```
{
    Arguments: nn: learning matrix; in: input training data; out: output training data;
    ni: number of inputs; no: number of outputs; lrate: learning tax;
    af: activation function; af': derivative of activation function;
    oaf: output activation function; oaf': derivative of output activation function.
}
function Learn(nn, in, out, ni, no, lrate, af, af', oaf, oaf')
begin
    dim_nn = dim(nn)
    dim_i = dim_nn[0]
    dim_j = dim_nn[1]
    first_out = dim_j - 1 - no

    for i = 0 to dim_i - 2 step 1 do {Clear inputs and outputs.}
    begin
        nn[0, i] = 0.0
        nn[i, 0] = 0.0
        nn[i, dim_j - 1] = 0.0
        nn[dim_i - 1, i] = 0.0
    end

    for j = 0 to ni - 1 step 1 do {Assign inputs.}
        nn[j + 1, 0] = in[j]

    for j = ni + 1 to dim_j - 2 step 1 do {Calculate the neurons output.}
    begin
        nn[0, j] = 0.0
        for i = 1 to dim_i - 2 step 1 do {Weighted sums.}
            if i < j then
                if nn[i, j] != 0 then {x = x1 * w1 + x2 * w2 + …}
                    nn[0, j] = nn[0, j] + nn[i, j] * nn[i, 0]
            else if i == j then
                if nn[i, j] != 0 then
                    nn[0, j] = nn[0, j] + nn[i, j]
            else
                break
        if j < first_out then {Activation function.}
        begin
            nn[j, 0] = @af(nn[0, j]) {Calculate y = f(x).}
            nn[j, dim_j - 1] = @af'(nn[0, j]) {Calculate df(x)/dx}
        end
        else {Activation function for the output layer.}
        begin
            nn[j, 0] = @oaf(nn[0, j]) {Calculate y = f(x).}
            nn[j, dim_j - 1] = @oaf'(nn[0, j]) {Calculate df(x)/dx}
        end
    end

    for i = 0 to no - 1 step 1 do {Calculate delta for the output neurons.}
        nn[dim_i - 1, first_out + i] = out[i] - nn[first_out + i, 0] {d = z - y}

    for j = dim_j - 2 to j ni + 1 step - 1 do {Calculate delta for hidden neurons.}
        for i = ni + 1 to i dim_i - 2 - no step 1 do
        begin
            if i == j then
                break
            if nn[i, j] != 0 then {d1 = w1 * d2 + w2 * d2 + ...}
                nn[dim_i - 1, i] = nn[dim_i - 1, i] + nn[i, j] * nn[dim_i - 1, j]
        end

    for j = no + 1 to dim_j - 2 step 1 do {Adjust weights.}
        for i = 1 to dim_i - 2 - no step 1 do
            if i < j then
                if nn[i, j] != 0 then {w1 = w1 + n * d * df(x)/dx * x1}
                    nn[i, j] = nn[i, j] + lrate * nn[dim_i - 1, j] * nn[j, dim_j - 1] * nn[i, 0]
            else if i == j then {Biases.}
                if nn[i, j] != 0 then
                    nn[i, j] = nn[i, j] + lrate * nn[dim_i - 1, j] * nn[j, dim_j - 1] * 1.0
            else
                break

    return nn
end
```

```
{
    Arguments: nn: learning matrix; in: input data;
    ni: number of inputs; no: number of outputs;
    af: activation function; oaf: output activation function; of: output function.
}
function Process(nn, in, ni, no, af, oaf, of)
begin
    dim_nn = dim(nn)
    dim_i = dim_nn[0]
    dim_j = dim_nn[1]
    first_out = dim_j - 1 - no

    for i = 0 to dim_i - 2 step 1 do {Clear inputs and outputs.}
    begin
        nn[0, i] = 0.0
        nn[i, 0] = 0.0
        nn[i, dim_j - 1] = 0.0
        nn[dim_i - 1, i] = 0.0
    end

    for j = 0 to ni - 1 step 1 do {Assign inputs.}
        nn[j + 1, 0] = in[j]

    for j = ni + 1 to dim_j - 2 step 1 do {Calculate the neurons output.}
    begin
        nn[0, j] = 0.0
        for i = 1 to dim_i - 2 step 1 do {Weighted sums.}
            if i < j then
                if nn[i, j] != 0 then {x = x1 * w1 + x2 * w2 + …}
                    nn[0, j] = nn[0, j] + nn[i, j] * nn[i, 0]
            else if i == j then
                if nn[i, j] != 0 then
                    nn[0, j] = nn[0, j] + nn[i, j]
            else
                break
        if j < first_out then {Activation function.}
        begin
            nn[j, 0] = @af(nn[0, j]) {Calculate y = f(x).}
        end
        else {Activation function for the output layer.}
        begin
            nn[j, 0] = @oaf(nn[0, j]) {Calculate y = f(x).}
        end
    end

    for i = 0 to no - 1 step 1 do {Set the output matrix.}
        out[i] = @of(nn[first_out + i, 0])

    return out
end
```