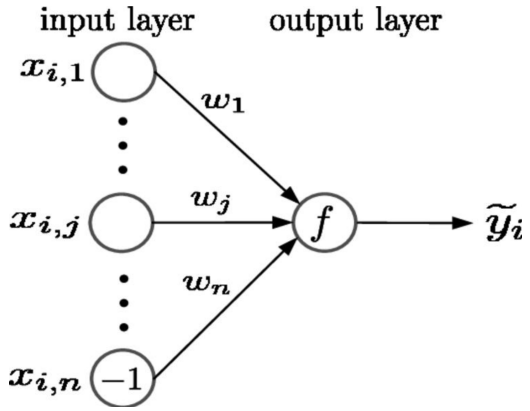
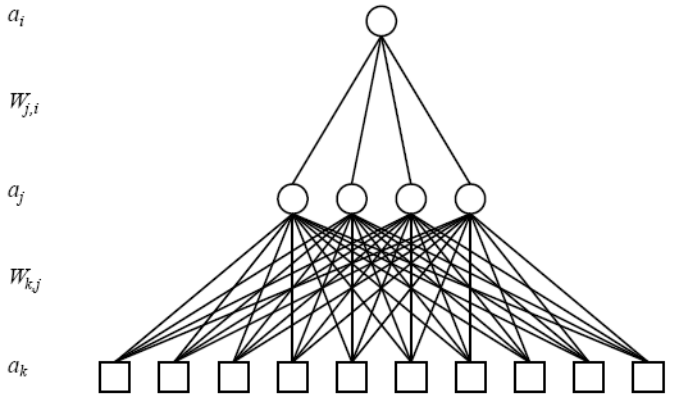
Artificial Neural Network

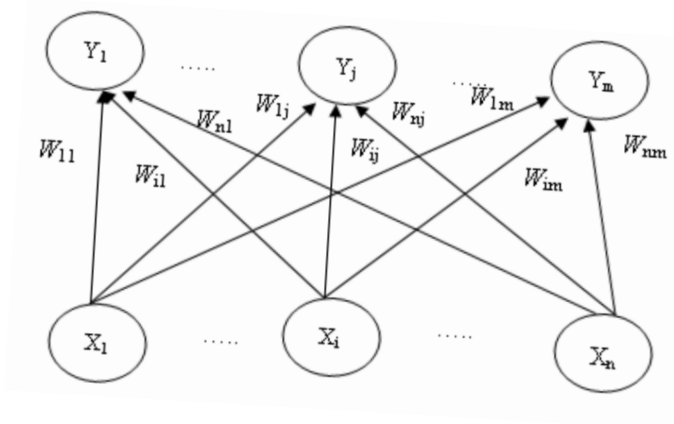
Artificial Neural Network (ANN) at Glance



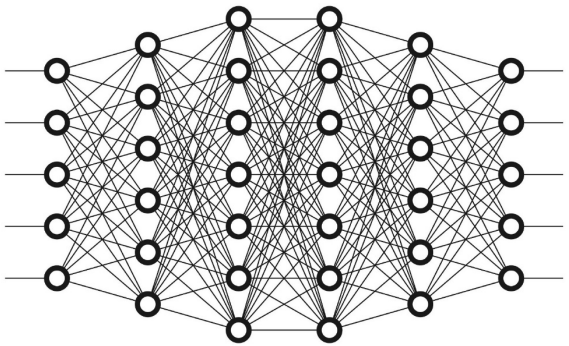
Single Layer Perceptron



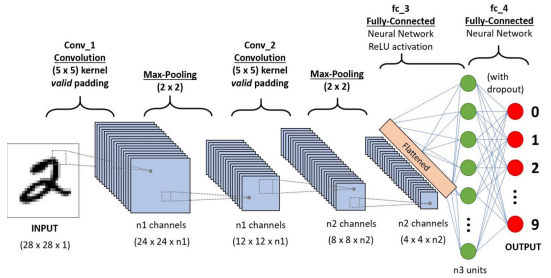
Multi Layer Perceptron



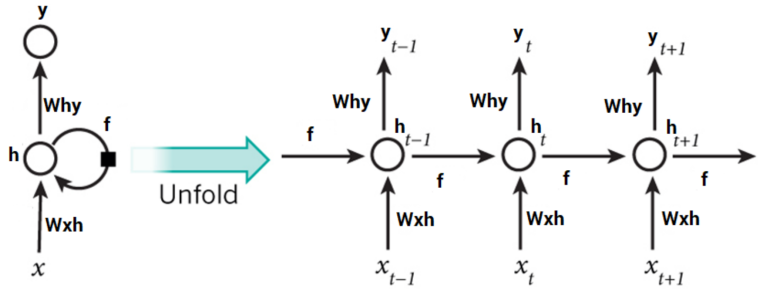
Learning Vector Quantization (LVQ)



Deep Learning

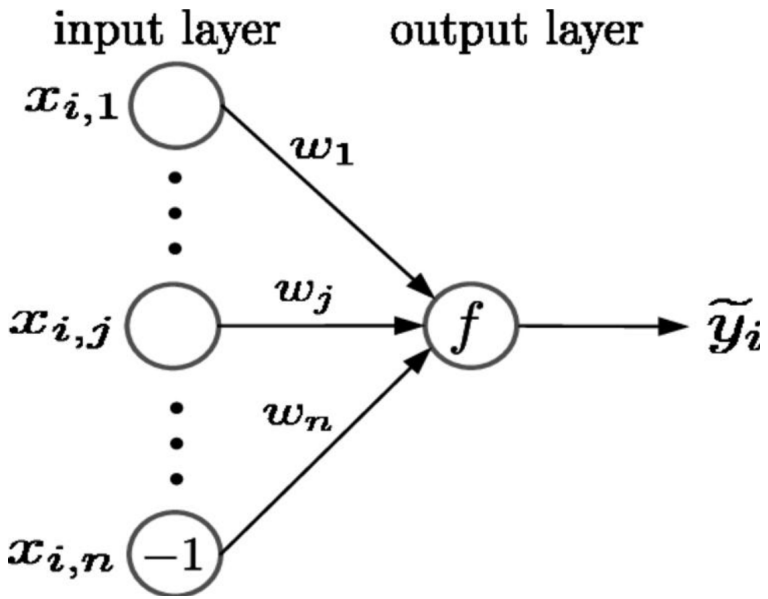


Convolutional Neural Network (CNN)



Recurrent Neural Network (RNN)

Module 1 : Single Layer Perceptron

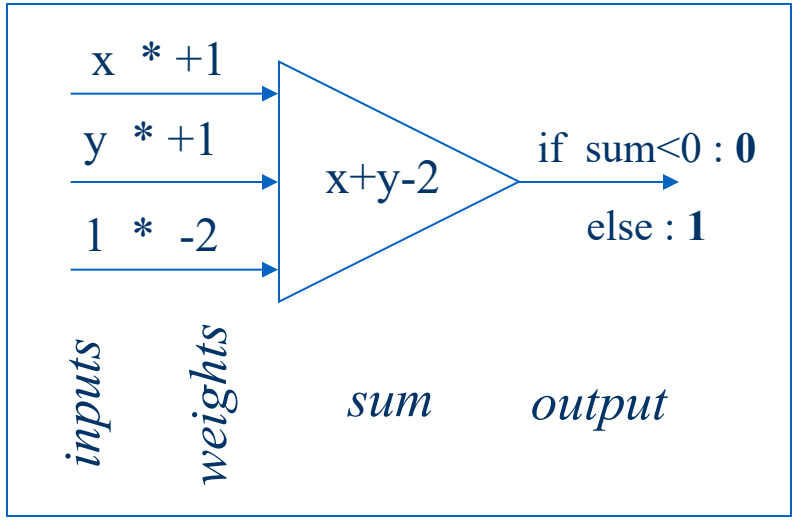


Module 1 : Perceptron

Prehistory

W.S. McCulloch & W. Pitts (1943). "A logical calculus of the ideas immanent in nervous activity", *Bulletin of Mathematical Biophysics*, 5, 115-137.

- This seminal paper pointed out that simple artificial "neurons" could be made to perform basic logical operations such as AND, OR and NOT.



Truth Table for Logical AND

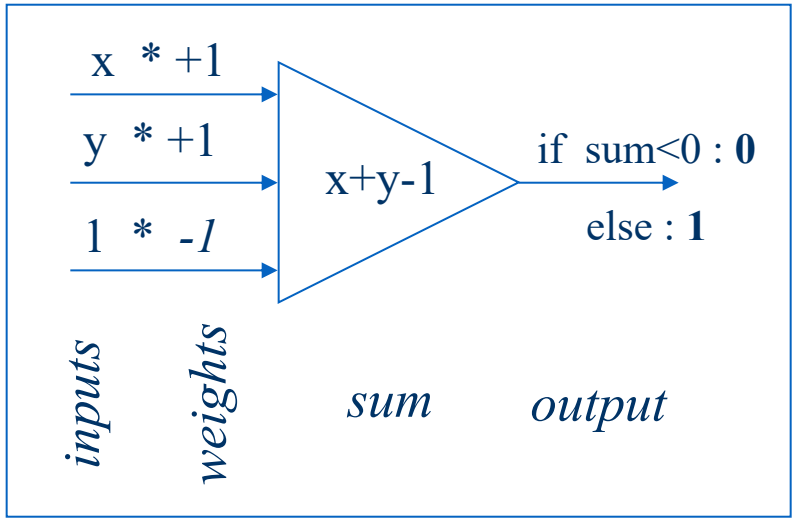
x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

inputs *output*

Nervous Systems as Logical Circuits

Groups of these "neuronal" logic gates could carry out *any* computation, even though each neuron was very limited.

- Could computers built from these simple units reproduce the computational power of biological brains?
- Were *biological* neurons performing logical operations?



Truth Table for Logical OR

x	y	x | y
0	0	0
0	1	1
1	0	1
1	1	1

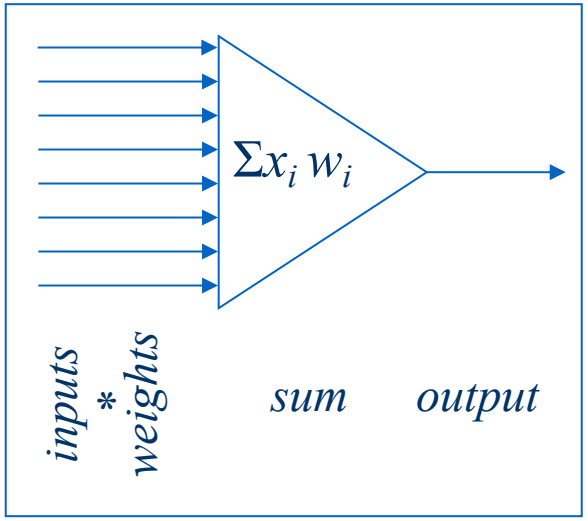
inputs *output*

The Perceptron

Frank Rosenblatt (1962). *Principles of Neurodynamics*, Spartan, New York, NY.

Subsequent progress was inspired by the invention of *learning rules* inspired by ideas from neuroscience…

Rosenblatt's *Perceptron* could automatically learn to categorise or classify input vectors into types.



It obeyed the following rule:

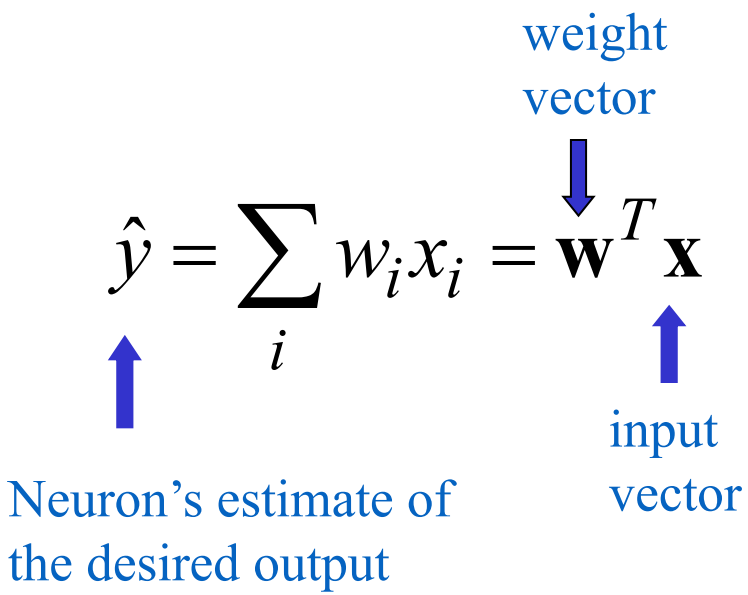
If the sum of the weighted inputs exceeds a threshold, output 1, else output -1.

1 if $\Sigma\ input_i * weight_i > threshold$

-1 if $\Sigma\ input_i * weight_i < threshold$

Linear neurons

- The neuron has a real-valued output which is a weighted sum of its inputs



- The aim of learning is to minimize the discrepancy between the desired output and the actual output
  - How de we measure the discrepancies?
  - Do we update the weights after every training case?
  - Why don't we solve it analytically?

