MULTI-LAYER PERCEPTRON (MLP) VIA VHDL

ECE 475: Computer Hardware Design – Final Project

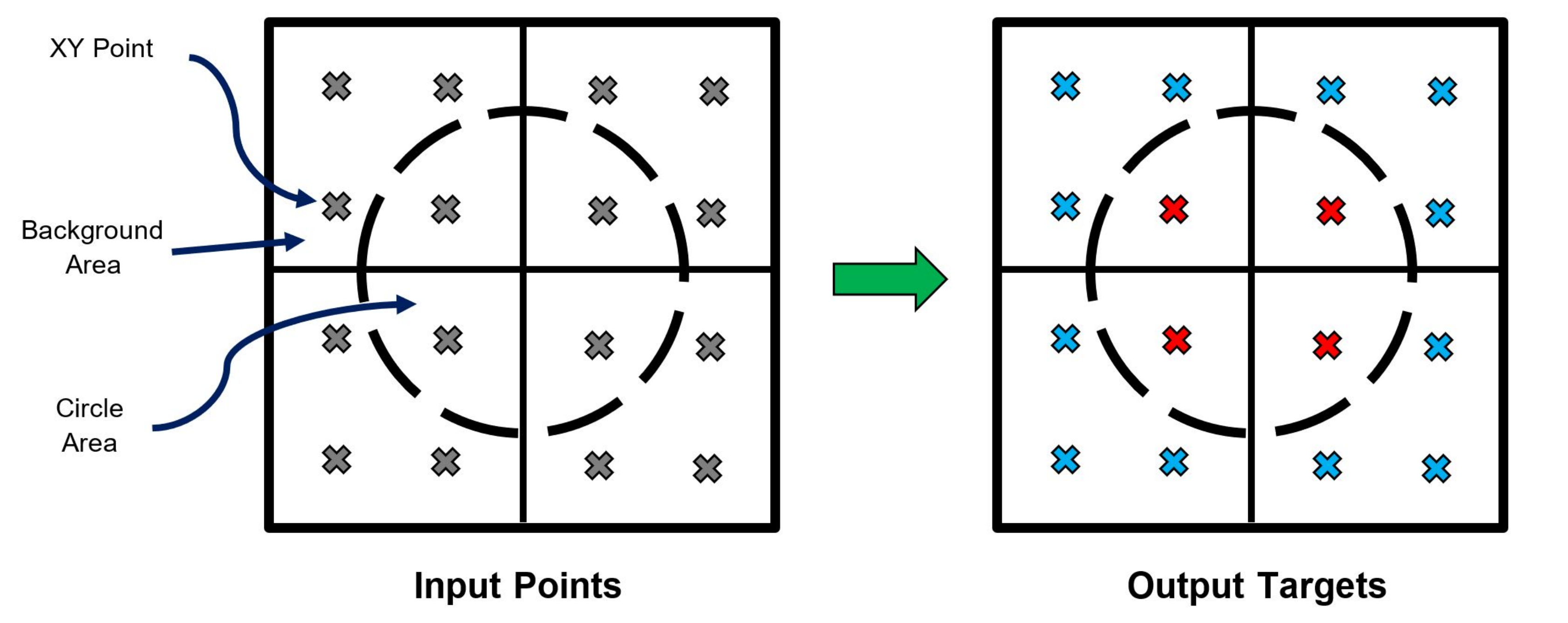
University of Michigan – Dearborn

Roderick L. Renwick

April 17, 2020

CONCEPT

- Use VHDL to create an MLP capable of learning the shape of a circle
- Given a set of coordinate points on an xy-plane, produce a set of binary classification values, corresponding to whether or not a point's location is within the area of a circle
- Evaluation will be based on a set of uniformly spaced coordinate points, where the prediction of '1' corresponds to a point inside the circle and '0' corresponds to a point outside the circle



REQUIREMENTS: FUNCTIONAL

For training, the system shall:

- Accept an input vector consisting of <u>x-y coordinate point pairs</u>
- Compute a corresponding vector of output targets
- Initialize <u>randomized</u> weight matrices
- Train for a given amount of epochs

