Neural Digit Recognizer

[ECE20008] Practical Project 1

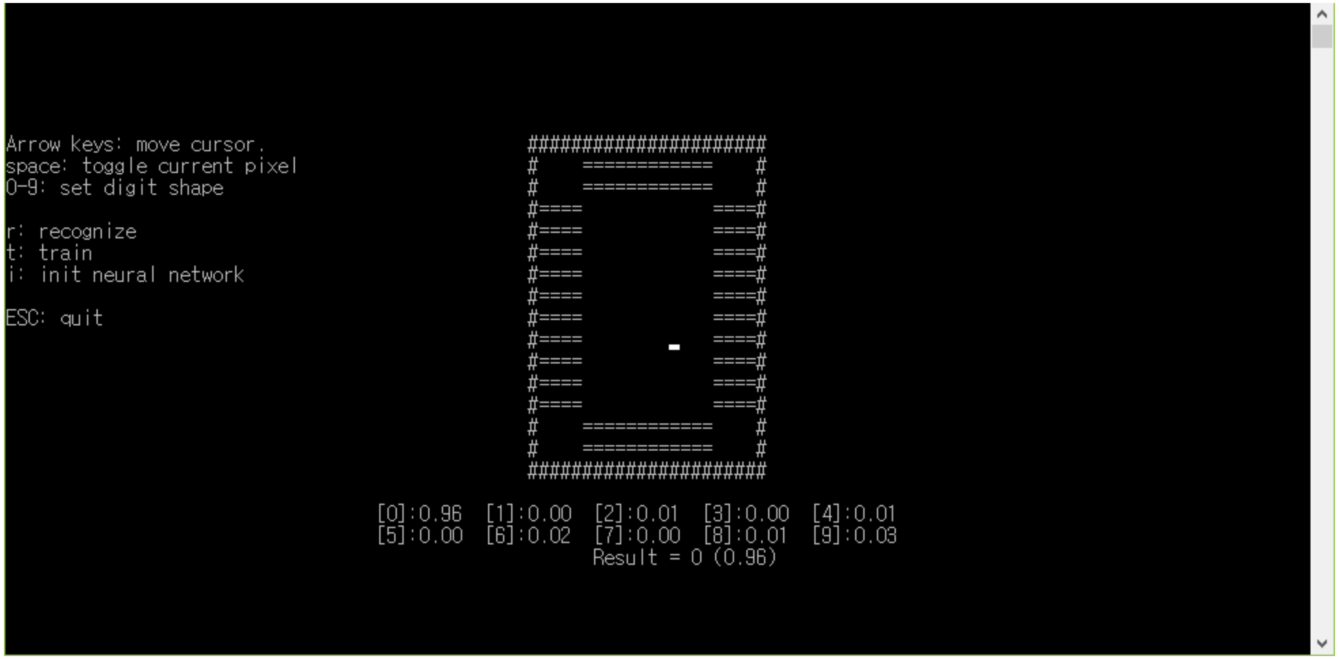
Mission

- Build a 5x7 digit recognizer using neural network.
  - Edit digit image
  - Train neural network
  - Recognize digit

- Provided
  - Neural networks in a nutshell
  - HGUDigitImage, HGUDigitUI
  - HGULayer, HGUNeuralNetwork
  - Reference code
    - See TestXOR() in main.cpp

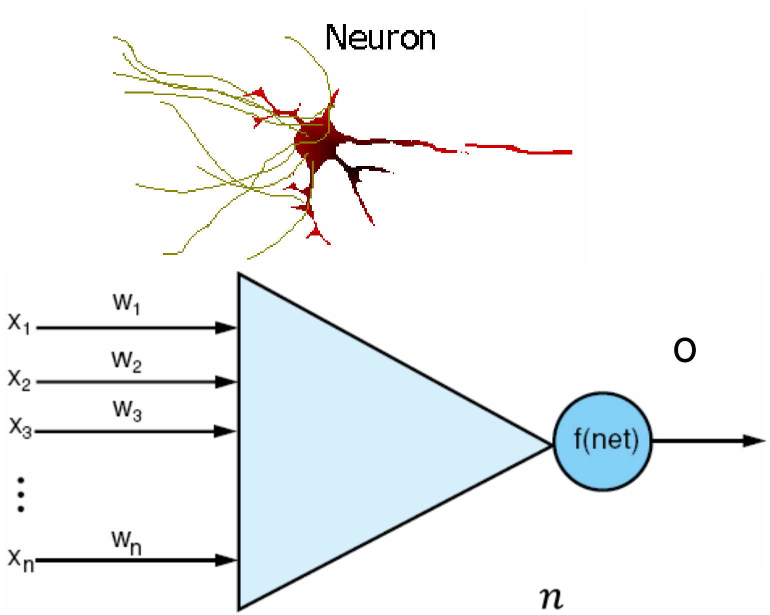
5x7 Digit Recognizer

- Feature
  - Recognize current digit with neural network
  - Train neural network

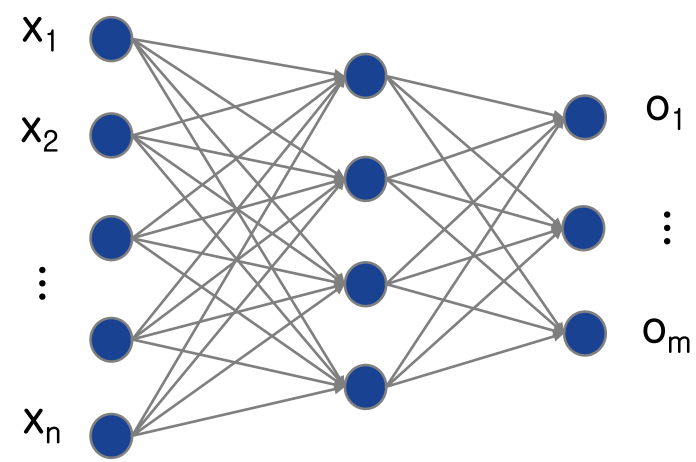


Neural Networks

- An artificial neural network is a mathematical model inspired by biological neural networks.
  - Intelligence comes from their connection weights
  - Connection weights are decided by learning or adaptation

