VIGNAN'S
Foundation for Science, Technology & Research
(Deemed to be University)
-Estd. u/s 3 of UGC Act 1956

Machine Learning

Unit – V

By
Mrs. P Jhansi Lakshmi
Assistant Professor
Department of CSE, VFSTR

Syllabus

UNIT – V

MULTILAYER PERCEPTRON: Understanding the brain; Neural networks as a paradigm for parallel processing; The perceptron; Training a perceptron; Learning boolean functions; Backpropagation algorithm; Nonlinear regression; Two class discrimination; Multiclass discrimination; Multiple hidden layers; Training procedures; Improving convergence; Overtraining; Structuring the network.

MULTILAYER PERCEPTRON

Multilayer Perceptron

- The multilayer perceptron is an artificial neural network structure and is a nonparametric estimator that can be used for classification and regression.
- Perceptron is one of the Artificial Neural Network models. It has took their inspiration from the brain.
- However, in engineering, our aim is not to understand the brain perse, but to build useful machines.
- We are interested in artificial neural networks because we believe that they may help us to build better computer systems.
- The brain is an information processing device that has some incredible abilities.

Understanding the Brain

- Levels of analysis (Marr, 1982)
  1. Computational theory
  2. Representation and algorithm
  3. Hardware implementation
- Reverse engineering: From hardware to theory
- Parallel processing: SIMD vs MIMD

  Neural net: SIMD with modifiable local memory

  Learning: Update by training/experience

Understanding the Brain

- Understanding an information processing system has three levels, called the levels of analysis:
  - Computational theory corresponds to the goal of computation and an abstract definition of the task.
  - Representation and algorithm is about how the input and the output are represented and about the specification of the algorithm for the transformation from the input to the output.
  - Hardware implementation is the actual physical realization of the system.

Machine Learning

Understanding the Brain

VIGNAN'S
Foundation for Science, Technology & Research
(Deemed to be UNIVERSITY)
-Estd. u/s 3 of UGC Act 1956

- One example is sorting: The computational theory is to order a given set of elements.
- The representation may use integers, and the algorithm may be Quicksort.
- After compilation, the executable code for a particular processor sorting integers represented in binary is one hardware implementation.

Understanding the Brain

- The idea is that for the same computational theory, there may be multiple representations and algorithms manipulating symbols in that representation.
- Similarly, for any given representation and algorithm, there may be multiple hardware implementations.
- We can use one of various sorting algorithms, and even the same algorithm can be compiled on computers with different processors and lead to different hardware implementations.

Understanding the Brain

The classic example is the difference between natural and artificial flying machines:

- A sparrow flaps its wings; a commercial airplane does not flap its wings but uses jet engines.
- The sparrow and the airplane are two hardware implementations built for different purposes, satisfying different constraints.
- But they both implement the same theory, which is aerodynamics.

Understanding the Brain

- Brain's abilities will look like the brain with networks of large numbers of processing units, until we discover the computational theory of intelligence.
- The brain is one hardware implementation for learning or pattern recognition.
- From this particular implementation, we can do reverse engineering and extract the representation and the algorithm used.
- From that, we can get the computational theory, we can then use another representation and algorithm, and in turn a hardware implementation more suited to the means and constraints we have.
- One hopes our implementation will be cheaper, faster, and more accurate.

Neural Networks as a Paradigm for Parallel Processing

There are mainly two paradigms for *parallel processing:*

- In single instruction, multiple data (SIMD) machines, all processors execute the same instruction but on different pieces of data.
- In multiple instruction, multiple data (MIMD) machines, different processors may execute different instructions on different data.
- SIMD machines are easier to program because there is only one program to write.
- MIMD machines are more general, but it is not an easy task to write separate programs for all the individual processors; additional problems are related to synchronization, data transfer between processors, and so forth.

Neural Networks as a Paradigm for Parallel Processing

- Assume we have simple processors with a small amount of local memory where some parameters can be stored.
- Each processor implements a fixed function and executes the same instructions as SIMD processors;
- But by loading different values into the local memory, they can be doing different things and the whole operation can be distributed over such processors.

Neural Networks as a Paradigm for Parallel Processing

- Then we need Neural Instruction, Multiple Data (NIMD) machines, where each processor corresponds to a neuron, local parameters correspond to its weights, and the whole structure is a neural network.
- If the function implemented in each processor is simple and if the local memory is small, then many such processors can be fit on a single chip.
- The problem now is to distribute a task over a network of such processors and to determine the local parameter values.
- This is where learning comes into play: We do not need to program such machines and determine the parameter values ourselves if such machines can learn from examples!

The Perceptron

- The *perceptron* is the symbol of ANN where several inputs determine together the output, through the combination of Linear and Activation model.

- It is the basic processing element. It has inputs that may come from the environment or may be the outputs of other perceptron's.

- Associated with each input, $x_j \in R, j = 1, \dots, d$ is a *connection weight*, or *synaptic weight* $w_j \in R,$ and the output, y, in the simplest case is a weighted sum of the inputs.

The Perceptron

$$y = \sum_{j=1}^{d} w_j x_j + w_0$$

- w_0 is the intercept value to make the model more general;
- it is generally modeled as the weight coming from an extra *bias unit*, x_0, which is always +1. We can write the output of the perceptron as a dot product.

$$y = w^T x$$

- Where $w = [w_0, w_1, \ldots, w_d]^T$ and $x = [1, x_1, \ldots, x_d]^T$ are *augmented vectors* to include the bias weight and input.

