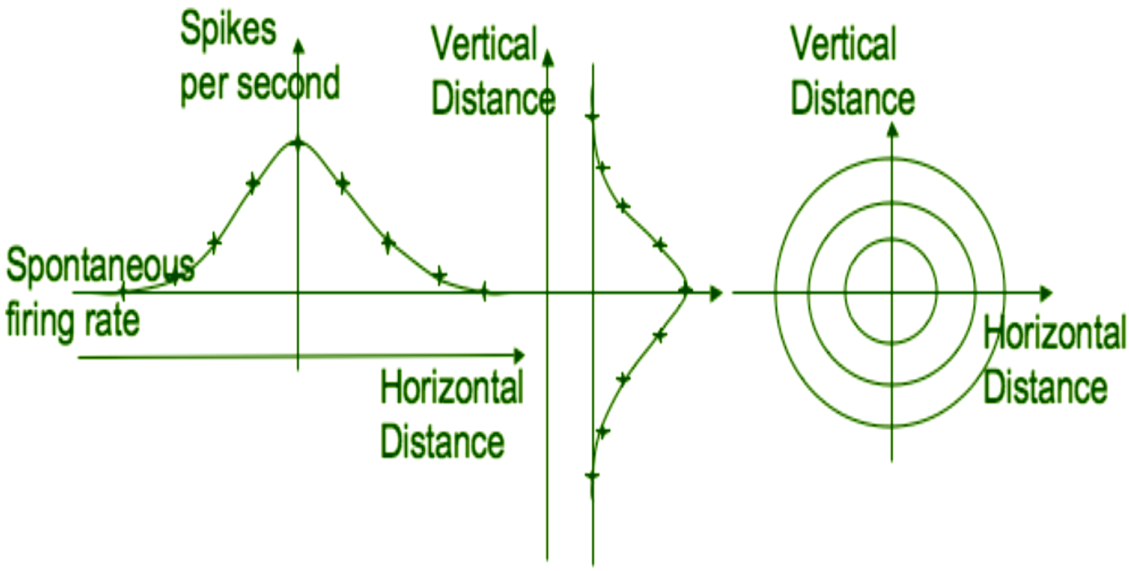
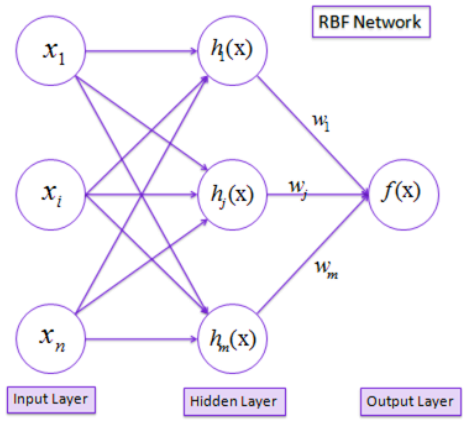


Fig 2.7.1 Left:Count of the number of spikes per second as the distance of a rod from the light varies horizontally. Note that it does not go to zero, but to the spontaneous firing rate of the neuron, which is how often it fires without input. Centre: The same thing for vertical motion. Right: The combination of the two makes a set of circles.

2.8 RBF Network

- Radial Basis Function Networks (RBFN) consists of 3 layers
- an input layer
- a hidden layer
- an output layer
- The hidden units provide a set of functions that constitute an arbitrary basis for the input patterns.
- Hidden units are known as radial centers and represented by the vectors c1; c2; …. ; c_h

- Transformation from input space to hidden unit space is nonlinear whereas transformation from hidden unit space to output space is linear



$$f(\mathrm{x}) = \sum_{j=1}^{m} w_j h_j(\mathrm{x})$$

$$g(\mathrm{x}, \mathrm{w}, \sigma) = \exp\left(\frac{-\|\mathrm{x} - \mathrm{w}\|^2}{2\sigma^2}\right).$$

2.8 RBF Network

- Using Gaussian activations, where the output of a neuron is proportional to the distance between the input and the weight, gives us receptive fields. The Gaussian activations mean that normalising the input vectors is very important for the RBF network;
- For any input that we present to a set of these neurons, some of them will fire strongly, some weakly, and some will not fire at all, depending upon the distance between the weights and the particular input in weight space.
- We can treat these nodes as a hidden layer, just as we did for the MLP, and connect up some output nodes in a second layer. This simply requires adding weights from each hidden (RBF) neuron to a set of output nodes.
- This is known as an RBF network, and a schematic is shown in Figure 2.8.1
- In the figure, the nodes in both the hidden and output layer are drawn the same, but we haven't decided what kind of nodes to use in the output layer—they don't need to have Gaussian activations.
- There is a bias input for the output layer, which deals with the situation when none of the RBF neurons fire. Since we already know exactly how to train the
- Perceptron, training this second part of the network is easy.

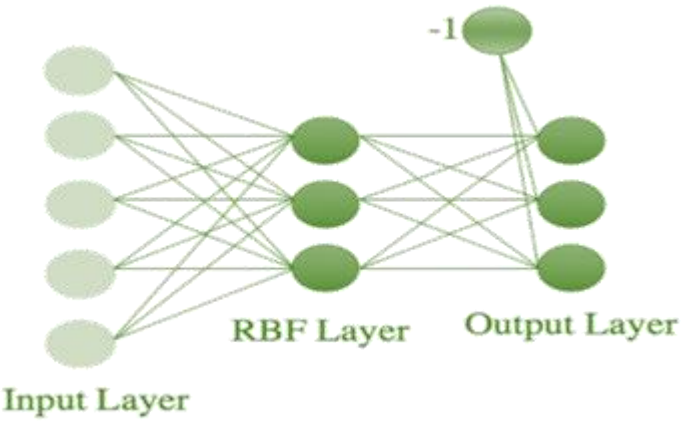


Fig 2.8.1 The Radial Basis Function network consists of input nodes connected by weights to a set of RBF neurons, which fire proportionally to the distance between the input and the neuron in weight space. The activations of these nodes are used as inputs to the second layer, which consists of linear nodes. The schematic looks very similar to the MLP except for the lack of a bias in the hidden layer.

