DeepDriver解密之一

本文作者： 李明 蔡龙军

DeepDriver创建者：蔡龙军

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

DeepDriver的ANN代码导读

-- 普通ANN（ArtifactNeuroNetworkV2：包括dropout,无预训练过程）

总体分析

代码位置:

LogicsticsActivationFunction
(ann/imp/LogicsticsActivationFunction.java)
每个神经元的非线性输出函数

InputParameters
(ann/InputParameters.java)
模型的参数以及数据集

ArtifactNeuroNetworkV2
(ann/ArtifactNeuroNetworkV2.java)

内部引用

LayerImpV2
(ann/imp/LayerImpV2.java)

内部引用

NeuroUnitImpV3
(ann/imp/NeuroUnitImpV3.Java)

增加dropout,以及
更新系数功能
(一个完整可用的

继承

继承

继承

ArtifactNeuroNetwork
(ann/ArtifactNeuroNetwork.java)

内部引用

LayerImp
(ann/imp/LayerImp.java)

内部引用

NeuroUnitImpV2
(ann/imp/NeuroUnitImpV2.Java)

优化初始化系数方

继承

ANNCfg
(ann/ANNCfg.java)
并行化线程个数，是否droupout

NeuroUnitImp
(ann/imp/NeuroUnitImp.Java)

实现BP

DNN网络

一层网络ILayer

一个神经元INeuroUnit

参数 InputParameters

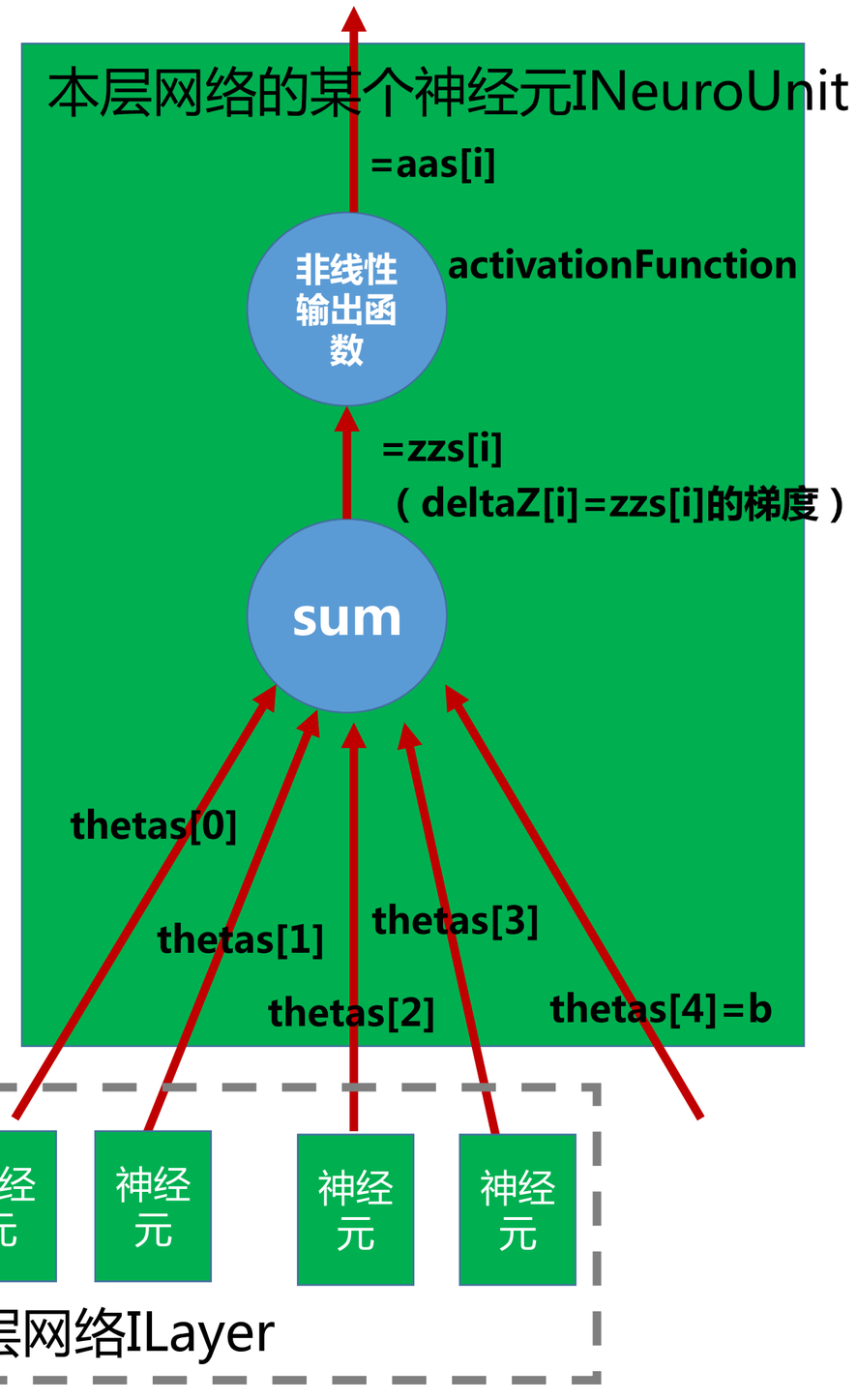
InputParameters
(ann/InputParameters.java)

