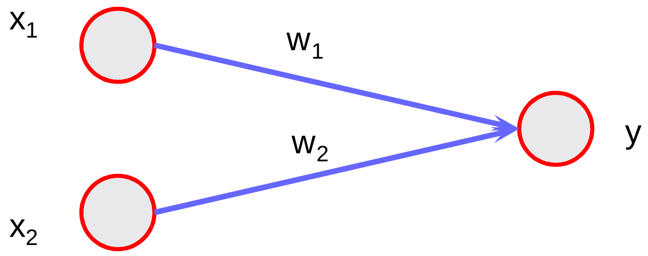
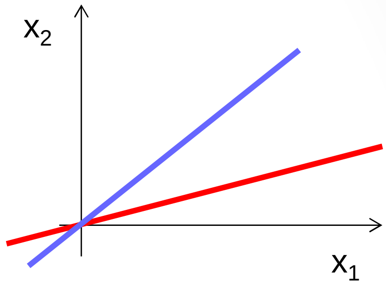
10. Perceptron

Perceptron (1)

$$y = w_1x_1 + w_2x_2 = \begin{bmatrix} w_1 & w_2 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$



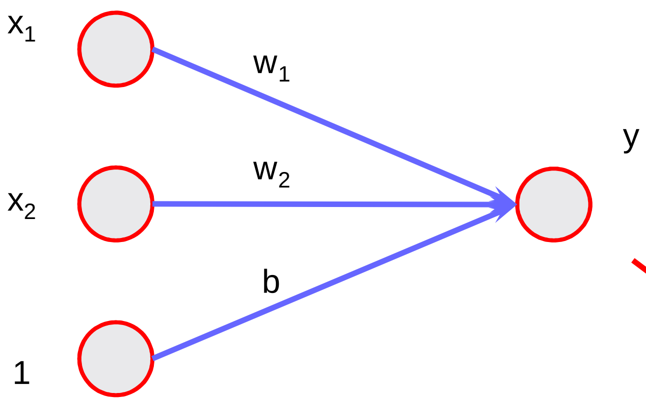


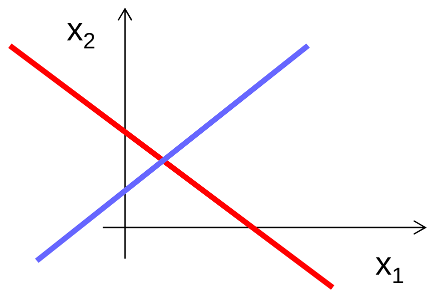
$$c = \begin{cases} c_1 & if\ \ y \geq 0 \\ c_2 & if\ \ y < 0 \end{cases}$$

$$0 = w_1x_1 + w_2x_2$$
$$w_2x_2 = -w_1x_1$$
$$x_2 = -\frac{w_1}{w_2}x_1$$

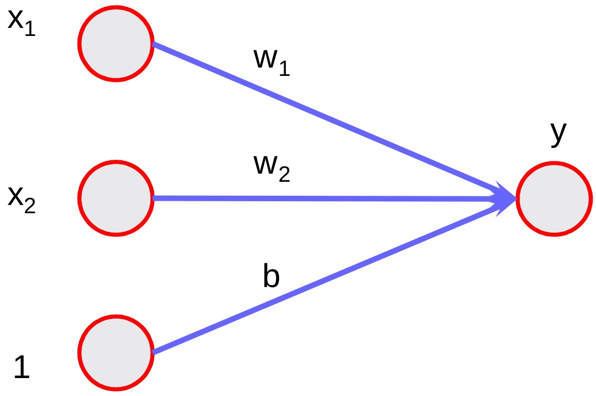
Perceptron (2)

$$y = w_1x_1 + w_2x_2 + b = \begin{bmatrix} w_1 & w_2 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + b$$





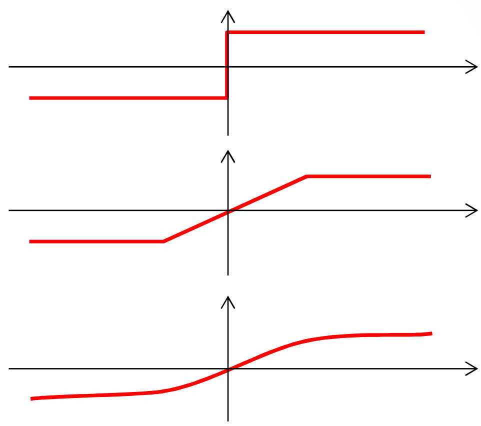
Perceptron (3)



$$y = f(w_1 x_1 + w_2 x_2 + b)$$
$$= f\left(\begin{bmatrix} w_1 & w_2 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + b\right)$$