For testing, the system shall:

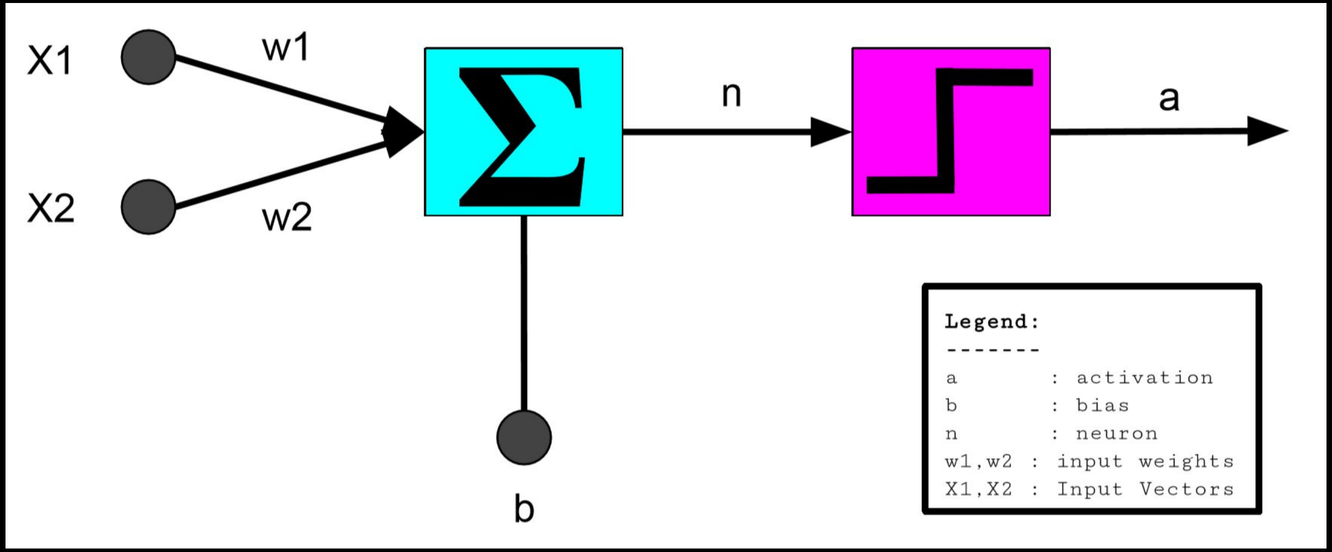
- Accept an input vector consisting of <u>uniformly spaced</u> x-y coordinate point pairs
- Compute a corresponding vector of output targets
- Load the <u>previously trained</u> weight matrices
- Produce circle prediction results