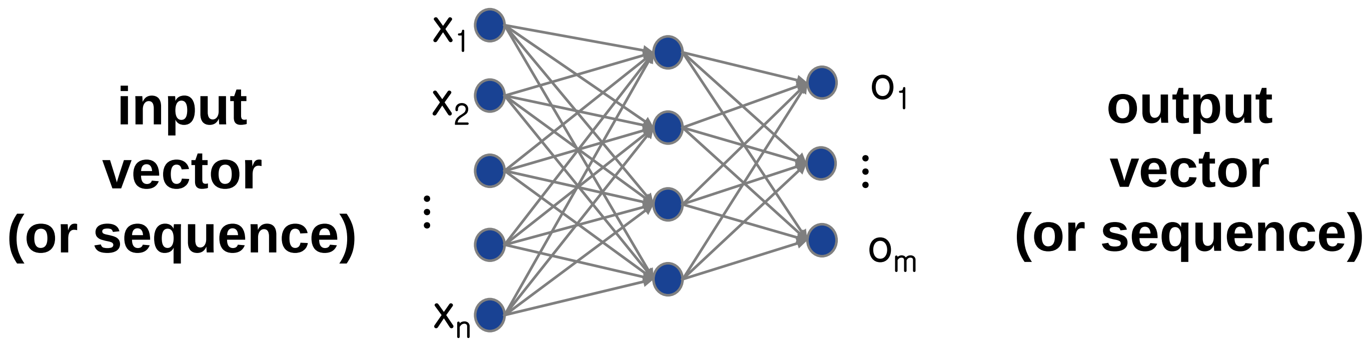


$$o = f(\sum_{i=1}^{n} w_i x_i + \theta)$$



Neural Networks

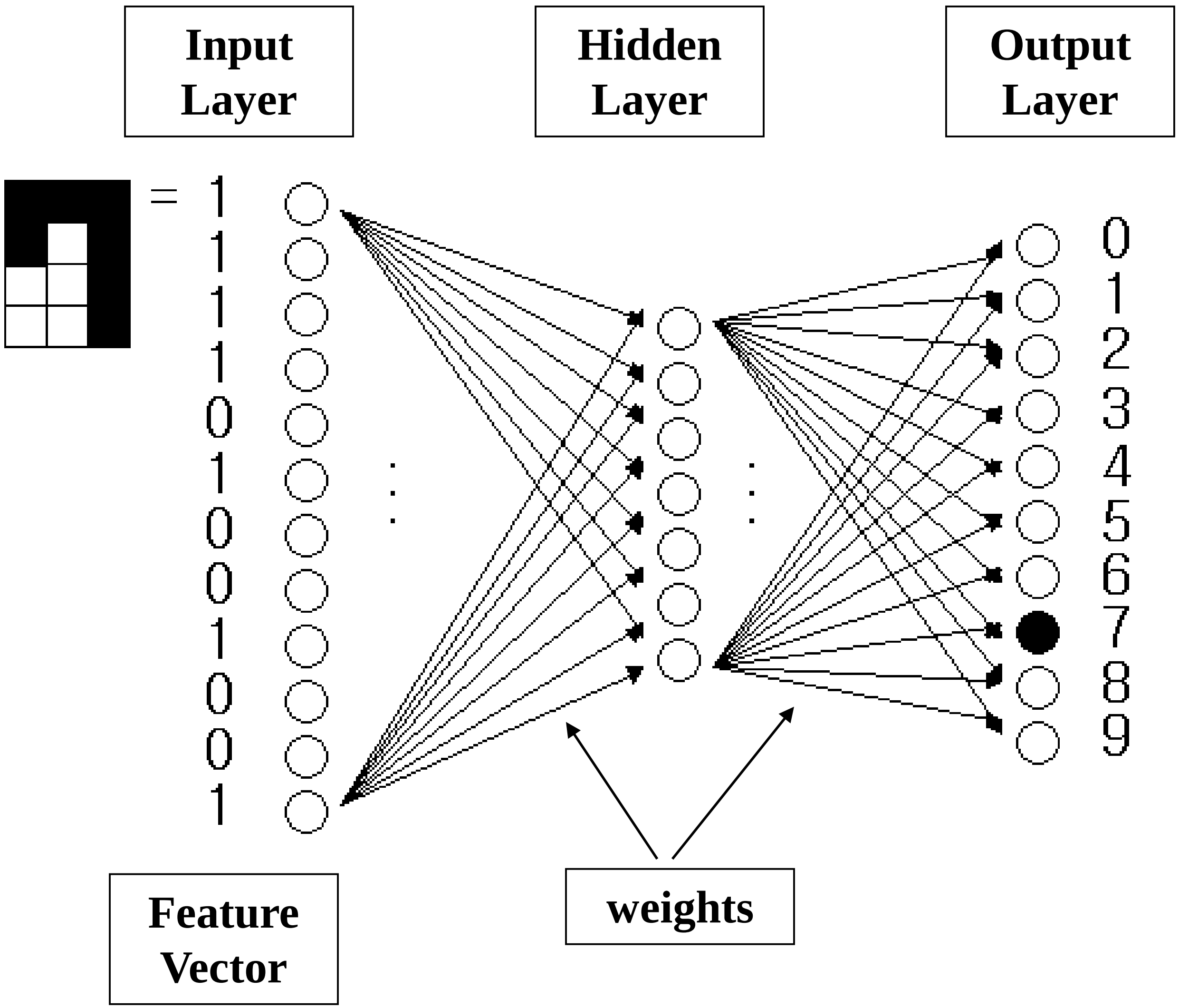
- Neural networks is a mathematical model to learn mappings
  - Mapping from a vector to another vector (or a scalar value)