- The RBF network is a universal approximator, just like the MLP. Imagine that we fill the entire space with RBF nodes equally spaced in all directions, so that their receptive fields just overlap, as in Figure 2.8.2.
- Now, no matter what the input, there is an RBF node that recognises it and can respond appropriately to it. If we need to make the outputs more finely grained, then we just add more RBFs at the relevant positions and reduce the radius of the receptive fields; and if we don't care, we can just make the receptive fields of each node bigger and use fewer of them.
- RBF networks never have more than one layer of non-linear neurons, in contrast to the MLP. However, there are many similarities between the two networks: they are both supervised learning algorithms that form universal approximators.
- In an RBF network, when we see an input several of the nodes will activate to some degree or other, according to how close they are to the input, and the combination of these activations will enable the network to decide how to respond.

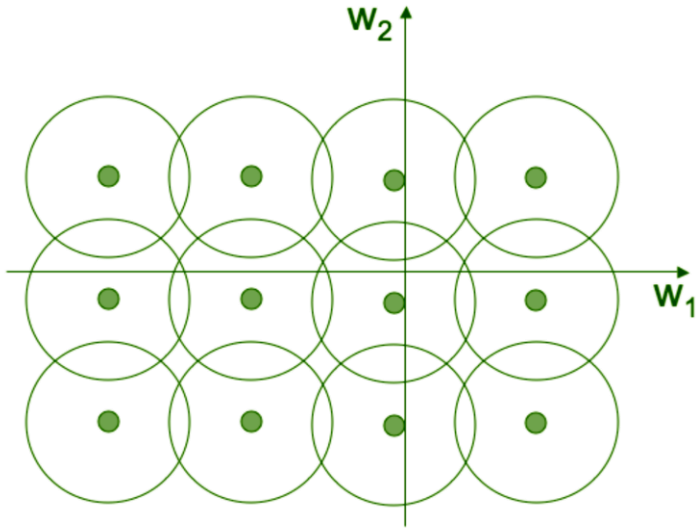


FIGURE 2.8.2 We can space out RBF nodes to cover the whole of space by continuing this pattern everywhere, so that the network acts as a universal approximator, since there is an output for every possible input.

2.8.1 Training the RBF Network

- In the MLP we used back-propagation of error to adjust first the output layer weights, and then the hidden layer weights.
- We can do exactly the same thing with the RBF network, by differentiating the relevant activation functions. However, there are simpler and better alternatives for RBF networks.
- They do not need to compute gradients for the hidden nodes and so they are significantly faster.
- The important thing to notice is that the two types of node provide different functions, and so they do not need to be trained together.
- The purpose of the RBF nodes in the hidden layer is to find a non-linear representation of the inputs, while the purpose of the output layer is to find a linear combination of those hidden nodes that does the classification.

- So we can split the training into two parts: position the RBF nodes, and then use the activations of those nodes to train the linear outputs.

The Radial Basic Function Algorithm

- Position the RBF centres by either:
  - using the k-means algorithm to initialise the positions of the RBF centres OR
  - setting the RBF centres to be randomly chosen datapoints • Calculate the actions of the RBF nodes using Equation (5.1)
- Train the output weights by either:
  - using the Perceptron OR
  - computing the pseudo-inverse of the activations of the RBF centres (this will be described shortly)
- separation of the two learning parts, we can do better than a Perceptron for training the outputs weights. For each input vector, we compute the activation of all the hidden nodes, and assemble them into a matrix G.
- So each element of G, say G_{ij} , consists of the activation of hidden node j for input i. The outputs of the network can then be computed as y = GW for set of weights W.
- If we were able to get all of the outputs correct, then we could write t = GW. Now we just need to calculate the matrix inverse of G, to get $W= G^{-1}t$.
- The matrix inverse is only defined if a matrix is square, and this one probably isn't—there is no reason why the number of hidden nodes should be the same as the number of training inputs.
- There is one thing that we haven't considered yet, and that is the size of the receptive fields σ for the nodes.We can avoid the problem by giving all of the nodes the same size, and testing lots of different sizes using a validation set to select one that works.
- The most common choice is to pick the width of the Gaussian as $\sigma= d/\sqrt{2M}$, where M is the number of RBFs.
- There is another way to deal with the fact that there may be inputs that are outside the receptive fields of all nodes, and that is to use normalised Gaussians, so that there

is always at least one input firing: the node that is closest to the current input, even if that is a long way off. It is a modification of Equation (5.1) and it looks like the soft-max function:

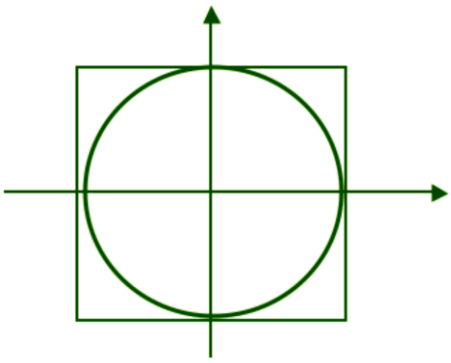
$$g(\mathbf{x}, \mathbf{w}, \sigma) = \frac{\exp(-\|\mathbf{x} - \mathbf{w}\|/2\sigma^2)}{\sum_i \exp(-\|\mathbf{x} - \mathbf{w}_i\|/2\sigma^2)}.$$

- Using the RBF network on the iris dataset with five RBF centres gives similar results to the MLP, with well over 90% classification accuracy.
- With the MLP, one question that we failed to find a nice answer to was how to pick the number of hidden nodes, and we were reduced to training lots of networks with different numbers of nodes and using the one that performed best on the validation set.