The Perceptron

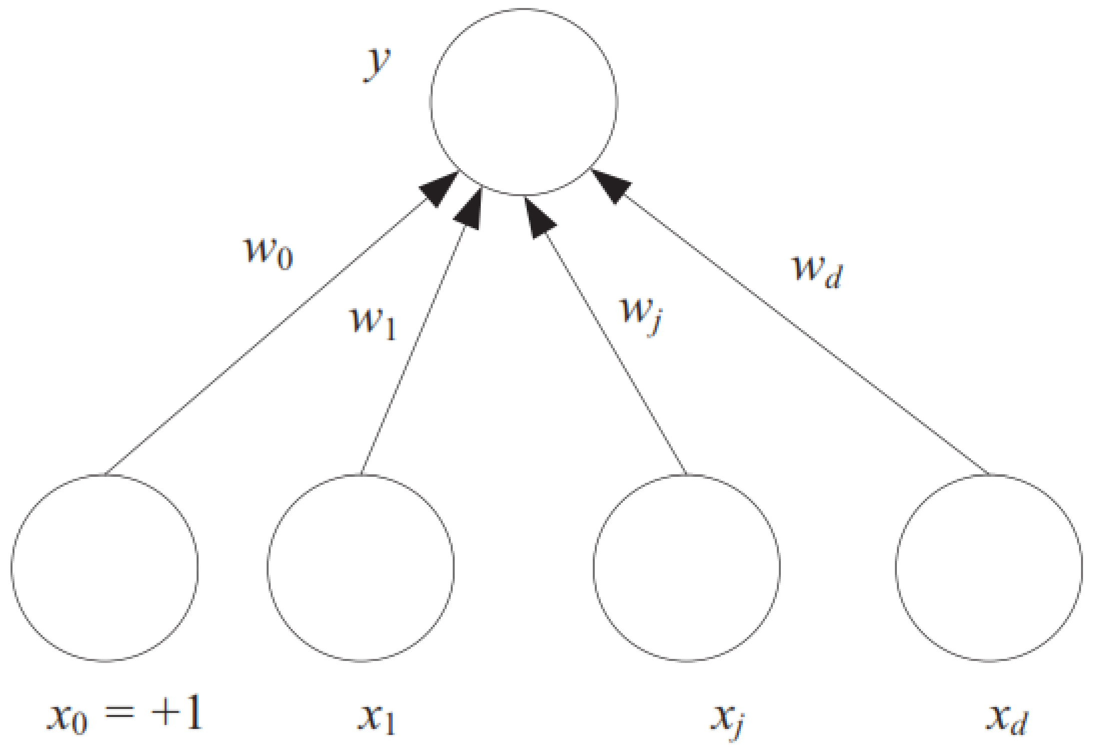
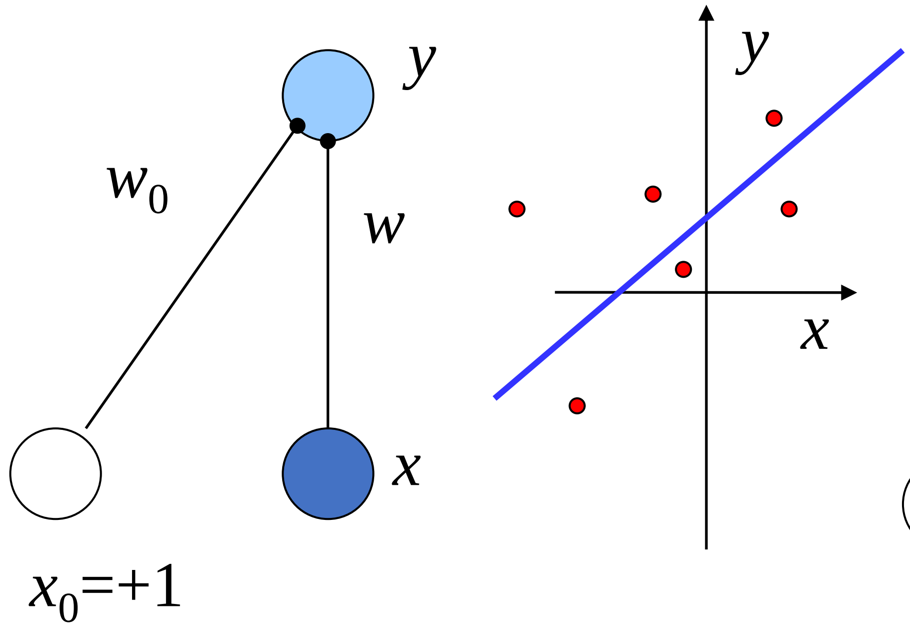
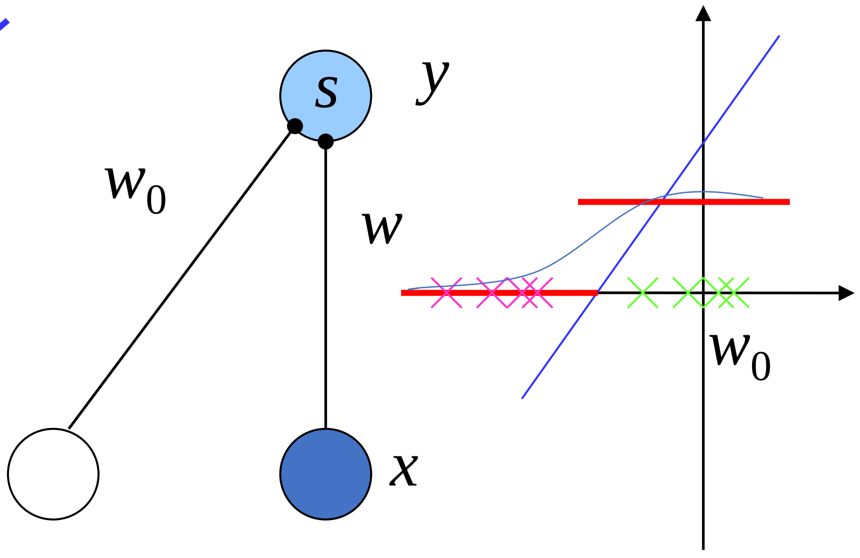


Figure 11.1 Simple perceptron. $x_j, j = 1, \ldots, d$ are the input units. x_0 is the bias unit that always has the value 1. y is the output unit. w_j is the weight of the directed connection from input x_j to the output.