A motivating example

- Each day you get lunch at the cafeteria.
  - Your diet consists of fish, chips, and beer.
  - You get several portions of each
- The cashier only tells you the total price of the meal
  - After several days, you should be able to figure out the price of each portion.
- Each meal price gives a linear constraint on the prices of the portions:

$$price = x_{fish} w_{fish} + x_{chips} w_{chips} + x_{beer} w_{beer}$$

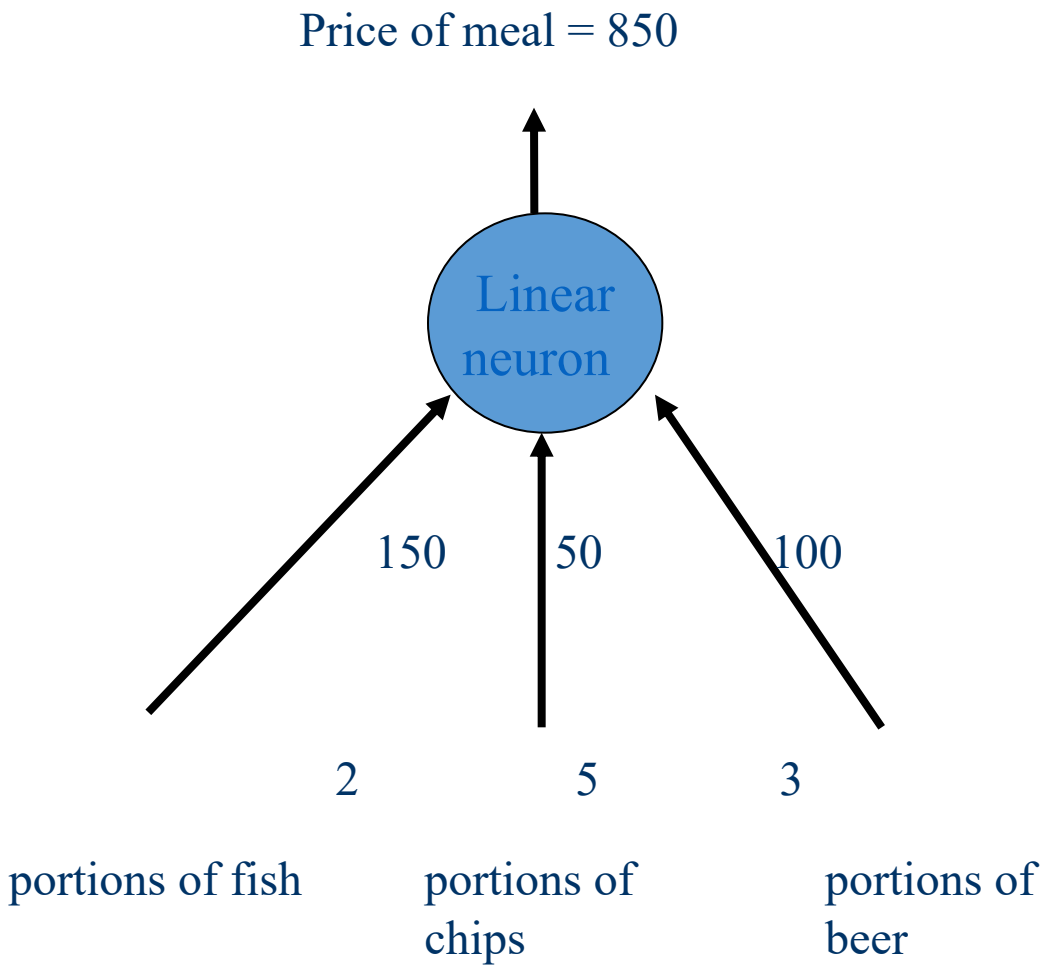
Two ways to solve the equations

- The obvious approach is just to solve a set of simultaneous linear equations, one per meal.
- But we want a method that could be implemented in a neural network.
- The prices of the portions are like the weights in of a linear neuron.

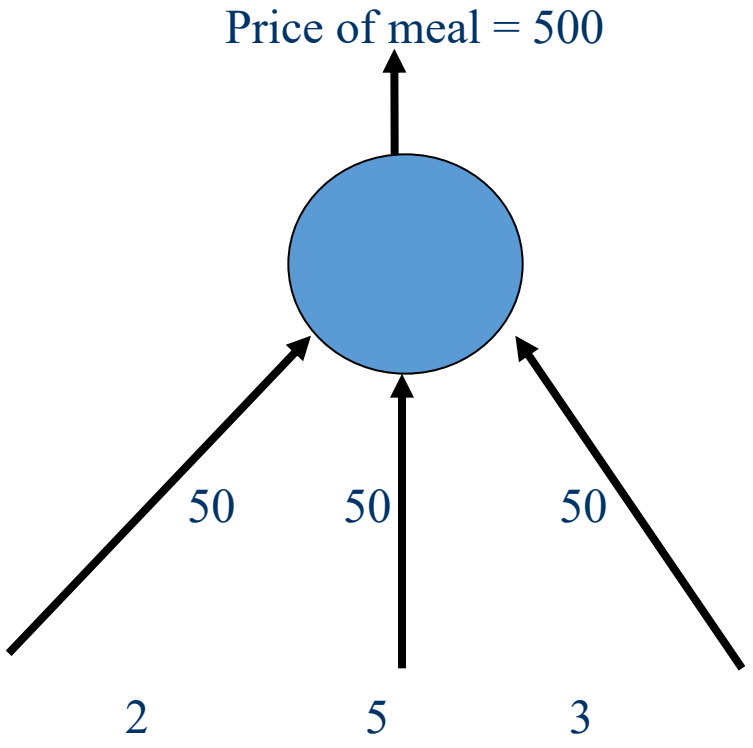
$$\mathbf{w} = (w_{fish}, w_{chips}, w_{beer})$$

- We will start with guesses for the weights and then adjust the guesses to give a better fit to the prices given by the cashier.

The cashier's brain



A model of the cashier's brain with arbitrary initial weights



- Residual error = 350
- The learning rule is:

$$\Delta w_i = \varepsilon \, x_i \, (y - \hat{y})$$

- With a learning rate ε of 1/35, the weight changes are +20, +50, +30
- This gives new weights of 70, 100, 80
- Notice that the weight for chips got worse!

Behavior of the iterative learning procedure

- Do the updates to the weights always make them get closer to their correct values? No!
- Does the online version of the learning procedure eventually get the right answer? Yes, if the learning rate gradually decreases in the appropriate way.
- How quickly do the weights converge to their correct values? It can be very slow if two input dimensions are highly correlated (e.g. ketchup and chips).
- Can the iterative procedure be generalized to much more complicated, multi-layer, non-linear nets? YES!

Deriving the delta rule

- Define the error as the squared residuals summed over all training cases:

$$E = \frac{1}{2}\sum_n (y_n - \hat{y}_n)^2$$

- Now differentiate to get error derivatives for weights

$$\frac{\partial E}{\partial w_i} = \frac{1}{2}\sum_n \frac{\partial \hat{y}_n}{\partial w_i}\frac{\partial E_n}{\partial \hat{y}_n}$$

$$= -\sum_n x_{i,n}\,(y_n - \hat{y}_n)$$

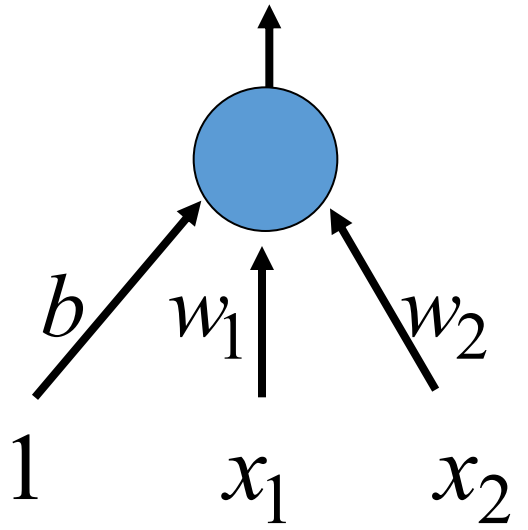
- The batch delta rule changes the weights in proportion to their error derivatives summed over all training cases

$$\Delta w_i = -\varepsilon \frac{\partial E}{\partial w_i}$$

Adding biases

- A linear neuron is a more flexible model if we include a bias.
- We can avoid having to figure out a separate learning rule for the bias by using a trick:
  - A bias is exactly equivalent to a weight on an extra input line that always has an activity of 1.

$$\hat{y} = b + \sum_i x_i w_i$$



Preprocessing the input vectors

- Instead of trying to predict the answer directly from the raw inputs we could start by extracting a layer of “features”.
  - Sensible if we already know that certain combinations of input values would be useful
  - The features are equivalent to a layer of hand-coded non-linear neurons.
- So far as the learning algorithm is concerned, the hand-coded features are the input.

Is preprocessing cheating?

- It seems like cheating if the aim to show how powerful learning is. The really hard bit is done by the preprocessing.
- Its not cheating if we learn the non-linear preprocessing.
  - This makes learning much more difficult and much more interesting..
- Its not cheating if we use a very big set of non-linear features that is task-independent.
  - Support Vector Machines make it possible to use a huge number of features without much computation or data.

Statistical and ANN Terminology

- A perceptron model with a linear transfer function is equivalent to a possibly multiple or multivariate linear regression model [Weisberg 1985; Myers 1986].
- A perceptron model with a logistic transfer function is a logistic regression model [Hosmer and Lemeshow 1989].
- A perceptron model with a threshold transfer function is a linear discriminant function [Hand 1981; McLachlan 1992; Weiss and Kulikowski 1991]. An ADALINE is a linear two-group discriminant.

Transfer functions

- Determines the output from a summation of the weighted inputs of a neuron.

$$O_j = f_j\left(\sum_i w_{ij} x_i\right)$$

- Maps any real numbers into a domain normally bounded by 0 to 1 or -1 to 1, i.e. squashing functions. Most common functions are sigmoid functions:

logistic: $f(x) = \dfrac{1}{1+e^{-x}}$

hyperbolic tangent: $f(x) = \dfrac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$

Healthcare Applications of ANNs

- Predicting/confirming myocardial infarction, heart attack, from EKG output waves
  - Physicians had a diagnostic sensitivity and specificity of 73.3% and 81.1% while ANNs performed 96.0% and 96.0%
- Identifying dementia from EEG patterns, performed better than both Z statistics and discriminant analysis; better than LDA for (91.1% vs. 71.9%) in classifying with Alzheimer disease.
- Papnet: A Pap Smear screening system by Neuromedical Syste used by US FDA
- Predict mortality risk of preterm infants, screening tool in olo etc.

Classification Applications of ANNs

- Credit Card Fraud Detection: AMEX, Mellon Bank, Eurocard Nederland
- Optical Character Recognition (OCR): Fax Software
- Cursive Handwriting Recognition: Lexicus
- Petroleum Exploration: Arco & Texaco
- Loan Assessment: Chase Manhattan for vetting commercial loans
- Bomb detection by SAIC

Time Series Applications of ANNs

- Trading systems: Citibank London (FX).
- Portfolio selection and Management: LBS Capital Management (>US$1b), Deere & Co. pension fund (US$100m).
- Forecasting weather patterns & earthquakes.
- Speech technology: verification and generation.
- Medical: Predicting heart attacks from EKGs and mental illness from EEGs.

Advantages of Using ANNs

- Works well with large sets of noisy data, in domains where experts are unavailable or there are no known rules.
- Simplicity of using it as a tool
- Universal approximator.
- Does not impose a structure on the data.
- Possible to extract rules.
- Ability to learn and adapt.
- Does not require an expert or a knowledge engineer.
- Well suited to non-linear type of problems.
- Fault tolerant

Problem with the Perceptron

- Can only learn linearly separable tasks.
- Cannot solve any ‘interesting problems’-linearly nonseparable problems e.g. exclusive-or function (XOR)-simplest nonseparable function. ☺

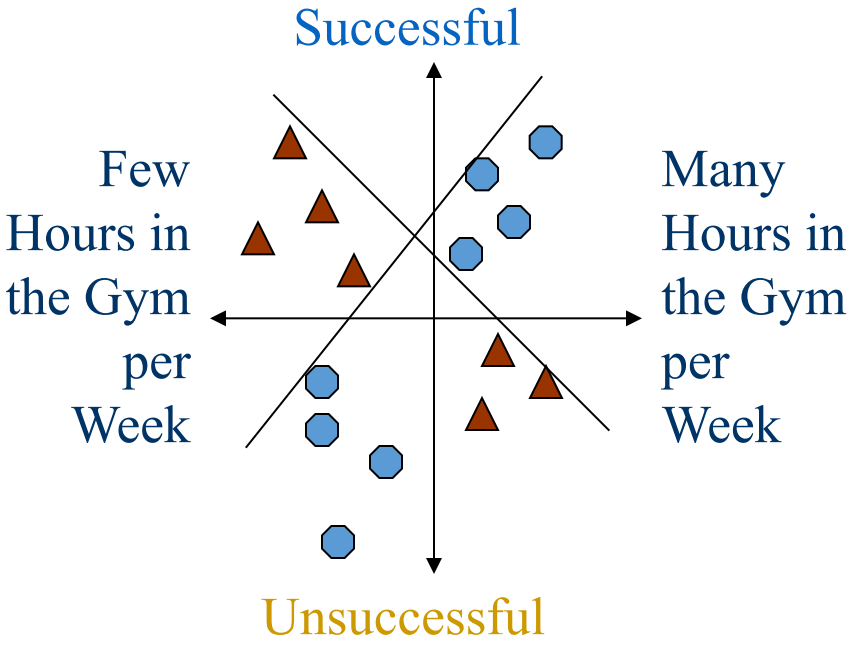
X_1	X_2	Output
0	0	0
0	1	1
1	0	1
1	1	0

The Fall of the Perceptron

Marvin Minsky & Seymour Papert (1969). *Perceptrons*, MIT Press, Cambridge, MA.

- Before long researchers had begun to discover the Perceptron's limitations.
- Unless input categories were "linearly separable", a perceptron could not learn to discriminate between them.
- Unfortunately, it appeared that many important categories were not linearly separable.
- E.g., those inputs to an XOR gate that give an output of 1 (namely 10 & 01) are not linearly separable from those that do not (00 & 11).

The Fall of the Perceptron



Footballers

Academics

…despite the simplicity of their relationship:

Academics = Successful XOR Gym

In this example, a perceptron would not be able to discriminate between the footballers and the academics…

This failure caused the majority of researchers to walk away.

