DeepDriver解密之三

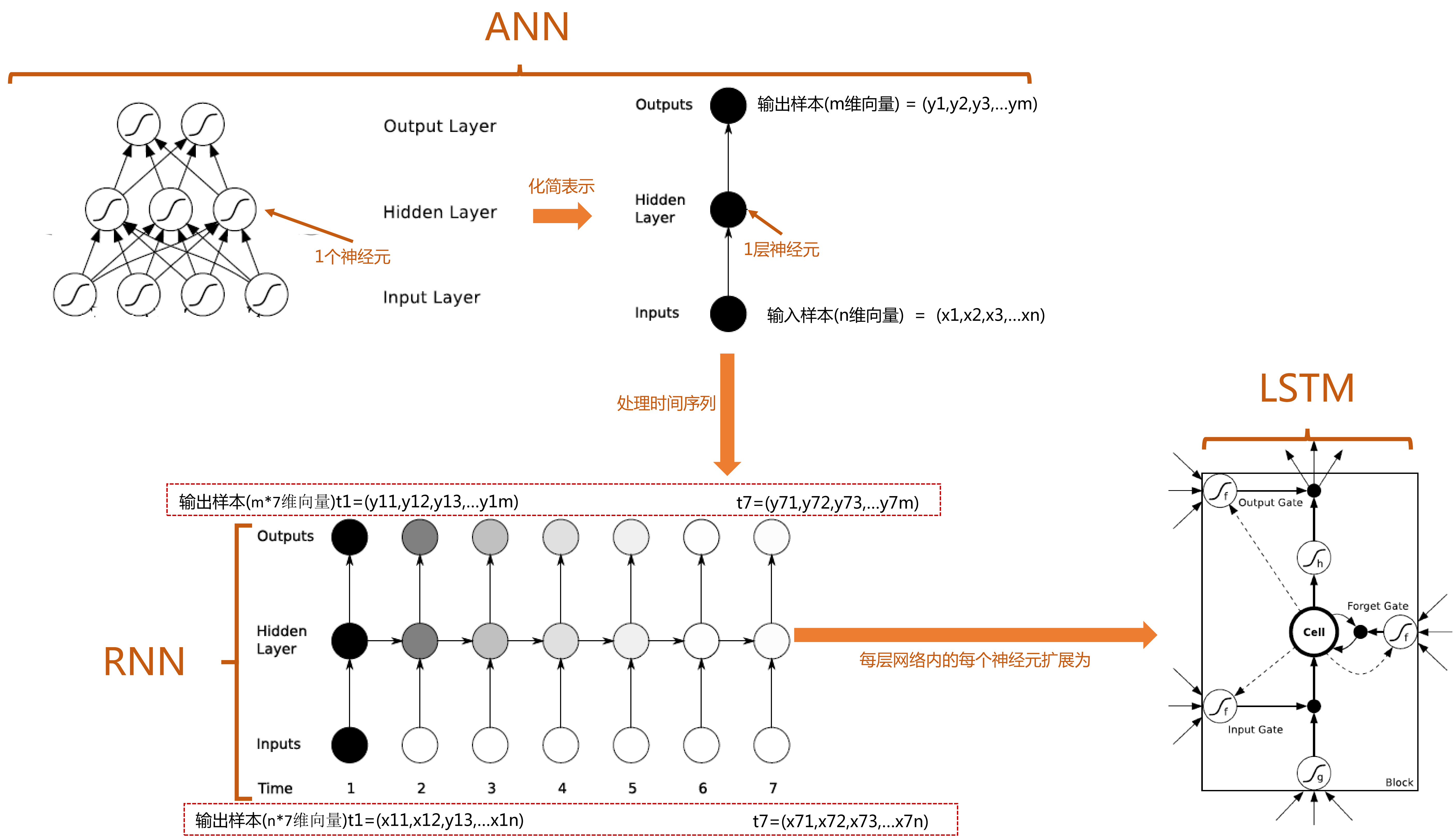
本文作者： 李明 蔡龙军

DeepDriver创建者：蔡龙军

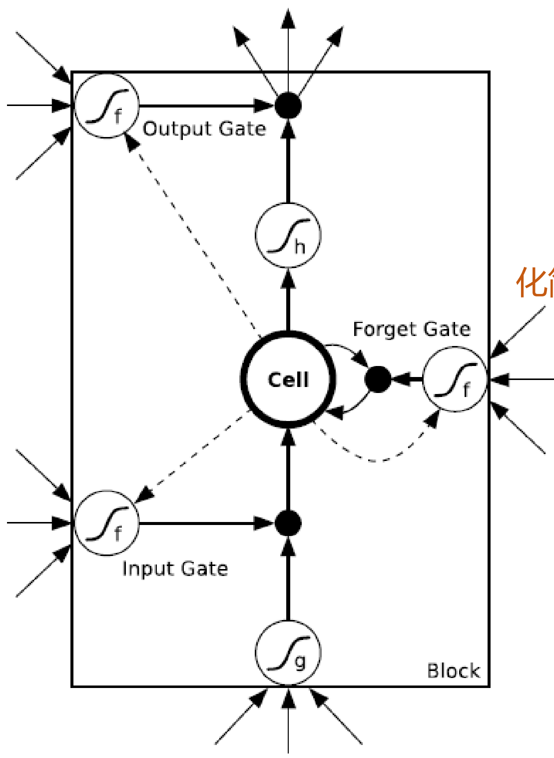
Source codes: https://github.com/LongJunCai/DeepDriver

LSTM 原理

ANN->RNN->LSTM

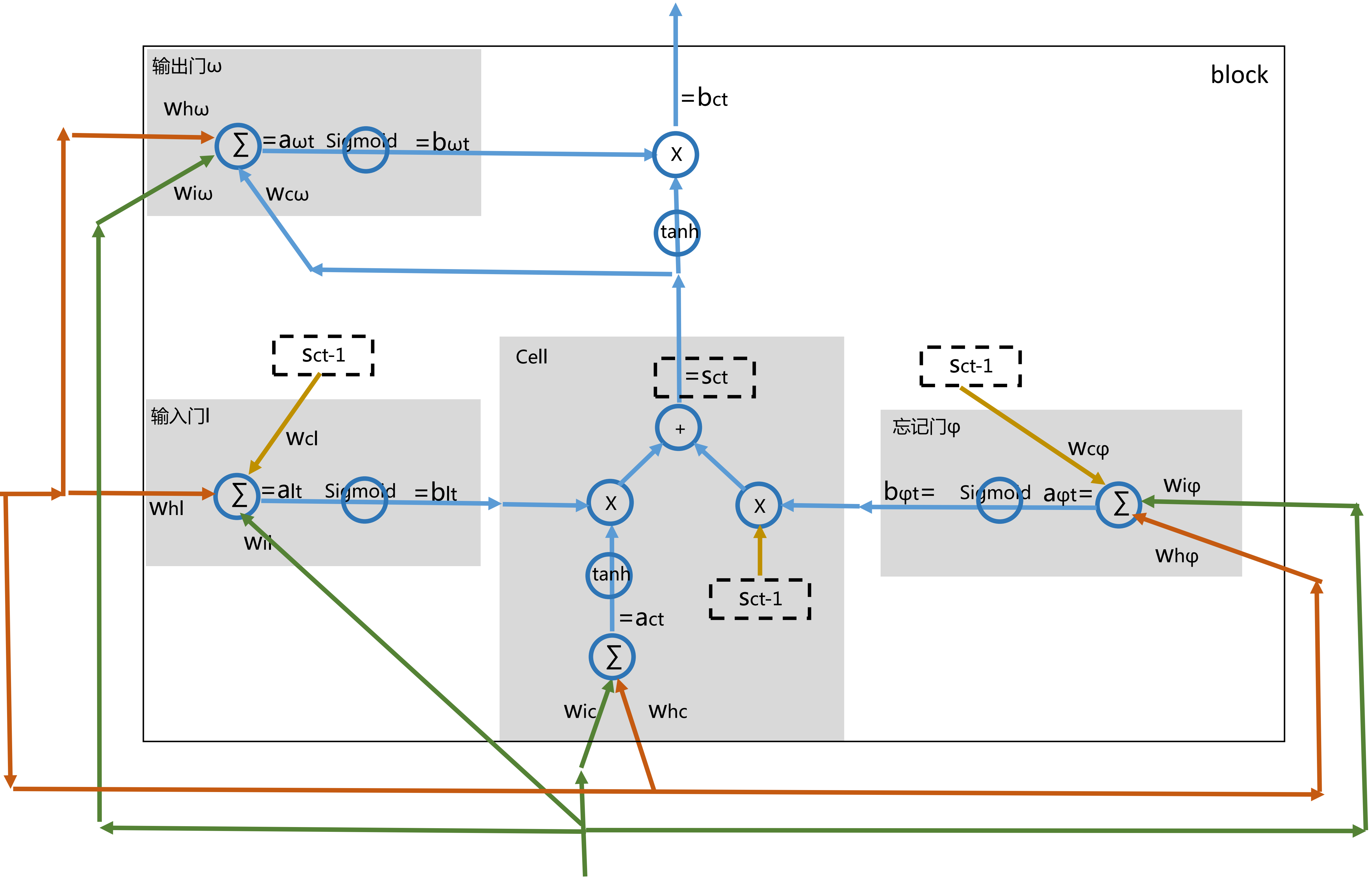

ANN
Output Layer
Hidden Layer
Input Layer
1个神经元
化简表示
Outputs
输出样本(m维向量) = (y1,y2,y3,...ym)
Hidden Layer
1层神经元
Inputs
输入样本(n维向量) = (x1,x2,x3,...xn)
处理时间序列
LSTM
Output Gate
Forget Gate
Cell
Input Gate
Block
输出样本(m*7维向量)t1=(y11,y12,y13,...y1m)
t7=(y71,y72,y73,...y7m)
RNN
Outputs
Hidden Layer
Inputs
Time
1
2
3
4
5
6
7
每层网络内的每个神经元扩展为
输出样本(n*7维向量)t1=(x11,x12,y13,...x1n)
t7=(x71,x72,x73,...x7n)

LSTM内Block的结构



化简取每个Block内
仅有1个cell

输入门和忘记门+cell的上一个状态决定cell的本次状态
输出门决定cell的本次状态中输出什么



上一个时序(t-1)中本层网络的第h个block的输出: bht-1

时间序列中第t个状态的输入的第i个分量特征: xit

LSTM内Block的结构(正向传播公式)

Output Gates

$$a_\omega^t = \sum_{i=1}^{I} w_{i\omega} x_i^t + \sum_{h=1}^{H} w_{h\omega} b_h^{t-1} + \sum_{c=1}^{C} w_{c\omega} s_c^t \tag{4.8}$$

