20220609-机器学习

1.学习内容
  1.1 机器学习
    卷积网络
2.结果描述

1.学习内容

1.1 机器学习

卷积网络

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     return false;
133 }
134
135 bool ResetWeight(MnistNet& stMnistNet)
136 {
137     ResetLayer(stMnistNet.stConvLayer_1);
138     ResetLayer(stMnistNet.stPoolLayer_2);
139     ResetLayer(stMnistNet.stConvLayer_3);
140     ResetLayer(stMnistNet.stPoolLayer_4);
141     ResetLayer(stMnistNet.stConvLayer_5);
142     ResetLayer(stMnistNet.stOutputLayer_6);
143     return true;
144 }
145
146 bool ResetLayer(Layer& stLayer)
147 {
148     for (int i = 0; i < stLayer.nKernelCount; i++)
149     {
150         memset(stLayer.pKernel[i].pDw, 0, sizeof(double) *
    stLayer.nKernelWidth * stLayer.nKernelHeight);
151     }
152     for (int i = 0; i < stLayer.nMapCount; i++)
153     {
154         stLayer.pMap[i].dDb = 0.0;
155     }
156     return true;
157 }
158
159 bool UpdataWeight(MnistNet& stMnistNet, double dLearningRate, int
    nBatchSize)
160 {
161     UpdateLayer(stMnistNet.stConvLayer_1, dLearningRate, nBatchSize);
162     UpdateLayer(stMnistNet.stPoolLayer_2, dLearningRate, nBatchSize);
163     UpdateLayer(stMnistNet.stConvLayer_3, dLearningRate, nBatchSize);
164     UpdateLayer(stMnistNet.stPoolLayer_4, dLearningRate, nBatchSize);
165     UpdateLayer(stMnistNet.stConvLayer_5, dLearningRate, nBatchSize);
166     UpdateLayer(stMnistNet.stOutputLayer_6, dLearningRate, nBatchSize);
167     return true;
168 }
169
170 bool UpdateLayer(Layer& stLayer, double dLearningRate, int nBatchSize)
171 {
172     static double dLambda = 0.005;
```

```
173         for (int i = 0; i < stLayer.nKernelCount; i++)
174         {
175             for (int j = 0; j < stLayer.nKernelWidth *
    stLayer.nKernelHeight; j++)
176             {
177                 double dTemp =
    GradientDescent(stLayer.pKernel[i].pWeight[j], stLayer.pKernel[i].pDw[j]
    / nBatchSize, dLearningRate, dLambda);
178                 stLayer.pKernel[i].pWeight[j] = dTemp;
179             }
180         }
181         for (int i = 0; i < stLayer.nMapCount; i++)
182         {
183             double dTemp = GradientDescent(stLayer.pMap[i].dBias,
    stLayer.pMap[i].dDb / nBatchSize, dLearningRate, dLambda);
184             stLayer.pMap[i].dBias = dTemp;
185         }
186         return true;
187     }
188
189     double GradientDescent(double dWeight, double dWd, double dLearningRate,
    double dLambda)
190     {
191         return dWeight - dLearningRate * (dWd + dLambda * dWeight);
192     }
193
194     bool ForwardPropagation(MnistNet& stMnistNet)
195     {
196         ForwardToConvolution(stMnistNet.stInputLayer_0,
    stMnistNet.stConvLayer_1);
197         ForwardToPooling(stMnistNet.stConvLayer_1,
    stMnistNet.stPoolLayer_2);
198         ForwardToConvolution(stMnistNet.stPoolLayer_2,
    stMnistNet.stConvLayer_3, NetConnectTable);
199         ForwardToPooling(stMnistNet.stConvLayer_3,
    stMnistNet.stPoolLayer_4);
200         ForwardToConvolution(stMnistNet.stPoolLayer_4,
    stMnistNet.stConvLayer_5);
201         ForwardToFullConnect(stMnistNet.stConvLayer_5,
    stMnistNet.stOutputLayer_6);
202         return true;
203     }
204
205     bool BackwardPropagation(MnistNet& stMnistNet, double* pLabelData)
206     {
207         for (int i = 0; i < stMnistNet.stOutputLayer_6.nMapCount; i++)
208         {
209             //计算输出值与实际值的误差
```