2.9 Curse of Dimensionality

- The essence of the curse is the realization that as the number of dimensions increases, the volume of the unit hypersphere does not increase with it.
- The unit hypersphere is the region we get if we start at the origin (the centre of our coordinate system) and draw all the points that are distance 1 away from the origin.
- In 2 dimensions we get a circle of radius 1 around (0, 0) (drawn in Figure 2.9.1), and in 3D we get a sphere around (0, 0, 0) (Figure 2.9.2).
- In higher dimensions, the sphere becomes a hypersphere.
- The following table shows the size of the unit hypersphere for the first few dimensions, and the graph in Figure 2.9.3 shows the same thing, but also shows clearly that as the number of dimensions tends to infinity, so the volume of the hypersphere tends to zero.

Dimension	Volume
1	2.0000
2	3.1416
3	4.1888
4	4.9348
5	5.2636
6	5.1677
7	4.7248
8	4.0587
9	3.2985
10	2.5502



.

Fig 2.9.1The unit circle in 2D with its bounding box

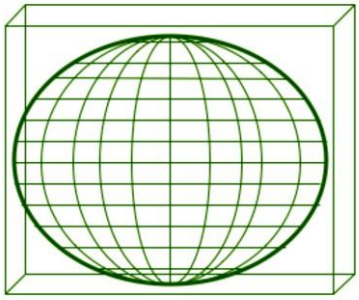


Fig 2.9.2 The unit sphere in 3D with its bounding cube. The sphere does not reach as far into the corners as the circle does, and this gets more noticeable as the number of dimensions increases

The curse of dimensionality will apply to our machine learning algorithms because as the number of input dimensions gets larger, we will need more data to enable the algorithm to generalize sufficiently well. Our algorithms try to separate data into classes based on the features; therefore as the number of features increases, so will the number of data points we need.

- For this reason, we will often have to be careful about what information we give to the algorithm, meaning that we need to understand something about the data in advance. Regardless of how many input dimensions there are, the point of machine learning is to make predictions on data inputs.

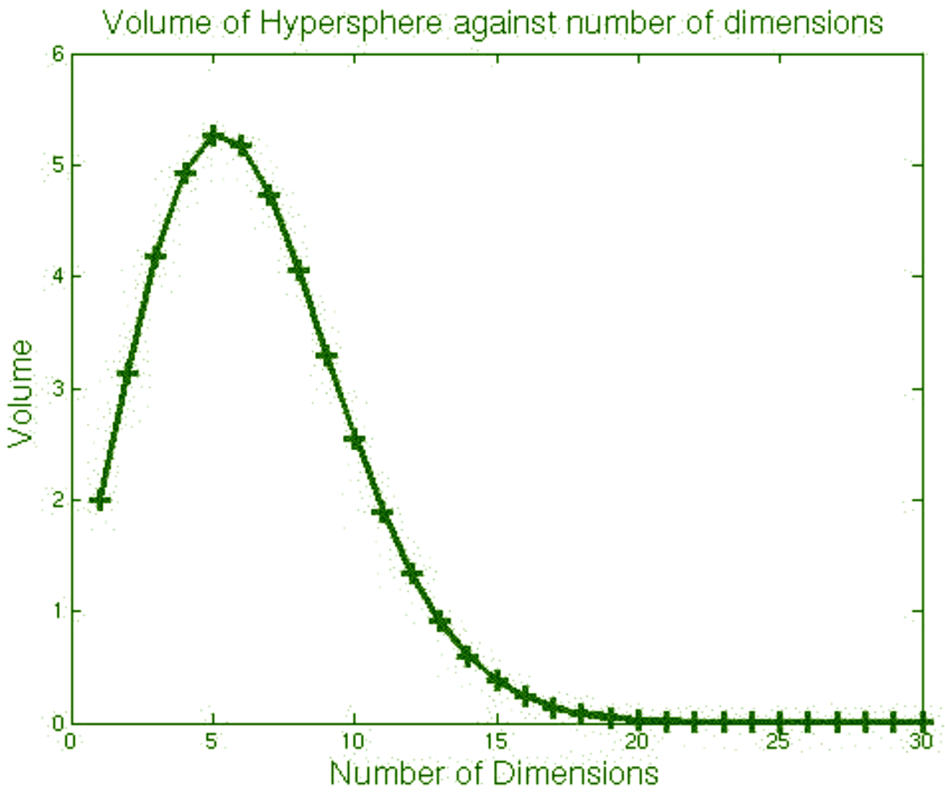


Fig 2.9.3 The volume of the unit hypersphere for different numbers of dimensions.

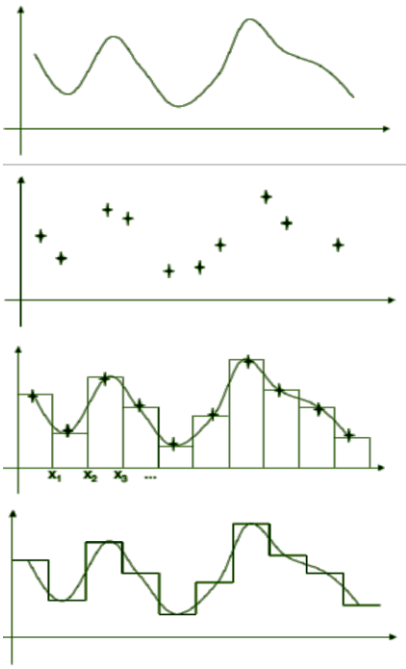
2.10 Interpolations and Basis Functions

Function approximation:

- Given some data, find a function that goes through the data without overfitting to the noise, so that values between the known datapoints can be inferred or interpolated.
- The RBF network solves this problem by each of the basis functions making a contribution to the output whenever the input is within its receptive field. So several RBF nodes will probably respond for each input.
- If the datapoint is within the receptive field of this function then we listen only to this function, otherwise we ignore it and listen to some other function.
- If each function just returns the average value within its patch, then for one-dimensional data we get a histogram output, as is shown in Figure 2.10.1. We can

extend this a bit further so that the lines are not horizontal, but instead reflect the first derivative of the curve at that point, as is shown in Figure 2.10.2.