Examples)

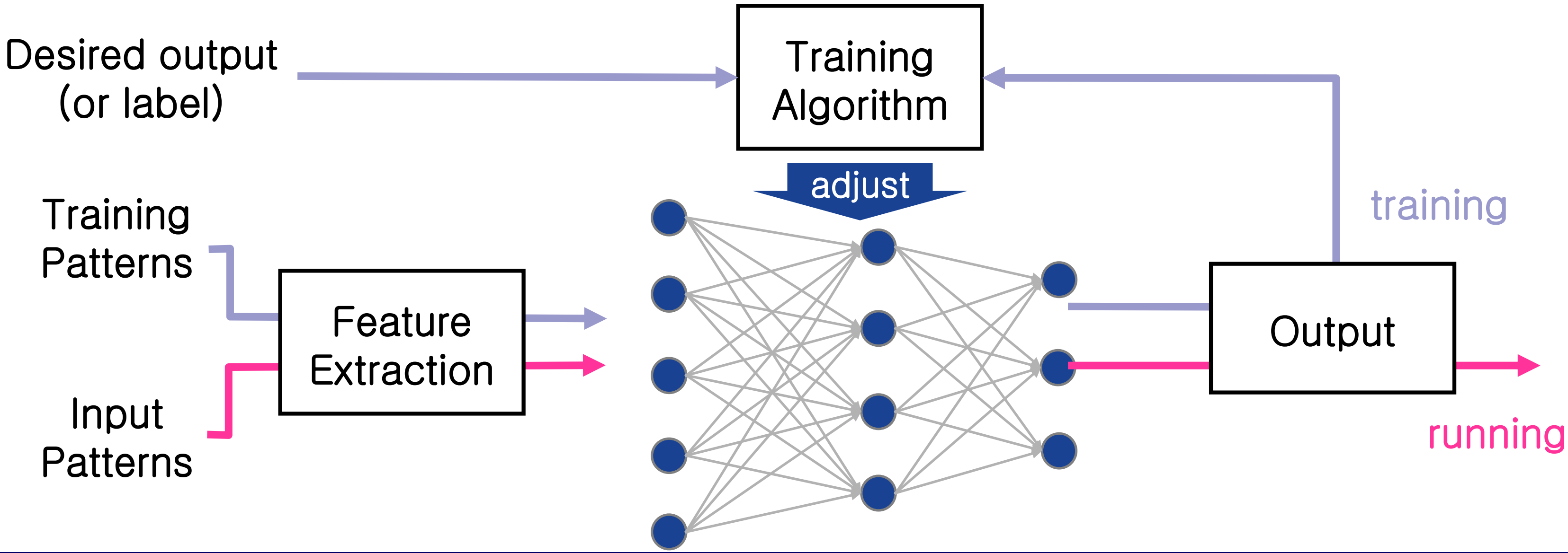
- Pattern → class (classification)
- Sentence → another sentence (translation, chatbot)
- Independent variables → dependent variables (regression)
- State → action (control, game play)
- Symptoms → diseases (diagnosis)
- History → future (prediction)

Ex) Digit Recognition using Neural Network



Building Neural Network Classifiers

1. Design network structure
2. Collect training samples (with labels)
3. Train connection weights (W)
   - Given training samples and desired outputs, adjust W to minimize loss
4. Apply the trained neural network to real problems



Class in C++

- C

```
struct Stack {
    int array[100];
    int top;
};

void Push(struct Stack *s,
    int item);
int Pop(struct Stack *s);

struct Stack stack1;
Push(&stack1, 10);
```

- C++

```
class Stack {
    int array[100];
    int top;

    void Push(int item);
    int Pop();
};

Stack stack2;
stack2.Push(10);
```

Neural Digit Recognizer Classes

- HGUDigitImage class
  - Represents 5x7 digit images
- HGUDigitUI class
  - Text-based user-interface
  - Image display, move cursor, process key presses
- HGULayer class
  - Represents neural network layers
  - Provides layer operators (eg. Propagate())
- HGUNeuralNetwork class
  - Represents neural networks composed of HGULayers
    - An array of HGULayers