属性	
double alpha = 0.1;	学习率lr 见ann/imp/NeuroUnitImpV3.Java中 **bpUpdateWws方法**
double [][] input;	每个元素表示一个样本的x,总体表示了一批样本的x
double [] result;	每个元素表示一个样本的y,总体表示了一批样本的y 这里的y是连续或0/1？？？？？
double [][] result2;	每个元素表示一个样本的y,总体表示了一批样本的y 这里的y是多分类？？？？
double m = -1;	梯度下降更新时使用的冲量 见ann/imp/NeuroUnitImpV3.Java中 **bpUpdateWws方法**
int iterationNum = 300000;	迭代次数
double lamda = 0.00001;	L2泛化系数 见ann/imp/NeuroUnitImpV3.Java中 **bpUpdateWws方法**
int [] neuros;	每一层网络包括的神经元个数
int layerNum = 1;	多少层神经网络=**neuros的长度**

某个神经元 NeuroUnitImp-属性

NeuroUnitImp
(ann/imp/NeuroUnitImp.Java)

属性	
Random *random* = ***new Random(System.currentTimeMillis());***	随机数种子,用于初始化连接系数
protected double min = 0;	用于初始化连接系数
protected double max = 1.0;	用于初始化连接系数
protected double length = max - min;	用于初始化连接系数
protected IActivationFunction activationFunction;	该神经元的非线性输出函数
protected double [] aas;	aas[i]表示第i个样本经过该神经元后的输出(经过了非线性输出函数)
protected double [] zzs;	zzs[i]表示第i个样本经过该神经元后的输出(没有经过非线性输出函数)
protected double [] deltaZ;	**deltaZ**[i]表示第i个样本经过该神经元后，由反向传播公式计算得到的导数（局部梯度）
protected double [] thetas;	该神经元的所有输入连接的系数w 最后一个元素表示b
protected double [] deltaThetas;	每个系数的导数 也是本次迭代的变化方向



某个神经元 NeuroUnitImp-方法：向前传播 forwardPropagation

NeuroUnitImp
(ann/imp/NeuroUnitImp.Java)

```
@Override
public void forwardPropagation(List<INeuroUnit> previousNeuros, double [][] input) {
  if (thetas == null) {
    this.thetas = new double[previousNeuros.size() + 1];
    initTheta();//初始化系数
  }
  this.aas = new double[input.length];//本批输入样本的个数,然后初始化这些个aas
  zzs = new double[input.length];
  for (int i = 0; i < aas.length; i++) {//遍历每个输入样本
    //Z=sum(上一层的输出【previousNeuros.get(j).getAaz(i)】*系数【thetas[j]】)+b【thetas[thetas.length - 1]】
    double z = 0;
    for (int j = 0; j < previousNeuros.size(); j++) {
      z = z + thetas[j] * previousNeuros.get(j).getAaz(i);
    }
    z = z + thetas[thetas.length - 1];
    zzs[i] = z;
    double a = activationFunction.activate(z);//经过非线性输出函数
    aas[i] = a;
  }
}
```

