20220610-机器学习

1.学习内容

1.1 机器学习

卷积网络

Net.cpp C++ 复制代码

```cpp
#include "Net.h"

bool InitializeKernel(double* pWeight, int nKernelSize, double dWeightBase)
{
    static int nScale = 5;
    for (int i = 0; i < nKernelSize; i++)
    {
        int nRandom = rand();
        double dTemp = static_cast<double>(nRandom % (nKernelSize * nScale));
        dTemp = (dTemp == 0) ? nScale : dTemp;
        dTemp = (dTemp > nKernelSize) ? nKernelSize / dTemp : nKernelSize / (dTemp + nKernelSize);
        pWeight[i] = dTemp * dWeightBase * 2.0;
        if (nRandom % 2) pWeight[i] = -pWeight[i];
        if (pWeight[i] > 1.0) pWeight[i] = sqrt(pWeight[i]);
    }
    return true;
}

bool InitializeLayer(Layer& stLayer,
    int nPreviousLayerMapNumber, int nOutputMapNumber,
    int nKernelWidth, int nKernelHeight,
    int nInputMapWidth, int nInputMapHeight,
    bool bIsPooling)
{
    int nInput = 4, nOutput = 1;
    if (!bIsPooling)
    {
        nInput = nPreviousLayerMapNumber * nKernelWidth * nKernelHeight;
        nOutput = nOutputMapNumber * nKernelWidth * nKernelHeight;
    }
    double dWeightBase = (nInput + nOutput) ? std::sqrt(6.0 / static_cast<double>(nInput + nOutput)) : 0.5;

    //保存图像宽高
    stLayer.nMapWidth = nInputMapWidth;
    stLayer.nMapHeight = nInputMapHeight;
    //保存图像数量
    stLayer.nMapCount = nOutputMapNumber;
    //保存卷积核宽高
    stLayer.nKernelWidth = nKernelWidth;
    stLayer.nKernelHeight = nKernelHeight;
    //卷积核数量
```

```
42      stLayer.nKernelCount = nPreviousLayerMapNumber * nOutputMapNumber;
43      if (stLayer.nKernelCount) stLayer.pKernel = new
    Kernel[stLayer.nKernelCount];
44      int nKernelSize = nKernelWidth * nKernelHeight;
45
46      for (int i = 0; i < nPreviousLayerMapNumber; i++)
47      {
48          for (int j = 0; j < nOutputMapNumber; j++)
49          {
50              if (nKernelSize)
51              {
52                  stLayer.pKernel[i * nOutputMapNumber + j].pWeight = new
    double[nKernelSize];
53                  InitializeKernel(stLayer.pKernel[i * nOutputMapNumber +
    j].pWeight, nKernelSize, dWeightBase);
54                  stLayer.pKernel[i + nOutputMapNumber + j].pDw = new
    double[nKernelSize];
55                  InitializeKernel(stLayer.pKernel[i * nOutputMapNumber +
    j].pDw, 0, sizeof(double)*nKernelSize);
56
57              }
58          }
59      }
60
61      int nMapSize = nInputMapWidth * nInputMapHeight;
62      stLayer.pMap = new Map[nOutputMapNumber];
63      for (int i = 0; i < nOutputMapNumber; i++)
64      {
65          stLayer.pMap[i].dBias = 0.0;
66          stLayer.pMap[i].dDb = 0.0;
67          if (nMapSize)
68          {
69              stLayer.pMap[i].pData = new double[nMapSize];
70              stLayer.pMap[i].pError = new double[nMapSize];
71              memset(stLayer.pMap[i].pData, 0, sizeof(double) * nMapSize);
72              memset(stLayer.pMap[i].pError, 0, sizeof(double) *
    nMapSize);
73          }
74      }
75      if (nMapSize)
76      {
77          stLayer.pMapCommon = new double[nMapSize];
78          memset(stLayer.pMapCommon, 0, sizeof(double) * nMapSize);
79      }
80      return true;
81  }
82
```

```
83  bool InitializeMnistNet(MnistNet& stMnistNet, int nWidth, int nHeight,
    int nClassNumber)
84  {
85      //初始化一个随机种子
86      std::srand(static_cast<unsigned int>
    (std::chrono::system_clock::now().time_since_epoch().count()));
87      //卷积核高度和宽度
88      int nKernelWidth = 0, nKernelHeight = 0;
89      //初始化输入层0
90      InitializeLayer(stMnistNet.stInputLayer_0,