HGUDigitUI

- HGUDigtiUI class displays digit image and handle key-press event

HGUDigitImage class

- Class to represent 5x7 digit images

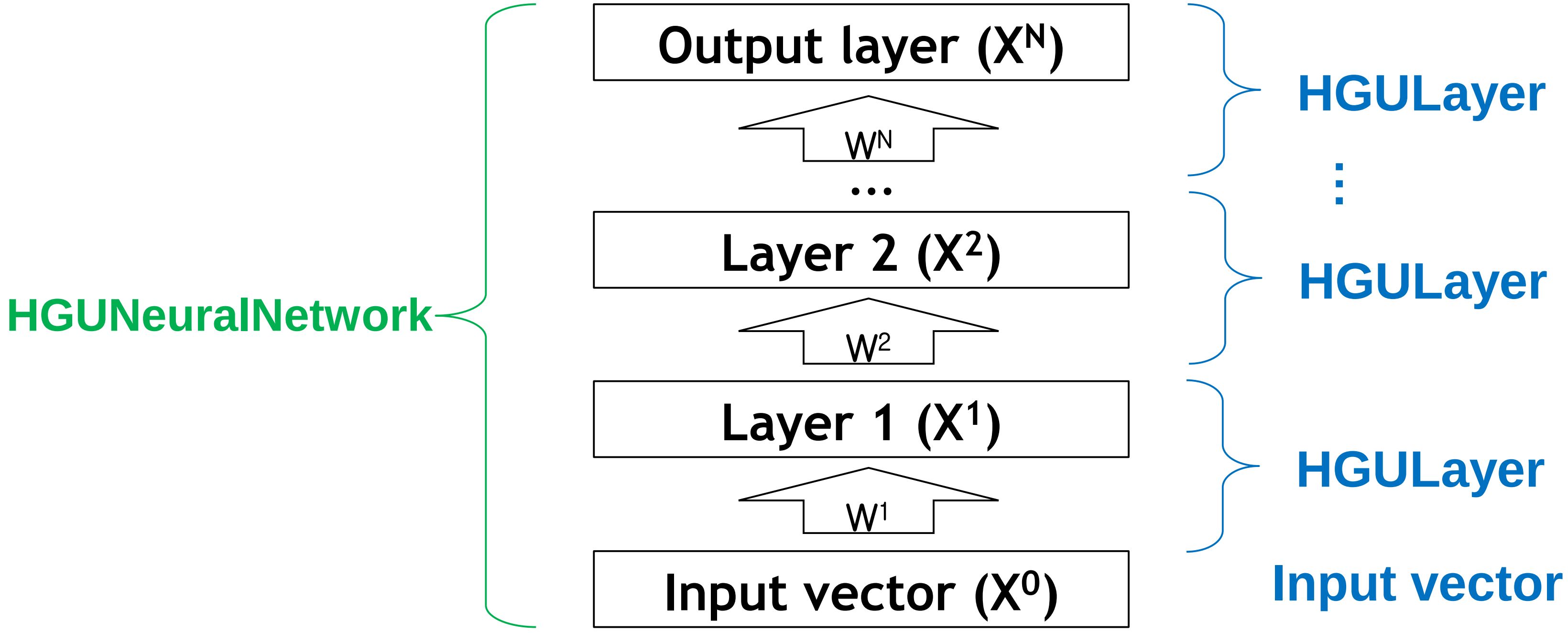
```
#define DigitWidth 5
#define DigitHeight 7
#define DigitSize 35

class HGUDigitImage {
        float vector[DigitSize];        // 5x7 = 35 dim array, 0.F: white, 1.F: black

public:
        HGUDigitImage();
        float& Pixel(int x, int y)       { return vector[y * DigitWidth + x]; }
        float& operator[](int idx)       { return vector[idx]; }
        float* GetVector()               { return vector; }

        void Clear(float v);
        void SetDigit(int d);                          // 0 <= d <= 9

        void SetImage(float *src);
        void Read(const char *file);

        HGUDigitImage& operator=(const HGUDigitImage &src);

        void Display();
        void DisplayAsArray();
};
```

HGUDigitUI class

```
class HGUDigitUI {
	int m_hScale;
	int m_vScale;

	HGUDigitImage m_digit;
	HGUNeuralNetwork *m_pNN;

public:
	HGUDigitUI();
	virtual ~HGUDigitUI();

	void SetScale(int hScale, int vScale);

	int Run();
	void DrawScreen();
	void DisplayDigit(int sx, int sy, HGUDigitImage *pDigit, int hScale = 0, int vScale = 0);
	void DrawBigDot(int x, int y, int width, int height, char v);
	void DrawBox(int left, int top, int right, int bottom, char v);

	int LoadRecognizer(int noLayer, int pNetStruct[], const char *weightFile);
	int RecognizeDigit();
	int TrainRecognizer(int maxEpoch);
};
```

HGUNeuralNetwork

- A neural networks is an array of layers
  - Each layer has a weight set and output nodes
    - n^{th} layer = (W^n, X^n)
      - Fields for training: gradient, delta_bar, delta, etc.



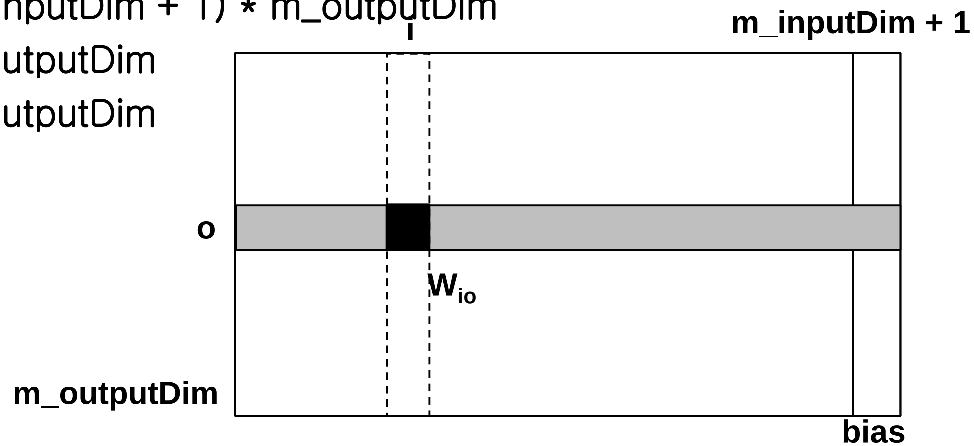
HGUNeuralNetwork

```
class HGUNeuralNetwork {
    int m_noLayer;
    HGULayer *m_aLayer;              // array of HGULayers

public:
    HGUNeuralNetwork();              // constructor
    ~HGUNeuralNetwork();             // destructor

    // for allocation, load, save
    int Alloc(int noLayer, int *pNoNode, HGUNeuralNetwork *pShareSrc = NULL);
    void InitializeWeight();         // initialization
    int Load(const char *fileName);
    int Save(const char *fileName);

    // for recognition
    int Propagate(float *pInput);    // forward propagation
    float GetOutput(int nodeIdx);    // get value of ith output node
    int GetMaxOutputIndex();         // get index of maximum output node

    // fro training
    int ComputeGradient(float *pInput, short *pDesiredOutput);        // compute gradient
    int UpdateWeight(float learningRate);        // add gradient to weight
    float GetError(float *pDesiredOutput);       // MSE loss
};
```

