fit(DataSetIterator iterator)源码阅读

1 网络模型

```java
//Create the network
int numInput = 1;
int numOutputs = 1;
int nHidden = 2;
MultiLayerNetwork net = new MultiLayerNetwork(new NeuralNetConfiguration.Builder()
        .seed(seed)
        .iterations(iterations)
        .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .learningRate(learningRate)
        .weightInit(WeightInit.XAVIER)
        .updater(Updater.SGD)     //To configure: .updater(new Nesterovs(0.9))
        .list()
        .layer(0, new DenseLayer.Builder().nIn(numInput).nOut(nHidden)
                .activation(Activation.RELU).dropOut(0.5)
                .build())
        .layer(1, new OutputLayer.Builder(LossFunctions.LossFunction.MSE)
                .activation(Activation.IDENTITY)
                .nIn(numInput).nOut(numOutputs).build())
        .pretrain(false).backprop(true).build()
);
```

调用`net.fit(iterator);`对源码进行单步阅读。

2 fit(DataSetIterator iterator)

```java
@Override
public void fit(DataSetIterator iterator) {
    DataSetIterator iter;
    // we're wrapping all iterators into AsyncDataSetIterator to provide background prefetch - where appropriate
    if (iterator.asyncSupported()) {
        iter = new AsyncDataSetIterator(iterator, 2);
    } else {
        iter = iterator;
    }

    if (trainingListeners.size() > 0) {
        for (TrainingListener tl : trainingListeners) {
            tl.onEpochStart(this);
        }
    }

    if (layerWiseConfigurations.isPretrain()) {
        pretrain(iter);
        if (iter.resetSupported()) {
            iter.reset();
        }
    }
    if (layerWiseConfigurations.isBackprop()) {
        update(TaskUtils.buildTask(iter));
        if (!iter.hasNext() && iter.resetSupported()) {
            iter.reset();
        }
        while (iter.hasNext()) {
            DataSet next = iter.next();
            if (next.getFeatureMatrix() == null || next.getLabels() == null)
                break;

            boolean hasMaskArrays = next.hasMaskArrays();

            if (layerWiseConfigurations.getBackpropType() == BackpropType.TruncatedBPTT) {
                doTruncatedBPTT(next.getFeatureMatrix(), next.getLabels(), next.getFeaturesMaskArray(),
                                next.getLabelsMaskArray());
            } else {
                if (hasMaskArrays)
                    setLayerMaskArrays(next.getFeaturesMaskArray(), next.getLabelsMaskArray());
                setInput(next.getFeatureMatrix());
                setLabels(next.getLabels());
                if (solver == null) {
                    solver = new
Solver.Builder().configure(conf()).listeners(getListeners()).model(this).build();
                        }
                        solver.optimize();
                    }

                    if (hasMaskArrays)
                        clearLayerMaskArrays();

                    Nd4j.getMemoryManager().invokeGcOccasionally();
                }
        } else if (layerWiseConfigurations.isPretrain()) {
            log.warn("Warning: finetune is not applied.");
        }

        if (trainingListeners.size() > 0) {
            for (TrainingListener tl : trainingListeners) {
                tl.onEpochEnd(this);
            }
        }
}
```

2.1 iterator.asyncSupported()

```
if (iterator.asyncSupported()) {
    iter = new AsyncDataSetIterator(iterator, 2);
} else {
    iter = iterator;
}
```

这里主要判断所给的迭代器是否支持异步，如果支持异步则生成异步迭代器。一般自己实现iterator的时候，对于`asyncSupported`的实现都是`return false;`。

2.2 trainingListeners.size() > 0

```
if (trainingListeners.size() > 0) {
    for (TrainingListener tl : trainingListeners) {
        tl.onEpochStart(this);
    }
}
```

这个`trainingListeners`字段在API文档和对应源码中没有找到对应的解释，从字面意思上是训练监听器。通常使用情况下，不涉及到这个字段

2.3 layerWiseConfigurations.isBackprop()

接下来判断神经网络是否使用Backprop，这个在神经网络的通常情况下，默认值为`true`。

```java
if (layerWiseConfigurations.isBackprop()) {
    update(TaskUtils.buildTask(iter));
    //如果iter没有下一个元素，且iter支持reset操作
    if (!iter.hasNext() && iter.resetSupported()) {
        //则调用一个reset，重置迭代器。
        iter.reset();
    }
    //当迭代器拥有元素的时候
    while (iter.hasNext()) {
        //调用next获取下一个批次需要训练的数据
        DataSet next = iter.next();
        //如果next中的特征矩阵或者标签矩阵为空的时候，则结束训练过程
        if (next.getFeatureMatrix() == null || next.getLabels() == null)
            break;
        //判断当选训练集合是否拥有掩码（掩码通常在RNN中使用，因为RNN可能会处理非等长序列，需要使用掩码-即填0操作，使得非等长序列等长）
        boolean hasMaskArrays = next.hasMaskArrays();

        //这里用于判断网络架构的反向传播类型。（TruncatedBPTT这个是RNN常用的方法，截断式反向传播，BPTT- backprop through time， 主要用于解决梯度消失的问题）
        if (layerWiseConfigurations.getBackpropType() == BackpropType.TruncatedBPTT) {
            doTruncatedBPTT(next.getFeatureMatrix(), next.getLabels(), next.getFeaturesMaskArray(),
                            next.getLabelsMaskArray());
        } else {
            //判断掩码
            if (hasMaskArrays)
                setLayerMaskArrays(next.getFeaturesMaskArray(), next.getLabelsMaskArray());
            //设置特征矩阵
            setInput(next.getFeatureMatrix());
            //设置标签
            setLabels(next.getLabels());

            //初始化Solver
            //Sovle的类标注是Generic purpose solver。简单理解为
            if (solver == null) {
                //根据网络架构构造Sovler
                solver = new Solver.Builder().configure(conf()).listeners(getListeners()).model(this).build();
            }
            solver.optimize();
        }

        if (hasMaskArrays)
            clearLayerMaskArrays();

        Nd4j.getMemoryManager().invokeGcOccasionally();
    }
} else if (layerWiseConfigurations.isPretrain()) {
    log.warn("Warning: finetune is not applied.");
}
```

2.3.1 update(TaskUtils.buildTask(iter));