91          0, 1, nKernelWidth, nKernelHeight, nWidth, nHeight);
92      //初始化卷积层1
93      nKernelWidth = nKernelHeight = 5;
94      InitializeLayer(stMnistNet.stConvLayer_1,
95          1, 6, nKernelWidth, nKernelHeight,
96          stMnistNet.stInputLayer_0.nMapWidth-nKernelWidth+1,
97          stMnistNet.stInputLayer_0.nMapHeight-nKernelHeight+1);
98      //初始化池化层2
99      nKernelWidth = nKernelHeight = 1;
100     InitializeLayer(stMnistNet.stPoolLayer_2,
101         1, 6, nKernelWidth, nKernelHeight,
102         stMnistNet.stConvLayer_1.nMapWidth / 2,
103         stMnistNet.stConvLayer_1.nMapHeight / 2, true);
104     //初始化卷积层3
105     nKernelWidth = nKernelHeight = 5;
106     InitializeLayer(stMnistNet.stConvLayer_3,
107         6, 16, nKernelWidth, nKernelHeight,
108         stMnistNet.stPoolLayer_2.nMapWidth - nKernelWidth + 1,
109         stMnistNet.stPoolLayer_2.nMapHeight - nKernelHeight + 1);
110     //初始化池化层4
111     nKernelWidth = nKernelHeight = 1;
112     InitializeLayer(stMnistNet.stPoolLayer_4,
113         6, 16, nKernelWidth, nKernelHeight,
114         stMnistNet.stConvLayer_3.nMapWidth / 2,
115         stMnistNet.stConvLayer_3.nMapHeight / 2);
116     //初始化卷积层5
117     nKernelWidth = nKernelHeight = 5;
118     InitializeLayer(stMnistNet.stConvLayer_5,
119         16, 120, nKernelWidth, nKernelHeight,
120         stMnistNet.stConvLayer_5.nMapWidth - nKernelWidth + 1,
121         stMnistNet.stConvLayer_5.nMapHeight - nKernelHeight + 1);
122     //初始化输出层6
123     nKernelWidth = nKernelHeight = 1;
124     InitializeLayer(stMnistNet.stOutputLayer_6,
125         120, nClassNumber, nKernelWidth, nKernelHeight,
126         1, 1);
127     return true;
128 }
```

```
129
130 bool trainModel(MnistNet& stMnistNet, MnistData& stMnistTrain,
    MnistData& stMnistTest, double dLearningRate, int nBatchSize, int
    nEpoch)
131 {
132     Time cTime;
133     int* pRandomSort = new int[stMnistTrain.nNumber];
134     int nBatchNumber = stMnistTrain.nNumber / nBatchSize;
135     for (int i = 0; i < nEpoch; i++)
136     {
137         //正常排序
138         for (int k = 0; k < stMnistTrain.nNumber; k++)
139         {
140             pRandomSort[k] = k;
141         }
142         //随机排序
143         for (int k = 0; k < stMnistTrain.nNumber; k++)
144         {
145             int nSortIndex = rand() % (stMnistTrain.nNumber - k) + k;
146             int nValue = pRandomSort[nSortIndex];
147             pRandomSort[nSortIndex] = pRandomSort[k];
148             pRandomSort[k] = nValue;
149         }
150         int nFinishRate = 0;
151         std::cout << std::endl;
152         std::cout << "-------------------------------------------" <<
    std::endl;
153         std::cout << "Epoch:[" << i << "] - ";
154         std::cout << "TrainSet Number:[" << stMnistTrain.nNumber << "] -
    ";
155         std::cout << "Mini Batch Size:[" << nBatchSize << "] - ";
156         std::cout << "Learning Rate:[" << dLearningRate << "] - " <<
    std::endl;
157         std::cout << "[FinishRate]: ";
158
159         cTime.ReSetTime();
160         for (int j = 0; j < nBatchNumber; j++)
161         {
162             //重置权重
163             ResetWeight(stMnistNet);
164             for (int k = 0; k < nBatchSize; k++)
165             {
166                 int nIndex = j * nBatchSize + k;
167                 memcpy_s(stMnistNet.stInputLayer_0.pMap[0].pData,
168                     sizeof(double) * stMnistNet.stInputLayer_0.nMapWidth
    * stMnistNet.stInputLayer_0.nMapHeight,
169                     stMnistTrain.pData[pRandomSort[nIndex]],