Data Members of HGULayer

$$o_j = f\left(\sum_i \boldsymbol{w_{ij} x_i} + \theta_j\right)$$

```
class HGULayer {
    int m_inputDim;
    int m_outputDim;

    float *m_pInput;         // size: m_inputDim
    float *m_aOutput;        // size: m_outputDim
    float *m_aWeight;        // size: (m_inputDim + 1) * m_outputDim
                             // should be initialized by small random numbers

    // only for training
    float *m_aGradient;      // size: (m_inputDim + 1) * m_outputDim
    float *m_aDelta;         // size: m_outputDim
    float *m_aDeltaBar;      // size: m_outputDim

    // member functions
}
```



Member Functions of HGULayer

```
class HGULayer {
    // data members
public:
    HGULayer();
    ~HGULayer()     { Delete(); }
    int Alloc(int inputDim, int outputDim);
    void InitializeWeight();                    // Xavier initialization
    virtual int Load(FILE *fp);
    virtual int Save(FILE *fp);

    int GetOutputDim() { return m_outputDim; }
    float GetOutput(int idx)    { return m_aOutput[idx]; }
    float* GetDeltaBar() { return m_aDeltaBar; }

    int Propagate(float *pInput);               // forward propagation
    int GetMaxOutputIndex();                    // retrieve the index of maximum output node

    // for training
    int ComputeDeltaBar(short *pDesiredOutput);
    int ComputeGradientFromDeltaBar();
    int Backpropagate(float *pPrevDeltaBar);
    virtual int UpdateWeight(float learningRate);

    float Activation(float net)     { return 1.F/(1.F + (float)exp(-net)); }
    float DerActivationFromOutput(float output){ return output * (1.F-output); }
};
```

Forward Propagation

```
int HGUNeuralNetwork::Propagate(float *pInput)
{
    m_aLayer[0].Propagate(pInput);
    for(int i = 1; i < m_noLayer; i++)
        m_aLayer[i].Propagate(m_aLayer[i-1].GetOutput());

    return TRUE;
}
```

Forward Propagation

- Forward propagation formula
  - Element notation

$$\mathrm{o}_{\boldsymbol{j}} = f\left(\sum_{\boldsymbol{i}} \boldsymbol{w}_{\boldsymbol{ij}} \boldsymbol{x}_{\boldsymbol{i}} + \theta_j\right)$$

  - Vector notation

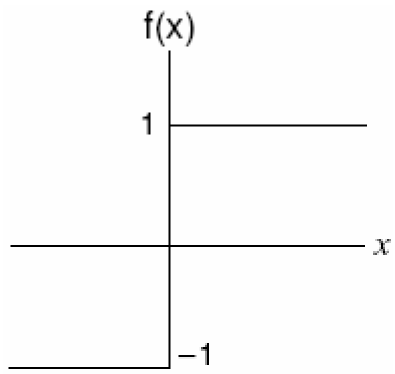
$$O = f(WX + \Theta)$$

Activation Functions

- Why activation functions?
  - Non-linearity
  - Restrict outputs in a specific range
  - Measurement → probability or decision

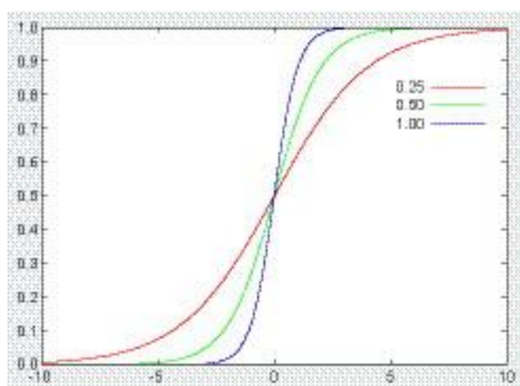
Hard-limit

$$f(x) = \begin{cases} +1 \text{ if } x \geq 0 \\ -1 \text{ otherwise} \end{cases}$$



