EEEN4/60151 Machine Learning LABORATORY

Neural Networks for Pattern Recognition

Objectives: Design and implement Perceptron, Single Layer Perceptrons and Multilayer Perceptrons (MLP) for learning to classify handwritten digits
(Optional) implement Convolutional Neural Networks for image classification
(Optional) Implement Self-Organising Map for object mapping and visualisation

Equipment: PC and any programming language such as Matlab, Python, or C/C++/C#

1. Single Perceptron (~2 hours, 30%)

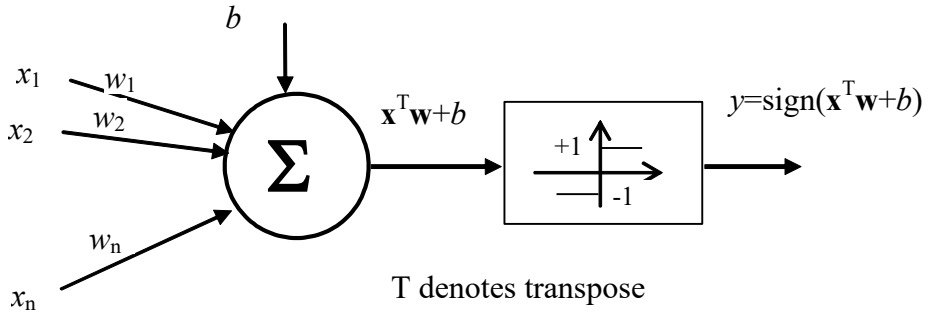
Introduction

Handwritten digit recognition is a typical example of machine learning and vision applications. This first experiment starts with the simplest task: classifying just two types of simulated handwritten digits, "0" and "1", with a single perceptron. Assume that some handwritten digits are digitalised into binary images of 5x5 pixels shown in the next page. There are six patterns for each of two classes, "1" or "0". First, express each of these image patterns, row by row, as a (binary) column vector (e.g. white pixel as 0 and black pixel as 1). For example, the 1st "1" image would be

$[0\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ 0\ 0]^T$, T denotes the transpose operator.

Then, choose four patterns (either manually or automatically) from each class as **training patterns** and leave the remaining patterns as **test patterns**. Write a program to construct a perceptron, shown below, and feed the training patterns (one by one) as input so the perceptron learns to classify them to a pre-specified classification rate (in this example, 100% can be expected). After training has reached a required performance, apply the trained network to the test patterns. Note: 100% classification rate on the test set may not be achieved on the test set.

Perceptron structure:



Perceptron training algorithm:

Step 1: Initialise the weights $\mathbf{w}=[w_1, w_2, ..w_n]^T$ and bias b to **small random numbers**.

Step 2: Present the network with an input $\mathbf{x}(k)$ (i.e. a column vector of 25 binary values of a randomly drawn training pattern) and its corresponding **desired output** $d(k)$ (1 for digit "1"s or –1 for digit "0"s). $1 \le k \le N$, N is the total number of training patterns. $\{d(k)\}$ can be pre-generated with the training set.

Step 3: Calculate the network output for this input: $y(k)=\text{sign}[\mathbf{x}(k)^T\mathbf{w}+b]$.

Step 4: Update the weights and bias according to:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha[d(k)-y(k)]\mathbf{x}(k)$$
$$b_{t+1} = b_t + \alpha[d(k)-y(k)]$$

where α is the learning rate, $0<\alpha<1$, e.g. $\alpha=0.1$ or 0.01, and t is the iteration number.

Step 5: After each training epoch (i.e. when every training pattern is presented once), if the total error is not within a permitted range (say 0 in this case), go back to step 2, otherwise stop.