What a Perceptron Does

- Regression: $y=wx+w_0$
- Classification: $y=1(wx+w_0>0)$





$$y = \text{sigmoid}(o) = \frac{1}{1+\exp\left[-\mathbf{w}^T\mathbf{x}\right]}$$

The Perceptron

- During testing, with given weights, w, for input x, we compute the output y.
- To implement a given task, we need to learn the weights w, the parameters of the system, such that correct outputs are generated given the inputs.
- When d = 1and x is fed from the environment through an input unit, we have

$$y = wx + w_0$$

which is the equation of a line with w as the slope and w_0 as the intercept.

- Thus this perceptron with one input and one output can be used to implement a linear fit.
- With more than one input, the line becomes a (hyper)plane, and the perceptron with more than one input can be used to implement multivariate linear fit.

- The perceptron defined in this equation $y = \sum_{j=1}^{d} w_j x_j + w_0$ defines a hyperplane.
- And it can be used to divide the input space into two: the half-space where it is positive and the half-space where it is negative .
- By using it to implement a linear discriminant function, the perceptron can separate two classes by checking the sign of the output.
- If we define s(·) as the threshold function

$$s(a) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases}$$

then we can

$$\text{choose} \begin{cases} C_1 & \text{if } s(\mathbf{w}^T\mathbf{x}) > 0 \\ C_2 & \text{otherwise} \end{cases}$$

K Outputs

Regression:

$$y_i = \sum_{j=1}^{d} w_{ij}x_j + w_{i0} = \mathbf{w}_i^T\mathbf{x}$$

$$\mathbf{y} = \mathbf{W}\mathbf{x}$$

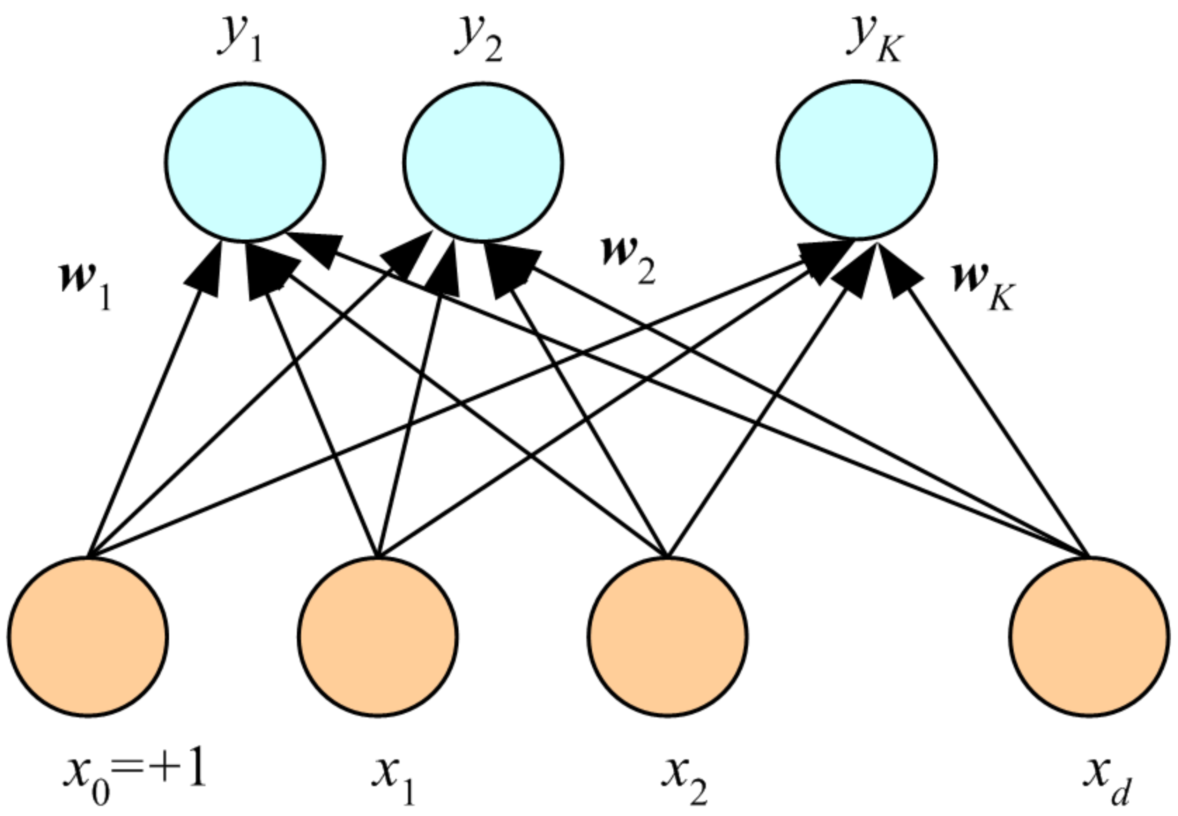
Classification:

$$o_i = \mathbf{w}_i^T\mathbf{x}$$

$$y_i = \frac{\exp o_i}{\sum_k \exp o_k}$$

choose C_i

if $y_i = \max_k y_k$



Training a Perceptron

- The strategy of training a perceptron is based on processing sample instances is one by one vs batch (whole sample) learning:
  - No need to store the whole sample
- Training strategy:
  - Start from random initial weights.
  - Update the weights in each iteration to minimize the error (use gradient descent if the error function is differentiable)
  - Repeat many times

Training a Perceptron

- The perceptron defines a hyperplane, and the neural network perceptron is just a way of implementing the hyperplane.
- Given a data sample, the weight can be calculated offline and then when they are plugged in, the perceptron can be used to calculate the output values.
- In training neural networks, we generally use online learning where we are not given the whole sample, but we are given instances one by one and would like the network to update its parameters after each instance, adapting itself slowly in time.