```
210             double dValue = stMnistNet.stOutputLayer_6.pMap[i].pData[0] -
    pLabelData[i];
211             dValue *=
    DerivativeTanh(stMnistNet.stOutputLayer_6.pMap[i].pData[0]);
212             stMnistNet.stOutputLayer_6.pMap[i].pError[0] = dValue;
213         }
214         BackwardToFullConnect(stMnistNet.stOutputLayer_6,
    stMnistNet.stConvLayer_5);
215         BackwardToConvolution(stMnistNet.stConvLayer_5,
    stMnistNet.stPoolLayer_4);
216         BackwardToPooling(stMnistNet.stPoolLayer_4,
    stMnistNet.stConvLayer_3);
217         BackwardToConvolution(stMnistNet.stConvLayer_3,
    stMnistNet.stPoolLayer_2,NetConnectTable);
218         BackwardToPooling(stMnistNet.stPoolLayer_2,
    stMnistNet.stConvLayer_1);
219         BackwardToConvolution(stMnistNet.stConvLayer_1,
    stMnistNet.stInputLayer_0);
220         return true;
221
222     }
223
224     bool ForwardToConvolution(Layer& stPreviousLayer, Layer& stCurrentLayer,
    const bool* pConnectTable)
225     {
226         int nMapSize = stCurrentLayer.nMapWidth * stCurrentLayer.nMapHeight;
227         int nIndex = 0;
228         for (int i = 0; i < stCurrentLayer.nMapCount; i++)
229         {
230             memset(stCurrentLayer.pMapCommon, 0, sizeof(double) * nMapSize);
231             for (int j = 0; j < stPreviousLayer.nMapCount; j++)
232             {
233                 nIndex = j * stCurrentLayer.nMapCount + i;
234                 if (pConnectTable != nullptr && !pConnectTable[nIndex])
235                     continue;
236                 ValidConvolution(stPreviousLayer.pMap[j].pData,
237                     stPreviousLayer.nMapWidth,
238                     stPreviousLayer.nMapHeight,
239                     stCurrentLayer.pKernel[nIndex].pWeight,
240                     stCurrentLayer.nKernelWidth,
241                     stCurrentLayer.nKernelHeight,
242                     stCurrentLayer.pMapCommon,
243                     stCurrentLayer.nMapWidth,
244                     stCurrentLayer.nMapHeight);
245             }
246             for (int k = 0; k < nMapSize; k++)
247             {
```

```
248             stCurrentLayer.pMap[i].pData[k] =
    ActivationTanh(stCurrentLayer.pMapCommon[k] +
    stCurrentLayer.pMap[i].dBias);
249         }
250     }
251     return true;
252 }
253
254 bool ForwardToPooling(Layer& stPreviousLayer, Layer& stCurrentLayer)
255 {
256     for (int k = 0; k < stCurrentLayer.nMapCount; k++)
257     {
258         for (int i = 0; i < stCurrentLayer.nMapHeight; i++)
259         {
260             for (int j = 0; j < stCurrentLayer.nMapWidth; j++)
261             {
262                 double dMax = stPreviousLayer.pMap[k].pData[2 * i *
    stPreviousLayer.nMapWidth + 2 * j];
263                 for (int n = i * 2; n < 2 * (i + 1); n++)
264                 {
265                     for (int m = j * 2; m < 2 * (j + 1); m++)
266                     {
267                         double dTemp = stPreviousLayer.pMap[k].pData[n *
    stPreviousLayer.nMapWidth + m];
268                         if (dTemp > dMax) dMax = dTemp;
269                     }
270                 }
271                 stCurrentLayer.pMap[k].pData[i *
    stCurrentLayer.nMapWidth + j] = ActivationTanh(dMax);
272             }
273         }
274     }
275     return true;
276 }
277
278 bool ForwardToFullConnect(Layer& stPreviousLayer, Layer& stCurrentLayer)
279 {
280     for (int i = 0; i < stCurrentLayer.nMapCount; i++)
281     {
282         double dSum = 0.0;
283         for (int j = 0; j < stPreviousLayer.nMapCount; j++)
284         {
285             dSum += stPreviousLayer.pMap[j].pData[0] *
    stCurrentLayer.pKernel[j * stCurrentLayer.nMapCount + i].pWeight[0];
286         }
287         dSum += stCurrentLayer.pMap[i].dBias;
288         stCurrentLayer.pMap[i].pData[0] = ActivationTanh(dSum);
289     }
```

