前面经过反向传播，已经计算出了模型的损失函数得分以及梯度，在反向传播完成之后会返回到`package org.deeplearning4j.optimize.solvers`包下的`BaseOptimizer.gradientAndScore()`方法体重继续执行，该方法体中继续执行。

反向传播计算完的参数还需要经过梯度正则化以及L1，L2参数惩罚

```java
@Override
public Pair<Gradient, Double> gradientAndScore() {
    oldScore = score;
    //包含反向传播，已经算出了模型的损失函数得分以及梯度
    model.computeGradientAndScore();

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

    //获取模型中的梯度和损失函数得分
    Pair<Gradient, Double> pair = model.gradientAndScore();
    //将模型的损失函数得分赋值为优化器的成员变量中
    score = pair.getSecond();
    //然后根据参数更新梯度
    updateGradientAccordingToParams(pair.getFirst(), model,
model.batchSize());
    return pair;
}
```

1 updateGradientAccordingToParams

```java
@Override
public void updateGradientAccordingToParams(Gradient gradient, Model m
odel, int batchSize) {
    //首先判断是ComputationGraph还是MultiLayerNetwork
    if (model instanceof ComputationGraph) {
        ComputationGraph graph = (ComputationGraph) model;
        if (computationGraphUpdater == null) {
            computationGraphUpdater = new ComputationGraphUpdater(graph
);
        }
        computationGraphUpdater.update(graph, gradient, getIterationCount(model), batchSize);
    } else {

        //获取更新器
        if (updater == null)
            updater = UpdaterCreator.getUpdater(model);

        //将model改为Layer类型，这个时候需要注意，在多层网络架构的时候
        //MultiLayerNetwork 可以认为是输出层
        //MultiLayerNetwork is a neural network with multiple layers in a stack, and usually an output layer.
        Layer layer = (Layer) model;
        updater.update(layer, gradient, getIterationCount(model), batchSize);
    }
}
```

1.1 UpdaterCreator.getUpdater(model)

首先需要根据模型设置来获取模型参数的更新器

```java
public class UpdaterCreator {

    private UpdaterCreator() {}

    public static org.deeplearning4j.nn.api.Updater getUpdater(Model layer) {
        //判断网络架构
        if (layer instanceof MultiLayerNetwork) {
            return new MultiLayerUpdater((MultiLayerNetwork) layer);
        } else {
            return new LayerUpdater();
        }
    }

}
```

之后构造一个新的更新器类`MultiLayerUpdater`。

在`package org.deeplearning4j.nn.updater;`包下，所调用的更新器的构造函数为：

```java
/**
 * MultiLayerUpdater: Gradient updater for MultiLayerNetworks.
 * Expects backprop gradients for all layers to be in single Gradient object,
 * keyed by "0_b", "1_w" etc., as per MultiLayerNetwork.backward()
 */
public MultiLayerUpdater(MultiLayerNetwork network) {
    //获取架构的网络层
    Layer[] layers = network.getLayers();
    //逐层判断是否为空
    for (int i = 0; i < layers.length; i++) {
        //守护条件，保证获取到的layer全都不为null
        while (layers[i] == null)
            layers = network.getLayers();
    }
    //根据网络层个数构造网络层更新器
    layerUpdaters = new Updater[layers.length];
    //更新器状态个数
    int updaterStateSize = 0;
    for (int i = 0; i < layers.length; i++) {
        Layer layer = layers[i];
        //这里依旧判断当前层是否为空，如果为空则会跑出空指针有慈航
        Preconditions.checkNotNull(layer);
        //根据当前网络层构建层更新器
        layerUpdaters[i] = UpdaterCreator.getUpdater(layer);

        //这里的更新器因为使用的是SGD，所以StateSize这里不管传入什么值，返回的均为0
        updaterStateSize += layerUpdaters[i].stateSizeForLayer(layer);
    }
    //初始化更新器状态
    //Initialize the updater state:
    if (updaterStateSize > 0) {
        //May be 0 if all SGD updaters, for example
        viewArray = Nd4j.createUninitialized(new int[] {1, updaterStateSize}, Nd4j.order());
    }

    //需要跨越多远获取子视图
    int soFar = 0;
    for (int i = 0; i < layers.length; i++) {
        //获取更新器状态
```

```java
            int thisSize = layerUpdaters[i].stateSizeForLayer(layers[i]);

            //如果为0
            if (thisSize == 0)
                continue;

            //如果不为0，则获取子视图
            INDArray view = viewArray.get(NDArrayIndex.point(0), NDArrayIndex.interval(soFar, soFar + thisSize));

            //设置到对应的更新器中
            layerUpdaters[i].setStateViewArray(layers[i], view, true);
            soFar += thisSize;
        }
    }
```