Training a Perceptron in Regression

- In online learning, we do not write the error function over the whole sample but on individual instances.
- Starting from random initial weights, at each iteration we adjust the parameters a little bit to minimize the error, without forgetting what we have previously learned.
- If this error function is differentiable, we can use gradient descent.
- For example, in regression the error on the single instance pair with index t, (x^t, r^t), is

$$E^t(\mathbf{w}|\mathbf{x}^t, r^t) = \frac{1}{2}(r^t - y^t)^2 = \frac{1}{2}[r^t - (\mathbf{w}^T\mathbf{x}^t)]^2$$

and for $j = 0, \ldots, d$, the online update is

$$\Delta w_j^t = \eta(r^t - y^t)x_j^t$$

Training a Perceptron in Classification

- Single sigmoid output

$$y^t = \text{sigmoid}\left(\mathbf{w}^T \mathbf{x}^t\right)$$

$$E^t(\mathbf{w}|\mathbf{x}^t, \mathbf{r}^t) = -r^t \log y^t - (1 - r^t)\log(1 - y^t)$$

$$\Delta w_j^t = \eta(r^t - y^t)x_j^t$$

- *K*>2 softmax outputs

$$y^t = \frac{\exp \mathbf{w}_i^T \mathbf{x}^t}{\sum_k \exp \mathbf{w}_k^T \mathbf{x}^t}$$

$$E^t(\{\mathbf{w}_i\}_i|\mathbf{x}^t, \mathbf{r}^t) = -\sum_i r_i^t \log y_i^t$$

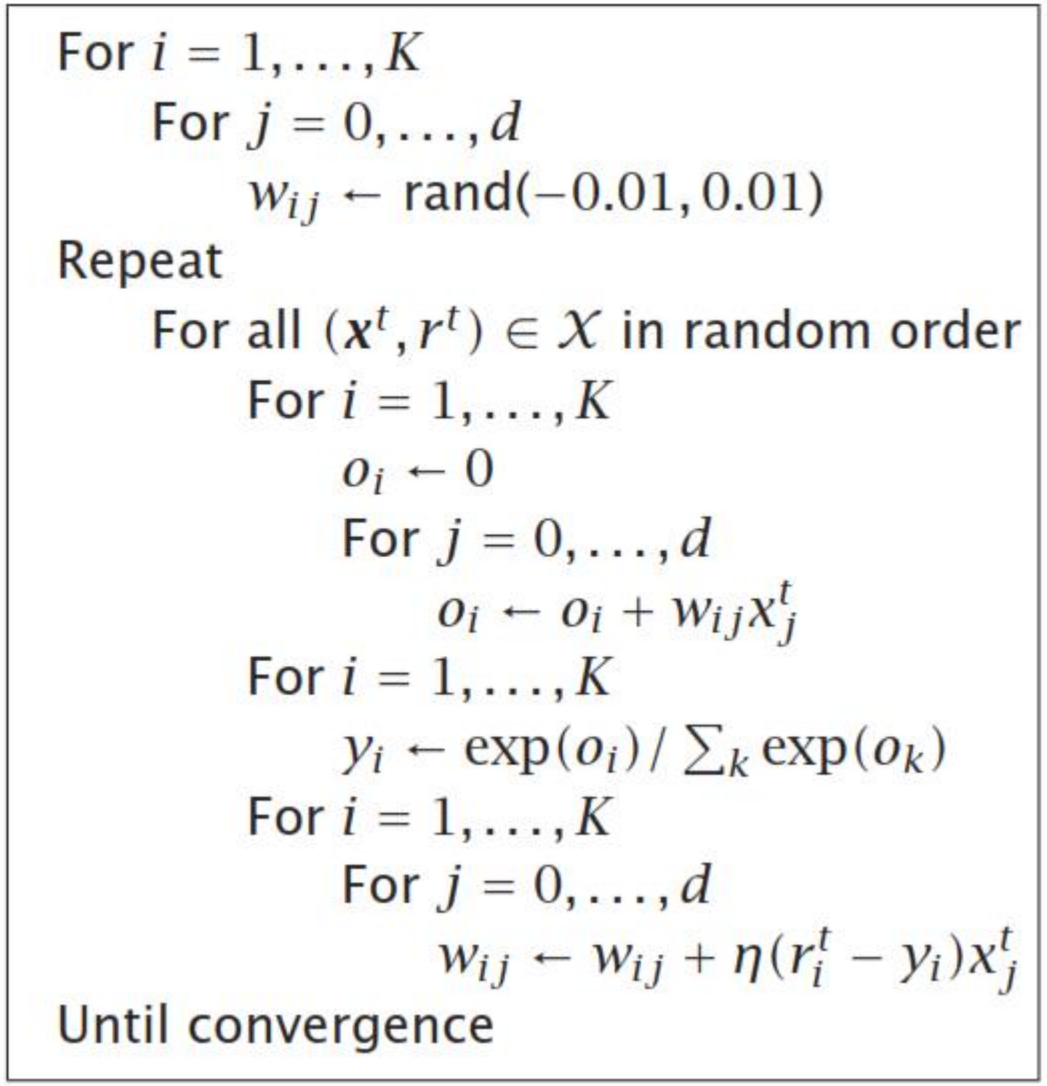
$$\Delta w_{ij}^t = \eta\left(r_i^t - y_i^t\right)x_j^t$$

$$\text{Update} = \text{LearningFactor} \cdot (\text{DesiredOutput} - \text{ActualOutput}) \cdot \text{Input}$$

Perceptron Training Algorithm

- Perceptron training algorithm implementing stochastic online gradient descent for the case with K>2 classes.