```
290         return true;
291     }
292 
293     bool ValidConvolution(double* pInputData, int nInputWidth, int
    nInputHeight, double* pKernelData, int nKernelWidth, int nKernelHeight,
    double* pOutputData, int nOutputWidth, int nOutputHeight)
294     {
295         double dSum;
296         for (int i = 0; i < nOutputHeight; i++)
297         {
298             for (int j = 0; j < nOutputWidth; j++)
299             {
300                 dSum = 0.0;
301                 for (int n = 0; n < nKernelHeight; n++)
302                 {
303                     for (int m = 0; m < nKernelWidth; m++)
304                     {
305                         dSum += pInputData[(i + n) * nInputWidth + j + m] *
    pKernelData[n * nKernelWidth + m];
306                     }
307                 }
308                 pOutputData[i * nOutputWidth + j] += dSum;
309             }
310         }
311         return true;
312     }
313 
314     double ActivationTanh(double dValue)
315     {
316         double _dValue1 = std::exp(dValue);
317         double _dValue2 = std::exp(-dValue);
318         return (_dValue1 - _dValue2) / (_dValue1 + _dValue2);
319     }
320 
321     double DerivativeTanh(double dValue)
322     {
323         return 1.0 - dValue * dValue;
324     }
325 
326     double ActivationRelu(double dValue)
327     {
328         return(dValue > 0.0) ? dValue : 0.0;
329     }
330 
331     double DerivativeRelu(double dValue)
332     {
333         return (dValue > 0.0) ? 1.0 : 0.0;
334     }
```

```
335
336 double ActivationSigmoid(double dValue)
337 {
338     return (1.0 / (1.0 + std::exp(-dValue)));
339 }
340
341 double DerivativeSigmoid(double dValue)
342 {
343     return dValue*(1.0 - dValue);
344 }
345
346 bool BackwardToFullConnect(Layer& stCurrentLayer, Layer&
    stPreviousLayer)
347 {
348     //层误差
349     for (int i = 0; i < stPreviousLayer.nMapCount; i++)
350     {
351         stPreviousLayer.pMap[i].pError[0] = 0.0;
352         for (int j = 0; j < stCurrentLayer.nMapCount; j++)
353         {
354             double dValue = stCurrentLayer.pMap[j].pError[0] *
    stCurrentLayer.pKernel[i * stCurrentLayer.nMapCount + j].pWeight[0];
355             stPreviousLayer.pMap[i].pError[0] += dValue;
356         }
357         stPreviousLayer.pMap[i].pError[0] *=
    DerivativeTanh(stPreviousLayer.pMap[i].pData[0]);
358     }
359
360     //DW
361     for (int i = 0; i < stPreviousLayer.nMapCount; i++)
362     {
363         for (int j = 0; j < stCurrentLayer.nMapCount; j++)
364         {
365             stCurrentLayer.pKernel[i * stCurrentLayer.nMapCount +
    j].pDw[0] += stCurrentLayer.pMap[j].pError[0] *
    stPreviousLayer.pMap[i].pData[0];
366         }
367     }
368
369     //总误差
370     for (int i = 0; i < stCurrentLayer.nMapCount; i++)
371     {
372         stCurrentLayer.pMap[i].dDb += stCurrentLayer.pMap[i].pError[0];
373     }
374 }
375
376 bool BackwardToConvolution(Layer& stCurrentLayer, Layer&
    stPreviousLayer, const bool* pConnectTable)
```