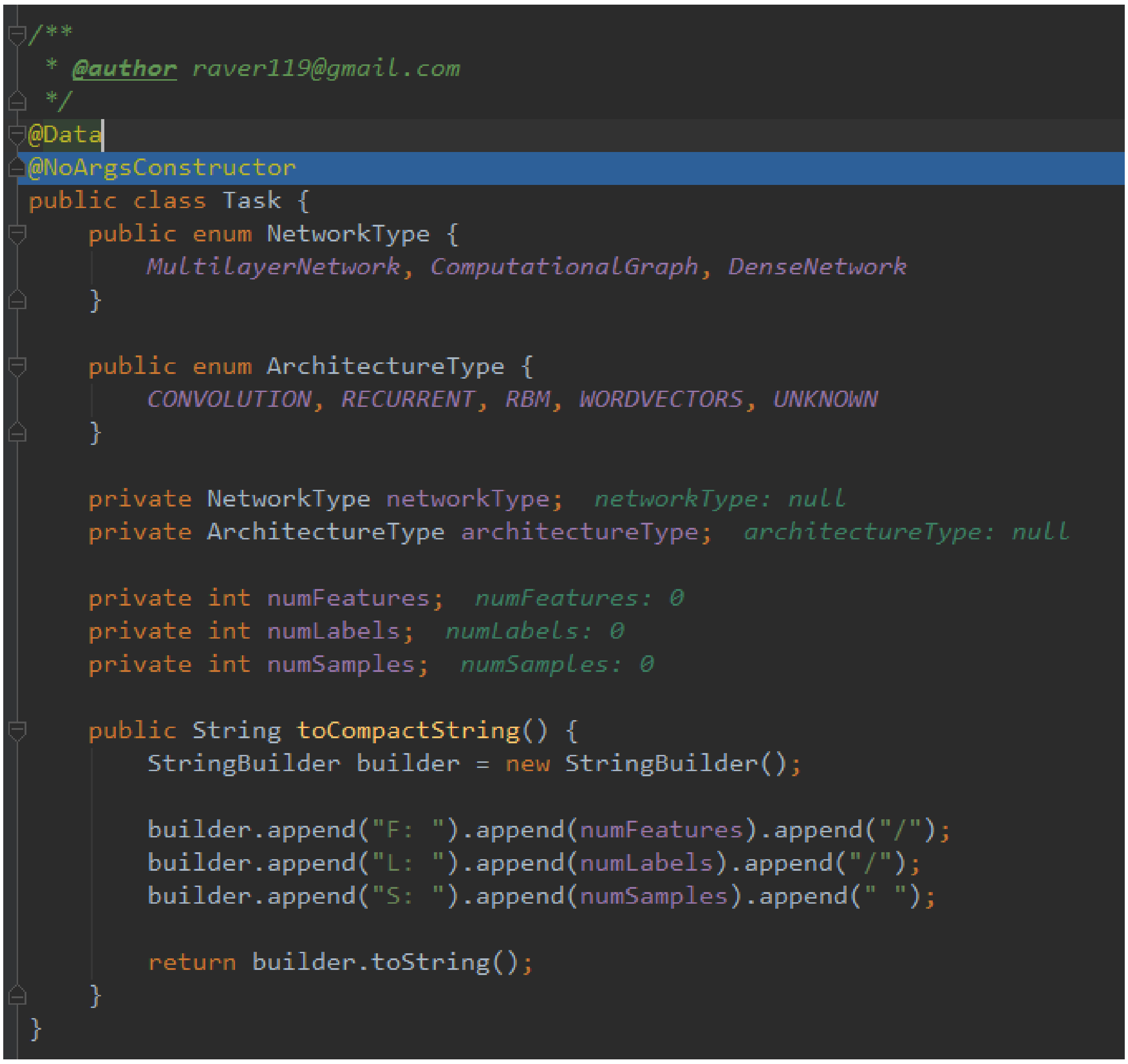
接下来执行

```
update(TaskUtils.buildTask(iter));
```

语句。根据后面源码的阅读，这个task的建立是根据当前的网络模型对训练任务目标的确立。

1. 首先根据传入的iter进行Task的建立。所调用的函数为

```
public static Task buildTask(DataSetIterator dataSetIterator) {
    return new Task();
}
```



```java
/**
 * @author raver119@gmail.com
 */
@Data
@NoArgsConstructor
public class Task {
    public enum NetworkType {
        MultilayerNetwork, ComputationalGraph, DenseNetwork
    }

    public enum ArchitectureType {
        CONVOLUTION, RECURRENT, RBM, WORDVECTORS, UNKNOWN
    }

    private NetworkType networkType;  networkType: null
    private ArchitectureType architectureType;  architectureType: null

    private int numFeatures;  numFeatures: 0
    private int numLabels;  numLabels: 0
    private int numSamples;  numSamples: 0

    public String toCompactString() {
        StringBuilder builder = new StringBuilder();

        builder.append("F: ").append(numFeatures).append("/");
        builder.append("L: ").append(numLabels).append("/");
        builder.append("S: ").append(numSamples).append(" ");

        return builder.toString();
    }
}
```

这里使用`lombok`的两个注解`@Data`、`@NoArgsConstructor`对这个类进行标注
这时候获取的类的样式如
下`Task(networkType=null, architectureType=null, numFeatures=0, numLabels=0, numSamples=0)`

2. 执行update函数

```java
private void update(Task task) {
    if (!initDone) {
        //因为`initDone`初始为false，到此时，`initDone`字段改变，标识网络模型的构造完成。
        initDone = true;
        Heartbeat heartbeat = Heartbeat.getInstance();
        //根据网络模型架构填充task类
        task = ModelSerializer.taskByModel(this);
        Environment env = EnvironmentUtils.buildEnvironment();
        heartbeat.reportEvent(Event.STANDALONE, env, task);
    }
}
```

