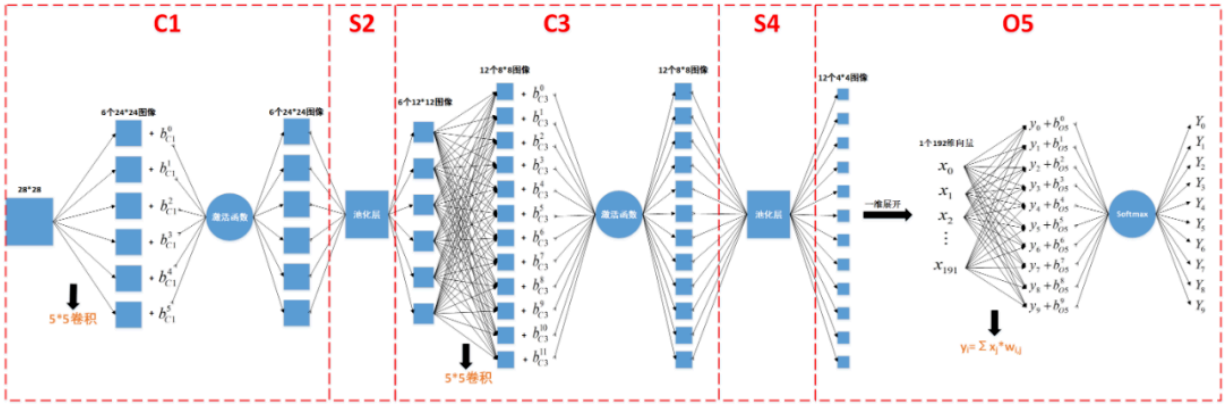
卷積神經網路原理及其C++/Opencv實作(6)—前向傳播程式碼實現

原創 sdff20201029 萌萌噠程序猴 2021-04-01 21:03

首先列出本系列部落格的連結：

1. 卷積神經網路原理及其C++/Opencv實作(1)
2. 卷積神經網路原理及其C++/Opencv實作(2)
3. 卷積神經網路原理及其C++/Opencv實作(3)
4. 卷積神經網路原理及其C++/Opencv實作(4)—誤反向傳播法
5. 卷積神經網路原理及其C++/Opencv實作(5)—參數更新

在以上文章中，我們基本上把5層網路的原理、公式推導講過了，從本文開始，我們來講一下基於C++和Opencv的5層卷積神經網路實作吧~



1. 結構體定義

(1) 卷積層的結構體

```
typedef struct convolutional_layer
{
```

```
  int inputWidth;   //输入图像的宽
  int inputHeight;  //输入图像的长
  int mapSize;      //卷积核的尺寸

  int inChannels;   //输入图像的数目
  int outChannels;  //输出图像的数目

  vector<vector<Mat>> mapData;  //四维float数组，卷积核本身是二维数据，m*n哥卷积核

  Mat basicData; //偏置，个数为outChannels， 一维float数组

  bool isFullConnect; //是否为全连接

  vector<Mat> v;   //进入激活函数的输入值,三维数组float型
  vector<Mat> y;   //激活函数后神经元的输出，三维数组float型
  vector<Mat> d;     // 网络的局部梯度,三维数组float型

}CovLayer;
```

(2) 池化層的結構體

```
typedef struct pooling_layer
{
  int inputWidth;   //输入图像的宽
  int inputHeight;  //输入图像的长
  int mapSize;      //卷积核的大小

  int inChannels;   //输入图像的数目
```

```
  int outChannels;  //输出图像的数目

  int poolType;     //池化的方法

  Mat basicData;    //偏置, 一维float数组

  vector<Mat> y;    //采样函数后神经元的输出,无激活函数，三维数组float型
  vector<Mat> d;    //网络的局部梯度,三维数组float型
  vector<Mat> max_position;    // 最大值模式下最大值的位置，三维数组float型

}PoolLayer;

```

(3) 輸出層的結構

```
typedef struct nn_layer
{
  int inputNum;   //输入数据的数目
  int outputNum;  //输出数据的数目

  Mat wData;            // 权重数据，为一个inputNum*outputNum大小
  Mat basicData;        //偏置，大小为outputNum大小

  Mat v;      // 进入激活函数的输入值
  Mat y;      // 激活函数后神经元的输出
  Mat d;      // 网络的局部梯度

  bool isFullConnect; //是否为全连接
}OutLayer;
```