某个神经元 NeuroUnitImp-方法：向后传播 backPropagation

NeuroUnitImp
(ann/imp/NeuroUnitImp.Java)

```
@Override
public void backPropagation(List<INeuroUnit> previousNeuros, List<INeuroUnit> nextNeuros, double [][] result, InputParameters
parameters) {
/*初始化*/
if (deltaZ == null) {
  deltaThetas = new double[thetas.length];//为每个系数w初始化一个局部梯度
}
deltaZ = new double[aas.length];//为每个样本产生的zzs(长度等于aas)初始化一个局部梯度

/*计算局部梯度deltaZ*/
if (nextNeuros == null) {//如果输出层为空,即本层就是输出层
  for (int i = 0; i < deltaZ.length; i++) {//遍历每个样本
    deltaZ[i] = (aas[i] - result[i][position]) * activationFunction.deActivate(zzs[i]);//输出层的局部梯度计算方式
  }
} else {//输入是中间层
  for (int i = 0; i < deltaZ.length; i++) {//遍历每个样本
    double sumDelta = 0;
    for (int j = 0; j < nextNeuros.size(); j++) {//遍历下一层网络的每一个神经元，并计算
      sumDelta = sumDelta + nextNeuros.get(j).get4PropagationPreviousDelta(i, position);
    }
    deltaZ[i] = (sumDelta) * activationFunction.deActivate(zzs[i]);//隐藏层的局部梯度
  }
}

/*计算每个系数的梯度deltaThetas*/
if (thetas == null) {
  return;
}
for (int i = 0; i < thetas.length; i++) {//遍历系数
  double delta4theta = 0;
  if (i < thetas.length - 1) {//系数w
    for (int j = 0; j < deltaZ.length; j++) {//遍历每个样本
      delta4theta = delta4theta + deltaZ[j] * previousNeuros.get(i).getAaz(j);
    }
```

$$\delta_i^{(2)} = \left(\sum_{j=1}^{s_2} W_{ji}^{(3)} \delta_j^{(3)} \right) f'(z_i^{(2)}),$$

```
@Override
public double get4PropagationPreviousDelta(int dataIndex,
        int previouNeuroIndex) {
    return deltaZ[dataIndex] * thetas[previouNeuroIndex];
}
```

某个神经元 NeuroUnitImpV2-属性

NeuroUnitImpV2
(ann/imp/NeuroUnitImpV2.Java)

和NeuroUnitImp相比,优化了初始化系数的公式（Xavier初始化）

```
protected void initTheta() {
  double b = Math.pow(6.0/(double)(layer.getNeuros().size() + layer.getPreviousLayer().getNeuros().size()), 0.5);
  length = 2*b;
  min = -b;
  max = b;
  if (randomize) {
    for (int i = 0; i < thetas.length; i++) {
      thetas[i] = length * random.nextDouble()+ min;
    }
  }
}
```

tion procedure to approximately satisfy our objectives of maintaining activation variances and back-propagated gradients variance as one moves up or down the network. We call it the **normalized initialization**:

$$W \sim U\Big[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\Big] \qquad (16)$$

某个神经元 NeuroUnitImpV3-属性

NeuroUnitImpV3
(ann/imp/NeuroUnitImpV3.Java)

boolean dropOut;该神经单元是否droupout

某个神经元 NeuroUnitImpV3-反向传播总体

NeuroUnitImpV3
(ann/imp/NeuroUnitImpV3.Java)

首先：根据BP算法计算出每个样本经过该神经元后产生的局部梯度deltaZ[i]

backPropagation方法

$$\delta_i^{(2)} = \left(\sum_{j=1}^{s_2} W_{ji}^{(3)} \delta_j^{(3)} \right) f'(z_i^{(2)}),$$

然后：根据梯度下降公式更新系数W/b

updateSelf方法

引用

bpUpdateWws方法

计算经过一批样本的训练后，每个系数W的梯度deltaThetas[i]
Delta4theta=sum(第j个样本产生的局部梯度deltaZ[j]*上一层连接的输出【第j个样本】)
deltaThetas[i] = - parameters.getAlpha() * delta4theta;