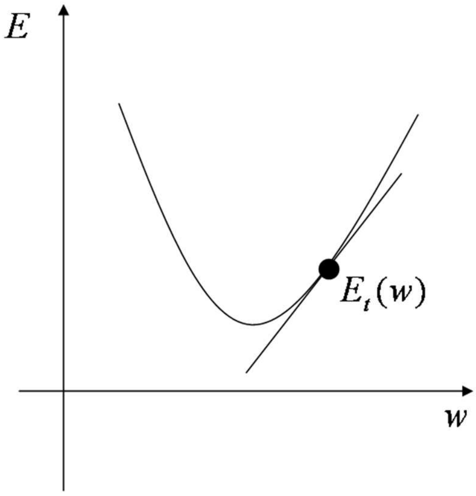
Activation function
(활성함수)

Perceptron (4)



Gradient descent (1)

$$w_i(t+1) = w_i(t) + \eta\big(d(t) - y(t)\big)x_i(t)$$



Gradient descent

$$E = \frac{1}{2}\big(d(t) - y(t)\big)^2$$

$$\frac{\partial E}{\partial w_i} = -\big(d(t) - y(t)\big)\frac{\partial y(t)}{\partial w_i}$$
$$= -\big(d(t) - y(t)\big)x_i(t)$$

Gradient descent (2)

$$\frac{\partial E}{\partial w_i} = -\big(d(t) - y(t)\big)\frac{\partial y(t)}{\partial w_i} = -\big(d(t) - y(t)\big)\frac{\partial y(t)}{\partial s}\frac{\partial s}{\partial w_i}$$
$$= -\big(d(t) - y(t)\big)\sigma(s)\big(1 - \sigma(s)\big)x_i(t) = -\big(d(t) - y(t)\big)y(t)\big(1 - y(t)\big)x_i(t)$$

$$y = \sigma\big(w_1x_1 + w_2x_2 + b\big) = \sigma(s), \quad \sigma'(s) = \sigma(s)\big(1 - \sigma(s)\big)$$

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

$$\sigma'(x) = \frac{d}{dx}\left(\frac{1}{1 + e^{-x}}\right) = \frac{d}{dx}\left(1 + e^{-x}\right)^{-1} = -\left(1 + e^{-x}\right)^{-2}\left(-e^{-x}\right)$$

$$= e^{-x}\left(1 + e^{-x}\right)^{-2} = \frac{1}{1 + e^{-x}}\frac{e^{-x}}{1 + e^{-x}} = \frac{1}{1 + e^{-x}}\left(1 - \frac{1}{1 + e^{-x}}\right) = \sigma(x)\big(1 - \sigma(x)\big)$$

KhuDaNet.h (1)

```
#include "KhuDaNetLayer.h"
#include <vector>
#define MAX_INFORMATION_STRING_SIZE  1000

class CKhuDaNet {
public:
  CKhuDaNet();
  virtual ~CKhuDaNet();

  std::vector<CKhuDaNetLayer*> m_Layers;

  int m_nInputSize, m_nOutputSize;
  char *m_Information;

  char *GetInformation();
  bool IsNetwork();
  void ClearAllLayers();
  void AddLayer(CKhuDaNetLayer *pLayer);
  void AddLayer(CKhuDaNetLayerOption LayerOptionInput);
  void AllocDeltaWeight();
  void InitWeight();
  int Forward(double *Input, double *Probability = 0);
  int TrainBatch(double **Input, double **Output, int nBatchSize, double *pLoss);
```


KhuDaNet.h (1)

```
  void SaveKhuDaNet(char *Filename);
  void LoadKhuDaNet(char *Filename);

  static int ArgMax(double *List, int nCnt);
  static double **dmatrix(int nH, int nW);
  static void free_dmatrix(double **Image, int nH, int nW);
  static double **dmatrix1d(int nH, int nW);
  static void free_dmatrix1d(double **Image, int nH, int nW);
  static double Identify(double x);
  static double DifferentialIdentify(double x);
  static double BinaryStep(double x);
  static double DifferentialBinaryStep(double x);
  static double Sigmoid(double x);
  static double DifferentialSigmoid(double x);
};
```

KhuDaNet.cpp (1)

```
…
CKhuDaNet::CKhuDaNet() {
  m_nInputSize = m_nOutputSize = 0;

  m_Information = new char[MAX_INFORMATION_STRING_SIZE];
}

CKhuDaNet::~CKhuDaNet() {
  ClearAllLayers();

  delete [] m_Information;
}

bool CKhuDaNet::IsNetwork() {
  if(m_Layers.size() < 2) return false;
  return true;
}
```


KhuDaNet.cpp (2)

```
char *CKhuDaNet::GetInformation() {
  memset(m_Information, 32, MAX_INFORMATION_STRING_SIZE);

  m_Information[MAX_INFORMATION_STRING_SIZE-1] = 0;

  int nPos = 0;
  for(auto &Layer : m_Layers) {
    if(Layer->m_LayerOption.nLayerType & KDN_LT_FC) {
      sprintf(m_Information+nPos, "%s(%5d)  ", "FC",
        Layer->m_LayerOption.nNodeCnt);
      nPos += 10;
    }
  }

  return m_Information;
}
```