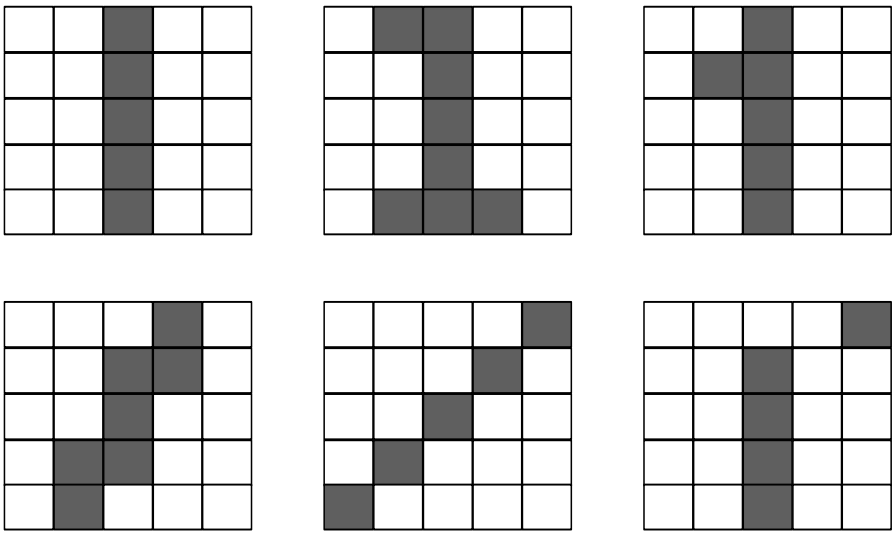
Procedure

You can randomly choose the training set either by hand (i.e. set aside four patterns from each class) or by a program (i.e. built in your program at the start to randomly pick four patterns from each class); implement the perceptron given in the algorithm above, and then train the perceptron with the training set so that it will output 1 if the input is a "1" digit or -1 if it's a "0" digit. After training, you test the trained perceptron on the test set (remaining patterns). The performance on the test set can be measured by percentage error rate (misclassification rate).

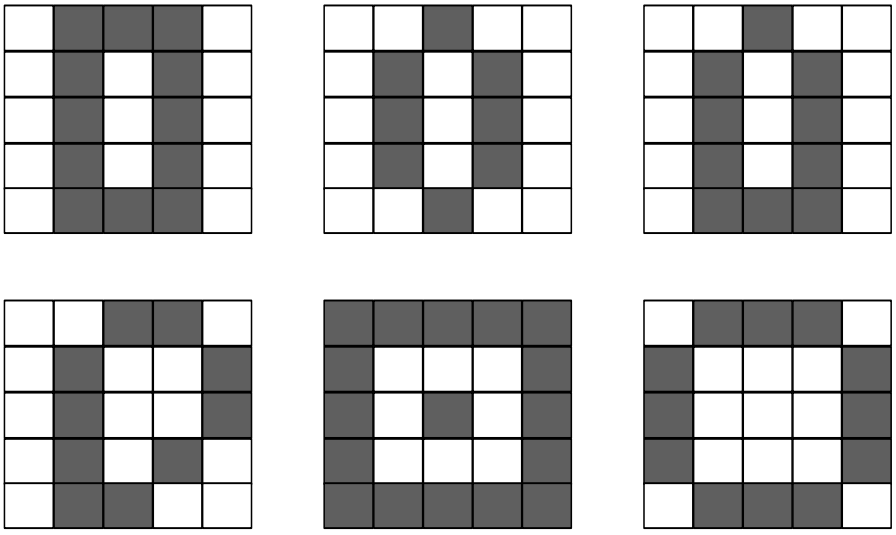
The training patterns can be presented in an orderly fashion or randomly, the latter is preferred in practice. You may explore this further and compare the end results. You should also explore the choices of training patterns and learning rate, and their influence on the performance.

You may repeat the simulation many times, say 10 (each time with random initial weights and randomly selected training and test patterns). What is the average performance? Any other observations on performance variations and choices of training patterns?

"1" patterns:

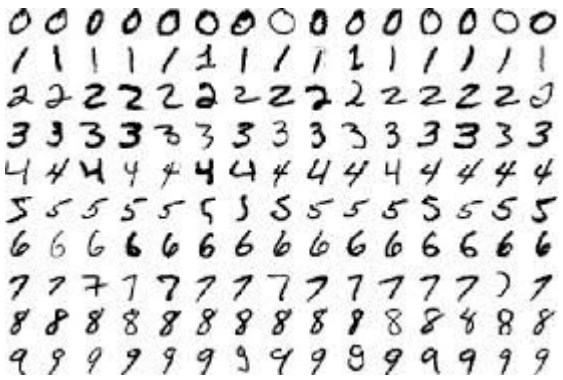


"0" patterns:



2. Single Layer Perceptrons (~4 hours, 40%)

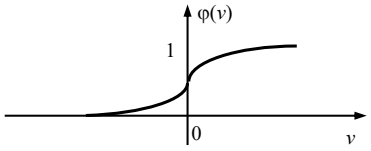
MNIST dataset (http://yann.lecun.com/exdb/mnist/) is a large collection of real handwritten digits, widely used in computer vision and machine learning. It contains 60,000 training images (8-bit grey-level, 28x28 pixels) of 10 classes of digits ("0", "1", … "9"). Each class has 6000 samples. Examples are shown below. Along with the images, there is also matching label for each training sample. The dataset also has a test set and respective labels of 1000 images. You may download its original format or Matlab .mat format (available from Blackboard. You load it using "d=load('mnist.mat'); then d.trainX is a 60000×784 matrix, i.e. 60,000 training patterns in 784 dimension row vectors, d.trainY contains their lables (0, 1, …9), d.testX contains test patterns, and d.testY labels of test patterns). You will need to normalise all images by dividing them by the maximum grey value 255. Each image can be expressed as a 784-dimensional column vector (28×28 =784). Corresponding label is 0, 1, …, or 9.



Extend the single perceptron to a single layer of 10 perceptrons, one for each class of digits. All 10 perceptrons take the same input, a vector of 784 dimensions. You will need to use the sigmoid activation function instead of the sign function, $y=\varphi(v)$, $v=\mathbf{x}^T\mathbf{w}+b$ for each perceptron, shown below and use the following weight updating equations (You should be able to derive these equations)

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \alpha[d(t) - y(t)]y(t)[1 - y(t)]\mathbf{x}(t)$$
$$b(t+1) = b(t) + \alpha[d(t) - y(t)]y(t)[1 - y(t)]$$

$$\varphi(v) = \frac{1}{1+e^{-v}}$$



Sigmoid activation function.

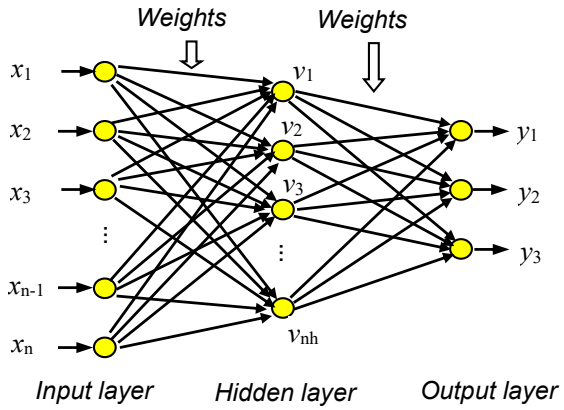
Training is to apply the weight and bias updating equations to each of the perceptrons on an input. You may use 10% of the training set to start with (e.g. first 6000 training images as the training set is randomised), and then expand to the entire training set. Target or desired outputs for 10 perceptrons should be formed based on the label of a training image. For example, if a training image is of digit '0', then its target output vector is $[1, 0, 0, 0, 0, 0, 0, 0, 0, 0]^T$. For a training image of digit '6', the target output is $[0, 0, 0, 0, 0, 0, 1, 0, 0, 0]^T$. This is the so-called 1-of-c coding scheme. From these you have the corresponding target output value for each perceptron.

You can set a fixed number of iterations or epochs (for each epoch all training patterns are used once, usually in random order) or you may also set a pre-specified training error rate as the stopping criterion, but it may never be satisfied if it is not set appropriately. After training, apply the network to the test dataset. For an input image, the maximum output among the 10 perceptrons is regarded as the predicted class of the input. Report performance in terms accuracy or error rate for both training set and test set. Discuss influence of the learning rate to the performance.

3. Multilayer Perceptron (MLP) (~4 hours, 30%)

Implement MLP to the MNIST data. Details of MLP can be found in the lecture notes, also below

MLP structure:



Activation function:

$$\varphi(v) = \frac{1}{1+e^{-v}}$$

Back-propagation training algorithm:

Step 1: *Initialisation*. Set all weights and nodes' biases to small random numbers.
Step 2: *Presentation of training examples*. input $\mathbf{x}=[x_1, x_2, \ldots x_n]^T$ and desired output $\mathbf{d}=[d_1, d_2, .. d_m]^T$.
Step 3: *Forward computation*. Calculate the outputs of the hidden and output layer,

$$y_k = \varphi_k^o\left(\sum_{j=1}^{n_h} w_{jk}^o v_j + b_k^o\right) \qquad v_j = \varphi_j^h\left(\sum_{i=1}^{n} w_{ij}^h x_i + b_j^h\right)$$