Classification Using Perceptron

- See plot_sgd_comparison.py

```
# Author: Rob Zinkov <rob at zinkov dot com>
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets

from sklearn.model_selection import train_test_split
from sklearn.linear_model import SGDClassifier, Perceptron
from sklearn.linear_model import PassiveAggressiveClassifier
from sklearn.linear_model import LogisticRegression

heldout = [0.95, 0.90, 0.75, 0.50, 0.01]
rounds = 20
digits = datasets.load_digits()
X, y = digits.data, digits.target

classifiers = [
    ("SGD", SGDClassifier(max_iter=100, tol=1e-3)),
    ("ASGD", SGDClassifier(average=True, max_iter=1000, tol=1e-3)),
    ("Perceptron", Perceptron(tol=1e-3)),
    ("Passive-Aggressive I", PassiveAggressiveClassifier(loss='hinge',
                                                         C=1.0, tol=1e-4)),
    ("Passive-Aggressive II", PassiveAggressiveClassifier(loss='squared_hinge',
                                                          C=1.0, tol=1e-4)),
    ("SAG", LogisticRegression(solver='sag', tol=1e-1, C=1.e4 / X.shape[0],
                               multi_class='auto'))
]

xx = 1. - np.array(heldout)
```

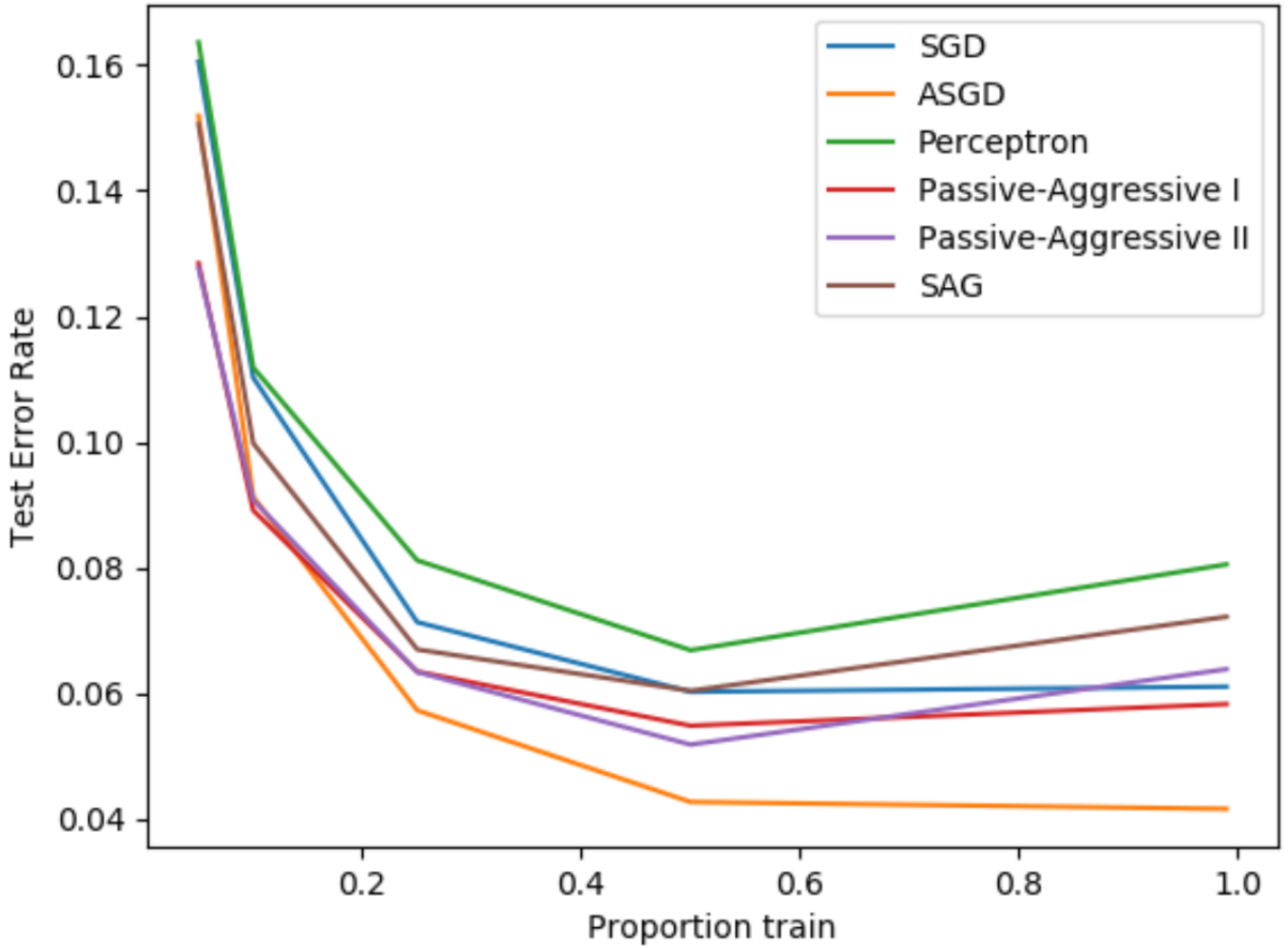
Classification Using Perceptron

- See plot_sgd_comparison.py

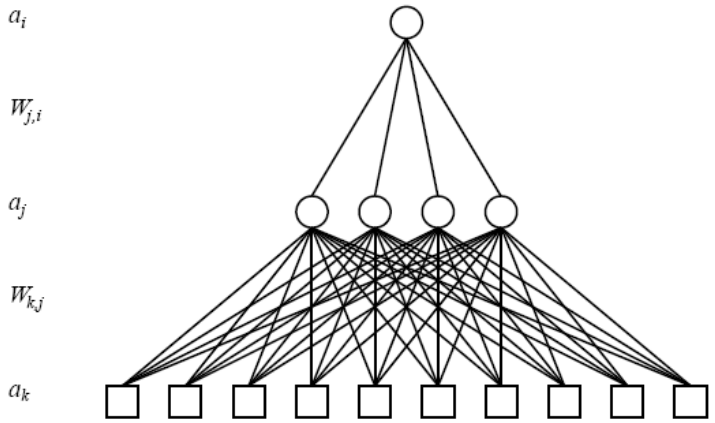
```
for name, clf in classifiers:
    print("training %s" % name)
    rng = np.random.RandomState(42)
    yy = []
    for i in heldout:
        yy_ = []
        for r in range(rounds):
            X_train, X_test, y_train, y_test = \
                train_test_split(X, y, test_size=i, random_state=rng)
            clf.fit(X_train, y_train)
            y_pred = clf.predict(X_test)
            yy_.append(1 - np.mean(y_pred == y_test))
        yy.append(np.mean(yy_))
    plt.plot(xx, yy, label=name)

plt.legend(loc="upper right")
plt.xlabel("Proportion train")
plt.ylabel("Test Error Rate")
plt.show()
```

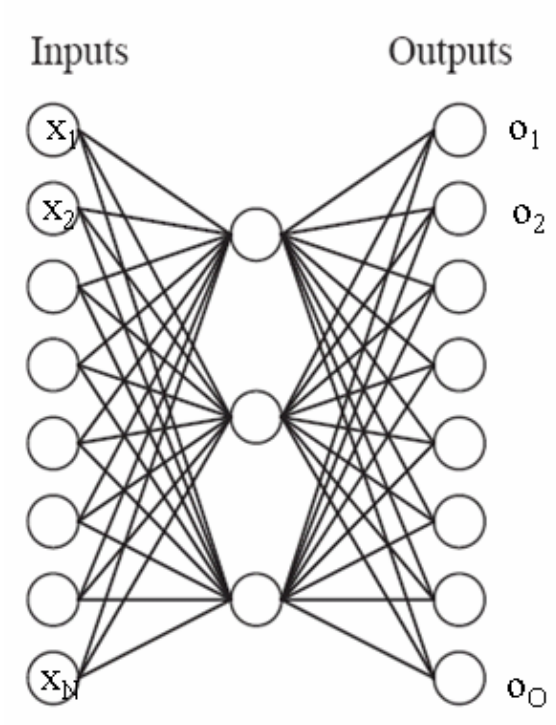
Classification Using Perceptron

- See plot_sgd_comparison.py



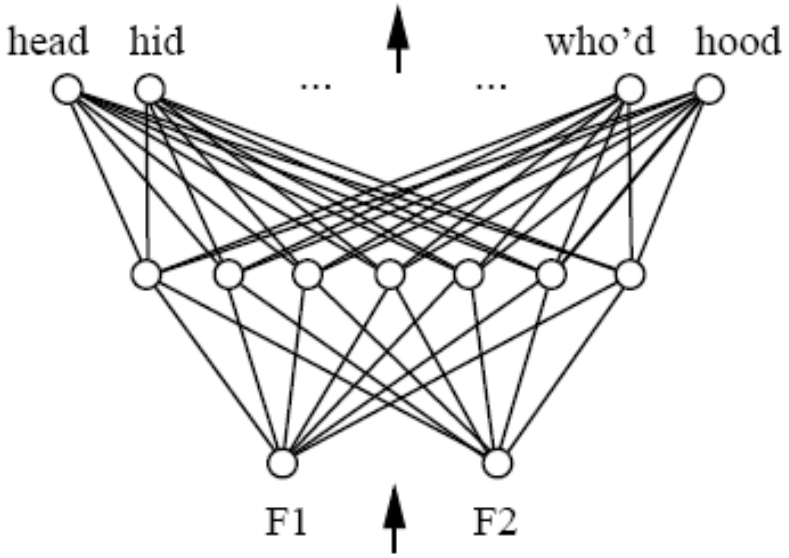
Module 2 : Multi Layer Perceptron

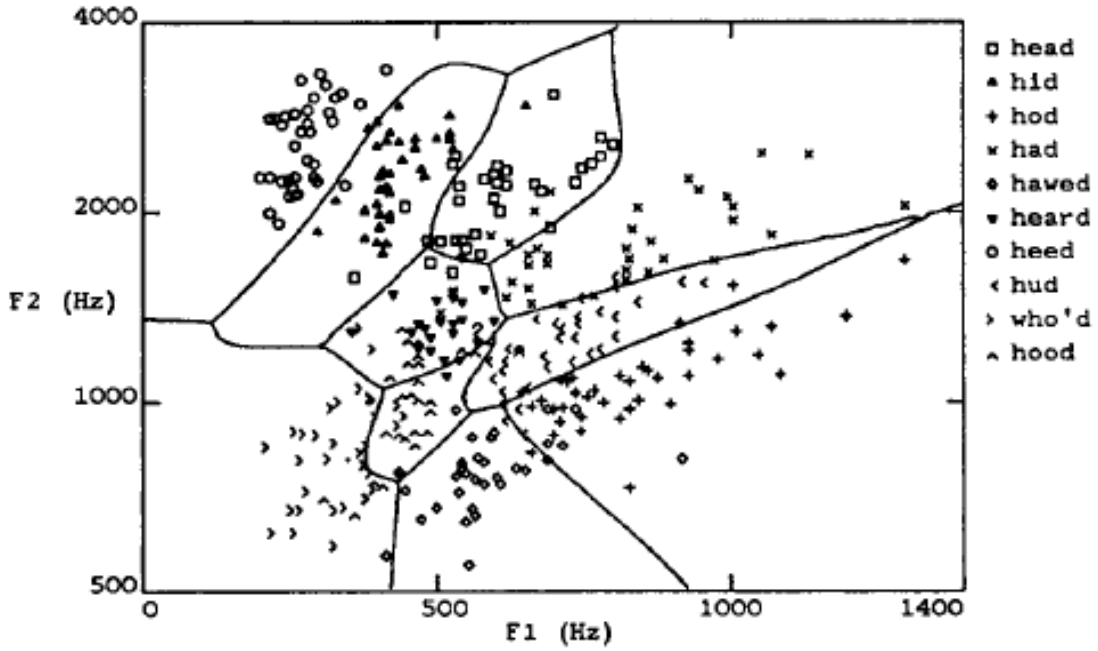




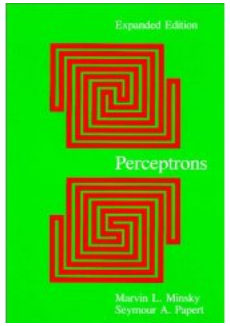
Multi-layer Perceptrons (MLPs)

- Single-layer perceptrons can only represent linear decision surfaces.
- Multi-layer perceptrons can represent non-linear decision surfaces.





Module 2 : Multi Layer Perceptron



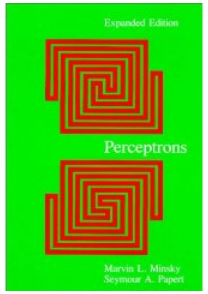
Bad news: No algorithm for learning in multi-layered networks, and no convergence theorem was known in 1969!

Minsky & Papert (1969) *"[The perceptron] has many features to attract attention: its linearity; its intriguing learning theorem; its clear paradigmatic simplicity as a kind of parallel computation. There is no reason to suppose that any of these virtues carry over to the many-layered version. Nevertheless, we consider it to be an important research problem to elucidate (or reject) our intuitive judgment that the extension is sterile."*

Minsky & Papert (1969) pricked the neural network balloon …they almost killed the field.

Rumors say these results may have killed Rosenblatt….

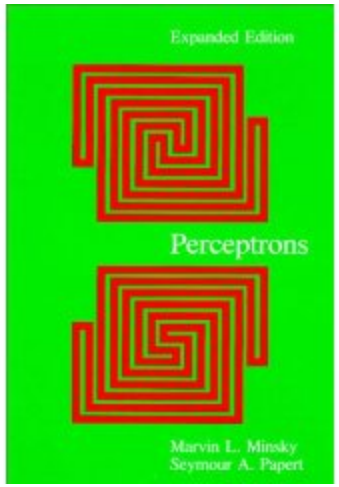
Module 2 : Multi Layer Perceptron

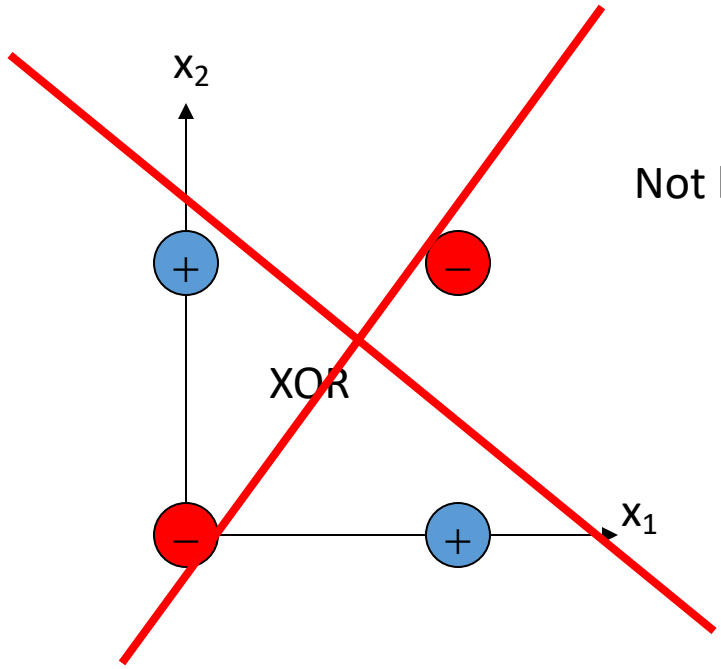


- Two major problems they saw were
  1. How can the learning algorithm apportion credit (or blame) to individual weights for incorrect classifications depending on a (sometimes) large number of weights?