KhuDaNet.cpp (3)

```
void CKhuDaNet::ClearAllLayers() {
  for(std::vector<CKhuDaNetLayer*>::reverse_iterator Iter = m_Layers.rbegin();
    Iter != m_Layers.rend(); ++Iter) {
    delete [] *Iter;
    *Iter = 0;
  }
  m_Layers.clear();
}

void CKhuDaNet::AddLayer(CKhuDaNetLayer *pLayer) {
  if(m_Layers.size() == 0)
    m_nInputSize = pLayer->m_LayerOption.nNodeCnt;

  m_nOutputSize = pLayer->m_LayerOption.nNodeCnt;

  m_Layers.push_back(pLayer);
}
```


KhuDaNet.cpp (4)

```
void CKhuDaNet::AddLayer(CKhuDaNetLayerOption LayerOptionInput) {
  CKhuDaNetLayer *pLayer;

  if(m_Layers.size() == 0) {
    pLayer = new CKhuDaNetLayer(LayerOptionInput, nullptr);
    m_nInputSize = pLayer->m_LayerOption.nNodeCnt;
  }
  else
    pLayer = new CKhuDaNetLayer(LayerOptionInput, m_Layers[m_Layers.size()-1]);

    m_nOutputSize = pLayer->m_LayerOption.nNodeCnt;

    m_Layers.push_back(pLayer);
}

void CKhuDaNet::InitWeight() {
  for(auto &Layer : m_Layers)
    Layer->InitWeight();
}
```

KhuDaNet.cpp (5)

```
void CKhuDaNet::AllocDeltaWeight() {
  for(auto &Layer : m_Layers)
    Layer->AllocDeltaWeight();
}

int CKhuDaNet::Forward(double *Input, double *Probability) {
  int MaxPos;

  for(auto &Layer : m_Layers) {
    if(Layer->m_LayerOption.nLayerType & KDN_LT_INPUT) {
      if(Layer->m_LayerOption.nLayerType & KDN_LT_FC)
        memcpy(Layer->m_Node, Input,
          Layer->m_LayerOption.nNodeCnt*sizeof(double));
    }
    else if(Layer->m_LayerOption.nLayerType & KDN_LT_OUTPUT)
      MaxPos = Layer->ComputeLayer(Probability);
    else
      Layer->ComputeLayer();
  }

  return MaxPos;
}
```


KhuDaNet.cpp (6)

```
int CKhuDaNet::TrainBatch(double **Input, double **Output, int nBatchSize,
double *pLoss) {
  int nTP = 0;
  *pLoss = 0;

  AllocDeltaWeight();

  for(int i = 0 ; i < nBatchSize ; ++i) {
    int MaxPos = Forward(Input[i]);

    if(m_nOutputSize == 1) {
      if(MaxPos == 1 && Output[i][0] > 0.5)  nTP++;
      else if(MaxPos == 0 && Output[i][0] < 0.5) nTP++;
    }
    else {
      if(MaxPos == ArgMax(Output[i], m_nOutputSize)) nTP++;
    }
```

KhuDaNet.cpp (7)

```
    for(std::vector<CKhuDaNetLayer*>::reverse_iterator Iter = m_Layers.rbegin();
      Iter != m_Layers.rend(); ++Iter) {
      (*Iter)->ComputeDelta(Output[i]);
      (*Iter)->ComputeDeltaWeight(i==0?true:false);

      if(Iter == m_Layers.rbegin())
        *pLoss += (*Iter)->GetLoss();
    }
  }

  *pLoss /= nBatchSize;

  for(auto &Layer : m_Layers)
    Layer->UpdateWeight(nBatchSize);

  return nTP;
}
```


KhuDaNet.cpp (8)

```
void CKhuDaNet::SaveKhuDaNet(char *Filename) {
  FILE *fp = fopen(Filename, "wb");
  if(!fp) return;

  fwrite("KhuDaNet", sizeof(char), 8, fp);

  int Cnt = m_Layers.size();
  fwrite(&Cnt, sizeof(int), 1, fp);

  for(auto &Layer : m_Layers) {
    fwrite(&(Layer->m_LayerOption), sizeof(CKhuDaNetLayerOption), 1, fp);
  }
```

KhuDaNet.cpp (9)

```
  CKhuDaNetLayer *pBackwardLayer = nullptr;
  for(auto &Layer : m_Layers) {
    if(pBackwardLayer) {
      if(Layer->m_LayerOption.nLayerType & KDN_LT_FC) {
        if(pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC)
          for(int i = 0 ; i < Layer->m_LayerOption.nNodeCnt ; ++i)
            fwrite(Layer->m_Weight[i], sizeof(double),
              pBackwardLayer->m_LayerOption.nNodeCnt, fp);

        fwrite(Layer->m_Bias, sizeof(double),
          Layer->m_LayerOption.nNodeCnt, fp);
      }
    }

    pBackwardLayer = Layer;
  }

  fclose(fp);
}
```


KhuDaNet.cpp (10)

```
void CKhuDaNet::LoadKhuDaNet(char *Filename) {
  ClearAllLayers();
  FILE *fp = fopen(Filename, "rb");
  if(!fp) return;

  char Buf[10];
  int nLayerCnt;

  fread(Buf, sizeof(char), 8, fp);
  fread(&nLayerCnt, sizeof(int), 1, fp);

  for(int s = 0 ; s < nLayerCnt ; ++s) {
    CKhuDaNetLayer *pLayer;
    char *pRawLayerOption = new char[sizeof(CKhuDaNetLayerOption)];

    fread(pRawLayerOption, sizeof(CKhuDaNetLayerOption), 1, fp);

    CKhuDaNetLayerOption
      KhuDaNetLayerOption(*(CKhuDaNetLayerOption *)pRawLayerOption);
```