Step 4: *Backward computation and updating weights*. Compute the error terms,

$$\delta_k^o = e_k^o(\varphi_k^o)' = e_k^o y_k(1-y_k) = y_k(1-y_k)(d_k - y_k) \qquad w_{jk}^o = w_{jk}^o + \alpha\delta_k^o v_j$$

b_k ...

$$\delta_j^h = (\varphi_j^h)' \sum_{k=1}^{m} \delta_k^o w_{jk}^o = v_j(1-v_j)\sum_{k=1}^{m} \delta_k^o w_{jk}^o \qquad w_{ij}^h = w_{ij}^h + \alpha\delta_j^h x_i$$

b_j ...

Step 5: *Iteration*. Repeat steps 3 and 4 and stop when the total error reaches the required level).

You may use 10-20 hidden nodes in the hidden layer, while the output layer should have 10 nodes representing 10 classes. After training, apply the MLP to the test dataset. Report performance in terms accuracy or error rate for both training set and test set.

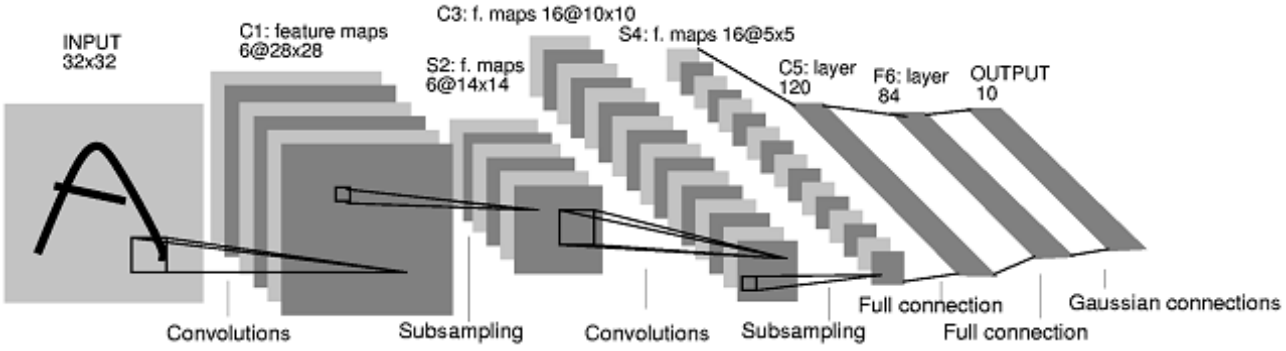
You may like to explore and discuss the influence of the following parameters,

- learning rate
- number of hidden nodes
- stopping rule

Optional Parts (5-10 hours, extra 10% each with overall total capped at 100%):

4. LeNet-5 for Handwritten Digit Classification (on MNIST dataset)

You may proceed to implementing LeNet-5 (or a similar CNN) on MNIST dataset, either by coding it yourself or using relevant toolboxes.



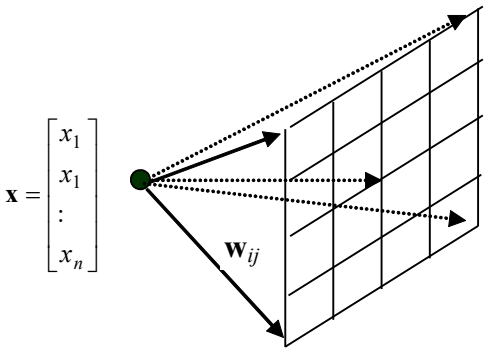
Further details of LeNet-5 can be found from MNIST website, http://yann.lecun.com/exdb/mnist/

Again, you should explore how network hyperparameters affect its performance. You will need to use a validation set (a small portion of training set) to monitor the training and to prevent overfitting.

5. Self-Organising Map (SOM)

Pattern recognition often requires clustering high dimensional objects/data and visualise them on a low dimensional (say 2D) plane. This exercise is to cluster a set of animals using SOM. The 13 attributes or properties of 16 animals are shown in the table on next page. Use a 10x10 SOM to train and map these animal properties according to the SOM algorithm given below. After learning, you can map the animals back onto the SOM, i.e. placing an animal (data item) to the grid position of the neuron that wins (i.e. closest to) this item.