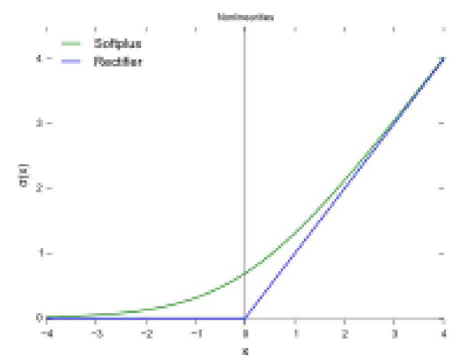
Sigmoid

$$f(x) = \frac{1}{(1 + e^{-\lambda x})}$$



ReLU

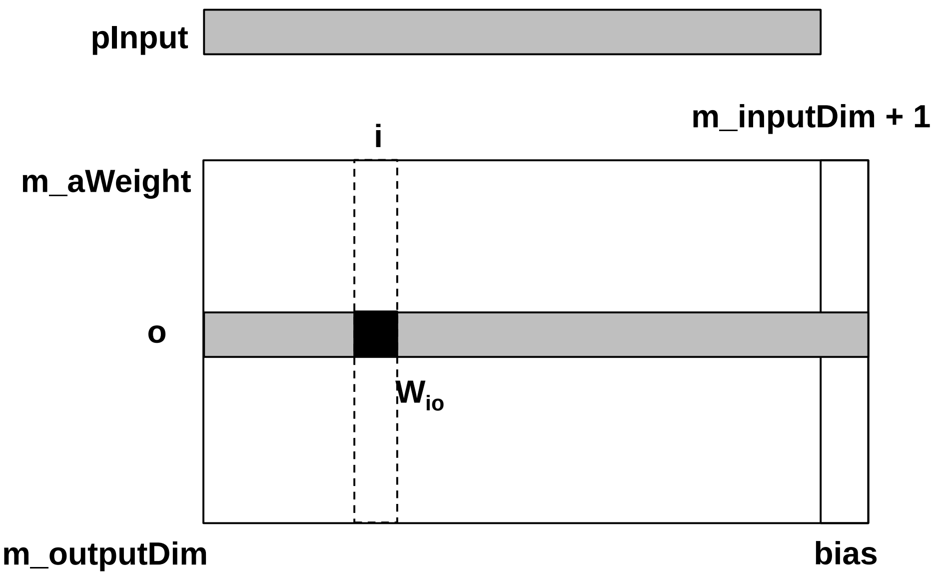
$$f(x) = \max(x, 0)$$



Forward Propagation

```
int HGULayer::Propagate(float *pInput)
{
    m_pInput = pInput; // for training

    for(int o = 0; o < m_outputDim; o++){
        float net = 0.F;
        float *inWeight = m_aWeight + o * (m_inputDim + 1);
        for(int i = 0; i < m_inputDim; i++)
            net += pInput[i] * inWeight[i];
        net += inWeight[m_inputDim]; // bias

        m_aOutput[o] = Activation(net);
    }

    return TRUE;
}
```

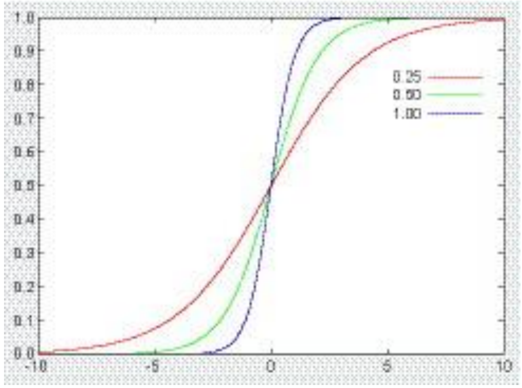
$$o_j = f\left(\sum_i \boldsymbol{w_{ij} x_i} + \theta_j\right)$$



Sigmoid Activation Function

- Sigmoid

$$f(x) = \frac{1}{1 + \exp(-x)}$$

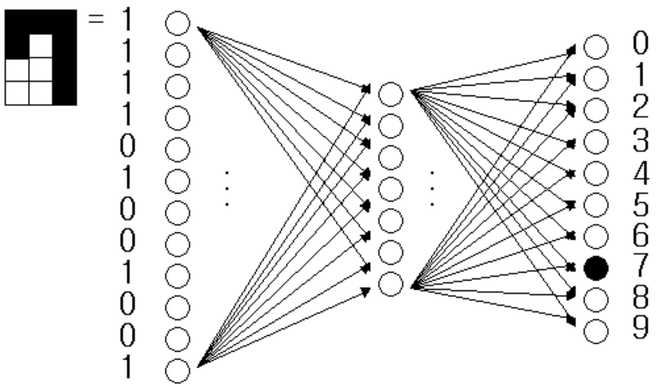


- Implementation

```
float Activation(float net)
{
    return 1.F/(1.F + (float)exp(-net));
}
```

Recognition using Neural Network

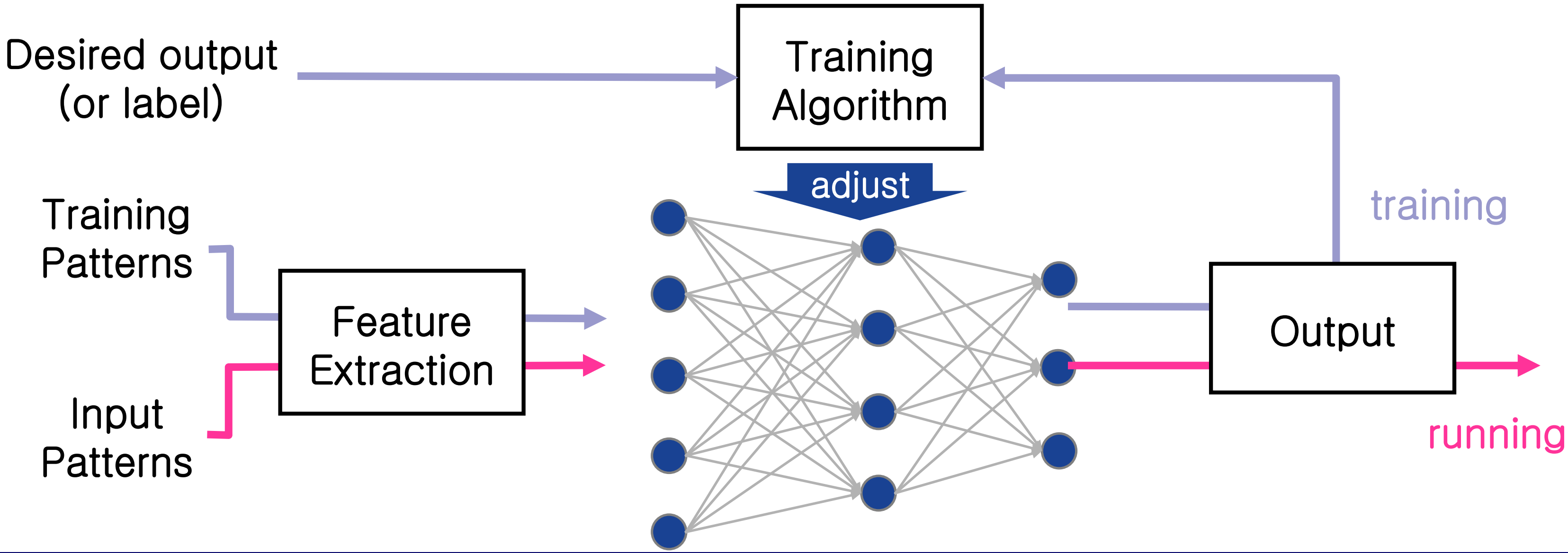
- Given
  - A neural network trained on training data.
    - HGUNeuralNetwork
  - Input pattern (vector)
    - HGUDigitImage