(4) 5層網絡的結構體

```
typedef struct cnn_network
{
  int layerNum;
  CovLayer C1;
  PoolLayer S2;
  CovLayer C3;
  PoolLayer S4;
  OutLayer O5;

  Mat e;   // 训练误差
  Mat L;   // 瞬时误差能量
}CNN;
```

(5) 訓練參數的結構體

```
typedef struct train_opts
{
  int numepochs; // 训练的迭代次数
  float alpha; // 学习率
}CNNOpts;
```

2. 5層網路的初始化

(1) 卷積層結構體初始化

```
CovLayer initCovLayer(int inputWidth, int inputHeight, int mapSize, int inCha
{
  CovLayer covL;

  covL.inputHeight = inputHeight;
  covL.inputWidth = inputWidth;
  covL.mapSize = mapSize;

  covL.inChannels = inChannels;
  covL.outChannels = outChannels;

  covL.isFullConnect = true;   // 默认为全连接

  // 权重空间的初始化，先行再列调用，[r][c]
  srand((unsigned)time(NULL));   //设置随机数种子
  for(int i = 0; i < inChannels; i++)   //输入通道数
  {
    vector<Mat> tmp;
    for(int j = 0; j < outChannels; j++)   //输出通道数
    {
      Mat tmpmat(mapSize, mapSize, CV_32FC1);  //初始化一个mapSize*mapSize的二
      for(int r = 0; r < mapSize; r++)   //卷积核的高
      {
        for(int c = 0; c < mapSize; c++)  //卷积核的宽
        {
          //使用随机数初始化卷积核
          float randnum=(((float)rand()/(float)RAND_MAX)-0.5)*2;    //生成-1~
          tmpmat.ptr<float>(r)[c] = randnum*sqrt(6.0/(mapSize*mapSize*(inChar
        }
      }
```

```
      tmp.push_back(tmpmat.clone());
    }
    covL.mapData.push_back(tmp);
  }

  covL.basicData = Mat::zeros(1, outChannels, CV_32FC1);   //初始化卷积层偏置的

  int outW = inputWidth - mapSize + 1;   //valid模式下卷积层输出的宽
  int outH = inputHeight - mapSize + 1;  //valid模式下卷积层输出的高

  Mat tmpmat2 = Mat::zeros(outH, outW, CV_32FC1);
  for(int i = 0; i < outChannels; i++)
  {
    covL.d.push_back(tmpmat2.clone());  //初始化局部梯度
    covL.v.push_back(tmpmat2.clone());  //初始化输入激活函数之前的值
    covL.y.push_back(tmpmat2.clone());  //初始化输入激活函数之后的值
  }

  return covL;   //返回初始化之后的卷积层结构体
}
```

(2) 池化層結構體初始化

```
PoolLayer initPoolLayer(int inputWidth, int inputHeight, int mapSize, int inC
{
  PoolLayer poolL;

  poolL.inputHeight=inputHeight;    //输入高度
```

```
    poolL.inputWidth=inputWidth;       //输入宽度
    poolL.mapSize=mapSize;             //卷积核尺寸，池化层相当于做一个特殊的卷积操作
    poolL.inChannels=inChannels;       //输入通道
    poolL.outChannels=outChannels;     //输出通道
    poolL.poolType=poolType;           //最大值模式1/平均值模式0

    poolL.basicData = Mat::zeros(1, outChannels, CV_32FC1);    //池化层无偏置，无

    int outW = inputWidth/mapSize;   //池化层的卷积核为2*2
    int outH = inputHeight/mapSize;

    Mat tmpmat = Mat::zeros(outH, outW, CV_32FC1);
    Mat tmpmat1 = Mat::zeros(outH, outW, CV_32SC1);
    for(int i = 0; i < outChannels; i++)
    {
      poolL.d.push_back(tmpmat.clone());   //局域梯度
      poolL.y.push_back(tmpmat.clone());   //采样函数后神经元输出，无激活函数
      poolL.max_position.push_back(tmpmat1.clone());   //最大值模式下最大值在原矩
    }

    return poolL;
}