KhuDaNet.cpp (11)

```
    if(s == 0)  {
      pLayer = new CKhuDaNetLayer(KhuDaNetLayerOption, nullptr);
      m_nInputSize = pLayer->m_LayerOption.nNodeCnt;
    }
    else
      pLayer = new CKhuDaNetLayer(KhuDaNetLayerOption,
                     m_Layers[m_Layers.size()-1]);

    m_Layers.push_back(pLayer);
    m_nOutputSize = pLayer->m_LayerOption.nNodeCnt;

    delete [] pRawLayerOption;
  }
```


KhuDaNet.cpp (12)

```
  CKhuDaNetLayer *pBackwardLayer = nullptr;
  for(int s = 0 ; s < nLayerCnt ; ++s) {
    if(pBackwardLayer) {
     if(m_Layers[s]->m_LayerOption.nLayerType & KDN_LT_FC) {
       if(pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC)
         for(int i = 0 ; i < m_Layers[s]->m_LayerOption.nNodeCnt ; ++i)
           fread(m_Layers[s]->m_Weight[i], sizeof(double),
             pBackwardLayer->m_LayerOption.nNodeCnt, fp);

         fread(m_Layers[s]->m_Bias, sizeof(double),
           m_Layers[s]->m_LayerOption.nNodeCnt, fp);
      }
    }

    pBackwardLayer = m_Layers[s];
  }

  fclose(fp);
}
```

KhuDaNet.cpp (13)

```
int CKhuDaNet::ArgMax(double *List, int nCnt) {
  int MaxPos = 0;

  for(int i = 0 ; i < nCnt ; ++i)
    if(List[i] > List[MaxPos]) MaxPos = i;

  return MaxPos;
}

double **CKhuDaNet::dmatrix(int nH, int nW) {…}
void CKhuDaNet::free_dmatrix(double **Image, int nH, int nW) {…}
double **CKhuDaNet::dmatrix1d(int nH, int nW) {…}
void CKhuDaNet::free_dmatrix1d(double **Image, int nH, int nW) {…}
```


KhuDaNet.cpp (14)

```
double CKhuDaNet::Identify(double x) {
  return x;
}

double CKhuDaNet::DifferentialIdentify(double x) {
  return 1;
}

double CKhuDaNet::BinaryStep(double x) {
  return (x>0)?1:0;
}

double CKhuDaNet::DifferentialBinaryStep(double x) {
  return 0;
}

double CKhuDaNet::Sigmoid(double x) {
  return 1./(1.+exp(-1. * x));
}

double CKhuDaNet::DifferentialSigmoid(double x) {
  return x*(1.-x);
}
```

KhuDaNetLayer.h (1)

```
#define KDN_LT_FC         0x0001

#define KDN_LT_INPUT      0x0100
#define KDN_LT_OUTPUT     0x0400

#define KDN_AF_NONE       0
#define KDN_AF_IDENTIFY   1
#define KDN_AF_BINARY_STEP        2
#define KDN_AF_SIGMOID    3
```


KhuDaNetLayer.h (2)

```
struct CKhuDaNetLayerOption{
  CKhuDaNetLayerOption(unsigned int nLayerTypeIntput, int nImageCntInput,
    int nNodeCntIput, int nWidthInput, int nHeightInput, int nKernelSizeInput,
    int nActicationFnInput, double dLearningRateInput);

  unsigned int nLayerType;
  int nImageCnt;
  int nNodeCnt;
  int nW, nH;
  int nKernelSize;
  int nActicationFn;

  double dLearningRate;
};
```

KhuDaNetLayer.h (3)

```
class CKhuDaNetLayer {
public:
  CKhuDaNetLayerOption m_LayerOption;
  CKhuDaNetLayer *m_pBackwardLayer;

  bool m_bTrained;

  double *m_Node;
  double **m_Weight;

  double *m_Bias;
```


KhuDaNetLayer.h (3)

```
  double *m_Loss;

  double *m_DeltaNode;
  double **m_DeltaWeight;
  double *m_DeltaBias;

  double (*Activation)(double);
  double (*DifferentialActivation)(double);

  CKhuDaNetLayer(CKhuDaNetLayerOption m_LayerOptionInput,
    CKhuDaNetLayer *pBackwardLayerInput);
  virtual ~CKhuDaNetLayer();

  void AllocDeltaWeight();
  void InitWeight();
  int ComputeLayer(double *Probability = 0);
  void ComputeDelta(double *Output);
  void ComputeDeltaWeight(bool bReset);
  void UpdateWeight(int nBatchSize);
  double GetLoss();
};
```