- This is all right, but we might want the output to be continuous, so that the line within
- the first bin meets up with the line in the second bin at the boundary, so we can add the extra constraint that the lines have to meet up as well. This gives the curve in Figure 2.10.3.
- if we use cubic functions (i.e., polynomials with x^3, x^2, x and constant components) to approximate each piece of data, then we can get results like those shown in Figure 2.10.4.



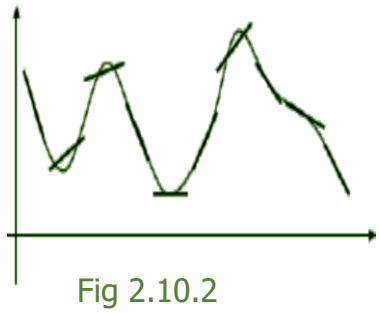


Fig 2.10.2

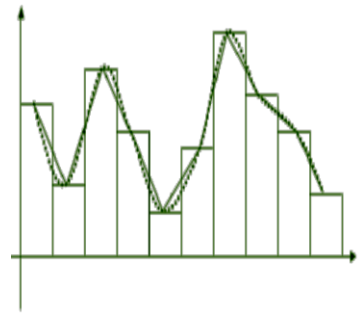


Fig 2.10.3

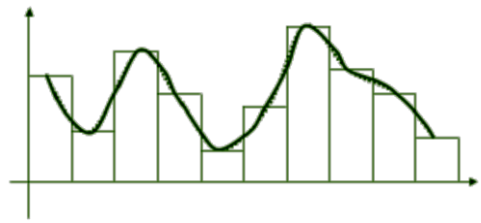


Fig2.10.1 Top: Curve showing a function. Second: A set of datapoints from the curve. Third:Putting a straight horizontal line through each point creates a histogram that describes an approximation to the curve. Bottom:That approximation.

2.10.1 Bases and Basis Expansion

- Radial basis functions and several other machine learning algorithms can be written in this form:

$$f(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i \Phi_i(\mathbf{x}),$$

- The Φi(x) are known as basis functions and they are parameters of the model that are chosen.
- Looking at the third graph of Figure 2.10.1, first function should only be defined between 0 and x_1, the next between x_1 and x_2, and so on. These points x_i are called knotpoints and they are generally evenly spaced, but choosing how many of them there should be is not necessarily easy.
- The more knotpoints there are,the more complex the model can be, in which case the model is more likely to overfit, and needs more training data
- Ways to choose Φi(x)
- constant function Φ(x) = 1.
- Now the model would have value α1 to the left of x_1, value α2 between x1 and x2, etc. So depending upon how we fit the spline model to the data, the model will have different values, but it will certainly be constant in each region. This is sufficient to make the straight line approximation shown at the bottom of Figure 2.10.1
- A linear function that has value Φ(x) = x within the region. In this case, we can make Figure 2.10.2, where each point is represented by a straight line that is not necessarily horizontal.
- This represents the line close to each point fairly well, but looks messy because the line segments do not meet up.
- There is a simpler way to encode this, which is to add some extra basis functions.
- As well as $\Phi_1(x) = 1$, $\Phi_2(x) = x$, we add some basis functions that insist that the value is 0 at the boundary with x_1: $\Phi_3(x) = (x - x_1)+$, and the next with the boundary at

x_2: $\Phi_4(x) = (x - x_2)+$, etc., where $(x)+ = x$ if $x > 0$ and 0 otherwise.

•These functions are sufficient to insist that the knotpoint values are enforced, since one is defined on each knotpoint. This is then enough for us to construct the approximation shown in Figure 5.7.

2.10.2 The Cubic Spline

- Adding extra powers of x, but it turns out that the cubic spline is generally sufficient. This has four basic basis functions
- ($\Phi_1(x) = 1$, $\Phi_2(x) = x$, $\Phi_3(x) = x^2$, $\Phi_4(x) = x^3$), and then as many extras as there are knotpoints, each of the form $\Phi_{4+i}(x) = (x - xi)^3 +$.
- This function constrains the function itself and also its first two derivatives to meet at each knotpoint. Notice that while the Φ s are not linear, we are simply adding up a weighted sumof them, and so the model is linear in them.

2.10.3 Fitting the Spline to the Data

• Defining the sum-of-squares error and to minimize that, which is known in the statistical literature as least-squares fitting. computing the least-squares fit is a linear problem. As with the MLP, the error that we are trying to minimise is:

$$E(y, f(x)) = \sum_{i=1}^{N} (y_i - f(x_i))^2.$$

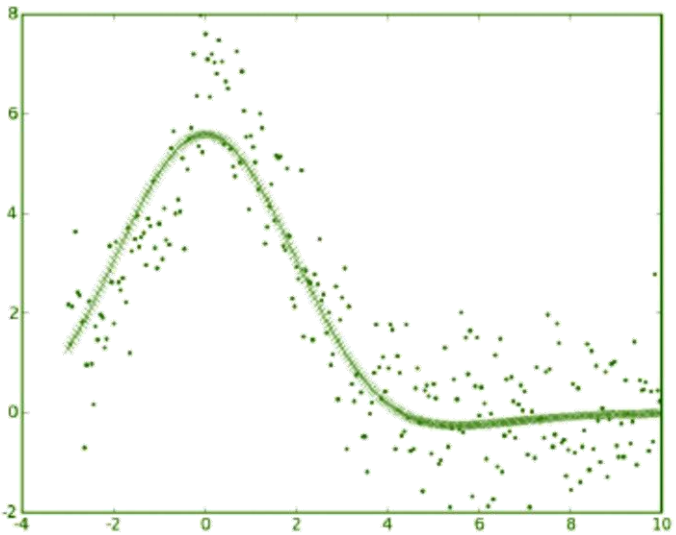


Fig 2.10.3 Using linear least-squares to fit parameters for two Gaussians produces the line from the noisy datapoints plotted as circles.