到这里`MultiLayerUpdater`执行完成，继续返回上层函数

1 updateGradientAccordingToParams

里面继续执行以下语句

```java
updater.update(layer, gradient, getIterationCount(model), batchSize);
```

1.2 updater.update()

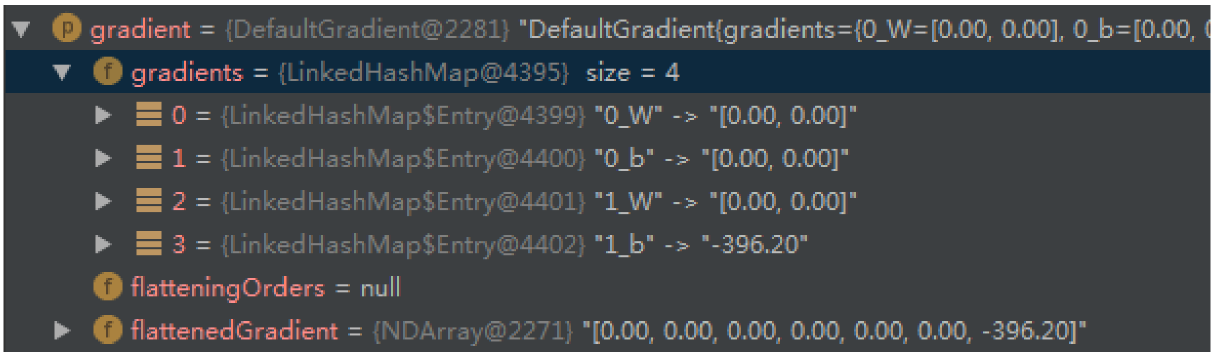
```java
@Override
public void update(Layer layer, Gradient gradient, int iteration, int batchSize) {
    MultiLayerNetwork mln = (MultiLayerNetwork) layer;

    //根据LayerUpdaters的个数构建 层梯度 的个数
    Gradient[] layerGradients = new Gradient[layerUpdaters.length];
    //实例化层梯度
    for (int i = 0; i < layerGradients.length; i++)
        layerGradients[i] = new DefaultGradient();

    //然后遍历已经计算好的梯度
    for (Map.Entry<String, INDArray> gradientPair : gradient.gradientForVariable().entrySet()) {
        //获取Key
        String key = gradientPair.getKey();
        //获取 '_'的位置
        int idx = key.indexOf('_');
        if (idx == -1)
            throw new IllegalStateException(
                                "Invalid key: MuliLayerNetwork Gradient key does not have layer separator: \"" + key
                                                + "\"");

        //截取网络层索引
        int layerIdx = Integer.parseInt(key.substring(0, idx));

        //截取后面的 W, b部分
        String newKey = key.substring(idx + 1);
        //根据网络层和w, b设置对应的梯度值
        layerGradients[layerIdx].gradientForVariable().put(newKey, gradientPair.getValue());
    }
```



```
▼ p gradient = {DefaultGradient@2281} "DefaultGradient{gradients={0_W=[0.00, 0.00], 0_b=[0.00, 
  ▼ f gradients = {LinkedHashMap@4395}  size = 4
    ▶ 0 = {LinkedHashMap$Entry@4399} "0_W" -> "[0.00, 0.00]"
    ▶ 1 = {LinkedHashMap$Entry@4400} "0_b" -> "[0.00, 0.00]"
    ▶ 2 = {LinkedHashMap$Entry@4401} "1_W" -> "[0.00, 0.00]"
    ▶ 3 = {LinkedHashMap$Entry@4402} "1_b" -> "-396.20"
    f flatteningOrders = null
  ▶ f flattenedGradient = {NDArray@2271} "[0.00, 0.00, 0.00, 0.00, 0.00, 0.00, -396.20]"
```