REQUIREMENTS: NONFUNCTIONAL

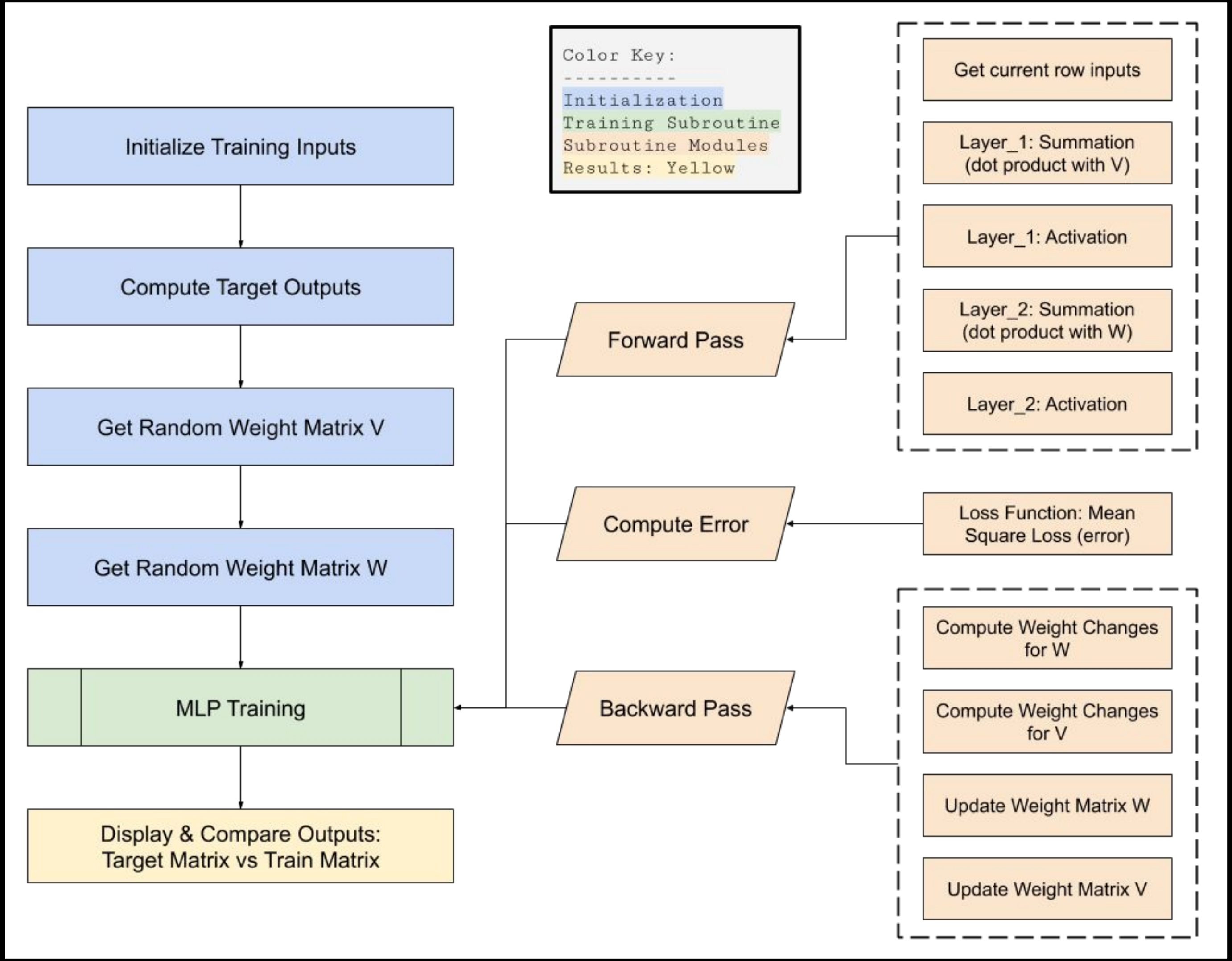
- Develop proof of concept trial in Python, for easy simulation, visualization and debugging tools
- Re-construct program in VHDL and use resulting waveforms to visualize the graph and gauge it's performance