  2. How can such a network learn useful higher-order features?

Perceptron: Linear Separable Functions



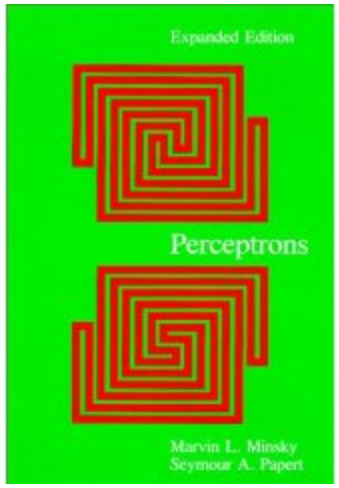


Not linearly separable

Minsky & Papert (1969)

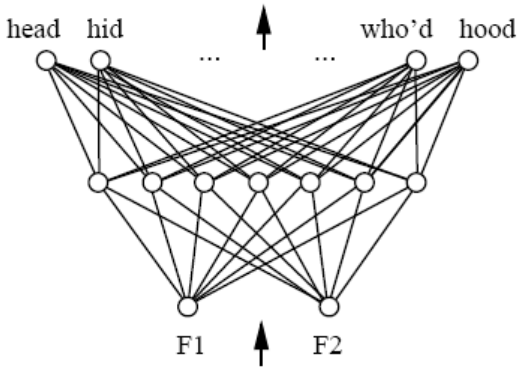
Perceptrons can only represent linearly separable functions.

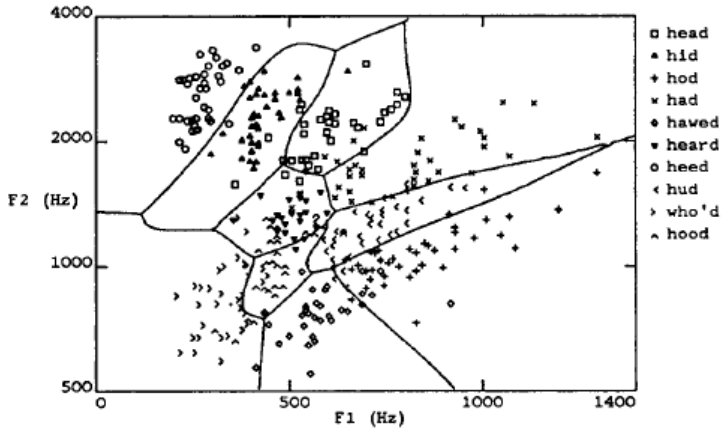
Module 2 : Multi Layer Perceptron



Good news: Adding hidden layer allows more target functions to be represented.

Minsky & Papert (1969)





Hidden Units

- **Hidden units are nodes that are situated between the input nodes and the output nodes.**
- **Hidden units allow a network to learn non-linear functions.**
- **Hidden units allow the network to represent combinations of the input features.**

Boolean functions

- Perceptron can be used to represent the following Boolean functions
  - AND
  - OR
  - Any m-of-n function
  - NOT
  - NAND (NOT AND)
  - NOR (NOT OR)

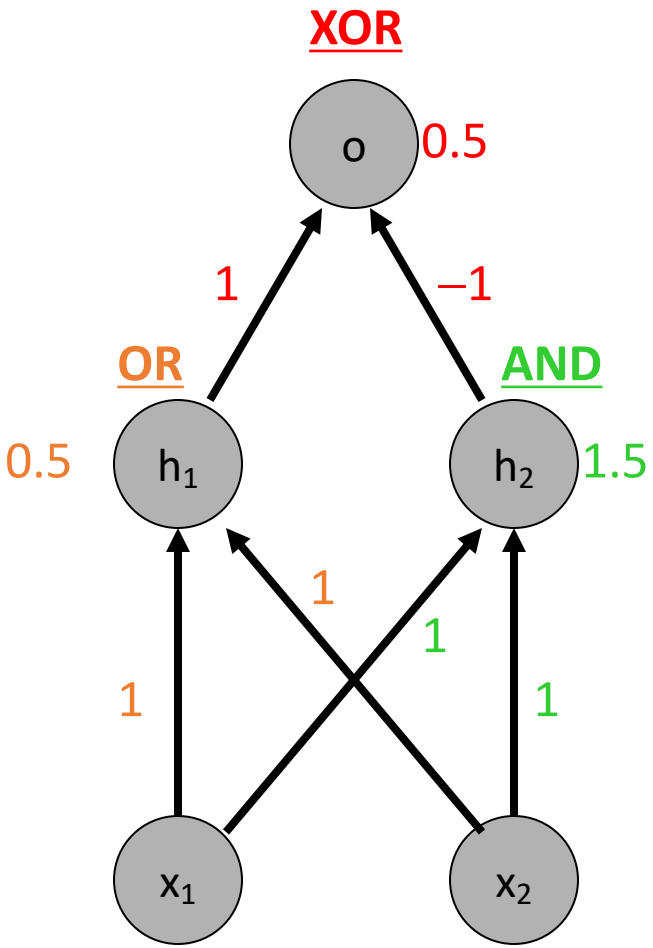
- Every Boolean function can be represented by a network of interconnected units based on these primitives

- Two levels (i.e., one hidden layer) is enough!!!

Boolean XOR

$X1 \oplus X2 \quad \Leftrightarrow (X1 \vee X2) \wedge \neg(X1 \wedge X2)$

Not Linear separable →
Cannot be represented by a single-layer perceptron



Let's consider a single hidden layer network, using as building blocks threshold units.

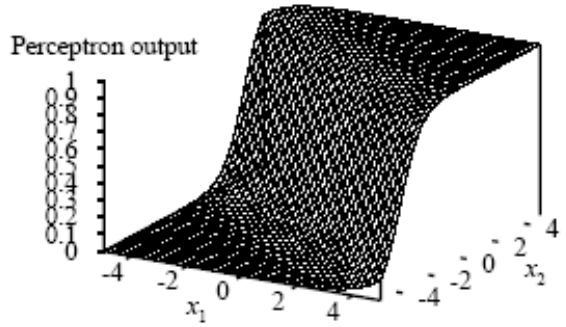
$w_1 x_1 + w_2 x_2 - w_0 > 0$

Expressiveness of MLP: Soft Threshold

- Advantage of adding hidden layers

→it enlarges the space of hypotheses that the network can represent.

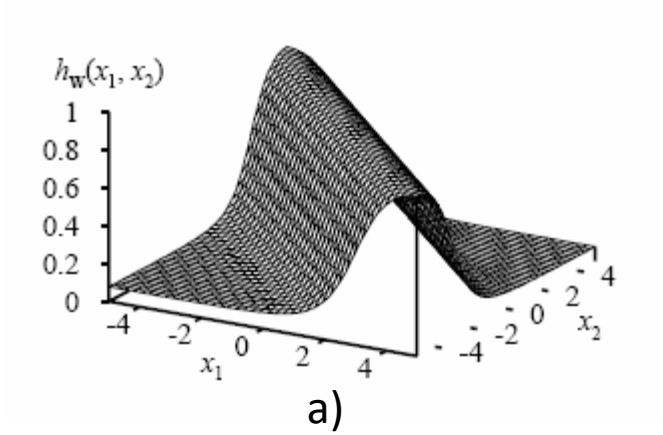
Example: we can think of each hidden unit as a perceptron that represents a soft threshold function in the input space, and an output unit as as a soft-thresholded linear combination of several such functions.



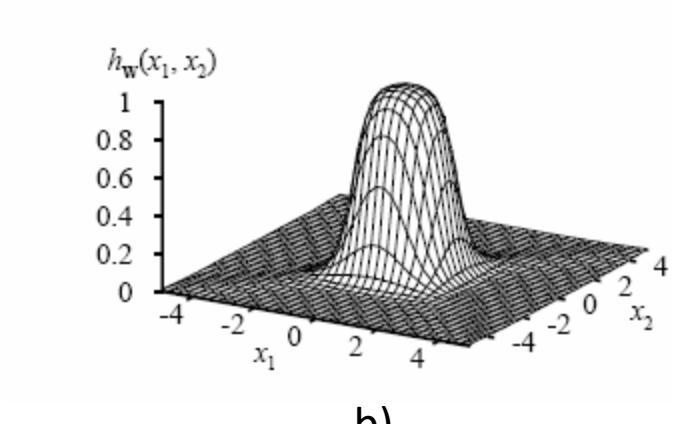
(b)

Soft threshold function

Expressiveness of MLP



a)



b)

(a) The result of combining two opposite-facing soft threshold functions to produce a ridge.

(b) The result of combining two ridges to produce a bump.

Add bumps of various sizes and locations to any surface

All continuous functions w/ 2 layers, all functions w/ 3 layers

Expressiveness of MLP

- With a single, sufficiently large hidden layer, it is possible to represent
- any continuous function of the inputs with arbitrary accuracy;

- With two layers, even discontinuous functions can be represented.

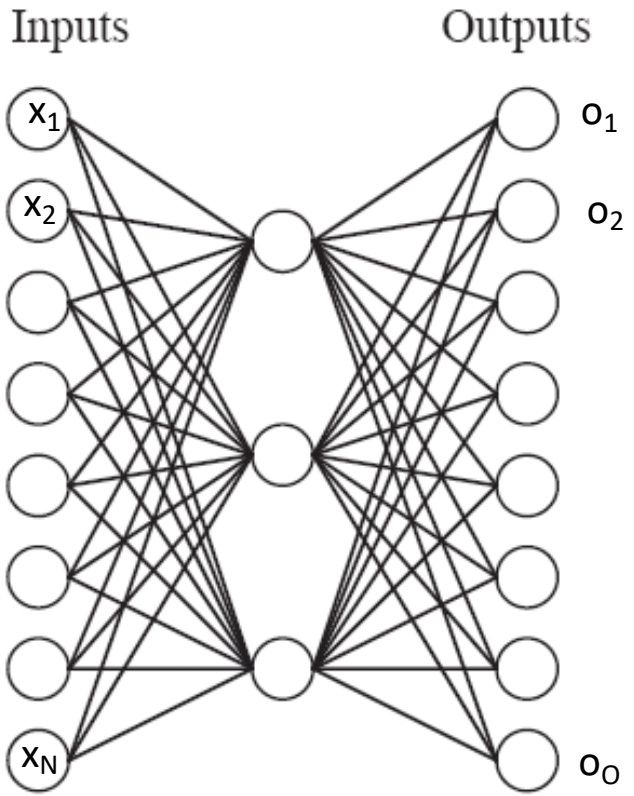
- The proof is complex → main point, required number of hidden units grows exponentially with the number of inputs.
- For example, $2^n/n$ hidden units are needed to encode all Boolean functions of n inputs.

- Issue: For any *particular* network structure, it is harder to characterize
- exactly which functions can be represented and which ones cannot.

Multi-Layer Feedforward Networks

- Boolean functions:
- Every boolean function can be represented by a network with single hidden layer
- But might require exponential (in number of inputs) hidden units
- Continuous functions:
- Every bounded continuous function can be approximated with arbitrarily small error, by network with single hidden layer [Cybenko 1989; Hornik et al. 1989]

- Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].



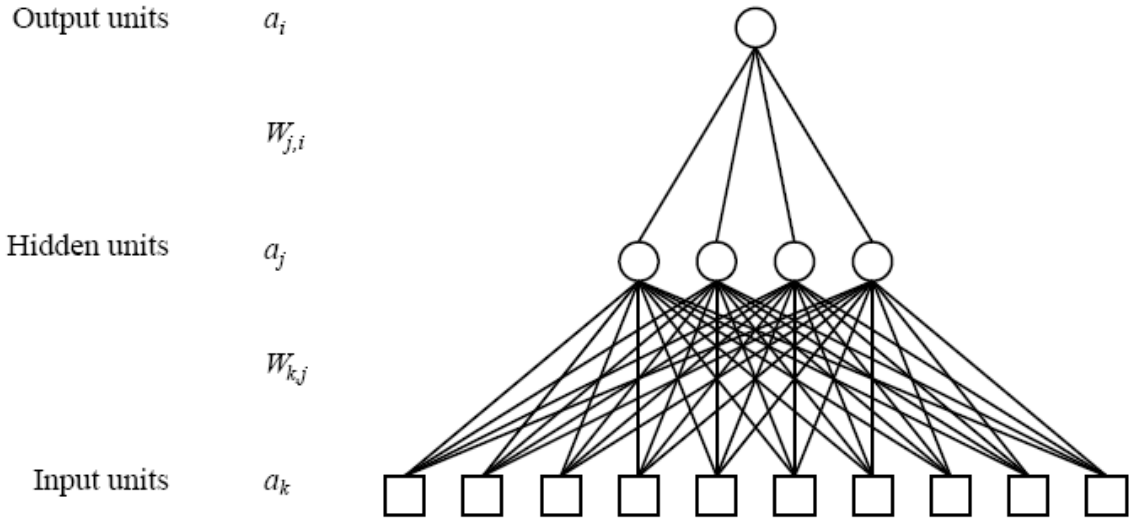
$$o_i = g\left(\sum_h w_{h,i}\, g\left(\sum_j w_{j,h}\, x_j\right)\right)$$

The Restaurant Example

- Problem: decide whether to wait for a table at a restaurant, based on the following attributes:
  1. Alternate: is there an alternative restaurant nearby?
  2. Bar: is there a comfortable bar area to wait in?
  3. Fri/Sat: is today Friday or Saturday?
  4. Hungry: are we hungry?
  5. Patrons: number of people in the restaurant (None, Some, Full)
  6. Price: price range ($, $$, $$$)
  7. Raining: is it raining outside?
  8. Reservation: have we made a reservation?
  9. Type: kind of restaurant (French, Italian, Thai, Burger)
  10. WaitEstimate: estimated waiting time (0-10, 10-30, 30-60, >60)

Goal predicate: WillWait?