- Recognition procedure
  - Convert input pattern into vector (if necessary)
    - float GetVector(); // HGUDigitImage
  - Feed input vector to input layer and propagate
    - int Propagate(float *pInput); // HGUNeuralNetwork
  - Get the index of maximum output node
    - int GetMaxOutputIndex(); // HGUNeuralNetwork

Building Neural Network Classifiers

1. Design network structure
2. Collect training samples (with labels)
3. Train connection weights (W)
   - Given training samples and desired outputs, adjust W to minimize loss
4. Apply the trained neural network to real problems

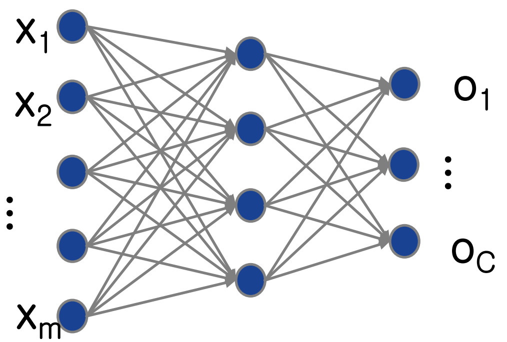


Training Neural Network

- Neural network training: error minimization
  - Given
    - A training sample $X = < x_1, x_2, \dots, x_m >$
    - Desired output $D = < d_1, d_2, \dots, d_C >$
      - m: input dim., C: output dim (# of classes)
  - Network propagation with weights W
    - output vector $O = < o_1, o_2, \dots, o_C >$
  - Loss function

    Ex) MSE (mean square error)

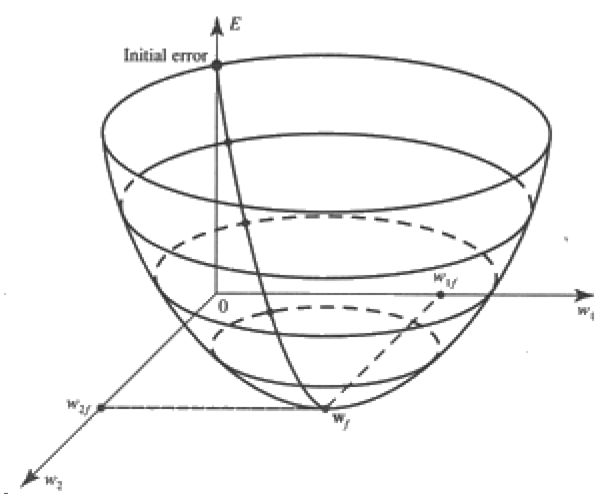
$$E_{MSE} = \frac{1}{2}\frac{\sum_c (o_c - d_c)^2}{C}$$

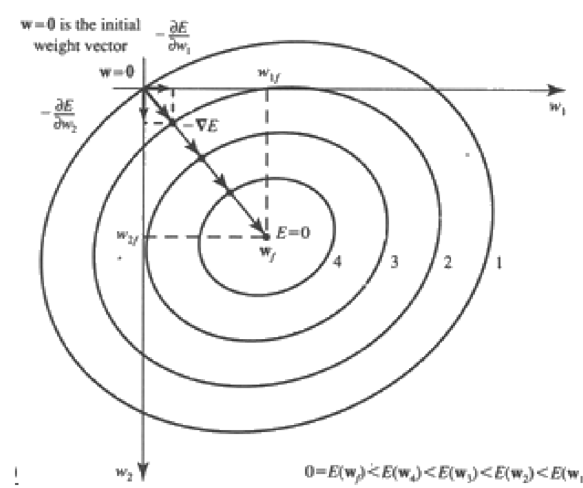


The Back-Propagation Algorithm

- Given fixed input and desired output, the error becomes a function of weights, $\boldsymbol{E}(\boldsymbol{W})$
- Given current weights W, the gradient gives a direction in which increases the error most rapidly
  - Gradient vector

$$\frac{\nabla E}{\nabla W} = \left(\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_i}, \ldots, \frac{\partial E}{\partial w_M}\right)$$