DESIGN: MLP ARCHITECTURE

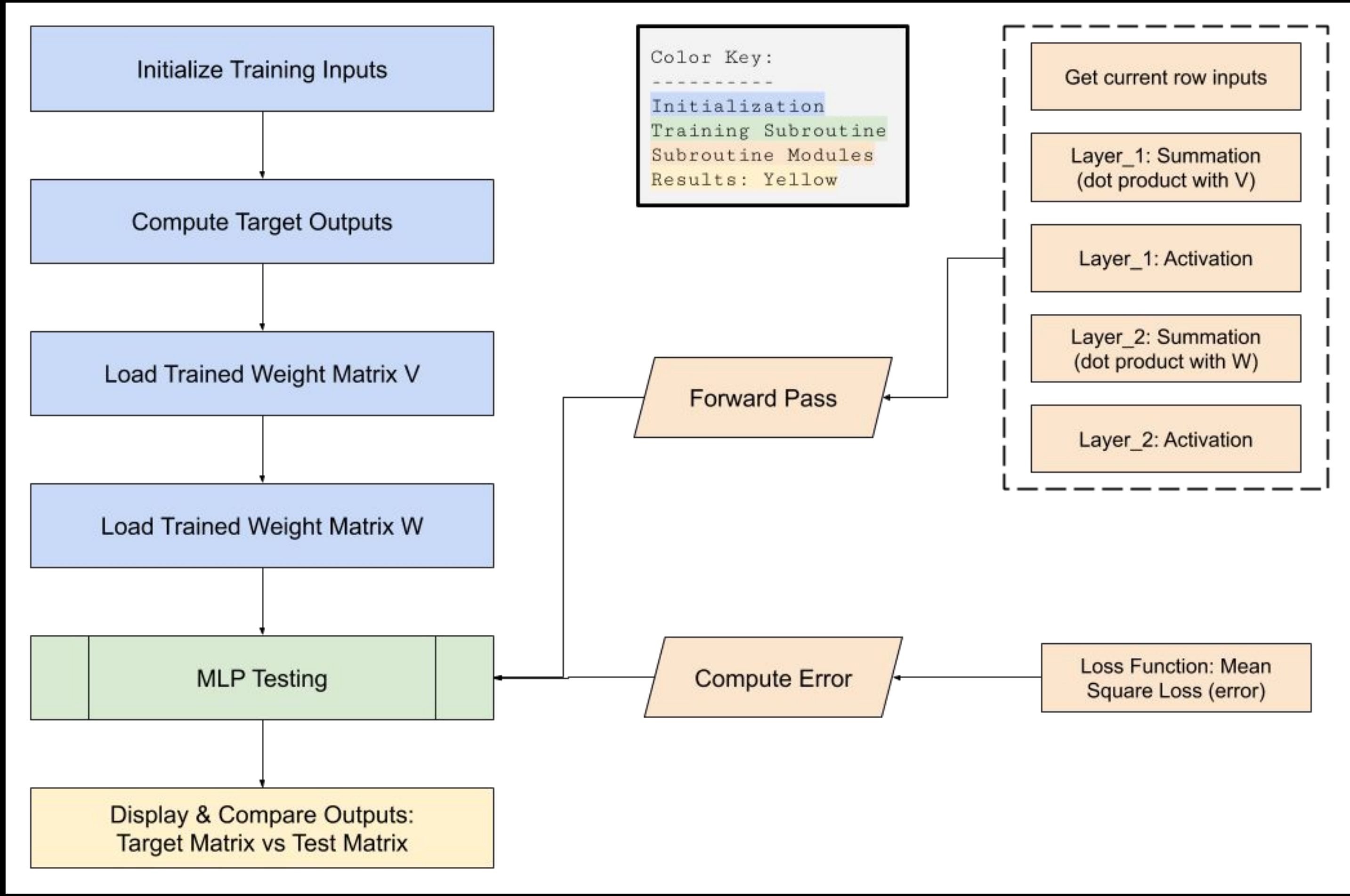
- MLP consists of two layers: one hidden layer, and one output layer
  - All neurons in each layer receive a weighted summation vector, produced from the dot product of the inputs against the weights
  - Each neuron is then passed to the sigmoid function, to determine its activation state
- The final result is a classification value of '1' or '0'



DESIGN: TRAINING



DESIGN: TESTING



IMPLEMENTATION: EQUATIONS

- **Sigmoid Activation:** $y = \frac{1}{1+ e^{-x}}$
- **Change Factor:** $\delta_k = (d_k - y_k) \, y_k (1 - y_k)$ *for k = 1, 2, ..., K*
- **Weight Matrix W:** $\Delta W_{kj} = \rho \, \delta_k \, z_j$ *for j = 1, 2, ..., J*
- **Weight Matrix V:** $\Delta V_{ji} = \rho \, z_j \left(1 - z_j \right) x_j \sum_{k=1}^{K} (\delta_k \, w_{kj})$ *for j = 1, 2, ..., J & i = 1, 2, ..., n*
- **Updated Weight Matrix W:** $\Delta W_{kj}^{t+1} = W_{kj}^{t} + \Delta W_{kj}$ *for j = 1, 2, ..., J*
- **Updated Weight Matrix V:** $\Delta V_{ji}^{t+1} = V_{kj}^{t} + \Delta V_{ji}$ *for j = 1, 2, ..., J & i = 1, 2, ..., n*