这里用于展开`ModelSerializer.taskByModel(this);`函数，这个函数主要是根据所传入的`model`的架构类型对`Task`进行字段的填充。

```java
public static Task taskByModel(Model model) {
    Task task = new Task();
    try {
        //先对网络架构设置一个默认值。如当前网络的架构是DenseLayer不满足下列任意一个网络模型，此时就拥有一个默认的网络架构类型。
        task.setArchitectureType(Task.ArchitectureType.RECURRENT);

        //如果传入的model是一个自定义的计算图模型
        if (model instanceof ComputationGraph) {
            //设置网络结构类型
            task.setNetworkType(Task.NetworkType.ComputationalGraph);
            ComputationGraph network = (ComputationGraph) model;
            try {
                //如果网络层数大于0
                if (network.getLayers() != null && network.getLayers().length > 0) {
                    //遍历网络层
                    for (Layer layer : network.getLayers()) {
                        //如果是RBM（受限玻尔兹曼机）
                        if (layer instanceof RBM
                                        || layer instanceof org.deeplearning4j.nn.layers.feedforward.rbm.RBM) {
                            task.setArchitectureType(Task.ArchitectureType.RBM);
                            break;
                        }

                        if (layer.type().equals(Layer.Type.CONVOLUTIONAL)) {
                            //如果是卷积
                            task.setArchitectureType(Task.ArchitectureType.CONVOLUTION);
                            break;
                        } else if (layer.type().equals(Layer.Type.RECURRENT)
                                        || layer.type().equals(Layer.Type.RECURSIVE)) {
                            //如果是循环神经网络
                            task.setArchitectureType(Task.ArchitectureType.RECURRENT);
                            break;
                        }
                    }
                } else
                    task.setArchitectureType(Task.ArchitectureType.UNKNOWN);
            } catch (Exception e) {
                // do nothing here
            }
        } else if (model instanceof MultiLayerNetwork) {
            //如果是多层网络
            task.setNetworkType(Task.NetworkType.MultilayerNetwork);
            MultiLayerNetwork network = (MultiLayerNetwork) model;
            try {
                if (network.getLayers() != null && network.getLayers().length > 0) {
                    for (Layer layer : network.getLayers()) {
                        if (layer instanceof RBM
                                || layer instanceof org.deeplearning4j.nn.layers.feedforward.rbm.RBM) {
                            task.setArchitectureType(Task.ArchitectureType.RBM);
                            break;
                        }
                        if (layer.type().equals(Layer.Type.CONVOLUTIONAL)) {
                            task.setArchitectureType(Task.ArchitectureType.CONVOLUTION);
                            break;
                        } else if (layer.type().equals(Layer.Type.RECURRENT)
                                || layer.type().equals(Layer.Type.RECURSIVE)) {
                            task.setArchitectureType(Task.ArchitectureType.RECURRENT);
                            break;
                        }
                    }
                } else
                    task.setArchitectureType(Task.ArchitectureType.UNKNOWN);
            } catch (Exception e) {
                // do nothing here
            }
        }
        return task;
    } catch (Exception e) {
        task.setArchitectureType(Task.ArchitectureType.UNKNOWN);
        task.setNetworkType(Task.NetworkType.DenseNetwork);
        return task;
    }
}
```

注： `initDone`字段是`MultiLayerNetwork`的一个字段。且初始值为false。

```java
@Setter
protected boolean initDone = false;
```

2.3.2 Solver

```java
/**
 3. Generic purpose solver
 4. @author Adam Gibson
 */
public class Solver {
    private NeuralNetConfiguration conf;
    private Collection<IterationListener> listeners;
    private Model model;
    private ConvexOptimizer optimizer;
    private StepFunction stepFunction;

    public void optimize() {
        if (optimizer == null)
            optimizer = getOptimizer();
        optimizer.optimize();

    }

    public ConvexOptimizer getOptimizer() {
        if (optimizer != null)
            return optimizer;
        switch (conf.getOptimizationAlgo()) {
            case LBFGS:
                optimizer = new LBFGS(conf, stepFunction, listeners, model);
                break;
            case LINE_GRADIENT_DESCENT:
                optimizer = new LineGradientDescent(conf, stepFunction, listeners, model);
                break;
            case CONJUGATE_GRADIENT:
                optimizer = new ConjugateGradient(conf, stepFunction, listeners, model);
                break;
            case STOCHASTIC_GRADIENT_DESCENT:
                optimizer = new StochasticGradientDescent(conf, stepFunction, listeners, model);
                break;
            default:
                throw new IllegalStateException("No optimizer found");
        }
        return optimizer;
    }

    public void setListeners(Collection<IterationListener> listeners) {
        this.listeners = listeners;
        if (optimizer != null)
            optimizer.setListeners(listeners);
    }

    public static class Builder {
        private NeuralNetConfiguration conf;
        private Model model;
        private List<IterationListener> listeners = new ArrayList<>();

        public Builder configure(NeuralNetConfiguration conf) {
            this.conf = conf;
            return this;
        }

        public Builder listener(IterationListener... listeners) {
            this.listeners.addAll(Arrays.asList(listeners));
            return this;
        }

        public Builder listeners(Collection<IterationListener> listeners) {
            this.listeners.addAll(listeners);
            return this;
        }

        public Builder model(Model model) {
            this.model = model;
            return this;
        }

        public Solver build() {
            Solver solver = new Solver();
            solver.conf = conf;
            solver.stepFunction = StepFunctions.createStepFunction(conf.getStepFunction());
            solver.model = model;
            solver.listeners = listeners;
            return solver;
        }
    }
}
```

以上是对`Solver`这个类的源码，接下来查看源码执行部分

```
solver = new Solver.Builder().configure(conf()).listeners(getListeners()).model(this).build();
```

1. 首先调用`configure()`、`listeners()`、`model()`等方法获取`MultiLayerNetwork`类的配置，然后再调用`build()`方法根据各种配置实例化对象
2. 除上述之外，主要观察`stepFunction`这个属性的配置。这里单步因为第一次调用的时候`conf.getStepFunction()`为null， 所以`stepFunction`也为null。
3. 之后就要执行`solver.optimize()`方法。

2.3.3 solver.optimize()

optimezie()方法首先需要判断solver类中的`optimizer`字段是否为空。

```
public void optimize() {
    if (optimizer == null)
        optimizer = getOptimizer();
    optimizer.optimize();
}
```

如果为空则需要调用`getOptimizer()`方法获取实例。

```java
public ConvexOptimizer getOptimizer() {
    if (optimizer != null)
        return optimizer;
    switch (conf.getOptimizationAlgo()) {
        case LBFGS:
            optimizer = new LBFGS(conf, stepFunction, listeners, model);
            break;
        case LINE_GRADIENT_DESCENT:
            optimizer = new LineGradientDescent(conf, stepFunction, listeners, model);
            break;
        case CONJUGATE_GRADIENT:
            optimizer = new ConjugateGradient(conf, stepFunction, listeners, model);
            break;
        case STOCHASTIC_GRADIENT_DESCENT:
            optimizer = new StochasticGradientDescent(conf, stepFunction, listeners, model);
            break;
        default:
            throw new IllegalStateException("No optimizer found");
    }
    return optimizer;
}
```