使用L2泛化和冲量momentum
1）使用冲量momentum计算本次系数thetas[i]的变化量
deltaW = deltaThetas[i] + momentum * lastDeltaThetas[i];
2）使用L2泛化，更新本次迭代的第i个系数值
thetas[i] = thetas[i] + deltaW - getAlpha()* lamda* thetas[i] ;
3）把deltaW记录为上次迭代的系数变化,用于下一次迭代使用
lastDeltaThetas[i] = deltaW;

某个神经元 NeuroUnitImpV3-backPropagation方法（同NeuroUnitImp一样

NeuroUnitImpV3
(ann/imp/NeuroUnitImpV3.Java)

首先：根据BP算法计算出每个样本经过该神经元后
产生的局部梯度deltaZ[i]

```
@Override
public void backPropagation(List<INeuroUnit> previousNeuros, List<INeuroUnit> nextNeuros, double [][] result,
InputParameters parameters) {
  this.previousNeuros = previousNeuros;
  this.parameters = parameters;
  if (deltaZ == null) {
    if (thetas != null) {
      deltaThetas = new double[thetas.length];
    }
  }
  deltaZ = new double[aas.length];
  if (nextNeuros == null) {
    for (int i = 0; i < deltaZ.length; i++) {
      deltaZ[i] = (aas[i] - result[i][position]) * activationFunction.deActivate(zzs[i]);
    }
  } else {
    for (int i = 0; i < deltaZ.length; i++) {
      double sumDelta = 0;
      for (int j = 0; j < nextNeuros.size(); j++) {
        sumDelta = sumDelta + nextNeuros.get(j).get4PropagationPreviousDelta(i, position);
      }
      deltaZ[i] = (sumDelta) * activationFunction.deActivate(zzs[i]);
    }
  }
}
```

某个神经元 NeuroUnitImpV3-bpUpdateWws方法

NeuroUnitImpV3
(ann/imp/NeuroUnitImpV3.Java)

计算经过一批样本的训练后，每个系数W的梯度deltaThetas[i]
Delta4theta=sum(第j个样本产生的局部梯度deltaZ[j]*上一层连接的输出【第j个样本】)
deltaThetas[i] = - parameters.getAlpha() * delta4theta;

```
public void bpUpdateWws() {
  if (thetas == null) {
    return;
  }
  for (int i = 0; i < thetas.length; i++) {
    double delta4theta = 0;
    if (i < thetas.length - 1) {
      for (int j = 0; j < deltaZ.length; j++) {
        delta4theta = delta4theta + deltaZ[j] * previousNeuros.get(i).getAaz(j);
      }
    } else {
      for (int j = 0; j < deltaZ.length; j++) {
        delta4theta = delta4theta + deltaZ[j];
      }
    }
    deltaThetas[i] = - parameters.getAlpha() * delta4theta;
    if (parameters.getM() > 0) {
      this.momentum = parameters.getM();
    }
    setAlpha(parameters.getAlpha());
    this.lamda = parameters.getLamda();
  }
}
```

某个神经元 NeuroUnitImpV3-bpUpdateWws方法

NeuroUnitImpV3
(ann/imp/NeuroUnitImpV3.Java)

使用L2泛化和冲量momentum

1）使用冲量momentum计算本次系数thetas[i]的变化量

deltaW = deltaThetas[i] + momentum * lastDeltaThetas[i]; 本次更新w的方向=本次计算的方向+上一次更新w的方向*m（防

2）使用L2泛化，更新本次迭代的第i个系数值

thetas[i] = thetas[i] + deltaW - getAlpha()* lamda* thetas[i] ;

3）把deltaW记录为上次迭代的系数变化,用于下一次迭代使用

lastDeltaThetas[i] = deltaW;

```
@Override
public void updateSelf() {
  bpUpdateWws();
  if (thetas == null) {
    return;
  }
  if (lastDeltaThetas == null) {
    lastDeltaThetas = new double [deltaThetas.length];
  }
  double deltaW = 0;
  for (int i = 0; i < thetas.length; i++) {//更新某个系数w[i]或b
    //no regularization
    //thetas[i] = thetas[i] + deltaThetas[i];
    /**
     * Add momentum to accelerate
     * */
     deltaW = deltaThetas[i] + momentum * lastDeltaThetas[i];
     /**
      * Add regularization to avoid overfitting
      * **/
     if (i == thetas.length - 1) {//更新b
       thetas[i] = thetas[i] + deltaW; //deltaThetas[i];
     } else {//更新w
       thetas[i] = thetas[i] + deltaW - getAlpha()* lamda*  thetas[i] ; //deltaThetas[i];
```