SOM structure:



SOM algorithm (for m×m SOM)

Step 1: At t=0, initialise the weights $\begin{Bmatrix} \mathbf{w}_{11} & \mathbf{w}_{12} & \mathbf{w}_{1m} \\ \mathbf{w}_{21} & \mathbf{w}_{22} & \mathbf{w}_{2m} \\ & & \\ \mathbf{w}_{m1} & \mathbf{w}_{m2} & \mathbf{w}_{mm} \end{Bmatrix}$ to small random numbers. $\mathbf{w}_{ij}$ is the weigh vector associated to neuron at 2D index ij and is a column vector of the same dimension, n, of the input.

Step 2: At t, present the network with an input $\mathbf{x}(t)$, a column vector, by taking a randomly chosen row from the table (of next page) and converting it to a column vector.

Step 3: Calculate the distances between the input and all the weight vectors,

$$d_{ij}(t) = \sqrt{[\mathbf{x}(t) - \mathbf{w}_{ij}(t)]^T [\mathbf{x}(t) - \mathbf{w}_{ij}(t)]}, \quad i,j=1, 2, \ldots m$$

Step 4: Choose a winner (winning neuron) according the smallest distance, say of index uv.

Step 5: Update the weight vectors of the winner and its neighbouring neurons according to:

$$\mathbf{w}_{ij}(t+1)=\mathbf{w}_{ij}(t)+\alpha(t)\eta[\mathbf{x}(t)-\mathbf{w}_{ij}(t)], \qquad i,j \in \underline{\textit{neighbourhood}} \text{ of } uv,$$

where $\alpha(t)$ is the learning rate (e.g. set to$100/(200+t)$. $\eta = \exp(-\frac{(i-u)^2+(j-v)^2}{2\sigma^2})$ is neighbourhood function. Set σ to 2, or gradually decrease it from 3 to 1 with time.

Step 5: $t=t+1$, If t<10,000, go back to step 2, otherwise stop.

Procedure

Covert the animals to 13-dimensional (binary) vectors according to their attributes. Train a 10x10 SOM on these vectors using the above algorithm. The training patterns should be presented in random order. After training (say 10, 20, or 50 passes), map the animals back onto the map. Do you see that similar animals are projected together or close by on the map? What applications can this technique be used for? Can it be used for classification?

Table: Animal names and their attributes

	is			has						likes to			
	small	medium	big	2legs	4legs	hair	hooves	mane	feather	hunt	run	fly	swim
Dove	1	0	0	1	0	0	0	0	1	0	0	1	0
Hen	1	0	0	1	0	0	0	0	1	0	0	0	0
Duck	1	0	0	1	0	0	0	0	1	0	0	1	1
Goose	1	0	0	1	0	0	0	0	1	0	0	1	1
Owl	1	0	0	1	0	0	0	0	1	1	0	1	0
Hawk	1	0	0	1	0	0	0	0	1	1	0	1	0
Eagle	0	1	0	1	0	0	0	0	1	1	0	1	0
Fox	0	1	0	0	1	1	0	0	0	1	0	0	0
Dog	0	1	0	0	1	1	0	0	0	0	1	0	0
Wolf	0	1	0	0	1	1	0	1	0	1	1	0	0
Cat	1	0	0	0	1	1	0	0	0	1	0	0	0
Tiger	0	0	1	0	1	1	0	0	0	1	1	0	0
Lion	0	0	1	0	1	1	0	1	0	1	1	0	0
Horse	0	0	1	0	1	1	1	1	0	0	1	0	0
Zebra	0	0	1	0	1	1	1	1	0	0	1	0	0
Cow	0	0	1	0	1	1	1	0	0	0	0	0	0

Reporting

Produce a clear, concise and structured report (up to 8 pages in main body, optional parts and appendix excluded) that describes your understanding, implementation and testing for each of these tasks. It should also include the key results (e.g. classification rates), analysis on the results and discussion of performances of these neural network models. You may include your code in appendix.

Submit your report via Blackboard before the deadline.

Assessment (each of the tasks is assessed on the following aspects)	Marks
Completion of experiment and implementation Reporting the simulation and tests	(Out of 30)
Plotting and describing your results in graphs or tables	(Out of 20)
Analysis of the simulation, testing and evaluation	(Out of 20)
Discussion, conclusions (and code)	(Out of 15)
Presentation and structure	(Out of 15)