我们这里使用的优化算法是`STOCHASTIC_GRADIENT_DESCENT`，`StochasticGradientDescent`这个类继承自`BaseOptimizer`。
构造方法的实例

```java
public StochasticGradientDescent(NeuralNetConfiguration conf, StepFunction stepFunction,
                Collection<IterationListener> iterationListeners, Model model) {
    super(conf, stepFunction, iterationListeners, model);
}
```

此时在单步到`BaseOptimizer`的构造方法中

```java
public BaseOptimizer(NeuralNetConfiguration conf, StepFunction stepFunction,
                Collection<IterationListener> iterationListeners, Model model) {
    this(conf, stepFunction, iterationListeners, Arrays.asList(new ZeroDirection(), new EpsTermination()), model);
}
```

其中Arrays里面生成的两个类均是继承`TerminationCondition`，具体如下：

1. ZeroDirection

```java
/**
 * Absolute magnitude of gradient is 0
 * @author Adam Gibson
 */
public class ZeroDirection implements TerminationCondition {
    @Override
    public boolean terminate(double cost, double oldCost, Object[] otherParams) {
        INDArray gradient = (INDArray) otherParams[0];
        return Nd4j.getBlasWrapper().level1().asum(gradient) == 0.0;
    }
}
```

代码中注释的意思是：绝对的梯度幅度是0。

2. EpsTermination

```java
/**
 * Epsilon termination (absolute change based on tolerance)
 *
 * @author Adam Gibson
 */
public class EpsTermination implements TerminationCondition {
    private double eps = 1e-4;
    private double tolerance = Nd4j.EPS_THRESHOLD;

    public EpsTermination(double eps, double tolerance) {
        this.eps = eps;
        this.tolerance = tolerance;
    }

    public EpsTermination() {}

    @Override
    public boolean terminate(double cost, double old, Object[] otherParams) {
        //special case for initial termination, ignore
        if (cost == 0 && old == 0)
            return false;

        if (otherParams.length >= 2) {
            double eps = (double) otherParams[0];
            double tolerance = (double) otherParams[1];
            return 2.0 * Math.abs(old - cost) <= tolerance * (Math.abs(old) + Math.a
bs(cost) + eps);
        }

        else
            return 2.0 * Math.abs(old - cost) <= tolerance * (Math.abs(old) + Math.a
bs(cost) + eps);
    }
}
```

代码中的注释意思是：Epsilon终止（基于容差的绝对变化）
注： 这两个类的用处从字面上的意思是用于辅助使用者判断网络模型训练的终止条件。

在以上两个类创建完成之后会继续调用`BaseOptimizer`的构造函数继续对优化器进行实例化：

```java
public BaseOptimizer(NeuralNetConfiguration conf, StepFunction stepFunction,
                Collection<IterationListener> iterationListeners,
                Collection<TerminationCondition> terminationConditions, Model model
) {
    this.conf = conf;
    this.stepFunction = (stepFunction != null ? stepFunction :
getDefaultStepFunctionForOptimizer(this.getClass()));
    this.iterationListeners = iterationListeners != null ? iterationListeners : new
ArrayList<IterationListener>();
    this.terminationConditions = terminationConditions;
    this.model = model;
    //构造线性搜索器，属于凸优化数学概念。。暂时不明
    lineMaximizer = new BackTrackLineSearch(model, this.stepFunction, this);
    //设置最大Step，默认值为 stepMax = Double.MAX_VALUE;
    lineMaximizer.setStepMax(stepMax);
    //线性优化器的迭代次数
    lineMaximizer.setMaxIterations(conf.getMaxNumLineSearchIterations());

}
```

在这个类里面会调用`getDefaultStepFunctionForOptimizer`对`stepFunction`进行实例化。

```java
public static StepFunction getDefaultStepFunctionForOptimizer(Class<? extends Conve
xOptimizer> optimizerClass) {
    if (optimizerClass == StochasticGradientDescent.class) {
        //Subtract the line
        return new NegativeGradientStepFunction();
    } else {
        //Inverse step function 翻转
        return new NegativeDefaultStepFunction();
    }
}
```

在该函数运行结束完毕之后，则会继续调用`optimizer.optimize();`。

```java
@Override
public boolean optimize() {
    for (int i = 0; i < conf.getNumIterations(); i++) {

        Pair<Gradient, Double> pair = gradientAndScore();
        Gradient gradient = pair.getFirst();

        INDArray params = model.params();
        stepFunction.step(params, gradient.gradient());
        //Note: model.params() is always in-place for MultiLayerNetwork and ComputationGraph, hence no setParams is necessary there
        //However: for pretrain layers, params are NOT a view. Thus a setParams call is necessary
        //But setParams should be a no-op for MLN and CG
        model.setParams(params);

        int iterationCount = BaseOptimizer.getIterationCount(model);
        for (IterationListener listener : iterationListeners)
            listener.iterationDone(model, iterationCount);

        checkTerminalConditions(pair.getFirst().gradient(), oldScore, score, i);

        BaseOptimizer.incrementIterationCount(model, 1);
    }
    return true;
}
```

该段函数应该是根据设置的优化算法的迭代次数对网络模型的梯度计算以及网络模型参数的更新。

> **这里是对改段源码中注释的翻译以及解释：**
> model.params()所返回的INDArray类型因为内存共享的机制，在参与运算之后就会被就地替换，因此setParams()操作在这里并不是必须的
> 但是，对于pretrain预训练网络层，params并不是视图，因此参与运算之后不会被就地替换，因此对于预训练网络层setParams()是必须的
> 但是这里需要再次提醒的是setParams()操作对于MultiLayerNetwork和ComputationGraph是非必须操作

2.3.3.1 gradientAndScore();

这里用于获取梯度和分数

```java
@Override
public Pair<Gradient, Double> gradientAndScore() {
    oldScore = score;
    model.computeGradientAndScore();

    if (iterationListeners != null && iterationListeners.size() > 0) {
        for (IterationListener l : iterationListeners) {
            if (l instanceof TrainingListener) {
                ((TrainingListener) l).onGradientCalculation(model);
            }
        }
    }

    Pair<Gradient, Double> pair = model.gradientAndScore();
    score = pair.getSecond();
    updateGradientAccordingToParams(pair.getFirst(), model, model.batchSize());
    return pair;
}
```