Note:
-- The learning rate value is ρ
-- The gradient of error value is (d – y) , i.e., (target_value – predicted_value)

IMPLEMENTATION: PYTHON

Training Loop

```
# GET TRAINING PATTERN VALUES
X_train = trainingInputs( X_train ) # random points in [-1,1]x[-1,1]
D_train = targetOutputs( D_train, X_train, area = 2.0 )

# INITIALIZE WEIGHT MATRICIES
V = randomMatrix( V, -10.0, 10.0) # Weight matrix to the hidden layer
W = randomMatrix( W, -10.0, 10.0) # Weight matrix to the output layer

new_lr = 0.5

for epoch in range( EPOCHS ):

    Error = 0.0
    for m in range( M ):

        # FORWARD PASS

        x = X_train[ m ][ : ]                              # Get mth row of X
        d = D_train[ m ][ : ]                              # Get mth row of D

        h = dotProduct( V, x )                             # Get weighted sums of hidden layer
        z = sigmoidActivation( h )                         # Get weighted outputs of hidden layer

        o = dotProduct( W, z )                             # Get weighted sums of output layer
        y = sigmoidActivation( o )

        Y_train[ m ] = y

        # BACKWARD PASS

        E = ( d - y )                                      # Gradient of error
        delta = E * y * ( 1 - y )                          # Change factor (delta) at output layer
        Error = Error + ( E * E ) / 2                      # Actual error: mean square loss

        delta_W = weightChanges_W( delta, z, new_lr )      # Compute weight changes of W
        delta_V = weightChanges_V( W, delta, z, x, new_lr ) # Compute weight changes of V

        W = updateWeights( delta_W, W )                    # Compute weight update of W
        V = updateWeights( delta_V, V )                    # Compute weight update of V
```

Testing Loop

```
# GET TESTING PATTERN VALUES
X_test = testingInputs( X_test )
D_test = targetOutputs( D_test, X_test, area = 2.0 )

Error = 0.0
for m in range( MTEST ):

    x = X_test[ m ][ : ]                               # Get mth row of X
    d = D_test[ m ][ : ]                               # Get mth row of D

    h = dotProduct( V, x )                             # Get weighted sums of hidden layer
    z = sigmoidActivation( h )                         # Get weighted outputs of hidden layer

    o = dotProduct( W, z )                             # Get weighted sums of output layer
    y = sigmoidActivation( o )                         # Get weighted outputs of output layer

    Error = Error + ( d - y ) * ( d - y ) / 2  # Actual error: mean square loss
    Y_test[ m ] = y
```

IMPLEMENTATION: VHDL

Initialization

Library Imports

```
library work;
library IEEE;
use work.functions.all;
use IEEE.NUMERIC_STD.all;
use IEEE.STD_LOGIC_1164.ALL;
```

Matrix Signal Declarations

```
-- WEIGHT MATRICES
   signal V_WEIGHT : matrix_JI;
   signal W_WEIGHT : matrix_KJ;

   -- TRAIN MATRICES
   signal X_TRAIN : matrix_MI;
   signal D_TRAIN : matrix_MK;
   signal Y_TRAIN : matrix_MK;

   -- TEST MATRICES
   signal X_TEST : matrix_MI;
   signal D_TEST : matrix_MK;
   signal Y_TEST : matrix_MK;
```

Initialization Process

```
INIT : process
    begin
        wait on go;

        X_TRAIN <= setTrainingInputs;
        wait on X_TRAIN;
        D_TRAIN <= targetOutputs( X_TRAIN );
        wait on D_TRAIN;

        X_TEST <= setTestingInputs;
        wait on X_TEST;
        D_TEST <= targetOutputs( X_TEST );
        wait on D_TEST;

        initialization_finished <= TRUE;
end process INIT;
```