    sizeof(double) * stMnistTrain.nWidth * stMnistTrain.nHeight
```

```
170                 );
171                 ForwardPropagation(stMnistNet);
172                 BackwardPropagation(stMnistNet,
    stMnistTrain.pLabel[pRandomSort[nIndex]]);
173                 if (nIndex && (nIndex % (stMnistTrain.nNumber / 10)) ==
    0)
174                 {
175                     nFinishRate += 10;
176                     if (nFinishRate < 90)
177                         std::cout << nFinishRate << "% -> ";
178                     else
179                         std::cout << nFinishRate << "%..." << std::endl;
180                 }
181             }
182             UpdataWeight(stMnistNet, dLearningRate, nBatchSize);
183         }
184         std::cout << "Total Training Time:[" << cTime.GetTimeCount() <<
    "]Minutes..." << std::endl;
185         Predicts(stMnistNet, stMnistTest);
186         std::cout << "Epoch:[" << i << "] - End Iteration Training..."
    << std::endl;
187         std::cout << "-----------------------------------" << std::endl;
188         dLearningRate *= 0.85;
189     }
190     delete[] pRandomSort;
191     return true;
192 }
193
194 bool ResetWeight(MnistNet& stMnistNet)
195 {
196     ResetLayer(stMnistNet.stConvLayer_1);
197     ResetLayer(stMnistNet.stPoolLayer_2);
198     ResetLayer(stMnistNet.stConvLayer_3);
199     ResetLayer(stMnistNet.stPoolLayer_4);
200     ResetLayer(stMnistNet.stConvLayer_5);
201     ResetLayer(stMnistNet.stOutputLayer_6);
202     return true;
203 }
204
205 bool ResetLayer(Layer& stLayer)
206 {
207     for (int i = 0; i < stLayer.nKernelCount; i++)
208     {
209         memset(stLayer.pKernel[i].pDw, 0, sizeof(double) *
    stLayer.nKernelWidth * stLayer.nKernelHeight);
210     }
211     for (int i = 0; i < stLayer.nMapCount; i++)
212     {
```

```
213             stLayer.pMap[i].dDb = 0.0;
214         }
215         return true;
216     }
217
218     bool UpdataWeight(MnistNet& stMnistNet, double dLearningRate, int
    nBatchSize)
219     {
220         UpdateLayer(stMnistNet.stConvLayer_1, dLearningRate, nBatchSize);
221         UpdateLayer(stMnistNet.stPoolLayer_2, dLearningRate, nBatchSize);
222         UpdateLayer(stMnistNet.stConvLayer_3, dLearningRate, nBatchSize);
223         UpdateLayer(stMnistNet.stPoolLayer_4, dLearningRate, nBatchSize);
224         UpdateLayer(stMnistNet.stConvLayer_5, dLearningRate, nBatchSize);
225         UpdateLayer(stMnistNet.stOutputLayer_6, dLearningRate, nBatchSize);
226         return true;
227     }
228
229     bool UpdateLayer(Layer& stLayer, double dLearningRate, int nBatchSize)
230     {
231         static double dLambda = 0.005;
232         for (int i = 0; i < stLayer.nKernelCount; i++)
233         {
234             for (int j = 0; j < stLayer.nKernelWidth *
    stLayer.nKernelHeight; j++)
235             {
236                 double dTemp =
    GradientDescent(stLayer.pKernel[i].pWeight[j], stLayer.pKernel[i].pDw[j]
    / nBatchSize, dLearningRate, dLambda);
237                 stLayer.pKernel[i].pWeight[j] = dTemp;
238             }
239         }
240         for (int i = 0; i < stLayer.nMapCount; i++)
241         {
242             double dTemp = GradientDescent(stLayer.pMap[i].dBias,
    stLayer.pMap[i].dDb / nBatchSize, dLearningRate, dLambda);
243             stLayer.pMap[i].dBias = dTemp;
244         }
245         return true;
246     }
247
248     double GradientDescent(double dWeight, double dWd, double dLearningRate,
    double dLambda)
249     {
250         return dWeight - dLearningRate * (dWd + dLambda * dWeight);