前面反向传播计算的梯度存储形式。

```
    //然后根据对应的值进行模型参数的更新
    for (int i = 0; i < layerUpdaters.length; i++) {
        layerUpdaters[i].update(mln.getLayer(i), layerGradients[i], iteration, batchSize);
    }
}
```

1.2.1 layerUpdaters[i].update()

```
 @Override
```

```java
public void update(Layer layer, Gradient gradient, int iteration, int miniBatchSize) {
    //参数名称
    String paramName;
    //原始梯度，更新之后的梯度
    INDArray gradientOrig, gradient2;
    //更新器
    GradientUpdater updater;

    //如果当前层是FrozenLayer，不更新网络参数
    if (layer instanceof FrozenLayer)
        return;

    preApply(layer, gradient, iteration);
```

1.2.1.1 preApply(layer, gradient, iteration)

根据函数数值是对梯度实现正则化，根据不同的策略对梯度进行处理。

```java
/**
 * Apply gradient normalization: scale based on L2, clipping etc.
 * RenormalizeL2PerLayer: divide all layer gradients by L2 to rescale
 * RenormalizeL2PerParamType: divide each parameter type gradient in a layer by L2 to rescale
 * ClipElementWiseAbsoluteValue: clip gradients per-element
 * ClipL2PerLayer: same as RenormalizeL2PerLayer but limited by gradient L2 norm for the layer meeting a threshold
 * ClipL2PerParamType: same as RenormalizeL2PerParamType but limited by gradient L2 norm for each parameter type in a layer meeting a threshold
 */
public void preApply(Layer layer, Gradient gradient, int iteration) {

    GradientNormalization normalization = layer.conf().getLayer().getGradientNormalization();
    if (normalization == null || normalization == GradientNormalization.None || layer.conf().isPretrain())
        return; //no op

    final double threshold = layer.conf().getLayer().getGradientNormalizationThreshold();

    switch (normalization) {
        case RenormalizeL2PerLayer:
            double sumSquares = 0.0;
            for (INDArray g : gradient.gradientForVariable().values()) {
                double l2 = g.norm2Number().doubleValue();
                //l2 norm: sqrt(sum_i g_i^2)
                sumSquares += l2 * l2;
            }
            double layerL2 = FastMath.sqrt(sumSquares);
            for (INDArray g : gradient.gradientForVariable().values()) {
                g.divi(layerL2);
            }
            break;
        case RenormalizeL2PerParamType:
            for (INDArray g : gradient.gradientForVariable().values()) {
                double l2 = Nd4j.getExecutioner().execAndReturn(new Norm2(g)).getFinalResult().doubleValue();
                g.divi(l2);
            }
            break;
        case ClipElementWiseAbsoluteValue:
            for (INDArray g : gradient.gradientForVariable().values()) {
                BooleanIndexing.replaceWhere(g, threshold, Conditions.greaterThan(threshold));
                BooleanIndexing.replaceWhere(g, -threshold, Conditions.lessThan(-threshold));
            }
            break;
        case ClipL2PerLayer:
            double sumSquares2 = 0.0;
            for (INDArray g : gradient.gradientForVariable().values()) {
                double l2 = Nd4j.getExecutioner().execAndReturn(new Norm2(g)).getFinalResult().doubleValue();
                //l2 norm: sqrt(sum_i g_i^2)
                sumSquares2 += l2 * l2;
            }
            double layerL22 = FastMath.sqrt(sumSquares2);
            if (layerL22 > threshold) {
                double scalingFactor = threshold / layerL22; // g = g / l2 * threshold ->
                for (INDArray g : gradient.gradientForVariable().values()) {
                    g.muli(scalingFactor);
                }
            }
            break;
        case ClipL2PerParamType:
            for (INDArray g : gradient.gradientForVariable().values()) {
                double l2 = g.norm2Number().doubleValue();
                if (l2 > threshold) {
                    double scalingFactor = l2 / threshold;
                    g.divi(scalingFactor);
                }
            }
            break;
        default:
            throw new RuntimeException(
                            "Unknown (or not implemented) gradient normalization strategy: " + normalization);
    }
}
```