Multi-layer Feedforward Neural Networks or Multi-Layer Perceptrons (MLP)

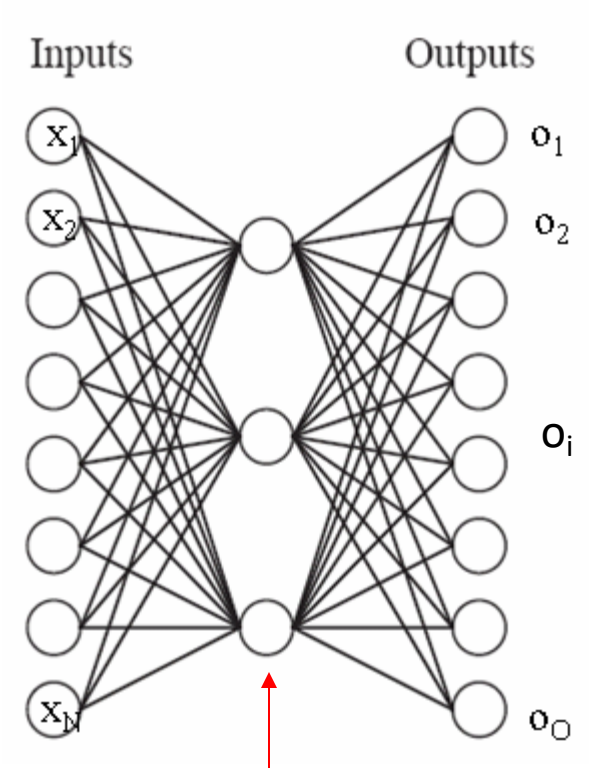


- A multilayer neural network with one hidden layer and 10 inputs, suitable
- for the restaurant problem.
- Layers are often fully connected (but not always).
- Number of hidden units typically chosen by hand.

Module 2 : Multi Layer Perceptron

Learning Algorithm for MLP

Learning Algorithms for MLP



How to compute the errors for the hidden units?

Goal: minimize sum squared errors

$Err_1=y_1-o_1$

$Err_2=y_2-o_2$

$Err_i=y_i-o_i$

$Err_o=y_o-o_o$

Clear error at the output layer

$$E = \frac{1}{2}\sum_i (y_i - o_i)^2$$

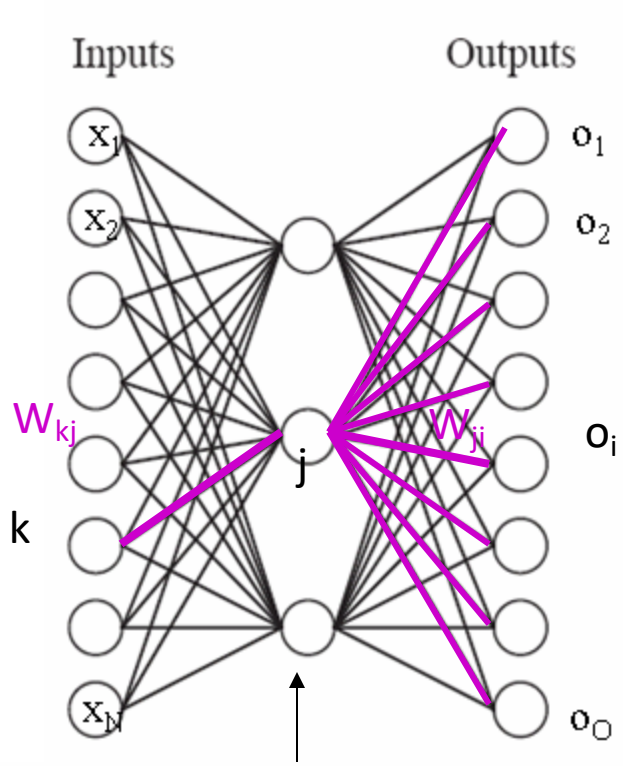
$$o_i = g\left(\sum_h w_{h,i}\, g\left(\sum_j w_{j,h}\, x_j\right)\right)$$

parameterized function of inputs: weights are the parameters of the function.

We can **back-propagate** the error from the output layer to the hidden layers.

The back-propagation process emerges directly from a derivation of the overall error gradient.

Backpropagation Learning Algorithm for MLP



Perceptron update:

$$W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j$$

$Err_i = y_i - o_i$

Output layer weight update (similar to perceptron)

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$$

$$\Delta_i = Err_i \times g'(in_i)$$

Hidden layer: back-propagate the error from the output layer:

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j \ .$$

$$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i \ .$$

Err_j → "Error" for hidden node j

Hidden node j is "responsible" for some fraction of the error i in each of the output nodes to which it connects

→ depending on the strength of the connection between the hidden node and the output node i.

Backpropagation Training (Overview)

- Optimization Problem
  - Obj.: minimize E

$$E = \frac{1}{2}\sum_i (y_i - a_i)^2 \ ,$$

Choice of learning rate α

How many restarts (local optima) of search to find good optimum of objective function?

- Variables: network weights w_{ij}
- Algorithm: local search via gradient descent.
- **Randomly initialize weights.**
- **Until performance is satisfactory, cycle through examples (epochs):**
  - Update each weight:

Output node:

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$$

$$\Delta_i = Err_i \times g'(in_i)$$

Hidden node:

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j$$

$$\Delta_j = g'(in_j)\sum_i W_{j,i}\Delta_i \ .$$

See derivation details in the next hidden slides (pages 745-747 R&N)

Learning Algorithms for MLP

- Similar to the perceptron learning algorithm:
  - One minor difference is that we may have several outputs, so we have an output vector $\mathbf{h_W(x)}$ rather than a single value, and each example has an output vector **y**.
  - The major difference is that, whereas the error **y** - $\mathbf{h_W}$ at the perceptron output layer is clear, the error at the hidden layers seems mysterious because the training data does not say what value the hidden nodes should have

We can **back-propagate** the error from the output layer to the hidden layers. The back-propagation process emerges directly from a derivation of the overall error gradient.

Back Propagation Learning

Output layer: same as for single-layer perceptron,

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$$

where $\Delta_i = Err_i \times g'(in_i)$

Perceptron update:

$$W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j$$

Err_i → i^{th} component of vector y - h_W

Hidden layer: **back-propagate** the error from the output layer:

$$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i \ .$$

Update rule for weights in hidden layer:

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j \ .$$

Hidden node j is "responsible"
for some fraction of the error i
in each of the output nodes to which it connects →
depending on . the strength of the connection
between the hidden node and the output node i.

Back Propagation Learning (Derivation)

The squared error on a single example is defined as

$$E = \frac{1}{2}\sum_i (y_i - a_i)^2 \ ,$$

where the sum is over the nodes in the output layer.

$$\begin{aligned}
\frac{\partial E}{\partial W_{j,i}} &= -(y_i - a_i)\frac{\partial a_i}{\partial W_{j,i}} = -(y_i - a_i)\frac{\partial g(in_i)}{\partial W_{j,i}} \\
&= -(y_i - a_i)g'(in_i)\frac{\partial in_i}{\partial W_{j,i}} = -(y_i - a_i)g'(in_i)\frac{\partial}{\partial W_{j,i}}\left(\sum_j W_{j,i}a_j\right) \\
&= -(y_i - a_i)g'(in_i)a_j = -a_j\Delta_i
\end{aligned}$$

Back Propagation Learning (Derivation)

$$
\begin{aligned}
\frac{\partial E}{\partial W_{k,j}} &= -\sum_i (y_i - a_i)\frac{\partial a_i}{\partial W_{k,j}} = -\sum_i (y_i - a_i)\frac{\partial g(in_i)}{\partial W_{k,j}} \\
&= -\sum_i (y_i - a_i) g'(in_i)\frac{\partial in_i}{\partial W_{k,j}} = -\sum_i \Delta_i \frac{\partial}{\partial W_{k,j}}\left(\sum_j W_{j,i} a_j\right) \\
&= -\sum_i \Delta_i W_{j,i}\frac{\partial a_j}{\partial W_{k,j}} = -\sum_i \Delta_i W_{j,i}\frac{\partial g(in_j)}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j)\frac{\partial in_j}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j)\frac{\partial}{\partial W_{k,j}}\left(\sum_k W_{k,j} a_k\right) \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j) a_k = -a_k \Delta_j
\end{aligned}
$$

Back-Propagation Learning Algorithm

---

function BACK-PROP-LEARNING(*examples*, *network*) **returns** a neural network
  inputs: *examples*, a set of examples, each with input vector **x** and output vector **y**
    network, a multilayer network with L layers, weights $W_{j,i}$, activation function g

  repeat
    for each e **in** *examples* **do**
      for each node j in the input layer **do** $a_j \leftarrow x_j[e]$
      for $\ell = 2$ **to** M **do**
        $in_i \leftarrow \sum_j W_{j,i}\, a_j$
        $a_i \leftarrow g(in_i)$
      for each node i in the output layer **do**
        $\Delta_i \leftarrow g'(in_i) \times (y_i[e] - a_i)$
      for $\ell = M - 1$ **to** 1 **do**
        for each node j in layer ℓ **do**
          $\Delta_j \leftarrow g'(in_j) \sum_i W_{j,i}\, \Delta_i$
          for each node i in layer $\ell + 1$ **do**
            $W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$
  until some stopping criterion is satisfied
  return NEURAL-NET-HYPOTHESIS(*network*)

Classification Using MLP

- **Compare Stochastic learning strategies for MLPClassifier :**

See plot_mlp_training_curves.py

```
print(__doc__)
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import MinMaxScaler
from sklearn import datasets

# different learning rate schedules and momentum parameters
params = [{'solver': 'sgd', 'learning_rate': 'constant', 'momentum': 0,
           'learning_rate_init': 0.2},
          {'solver': 'sgd', 'learning_rate': 'constant', 'momentum': .9,
           'nesterovs_momentum': False, 'learning_rate_init': 0.2},
          {'solver': 'sgd', 'learning_rate': 'constant', 'momentum': .9,
           'nesterovs_momentum': True, 'learning_rate_init': 0.2},
          {'solver': 'sgd', 'learning_rate': 'invscaling', 'momentum': 0,
           'learning_rate_init': 0.2},
          {'solver': 'sgd', 'learning_rate': 'invscaling', 'momentum': .9,
           'nesterovs_momentum': True, 'learning_rate_init': 0.2},
          {'solver': 'sgd', 'learning_rate': 'invscaling', 'momentum': .9,
           'nesterovs_momentum': False, 'learning_rate_init': 0.2},
          {'solver': 'adam', 'learning_rate_init': 0.01}]

labels = ["constant learning-rate", "constant with momentum",
          "constant with Nesterov's momentum",
          "inv-scaling learning-rate", "inv-scaling with momentum",
          "inv-scaling with Nesterov's momentum", "adam"]

plot_args = [{'c': 'red', 'linestyle': '-'},
             {'c': 'green', 'linestyle': '-'},
             {'c': 'blue', 'linestyle': '-'},
             {'c': 'red', 'linestyle': '--'},
             {'c': 'green', 'linestyle': '--'},
             {'c': 'blue', 'linestyle': '--'},
             {'c': 'black', 'linestyle': '-'}]
```

```
def plot_on_dataset(X, y, ax, name):
    # for each dataset, plot learning for each learning strategy
    print("\nlearning on dataset %s" % name)
    ax.set_title(name)
    X = MinMaxScaler().fit_transform(X)
    mlps = []
    if name == "digits":
        # digits is larger but converges fairly quickly
        max_iter = 15
    else:
        max_iter = 400

    for label, param in zip(labels, params):
        print("training: %s" % label)
        mlp = MLPClassifier(verbose=0, random_state=0,
                            max_iter=max_iter, **param)
        mlp.fit(X, y)
        mlps.append(mlp)
        print("Training set score: %f" % mlp.score(X, y))
        print("Training set loss: %f" % mlp.loss_)
    for mlp, label, args in zip(mlps, labels, plot_args):
            ax.plot(mlp.loss_curve_, label=label, **args)


fig, axes = plt.subplots(2, 2, figsize=(15, 10))
# load / generate some toy datasets
iris = datasets.load_iris()
digits = datasets.load_digits()
data_sets = [(iris.data, iris.target),
             (digits.data, digits.target),
             datasets.make_circles(noise=0.2, factor=0.5, random_state=1),
             datasets.make_moons(noise=0.3, random_state=0)]

for ax, data, name in zip(axes.ravel(), data_sets, ['iris', 'digits',
                                                    'circles', 'moons']):
    plot_on_dataset(*data, ax=ax, name=name)

fig.legend(ax.get_lines(), labels, ncol=3, loc="upper center")
plt.show()
```

Classification Using MLP

- **Compare Stochastic learning strategies for MLPClassifier :**

Output:

```
learning on dataset iris
training: constant learning-rate
Training set score: 0.980000
Training set loss: 0.096950
training: constant with momentum
Training set score: 0.980000
Training set loss: 0.049530
training: constant with Nesterov's mome
Training set score: 0.980000
Training set loss: 0.049540
training: inv-scaling learning-rate
Training set score: 0.360000
Training set loss: 0.978444
training: inv-scaling with momentum
Training set score: 0.860000
Training set loss: 0.503452
training: inv-scaling with Nesterov's m
Training set score: 0.860000
Training set loss: 0.504185
training: adam
Training set score: 0.980000
Training set loss: 0.045311
```

```
learning on dataset digits
training: constant learning-rate
Training set score: 0.956038
Training set loss: 0.243802
training: constant with momentum
Training set score: 0.992766
Training set loss: 0.041297
training: constant with Nesterov's momentum
Training set score: 0.993879
Training set loss: 0.042898
training: inv-scaling learning-rate
Training set score: 0.638843
Training set loss: 1.855465
training: inv-scaling with momentum
Training set score: 0.912632
Training set loss: 0.290584
training: inv-scaling with Nesterov's momentum
Training set score: 0.909293
Training set loss: 0.318387
training: adam
Training set score: 0.991653
Training set loss: 0.045934
```