2.3.3.2 model.computeGradientAndScore()

```java
@Override
public void computeGradientAndScore() {
    //Calculate activations (which are stored in each layer, and used in backprop)
    if (layerWiseConfigurations.getBackpropType() == BackpropType.TruncatedBPTT) {
        List<INDArray> activations = rnnActivateUsingStoredState(getInput(), true, true);
        if (trainingListeners.size() > 0) {
            for (TrainingListener tl : trainingListeners) {
                tl.onForwardPass(this, activations);
            }
        }
        truncatedBPTTGradient();
    } else {
        //First: do a feed-forward through the network
        //Note that we don't actually need to do the full forward pass through the output layer right now; but we do
        // need the input to the output layer to be set (such that backprop can be done)
        List<INDArray> activations = feedForwardToLayer(layers.length - 2, true);
        if (trainingListeners.size() > 0) {
            //TODO: We possibly do want output layer activations in some cases here...
            for (TrainingListener tl : trainingListeners) {
                tl.onForwardPass(this, activations);
            }
        }
        INDArray actSecondLastLayer = activations.get(activations.size() - 1);
        if (layerWiseConfigurations.getInputPreProcess(layers.length - 1) != null)
            actSecondLastLayer = layerWiseConfigurations.getInputPreProcess(layers.length - 1)
                                .preProcess(actSecondLastLayer, getInputMiniBatchSize());
        getOutputLayer().setInput(actSecondLastLayer);
        //Then: compute gradients
        backprop();
    }

    //Calculate score
    if (!(getOutputLayer() instanceof IOutputLayer)) {
        throw new IllegalStateException(
                        "Cannot calculate gradient and score with respect to labels: final layer is not an IOutputLayer");
    }
    score = ((IOutputLayer) getOutputLayer()).computeScore(calcL1(true), calcL2(true), true);

    //Listeners
    if (trainingListeners.size() > 0) {
        for (TrainingListener tl : trainingListeners) {
            tl.onBackwardPass(this);
        }
    }
}
```

在dl4j中，除非在网络模型建立的过程中，通过`.backpropType(BackpropType.TruncatedBPTT)`方法来改变模型的反向传播方式，那么默认的反向传播方式一定是`BackpropType.Standard`（包括RNN、LSTM）。在确定本次的反向传播方式为`BackpropType.Standard`之后，需要执行如下的语句。

```java
//First: 首先对网络做一个前向传播
//Note：现在我们并不需要作完整的前向传播到输出层
//但是我们确实需要算出给输出层的输入（这样Backprop就可以完成）
List<INDArray> activations = feedForwardToLayer(layers.length - 2, true);
```

接下来进入这个函数体内：

```java
 /** Compute the activations from the input to the specified layer, using the currently set input for the network.<br>
 * To compute activations for all layers, use feedForward(...) methods<br>
 * Note: output list includes the original input. So list.get(0) is always the original input, and
 * list.get(i+1) is the activations of the ith layer.
 * @param layerNum Index of the last layer to calculate activations for. Layers are zero-indexed.
 *                    feedForwardToLayer(i,input) will return the activations for layers 0..i (inclusive)
 * @param train true for training, false for test (i.e., false if using network after training)
 * @return list of activations.
 */
public List<INDArray> feedForwardToLayer(int layerNum, boolean train) {
    INDArray currInput = input;
    List<INDArray> activations = new ArrayList<>();
    activations.add(currInput);

    for (int i = 0; i <= layerNum; i++) {
        currInput = activationFromPrevLayer(i, currInput, train);
        //applies drop connect to the activation
        activations.add(currInput);
    }
    return activations;
}
```

这个函数是计算出所有隐藏层的输出（除去输入层和输出层），并且组成一个INDArray的List，并且包含原始的输入。之后我们单步进入`activationFromPrevLayer(i, currInput, train);`函数，查看神经网络的前向传播计算过程。

```
/**
 * Calculate activation from previous layer including pre processing where necessary
 *
 * @param curr  the current layer
 * @param input the input
 * @return the activation from the previous layer
 */
public INDArray activationFromPrevLayer(int curr, INDArray input, boolean training) {
    if (getLayerWiseConfigurations().getInputPreProcess(curr) != null)
        input = getLayerWiseConfigurations().getInputPreProcess(curr).preProcess(input, getInputMiniBatchSize());
    INDArray ret = layers[curr].activate(input, training);
    return ret;
}
```

使用前一层的输出作为当前层的输入，如果有数据预处理则先进行预处理。然后调用当前层的`activate()`方法计算结果。该方法的调用链条如下：

```
@Override
public INDArray activate(INDArray input, boolean training) {
    setInput(input);
    return activate(training);
}

@Override
public INDArray activate(boolean training) {
    INDArray z = preOutput(training);
    //INDArray ret = Nd4j.getExecutioner().execAndReturn(Nd4j.getOpFactory().createTransform(
    //        conf.getLayer().getActivationFunction(), z, conf.getExtraArgs() ));
    INDArray ret = conf().getLayer().getActivationFn().getActivation(z, training);

    if (maskArray != null) {
        ret.muliColumnVector(maskArray);
    }

    return ret;
}
```

preOut这一部分就是网络模型前向传播的重点。

```java
public INDArray preOutput(boolean training) {
    applyDropOutIfNecessary(training);
    INDArray b = getParam(DefaultParamInitializer.BIAS_KEY);
    INDArray W = getParam(DefaultParamInitializer.WEIGHT_KEY);

    //Input validation:
    if (input.rank() != 2 || input.columns() != W.rows()) {
        if (input.rank() != 2) {
            throw new DL4JInvalidInputException("Input that is not a matrix; expected matrix (rank 2), got rank "
                            + input.rank() + " array with shape " + Arrays.toString(input.shape()));
        }
        throw new DL4JInvalidInputException("Input size (" + input.columns() + " columns; shape = "
                        + Arrays.toString(input.shape())
                        + ") is invalid: does not match layer input size (layer # inputs = " + W.size(0) + ")");
    }

    if (conf.isUseDropConnect() && training && conf.getLayer().getDropOut() > 0) {
        W = Dropout.applyDropConnect(this, DefaultParamInitializer.WEIGHT_KEY);
    }

    INDArray ret = input.mmul(W).addiRowVector(b);

    if (maskArray != null) {
        applyMask(ret);
    }

    return ret;
}
```