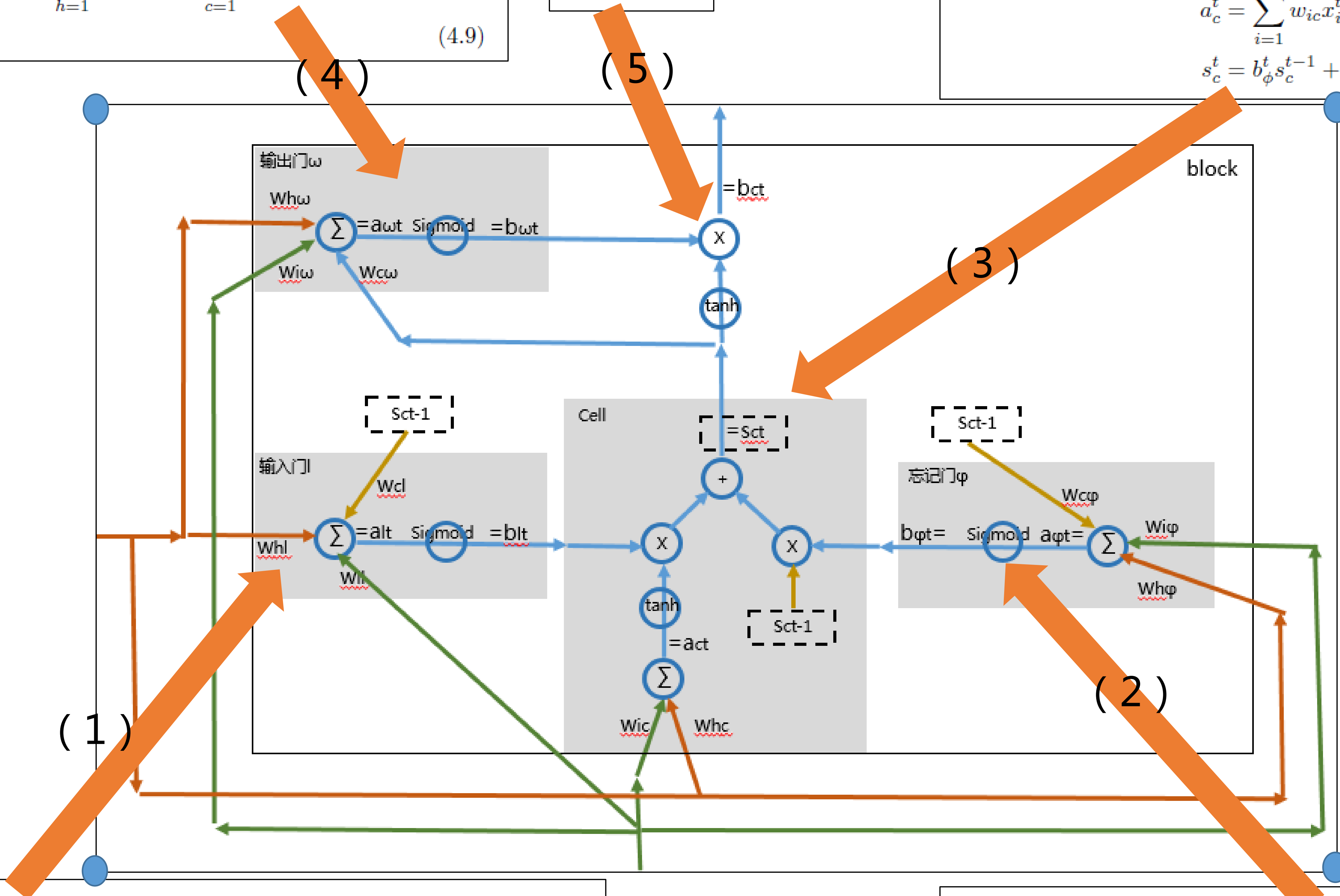
$$b_\omega^t = f(a_\omega^t) \tag{4.9}$$

$$b_c^t = b_\omega^t h(s_c^t)$$

Cells

$$a_c^t = \sum_{i=1}^{I} w_{ic} x_i^t + \sum_{h=1}^{H} w_{hc} b_h^{t-1} \tag{4.6}$$

$$s_c^t = b_\phi^t s_c^{t-1} + b_\iota^t g(a_c^t) \tag{4.7}$$



Input Gates

$$a_\iota^t = \sum_{i=1}^{I} w_{i\iota} x_i^t + \sum_{h=1}^{H} w_{h\iota} b_h^{t-1} + \sum_{c=1}^{C} w_{c\iota} s_c^{t-1} \tag{4.2}$$

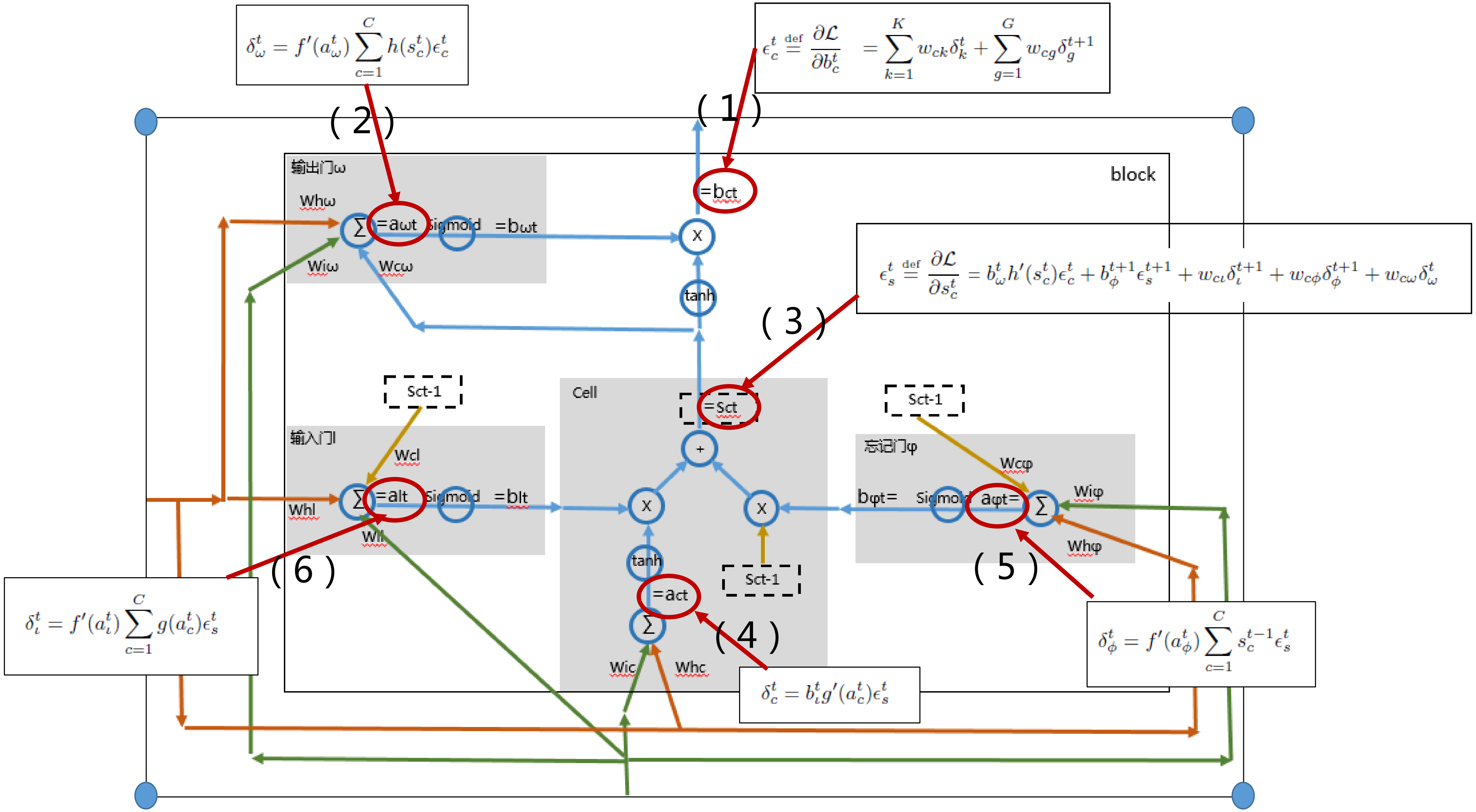
$$b_\iota^t = f(a_\iota^t) \tag{4.3}$$

Forget Gates

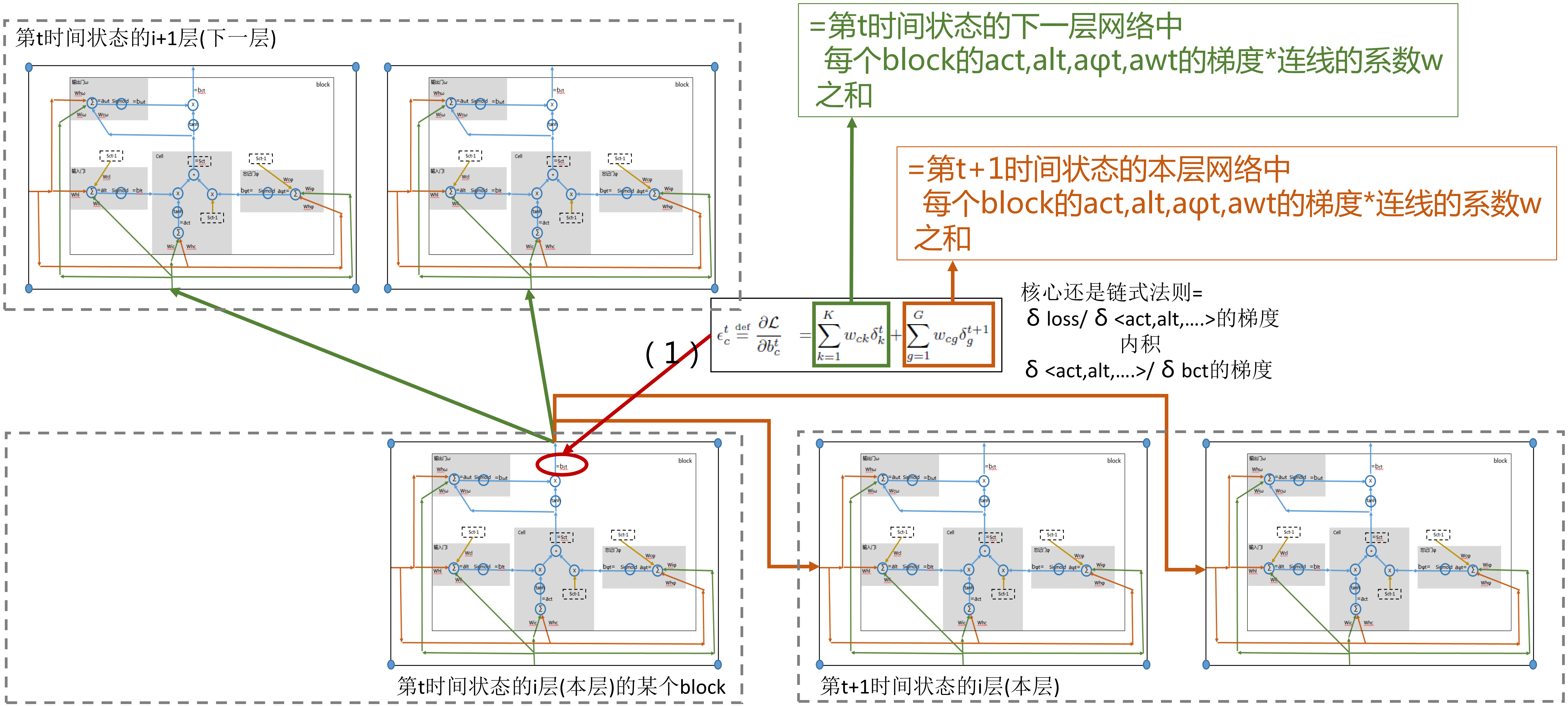
$$a_\phi^t = \sum_{i=1}^{I} w_{i\phi} x_i^t + \sum_{h=1}^{H} w_{h\phi} b_h^{t-1} + \sum_{c=1}^{C} w_{c\phi} s_c^{t-1} \tag{4.4}$$

$$b_\phi^t = f(a_\phi^t) \tag{4.5}$$

LSTM内Block的结构(反向传播公式)