KhuDaNetLayer.cpp (1)

```
…
CKhuDaNetLayerOption::CKhuDaNetLayerOption(unsigned int nLayerTypeIntput,
  int nImageCntInput, int nNodeCntIput,
  int nWidthInput, int nHeightInput, int nKernelSizeInput,
  int nActicationFnInput, double dLearningRateInput)  {

  nLayerType = nLayerTypeIntput;
  nImageCnt = nImageCntInput;
  nNodeCnt = nNodeCntIput;

  nW = nWidthInput;
  nH = nHeightInput;

  nKernelSize = nKernelSizeInput;
  nActicationFn = nActicationFnInput;

  dLearningRate = dLearningRateInput;
}
```


KhuDaNetLayer.cpp (2)

```
CKhuDaNetLayer::CKhuDaNetLayer(CKhuDaNetLayerOption m_LayerOptionInput,
  CKhuDaNetLayer *pBackwardLayerInput)
  : m_LayerOption(m_LayerOptionInput), m_bTrained(false) {

  if(m_LayerOption.nActicationFn == KDN_AF_IDENTIFY) {
    Activation = CKhuDaNet::Identify;
    DifferentialActivation = CKhuDaNet::DifferentialIdentify;
  }
  else if(m_LayerOption.nActicationFn == KDN_AF_BINARY_STEP) {
    Activation = CKhuDaNet::BinaryStep;
    DifferentialActivation = CKhuDaNet::DifferentialBinaryStep;
  }
  else if(m_LayerOption.nActicationFn == KDN_AF_SIGMOID) {
    Activation = CKhuDaNet::Sigmoid;
    DifferentialActivation = CKhuDaNet::DifferentialSigmoid;
  }
```

KhuDaNetLayer.cpp (3)

```
  m_pBackwardLayer = pBackwardLayerInput;

  if((m_LayerOption.nLayerType & KDN_LT_OUTPUT) &&
     (m_LayerOption.nLayerType & KDN_LT_FC)){
    m_Loss = new double [m_LayerOption.nNodeCnt];
  }

  if((m_LayerOption.nLayerType & KDN_LT_INPUT) &&
     (m_LayerOption.nLayerType & KDN_LT_FC)) {
    m_Node = new double [m_LayerOption.nNodeCnt];
  }
  else if(m_LayerOption.nLayerType & KDN_LT_FC) {
    m_Node = new double [m_LayerOption.nNodeCnt];
    if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC)
      m_Weight = CKhuDaNet::dmatrix1d(m_LayerOption.nNodeCnt,
                                 m_pBackwardLayer->m_LayerOption.nNodeCnt);

    m_Bias = new double[m_LayerOption.nNodeCnt];
  }
}
```


KhuDaNetLayer.cpp (4)

```
CKhuDaNetLayer::~CKhuDaNetLayer() {
  if((m_LayerOption.nLayerType & KDN_LT_OUTPUT) &&
     (m_LayerOption.nLayerType & KDN_LT_FC)) {
    delete [] m_Loss;
  }
  if((m_LayerOption.nLayerType & KDN_LT_INPUT) &&
     (m_LayerOption.nLayerType & KDN_LT_FC)) {
    delete [] m_Node;
  }
```

KhuDaNetLayer.cpp (5)

```
  else if(m_LayerOption.nLayerType & KDN_LT_FC) {
    delete [] m_Node;
    if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC)
      CKhuDaNet::free_dmatrix1d(m_Weight, m_LayerOption.nNodeCnt,
                                m_pBackwardLayer->m_LayerOption.nNodeCnt);
    delete [] m_Bias;

    if(m_bTrained) {
      delete [] m_DeltaNode;
      if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC)
        CKhuDaNet::free_dmatrix1d(m_DeltaWeight,
           m_LayerOption.nNodeCnt, m_pBackwardLayer->m_LayerOption.nNodeCnt);

      delete [] m_DeltaBias;
    }
  }
}
```


KhuDaNetLayer.cpp (6)

```
void CKhuDaNetLayer::AllocDeltaWeight() {
  if((m_LayerOption.nLayerType & KDN_LT_OUTPUT) &&
     (m_LayerOption.nLayerType & KDN_LT_FC))
  {
  }
  if((m_LayerOption.nLayerType & KDN_LT_INPUT) &&
     (m_LayerOption.nLayerType & KDN_LT_FC)) {
  }
  else if(m_LayerOption.nLayerType & KDN_LT_FC) {
    if(!m_bTrained) {
      m_DeltaNode = new double [m_LayerOption.nNodeCnt];
      if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC)
        m_DeltaWeight = CKhuDaNet::dmatrix1d(m_LayerOption.nNodeCnt,
                       m_pBackwardLayer->m_LayerOption.nNodeCnt);

      m_DeltaBias = new double[m_LayerOption.nNodeCnt];
    }
  }

  m_bTrained = true;
}
```