1.2.1 layerUpdaters[i].update()

在对梯度进行正则化之后

```java
    //遍历梯度 map
    for (Map.Entry<String, INDArray> gradientPair : gradient.gradientForVariable().entrySet()) {
        paramName = gradientPair.getKey();
        if (!layer.conf().isPretrain() && PretrainParamInitializer.VISIBLE_BIAS_KEY.equals(paramName.split("_")[0]))
            continue;
        //首先获取原始梯度
        gradientOrig = gradientPair.getValue();
        //获取学习率衰减策略
        LearningRatePolicy decay = layer.conf().getLearningRatePolicy();

        //衰减率不为0或者更新器为NESTEROVS则应用衰减策略
        if (decay != LearningRatePolicy.None
                        || layer.conf().getLayer().getUpdater() == org.deeplearning4j.nn.conf.Updater.NESTEROVS)
            applyLrDecayPolicy(decay, layer, iteration, paramName);

        //根据名称和网络层初始化更新器
        updater = init(paramName, layer);
        //根据原始的提取新梯度
        gradient2 = updater.getGradient(gradientOrig, iteration);
```

1.2.1.2 updater.getGradient(gradientOrig, iteration);

使用学习率乘以当前的梯度

```
@Override
public INDArray getGradient(INDArray gradient, int iteration) {
    return gradient.muli(learningRate);
}
```

1.2.1 layerUpdaters[i].update()

在获取新地图之后继续执行以下步骤

```
            //使用正则化更新梯度以及参数
            postApply(layer, gradient2, paramName, miniBatchSize);
            //实现正则化之后更新梯度
            gradient.setGradientFor(paramName, gradient2);
        }
```

1.2.1.2 postApply(layer, gradient2, paramName, miniBatchSize);

实现正则化

```
/**
 * Apply the regularization
 *
 * @param layer
 * @param gradient
 * @param param
 */
public void postApply(Layer layer, INDArray gradient, String param, int
miniBatchSize) {
    NeuralNetConfiguration conf = layer.conf();
    INDArray params = layer.getParam(param);
    if (conf.isUseRegularization() && conf.getL2ByParam(param) > 0)
```

```
            gradient.addi(params.mul(conf.getL2ByParam(param))); //dC/dw = dC0/dw + lambda/n * w where C0 is pre-l2 cost function
        if (conf.isUseRegularization() && conf.getL1ByParam(param) > 0)
            gradient.addi(Transforms.sign(params).muli(conf.getL1ByParam(param)));
        if (conf.isMiniBatch())
            gradient.divi(miniBatchSize);

    }
```