```
{
    for (int i = 0; i < stPreviousLayer.nMapCount; i++)
    {
        memset(stPreviousLayer.pMapCommon, 0, sizeof(double) * stPreviousLayer.nMapWidth * stPreviousLayer.nMapHeight);
        for (int j = 0; j < stCurrentLayer.nMapCount; j++)
        {
            int nIndex = i * stCurrentLayer.nMapCount + j;
            if (pConnectTable != nullptr && !pConnectTable[nIndex])
                continue;
            for (int n = 0; n < stCurrentLayer.nMapHeight; n++)
            {
                for (int m = 0; m < stCurrentLayer.nMapWidth; m++)
                {
                    double dError = stCurrentLayer.pMap[j].pError[n * stCurrentLayer.nMapWidth + m];
                    for (int y = 0; y < stCurrentLayer.nKernelHeight; y++)
                    {
                        for (int x = 0; x < stCurrentLayer.nKernelWidth; x++)
                        {
                            double dValue = dError * stCurrentLayer.pKernel[nIndex].pWeight[y * stCurrentLayer.nKernelWidth + x];
                            stPreviousLayer.pMapCommon[(n + y) * stPreviousLayer.nMapWidth + m + x] += dValue;
                        }
                    }
                }
            }
        }
        for (int k = 0; k < stPreviousLayer.nMapHeight * stPreviousLayer.nMapWidth; k++)
            stPreviousLayer.pMap[i].pError[k] = stPreviousLayer.pMapCommon[k] * DerivativeTanh(stPreviousLayer.pMap[i].pData[k]);
    }
    //DW
    for (int i = 0; i < stPreviousLayer.nMapCount; i++)
    {
        for (int j = 0; j < stCurrentLayer.nMapCount; j++)
        {
            int nIndex = i * stCurrentLayer.nMapCount + j;
            if (pConnectTable != nullptr && !pConnectTable[nIndex])
                continue;
            ValidConvolution(stPreviousLayer.pMap[i].pData,
                stPreviousLayer.nMapWidth,
```

```
                    stPreviousLayer.nMapHeight,
                    stCurrentLayer.pMap[j].pError,
                    stCurrentLayer.nMapWidth,
                    stCurrentLayer.nMapHeight,
                    stCurrentLayer.pKernel[nIndex].pDw,
                    stCurrentLayer.nKernelWidth,
                    stCurrentLayer.nKernelHeight);
            }
        }
        //总误差
        for (int i = 0; i < stCurrentLayer.nMapCount; i++)
        {
            double dSum = 0.0;
            for (int k = 0; k < stCurrentLayer.nMapWidth *
stCurrentLayer.nMapHeight; k++)
            {
                dSum += stCurrentLayer.pMap[i].pError[k];
            }
            stCurrentLayer.pMap[i].dDb += dSum;
        }
        return true;
    }

    bool BackwardToPooling(Layer& stCurrentLayer, Layer& stPreviousLayer)
    {
        for (int k = 0; k < stCurrentLayer.nMapCount; k++)
        {
            for (int i = 0; i < stCurrentLayer.nMapHeight; i++)
            {
                for (int j = 0; j < stCurrentLayer.nMapWidth; j++)
                {
                    int nHeight = 2 * i, nWidth = 2 * j;
                    double dMax = stPreviousLayer.pMap[k].pData[nHeight *
stPreviousLayer.nMapWidth + nWidth];
                    for (int n = i * 2; n < 2 * (i + 1); n++)
                    {
                        for (int m = j * 2; m < 2 * (j + 1); m++)
                        {
                            if (stPreviousLayer.pMap[k].pData[n *
stPreviousLayer.nMapWidth + m] > dMax)
                            {
                                nHeight = m;
                                nWidth = n;
                                dMax = stPreviousLayer.pMap[k].pData[n *
stPreviousLayer.nMapWidth + m];
                            }
                            else
                            {
```