2.10.4 Smoothing Splines

- Almost all of the data that we ever see will be noisy, and insisting that the data goes through the knotpoints therefore overfits: imagine that the line in Figure 2.10.3 went through each datapoint.
- As we try to make the spline model match the data more and more accurately, we will add further knotpoints, which leads to further overfitting.
- We can deal with this by using regularisation. This is a very important idea in optimization , it means adding an extra constraint that makes the problem simpler to solve by providing some way to choose from amongst the set of possible solutions.
- The most common regulariser that is used for splines is to make the spline model as
- ‘smooth’ as possible, where the smoothness is measured by computing the second derivative of the curve at each point, squaring it so that it is always positive, and integrating it alongthe curve.
- A parameter λ that describes the tradeoff between the two parts. We regain the interpolating spline for λ = 0, whereas for λ □□1 we get the least-squares straight line.
- This type of spline is known as a smoothing spline. The cubic smoothing spline is often used. While there are automated methods of choosing λ , it is more normal to use cross-validation to find a value that seems to work well. The form of the optimisation is now:

$$E(y, f(x), \lambda) = \sum_{i=1}^{N} (y_i - f(x_i))^2 + \lambda \int \left(\frac{d^2 f}{dt^2} \right)^2 dt.$$

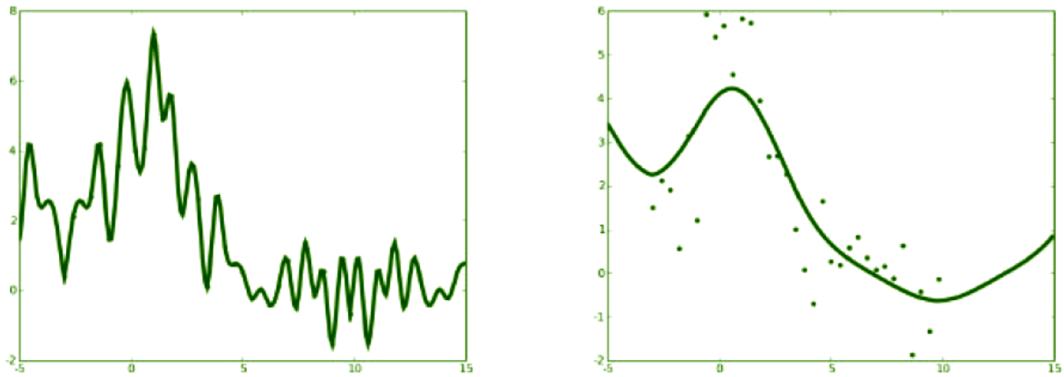


FIGURE 2.10.4.1 B-spline fitting of the data shown in Figure 2.10.4.2 with left: = 0 and right: = 100.

5.3.5 Higher Dimensions

- Everything that we have done so far is aimed at one spatial dimension and all of our effort has gone into the cubic spline.
- However, it is not very clear what to do with higher dimensional data. One common thing that is done is to take a set of independent basis functions in each different coordinate (x, y, and z in 3D) and then to combine them in all possible combinations $(\Phi_{xi}(x)\Phi_{yj}(y)\Phi_{zk}(z))$
- This is known as the tensor product basis, and suffers from the curse of dimensionality very quickly, but works well in 2D and 3D.
- It leads to a penalty term that consists of:

$$\int\int_{\mathbb{R}^2} \left(\frac{\partial^2 f}{\partial x_1^2}\right)^2 + 2\left(\frac{\partial^2 f}{\partial x_1 \partial x_2}\right)^2 + \left(\frac{\partial^2 f}{\partial x_2^2}\right)^2 dx_1 dx_2.$$

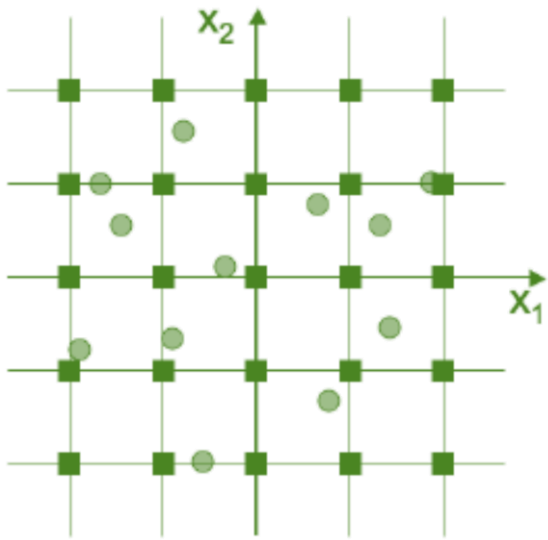


Fig 5.3.5 In 2D the knotpoints (dark green squares) can be used to interpolate other points (light green circles) in each dimension individually.