Note: Functions library was created and used to implement matrix initializations, training equations, and other function definitions similar to Python implementation (see appendices in final project report)

IMPLEMENTATION: VHDL

Training

Variable Declarations

```
TRAIN : process

        variable error : real := ( 0.0 );
        variable error_gradient : real := ( 0.0 );

        variable x_vec : vector_I;    -- Input vector
        variable d_vec : vector_K;    -- Target vector

        variable h_vec : vector_J;    -- Weighted sum vector for hidden layer
(z)
        variable z_vec : vector_J;    -- Neuron vector for hidden layer

        variable o_vec : vector_K;    -- Weighted sum vector for output layer
(y)
        variable y_vec : vector_K;    -- Neuron vector for output layer

        variable delta : vector_K;    -- Change factor gauge from error
gradient

        variable V_UPDATE : matrix_JI;
        variable W_UPDATE : matrix_KJ;
```

Training Epochs

```
begin
        wait until initialization_finished = TRUE;
        W_WEIGHT_VAR := setRandomMatrix_W;
        V_WEIGHT_VAR := setRandomMatrix_V;

        for epoch_count in 0 to EPOCHS loop  -- Loop through all epochs
            for m_count in 0 to M loop       -- Loop through elements in training matrices

                -- FORWARD PASS
                x_vec := ( X_TRAIN( m_count, 0 ), X_TRAIN( m_count, 1 ), X_TRAIN( m_count, 2
) );
                d_vec( 0 ) := D_TRAIN( m_count, 0 );
                h_vec := dotProduct_Vx( V_WEIGHT_VAR, x_vec );
                z_vec := sigmoidActivation_h( h_vec );
                o_vec := dotProduct_Wz( W_WEIGHT_VAR, z_vec );
                y_vec := sigmoidActivation_o( o_vec );
                Y_TRAIN( m_count, 0 ) <= y_vec( 0 );

                -- BACKWARD PASS
                error_gradient := d_vec( 0 ) - y_vec( 0 );
                error := error + ( error_gradient * error_gradient ) / 2.0;
                delta( 0 ) := error_gradient * y_vec( 0 ) * ( 1.0 - y_vec( 0 ) );
                W_UPDATE := changeWeights_W( z_vec, delta );
                V_UPDATE := changeWeights_V( W_WEIGHT_VAR, z_vec, x_vec, delta );
                W_WEIGHT_VAR := updateWeights_W( W_UPDATE, W_WEIGHT_VAR );
                V_WEIGHT_VAR := updateWeights_V( V_UPDATE, V_WEIGHT_VAR );

            end loop; -- End m loop
        end loop; -- End epoch loop

        V_WEIGHT <= V_WEIGHT_VAR;
        W_WEIGHT <= W_WEIGHT_VAR;
        training_finished <= TRUE;
end process TRAIN;
```

IMPLEMENTATION: VHDL

Testing

Variable Declarations

```
TEST : process

        variable error : real := ( 0.0 );
        variable error_gradient : real := ( 0.0 );

        variable x_vec : vector_I;    -- Input vector
        variable d_vec : vector_K;    -- Target vector

        variable h_vec : vector_J;    -- Weighted sum vector for hidden layer (z)
        variable z_vec : vector_J;    -- Neuron vector for hidden layer

        variable o_vec : vector_K;    -- Weighted sum vector for output layer (y)
        variable y_vec : vector_K;    -- Neuron vector for output layer
```