冲量：momentum

“冲量”这个概念源自于物理中的力学，表示力对时间的积累效应。

在普通的梯度下降法 `x += v` 中，每次 `x` 的更新量 `v` 为 `v = - dx * lr` ，其中 `dx` 为目标函数 `func(x)` 对 `x` 的一阶导数，。

当使用冲量时，则把每次 `x` 的更新量 `v` 考虑为本次的梯度下降量 `- dx * lr` 与上次 `x` 的更新量 `v` 乘上一个介于 `[0, 1]` 的因子 `momentum` 的和，即 `v = - dx * lr + v * momemtum` 。

从公式上可看出：

- 当本次梯度下降 `- dx * lr` 的方向与上次更新量 `v` 的方向**相同**时，上次的更新量能够本次的搜索起到一个正向**加速**的作用。
- 当本次梯度下降 `- dx * lr` 的方向与上次更新量 `v` 的方向**相反**时，上次的更新量能够本次的搜索起到一个**减速**的作用。

某一层神经网络 LayerImpV2

LayerImpV2
(ann/imp/LayerImpV2.java)

int pos;	本层的标号(继承LayerImp)
ILayer nextLayer;	指向下一层的指针(继承LayerImp)
ILayer previousLayer;	指向上一层的指针(继承LayerImp)
List<INeuroUnit> neuros **= new ArrayList<INeuroUnit>();**	本层所有的神经元(继承LayerImp)
ANNCfg aNNCfg;	是否dropout以及并行化线程个数
double [] rs;	一个批量向前后的总误差？？？？
double [][] rss;	？？？？

某一层神经网络 LayerImpV2 –buildup

LayerImpV2
(ann/imp/LayerImpV2.java)

```
@Override
public void buildup(ILayer previousLayer, double[][] input, IActivationFunction acf, boolean isLastLayer, int neuroCount) {
  setPreviousLayer(previousLayer);
  if (previousLayer != null) {
    previousLayer.setNextLayer(this);                    //设置本层网络为前一层网络的下一层网络指针
  }
  if (previousLayer == null) {
    updateValues4FirstLayer(input);                      //如果前一层为null则表示是建立输入层
  } else {//input[0].length
    for (int i = 0; i < neuroCount; i++) {               //依次建立该层网络中的每个神经元
      NeuroUnitImp neuroUnitImp = createNeuroUnitImp();  //创建一个NeuroUnitImpV3神经元
      neuroUnitImp.buildup(input, i);
      neuroUnitImp.setActivationFunction(acf);           //设置神经元的非线性输出函数
      getNeuros().add(neuroUnitImp);                     //把该神经元插入到该层神经网络中
    }
  }
}

public NeuroUnitImp createNeuroUnitImp() {
  return new NeuroUnitImpV3(this);
}
```

某一层神经网络 LayerImpV2 –向前传播forwardPropagation

LayerImpV2
(ann/imp/LayerImpV2.java)

```
public void forwardPropagation(double[][] input) {
  。。。。。。。。。
  forwardPropagation4PartialLayer(input);
  。。。。。。。。。
  计算误差rs
}
```

—批样本做向前传播

—批样本对于部分系数做向前传播

```
ThreadParallel threadParallel = new ThreadParallel();
public void forwardPropagation4PartialLayer(final double [][] input) {
  int tn = 1;
  if (aNNCfg == null || (tn = aNNCfg.getThreadsNum()) <= 1) {
    forwardPropagation4PartialLayer(input,0, neuros.size());
  } else {
    threadParallel.runMutipleThreads(neuros.size(), new PartialCallback() {
      public void runPartial(int offset, int runLen) {
        forwardPropagation4PartialLayer(input, offset,runLen);
      }
    }, tn);
  }
}
```