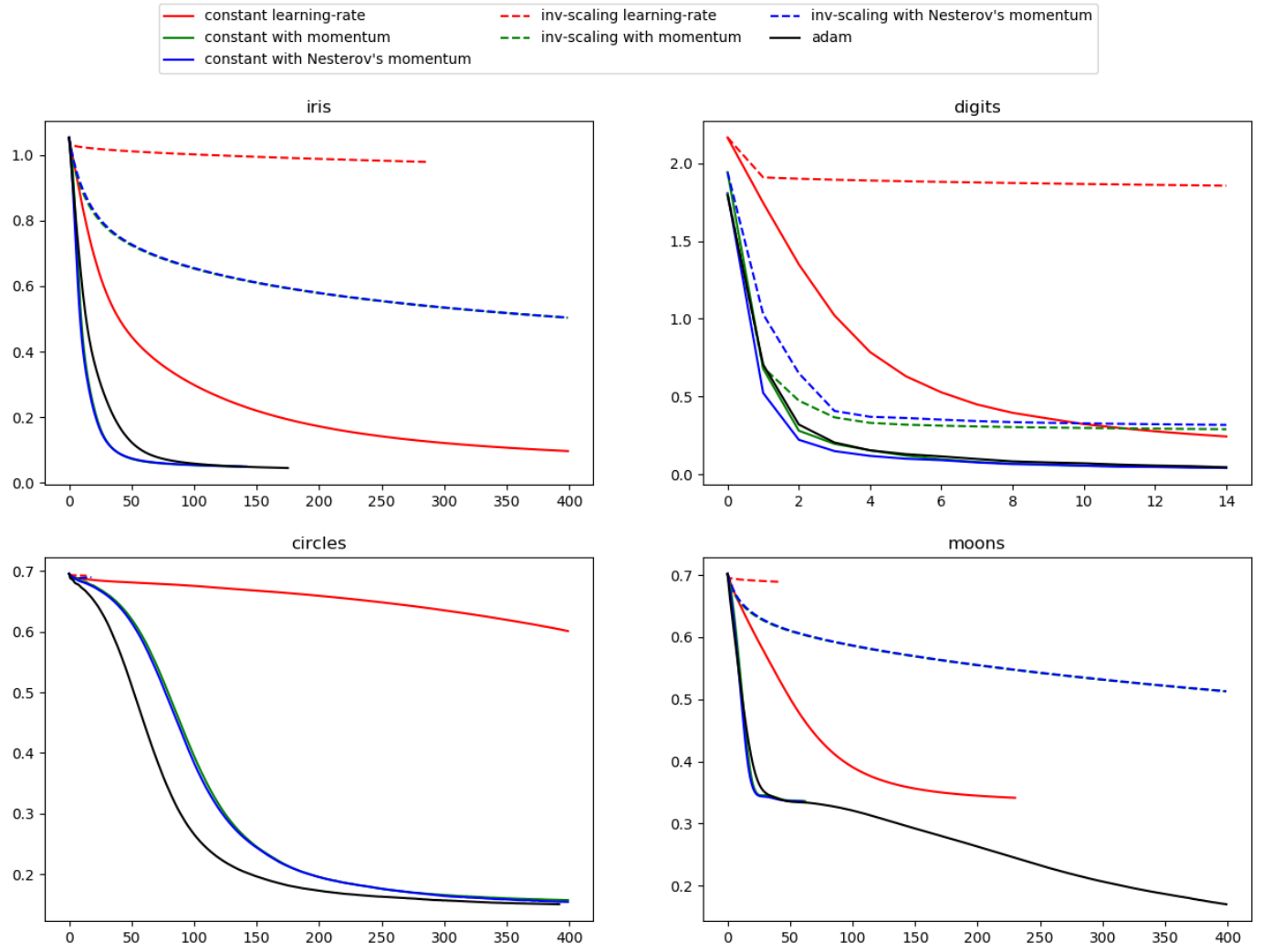
```
learning on dataset circles
training: constant learning-rate
Training set score: 0.840000
Training set loss: 0.601052
training: constant with momentum
Training set score: (
Training set loss: 0
training: constant w
Training set score: (
Training set loss: 0
training: inv-scalin
Training set score: (
Training set loss: 0
training: inv-scalin
Training set score: (
Training set loss: 0
training: inv-scalin
Training set score: (
Training set loss: 0
training: adam
Training set score: (
Training set loss: 0
```

```
learning on dataset moons
training: constant learning-rate
Training set score: 0.850000
Training set loss: 0.341523
training: constant with momentum
Training set score: 0.850000
Training set loss: 0.336188
training: constant with Nesterov's momentum
Training set score: 0.850000
Training set loss: 0.335919
training: inv-scaling learning-rate
Training set score: 0.500000
Training set loss: 0.689015
training: inv-scaling with momentum
Training set score: 0.830000
Training set loss: 0.512595
training: inv-scaling with Nesterov's moment
Training set score: 0.830000
Training set loss: 0.513034
training: adam
Training set score: 0.930000
Training set loss: 0.170087
```

Classification Using MLP

- **Compare Stochastic learning strategies for MLPClassifier :**

Output (Plot):



Classification Using MLP

- **Varying regularization in Multi-layer Perceptron:**

MLPClassifier uses parameter alpha for regularization (L2 regularization) term which helps in avoiding overfitting by penalizing weights with large magnitudes. Following plot displays varying decision function with value of alpha.

See plot_mlp_alpha.py

Module 2 : Multi Layer Perceptron

Classification Using MLP

- **Varying regularization in Multi-layer Perceptron:**

See plot_mlp_alpha.py

```
# Author: Issam H. Laradji
# License: BSD 3 clause

import numpy as np
from matplotlib import pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons, make_circles, make_classification
from sklearn.neural_network import MLPClassifier

h = .02  # step size in the mesh

alphas = np.logspace(-5, 3, 5)
names = ['alpha ' + str(i) for i in alphas]

classifiers = []
for i in alphas:
    classifiers.append(MLPClassifier(alpha=i, random_state=1))

X, y = make_classification(n_features=2, n_redundant=0, n_informative=2,
                           random_state=0, n_clusters_per_class=1)
rng = np.random.RandomState(2)
X += 2 * rng.uniform(size=X.shape)
linearly_separable = (X, y)

datasets = [make_moons(noise=0.3, random_state=0),
            make_circles(noise=0.2, factor=0.5, random_state=1),
            linearly_separable]

figure = plt.figure(figsize=(17, 9))
i = 1
```

```
# iterate over datasets
for X, y in datasets:
    # preprocess dataset, split into training and test part
    X = StandardScaler().fit_transform(X)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.4)

    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    # just plot the dataset first
    cm = plt.cm.RdBu
    cm_bright = ListedColormap(['#FF0000', '#0000FF'])
    ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
    # Plot the training points
    ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright)
    # and testing points
    ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright, alpha=0.6)
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xticks(())
    ax.set_yticks(())
    i += 1

    # iterate over classifiers
    for name, clf in zip(names, classifiers):
        ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
        clf.fit(X_train, y_train)
        score = clf.score(X_test, y_test)

        # Plot the decision boundary. For that, we will assign a color to each
        # point in the mesh [x_min, x_max]x[y_min, y_max].
        if hasattr(clf, "decision_function"):
            Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
        else:
            Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]

        # Put the result into a color plot
```

Classification Using MLP

- **Varying regularization in Multi-layer Perceptron:**

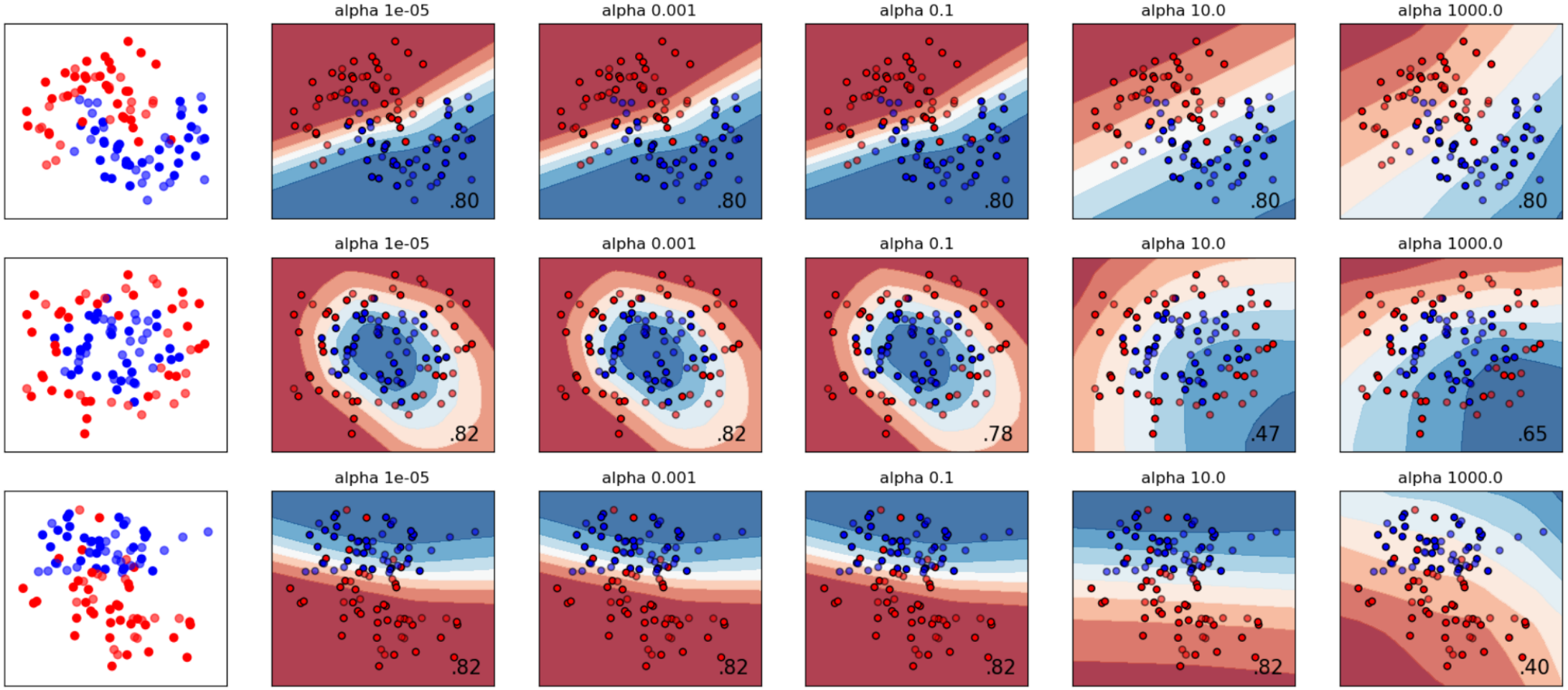
See plot_mlp_alpha.py

```
        # Put the result into a color plot
        Z = Z.reshape(xx.shape)
        ax.contourf(xx, yy, Z, cmap=cm, alpha=.8)

        # Plot also the training points
        ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright,
                   edgecolors='black', s=25)
        # and testing points
        ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright,
                   alpha=0.6, edgecolors='black', s=25)

        ax.set_xlim(xx.min(), xx.max())
        ax.set_ylim(yy.min(), yy.max())
        ax.set_xticks(())
        ax.set_yticks(())
        ax.set_title(name)
        ax.text(xx.max() - .3, yy.min() + .3, ('%.2f' % score).lstrip('0'),
                size=15, horizontalalignment='right')
        i += 1

figure.subplots_adjust(left=.02, right=.98)
plt.show()
```

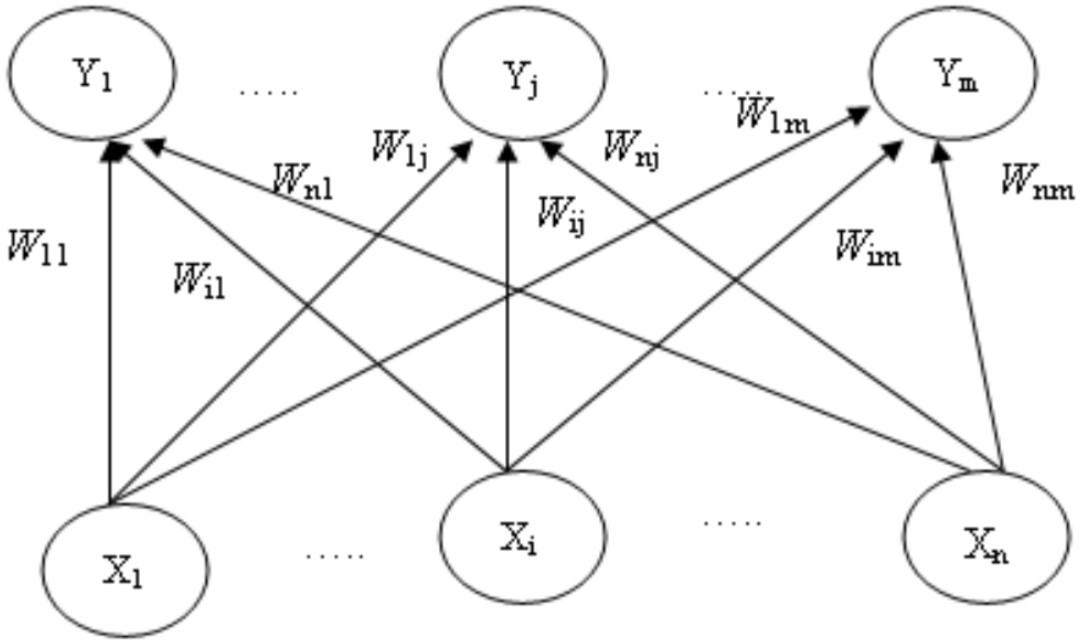
Classification Using MLP

- **Varying regularization in Multi-layer Perceptron:**

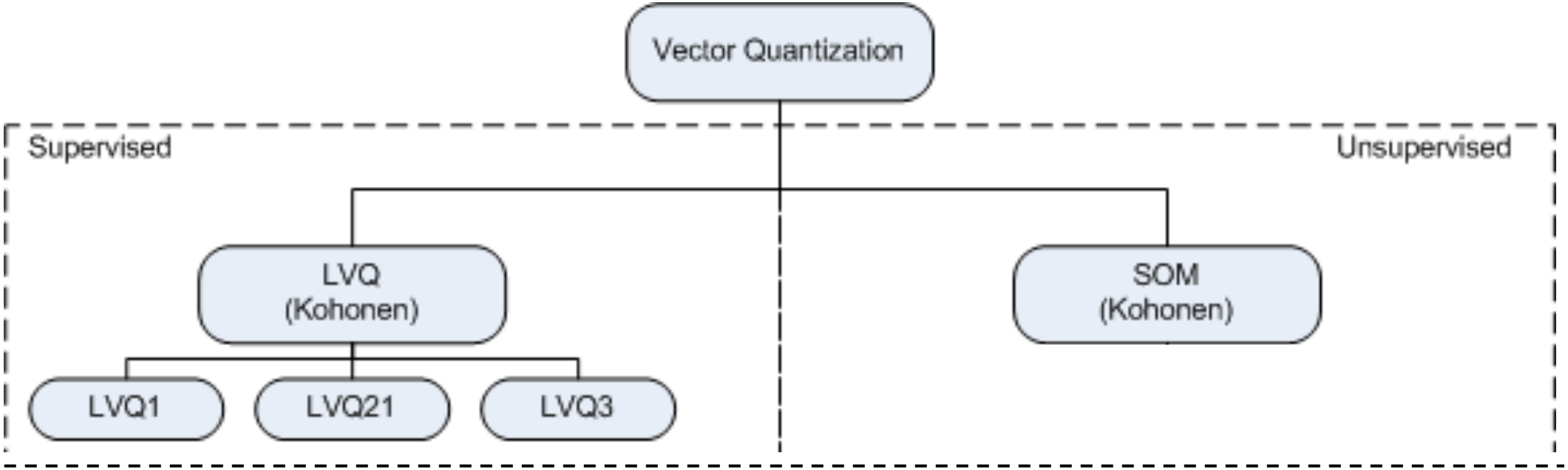
See plot_mlp_alpha.py (output)