251     }
252
253     bool ForwardPropagation(MnistNet& stMnistNet)
254     {
```

```
255         ForwardToConvolution(stMnistNet.stInputLayer_0,
    stMnistNet.stConvLayer_1);
256         ForwardToPooling(stMnistNet.stConvLayer_1,
    stMnistNet.stPoolLayer_2);
257         ForwardToConvolution(stMnistNet.stPoolLayer_2,
    stMnistNet.stConvLayer_3, NetConnectTable);
258         ForwardToPooling(stMnistNet.stConvLayer_3,
    stMnistNet.stPoolLayer_4);
259         ForwardToConvolution(stMnistNet.stPoolLayer_4,
    stMnistNet.stConvLayer_5);
260         ForwardToFullConnect(stMnistNet.stConvLayer_5,
    stMnistNet.stOutputLayer_6);
261         return true;
262     }
263
264     bool BackwardPropagation(MnistNet& stMnistNet, double* pLabelData)
265     {
266         for (int i = 0; i < stMnistNet.stOutputLayer_6.nMapCount; i++)
267         {
268             //计算输出值与实际值的误差
269             double dValue = stMnistNet.stOutputLayer_6.pMap[i].pData[0] -
    pLabelData[i];
270             dValue *=
    DerivativeTanh(stMnistNet.stOutputLayer_6.pMap[i].pData[0]);
271             stMnistNet.stOutputLayer_6.pMap[i].pError[0] = dValue;
272         }
273         BackwardToFullConnect(stMnistNet.stOutputLayer_6,
    stMnistNet.stConvLayer_5);
274         BackwardToConvolution(stMnistNet.stConvLayer_5,
    stMnistNet.stPoolLayer_4);
275         BackwardToPooling(stMnistNet.stPoolLayer_4,
    stMnistNet.stConvLayer_3);
276         BackwardToConvolution(stMnistNet.stConvLayer_3,
    stMnistNet.stPoolLayer_2,NetConnectTable);
277         BackwardToPooling(stMnistNet.stPoolLayer_2,
    stMnistNet.stConvLayer_1);
278         BackwardToConvolution(stMnistNet.stConvLayer_1,
    stMnistNet.stInputLayer_0);
279         return true;
280
281     }
282
283     bool ForwardToConvolution(Layer& stPreviousLayer, Layer& stCurrentLayer,
    const bool* pConnectTable)
284     {
285         int nMapSize = stCurrentLayer.nMapWidth * stCurrentLayer.nMapHeight;
286         int nIndex = 0;
287         for (int i = 0; i < stCurrentLayer.nMapCount; i++)
```

```
288         {
289             memset(stCurrentLayer.pMapCommon, 0, sizeof(double) * nMapSize);
290             for (int j = 0; j < stPreviousLayer.nMapCount; j++)
291             {
292                 nIndex = j * stCurrentLayer.nMapCount + i;
293                 if (pConnectTable != nullptr && !pConnectTable[nIndex])
294                     continue;
295                 ValidConvolution(stPreviousLayer.pMap[j].pData,
296                     stPreviousLayer.nMapWidth,
297                     stPreviousLayer.nMapHeight,
298                     stCurrentLayer.pKernel[nIndex].pWeight,
299                     stCurrentLayer.nKernelWidth,
300                     stCurrentLayer.nKernelHeight,
301                     stCurrentLayer.pMapCommon,
302                     stCurrentLayer.nMapWidth,
303                     stCurrentLayer.nMapHeight);
304             }
305             for (int k = 0; k < nMapSize; k++)
306             {
307                 stCurrentLayer.pMap[i].pData[k] =
    ActivationTanh(stCurrentLayer.pMapCommon[k] +
    stCurrentLayer.pMap[i].dBias);
308             }
309         }
310         return true;
311     }
312
313     bool ForwardToPooling(Layer& stPreviousLayer, Layer& stCurrentLayer)
314     {
315         for (int k = 0; k < stCurrentLayer.nMapCount; k++)
316         {
317             for (int i = 0; i < stCurrentLayer.nMapHeight; i++)
318             {
319                 for (int j = 0; j < stCurrentLayer.nMapWidth; j++)