- This is the online version of the algorithm

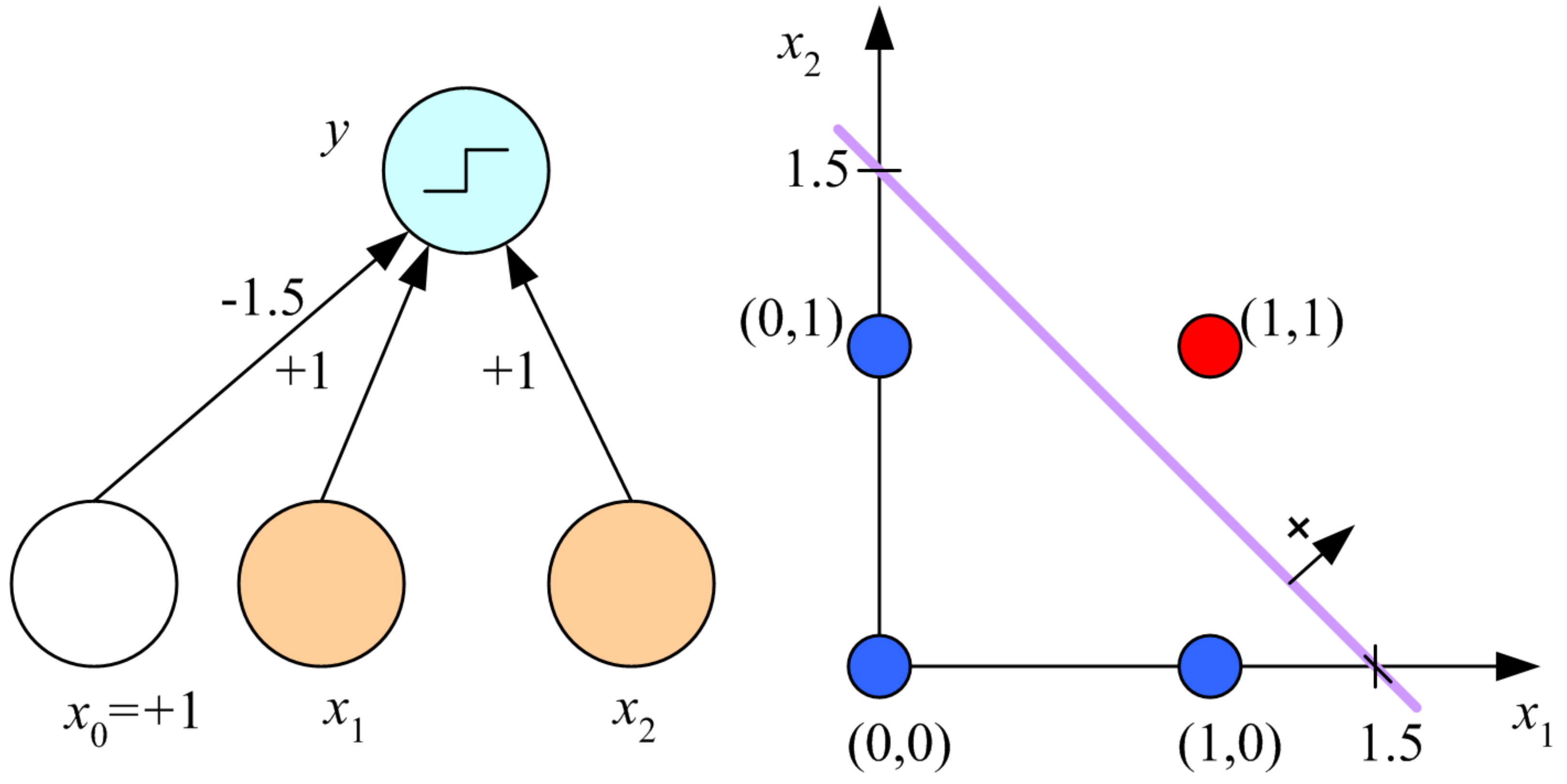


```
For $i = 1, \ldots, K$
    For $j = 0, \ldots, d$
        $w_{ij} \leftarrow \text{rand}(-0.01, 0.01)$
Repeat
    For all $(\mathbf{x}^t, r^t) \in \mathcal{X}$ in random order
        For $i = 1, \ldots, K$
            $o_i \leftarrow 0$
            For $j = 0, \ldots, d$
                $o_i \leftarrow o_i + w_{ij} x_j^t$
        For $i = 1, \ldots, K$
            $y_i \leftarrow \exp(o_i) / \sum_k \exp(o_k)$
        For $i = 1, \ldots, K$
            For $j = 0, \ldots, d$
                $w_{ij} \leftarrow w_{ij} + \eta(r_i^t - y_i) x_j^t$
Until convergence
```

Learning Boolean Functions

- In a Boolean function, the inputs are binary and the output is 1 if the corresponding function value is true and 0 otherwise.
- Therefore, it can be seen as a two-class classification problem. As an example, for learning to AND two inputs, the table of inputs and required outputs is given.
- An example of a perceptron that implements AND and its geometric interpretation in two dimensions is given in figure.
- The discriminant is $y = S(x_1 + x_2 - 1 \cdot 5)$

Learning Boolean AND



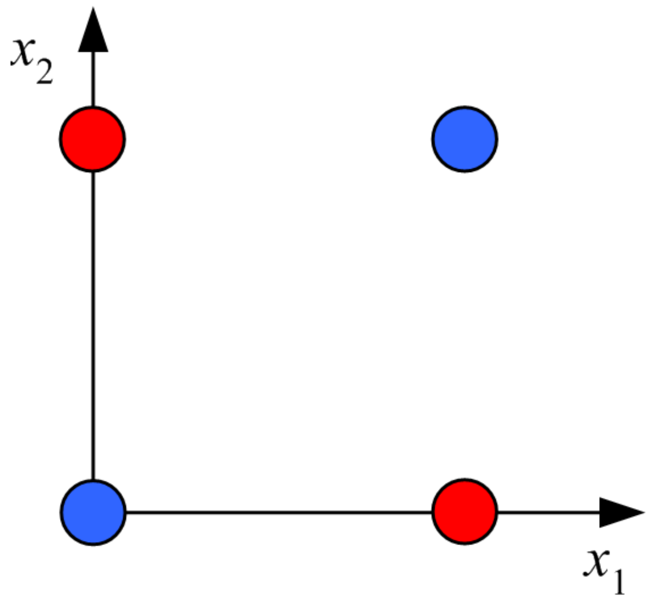
x_1	x_2	r
0	0	0
0	1	0
1	0	0
1	1	1

Learning Boolean XOR

- No w_0, w_1, w_2 satisfy:

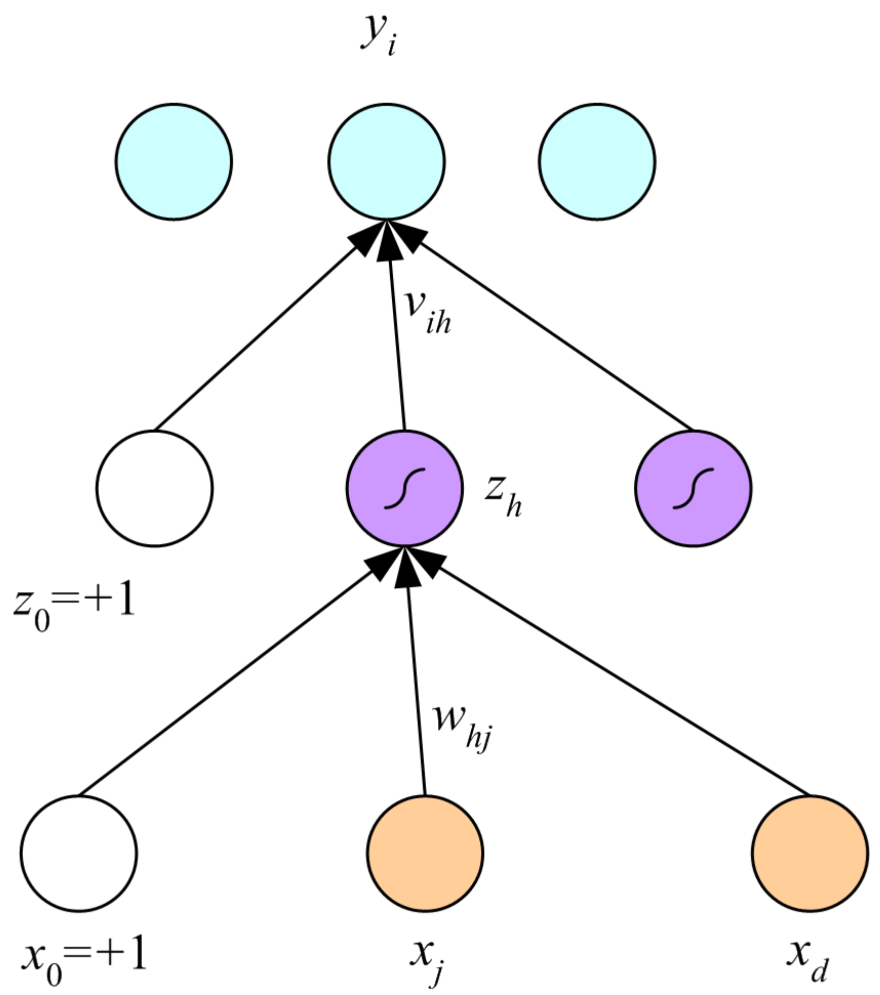
$$\begin{aligned} & & w_0 &\leq 0 \\ & w_2 + & w_0 &> 0 \\ w_1 + & & w_0 &> 0 \\ w_1 + & w_2 + & w_0 &\leq 0 \end{aligned}$$

x_1	x_2	r
0	0	0
0	1	1
1	0	1
1	1	0



(Minsky and Papert, 1969)

Backpropagation Algorithm



$$y_i = \mathbf{v}_i^T \mathbf{z} = \sum_{h=1}^{H} v_{ih} z_h + v_{i0}$$

$$z_h = \text{sigmoid}\left(\mathbf{w}_h^T \mathbf{x}\right)$$

$$= \frac{1}{1+\exp\left[-\left(\sum_{j=1}^{d} w_{hj} x_j + w_{h0}\right)\right]}$$

$$\boxed{\frac{\partial E}{\partial w_{hj}} = \frac{\partial E}{\partial y_i} \frac{\partial y_i}{\partial z_h} \frac{\partial z_h}{\partial w_{hj}}}$$

Regression

$$E(\mathbf{W},\mathbf{v}\,|\,\mathcal{X})=\frac{1}{2}\sum_t\left(r^t-y^t\right)^2$$

$$\boxed{\Delta v_h=\sum_t\left(r^t-y^t\right)z_h^t}$$

Backward

$$\boxed{\begin{aligned}\Delta w_{hj}&=-\eta\frac{\partial E}{\partial w_{hj}}\\&=-\eta\sum_t\frac{\partial E}{\partial y^t}\frac{\partial y^t}{\partial z_h^t}\frac{\partial z_h^t}{\partial w_{hj}}\\&=-\eta\sum_t-\left(r^t-y^t\right)v_h\boxed{z_h^t\left(1-z_h^t\right)}x_j^t\\&=\eta\sum_t\left(r^t-y^t\right)v_h z_h^t\left(1-z_h^t\right)x_j^t\end{aligned}}$$

$$y^t=\sum_{h=1}^{H}v_h z_h^t+v_0$$

Forward

$$z_h=\boxed{\text{sigmoid}}\left(\mathbf{w}_h^T\mathbf{x}\right)$$

$$\boldsymbol{x}$$

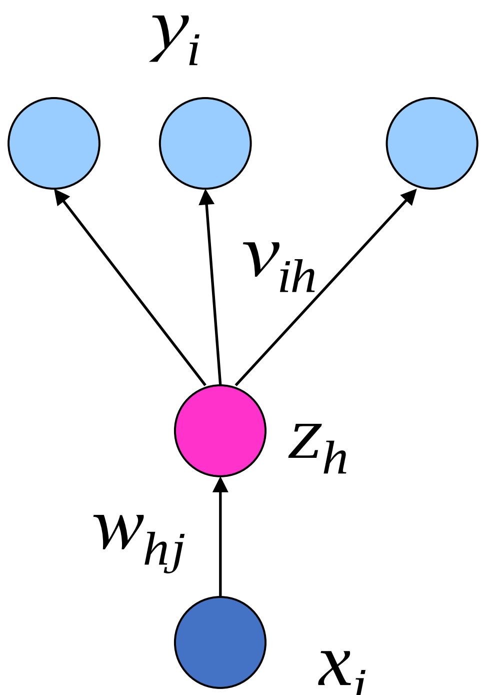
Regression with Multiple Outputs

$$E(\mathbf{W},\mathbf{V}\mid\mathcal{X})=\frac{1}{2}\sum_t\sum_i\left(r_i^t-y_i^t\right)^2$$