2.11 Support Vector Machines.

Introduction:

Binary classification can be viewed as the task of separating classes in feature space: All three of the lines that are drawn separate out the two classes, so in some sense they are ‘correct’, and the Perceptron would stop its training if it reached any one of them.

2.11.1 OPTIMAL SEPARATION

- Figure 2.11.1 shows a simple classification problem with three different possible linear classification lines.
- All three of the lines that are drawn separate out the two classes, so in some sense they are ‘correct’, and the Perceptron would stop its training if it reached any one of them.
- we prefer a line that runs through the middle of the separation between the datapoints from the two classes, staying approximately equidistant from the data in both classes. Of course, you might have asked what criteria you were meant to pick a line based on, and why one of the lines should be any better than the others.

- To answer that, we are going to try to define why the line that runs halfway between the two sets of datapoints is better, and then work out some way to quantify that so we can identify the ‘optimal’ line, that is, the best line according to our criteria. The data that we have used to identify the classification line is our training data.
- If we pick the lines shown in the left or right graphs of Figure 2.11.1, then there is a chance that a datapoint from one class will be on the wrong side of the line, just because we have put the line tight up against some of the datapoints we have seen in the training set. The line in the middle picture doesn’t have this problem;

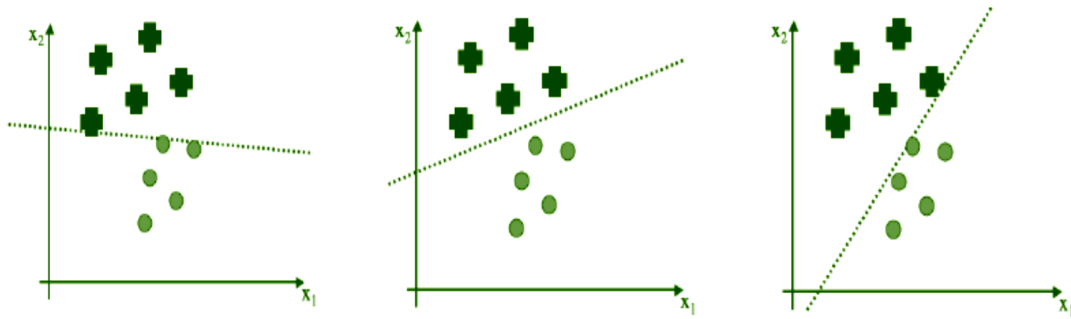
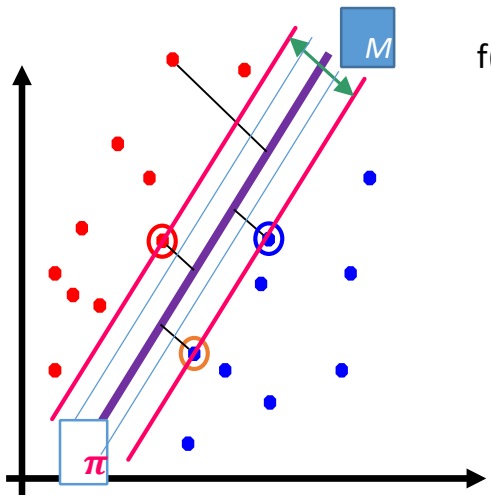
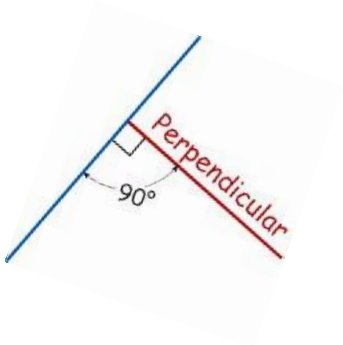


FIGURE 2.11.1 Three different classification lines. Is there any reason why one is better than the others?

- Keep drawing hyper plane parallel to $\boldsymbol{\pi}$
- Stop when it reaches the data point.
- The datapoints in each class that lie closest to the classification line is called as **Support Vector**
- Distance between support vector or the largest radious is known as the margin, labelled M.
- Two considerartions are
    i. The margin should be as large as possible
    ii. The support vectors are the most useful datapoints because they are the ones that we might get wrong.
- An interesting feature of these algorithms: **after training we can throw away all of the data except for the support vectors**, and use them for classification, which is a useful saving in data storage.



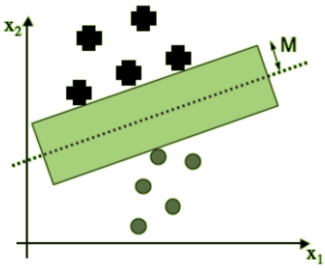
$f(\mathbf{x}_i) = \text{sign}(\mathbf{w}^T\mathbf{x}_i + b)$



- Weight vector → w (Normal to hyperplane)
- Input vector → x
- y = w· x+b , (with b→ bias weight)

- In general We use the classifier line by saying that any x value that gives a positive value for $\mathbf{w} \cdot \mathbf{x} + \mathbf{b}$ is above the line, and so is an example of the '+' class, while any x that gives a negative value is in the 'o' class.
- The value of $\mathbf{w} \cdot \mathbf{x} + \mathbf{b}$ is positive or negative, we also check whether the absolute value is less than our margin M, which would put it inside the green in Figure.

w · x is the inner or scalar product $\mathbf{w} \cdot \mathbf{x} = \sum_i w_i x_i.$



- rewrite as $\mathbf{w}^T\mathbf{x}_i$

- For a given margin value M we can say that any point x where

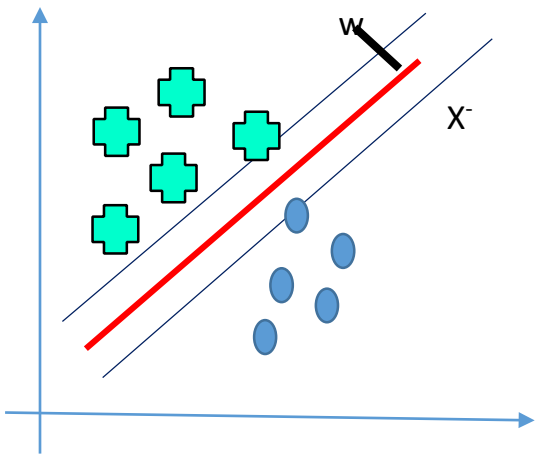
  $w^T x + b \geq M$ → is a plus
- Any point where $w^T x + b \leq -M$ → is a circle.
- The actual separating hyperplane is specified by $w^T x + b = 0$.