```

(3) 輸出層結構體初始化

```
OutLayer initOutLayer(int inputNum, int outputNum)
{
```

```
    OutLayer outL;

    outL.inputNum = inputNum;
    outL.outputNum = outputNum;
    outL.isFullConnect = true;

    outL.basicData = Mat::zeros(1, outputNum, CV_32FC1);    //偏置,分配内存的同时
    outL.d = Mat::zeros(1, outputNum, CV_32FC1);
    outL.v = Mat::zeros(1, outputNum, CV_32FC1);
    outL.y = Mat::zeros(1, outputNum, CV_32FC1);

    // 权重的初始化
    outL.wData = Mat::zeros(outputNum, inputNum, CV_32FC1);   // 输出行，输入列,
    srand((unsigned)time(NULL));
    for(int i = 0; i < outputNum; i++)
    {
      float *p = outL.wData.ptr<float>(i);
      for(int j = 0; j < inputNum; j++)
      {
        //使用随机数初始化权重
        float randnum = (((float)rand()/(float)RAND_MAX)-0.5)*2; // 产生一个-1到
        p[j] = randnum*sqrt(6.0/(inputNum+outputNum));
      }
    }

    return outL;
}
```

(4) 5層網絡結構體初始化

```
void cnnsetup(CNN &cnn, int inputSize_r, int inputSize_c, int outputSize)   /
{
  cnn.layerNum = 5;

  //C1层
  int mapSize = 5;
  int inSize_c = inputSize_c;   //28
  int inSize_r = inputSize_r;   //28
  int C1_outChannels = 6;
  cnn.C1 = initCovLayer(inSize_c, inSize_r, mapSize, 1, C1_outChannels);   //

  //S2层
  inSize_c = inSize_c - cnn.C1.mapSize + 1;  //24
  inSize_r = inSize_r - cnn.C1.mapSize + 1;  //24
  mapSize = 2;
  cnn.S2 = initPoolLayer(inSize_c, inSize_r, mapSize, cnn.C1.outChannels, cnn

  //C3层
  inSize_c = inSize_c / cnn.S2.mapSize;   //12
  inSize_r = inSize_r / cnn.S2.mapSize;   //12
  mapSize = 5;
  int C3_outChannes = 12;
  cnn.C3 = initCovLayer(inSize_c, inSize_r, mapSize, cnn.S2.outChannels, C3_c

  //S4层
  inSize_c = inSize_c - cnn.C3.mapSize + 1;   //8
  inSize_r = inSize_r - cnn.C3.mapSize + 1;   //8
  mapSize = 2;
```

```
  cnn.S4 = initPoolLayer(inSize_c, inSize_r, mapSize, cnn.C3.outChannels, cnn
  //05层
  inSize_c = inSize_c / cnn.S4.mapSize;   //4
  inSize_r = inSize_r / cnn.S4.mapSize;   //4
  cnn.O5 = initOutLayer(inSize_c*inSize_r*cnn.S4.outChannels, outputSize);

  cnn.e = Mat::zeros(1, cnn.O5.outputNum, CV_32FC1);   //输出层的输出值与标签值
}
```

3. 二維影像的捲積實現

呼叫Opencv的filter2D函數，可以很方便、很快速地實現二維卷積運算。我們先實作full模式，valid和same模式地捲積結果可以直接從full模式的結果中截取。

要注意的是，在卷積神經網路中，我們說的捲積運算其實是互相關運算，也即開始卷積運算之前卷積核不需要做順時針180°的旋轉。

```
Mat correlation(Mat map, Mat inputData, int type)
{
  const int map_row = map.rows;
  const int map_col = map.cols;
  const int map_row_2 = map.rows/2;
  const int map_col_2 = map.cols/2;
  const int in_row = inputData.rows;
  const int in_col = inputData.cols;

  //先按full模式扩充图像边缘
  Mat exInputData;
```

```
12    copyMakeBorder(inputData, exInputData, map_row_2, map_row_2, map_col_2, map
13    Mat OutputData;
14    filter2D(exInputData, OutputData, exInputData.depth(), map);
15
16
17    if(type == full)  //full模式
18    {
19      return OutputData;
20    }
21    else if(type == valid)  //valid模式
22    {
23      int out_row = in_row - (map_row - 1);
24      int out_col = in_col - (map_col - 1);
25      Mat outtmp;
26      OutputData(Rect(2*map_col_2, 2*map_row_2, out_col, out_row)).copyTo(outtm
27      return outtmp;
28    }
29    else   //same模式
30    {
31      Mat outtmp;
32      OutputData(Rect(map_col_2, map_row_2, in_col, in_row)).copyTo(outtmp);
33      return outtmp;
34    }
35
36  }
```