$$y_i^t=\sum_{h=1}^{H}v_{ih}z_h^t+v_{i0}$$

$$\Delta v_{ih}=\eta\sum_t\left(r_i^t-y_i^t\right)z_h^t$$

$$\Delta w_{hj}=\eta\sum_t\left[\sum_i\left(r_i^t-y_i^t\right)v_{ih}\right]z_h^t\left(1-z_h^t\right)x_j^t$$



```
Initialize all $v_{ih}$ and $w_{hj}$ to rand$(-0.01, 0.01)$
Repeat
    For all $(\boldsymbol{x}^t, r^t) \in \mathcal{X}$ in random order
        For $h = 1, \ldots, H$
            $z_h \leftarrow \mathrm{sigmoid}(\boldsymbol{w}_h^T \boldsymbol{x}^t)$
        For $i = 1, \ldots, K$
            $y_i = \boldsymbol{v}_i^T \boldsymbol{z}$
        For $i = 1, \ldots, K$
            $\Delta \boldsymbol{v}_i = \eta(r_i^t - y_i^t)\boldsymbol{z}$
        For $h = 1, \ldots, H$
            $\Delta \boldsymbol{w}_h = \eta(\sum_i (r_i^t - y_i^t) v_{ih}) z_h (1 - z_h) \boldsymbol{x}^t$
        For $i = 1, \ldots, K$
            $\boldsymbol{v}_i \leftarrow \boldsymbol{v}_i + \Delta \boldsymbol{v}_i$
        For $h = 1, \ldots, H$
            $\boldsymbol{w}_h \leftarrow \boldsymbol{w}_h + \Delta \boldsymbol{w}_h$
Until convergence
```

Two-Class Discrimination

- One sigmoid output y^t for $P(C_1|\boldsymbol{x}^t)$ and

  $P(C_2|\boldsymbol{x}^t) \equiv 1\text{-}y^t$

$$y^t = \text{sigmoid}\left(\sum_{h=1}^{H} v_h z_h^t + v_0\right)$$

$$E(\mathbf{W},\mathbf{v}\,|\,\mathcal{X}) = -\sum_t r^t \log y^t + \left(1-r^t\right)\log\left(1-y^t\right)$$

$$\Delta v_h = \eta \sum_t \left(r^t - y^t\right) z_h^t$$

$$\Delta w_{hj} = \eta \sum_t \left(r^t - y^t\right) v_h z_h^t \left(1-z_h^t\right) x_j^t$$

K>2 Classes

$$o_i^t = \sum_{h=1}^{H} v_{ih} z_h^t + v_{i0} \qquad y_i^t = \frac{\exp o_i^t}{\sum_k \exp o_k^t} \equiv P\left(C_i \mid \mathbf{x}^t\right)$$

$$E\left(\mathbf{W}, \mathbf{v} \mid \mathcal{X}\right) = -\sum_t \sum_i r_i^t \log y_i^t$$

$$\Delta v_{ih} = \eta \sum_t \left(r_i^t - y_i^t\right) z_h^t$$

$$\Delta w_{hj} = \eta \sum_t \left[\sum_i \left(r_i^t - y_i^t\right) v_{ih} \right] z_h^t \left(1 - z_h^t\right) x_j^t$$

Multiple Hidden Layers

- MLP with one hidden layer is a universal approximator (Hornik et al., 1989), but using multiple layers may lead to simpler networks

$$z_{1h} = \text{sigmoid}\left(\mathbf{w}_{1h}^T \mathbf{x}\right) = \text{sigmoid}\left(\sum_{j=1}^{d} w_{1hj} x_j + w_{1h0}\right), h = 1, \dots, H_1$$

$$z_{2l} = \text{sigmoid}\left(\mathbf{w}_{2l}^T \mathbf{z}_1\right) = \text{sigmoid}\left(\sum_{h=1}^{H_1} w_{2lh} z_{1h} + w_{2l0}\right), l = 1, \dots, H_2$$

$$y = \mathbf{v}^T \mathbf{z}_2 = \sum_{l=1}^{H_2} v_l z_{2l} + v_0$$

Training Procedures

Improving Convergence

- Gradient descent has various advantages. It is simple. The change in a weight uses only the values of the presynaptic and postsynaptic units and the error.
- When online training is used, it does not need to store the training set.
- Because of these reasons, it can be implemented in hardware. But by itself, gradient descent converges slowly.
- However, there are two frequently used simple techniques that improve the performance of the gradient descent considerably, making gradient-based methods feasible in real applications:
  - *Momentum*
  - *Adaptive Learning Rate*

Improving Convergence

Momentum:

- Let us say w_i is any weight in a multilayer perceptron in any layer, including the biases.
- At each parameter update, successive Δw_i^t values may be so different that large oscillations may occur and slow convergence.
- The idea is to take a running average by incorporating the previous update in the current change as if there is a *momentum* due to previous updates:

$$\Delta w_i^t = -\eta \frac{\partial E^t}{\partial w_i} + \alpha \Delta w_i^{t-1}$$

Improving Convergence

- α is generally taken between 0.5 and 1.0.
- This approach is especially useful when online learning is used, where as a result we get an effect of averaging and smooth the trajectory during convergence.
- The disadvantage is that the past Δw_i^{t-1} values should be stored in extra memory.