LSTM内Block的结构(反向传播公式-推导-bct的梯度)



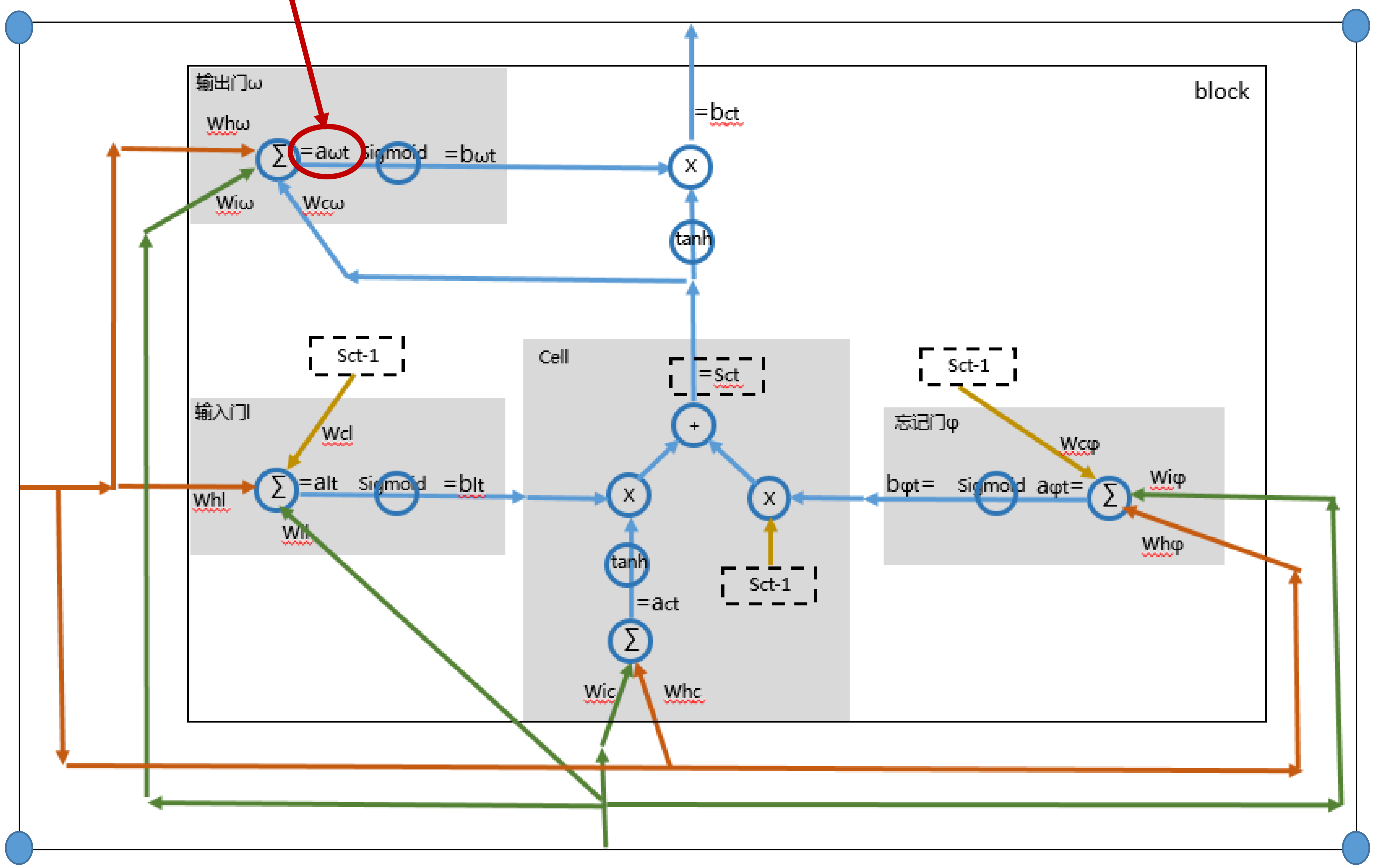
LSTM内Block的结构(反向传播公式-推导-awt的梯度)

取C=1

awt的梯度=δLoss/δawt= δLoss/δbct* δbct/ δawt

δbct/ δawt= δ(tanh(sct)*sigmod(awt))/δawt=tanh(sct)*dsigmod(awt)

$$\delta_\omega^t = f'(a_\omega^t) \sum_{c=1}^{C} h(s_c^t) \epsilon_c^t$$



LSTM内Block的结构(反向传播公式-推导-sct的梯度)

取C=1

Sct会影响　第t时间状态的awt，　第t时间状态的bct，　第t+1时间状态的alt，　第t+1时间状态的aφt，　第t+1时间状态的sct

=> **δL/δawt*δawt/δsct** + **δL/δbct*δbct/δsct** + **δL/δalt+1*δalt+1/δsct** + **δL/δaφt+1*δaφt+1/δsct** + **δL/δsct+1*δsct+1/δsct**

1）**δL/δawt*δawt/δsct** = δL/δawt* δ(wcw*sct+....)/δsct= δL/δawt* wcw

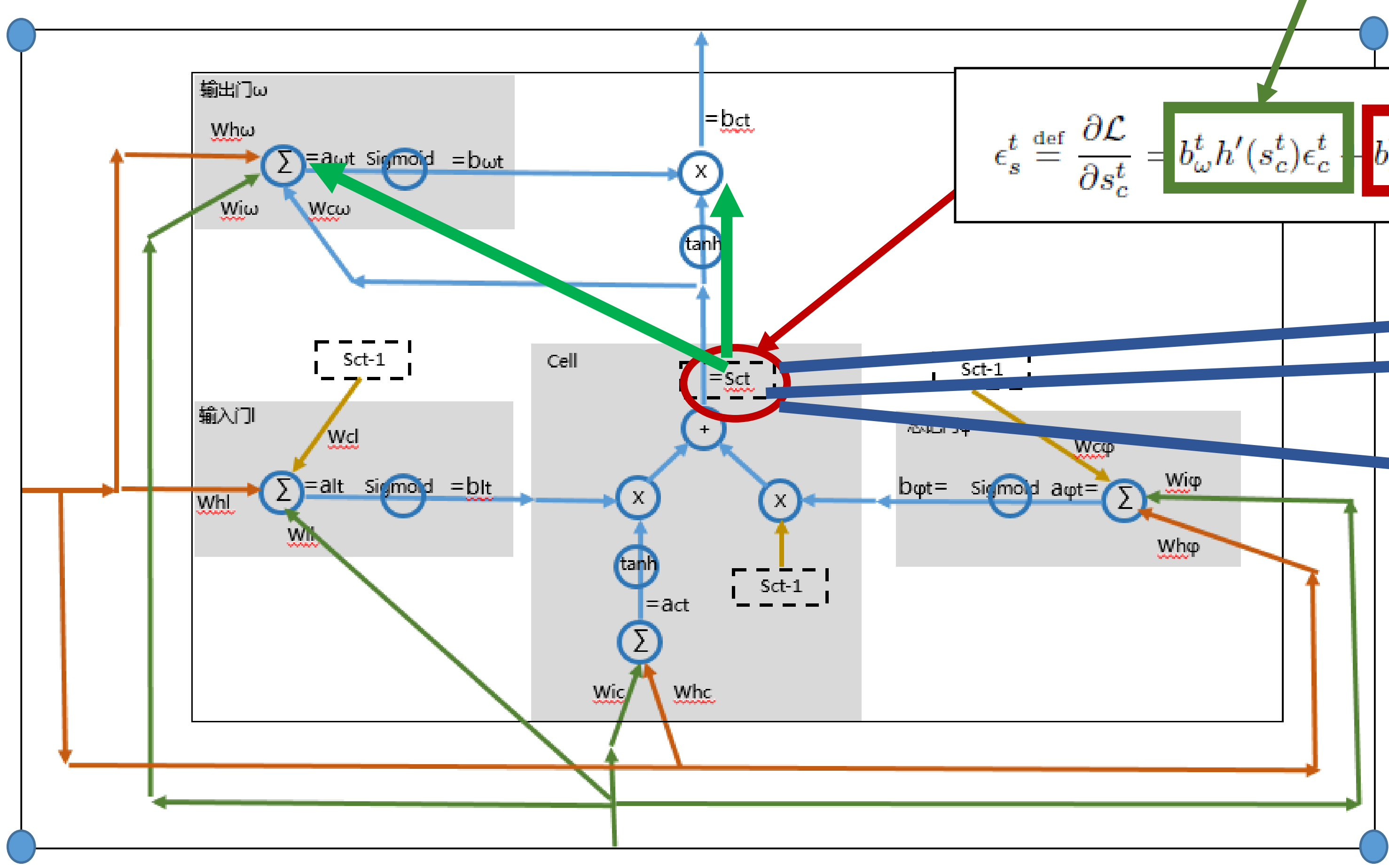
2）**δL/δbct*δbct/δsct** = δL/δbct*δ(bwt*tanh(sct))/δsct= δL/δbct*bwt*dtanh(sct)