并行化运行,每个线程运行offset
以后的length个系数的向前传播算法

```
public void forwardPropagation4PartialLayer(double [][] input, int offset, int length) {
  if (getPreviousLayer() == null) {
    updateValues4PartialFirstLayer(input, offset, length);
    return;
  }
  for (int i = offset; i < offset + length; i++) {//依次处理offset开始的length个神经元
    INeuroUnit neuro = neuros.get(i);
    neuro.forwardPropagation(this.getPreviousLayer().getNeuros(), input);//神经元i做向前传播
  }
}
```

运行offset以后的length
个系数的向前传播算法

某一层神经网络 LayerImpV2 –向后传播backPropagation

LayerImpV2
(ann/imp/LayerImpV2.java)

一批样本做向后传播

```
public void backPropagation(double [][] finalResult, InputParameters parameters) {
  if (enableDropOut()) {
    if (firstBp) {
      backPropagation4PartialLayer(finalResult, parameters);//并行化处理
      firstBp = false;
    }
    bp4DropOut(finalResult, parameters);
  } else {
    backPropagation4PartialLayer(finalResult, parameters);  //并行化处理
  }
  if (getNextLayer() == null && costFunction != null) {
    for (int i = 0; i < finalResult.length; i++) {
      costFunction.setzZIndex(i);
      costFunction.setTarget(finalResult[i]);
      costFunction.caculateCostError();                     //计算cost
    }
  }
}
```

并行化运行,每个线程运行offset
以后的length个系数的向后传播算法

```
public void backPropagation4PartialLayer(final double [][] finalResult, final InputParameters parameters) {
  int tn = 1;
  if (aNNCfg == null || (tn = aNNCfg.getThreadsNum()) <= 1) {
    backPropagation4PartialLayer(finalResult, parameters,0, neuros.size());
  } else {
    threadParallel.runMutipleThreads(neuros.size(), new PartialCallback() {
      public void runPartial(int offset, int runLen) {
        backPropagation4PartialLayer(finalResult, parameters, offset,runLen);
      }}, tn);
  }
}

public void backPropagation4PartialLayer(double [][] finalResult, InputParameters parameters,
  int offset, int length) {
  for (int i = offset; i < offset + length; i++) {
```

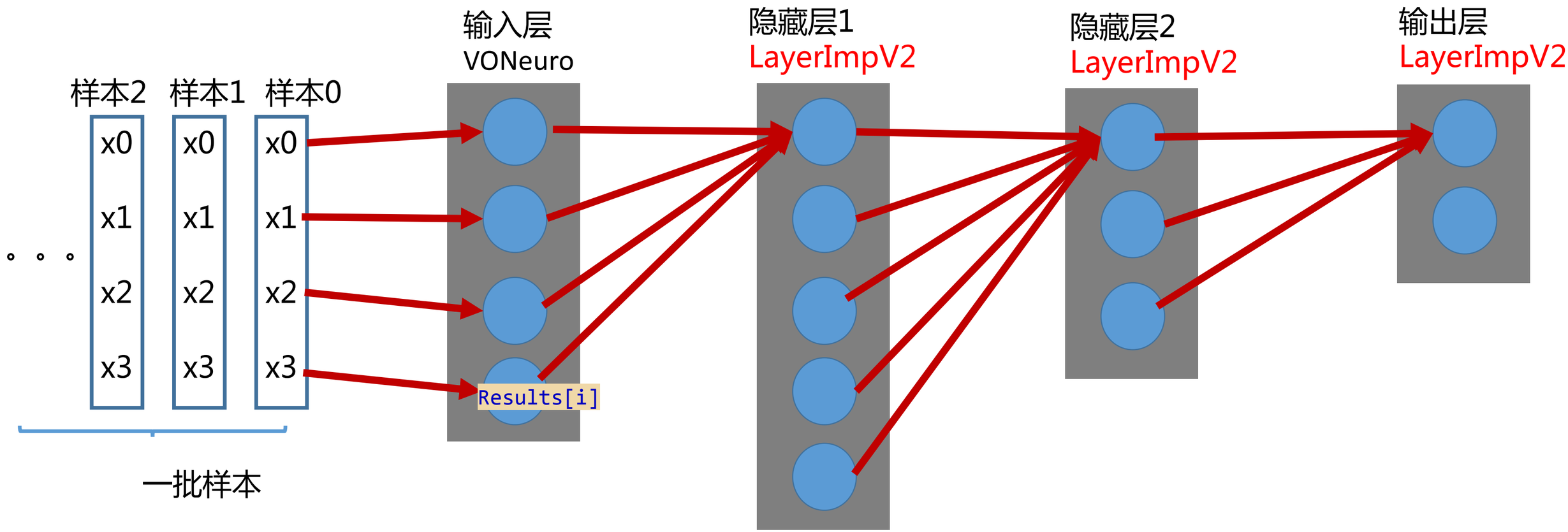
某一层神经网络 LayerImpV2 –dropout

LayerImpV2
(ann/imp/LayerImpV2.java)