- Now suppose that we pick a point x^+ that lies on the '+' class boundary line, so that $w^T x^+ = M$. This is a support vector.
- If we want to find the closest point that lies on the boundary line for the 'o' class, then we travel perpendicular to the '+' boundary line until we hit the 'o' boundary line. The point that we hit is the closest point, and we'll call it x^-.
- We know that , 'w ' is perpendicular to the classifier line, then it is obviously perpendicular to the '+' and 'o' boundary lines too, so the direction we travelled from x^+ to $x-$ is along w.

- Plus-plane = { **x**: **w**. **x**+ b = +1 }
- Minus-plane = { **x**: **w**. **x**+ b = -1 }
- The vector **w** is perpendicular to the Plus Plane
- Let $\mathbf{x}^-$ be any point on the minus plane

$$w^\top x_n + b \geq 1 \text{ for } y_n = +1$$
$$w^\top x_n + b \leq -1 \text{ for } y_n = -1$$

$$y_n(w^\top x_n + b) \geq 1$$



$$W.x_1 + b = 1$$

$$W.x_2 + b = -1$$

$$W.(x_1 - x_2) = 2$$

$$d = \frac{2}{||W||}$$

where $||W|| = \sqrt{w_1^2 + w_2^2 + ...w_m^2}$ in an m-dimension space.

X+

Let $\mathbf{x}^+$ be the closest plus-plane-point to $\mathbf{x}^-$.

Total margin (2γ)

$$(\hat{\boldsymbol{w}}, \hat{b}) = \arg\max_{\boldsymbol{w},b} \frac{2}{||\boldsymbol{w}||}, \quad \text{s.t.} \quad y_n(\boldsymbol{w}^\top \boldsymbol{x}_n + b) \geq 1$$

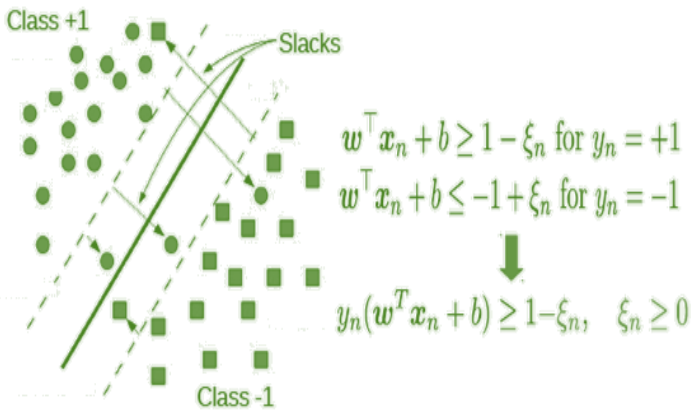
(equivalent to)

$$(\hat{\boldsymbol{w}}, \hat{b}) = \arg\min_{\boldsymbol{w},b} \frac{||\boldsymbol{w}||^2}{2}, \quad \text{s.t.} \quad y_n(\boldsymbol{w}^\top \boldsymbol{x}_n + b) \geq 1$$

Hard-margin SVM
("hard" = want all points to satisfy the margin constraint)

Maximum-Margin Hyperplane with Slacks

- Still want a max-margin hyperplane but want to relax the hard constraint $y_n(\boldsymbol{w}^\top \boldsymbol{x}_n + b) \geq 1$
- Let's allow every point $\boldsymbol{x}_n$ to "slack the constraint" by a distance $\xi_n \geq 0$



- Points with $\xi_n \geq 0$ will be either in the margin region or totally on the wrong side
- **New Objective:** Maximize the margin while keeping the sum of slacks $\sum_{n=1}^{N} \xi_n$ small
- **Note:** Can also think of the sum of slacks as the total training error

PART - A

1. What is multilayer perceptron (MLP)? (CO2,K1)

A multilayer perceptron (MLP) is a class of feed forward artificial neural network. The MLP consists of three or more layers (an input and an output layer with one or more hidden layers) of nonlinearly-activating nodes making it a deep neural network. Since MLPs are fully connected, each node in one layer connects with a certain weight wijto every node in the following layer.

2. What are the 2 steps of MLP training? (CO2,K1) Training the MLP consists of two parts:

- Working out what the outputs are for the given inputs and the current weights • Updating the weights according to the error, which is a function of the difference between the outputs and the targets.

3. What is biases in MLP? (CO2,K1)

Just like in the perceptron case, we need to include a bias input to each neuron. We do this in the same way, by having extra input that is permanently set to -1, and adjusting the weights to each neuron as part of the training. Thus each neuron in the network has 1 extra input, with fixed value.

4. What do you mean by going forwards and backwards through the network? (CO2,K1)

Training the MLP consists of two parts:

- Working out what the outputs are for the given inputs and the current weights • Updating the weights according to the error, which is a function of the difference between the outputs and the targets. These are generally known as going forwards and backwards through the network.

5. What do you mean by back propagation? (CO2,K1)

Back propagation is a method used in artificial neural networks to calculate the error contribution of each neuron after a batch of data (e.g. in image recognition, multiple images) is processed. This is used by an enveloping optimization algorithm to adjust the weight of each neuron, completing the learning process for that case.