3）**δL/δalt+1*δalt+1/δsct** = δL/δalt+1* δ(sct*wcl+...)/δsct= δL/δalt+1* wcl

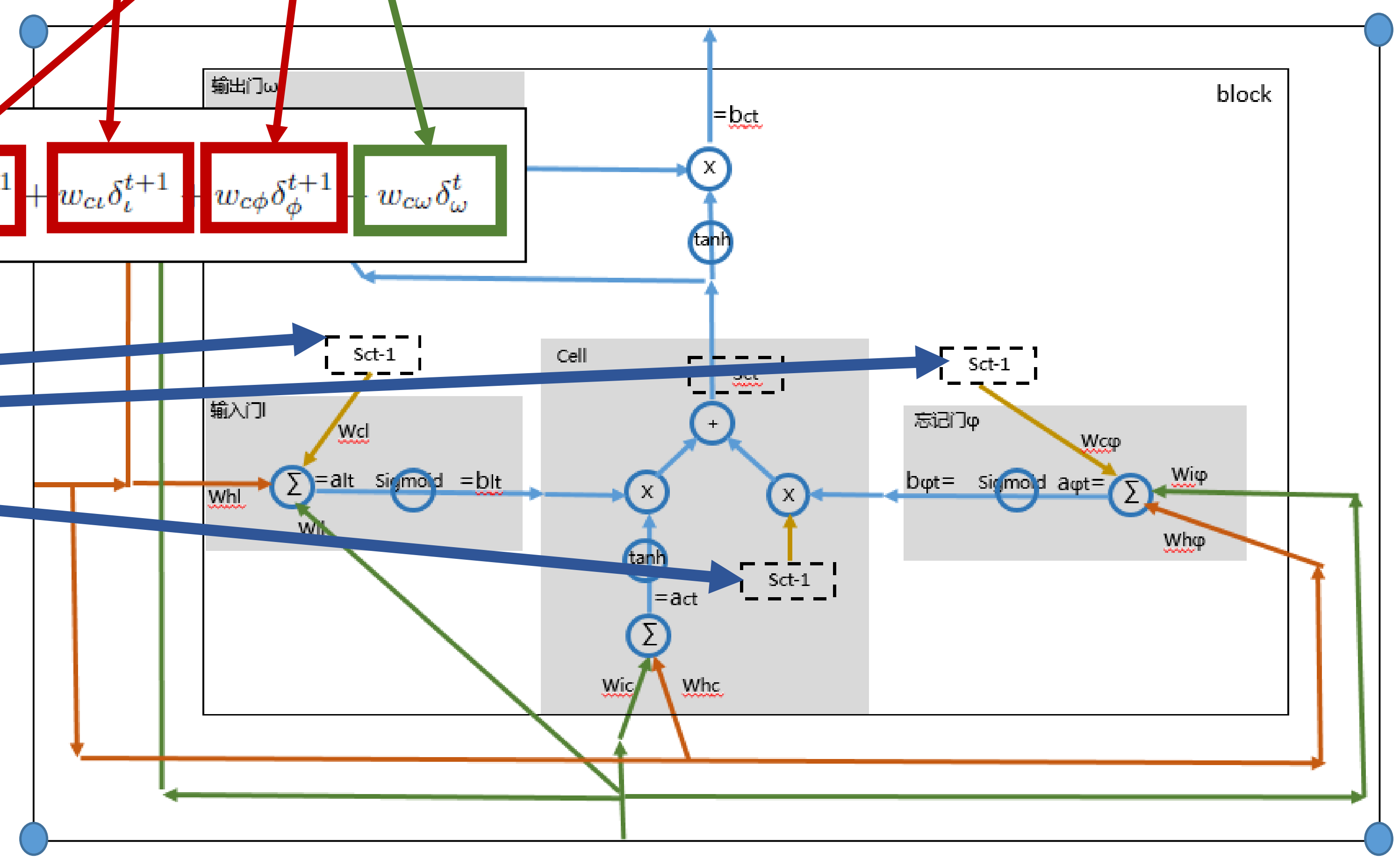
4）**δL/δaφt+1*δaφt+1/δsct** = δL/δaφt+1* δ(sct*wcφ+....)/δsct= δL/δaφt+1* wcφ

5）**δL/δsct+1*δsct+1/δsct** = δL/δsct+1* δ(sct*bφt+1 +....)/δsct= δL/δsct+1 * bφt+1

$$\epsilon_s^t \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial s_c^t} = b_\omega^t h'(s_c^t)\epsilon_c^t + b_\phi^{t+1}\epsilon_s^{t+1} + w_{c\iota}\delta_\iota^{t+1} + w_{c\phi}\delta_\phi^{t+1} + w_{c\omega}\delta_\omega^t$$



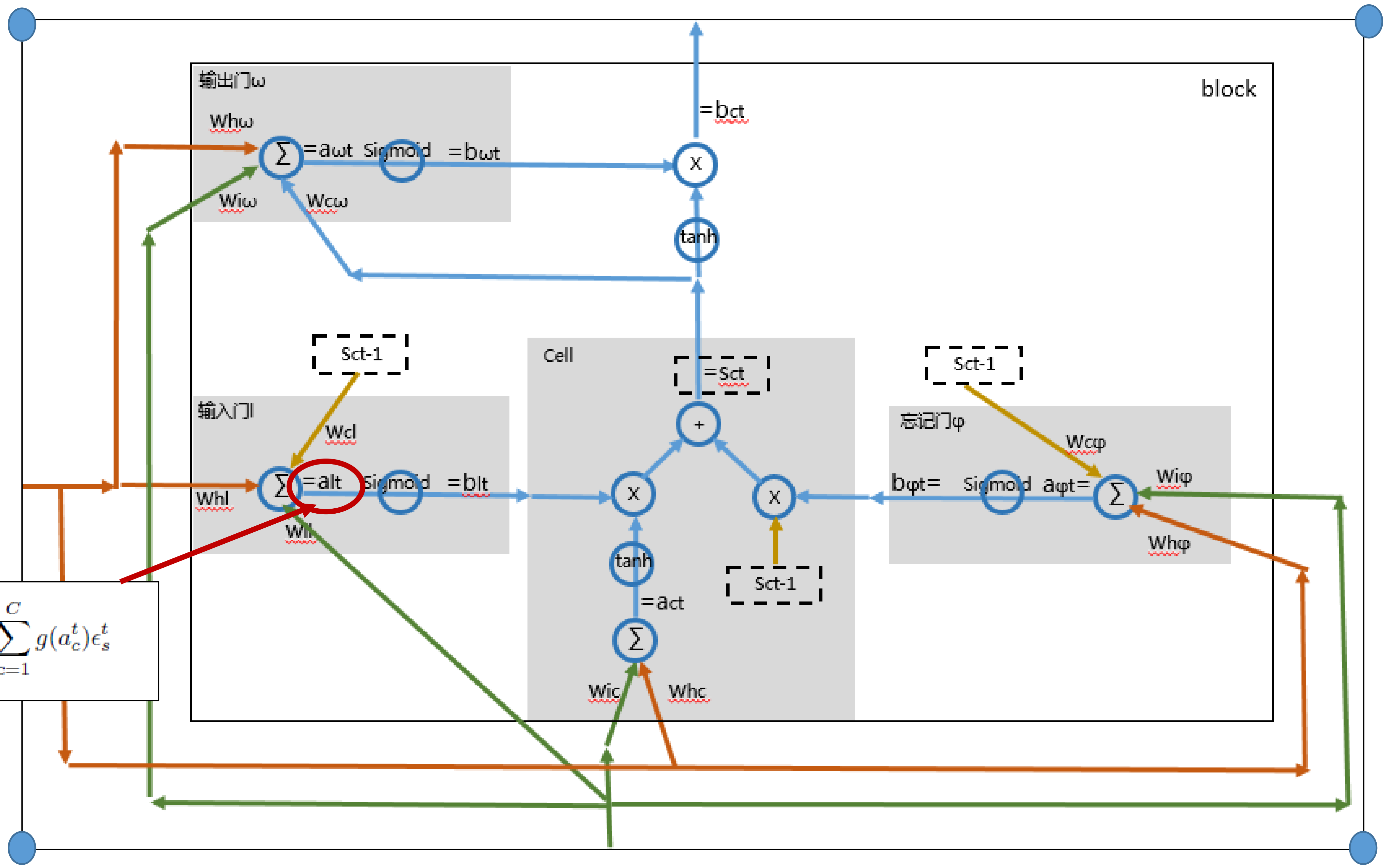
第t时间状态的本层的该block



第t+1时间状态的本层的该block

LSTM内Block的结构(反向传播公式-推导-alt的梯度)

取C=1



$$\delta_\iota^t = f'(a_\iota^t)\sum_{c=1}^{C} g(a_c^t)\epsilon_s^t$$

alt的梯度=δLoss/δalt= δLoss/δsct* δsct/ δalt

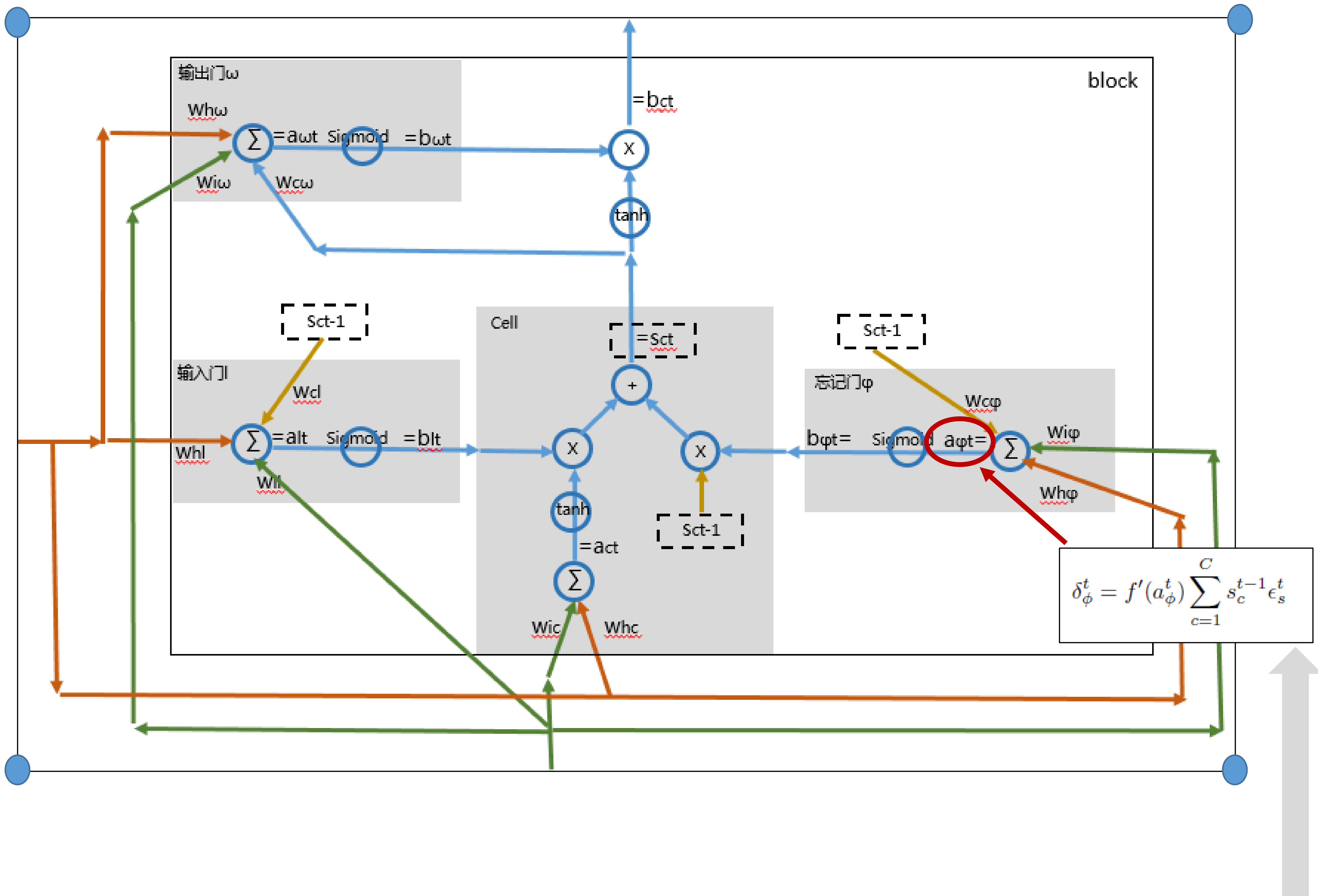
δsct/ δalt= δ(tanh(act)*sigmod(alt)+sct-1 * bφt)/δalt