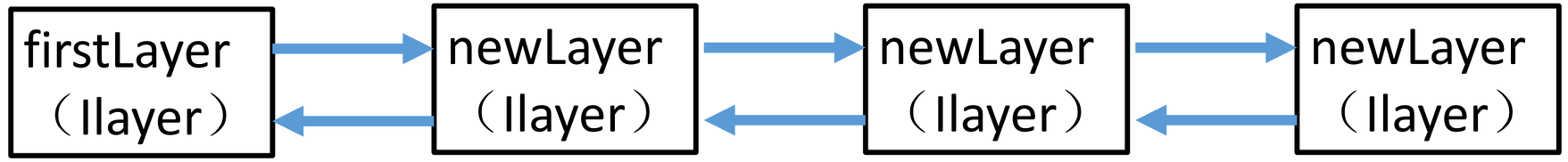
```
private void bp4DropOut(final double [][] finalResult, final InputParameters parameters) {
  int tn = 1;
  if (aNNCfg == null || (tn = aNNCfg.getThreadsNum()) <= 1) {
    bp4DropOut4Partial(finalResult, parameters,0, neuros.size());
  } else {
    threadParallel.runMutipleThreads(neuros.size(), new PartialCallback() {
      public void runPartial(int offset, int runLen) {
        bp4DropOut4Partial(finalResult, parameters, offset,runLen);
      }}, tn);
  }
}

private void bp4DropOut4Partial(double [][] finalResult, InputParameters parameters,
  int offset, int length) {
for (int i = offset; i < offset + length; i++) {                    //遍历offset以后的length个神经元
  NeuroUnitImpV3 neuro = (NeuroUnitImpV3) neuros.get(i);
  if (enableDropOut()) {
    if (aNNCfg.isTesting()) {//测试阶段(模型预测)不做dropout    by default, there is no bp in testing.
    } else {//训练阶段才做dropout
      if (neuro.isDropOut()) {          //如果这个神经元需要dropout，则该神经元所经过的所有样本的局部梯度DeltaZ全部为0
        double [] dzs = neuro.getDeltaZ();//把第i个神经元的每个样本产生的局部梯度 都变为0   （局部梯度为0则不更新模型）
        for (int j = 0; j < dzs.length; j++) {
          dzs[j] = 0;
        }
      } else {
        neuro.backPropagation(                                      //如果不需要dropout则进行反向传播
          getPreviousLayer() == null? null :this.getPreviousLayer().getNeuros(),
          getNextLayer() == null? null : this.getNextLayer().getNeuros(),
          finalResult,
          parameters);
      }
    }
  }
```

神经网络 ArtifactNeuroNetworkV2-buildup

代码结合
ArtifactNeuroNetwork (ann/ArtifactNeuroNetwork.java)
和
ArtifactNeuroNetworkV2 (ann/ArtifactNeuroNetworkV2.java



双向表链结构



虚拟添加了一个输入层,实际是没有的

InputParameters中的layerNum=3
InputParameters中的neuros=[5,3,2]