320                 {
321                     double dMax = stPreviousLayer.pMap[k].pData[2 * i *
    stPreviousLayer.nMapWidth + 2 * j];
322                     for (int n = i * 2; n < 2 * (i + 1); n++)
323                     {
324                         for (int m = j * 2; m < 2 * (j + 1); m++)
325                         {
326                             double dTemp = stPreviousLayer.pMap[k].pData[n *
    stPreviousLayer.nMapWidth + m];
327                             if (dTemp > dMax) dMax = dTemp;
328                         }
329                     }
330                     stCurrentLayer.pMap[k].pData[i *
    stCurrentLayer.nMapWidth + j] = ActivationTanh(dMax);
```

```
            }
        }
    }
    return true;
}

bool ForwardToFullConnect(Layer& stPreviousLayer, Layer& stCurrentLayer)
{
    for (int i = 0; i < stCurrentLayer.nMapCount; i++)
    {
        double dSum = 0.0;
        for (int j = 0; j < stPreviousLayer.nMapCount; j++)
        {
            dSum += stPreviousLayer.pMap[j].pData[0] *
stCurrentLayer.pKernel[j * stCurrentLayer.nMapCount + i].pWeight[0];
        }
        dSum += stCurrentLayer.pMap[i].dBias;
        stCurrentLayer.pMap[i].pData[0] = ActivationTanh(dSum);
    }
    return true;
}

bool ValidConvolution(double* pInputData, int nInputWidth, int
nInputHeight, double* pKernelData, int nKernelWidth, int nKernelHeight,
double* pOutputData, int nOutputWidth, int nOutputHeight)
{
    double dSum;
    for (int i = 0; i < nOutputHeight; i++)
    {
        for (int j = 0; j < nOutputWidth; j++)
        {
            dSum = 0.0;
            for (int n = 0; n < nKernelHeight; n++)
            {
                for (int m = 0; m < nKernelWidth; m++)
                {
                    dSum += pInputData[(i + n) * nInputWidth + j + m] *
pKernelData[n * nKernelWidth + m];
                }
            }
            pOutputData[i * nOutputWidth + j] += dSum;
        }
    }
    return true;
}

double ActivationTanh(double dValue)
{
```

```
375         double _dValue1 = std::exp(dValue);
376         double _dValue2 = std::exp(-dValue);
377         return (_dValue1 - _dValue2) / (_dValue1 + _dValue2);
378     }
379 
380     double DerivativeTanh(double dValue)
381     {
382         return 1.0 - dValue * dValue;
383     }
384 
385     double ActivationRelu(double dValue)
386     {
387         return(dValue > 0.0) ? dValue : 0.0;
388     }
389 
390     double DerivativeRelu(double dValue)
391     {
392         return (dValue > 0.0) ? 1.0 : 0.0;
393     }
394 
395     double ActivationSigmoid(double dValue)
396     {
397         return (1.0 / (1.0 + std::exp(-dValue)));
398     }
399 
400     double DerivativeSigmoid(double dValue)
401     {
402         return dValue*(1.0 - dValue);
403     }
404 
405     bool BackwardToFullConnect(Layer& stCurrentLayer, Layer&
    stPreviousLayer)
406     {
407         //层误差
408         for (int i = 0; i < stPreviousLayer.nMapCount; i++)
409         {
410             stPreviousLayer.pMap[i].pError[0] = 0.0;
411             for (int j = 0; j < stCurrentLayer.nMapCount; j++)
412             {
413                 double dValue = stCurrentLayer.pMap[j].pError[0] *
    stCurrentLayer.pKernel[i * stCurrentLayer.nMapCount + j].pWeight[0];
414                 stPreviousLayer.pMap[i].pError[0] += dValue;
415             }
416             stPreviousLayer.pMap[i].pError[0] *=
    DerivativeTanh(stPreviousLayer.pMap[i].pData[0]);
417         }
418 
419         //DW
```

```
420         for (int i = 0; i < stPreviousLayer.nMapCount; i++)