PART - A

6. What is backward propagation of errors? (CO2,K1)

Technically it calculates the gradient of the loss function. It is commonly used in the gradient descent optimization algorithm. It is also called backward propagation of errors, because the error is calculated at the output and distributed back through the network layers.

7. What is activation function? (CO2,K1)

The activation function of a node defines the output of that node given an input or set of inputs. A standard computer chip circuit can be seen as a digital network of activation functions that can be "ON" (1) or "OFF" (0), depending on input. This is similar to the behavior of the linear perceptron

8. List the various activation function. (CO2,K1)

- Step function
- Linear function
- Sigmoid function
- Tanh function

9. Write about Sigmoid Activation function. (CO2,K1)

It is an activation function of form f(x) = 1 / 1 + exp (-x). Its Range is between 0 and 1. It is an S-shaped curve. It is easy to understand.

10. What are the limitations of sigmoid function? (CO2,K1)

- Vanishing gradient problem
- Secondly, its output isn't zero centered. It makes the gradient updates go too far in different directions. 0 < output < 1, and it makes optimization harder.
- Sigmoid saturate and kill gradients.
- Sigmoid have slow convergence.

11. What are the two ways to implement BP learning? (CO2,K1)

- Sequential mode
- Batch mode

PART - A

12. What is sequential mode? (CO2,K1)

In this mode of BP learning, adjustments are made to the free parameters of the network on an example-by-example basis. The sequential mode is also suited for pattern classification problem. Sequential mode also referred as on-line mode or stochastic mode.

13. What is batch mode? (CO2,K1)

In this mode of BP learning, adjustments are made to the free parameters of the network on an epoch-by-epoch basis, where each epoch consists of the entire set of training examples.

This batch mode is best suited for nonlinear regression.

14. How many weights are there in MLP with one hidden layer? (CO2,K2)

For the MLP with one hidden layer there are (m+1) x n + (n x1) x p weights where m, n, p are the number of nodes in the input, hidden and output layers respectively. The extra +1 come from the bias node, which also have adjustable weights.

15. What are the various parameters that the hidden unit depends on? (CO2,K1)

The best number of hidden units depends in a complex way on many factors, including:

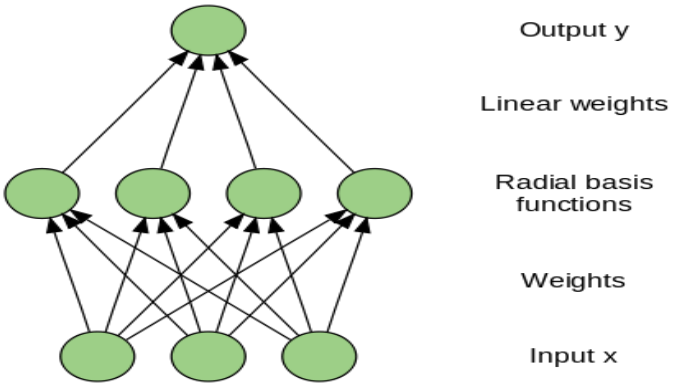
- The number of training patterns
- The numbers of input and output units
- The amount of noise in the training data
- The complexity of the function or classification to be learned
- The type of hidden unit activation function
- The training algorithm

16. What is a radial basis function network? (CO2,K1)

A Radial basis function network is an artificial neural network that uses radial basis functions as activation functions. The output of the network is a linear combination of radial basis functions of the inputs and neuron parameters. Radial basis function networks have many uses, including function approximation, time series prediction, classification, and system control.

PART - A

17. Draw the architecture of a radial basis function network. (CO2,K1)



18. What is SVM? (CO2,K1)

In machine learning, support vector machines (SVMs, also support vector networks) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier

19. Why can't we do it without activating the input signal? (CO2,K1)

If we do not apply activation function then the output signal would simply be a simple linear function. A linear function is just a polynomial of one degree. Now, a linear equation is easy to solve but they are limited in their complexity and have less power to learn complex functional mappings from data. A Neural Network without Activation function would simply be a linear regression Model, which has limited power and does not performs good most of the times.

20. Why do we need Non-Linearities? (CO2,K1)

- Non-linear functions are those which have degree more than one and they have a curvature when we plot a Non-Linear function. Now we need a Neural Network Model to learn and represent almost anything and any arbitrary complex function which maps inputs to outputs. Neural-Networks are considered Universal Function Approximators. It means that they can compute and learn any function at all. Almost

any process we can think of can be represented as a functional computation in Neural Networks.

- Hence it all comes down to this, we need to apply a Activation function f(x) so as to make the network more powerful and add ability to it to learn something complex and complicated form data and represent non-linear complex arbitrary functional mappings between inputs and outputs. Hence using a nonlinear Activation we are able to generate non-linear mappings from inputs to outputs.

PART - B

1. Explain the multilayer perceptron algorithm in detail. **(CO2,K2)**
2. Explain the following with respect to MLP. **(CO2,K2)**
    a) Initializing the weights
    b) Different output activation functions
3. Explain the following in detail **(CO2,K2)**
    a) Sequential and batch training
    b) Local minima
4. How regression problem can be solved using MLP? Explain with example. **(CO2,K2)**
5. Explain classification with MLP in detail. **(CO2,K2)**
6. Explain time series prediction problem with MLP in detail. **(CO2,K2)**
7. Explain Radial Basis Function (RBF) network & training RBFN in detail.
8. Explain the following in detail **(CO2,K2)**
    a) The curse of dimensionality
    b) Interpolation and basis function.

PART - C

1. Explain how the XOR problem can be solved by an MLP? **(CO2,K3)**
2. Explain support vector machine in detail. **(CO2,K3)**