首先使用`applyDropOutIfNecessary(training);`函数判断当前是否使用dropout。

```java
protected void applyDropOutIfNecessary(boolean training) {
    if (conf.getLayer().getDropOut() > 0 && !conf.isUseDropConnect() && training && !dropoutApplied) {
        input = input.dup();
        Dropout.applyDropout(input, conf.getLayer().getDropOut());
        dropoutApplied = true;
    }
}
```

使用dropout的条件如下：

1. 当前层设置 dropout > 0

2. 当前配置没有使用dropConnect(), 这一配置在卷积神经网络常见。
3. 当前是训练过程，也就是training的值为true。 在预测的时候dropout不会被应用
4. dropout在之前没有被调用。

如果以上条件都满足，则先对当前的输入使用`dup()`函数进行复制（注：dup取自单词duplicate，复制的意思），然后传入下一个函数。

```
/**
 5. Apply dropout to the given input
 6. and return the drop out mask used
 7. @param input the input to do drop out on
 8. @param dropout the drop out probability
 */
public static void applyDropout(INDArray input, double dropout) {
    if (Nd4j.getRandom().getStatePointer() != null) {
        Nd4j.getExecutioner().exec(new DropOutInverted(input, dropout));
    } else {
        Nd4j.getExecutioner().exec(new LegacyDropOutInverted(input, dropout));
    }
}
```

dropout的实现方式很多，根据这个源码阅读方式发现，dl4j的dropout实现方式是根据截断当前层的输入来实现drpout。

```
/**
 9. This method returns pointer to RNG state structure.
 10. Please note: DefaultRandom implementation returns NULL here, making it
impossible to use with RandomOps
 11.  - @return
 */
@Override
public Pointer getStatePointer() {
    return statePointer;
}
```

这个getStatePointer()的目的从代码的注释情况上来还不是很清楚。接下来查看两种实现方式

1. DropOutInverted

```
/**
 * Inverted DropOut implementation as Op
 *
 * @author raver119@gmail.com
 */
public class DropOutInverted extends BaseRandomOp {

    private double p;
```

```java

    public DropOutInverted() {

    }

    public DropOutInverted(@NonNull INDArray x, double p) {
        this(x, x, p, x.lengthLong());
    }

    public DropOutInverted(@NonNull INDArray x, @NonNull INDArray z, double p) {
        this(x, z, p, x.lengthLong());
    }

    public DropOutInverted(@NonNull INDArray x, @NonNull INDArray z, double p, long
n) {
        this.p = p;
        init(x, null, z, n);
    }

    @Override
    public int opNum() {
        return 2;
    }

    @Override
    public String name() {
        return "dropout_inverted";
    }

    @Override
    public void init(INDArray x, INDArray y, INDArray z, long n) {
        super.init(x, y, z, n);
        this.extraArgs = new Object[] {p};
    }
}
```

1. LegacyDropOutInverted

```java
/**
 * Inverted DropOut implementation as Op
 *
 * PLEASE NOTE: This is legacy DropOutInverted implementation, please consider
using op with the same name from randomOps
 * @author raver119@gmail.com
 */
public class LegacyDropOutInverted extends BaseTransformOp {

    private double p;

    public LegacyDropOutInverted() {

    }

    public LegacyDropOutInverted(INDArray x, double p) {
        super(x);
        this.p = p;
        init(x, null, x, x.length());
    }

    public LegacyDropOutInverted(INDArray x, INDArray z, double p) {
        super(x, z);
        this.p = p;
        init(x, null, z, x.length());
    }

    public LegacyDropOutInverted(INDArray x, INDArray z, double p, long n) {
        super(x, z, n);
        this.p = p;
        init(x, null, z, n);
    }

    @Override
    public int opNum() {
        return 44;
    }

    @Override
    public String name() {
        return "legacy_dropout_inverted";
    }

    @Override
    public IComplexNumber op(IComplexNumber origin, double other) {
        return null;
    }

    @Override
    public IComplexNumber op(IComplexNumber origin, float other) {
        return null;
    }

    @Override
    public IComplexNumber op(IComplexNumber origin, IComplexNumber other) {
        return null;
    }

    @Override
    public float op(float origin, float other) {
        return 0;
    }

    @Override
    public double op(double origin, double other) {
        return 0;
    }

    @Override
    public double op(double origin) {
        return 0;
    }

    @Override
    public float op(float origin) {
        return 0;
    }

    @Override
    public IComplexNumber op(IComplexNumber origin) {
        return null;

    }

    @Override
    public Op opForDimension(int index, int dimension) {
        INDArray xAlongDimension = x.vectorAlongDimension(index, dimension);

        if (y() != null)
            return new LegacyDropOutInverted(xAlongDimension, z.vectorAlongDimension(index, dimension), p,
                            xAlongDimension.length());
        else
            return new LegacyDropOutInverted(xAlongDimension, z.vectorAlongDimension(index, dimension), p,
                            xAlongDimension.length());
    }

    @Override
    public Op opForDimension(int index, int... dimension) {
        INDArray xAlongDimension = x.tensorAlongDimension(index, dimension);

        if (y() != null)
            return new LegacyDropOutInverted(xAlongDimension, z.tensorAlongDimension(index, dimension), p,
                            xAlongDimension.length());
        else
            return new LegacyDropOutInverted(xAlongDimension, z.tensorAlongDimension(index, dimension), p,
                            xAlongDimension.length());

    }

    @Override
    public void init(INDArray x, INDArray y, INDArray z, long n) {
        super.init(x, y, z, n);
        this.extraArgs = new Object[] {p, (double) n};
        }
    }
```

这个dropout有些难以理解，这里用单步的调试信息来查看计算流程来尝试理解：
当前程序运行的dropout的类型为`DropOutInverted`。此时调用的函数如下：

```
public DropOutInverted(@NonNull INDArray x, double p) {
    this(x, x, p, x.lengthLong());
}
```

当前输入的x的值为：
[-10.0,-9.99,-9.98,-9.97,-9.96,-9.95,-9.94,-9.93,-9.92,-9.91,-9.9,-9.89,-9.88,-9.87,-9.86,-9.85,-9.84,-9.83,-9.82,-9.81]
它的shape为[20, 1]，也就是一个 20 x 1的列向量。其中调用的`x.lengthLong()`的值也为20。当前的p值也有改变，p值变为当前层的dropout值，即当前`p = 0.5`。之后调用this运行到另外一个构造函数中：

```
public DropOutInverted(@NonNull INDArray x, @NonNull INDArray z, double p, long n) {
    this.p = p;
    init(x, null, z, n);
}
```