LayerUpdater

```
package org.deeplearning4j.nn.updater;

/**
 * @author Adam Gibson
 */
public class LayerUpdater implements Updater {
    protected Map<String, GradientUpdater> updaterForVariable = new LinkedHashMap<>();
    protected INDArray viewArray;

    @Override
    public void setStateViewArray(Layer layer, INDArray viewArray, boolean initialize) {
        //Need to split this up into each parameter type...

        Map<String, INDArray> params = layer.paramTable();
        int count = 0;
        for (Map.Entry<String, INDArray> entry : params.entrySet()) {
            INDArray paramsArray = entry.getValue();
            GradientUpdater gu = init(entry.getKey(), layer);
            int thisSize = gu.stateSizeForInputSize(entry.getValue().length());
            if (thisSize == 0)
                continue;
            INDArray subset = viewArray.get(NDArrayIndex.point(0), NDArrayIndex.interval(count, count + thisSize));
            gu.setStateViewArray(subset, paramsArray.shape(), paramsArray.ordering(), initialize);
            count += thisSize;
        }
```

```java
    }

    public Map<String, GradientUpdater> getUpdaterForVariable() {
        return updaterForVariable;
    }

    @Override
    public INDArray getStateViewArray() {
        return viewArray;
    }

    @Override
    public int stateSizeForLayer(Layer layer) {
        Preconditions.checkNotNull(layer);
        Map<String, INDArray> params = layer.paramTable();
        int count = 0;
        for (Map.Entry<String, INDArray> entry : params.entrySet()) {
            GradientUpdater gu = init(entry.getKey(), layer);
            count += gu.stateSizeForInputSize(entry.getValue().length());
        }
        return count;
    }

    @Override
    public void update(Layer layer, Gradient gradient, int iteration, int miniBatchSize) {
        String paramName;
        INDArray gradientOrig, gradient2;
        GradientUpdater updater;

        if (layer instanceof FrozenLayer)
            return;

        preApply(layer, gradient, iteration);
        for (Map.Entry<String, INDArray> gradientPair : gradient.gradientForVariable().entrySet()) {
            paramName = gradientPair.getKey();
            if (!layer.conf().isPretrain() && PretrainParamInitializer.VISIBLE_BIAS_KEY.equals(paramName.split("_")[0]))
                continue;
            gradientOrig = gradientPair.getValue();
            LearningRatePolicy decay = layer.conf().getLearningRatePolicy();
            if (decay != LearningRatePolicy.None
                            || layer.conf().getLayer().getUpdater() == org.deeplearning4j.nn.conf.Updater.NESTEROVS)
                applyLrDecayPolicy(decay, layer, iteration, paramName);
            updater = init(paramName, layer);
            gradient2 = updater.getGradient(gradientOrig, iteration);
            postApply(layer, gradient2, paramName, miniBatchSize);
            gradient.setGradientFor(paramName, gradient2);
        }
    }

    /**
     * Apply the regularization
     *
     * @param layer
     * @param gradient
     * @param param
     */
    public void postApply(Layer layer, INDArray gradient, String param, int miniBatchSize) {
        NeuralNetConfiguration conf = layer.conf();
        INDArray params = layer.getParam(param);
        if (conf.isUseRegularization() && conf.getL2ByParam(param) > 0)
            gradient.addi(params.mul(conf.getL2ByParam(param))); //dC/dw = dC0/dw + lambda/n * w where C0 is pre-l2 cost function
        if (conf.isUseRegularization() && conf.getL1ByParam(param) > 0)
            gradient.addi(Transforms.sign(params).muli(conf.getL1ByParam(param)));
        if (conf.isMiniBatch())
            gradient.divi(miniBatchSize);

    }

    /**
     *  Update momentum if schedule exist
     */
    public void applyMomentumDecayPolicy(Layer layer, int iteration, String variable) {
        NeuralNetConfiguration conf = layer.conf();
        if (conf.getLayer().getMomentumSchedule().containsKey(iteration)) {
            conf.getLayer().setMomentum(conf.getLayer().getMomentumSchedule().get(iteration));
            if (updaterForVariable.get(variable) != null) {