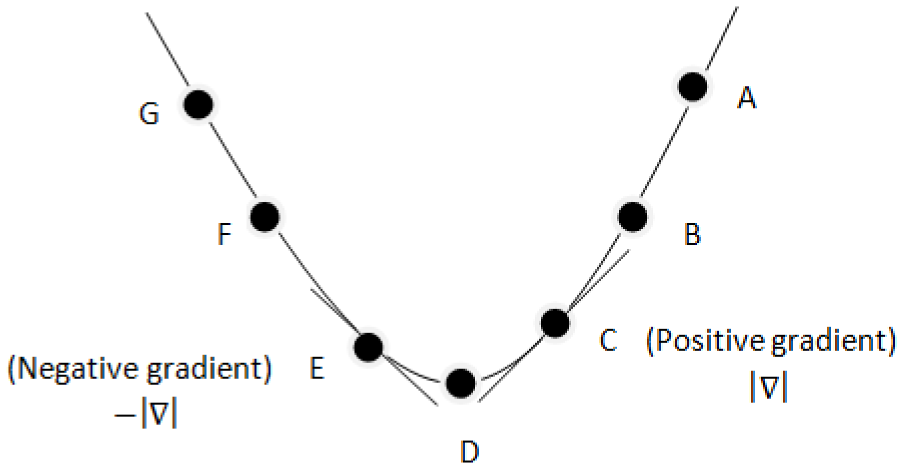




Gradient Descent Algorithm

- Given current weight vector W^t

$$W^{t+1} = W^t - \eta \frac{\partial E}{\partial W^t}$$



Training Neural Network

- Given
  - Training samples
  - Desired output of each training sample
  - Termination condition (maxEpoch or minError)

- Training procedure

  Repeat while (epoch < maxEpoch and avgError > minError)

  For each training sample, compute and accumulate gradient
  - int ComputeGradient(float *pInput, float *pDesiredOutput);
  - Measure error (float GetError(float *pDesiredOutput);)
  - Count samples

  Update weights with average gradient
  - int UpdateWeight(float learningRate);

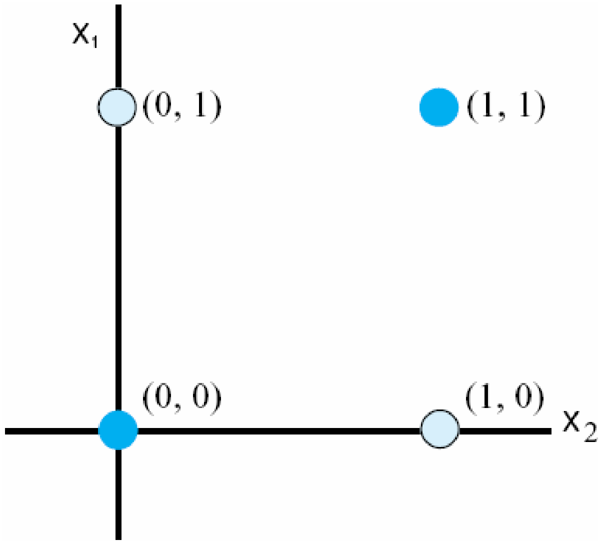
  Periodically, print loss and recognition rate

Example: XOR Problem

- XOR problem
  - Input: 2D vector (x_1, x_2)
  - Output: (y_0, y_1)
    - y_0 is for 0, y_1 is for 1

Ex) (1, 1) ➔ (1, 0), (1, 0) ➔ (0, 1)

x_1	x_2	Output
1	1	0
1	0	1
0	1	1
0	0	0



XOR Problem: Data Variables

```
int noLayer = 2; // # of layers not including input layer
int aNetStruct[] = { 2, 4, 2 }; // (inputDim, hiddenDim, outputDim)

HGUNeuralNetwork nn;
nn.Alloc(noLayer, aNetStruct, NULL);

// training samples and desired output
const int noSample = 4;
float aSample[4][2] = {
        { 0.F, 0.F },
        { 0.F, 1.F },
        { 1.F, 0.F },
        { 1.F, 1.F }
};
float aDesiredOutput[4][2] = {
    { 1, 0 },
    { 0, 1 },
    { 0, 1 },
    { 1, 0 }
}; // outputDim = 2
```

XOR Problem: Training

```
const int maxEpoch = 1000000;
float error = 0.F;
int n = 0;
int correct = 0;

for(int epoch = 0; epoch < maxEpoch; epoch++){
    // compute gradient on each sample
    for(int i = 0; i < noSample; i++){
        nn.ComputeGradient(aSample[i], aDesiredOutput[i]);  // compute and accumulate gradient

        error += nn.GetError(aDesiredOutput[i]);            // accumulate MSE loss

        if(nn.GetMaxOutputIndex() == aDesiredOutput[i][1])
            correct++;
    }
    n += noSample;

    // update once an epoch
    nn.UpdateWeight(0.01F / noSample);                      // update weight with average gradient

    if(error / n < 0.0001F)
        break;
}
```

XOR Problem: Recognition Test

```
for(int j = 0; j < noSample; j++){
    nn.Propagate(aSample[j]);
    float *hidden = nn[0]->GetOutput();
    float *output = nn[1]->GetOutput();
    printf("sample %d: (%.3f %.3f) --> (%.3f %.3f %.3f %.3f) --> (%.3f, %.3f)₩n",
        j, aSample[j][0], aSample[j][1],
        hidden[0], hidden[1], hidden[2], hidden[3],
        output[0], output[1]);
}
```

- Output

```
=== Testing MLP...
sample 0: (0.000 0.000) --> (0.145 0.428 0.033 0.904) --> (0.941, 0.063)
sample 1: (0.000 1.000) --> (0.804 0.156 0.000 0.140) --> (0.058, 0.941)
sample 2: (1.000 0.000) --> (0.004 0.346 0.950 0.999) --> (0.061, 0.937)
sample 3: (1.000 1.000) --> (0.091 0.116 0.031 0.928) --> (0.942, 0.056)
```

Mission

- Complete Neural Digit Recognzier by implementing the following functions:
  - int HGUDigitUI::RecognizeDigit();
    - Called by pressing ‘r’ key
  - int HGUDigitUI::TrainRecognizer(int maxEpoch);
    - Called by pressing ‘t’ key

Q&A

Thank you
for your attention!