=δ(tanh(act)*sigmod(alt))/δalt

=tanh(act)*dsigmod(alt)

LSTM内Block的结构(反向传播公式-推导-aφt的梯度)

取C=1



aφt的梯度=δLoss/δaφt= δLoss/δsct* δsct/ δaφt

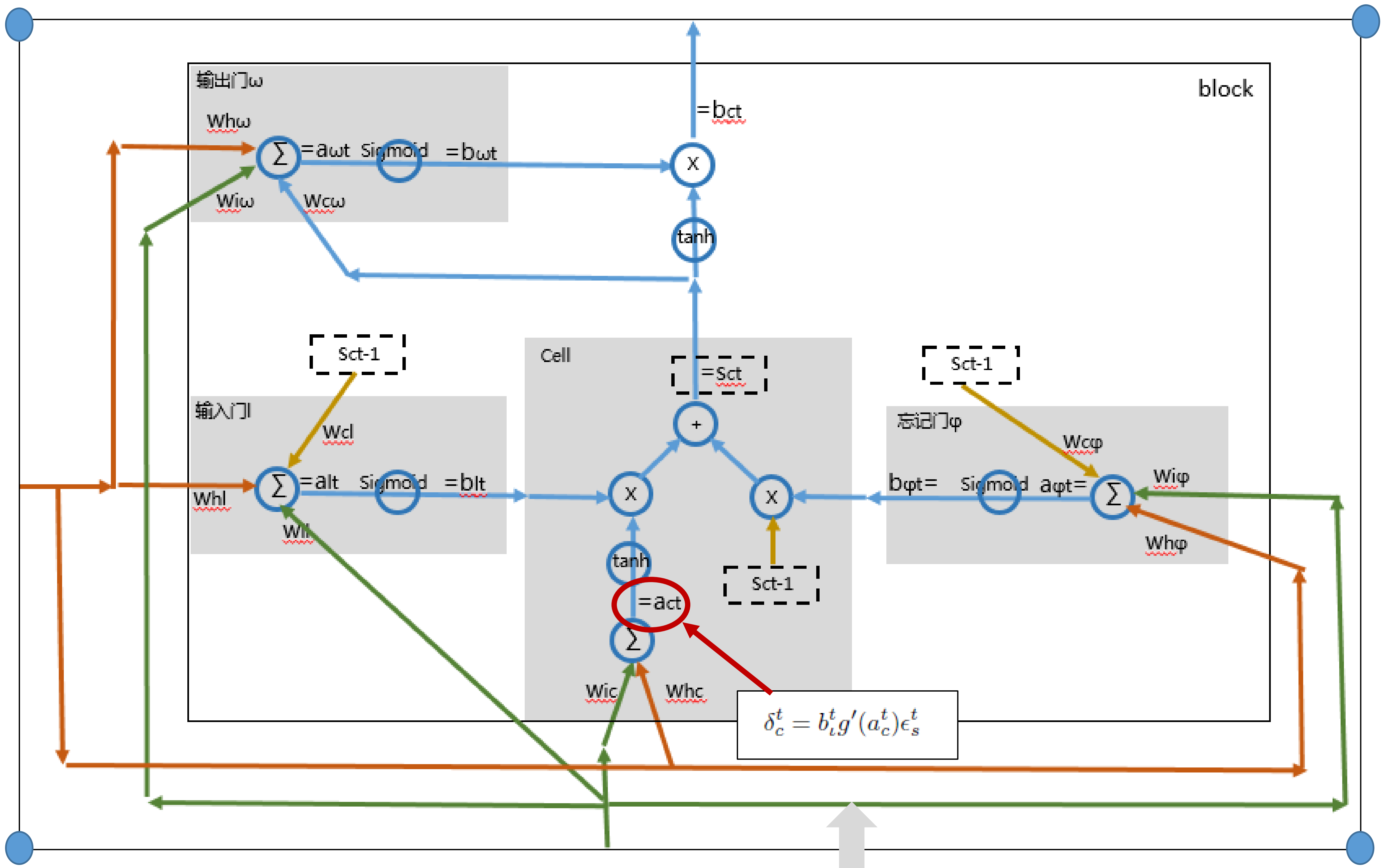
δsct/ δaφt= δ(sct-1*sigmod(aφt)+blt*tanh(act))/δaφt

= δ(sct-1*sigmod(aφt))/δaφt

=sct-1 * dsigmod(aφt)

LSTM内Block的结构(反向传播公式-推导-act的梯度)

取C=1



act的梯度=δLoss/δact= δLoss/δsct* δsct/ δact

δsct/ δact= δ(blt*tanh(act)+sct-1*bφt)/δact

=δ(blt*tanh(act))/δact

=blt*dtanh(act)