KhuDaNetLayer.cpp (7)

```
void CKhuDaNetLayer::InitWeight() {
  static unsigned int seed = (unsigned int)std::chrono::
    system_clock::now().time_since_epoch().count();
  static std::default_random_engine generator(seed);
  if(m_LayerOption.nLayerType & KDN_LT_INPUT) return;
  if(m_LayerOption.nLayerType & KDN_LT_FC) {
    double var = 1;
    if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC)
      var = sqrt(2./
        (m_pBackwardLayer->m_LayerOption.nNodeCnt + m_LayerOption.nNodeCnt));

    std::normal_distribution<double> distribution(0., var);

    for(int i = 0 ; i < m_LayerOption.nNodeCnt ; ++i) {
      if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC) {
        for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nNodeCnt ; ++j)
          m_Weight[i][j] = distribution(generator);
      }
      m_Bias[i] = 0;
    }
  }
}
```


KhuDaNetLayer.cpp (8)

```
int CKhuDaNetLayer::ComputeLayer(double *Probability) {
  if(m_LayerOption.nLayerType & KDN_LT_INPUT) return 0;

  if((m_LayerOption.nLayerType & KDN_LT_FC) &&
      (m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC)) {
    for(int i = 0 ; i < m_LayerOption.nNodeCnt ; ++i) {
      m_Node[i] = 0;
      for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nNodeCnt ; ++j )
        m_Node[i] += m_pBackwardLayer->m_Node[j] * m_Weight[i][j];

      m_Node[i] = Activation(m_Node[i] + m_Bias[i]);
    }
  }
  int nMaxNode;
  if((m_LayerOption.nLayerType & KDN_LT_OUTPUT) &&
      (m_LayerOption.nLayerType & KDN_LT_FC)) {
    if(m_Node[0] < 0.5) nMaxNode = 0;
    else nMaxNode = 1;
  }
  if(Probability) *Probability = 0;
  return nMaxNode;
}
```

KhuDaNetLayer.cpp (9)

```
void CKhuDaNetLayer::ComputeDelta(double *Output) {
  if(m_LayerOption.nLayerType & KDN_LT_INPUT) return;

  if(m_LayerOption.nLayerType & KDN_LT_OUTPUT) {
    if(m_LayerOption.nLayerType & KDN_LT_FC) {
      for(int i = 0 ; i < m_LayerOption.nNodeCnt ; ++i)
        m_DeltaNode[i]
           = (Output[i]-m_Node[i]) * DifferentialActivation(m_Node[i]);
      for(int i = 0 ; i < m_LayerOption.nNodeCnt ; ++i)
        m_Loss[i] = (Output[i]-m_Node[i])*(Output[i]-m_Node[i]);
    }
  }

  if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_INPUT)
  return;
}
```


KhuDaNetLayer.cpp (10)

```
void CKhuDaNetLayer::ComputeDeltaWeight(bool bReset) {
  if(m_LayerOption.nLayerType & KDN_LT_INPUT) return;
  if(bReset) {
    if(m_LayerOption.nLayerType & KDN_LT_FC) {
      if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC) {
        for(int i = 0 ; i < m_LayerOption.nNodeCnt ; ++i) {
          for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nNodeCnt ; ++j )
            m_DeltaWeight[i][j] = 0;
          m_DeltaBias[i] = 0;
        }
      }
    }
  }
  if(m_LayerOption.nLayerType & KDN_LT_FC) {
    if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC) {
      for(int i = 0 ; i < m_LayerOption.nNodeCnt ; ++i) {
        for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nNodeCnt ; ++j )
          m_DeltaWeight[i][j] += m_DeltaNode[i] * m_pBackwardLayer->m_Node[j];
        m_DeltaBias[i] += m_DeltaNode[i];
      }
    }
  }
}
```

KhuDaNetLayer.cpp (11)

```
void CKhuDaNetLayer::UpdateWeight(int nBatchSize) {
  if(m_LayerOption.nLayerType & KDN_LT_INPUT) return;

  if(m_LayerOption.nLayerType & KDN_LT_FC) {
    if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC) {
      for(int i = 0 ; i < m_LayerOption.nNodeCnt ; ++i) {
        for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nNodeCnt ; ++j )
          m_Weight[i][j]
            += m_LayerOption.dLearningRate * m_DeltaWeight[i][j]/nBatchSize;

        m_Bias[i] += m_LayerOption.dLearningRate * m_DeltaBias[i]/nBatchSize;
      }
    }
  }
}
```


KhuDaNetLayer.cpp (12)

```
double CKhuDaNetLayer::GetLoss() {
  double Loss = 0;
  if((m_LayerOption.nLayerType & KDN_LT_OUTPUT) &&
     (m_LayerOption.nLayerType & KDN_LT_FC)) {
    for(int i = 0 ; i < m_LayerOption.nNodeCnt ; ++i)
      Loss += m_Loss[i];
  }
  return Loss;
}
```

MNIST (1)

- MNIST (modified national institute of standards and technology) database
- The MNIST Database of handwritten digits
  - http://yann.lecun.com/exdb/mnist/
  - Training set: 60,000, Test set: 10,000
  - Image file
    - [0]: 2051(0x00000803 – 08: unsigned char, 00000011: 2D),
    - [4]: 60000/10000, [8]: 28(rows), [12]: 28(columns),
    - [16]: pixel, raw data, row-wise, [0-255], unsigned char
  - Label file
    - [0]: 2049(0x00000801 – 08: unsigned char, 00000001: 1D),
    - [4]: 60000/10000,
    - [8]: label, [0-9], unsigned char