```
459                                 stPreviousLayer.pMap[k].pError[n * stPreviousLayer.nMapWidth + m] = 0.0;
460                             }
461                         }
462                     }
463                     double dValue = stCurrentLayer.pMap[k].pError[i * stCurrentLayer.nMapWidth + j] * DerivativeTanh(dMax);
464                     stPreviousLayer.pMap[k].pError[nHeight * stPreviousLayer.nMapWidth + nWidth] = dValue;
465                 }
466             }
467         }
468         return true;
469     }
470
471     bool Predicts(MnistNet& stMnistNet, MnistData& stMnistData)
472     {
473         return false;
474     }
475
476     int GetOutputIndex(Layer& stOutputLayer)
477     {
478         double dMaxValue = stOutputLayer.pMap[0].pData[0];
479         int nMaxIndex = 0;
480         for (int i = 1; i < stOutputLayer.nMapCount; i++)
481         {
482             if (stOutputLayer.pMap[i].pData[0] > dMaxValue)
483             {
484                 dMaxValue = stOutputLayer.pMap[i].pData[0];
485                 nMaxIndex = i;
486             }
487         }
488         return nMaxIndex;
489     }
490
491     int GetActualIndex(double* pLabel, int nClassNumber)
492     {
493         int nMaxIndex = 0;
494         double dMaxValue = pLabel[0];
495         for (int i = 1; i < nClassNumber; i++)
496         {
497             if (pLabel[i] > dMaxValue)
498             {
499                 dMaxValue = pLabel[i];
500                 nMaxIndex = i;
501             }
502         }
503         return nMaxIndex;
```

```
504   }
505
506   bool ReleaseMnistNet(MnistNet& stMnistNet)
507   {
508       ReleaseLayer(stMnistNet.stInputLayer_0);
509       ReleaseLayer(stMnistNet.stConvLayer_1);
510       ReleaseLayer(stMnistNet.stPoolLayer_2);
511       ReleaseLayer(stMnistNet.stConvLayer_3);
512       ReleaseLayer(stMnistNet.stPoolLayer_4);
513       ReleaseLayer(stMnistNet.stConvLayer_5);
514       ReleaseLayer(stMnistNet.stOutputLayer_6);
515       return true;
516   }
517
518   bool ReleaseLayer(Layer& stLayer)
519   {
520       if (stLayer.pKernel)
521       {
522           for (int i = 0; i < stLayer.nKernelCount; i++)
523           {
524               if(stLayer.pKernel[i].pWeight)
525                   delete[] stLayer.pKernel[i].pWeight;
526               if (stLayer.pKernel[i].pDw)
527                   delete[] stLayer.pKernel[i].pDw;
528           }
529           delete[] stLayer.pKernel;
530       }
531
532       if (stLayer.pMap)
533       {
534           for (int i = 0; i < stLayer.nMapCount; i++)
535           {
536               if (stLayer.pMap[i].pData)
537                   delete[] stLayer.pMap[i].pData;
538               if (stLayer.pMap->pError)
539                   delete[] stLayer.pMap[i].pError;
540           }
541           delete[] stLayer.pMap;
542       }
543       if (stLayer.pMapCommon)
544           delete[] stLayer.pMapCommon;
545
546       stLayer.Release();
547       return true;
548   }
549
```

2.结果描述

Net类目前还差训练以及预测代码的代码，明天继续。