4 . 池化層的實現

(1) 均值池化

```
void avgPooling(Mat input, Mat &output, int mapSize)
{
  const int outputW = input.cols/mapSize;   //输出宽=输入宽/核宽
  const int outputH = input.rows/mapSize;   //输出高=输入高/核高
  float len = (float)(mapSize*mapSize);
  int i,j,m,n;
  for(i = 0;i < outputH; i++)
  {
    for(j = 0; j < outputW; j++)
    {
      float sum=0.0;
      for(m = i*mapSize; m < i*mapSize+mapSize; m++)  //取卷积核大小的窗口求和平
      {
        for(n = j*mapSize; n < j*mapSize+mapSize; n++)
        {
          sum += input.ptr<float>(m)[n];
        }
      }

      output.ptr<float>(i)[j] = sum/len;
    }
  }
}
```

(2) 最大值池化

```
void maxPooling(Mat input, Mat &max_position, Mat &output, int mapSize)
{
  int outputW = input.cols / mapSize;   //输出宽=输入宽/核宽
  int outputH = input.rows / mapSize;   //输出高=输入高/核高

  int i, j, m, n;
  for (i = 0; i < outputH; i++)
  {
    for (j = 0; j < outputW; j++)
    {
      float max = -999999.0;
      int max_index = 0;

      for (m = i*mapSize; m<i*mapSize + mapSize; m++)  //取卷积核大小的窗口的最
      {
        for (n = j*mapSize; n<j*mapSize + mapSize; n++)
        {
          if (max < input.ptr<float>(m)[n])  //求池化窗口中的最大值，并记录最大值
          {
            max = input.ptr<float>(m)[n];
            max_index = m*input.cols + n;
          }
        }
      }

      output.ptr<float>(i)[j] = max;    //求得最大值作为池化输出
      max_position.ptr<int>(i)[j] = max_index;   //记录最大值在原矩阵中的位置，
    }
  }
```

```
30  }
```

5 . 激活函數與向量點乘函數的實現

(1) Relu函數

```
1  float activation_Sigma(float input, float bas)
2  {
3    float temp = input + bas;
4    return (temp > 0 ? temp: 0);
5  }
```

(2) Softmax函數

```
1  void softmax(OutLayer &O)
2  {
3    float sum = 0.0;
4    float *p_y = O.y.ptr<float>(0);
5    float *p_v = O.v.ptr<float>(0);
6    float *p_b = O.basicData.ptr<float>(0);
7    for (int i = 0; i < O.outputNum; i++)
8    {
9      float Yi = exp(p_v[i]+ p_b[i]);
10     sum += Yi;
11     p_y[i] = Yi;
12   }
13
```

```
  for (int i = 0; i < O.outputNum; i++)
  {
    p_y[i] = p_y[i]/sum;
  }
}
```

(3) 兩個一維向量的點乘函數

以下函數中，vec1和vec2是兩個長度相同的一維向量，點乘的結果就是它們對應位置的值相乘，然後把所有乘積相加的結果。

```
float vecMulti(Mat vec1, float *vec2)// 两向量相乘
{
  float *p1 = vec1.ptr<float>(0);
  float m = 0;
  for (int i = 0; i < vec1.cols; i++)
    m = m + p1[i] * vec2[i];
  return m;
}
```

6．5層網路前向傳播的實現

(1) 卷積層前向傳播

```
//输入的inputData有可能是一张图像，也有可能是多张图像，如果是多张图像，则把它们的卷积
void cov_layer_ff(vector<Mat> inputData, int cov_type, CovLayer &C)
{
  for (int i = 0; i < (C.outChannels); i++)
  {
```

```
    for (int j = 0; j < (C.inChannels); j++)
    {
      //计算卷积，mapData为四维矩阵
      Mat mapout = correlation(C.mapData[j][i], inputData[j], cov_type);
      C.v[i] += mapout;     //所有输入通道的卷积结果累加
    }

    int output_r = C.y[i].rows;
    int output_c = C.y[i].cols;
    for (int r = 0; r < output_r; r++)
    {
      for (int c = 0; c < output_c; c++)
      {
        C.y[i].ptr<float>(r)[c] = activation_Sigma(C.v[i].ptr<float>(r)[c], C
      }
    }
  }
}