在调用到当前构造函数的时候，调用init函数，此时的 z和x是相同的值。

```
@Override
public void init(INDArray x, INDArray y, INDArray z, long n) {
    super.init(x, y, z, n);
    this.extraArgs = new Object[] {p};
}
```

执行到当前步，各项参数如下：

```
x = [-10.00, -9.99, -9.98, -9.97, -9.96, -9.95, -9.94, -9.93, -9.92, -9.91, -9.90, -9.89, -9.88, -9.87, -9.86, -9.85, -9.84, -9.83, -9.82, -9.81]
y = null
z = [-10.00, -9.99, -9.98, -9.97, -9.96, -9.95, -9.94, -9.93, -9.92, -9.91, -9.90, -9.89, -9.88, -9.87, -9.86, -9.85, -9.84, -9.83, -9.82, -9.81]
n = 20
p = 0.5
```

之后就会跳转到父类的init()方法：

```
@Override
```

```java
public void init(INDArray x, INDArray y, INDArray z, long n) {
    this.x = x;
    this.y = y;
    this.z = z;
    this.n = n;
}
```

父类方法只是对成员变量进行简单赋值。
在以上变量初始化完成之后，继续执行`Nd4j.getExecutioner().exec(new DropOutInverted(input, dropout));`方法。

```java
/**
 * This method executes specified RandomOp using default RNG available via Nd4j.getRandom()
 *
 * @param op
 */
@Override
public INDArray exec(RandomOp op) {
    return exec(op, Nd4j.getRandom());
}
```

根据注释，两个dropout类是特殊的RandomOp。之后继续调用下一个`exec()`方法。

```java
/**
 * This method executes specific
 * RandomOp against specified RNG
 *
 * @param op
 * @param rng
 */
@Override
public INDArray exec(RandomOp op, Random rng) {
    if (rng.getStateBuffer() == null)
        throw new IllegalStateException(
                                "You should use one of NativeRandom classes for NativeOperations execution");

    long st = profilingHookIn(op);

    validateDataType(Nd4j.dataType(), op);

    if (op.x() != null && op.y() != null && op.z() != null) {
        // triple arg call
        if (Nd4j.dataType() == DataBuffer.Type.FLOAT) {
            loop.execRandomFloat(null, op.opNum(), rng.getStatePointer(), // rng state ptr
                            (FloatPointer) op.x().data().addressPointer(),
                            (IntPointer) op.x().shapeInfoDataBuffer().addressPointer(),
                            (FloatPointer) op.y().data().addressPointer(),
                            (IntPointer) op.y().shapeInfoDataBuffer().addressPointer(),
                            (FloatPointer) op.z().data().addressPointer(),
                            (IntPointer) op.z().shapeInfoDataBuffer().addressPointer(),
                            (FloatPointer) op.extraArgsDataBuff().addressPointer());
        } else if (Nd4j.dataType() == DataBuffer.Type.DOUBLE) {
            loop.execRandomDouble(null, op.opNum(), rng.getStatePointer(), // rng state ptr
                            (DoublePointer) op.x().data().addressPointer(),
                            (IntPointer) op.x().shapeInfoDataBuffer().addressPointer(),
                            (DoublePointer) op.y().data().addressPointer(),
                            (IntPointer) op.y().shapeInfoDataBuffer().addressPointer(),
                            (DoublePointer) op.z().data().addressPointer(),
                            (IntPointer) op.z().shapeInfoDataBuffer().addressPointer(),
                            (DoublePointer) op.extraArgsDataBuff().addressPointer());
        }
    } else if (op.x() != null && op.z() != null) {
        //double arg call
        if (Nd4j.dataType() == DataBuffer.Type.FLOAT) {
            loop.execRandomFloat(null, op.opNum(), rng.getStatePointer(), // rng state ptr
                            (FloatPointer) op.x().data().addressPointer(),
                            (IntPointer) op.x().shapeInfoDataBuffer().addressPointer(),
                            (FloatPointer) op.z().data().addressPointer(),
                            (IntPointer) op.z().shapeInfoDataBuffer().addressPointer(),
                            (FloatPointer) op.extraArgsDataBuff().addressPointer());
        } else if (Nd4j.dataType() == DataBuffer.Type.DOUBLE) {
            loop.execRandomDouble(null, op.opNum(), rng.getStatePointer(), // rng state ptr
                            (DoublePointer) op.x().data().addressPointer(),
                            (IntPointer) op.x().shapeInfoDataBuffer().addressPointer(),
                            (DoublePointer) op.z().data().addressPointer(),
                            (IntPointer) op.z().shapeInfoDataBuffer().addressPointer(),
                            (DoublePointer) op.extraArgsDataBuff().addressPointer());
        }

    } else {
        // single arg call

        if (Nd4j.dataType() == DataBuffer.Type.FLOAT) {
            loop.execRandomFloat(null, op.opNum(), rng.getStatePointer(), // rng state ptr
                                (FloatPointer) op.z().data().addressPointer(),
                                (IntPointer) op.z().shapeInfoDataBuffer().addressPointer(),
                                (FloatPointer) op.extraArgsDataBuff().addressPointer());
        } else if (Nd4j.dataType() == DataBuffer.Type.DOUBLE) {
            loop.execRandomDouble(null, op.opNum(), rng.getStatePointer(), // rng state ptr
                                (DoublePointer) op.z().data().addressPointer(),
                                (IntPointer) op.z().shapeInfoDataBuffer().addressPointer(),
                                (DoublePointer) op.extraArgsDataBuff().addressPointer());
        }
    }

    profilingHookOut(op, st);

    return op.z();
}
```

这个函数首先使用`validateDataType(Nd4j.dataType(), op);`用于检验当前数据类型的合法性。然后根据传入的op的三个成员变量x, y, z来判断进入哪一分支。在上面的debug信息我们可以看到，我们的x和z是两个非空变量，因此进入第二个分支，并且我们当前的`Nd4j.dataType()`为`DataBuffer.Type.FLOAT`。为此在当前环境下会执行以下语句：

```
loop.execRandomFloat(null, op.opNum(), rng.getStatePointer(), // rng state ptr
                                (FloatPointer) op.x().data().addressPointer(),
                                (IntPointer) op.x().shapeInfoDataBuffer().addressPointer(),
                                (FloatPointer) op.z().data().addressPointer(),
                                (IntPointer) op.z().shapeInfoDataBuffer().addressPointer(),
                                (FloatPointer) op.extraArgsDataBuff().addressPointer());
```