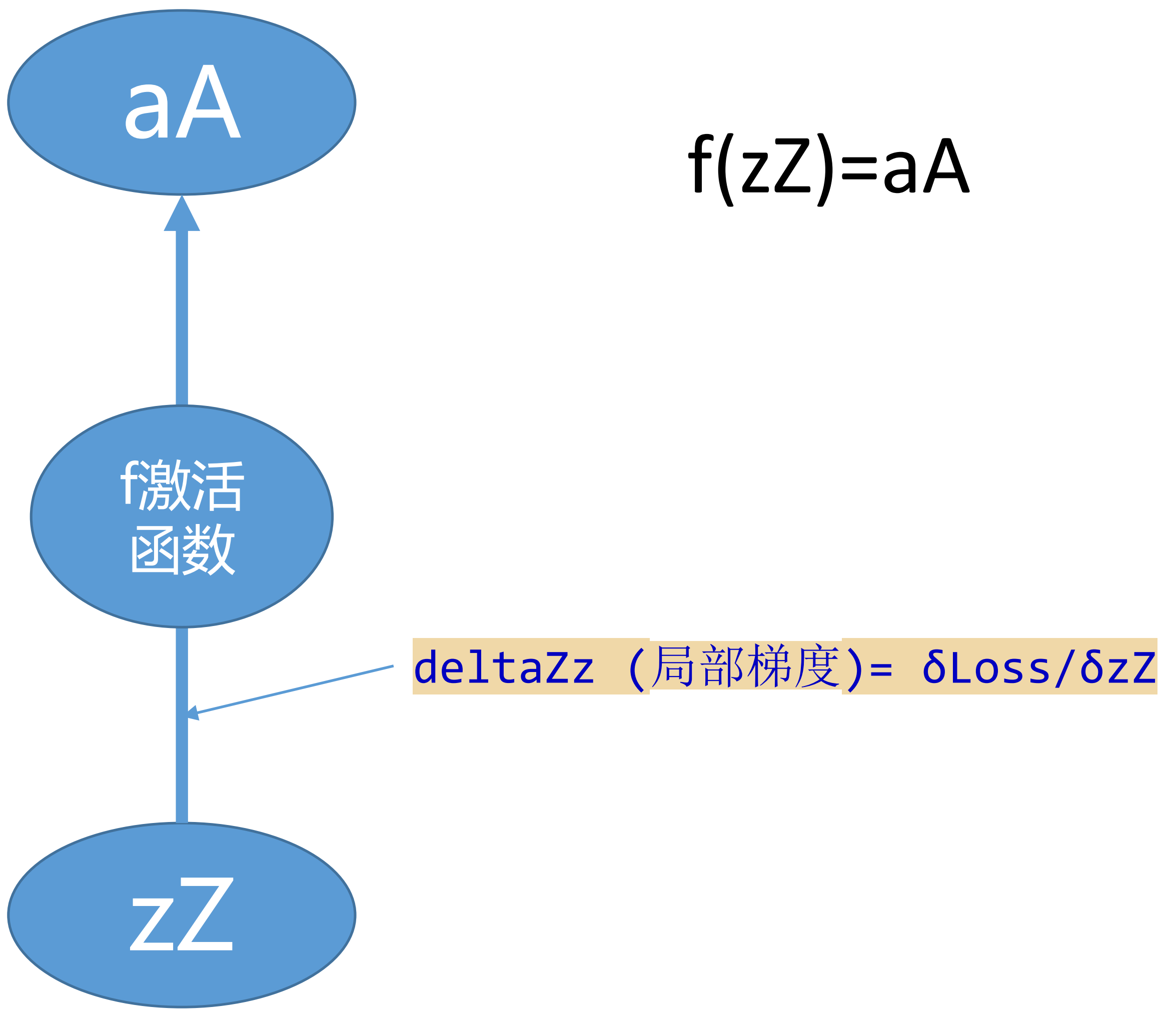
DeepDriver的BPTT代码导读

某层layer内的某个block内的某个门的输出状态类 SimpleNeuroVo

代码位置:lstm>SimpleNeuroVo.java

主要用途:存储每个时间状态的结果和局部梯度

向前传播参加 `lstm>BPTT.java`中的`fTTiRNNNeuroVo`方法



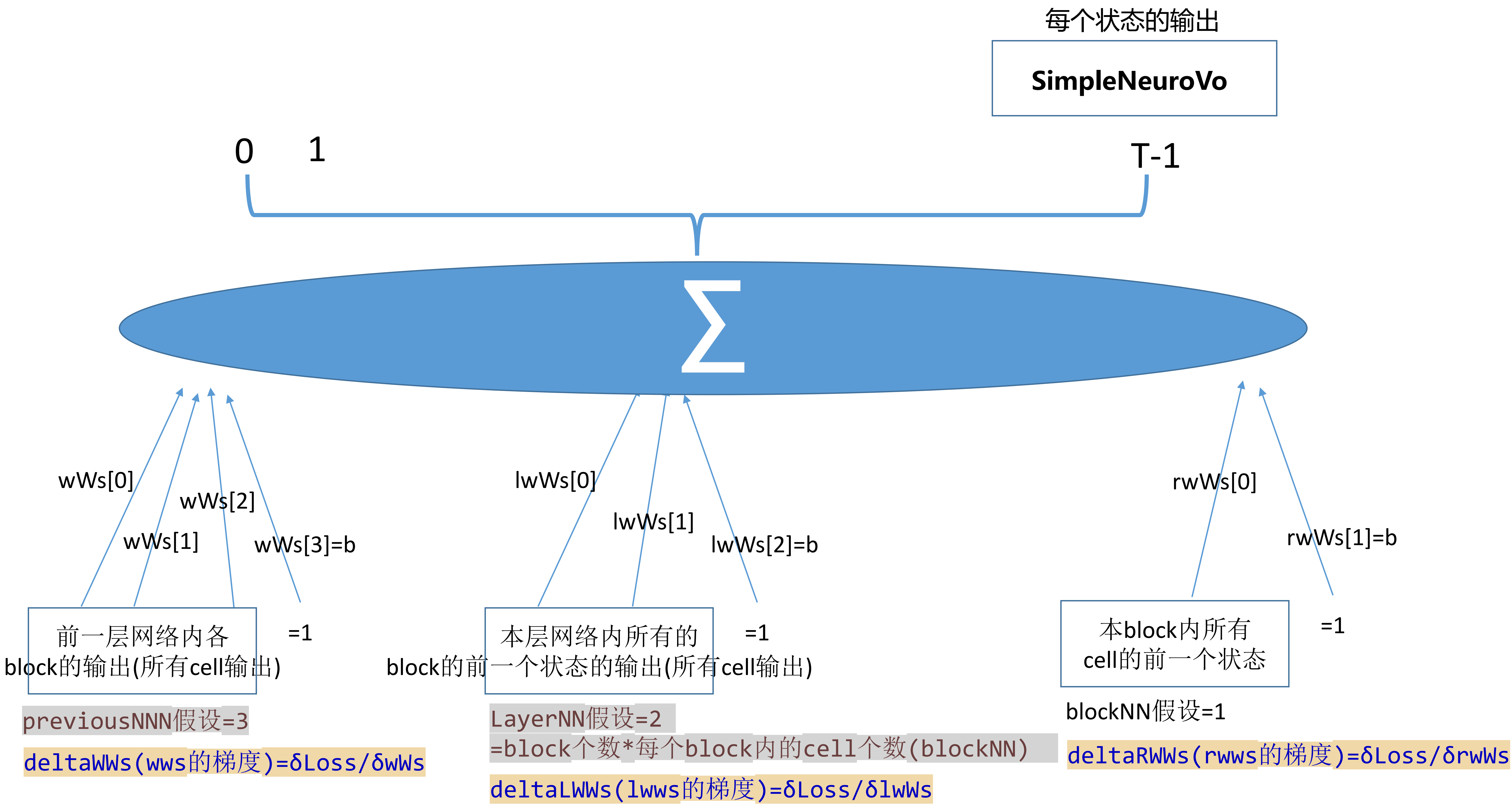
f(zZ)=aA

某层layer内的某个block内的某个门的父类RNNNeuroVo-用于输入输出忘记门

代码位置:lstm>RNNNeuroVo.java
代码作业：block内的输入输出和忘记门
一般情况下取blockNN=1 即每个block内的cell个数=1

牛逼（相对于我）：
注意如何抽象的：把每个时间状态的输出记录了，但是w是公用的

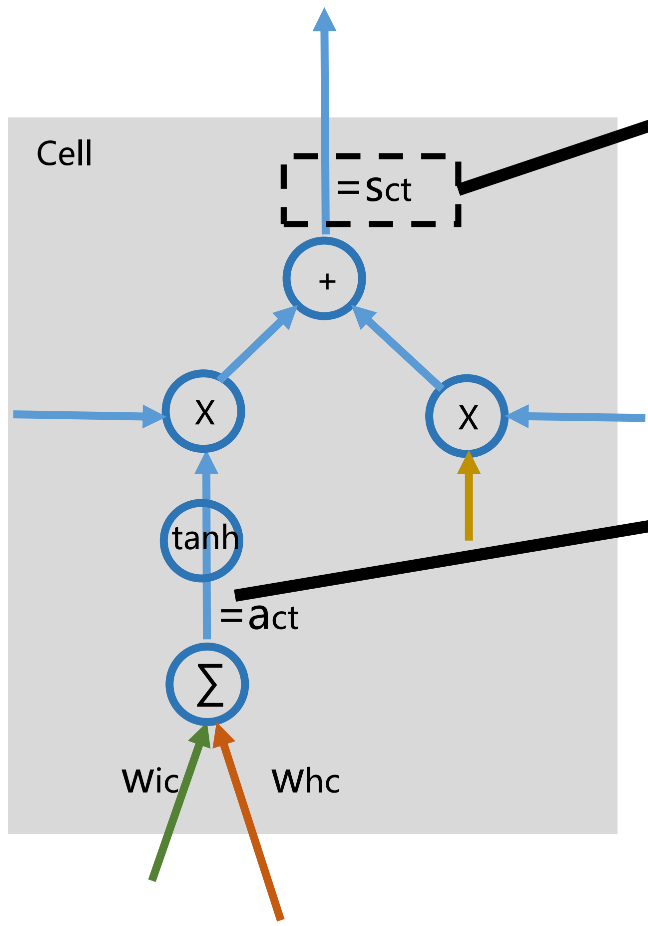
向前传播参加 lstm>BPTT.java中的fTTiRNNNeuroVo方法



某层layer内的某个block内的某个cell类 Cell extends RNNNeuroVo（1）

代码位置:lstm>Cell.java
代码作业：block内的cell状态等

向前传播参加 lstm>BPTT.java中的fTT4PartialLstmLayer方法



Cell类中存储t个状态下的所有状态 **transient double [] sc;**

每个状态的梯度 **transient double [] deltaSc;**

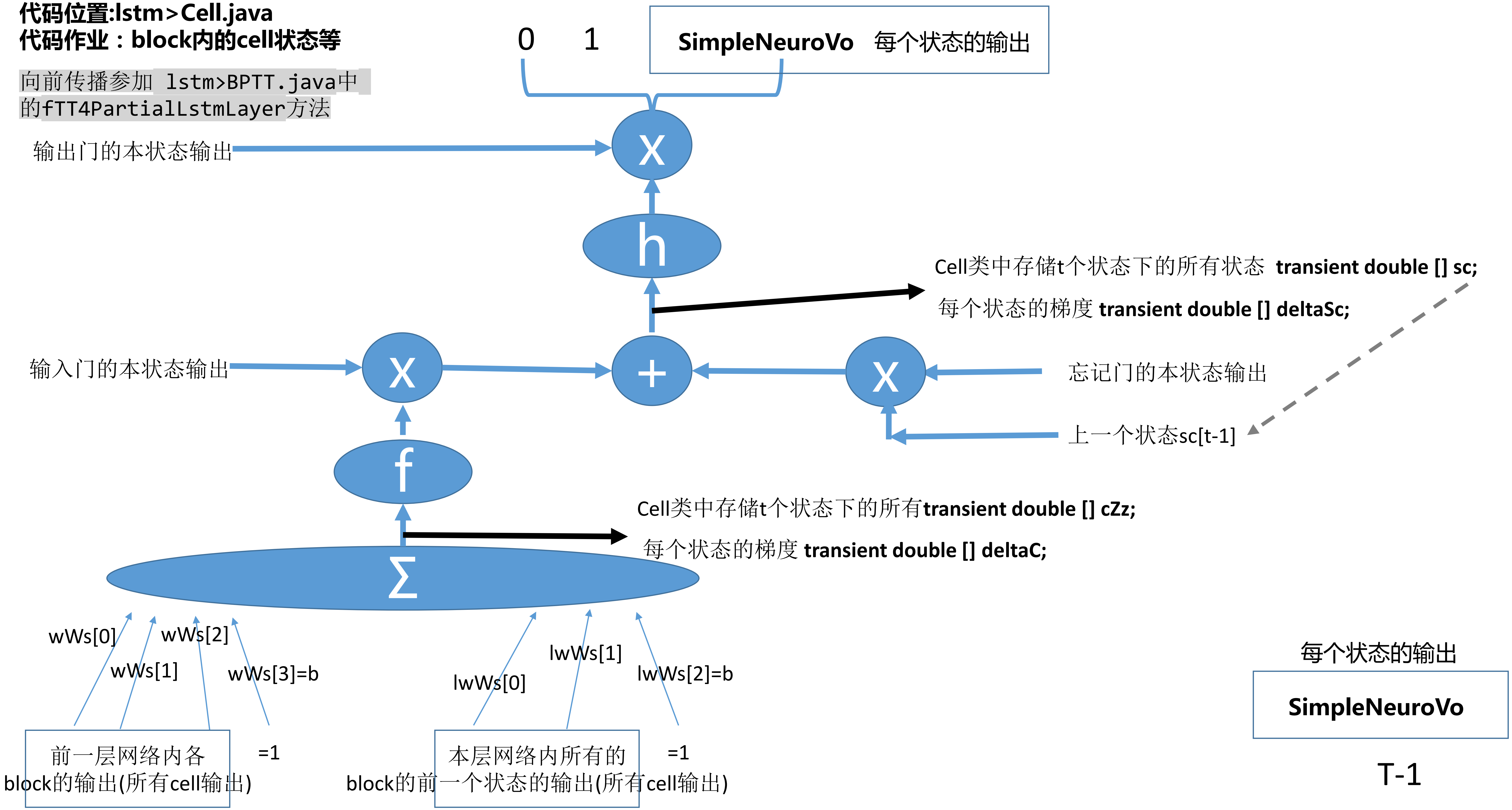
Cell类中存储t个状态下的所有**transient double [] cZz;**

每个状态的梯度 **transient double [] deltaC;**

某层layer内的某个block内的某个cell类 Cell　extends RNNNeuroVo（2）

代码位置:lstm>Cell.java
代码作业：block内的cell状态等

向前传播参加 lstm>BPTT.java中的fTT4PartialLstmLayer方法



previousNNN假设=3

deltaWWs(wws的梯度)=δLoss/δwWs

LayerNN假设=2
=block个数*每个block内的cell个数(blockNN)

deltaLWWs(lwws的梯度)=δLoss/δlwWs

某层layer内的某个block类 Block

代码位置:lstm>Block.java
代码作业：每个层的每个block

向前传播参加 `lstm>BPTT.java`中的`fTT4PartialLstmLayer`方法

BP过程

BPTT.java的runEpich

```
//使用一个样本做一次训练
public double runEpich(double [][] sample, double [][] targets) {
  tLength = sample.length;
  //向前传播
  fTT(sample, false);
  //向后传播
  bptt(targets);
  if (!cfg.isMeasureOnly()) {
    updateWws();//更新权重
  }
  return error;
}
```