MNIST (2)

```
void CPerceptronTest::LoadMnistTrain() {
  char TrainImagePath[MAX_PATH], TrainLabelPath[MAX_PATH];

  sprintf(TrainImagePath, "%s\\train-images.idx3-ubyte", m_ExePath);
  sprintf(TrainLabelPath, "%s\\train-labels.idx1-ubyte", m_ExePath);

  FILE *fp = fopen(TrainImagePath, "rb");
  if(fp) {
    unsigned char Buf[28*28];
    fread(Buf, 1, 16, fp);
    int nCnt = 0;
    for(int i = 0 ; i < m_nMnistTrainTotal ; ++i) {
      fread(Buf, 1, 28*28, fp);

      for(int k = 0 ; k < 28*28 ; k++)
        m_MnistTrainInput[nCnt][k] = (double)Buf[k]/255.;
      nCnt++;
    }
    fclose(fp);
  }
```

MNIST (3)

```
  fp = fopen(TrainLabelPath, "rb");
  if(fp) {
    unsigned char Buf[32];
    fread(Buf, 1, 8, fp);

    int nCnt = 0;
    for(int i = 0 ; i < m_nMnistTrainTotal ; ++i) {
      fread(Buf, 1, 1, fp);
      m_MnistTrainOutput[nCnt] = Buf[0];
      nCnt++;
    }
    fclose(fp);
  }
}
```


Main.cpp (1)

```
class CKhuGleGraphLayer : public CKhuGleLayer {
public:
  // double m_TrainAccuacy, m_TrainLoss;
};
```

Main.cpp (2)

```
class CPerceptronTest : public CKhuGleWin {
public:
  CKhuGleGraphLayer *m_pTrainGraphLayer, *m_pTestGraphLayer;
  CKhuDaNet m_Perceptron;
  bool m_bTrainingRun;

  char m_ExePath[MAX_PATH];
  int m_nBatchCnt, m_nEpochCnt, m_nBatch;
  int m_nMnistTrainTotal, m_nMnistTestTotal;

  double **m_MnistTrainInput, **m_MnistTestInput;
  int *m_MnistTrainOutput, *m_MnistTestOutput;

  CPerceptronTest(int nW, int nH, char *ExePath);
  ~CPerceptronTest();
  void LoadMnistTrain();
  void LoadMnistTest();
  void Update();
};
```


Main.cpp (3)

```
CPerceptronTest::CPerceptronTest(int nW, int nH, char *ExePath)
  : CKhuGleWin(nW, nH) {
  strcpy(m_ExePath, ExePath);

  m_pScene = new CKhuGleScene(640, 480, KG_COLOR_24_RGB(100, 100, 150));
  m_pTrainGraphLayer = new CKhuGleGraphLayer(600, 200,
    KG_COLOR_24_RGB(150, 150, 200),2,  CKgPoint(20, 30));
  m_pTrainGraphLayer->SetMaxData(0, 100.);
  m_pTrainGraphLayer->SetMaxData(1, 2.5);
  m_pScene->AddChild(m_pTrainGraphLayer);

  m_pTestGraphLayer = new CKhuGleGraphLayer(600, 200,
    KG_COLOR_24_RGB(150, 150, 200), 1, CKgPoint(20, 260));
  m_pTestGraphLayer->SetMaxData(0, 100.);
  m_pScene->AddChild(m_pTestGraphLayer);

  m_Perceptron.AddLayer(CKhuDaNetLayerOption(KDN_LT_INPUT | KDN_LT_FC,
    0, 28*28, 0, 0, 0, 0, 0.15));
  m_Perceptron.AddLayer(CKhuDaNetLayerOption(KDN_LT_FC | KDN_LT_OUTPUT,
    0, 1, 0, 0, 0, KDN_AF_SIGMOID, 0.15));

  m_Perceptron.InitWeight();
```

Main.cpp (4)

```
  m_nBatchCnt = 0;
  m_nEpochCnt = 0;
  m_nBatch = 100;
  m_nMnistTrainTotal = 60000;
  m_nMnistTestTotal = 10000;

  m_MnistTrainInput = m_MnistTestInput = nullptr;
  m_MnistTrainOutput = m_MnistTestOutput = nullptr;

  std::cout << m_Perceptron.GetInformation() << std::endl;
  int i;
  if(!m_MnistTrainInput){
    m_MnistTrainInput = new double *[m_nMnistTrainTotal];

    for(i = 0 ; i < m_nMnistTrainTotal ; i++)
      m_MnistTrainInput[i] = new double[28*28];
  }

  if(!m_MnistTrainOutput)
    m_MnistTrainOutput = new int [m_nMnistTrainTotal];
```


Main.cpp (5)

```
  if(!m_MnistTestInput){
    m_MnistTestInput = new double *[m_nMnistTestTotal];

    for(i = 0 ; i < m_nMnistTestTotal ; i++)
      m_MnistTestInput[i] = new double[28*28];
  }

  if(!m_MnistTestOutput)
    m_MnistTestOutput = new int [m_nMnistTestTotal];

  LoadMnistTrain();
  LoadMnistTest();

  m_bTrainingRun = false;
}
```