然后这部分的具体实现应该是JNI调用的底层

```
public native void execRandomFloat(@Cast("Nd4jPointer*") PointerPointer extraPointers, int opNum, @Cast("Nd4jPointer") Pointer state, FloatPointer x, IntPointer xShapeBuffer, FloatPointer z, IntPointer zShapeBuffer, FloatPointer extraArguments);
```

经过如上方法的的运行之后，返回z值，这时候通过debug信息看到的z值为：

```
[-20.00, -19.98, -19.96, 0.00, 0.00, -19.90, -19.88, 0.00, -19.84, -19.82, 0.00, 0.00,
-19.76, -19.74, -19.72, -19.70, -19.68, -19.66, -19.64, 0.00]
```

因为在前面输入的时候z其实和x是等同的。在执行以上方法之后，相当于对x做了一个变幻。使得x变为如上的数值（这里猜测实现的方式是部分位置随机置0，然后再所有的数据除以dropout的值）。到这里就使得dl4j的`applyDropOutIfNecessary(training)`方法部分完成，继续回到`preOutput()`方法体内继续执行。接下来执行的就是`preOutput()`如下的两条语句：

```java
INDArray b = getParam(DefaultParamInitializer.BIAS_KEY);
INDArray W = getParam(DefaultParamInitializer.WEIGHT_KEY);
```

用于获取当前层的权重和偏值。之后继续执行的是输入的有效性判断以及是否使用dropoutConnect，当前的网络架构没有使用该种网络，暂时不谈。

```java
 //Input validation:
if (input.rank() != 2 || input.columns() != W.rows()) {
    if (input.rank() != 2) {
        throw new DL4JInvalidInputException("Input that is not a matrix; expected matrix (rank 2), got rank "
                        + input.rank() + " array with shape " + Arrays.toString(input.shape()));
    }
    throw new DL4JInvalidInputException("Input size (" + input.columns() + " columns; shape = "
                    + Arrays.toString(input.shape())
                    + ") is invalid: does not match layer input size (layer # inputs = " + W.size(0) + ")");
}

if (conf.isUseDropConnect() && training && conf.getLayer().getDropOut() > 0) {
    W = Dropout.applyDropConnect(this, DefaultParamInitializer.WEIGHT_KEY);
}
```

再之后执行的就是神经元中最经典且常见的数学公式`y = xw + b`:

```java
INDArray ret = input.mmul(W).addiRowVector(b);
```

然后就是判断掩码，如果使用掩码，则对计算之后的结果ret进行一个变换。

```java
if (maskArray != null) {
    applyMask(ret);
}

//掩码对结果变换的实现方式。
protected void applyMask(INDArray to) {
    to.muliColumnVector(maskArray);
}
```

然后此时`preOutput()`方法执行结束，返回上层方法`activate(boolean training)`中：

```java
@Override
public INDArray activate(boolean training) {
    INDArray z = preOutput(training);
    //INDArray ret =
Nd4j.getExecutioner().execAndReturn(Nd4j.getOpFactory().createTransform(
    //          conf.getLayer().getActivationFunction(), z, conf.getExtraArgs() ));
    INDArray ret = conf().getLayer().getActivationFn().getActivation(z, training);

    if (maskArray != null) {
        ret.muliColumnVector(maskArray);
    }

    return ret;
}
```

在`preOutput()`方法中执行了`z = xw + b`，接下来就需要使用如下的方法进行激励函数的变换：

```java
INDArray ret = conf().getLayer().getActivationFn().getActivation(z, training);
```

等同于`ret = f(z)`。运行之后继续返回上层`activationFromPrevLayer()`-> `feedForwardToLayer()`方法中。

```java
public List<INDArray> feedForwardToLayer(int layerNum, boolean train) {
    INDArray currInput = input;
    List<INDArray> activations = new ArrayList<>();
    activations.add(currInput);

    for (int i = 0; i <= layerNum; i++) {
        currInput = activationFromPrevLayer(i, currInput, train);
        //applies drop connect to the activation
        activations.add(currInput);
    }
    return activations;
}
```

以上单步的只是一层网络的运行方法，使用for loop不断重复以上流程，并且将中间结果添加到activations中，并进行返回。

2.3.3.2 model.computeGradientAndScore()

这里重新粘贴正在执行的代码部分：

```
List<INDArray> activations = feedForwardToLayer(layers.length - 2, true);
if (trainingListeners.size() > 0) {
    //TODO: We possibly do want output layer activations in some cases here...
    for (TrainingListener tl : trainingListeners) {
        tl.onForwardPass(this, activations);
    }
}
INDArray actSecondLastLayer = activations.get(activations.size() - 1);
if (layerWiseConfigurations.getInputPreProcess(layers.length - 1) != null)
    actSecondLastLayer = layerWiseConfigurations.getInputPreProcess(layers.length -
1)
                    .preProcess(actSecondLastLayer, getInputMiniBatchSize());
getOutputLayer().setInput(actSecondLastLayer);
//Then: compute gradients
backprop();
```

此时相当于重新返回到`model.computeGradientAndScore()`方法继续向下执行。

```
List<INDArray> activations = feedForwardToLayer(layers.length - 2, true);
```

在这条语句获取到输出层之前的所有层的中间结果之后，执行以下语句：

```
INDArray actSecondLastLayer = activations.get(activations.size() - 1);
```

此时`actSecondLastLayer`相当于获取最后一层输出层的输入。
之后调用一下的语句：

```
getOutputLayer().setInput(actSecondLastLayer);
```

设置输出层的输入，然后就进入反向传播环节计算梯度：

```
backprop();
```

2.3.3.3 backprop()

```
/** Calculate and set gradients for MultiLayerNetwork, based on OutputLayer and la
bels*/
protected void backprop() {
    Pair<Gradient, INDArray> pair = calcBackpropGradients(null, true);
    this.gradient = (pair == null ? null : pair.getFirst());
    this.epsilon = (pair == null ? null : pair.getSecond());
}
```