参考向前传播部分

参考向后传播部分

参考参数更新部分

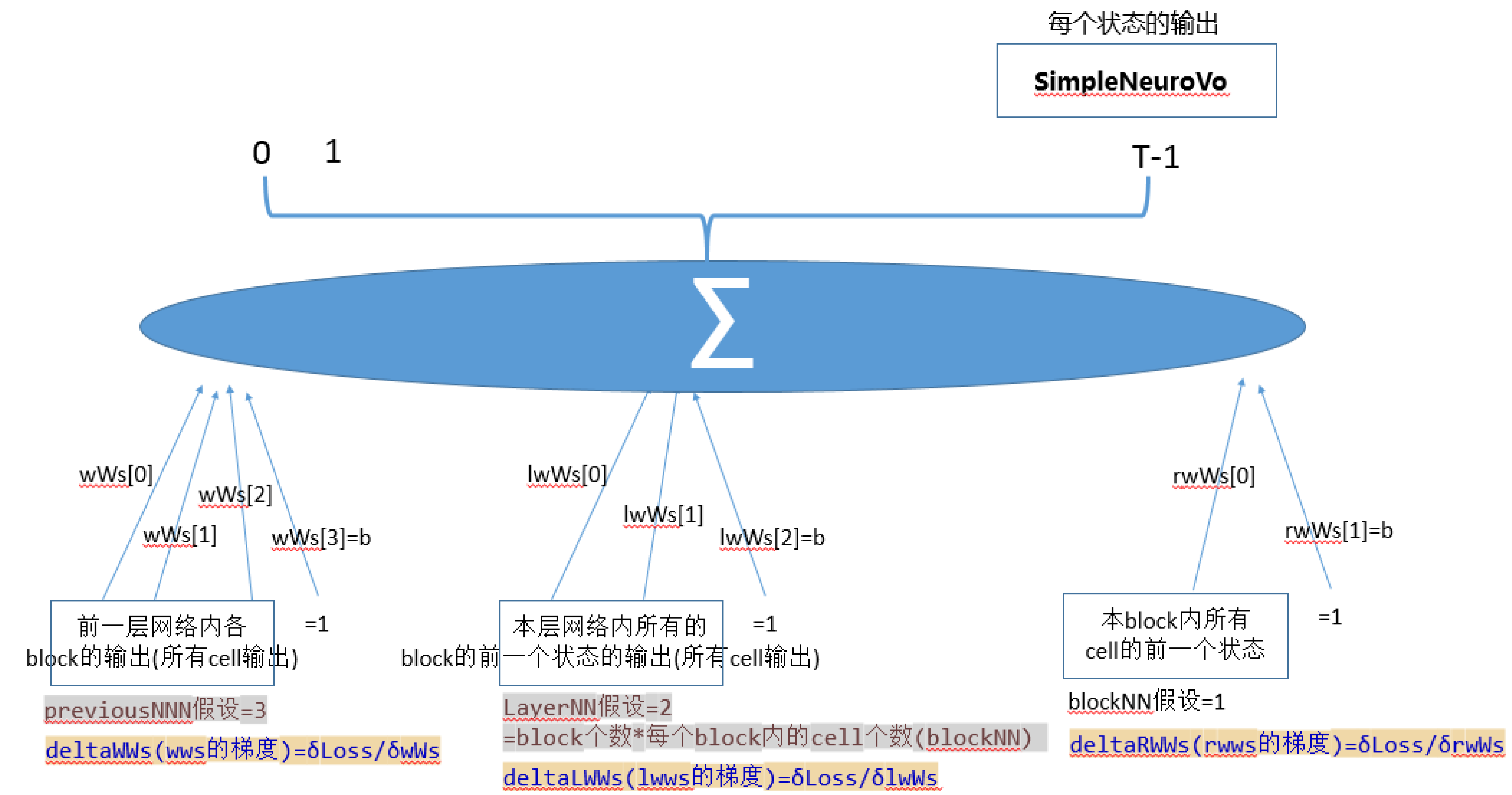
正向传播

代码位置:lstm>BPTT.java
代码作业：LSTM/RNN网络使用BPTT算法实现的核心构建,包括先前ftt和向后bptt两个大功能

```
/*
 * 向前传播算法的过程
 * 对一个输入序列做向前传播:BPTT.java的fTT()
 * 引用了
 * RNNLayer.java中的fTT()
 * 引用了
 * 对某一层网络的向前传播:BPPT.java的fTT4RNNLayer()
 * 如果是rnn则引用了(fTT4RNNLayer的输入参数为RNNLayer layer)
 *     对某一层的部分神经元/blocks的向前传播(offset开始的其后length个神经元/blocks):BPTT.java的fTT4PartialRNNLayer()
 *     (上一个状态的加和使用了 fTTRecurrentAa 函数)
 * 如果是lstm则引用了(fTT4RNNLayer的输入参数为LSTMLayer layer)
 *     对某一层的部分神经元/blocks的向前传播(offset开始的其后length个神经元/blocks):BPPT.java的fTT4PartialLstmLayer()
 *     引用了
 *     某一层的某个神经元素/blocks内的某个门(输入/输出/忘记门)的向前传播:BPTT.java的fTTiRNNNeuroVo
 *      (上一个状态的加和使用了 fTTRecurrentAa 函数)
 * */
```

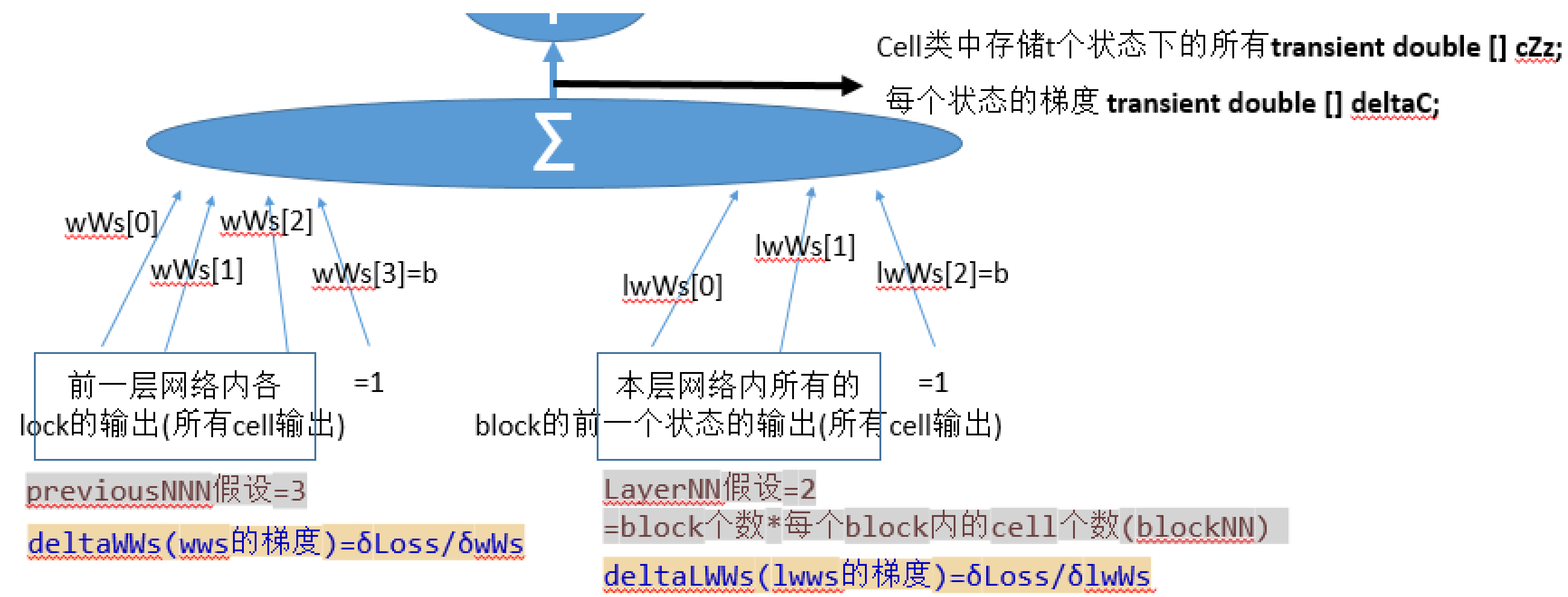
正向传播 –LSTM网络的向前传播-3个门的向前传播

```
public void fTTiRNNNeuroVo(IRNNNeuroVo vo, RNNNeuroVo [] previousVos, Block block,
int scOffset, int binaryPos, boolean speedUpLearning, LSTMLayer layer, int abs) {
/*
 * zZ=sum(本状态的前一层的输入aA[j]*wWs[j])       //zZ + previousVos[j].getNvTT()[t].aA * vo.getwWs()[j];
 *      +
 *    sum(上一个状态的block的cells[j]*RwWs[j]) //zZ + cells[j].getSc()[t - scOffset]* vo.getRwWs()[j]
 *      +
 *    sum(使用t-1状态下的本层神经元的输出*lWws)     //zZ + fTTRecurrentAa(vo, layer.getCells());
 * aA=f.activate(zZ)
 * */
  double zZ = 0;
  if (speedUpLearning) {
    zZ = zZ + vo.getwWs()[binaryPos];
  } else {
    for (int j = 0; j < previousVos.length; j++) {
      zZ = zZ + previousVos[j].getNvTT()[t].aA * vo.getwW
    }
  }
  ICell[] cells = block.getCells();
  for (int j = 0; j < cells.length; j++) {
    if (t >= scOffset) {
      zZ = zZ + cells[j].getSc()[t - scOffset] * vo.getR
    } else {
      zZ = zZ + preCxtSc(layerPos)[abs + j] * vo.getRwWs
    }
  }
  if (cfg.isUseBias()) {
    zZ = zZ + vo.getwWs()[vo.getwWs().length - 1];
  }
  /****
   * <add the activation of last moment>
   * **/
  //zZ = zZ + fTTUseCellAa(vo, cells);
  if (useCAa4Gate) {
```