```

```java
updaterForVariable.get(variable).update(conf.getLearningRateByParam(variable),
                                conf.getLayer().getMomentumSchedule().get(iteration));
            }
        } else if (updaterForVariable.get(variable) != null) {
updaterForVariable.get(variable).update(conf.getLearningRateByParam(variable),
                            conf.getLayer().getMomentum());
        }
    }

    /**
     *  Update learning rate based on policy
     */
    public void applyLrDecayPolicy(LearningRatePolicy decay, Layer layer, int iteration, String variable) {
        NeuralNetConfiguration conf = layer.conf();
        double decayRate = layer.conf().getLrPolicyDecayRate();
        double lr = conf.getLearningRateByParam(variable);
        switch (decay) {
            case Exponential:
                conf.setLearningRateByParam(variable, lr * Math.pow(decayRate, iteration));
                break;
            case Inverse:
                conf.setLearningRateByParam(variable,
                                lr / Math.pow((1 + decayRate * iteration), conf.getLrPolicyPower()));
                break;
            case Step:
                conf.setLearningRateByParam(variable,
                                lr * Math.pow(decayRate, Math.floor(iteration / conf.getLrPolicySteps())));
                break;
            case TorchStep:
                if (iteration > 1 && conf.getLrPolicySteps() % iteration == 0)
                    conf.setLearningRateByParam(variable, lr * decayRate);
                break;
            case Poly:
                conf.setLearningRateByParam(variable, lr * Math
                                .pow((1 - ((double) iteration) / conf.ge
tNumIterations()), conf.getLrPolicyPower()));
                break;
            case Sigmoid:
                conf.setLearningRateByParam(variable,
                                lr / (1 + Math.exp(-decayRate * (iteration - conf.getLrPolicySteps()))));
                break;
            case Schedule:
                if (conf.getLayer().getLearningRateSchedule().containsKey(iteration))
                    conf.setLearningRateByParam(variable, conf.getLayer().getLearningRateSchedule().get(iteration));
                break;
        }
        if (layer.conf().getLayer().getUpdater() == org.deeplearning4j.nn.conf.Updater.NESTEROVS) {
            applyMomentumDecayPolicy(layer, iteration, variable);
        } else if (updaterForVariable.get(variable) != null) {
            updaterForVariable.get(variable).update(conf.getLearningRateByParam(variable));
        }
    }

    /**
     *  Apply gradient normalization: scale based on L2, clipping etc.
     *  RenormalizeL2PerLayer: divide all layer gradients by L2 to rescale
     *  RenormalizeL2PerParamType: divide each parameter type gradient in a layer by L2 to rescale
     *  ClipElementWiseAbsoluteValue: clip gradients per-element
     *  ClipL2PerLayer: same as RenormalizeL2PerLayer but limited by gradient L2 norm for the layer meeting a threshold
     *  ClipL2PerParamType: same as RenormalizeL2PerParamType but limited by gradient L2 norm for each parameter type in a layer meeting a threshold
     */
    public void preApply(Layer layer, Gradient gradient, int iteration) {

        GradientNormalization normalization = layer.conf().getLayer().getGradientNormalization();
        if (normalization == null || normalization == GradientNormalization.None || layer.conf().isPretrain())
            return; //no op

        final double threshold = layer.conf().getLayer().getGradientNormalizationThreshold();

        switch (normalization) {
            case RenormalizeL2PerLayer:
                double sumSquares = 0.0;
                for (INDArray g : gradient.gradientForVariable().values()) {
                    double l2 = g.norm2Number().doubleValue();
                    //l2 norm: sqrt(sum_i g_i^2)
                    sumSquares += l2 * l2;
                }
                double layerL2 = FastMath.sqrt(sumSquares);
                for (INDArray g : gradient.gradientForVariable().values()) {
                    g.divi(layerL2);
                }
                break;
            case RenormalizeL2PerParamType:
                for (INDArray g : gradient.gradientForVariable().values()) {
                    double l2 = Nd4j.getExecutioner().execAndReturn(new Norm2(g)).getFinalResult().doubleValue();
                    g.divi(l2);
                }
                break;
            case ClipElementWiseAbsoluteValue:
                for (INDArray g : gradient.gradientForVariable().values()) {
                    BooleanIndexing.replaceWhere(g, threshold, Conditions.greaterThan(threshold));
                    BooleanIndexing.replaceWhere(g, -threshold, Conditions.lessThan(-threshold));
                }
                break;
            case ClipL2PerLayer:
                double sumSquares2 = 0.0;
                for (INDArray g : gradient.gradientForVariable().values()) {