Module 3 : Learning Vector Quantization

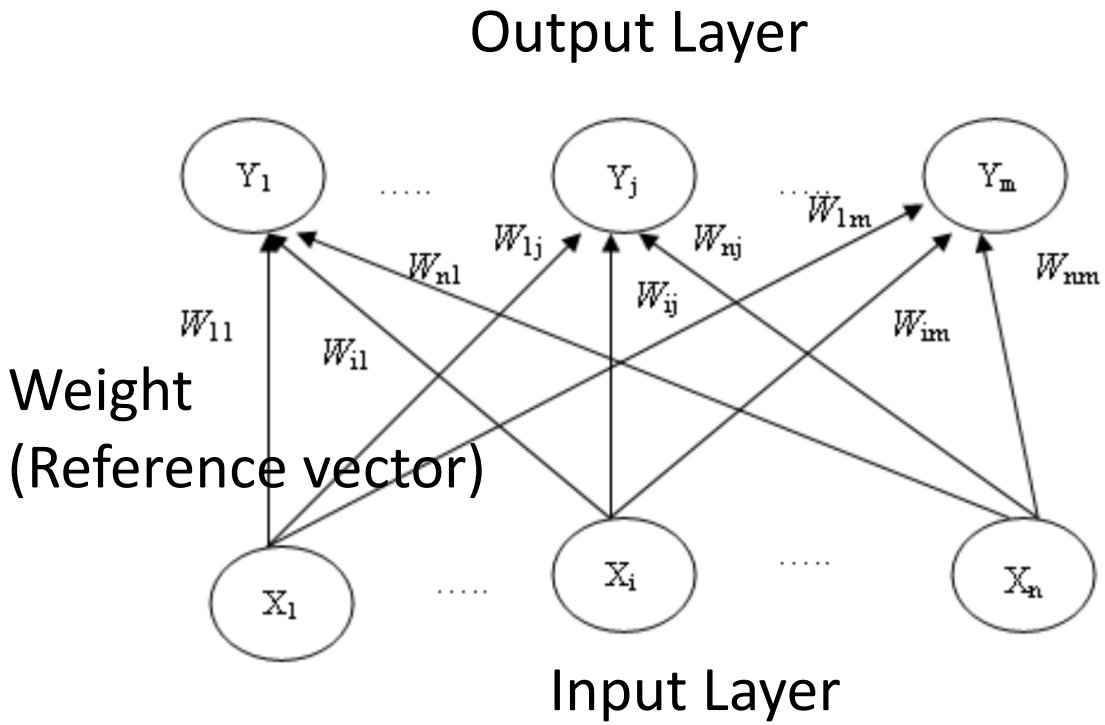


Hierarchy

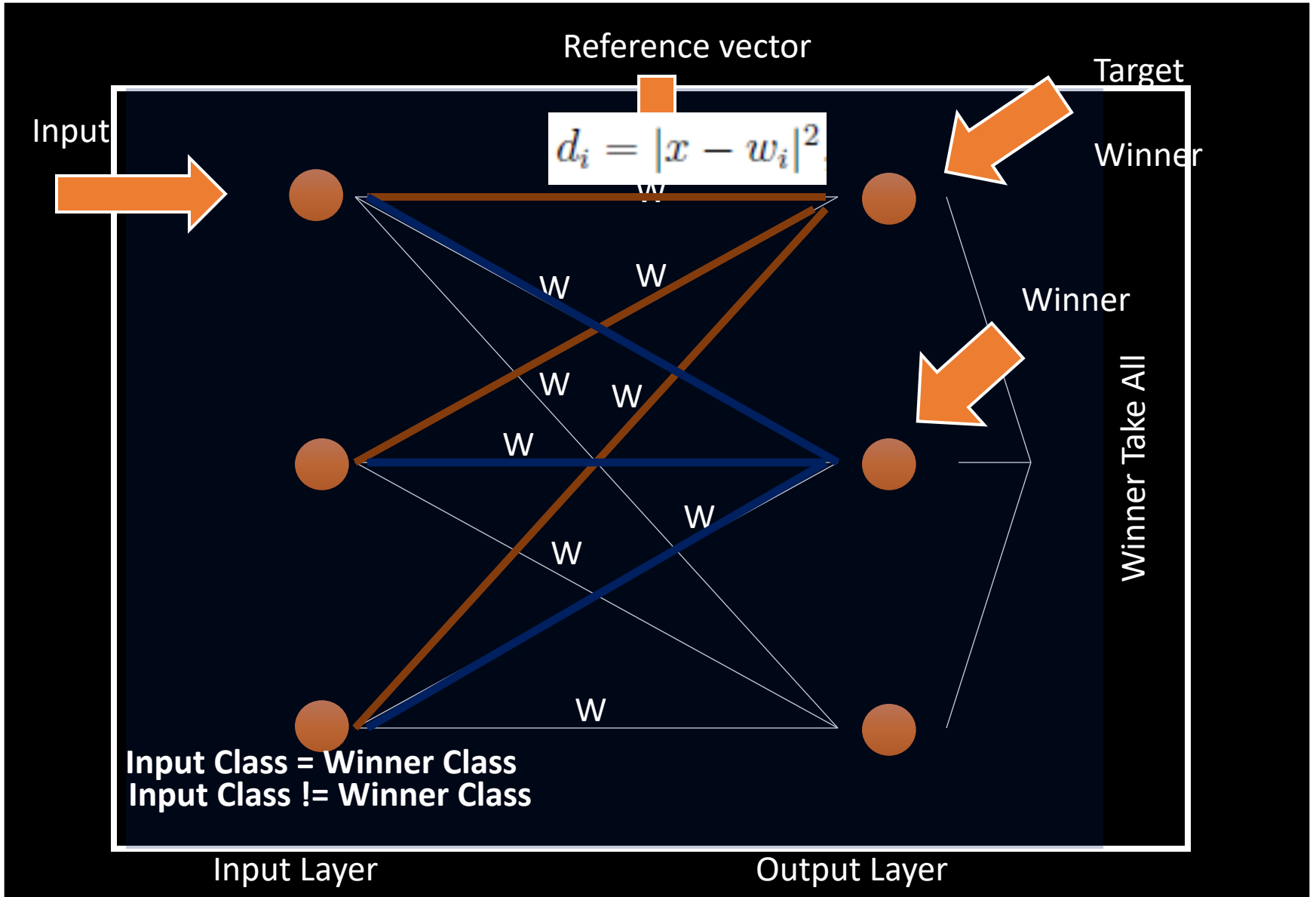

Vector Quantization
Supervised
Unsupervised
LVQ
(Kohonen)
SOM
(Kohonen)
LVQ1
LVQ21
LVQ3

Competition Based Learning

- Learning Vector Quantization – (LVQ)
  - Simple structure, without hidden layers,
  - Reference vector / Weight (prototype) → class representation
  - Winner-take-all learning method



LVQ Learning method



LVQ Learning method

Algorithm LVQ1 Train(W,x)

Require: W, x

$w_p \leftarrow \text{ClosestDistanceWeight}(x, W)$

if $C_x = C_{w_p}$ **then**

$w_{p,t+1} \leftarrow w_{p,t} + \alpha_t . (x - w_{p,t})$

else if $C_x \neq C_{w_p}$ **then**

$w_{p,t+1} \leftarrow w_{p,t} - \alpha_t . (x - w_{p,t})$

end if

$\alpha \leftarrow \text{getNextLearningRate}()$

LVQ 2

- LVQ2 : Variant of LVQ1
- Window parameter (ω)
- Check distance for winner and runner-up
- Limitation in update rule formula
- Only update if input class = winner class

LVQ2 Learning

Algorithm **LVQ2 Train(W,x)**

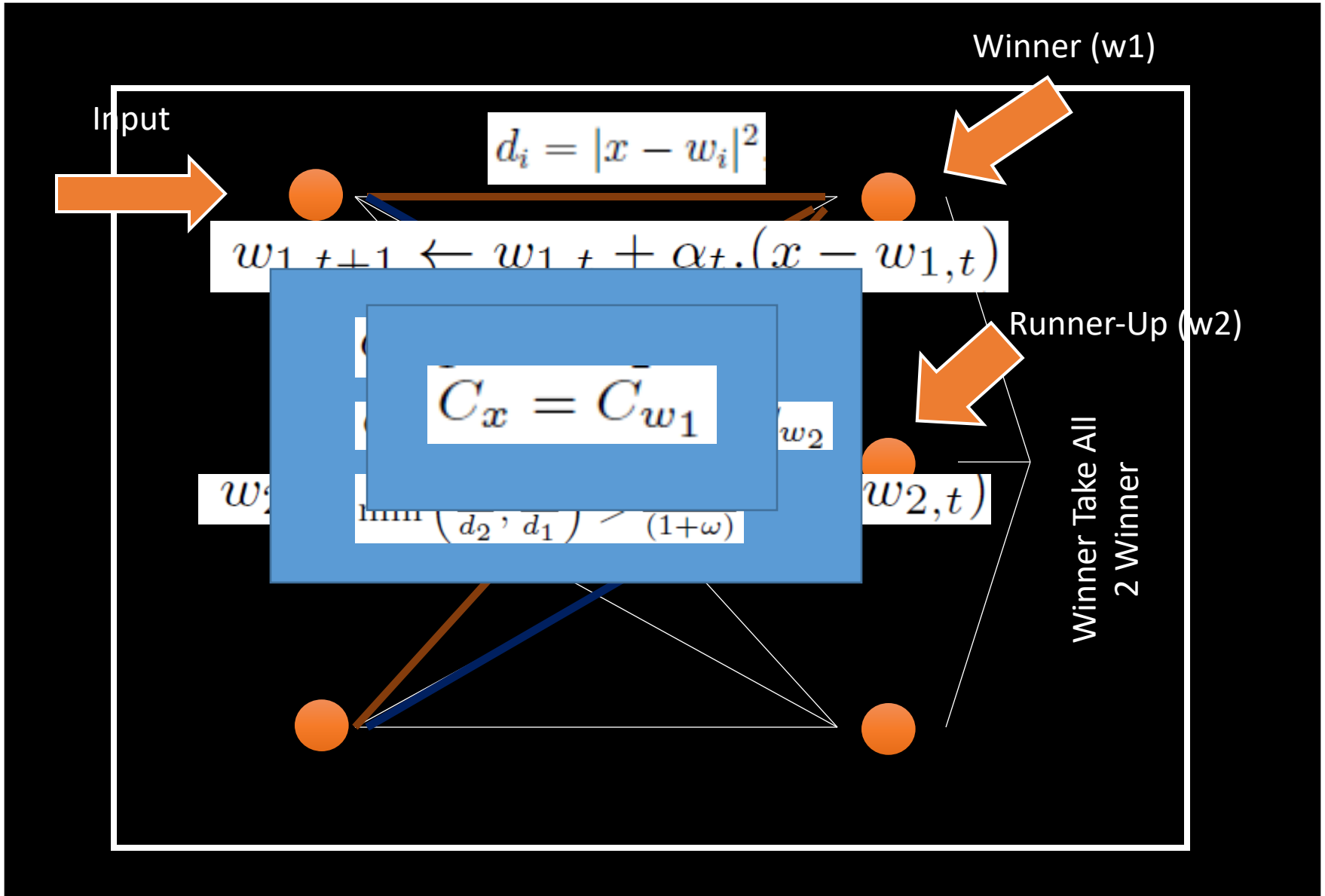
Require: W, x

$w_p \leftarrow \text{ClosesDistanceWeighht}(x, W)$
$w_r \leftarrow \text{RunnerUpClosestDistanceWeight}(x, W)$
$d_p \leftarrow distance(x, w_p)$
$d_r \leftarrow distance(x, w_r)$
if $C_{w_p} \neq C_{w_r}$ **then**
  if $C_x = C_{w_r}$ **then**
    if $\frac{d_p}{d_r} > (1-\omega)$ AND $\frac{d_r}{d_p} < (1+\omega)$ **then**
      $w_{r,t+1} \leftarrow w_{r,t} + \alpha_t . (x - w_{r,t})$
      $w_{p,t+1} \leftarrow w_{p,t} - \alpha_t . (x - w_{p,t})$
    end if
  end if
end if
$\alpha \leftarrow \text{getNextLearningRate}()$

LVQ 2.1

- Variant of LVQ 2
- Different limitation compared to LVQ2
- Update in each instance
  (not only if input class = winner class)

LVQ21


Winner (w1)
Input
$d_i = |x - w_i|^2$
$w_{1,t+1} \leftarrow w_{1,t} + \alpha_t.(x - w_{1,t})$
$C_x = C_{w_1}$
Runner-Up (w2)
$w_{2,t})$
Winner Take All
2 Winner