```

(2) 池化層前向傳播

```
#define AvePool 0
#define MaxPool 1

void pool_layer_ff(vector<Mat> inputData, int pool_type, PoolLayer &S)
{
  if (pool_type == AvePool)  //均值池化
```

```
  {
    for (int i = 0; i < S.outChannels; i++)
    {
      avgPooling(inputData[i], S.y[i], S.mapSize);
    }
  }
  else if(pool_type == MaxPool)   //最大值池化
  {
    for (int i = 0; i < S.outChannels; i++)
    {
      maxPooling(inputData[i], S.max_position[i], S.y[i], S.mapSize);
    }
  }
  else
  {
    printf("pool type erroe!\n");
  }
}
```

(3) 輸出層前向傳播

```
void nnff(Mat input, Mat wdata, Mat &output)
{
  for (int i = 0; i < output.cols; i++)  //分别计算多个向量相乘的乘积
    output.ptr<float>(0)[i] = vecMulti(input, wdata.ptr<float>(i));   //由于
}


void out_layer_ff(vector<Mat> inputData, OutLayer &O)
```

```
{
  Mat OinData(1, O.inputNum, CV_32FC1);   //输入192通道
  float *OinData_p = OinData.ptr<float>(0);
  int outsize_r = inputData[0].rows;
  int outsize_c = inputData[0].cols;
  int last_output_len = inputData.size();
  for (int i = 0; i < last_output_len; i++)   //上一层S4输出12通道的4*4矩阵
  {
    for (int r = 0; r < outsize_r; r++)
    {
      for (int c = 0; c < outsize_c; c++)
      {
        //将12通道4*4矩阵展开成长度为192的一维向量
        OinData_p[i*outsize_r*outsize_c + r*outsize_c + c] = inputData[i].ptr
      }
    }
  }

  //192*10个权重
  nnff(OinData, O.wData, O.v);   //10通道输出,1个通道的输出等于192个输入分别与192

  //Affine层的输出经过Softmax函数，转换成0~1的输出结果
  softmax(O);
}
```

(4) 5層網絡前向傳播

```
void cnnff(CNN &cnn, Mat inputData)
```

```
{
  //C1
  //5*5卷积核
  //输入28*28矩阵
  //输出(28-25+1)*(28-25+1) = 24*24矩阵
  vector<Mat> input_tmp;
  input_tmp.push_back(inputData);
  cov_layer_ff(input_tmp, valid, cnn.C1);

  //S2
  //24*24-->12*12
  pool_layer_ff(cnn.C1.y, MaxPool, cnn.S2);

  //C3
  //12*12-->8*8
  cov_layer_ff(cnn.S2.y, valid, cnn.C3);

  //S4
  //8*8-->4*4
  pool_layer_ff(cnn.C3.y, MaxPool, cnn.S4);

  //O5
  //12*4*4-->192-->1*10
  out_layer_ff(cnn.S4.y, cnn.O5);
}
```

好了，本文就講到這裡，接下來的文章我們來講反向傳播的實現和參數更新的實現，敬請期待~

歡迎掃碼追蹤以下微信公眾號，接下來會不定時更新更加精彩的內容噢～

C++ 70　Opencv 50　人工智慧 27　深度學習 26　機器學習 33

C++ · 目錄

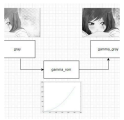
上一篇
基於LK光流金字塔演算法與TPS變換的連續時間序列影像配準

下一篇
卷積神經網路原理及其C++/Opencv實作(7)—誤反向傳播程式碼實現

閱讀原文

喜歡此內容的人還喜歡

數位影像處理之gamma矯正
FPGA開源工作室



混凝土模板荷載與壓力計算

忒修斯破船

NJ系列電子凸輪應用分享

Karl工控