正向传播 –LSTM网络的向前传播 –block的总体向前传播

```
public void fTT4PartialLstmLayer(Block [] blocks, int offset, int length, RNNNeuroVo [] previousVos, LSTMLayer layer, int
binaryPos, boolean speedUpLearning) {
  //遍历本层中offset以后的length个block
  for (int i = offset; i < offset + length; i++) {
    Block block = blocks[i];
    int abs = 0;
    if (useAbsoluteSc) {
      abs = i;
    }
    //计算block内的输入门的值,最终修改了block.getInputGate()输入门的aA
    fTTiRNNNeuroVo(block.getInputGate(), previousVos, block, 1, binaryPos, speedUpLearning, layer, abs);
    //计算block内的忘记门的值,最终修改了block.getForgetGate()忘记门的aA
    fTTiRNNNeuroVo(block.getForgetGate(), previousVos, block, 1, binaryPos, speedUpLearning, layer, abs);
    ICell[] cells = block.getCells();
    //遍历block内的所有cell,更新每个cell的状态Sc  snv的zZ
    for (int j = 0; j < cells.length; j++) {
      ICell cell = cells[j];
      /* 某个cell的第t个时间状态下的输出部分,参考RNNNeuroVo.java
       * zZ=sum(本状态的前一层的输入aA[j]*wWs[j])       //zZ + previousVos[k].neuroVos[t].aA * cell.getwWs()[k];
       *      +
       *    sum(本次网络所有cell的输出*lwWs[j])          //zZ + fTTRecurrentAa(cell, layer.getCells());
       * 计算好zZ后,把其赋值到底t个时间状态的zZ cell.getCZz()[t] = zZ;
       **/
      SimpleNeuroVo snv = cell.getNvTT()[t];
      /**
       * <apply drop out>
       * ***/
      if (dropOut > 0) {
        if (!isTesting) {
          if (random.nextDouble() > dropOut) {
            snv.dropOut = false;
          } else {
            snv.dropOut = true;
            snv.zZ = 0;
            snv.aA = 0;
```



反向传播

代码位置:lstm>BPTT.java
代码作业：LSTM/RNN网络使用BPTT算法实现的核心构建,包括先前ftt和向后bptt两个大功能

代码中给出了《Supervised-Sequence-Labelling-with-Recurrent-Neural-Networks.pdf》的4.6.2节中的4.11中的同一时刻下一层网络的部分

BPPT.java的bptt()

BPTT.java的bpTT4RNNLayer(LSTMLayer layer)

BPTT.java的bpttFromNextLayer

BPTT.java的bpttPartialFromNextLayer

BPTT.java的bpTT4PartialRNNLayerCell

bpTTUseCellAa

bpTTUseBlocks

BPTT.java的bpTT4PartialRNNLayerBlocks

每个block的Cell部分

每个block的3个门部分

代码中给出了《Supervised-Sequence-Labelling-with-Recurrent-Neural-Networks.pdf》的4.6.2节中的4.12,4.13,4.14,4.15,4.16

代码中给出了《Supervised-Sequence-Labelling-with-Recurrent-Neural-Networks.pdf》的4.6.2节中的4.11中的下一时刻本层网络的部分

反向传播： LSTM反向传播 –主要入口bpTT4RNNLayer

BPTT.java的bpTT4RNNLayer(LSTMLayer layer)

```java
/*对本层网络做bptt反向传播*/
@Override
public void bpTT4RNNLayer(LSTMLayer layer) {
  if (attention != null) {
    if (t + 1 < tLength) {
      attention.bp4RNNLayerAttention(layer, t + 1);
    }
  }
  if (layerPos == cfg.layers.length - 1) {
  }
  //RNNNeuroVo [] nextVos = cfg.layers[layerPos + 1].getRNNNeuroVos();
  //获取本层网络的所有cell
  ICell[] allCells = layer.getCells();
  if (layerPos != cfg.layers.length - 1) {//if lstm is the last layer, it means no need bp for it.
    bpttFromNextLayer(layer, false);//就算公式4.11中下一层网络部分
  } else {
    if (attentionDhj != null) {
      for (int j = 0; j < allCells.length; j++) {
        SimpleNeuroVo vo = allCells[j].getNvTT()[t];
        vo.deltaZz = attentionDhj[t][j];
      }
    } else {
      if (t == tLength - 1 || !cfg.isAutoSequence()) {//auto sequence, it should be ok all the time
        for (int j = 0; j < allCells.length; j++) {
          SimpleNeuroVo vo = allCells[j].getNvTT()[t];
          vo.deltaZz = this.cxtDeltaZz(layerPos)[j];
        }
      } else {
      //把本层内每个block的所有cell的第t个时间状态的局部梯度 deltaZz (局部梯度)= δLoss/δzZ 全部置为0
        for (int j = 0; j < allCells.length; j++) {
          SimpleNeuroVo vo = allCells[j].getNvTT()[t];
          vo.deltaZz = 0;
        }
      }
```

反向传播：LSTM反向传播 –block中的反向传播公式 4.11中同一时间状态的下一层网络的部分

BPTT.java的
bpttPartialFromNextLayer

```
//某一个隐藏层的反向传播代码入口
public void bpttPartialFromNextLayer(IRNNLayer nextLayer,  RNNNeuroVo [] vos, IRNNLayer layer, boolean useDeActivate, int offset,
int length, boolean addtive) {
  //do we need to reset all the values?
  if (nextLayer instanceof RNNLayer) {//rnn网络
    for (int i = offset; i < offset + length; i++) {
      SimpleNeuroVo vo = vos[i].getNvTT()[t];
      RNNNeuroVo[] nextVos = nextLayer.getRNNNeuroVos();
      double s = 0;
      for (int j = 0; j < nextVos.length; j++) {
        SimpleNeuroVo vo1 = nextVos[j].getNvTT()[t];
        s = s + vo1.deltaZz * nextVos[j].getwWs()[i];
      }
      if (layer instanceof RNNLayer && isHiddenLayer(layerPos)) {
        s = s + bpTTRecurrentAa(i, vos);
      }
      if (useDeActivate) {
        vo.deltaZz = s * f.deActivate(vo.zZ);
      } else {
        if (addtive) {
          vo.deltaZz = vo.deltaZz + s;
        } else {
          vo.deltaZz = s;
        }
      }
    }
  } else if (nextLayer instanceof LSTMLayer) {//下一层网络是lstm网络
    LSTMLayer nlayer = (LSTMLayer) nextLayer;//获取下一层网络
    //遍历本层网络的offset开始的length个block
    for (int i = offset; i < offset + length; i++) {
      SimpleNeuroVo vo = vos[i].getNvTT()[t];//本层中某个block的cell
      Block [] blocks = nlayer.getBlocks();//获取下一层网络的所有blocks
      double s = 0;
      for (int j = 0; j < blocks.length; j++) {
```

反向传播：LSTM反向传播–block中的反向传播公式 4.11中下一时间状态的同一层网络的部分

BPTT.java的
bpTT4PartialRNNLayerCell

```
public void bpTT4PartialRNNLayerCell(ICell[] allCells, LSTMLayer layer, int offset, int length) {
  for (int i = offset; i < offset + length; i++) {
    ICell cell = allCells[i];
    //double sc = cell.getSc()[t];
    SimpleNeuroVo vo = cell.getNvTT()[t];
    double s = 0;
    //<add cell activation>
    s = s + bpTTUseCellAa(i, layer.getCells());/***cells should be from layer***/
    //</add cell activation>
    s = s + bpTTUseBlocks(i, layer.getBlocks());
    //the deltaZz is re-initialized during next layer.
    vo.deltaZz = vo.deltaZz + s;
    /*drop out
     * **/
    if (dropOut > 0) {
      if (!isTesting) {
        if (vo.dropOut) {//it is checked before.
          vo.deltaZz = 0;
        }
      }
    }
    /*drop out
     * **/
  }
}
```

反向传播：LSTM反向传播–block中的反向传播公式 4.11中下一时间状态的同一层网络的部分-2

BPTT.java的bpTTUseCellAa和bpTTUseBlocks

```
//公式4.11的t+1时刻的本层部分--每个block的cell部分
public double bpTTUseCellAa(int pos, ICell[] cells) {
  double s = 0;
  if (enableUseCellAa && t < tLength -1) {
    for (int k = 0; k < cells.length; k++) {
      ICell lastTCell = cells[k];
      s = s + lastTCell.getDeltaC()[t + 1] * lastTCell.getLwWs()[pos];
    }
  }
  return s;
}

boolean useCAa4Gate = true;
//公式4.11的t+1时刻的本层部分--每个block的3个门部分
public double bpTTUseBlocks(int pos, IBlock [] blocks) {
  double s = 0;
  if (useCAa4Gate && t < tLength -1) {
    for (int k = 0; k < blocks.length; k++) {
      IBlock lastTBlock = blocks[k];
      SimpleNeuroVo fVo_t = getIRNNNeuroVo(lastTBlock.getForgetGate(), t + 1);
      SimpleNeuroVo iVo_t = getIRNNNeuroVo(lastTBlock.getInputGate(), t + 1);
      SimpleNeuroVo oVo_t = getIRNNNeuroVo(lastTBlock.getOutPutGate(), t + 1);
      s = s + fVo_t.deltaZz * lastTBlock.getForgetGate().getLwWs()[pos]
            + iVo_t.deltaZz * lastTBlock.getInputGate().getLwWs()[pos]
            + oVo_t.deltaZz * lastTBlock.getOutPutGate().getLwWs()[pos];
    }
  }
  return s;
}
```

反向传播：LSTM反向传播 –block中的反向传播公式 4.12,4.13,4.14,4.15,4.16

BPTT.java的
bpTT4PartialRNNLayerBlocks

```
public void bpTT4PartialRNNLayerBlocks(Block [] blocks, LSTMLayer layer, int offset, int length) {
  //遍历本层中offset以后的length个block
  for (int i = offset; i < offset + length; i++) {
    Block block = blocks[i];
    int abs = 0;
    if (useAbsoluteSc) {
      abs = i;
    }
    ICell[] cells = block.getCells();
    double outGateDeltaZz = 0;
    IOutputGate outGate = block.getOutPutGate();
    IInputGate inGate = block.getInputGate();
    IForgetGate fGate = block.getForgetGate();
    for (int j = 0; j < cells.length; j++) {
      ICell cell = cells[j];
      double sc = cell.getSc()[t];
      SimpleNeuroVo vo = cell.getNvTT()[t];
      //输出门的梯度 公式4.12
      outGateDeltaZz = outGateDeltaZz + vo.deltaZz * h.activate(sc) * f.deActivate(outGate.getNvTT()[t].zZ);
    }
    getIRNNNeuroVo(outGate, t).deltaZz = outGateDeltaZz;
    for (int j = 0; j < cells.length; j++) {
      ICell cell = cells[j];
      double sc = cell.getSc()[t];
      SimpleNeuroVo vo = getIRNNNeuroVo(cell, t);
      double deltaSc = vo.deltaZz * outGate.getNvTT()[t].aA * h.deActivate(sc)+ outGateDeltaZz * outGate.getRwWs()[j];
      if (t < tLength - 1) {
        SimpleNeuroVo fgVo = getIRNNNeuroVo(fGate, t + 1);
        SimpleNeuroVo inVo = getIRNNNeuroVo(inGate, t + 1);
        //cell的梯度 公式4.13
        deltaSc =deltaSc+cell.getDeltaSc()[t + 1]*fgVo.aA+fgVo.deltaZz*fGate.getRwWs()[j]+inVo.getDeltaZz()*inGate.getRwWs()[j];
      } else {
        if (layerPos == cfg.layers.length - 1) {
          deltaSc = deltaSc + cxtDeltaSc(layerPos)[abs + j];
```

请开始你们的表演