421         {
422             for (int j = 0; j < stCurrentLayer.nMapCount; j++)
423             {
424                 stCurrentLayer.pKernel[i * stCurrentLayer.nMapCount +
    j].pDw[0] += stCurrentLayer.pMap[j].pError[0] *
    stPreviousLayer.pMap[i].pData[0];
425             }
426         }
427 
428         //总误差
429         for (int i = 0; i < stCurrentLayer.nMapCount; i++)
430         {
431             stCurrentLayer.pMap[i].dDb += stCurrentLayer.pMap[i].pError[0];
432         }
433         return true;
434     }
435 
436     bool BackwardToConvolution(Layer& stCurrentLayer, Layer&
    stPreviousLayer, const bool* pConnectTable)
437     {
438         for (int i = 0; i < stPreviousLayer.nMapCount; i++)
439         {
440             memset(stPreviousLayer.pMapCommon, 0, sizeof(double) *
    stPreviousLayer.nMapWidth * stPreviousLayer.nMapHeight);
441             for (int j = 0; j < stCurrentLayer.nMapCount; j++)
442             {
443                 int nIndex = i * stCurrentLayer.nMapCount + j;
444                 if (pConnectTable != nullptr && !pConnectTable[nIndex])
445                     continue;
446                 for (int n = 0; n < stCurrentLayer.nMapHeight; n++)
447                 {
448                     for (int m = 0; m < stCurrentLayer.nMapWidth; m++)
449                     {
450                         double dError = stCurrentLayer.pMap[j].pError[n *
    stCurrentLayer.nMapWidth + m];
451                         for (int y = 0; y < stCurrentLayer.nKernelHeight;
    y++)
452                         {
453                             for (int x = 0; x < stCurrentLayer.nKernelWidth;
    x++)
454                             {
455                                 double dValue = dError *
    stCurrentLayer.pKernel[nIndex].pWeight[y * stCurrentLayer.nKernelWidth +
    x];
456                                 stPreviousLayer.pMapCommon[(n + y) *
    stPreviousLayer.nMapWidth + m + x] += dValue;
457                             }
```

```
458                     }
459                 }
460             }
461         }
462         for (int k = 0; k < stPreviousLayer.nMapHeight *
    stPreviousLayer.nMapWidth; k++)
463             stPreviousLayer.pMap[i].pError[k] =
    stPreviousLayer.pMapCommon[k] *
    DerivativeTanh(stPreviousLayer.pMap[i].pData[k]);
464     }
465     //DW
466     for (int i = 0; i < stPreviousLayer.nMapCount; i++)
467     {
468         for (int j = 0; j < stCurrentLayer.nMapCount; j++)
469         {
470             int nIndex = i * stCurrentLayer.nMapCount + j;
471             if (pConnectTable != nullptr && !pConnectTable[nIndex])
472                 continue;
473             ValidConvolution(stPreviousLayer.pMap[i].pData,
474                 stPreviousLayer.nMapWidth,
475                 stPreviousLayer.nMapHeight,
476                 stCurrentLayer.pMap[j].pError,
477                 stCurrentLayer.nMapWidth,
478                 stCurrentLayer.nMapHeight,
479                 stCurrentLayer.pKernel[nIndex].pDw,
480                 stCurrentLayer.nKernelWidth,
481                 stCurrentLayer.nKernelHeight);
482         }
483     }
484     //总误差
485     for (int i = 0; i < stCurrentLayer.nMapCount; i++)
486     {
487         double dSum = 0.0;
488         for (int k = 0; k < stCurrentLayer.nMapWidth *
    stCurrentLayer.nMapHeight; k++)
489         {
490             dSum += stCurrentLayer.pMap[i].pError[k];
491         }
492         stCurrentLayer.pMap[i].dDb += dSum;
493     }
494     return true;
495 }
496
497 bool BackwardToPooling(Layer& stCurrentLayer, Layer& stPreviousLayer)