Main.cpp (6)

```
CPerceptronTest::~CPerceptronTest() {
  int i;
  if(m_MnistTrainInput){
    for(i = 0 ; i < m_nMnistTrainTotal ; i++)
      delete [] m_MnistTrainInput[i];

    delete [] m_MnistTrainInput;
  }
  if(m_MnistTrainOutput)
    delete [] m_MnistTrainOutput;

  if(m_MnistTestInput){
    for(i = 0 ; i < m_nMnistTestTotal ; i++)
      delete [] m_MnistTestInput[i];

    delete [] m_MnistTestInput;
  }
  if(m_MnistTestOutput)
    delete [] m_MnistTestOutput;
}
```


Main.cpp (7)

```
void CPerceptronTest::Update() {
  if(m_bKeyPressed['S']) {
    m_bTrainingRun = !m_bTrainingRun;
    m_bKeyPressed['S'] = false;
  }
  if(!m_bTrainingRun) {
    m_pScene->Render();
    DrawSceneTextPos("Perceptron Test", CKgPoint(0, 0));

    CKhuGleWin::Update();
    return;
  }
```

Main.cpp (8)

```
  int nIndex = (m_nBatchCnt*m_nBatch)%m_nMnistTrainTotal;
  if(nIndex+m_nBatch >= m_nMnistTrainTotal)
    nIndex = m_nMnistTrainTotal-m_nBatch;

  int nOutputCnt = 1;
  double **OutputList = new double*[m_nBatch];
  for(int i = 0 ; i < m_nBatch ; ++i)
    OutputList[i] = new double[nOutputCnt];
  for(int i = 0 ; i < m_nBatch ; ++i) {
    for(int j = 0 ; j < nOutputCnt ; ++j) {
      OutputList[i][j] = 0;
      if(m_MnistTrainOutput[nIndex+i] > 4) OutputList[i][j] = 1;
    }
  }
```


Main.cpp (9)

```
  double Loss;
  int nTP = m_Perceptron.TrainBatch(m_MnistTrainInput+nIndex, OutputList,
    m_nBatch, &Loss);

  for(int i = 0 ; i < m_nBatch ; ++i)
    delete [] OutputList[i];
  delete [] OutputList;

  m_nBatchCnt++;

  char Msg[256];
  sprintf(Msg, "Train accuracy: %6.2lf, %5.3lf(batch index: %5d, \
total : %6d(%5.1lf), ep(%2d)",
    (double)nTP/(double)m_nBatch*100, Loss, m_nBatchCnt, nIndex+m_nBatch,
    (double)(nIndex+m_nBatch)/m_nMnistTrainTotal*100, m_nEpochCnt+1);
  std::cout << Msg << std::endl;
```

Main.cpp (10)

```
  if(nIndex+m_nBatch == m_nMnistTrainTotal) {
    m_nEpochCnt++;

    int nTP = 0;
    int i;
    for(i = 0 ; i < m_nMnistTestTotal ; i++) {
      int nResult = m_Perceptron.Forward(m_MnistTestInput[i]);
      if((m_MnistTestOutput[i]>4?1:0) == nResult) nTP++;
    }

    sprintf(Msg, "Test accuracy: %7.3lf\n",
      (double)nTP/(double)m_nMnistTestTotal*100.);
    std::cout << Msg << std::endl;

    m_pTestGraphLayer->m_Data[0].push_back
      ((double)nTP/(double)m_nMnistTestTotal*100);
    m_pTestGraphLayer->m_nCurrentCnt++;
    m_pTestGraphLayer->DrawBackgroundImage();
  }
```


Main.cpp (11)

```
  m_pTrainGraphLayer->m_Data[0].push_back((double)nTP/(double)m_nBatch*100);
  m_pTrainGraphLayer->m_Data[1].push_back(Loss);
  m_pTrainGraphLayer->m_nCurrentCnt++;
  m_pTrainGraphLayer->DrawBackgroundImage();

  m_pScene->Render();
  DrawSceneTextPos("Perceptron Test", CKgPoint(0, 0));

  CKhuGleWin::Update();
}

void CPerceptronTest::LoadMnistTrain() {
…
}

void CPerceptronTest::LoadMnistTest() {
…
}
```

Main.cpp (12)

```
int main() {
  char ExePath[MAX_PATH];
  GetModuleFileName(NULL, ExePath, MAX_PATH);

  int i;
  int LastBackSlash = -1;
  int nLen = strlen(ExePath);
  for(i = nLen-1 ; i >= 0 ; i--) {
    if(ExePath[i] == '\\') {
      LastBackSlash = i;
      break;
    }
  }
  if(LastBackSlash >= 0)
    ExePath[LastBackSlash] = '\0';

  CPerceptronTest *pPerceptronTest = new CPerceptronTest(640, 480, ExePath);

  KhuGleWinInit(pPerceptronTest);

  return 0;
}
```


Practice VIII

- Learning rate analysis

- Functional link network