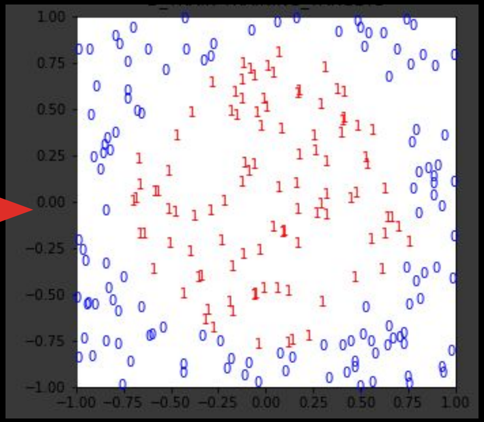
Testing Epochs

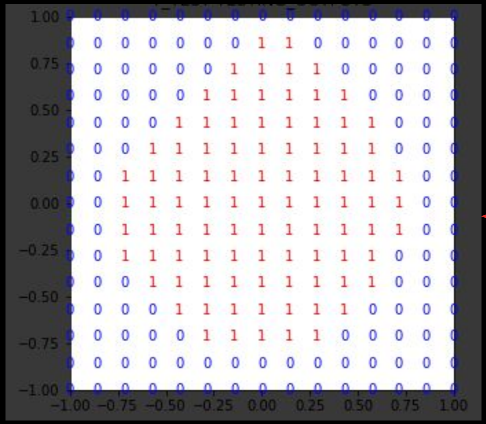
```
begin

        wait until training_finished = TRUE;

        for m_count in 0 to M loop        -- Loop through elements in training matrices

             -- FORWARD PASS
             x_vec := ( X_TEST( m_count, 0 ), X_TEST( m_count, 1 ), X_TEST( m_count, 2 )
);
             d_vec( 0 ) := D_TEST( m_count, 0 );

             h_vec := dotProduct_Vx( V_WEIGHT, x_vec );
             z_vec := sigmoidActivation_h( h_vec );
             o_vec := dotProduct_Wz( W_WEIGHT, z_vec );
             y_vec := sigmoidActivation_o( o_vec );

             Y_TEST( m_count, 0 ) <= y_vec( 0 );

             error_gradient := d_vec( 0 ) - y_vec( 0 );
             error := error + ( error_gradient * error_gradient ) / 2.0;

        end loop; -- End m loop

        testing_finished <= TRUE;
end process TEST;
```

TESTING RESULTS: PYTHON

The following figures show the resulting predictions on the xy-plane, where predictions of ‘**1**’ are highlighted as red and predictions of ‘**0**’ are highlighted as blue

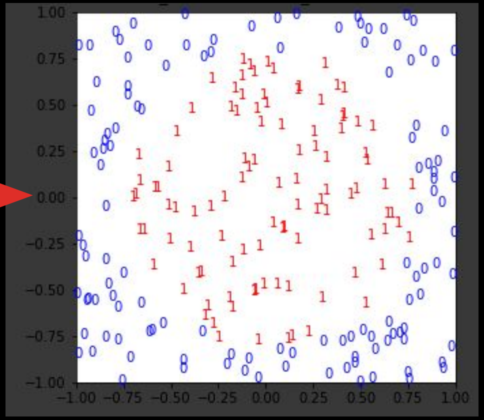
Predicted Output Matrix
(composed of randomly spaced coordinate points)

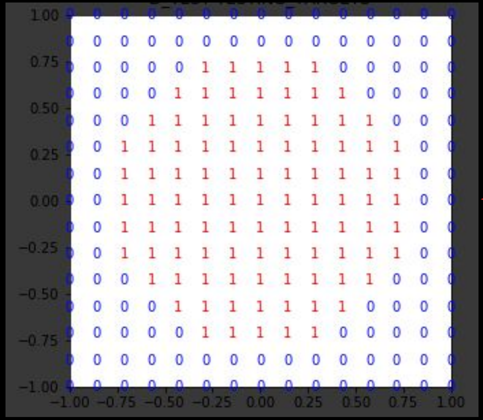




Predicted Output Matrix
(composed of uniformly spaced coordinate points)

Target Output Matrix
(composed of randomly spaced coordinate points)





Target Output Matrix
(composed of uniformly spaced coordinate points)

TESTING RESULTS: VHDL

The following figures show the resulting waveforms, representing output values of uniformly spaced coordinate points on the xy-plane

- Y-axis is represented by signals, corresponding to different rows of the output matrix
- X-axis is represented as a function of time, corresponding to different columns of the output matrix

Target Output Matrix

Predicted Output Matrix