498 {
499     for (int k = 0; k < stCurrentLayer.nMapCount; k++)
500     {
501         for (int i = 0; i < stCurrentLayer.nMapHeight; i++)
```

```
502             {
503                 for (int j = 0; j < stCurrentLayer.nMapWidth; j++)
504                 {
505                     int nHeight = 2 * i, nWidth = 2 * j;
506                     double dMax = stPreviousLayer.pMap[k].pData[nHeight *
    stPreviousLayer.nMapWidth + nWidth];
507                     for (int n = i * 2; n < 2 * (i + 1); n++)
508                     {
509                         for (int m = j * 2; m < 2 * (j + 1); m++)
510                         {
511                             if (stPreviousLayer.pMap[k].pData[n *
    stPreviousLayer.nMapWidth + m] > dMax)
512                             {
513                                 nHeight = m;
514                                 nWidth = n;
515                                 dMax = stPreviousLayer.pMap[k].pData[n *
    stPreviousLayer.nMapWidth + m];
516                             }
517                             else
518                             {
519                                 stPreviousLayer.pMap[k].pError[n *
    stPreviousLayer.nMapWidth + m] = 0.0;
520                             }
521                         }
522                     }
523                     double dValue = stCurrentLayer.pMap[k].pError[i *
    stCurrentLayer.nMapWidth + j] * DerivativeTanh(dMax);
524                     stPreviousLayer.pMap[k].pError[nHeight *
    stPreviousLayer.nMapWidth + nWidth] = dValue;
525                 }
526             }
527         }
528         return true;
529     }
530
531     bool Predicts(MnistNet& stMnistNet, MnistData& stMnistData)
532     {
533         //结果矩阵
534         int nMatrixSize = stMnistData.nClassNumber *
    stMnistData.nClassNumber;
535         int* pResultMatrix = new int[nMatrixSize];
536         memset(pResultMatrix, 0, sizeof(int) * nMatrixSize);
537         //成功预测的数量
538         int nSuccessNumber = 0;
539         Time cTime;
540         for (int i = 0; i < stMnistData.nNumber; i++)
541         {
542             //将图像数据复制到输出层
```

```
543             memcpy_s(stMnistNet.stInputLayer_0.pMap[0].pData,
544                 sizeof(double) * stMnistData.nWidth * stMnistData.nHeight,
545                 stMnistData.pData[i],
546                 sizeof(double) * stMnistData.nWidth * stMnistData.nHeight);
547             ForwardPropagation(stMnistNet);
548             int nPredictsIndex = GetOutputIndex(stMnistNet.stOutputLayer_6);
549             int nActualIndex = GetActualIndex(stMnistData.pLabel[i],
    stMnistData.nClassNumber);
550             if (nPredictsIndex == nActualIndex)
```

main.cpp C++ 复制代码

```cpp
#include "Net.h";
int main(int argc,char* argv[])
{
    MnistData stTrainSet;
    MnistData stTestSet;

    ReadMnistImage(stTrainSet, "train-images.idx3-ubyte");
    ReadMnistLabel(stTrainSet, "train-labels.idx1-ubyte");

    ReadMnistImage(stTestSet, "t10k-images.idx3-ubyte");
    ReadMnistLabel(stTestSet, "t10k-labels.idx1-ubyte");

    MnistNet stMnistNet;

    InitializeMnistNet(stMnistNet, stTrainSet.nWidth, stTrainSet.nHeight,
stTrainSet.nClassNumber);

    int nBatch_size = 10;
    double dLearningRate = 0.01 * std::sqrt(nBatch_size);

    trainModel(stMnistNet, stTrainSet, stTestSet, dLearningRate,
nBatch_size);

    ReleaseMnistNet(stMnistNet);

    ReleaseMnistData(stTrainSet);
    ReleaseMnistData(stTestSet);

    getchar();
    return 0;
}
```

2.结果描述

今天完成了Net类的实现，但在运行时还存在问题。争取明天将代码消化吸收。