LVQ2.1 Learning

Algorithm **LVQ21 Train(W,x)**

Require: W, x

$w_1 \leftarrow$ ClosesDistanceWeighht(x, W)
$w_2 \leftarrow$ RunnerUpClosestDistanceWeight(x, W)
$d_1 \leftarrow distance(x, w_1)$
$d_2 \leftarrow distance(x, w_2)$
if $C_{w_1} \neq C_{w_2}$ **then**
  if $C_x = C_{w_1}$ or $C_x = C_{w_2}$ **then**
    if $\min\left(\frac{d_1}{d_2}, \frac{d_2}{d_1}\right) > \frac{(1-\omega)}{(1+\omega)}$ **then**
      $w_{1,t+1} \leftarrow w_{1,t} + \alpha_t.(x - w_{1,t})$
      $w_{2,t+1} \leftarrow w_{2,t} - \alpha_t.(x - w_{2,t})$
    end if
  end if
end if
$\alpha \leftarrow$ getNextLearningRate()

Classification Using LVQ

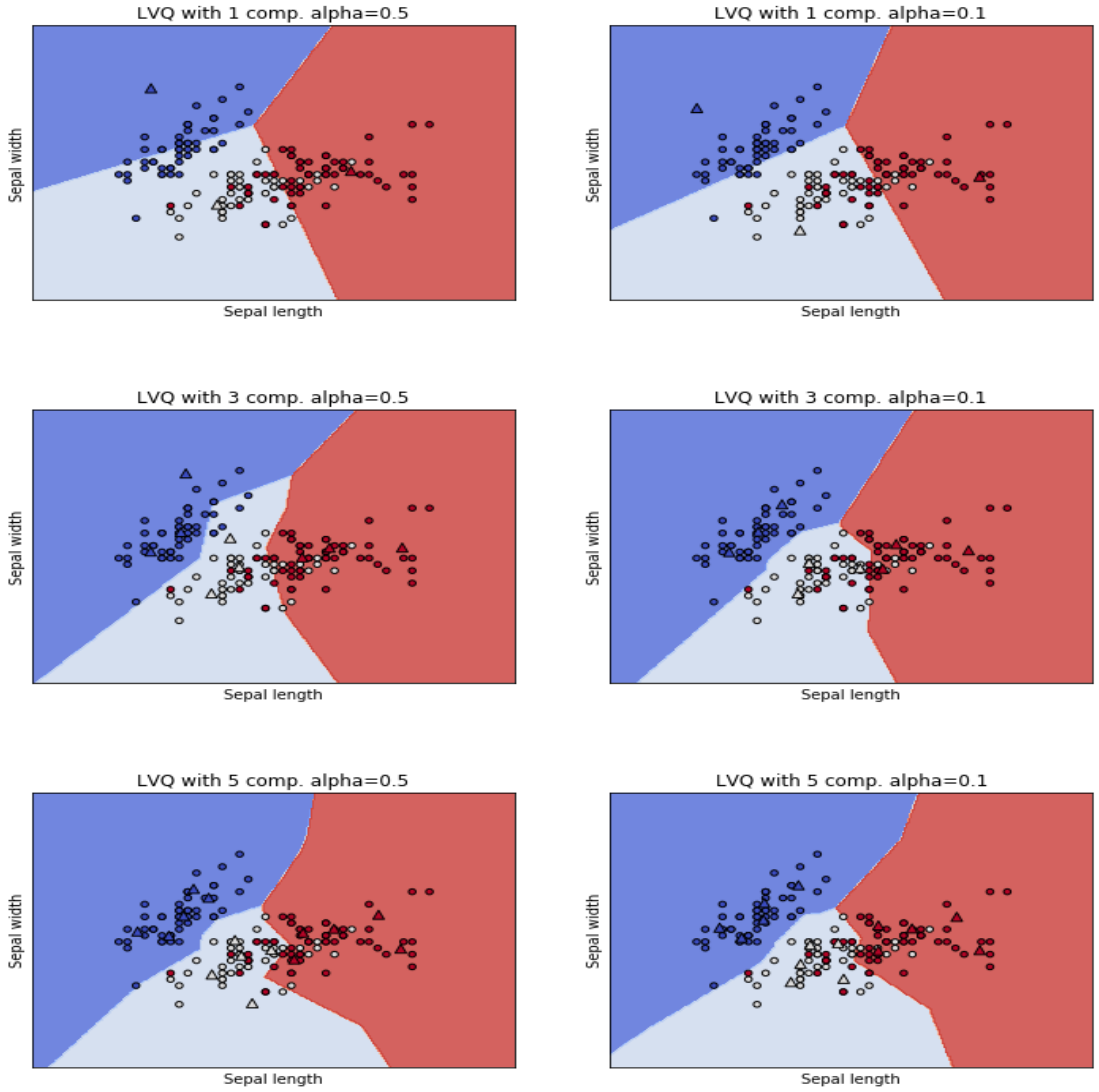
- LVQ Classifier : See LVQClassifier.py
- Classification example : See lvqclassification1.py

```
#print(__doc__)
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets
from LVQClassifier import LVQClassifier as LVQ

def make_meshgrid(x, y, h=.02):
    """Create a mesh of points to plot in

    Parameters
    ----------
    x: data to base x-axis meshgrid on
    y: data to base y-axis meshgrid on
    h: stepsize for meshgrid, optional
    Returns
    -------
    xx, yy : ndarray
    """
    x_min, x_max = x.min() - 1, x.max() + 1
    y_min, y_max = y.min() - 1, y.max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    return xx, yy

def plot_contours(ax, clf, xx, yy, **params):
    """Plot the decision boundaries for a classifier.

    Parameters
    ----------
    ax: matplotlib axes object
    clf: a classifier
    xx: meshgrid ndarray
    yy: meshgrid ndarray
    params: dictionary of params to pass to contourf, optional
    """
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    out = ax.contourf(xx, yy, Z, **params)
    return out
```

```
# import some data to play with
iris = datasets.load_iris()
# Take the first two features. We could avoid this by using a
# two-dim dataset
X = iris.data[:, :2]
y = iris.target

# LVQ parameter
epochs = 10
LVQ2 = False
models = (LVQ(n_components=1,alpha=0.5,epochs=epochs,LVQ2=LVQ2),
          LVQ(n_components=1,alpha=0.1,epochs=epochs,LVQ2=LVQ2),
          LVQ(n_components=3,alpha=0.5,epochs=epochs,LVQ2=LVQ2),
          LVQ(n_components=3,alpha=0.1,epochs=epochs,LVQ2=LVQ2),
          LVQ(n_components=5,alpha=0.5,epochs=epochs,LVQ2=LVQ2),
          LVQ(n_components=5,alpha=0.1,epochs=epochs,LVQ2=LVQ2))

models = (clf.fit(X, y) for clf in models) # sklearn loop over the models

# title for the plots
titles = ('LVQ with 1 comp. alpha=0.5',
          'LVQ with 1 comp. alpha=0.1',
          'LVQ with 3 comp. alpha=0.5',
          'LVQ with 3 comp. alpha=0.1',
          'LVQ with 5 comp. alpha=0.5',
          'LVQ with 5 comp. alpha=0.1')

fig, sub = plt.subplots(3, 2, figsize=(12,15))
plt.subplots_adjust(wspace=0.2, hspace=0.4)

X0, X1 = X[:, 0], X[:, 1]
xx, yy = make_meshgrid(X0, X1)

for clf, title, ax in zip(models, titles, sub.flatten()):
    X_LVQ = clf.weights
    y_LVQ = clf.label_weights
    plot_contours(ax, clf, xx, yy,
                  cmap=plt.cm.coolwarm, alpha=0.8)
    ax.scatter(X0, X1, c=y, cmap=plt.cm.coolwarm, s=20, edgecolors='k')
    ax.scatter(X_LVQ[:, 0], X_LVQ[:, 1], c=y_LVQ,
               cmap=plt.cm.coolwarm, s=50, marker='^', edgecolors='k')
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xlabel('Sepal length')
    ax.set_ylabel('Sepal width')
    ax.set_xticks(())
    ax.set_yticks(())
    ax.set_title(title)
plt.show()
```

Classification Using LVQ

- lvqclassification1.py



Classification Using LVQ

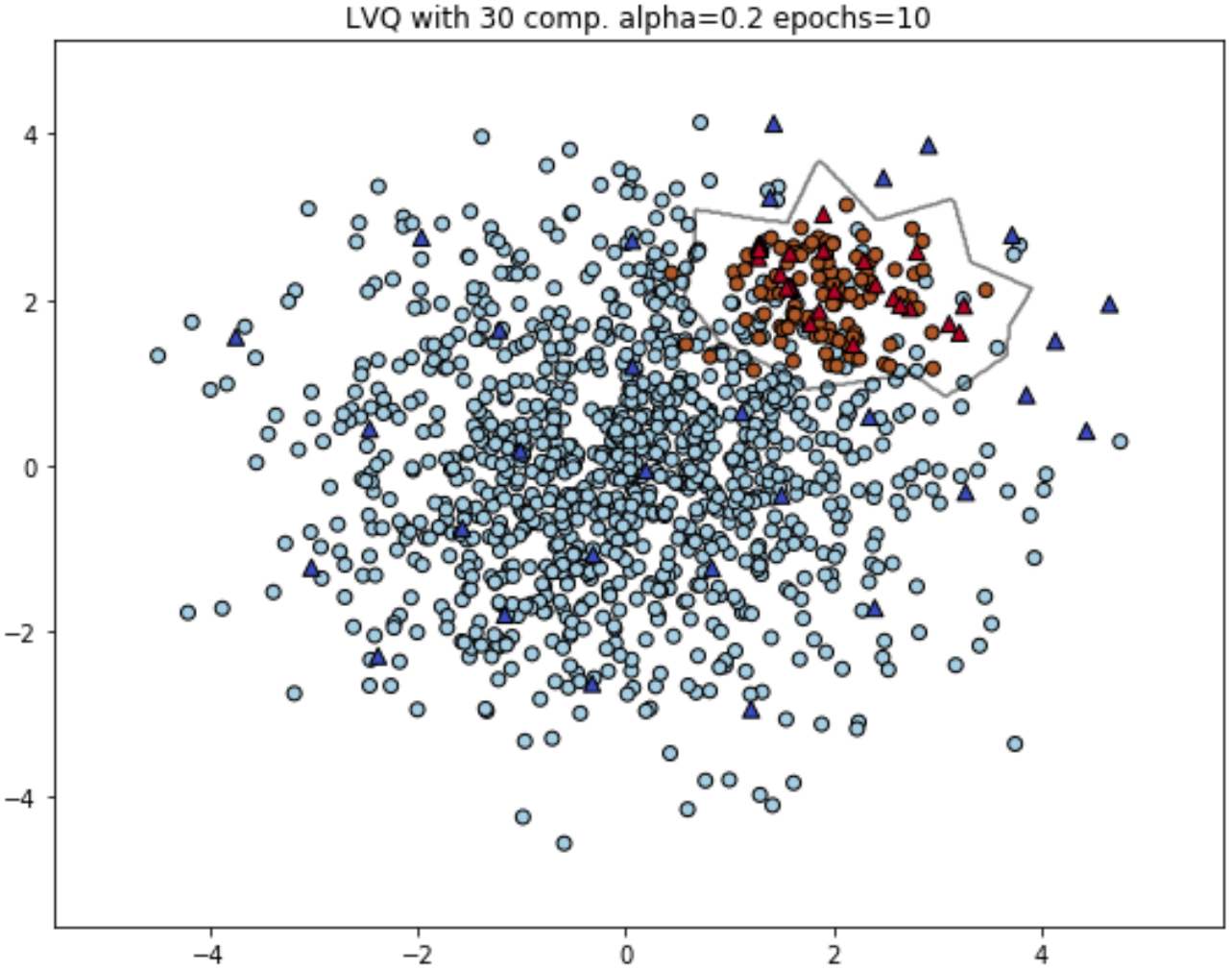
- See lvqclassification2_imbalance.py

```
#print(__doc__)

from LVQClassifier import LVQClassifier as LVQ
# we create clusters with 1000 and 100 points
rng = np.random.RandomState(0)
n_samples_1 = 1000
n_samples_2 = 100
X = np.r_[1.5 * rng.randn(n_samples_1, 2),
          0.5 * rng.randn(n_samples_2, 2) + [2, 2]]
y = [0] * (n_samples_1) + [1] * (n_samples_2)

# LVQ parameter
epochs = 10
# LVQ1, no bias correction, uniform random initial state
clf = LVQ(n_components=30,alpha=0.1,epochs=epochs,
          initial_state='Uniform',bias_decrease_rate=1.0)
clf.fit(X, y)
X_LVQ = clf.weights
y_LVQ = clf.label_weights
title = 'LVQ with 30 comp. alpha=0.2 epochs=10'
```

```
plt.figure(figsize=(9, 7))
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired, edgecolors='k')
plt.scatter(X_LVQ[:, 0], X_LVQ[:, 1], c=y_LVQ,
                cmap=plt.cm.coolwarm, s=50, marker='^', edgecolors='k')
plt.title(title)

# create grid to evaluate model
X0, X1 = X[:, 0], X[:, 1]
xx, yy = make_meshgrid(X0, X1)
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.contour(xx, yy, Z,  colors='k', levels=[0], alpha=0.5,
                linestyles=['-'])
plt.show()
```

Classification Using LVQ

- See lvqclassification2_imbalance.py



Exercise:

- Take a dataset
- Classify the dataset using Perceptron, MLP, and LVQ
- Conduct several experiments to explore parameters impact of each classifier to its performance
- Compare the performance of those classifier
- Analysis and Conclude your experiment

Attribution:

- Single Layer Perceptron

https://www.cs.tau.ac.il/~nin/Courses/NC05/SingLayerPerc.ppt

Multi Layer Perceptron

https://scikit-learn.org/stable/auto_examples/linear_model/plot_sgd_comparison.html#sphx-glr-auto-examples-linear-model-plot-sgd-comparison-py

LVQ

http://www.astrowing.eu/data-science/learning-vector-quantization

IKUTI KAMI

DIGITAL TALENT SCHOLARSHIP

digitalent.kominfo
digitalent.kominfo
DTS_kominfo
Digital Talent Scholarship 2019

Pusat Pengembangan Profesi dan Sertifikasi
Badan Penelitian dan Pengembangan SDM
Kementerian Komunikasi dan Informatika
Jl. Medan Merdeka Barat No. 9
(Gd. Belakang Lt. 4 - 5)
Jakarta Pusat, 10110