神经网络 ArtifactNeuroNetworkV2-buildup2

代码在ArtifactNeuroNetwork (ann/ArtifactNeuroNetwork.java) 中的trainModel方法

LayerImpV2 (ann/imp/LayerImpV2.java)

```
public void trainModel(InputParameters parameters) {
  //1.set value into first layer
  double [][] input = parameters.getInput();
  if (useNormalizer) {
    input = normalizer.transformParameters(parameters.getInput());
  }
  double [][] result = getResults(parameters);
  IActivationFunction acf = createAcf();
  firstLayer = createLayer();
  debugPrint("Begin to build up the ann:");
  firstLayer.buildup(null, input, acf, false, input[0].length);
  ILayer tlayer = firstLayer;
  for (int i = 0; i < parameters.getLayerNum(); i++) {
    ILayer newLayer = createLayer();
    int neuroCnt = input[0].length;
    if (parameters.getNeuros() != null) {
      neuroCnt = parameters.getNeuros()[i];
    }
    newLayer.setPos(i+1);
    newLayer.buildup(tlayer, input, acf,
    i == parameters.getLayerNum() - 1, neuroCnt);
    tlayer = newLayer;
  }
  debugPrint("Complete to build ann");
  //2.set value into first layer
  debugPrint("Begin training.");
  double error = 0;
  int errorCnt = 0;
  for (int i = 0; i < parameters.getIterationNum(); i++) {
    debugPrint("Iteration "+(i+1));
    error = 0;
```

建立双向表链结构

```
@Override
public void buildup(ILayer previousLayer, double[][]
input, IActivationFunction acf, boolean isLastLayer,
int neuroCount) {
  setPreviousLayer(previousLayer);
  if (previousLayer != null) {
    previousLayer.setNextLayer(this);
  }
  if (previousLayer == null) {
    updateValues4FirstLayer(input);
  } else {//input[0].length
    for (int i = 0; i < neuroCount; i++) {
      NeuroUnitImp neuroUnitImp = createNeuroUnitImp();
      NeuroUnitImp neuroUnitImp = createNeuroUnitImp();
      neuroUnitImp.buildup(input, i);
      neuroUnitImp.setActivationFunction(acf);
      getNeuros().add(neuroUnitImp);
    }
  }
}

public NeuroUnitImp createNeuroUnitImp() {
  return new NeuroUnitImpV3(this);
}
```

神经网络 ArtifactNeuroNetworkV2-训练模型1

代码在ArtifactNeuroNetwork (ann/ArtifactNeuroNetwork.java) 中的trainModel方法

```
public void trainModel(InputParameters parameters) {
  //1.set value into first layer
  //2.set value into first layer
  debugPrint("Begin training.");
  double error = 0;
  int errorCnt = 0;
  for (int i = 0; i < parameters.getIterationNum(); i++) {//迭代getIterationNum次
    debugPrint("Iteration "+(i+1));
    error = 0;
    //1. optimize the ann one by one
    if (isIncrementalMode) {//样本0进行fp->bp，样本1进行fp->bp，。。。。。，样本n进行fp->bp
      for (int j = 0; j < input.length; j++) {
        error = error + runEpoch(input[j], j, result[j], parameters);
      }
    } else {//所有样本fp->所有样本bp
      error = runEpoch(input, i, result, parameters);//完成一个批次的训练
    }
    //2. optimize the ann over all
    errorCnt++;
    if (errorCnt % 1 == 0) {
      info("Error ="+error);
    }
  }
}
```

神经网络 ArtifactNeuroNetworkV2-训练模型2

代码在ArtifactNeuroNetwork (ann/ArtifactNeuroNetwork.java) 中的runEpoch方法

```
public double runEpoch(double [][] input, int i, double [][] result, InputParameters parameters) {
  double old = 0;
  double newValue = -old;
  ILayer layer = firstLayer;
  ILayer lastLayer = firstLayer;
  while (layer != null) {
    debugPrint("ForwardPropagation "+(i+1)+" on layer "+layer);
    layer.forwardPropagation(input);
    lastLayer = layer;
    layer = layer.getNextLayer();
  }
  layer = lastLayer;
  newValue = lastLayer.getStdError(result);
  old = newValue;
  while (layer != null && firstLayer != layer) {
    debugPrint("BackPropagation "+(i+1)+" on layer "+layer);
    layer.backPropagation(result, parameters);
    lastLayer = layer;
    layer = layer.getPreviousLayer();
  }
  layer = firstLayer;
  while (layer != null) {
    debugPrint("update layer "+(i+1)+" on layer "+layer);
    layer.updateNeuros();
    lastLayer = layer;
    layer = layer.getNextLayer();
  }
  return newValue;
}

public double runEpoch(double [] x, int i, double []y, InputParameters parameters) {
  double [][] result = new double[][]{y};
  double [][] input = new double[][]{x};
```

请开始你们的表演