Improving Convergence

Adaptive Learning Rate

- In gradient descent, the learning factor η determines the magnitude of change to be made in the parameter.
- It is generally taken between 0.0 and 1.0, mostly less than or equal to 0.2.
- It can be made adaptive for faster convergence, where it is kept large when learning takes place and is decreased when learning slows down:

$$\Delta\eta = \begin{cases} +a & \text{if } E^{t+\tau} < E^{t} \\ -b\eta & \text{otherwise} \end{cases}$$

Improving Convergence

- Thus we increase η by a constant amount if the error on the training set decreases and decrease it geometrically if it increases.
- Because E may oscillate from one epoch to another, it is a better idea to take the average of the past few epochs as E^t

Overtraining

- A multilayer perceptron with d inputs, H hidden units, and K outputs has H(d+1) weights in the first layer and K(H+1) weights in the second layer.
- Both the space and time complexity of an MLP is $O(H \cdot (K + d))$.
- When e denotes the number of training epochs, training time complexity is $O(e \cdot H \cdot (K + d))$.
- In an application, d and K are predefined and H is the parameter that we play with to tune the complexity of the model.

Overtraining

- we have previously seen this phenomenon in the case of polynomial regression where we noticed that in the presence of noise or small samples, increasing the polynomial order leads to worse generalization.
- Similarly in an MLP, when the number of hidden units is large, the generalization accuracy deteriorates and the bias/variance dilemma also holds for the MLP.
- A similar behavior happens when training is continued too long: As more training epochs are made, the error on the training set decreases, but the error on the validation set starts to increase beyond a certain point .

Overtraining

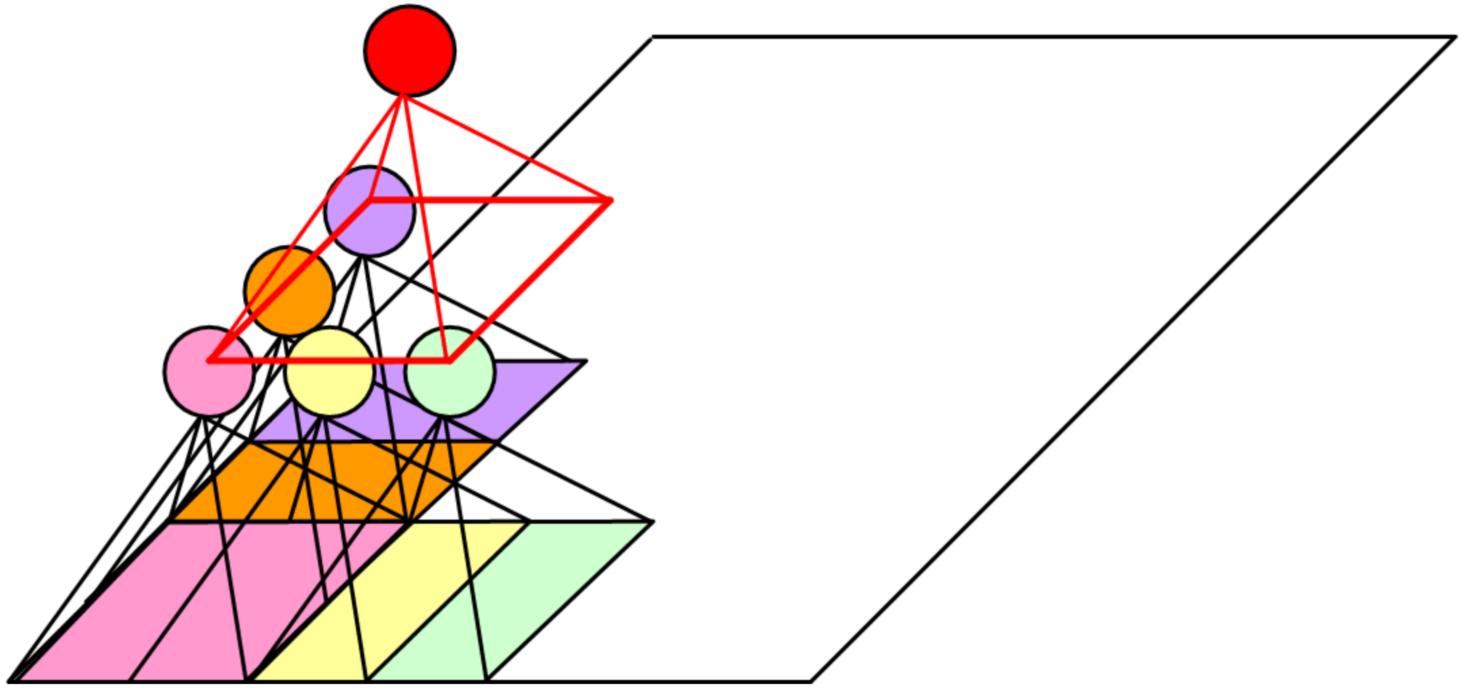
- Thus as training continues, if new parameters are added to the system, increasing the complexity and leading to poor generalization.
- Learning should be stopped early to alleviate this problem of overtraining.
- The optimal point to stop training, and the optimal number of hidden units, is determined through cross-validation, which involves testing the network's performance on validation data unseen during training.

Structuring the Network

- In some applications, we may believe that the input has a local structure.
- For example, in vision we know that nearby pixels are correlated and there are local features like edges and corners;
- Similarly, in speech, locality is in time and inputs close in time can be grouped as speech primitives.
- By combining these primitives, longer utterances, for example, speech phonemes, may be defined.
- In such a case when designing the MLP, hidden units are not connected to all input units because not all inputs are correlated.
- Instead, we define hidden units that define a window over the input space and are connected to only a small local subset of the inputs.
- This decreases the number of connections and therefore the number of free parameters

Structured MLP

- Convolutional networks (Deep learning)



Weight Sharing

- In weight sharing, different units have connections to different
- inputs but share the same weight value (denoted by line type). Only one set of
- units is shown; there should be multiple sets of units, each checking for different
- features.