                    double l2 = Nd4j.getExecutioner().execAndReturn(new Norm2(g)).getFinalResult().doubleValue();
                    //l2 norm: sqrt(sum_i g_i^2)
                    sumSquares2 += l2 * l2;
                }
```

```
                    double layerL22 = FastMath.sqrt(sumSquares2);
                    if (layerL22 > threshold) {
                        double scalingFactor = threshold / layerL22; // g = g / l2 * threshold ->
                        for (INDArray g : gradient.gradientForVariable().values()) {
                            g.muli(scalingFactor);
                        }
                    }
                    break;
                case ClipL2PerParamType:
                    for (INDArray g : gradient.gradientForVariable().values()) {
                        double l2 = g.norm2Number().doubleValue();
                        if (l2 > threshold) {
                            double scalingFactor = l2 / threshold;
                            g.divi(scalingFactor);
                        }
                    }
                    break;
                default:
                    throw new RuntimeException(
                                    "Unknown (or not implemented) gradient normalization strategy: " + normalization);
            }
        }


    public void init() {
        //No op
    }

    public GradientUpdater init(String variable, Layer layer) {
        GradientUpdater updater = updaterForVariable.get(variable);
        if (updater == null) {
            org.deeplearning4j.nn.conf.Updater u = layer.conf().getLayer().getUpdaterByParam(variable);
            switch (u) {
                case SGD:
                    updater = new org.nd4j.linalg.learning.Sgd(layer.conf().getLearningRateByParam(variable));
                    break;
                case ADAM:
                    updater = new
```

```java
Adam(layer.conf().getLearningRateByParam(variable),
                                    layer.conf().getLayer().getAdamMeanDecay(),
                                    layer.conf().getLayer().getAdamVarDecay(), layer.conf().getLayer().getEpsilon());
                    break;
                case ADADELTA:
                    updater = new AdaDelta(layer.conf().getLayer().getRho(), layer.conf().getLayer().getEpsilon());
                    break;
                case NESTEROVS:
                    updater = new Nesterovs(layer.conf().getLayer().getMomentum(),
                                    layer.conf().getLearningRateByParam(variable));
                    break;
                case ADAGRAD:
                    updater = new AdaGrad(layer.conf().getLearningRateByParam(variable),
                                    layer.conf().getLayer().getEpsilon());
                    break;
                case RMSPROP:
                    updater = new org.nd4j.linalg.learning.RmsProp(layer.conf().getLearningRateByParam(variable),
                                    layer.conf().getLayer().getRmsDecay(), layer.conf().getLayer().getEpsilon());
                    break;
                case NONE:
                    updater = new NoOpUpdater();
                    break;
                case CUSTOM:
                    throw new UnsupportedOperationException("Custom updaters: not yet implemented");
                default:
                    throw new IllegalArgumentException("Unknown updater: " + u);
            }
            updaterForVariable.put(variable, updater);
        }
        return updater;
    }

    @Override
    public boolean equals(Object other) {
        if (!(other instanceof LayerUpdater))
            return false;
        return updaterForVariable.equals(((LayerUpdater) other).updaterForVariable);
    }

    @Override
    public int hashCode() {
        int result = 19;
        result = 31 * result + (updaterForVariable == null ? 0 : updaterForVariable.hashCode());
        return result;
    }

    @Override
    public Updater clone() {
        Map<String, GradientUpdater> newMap = new HashMap<>();
        for (Map.Entry<String, GradientUpdater> entry : updaterForVariable.entrySet()) {
            newMap.put(entry.getKey(), entry.getValue().getAggregator(true).getUpdater());
        }

        LayerUpdater updater;
        try {
            updater = this.getClass().getConstructor().newInstance();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        updater.updaterForVariable = newMap;
        return updater;
    }
}
```

applyLrDecayPolicy

```java
/**
     *  Update learning rate based on policy
     */
    public void applyLrDecayPolicy(LearningRatePolicy decay, Layer laye
r, int iteration, String variable) {
    NeuralNetConfiguration conf = layer.conf();
    double decayRate = layer.conf().getLrPolicyDecayRate();
    double lr = conf.getLearningRateByParam(variable);
    switch (decay) {
        case Exponential:
            conf.setLearningRateByParam(variable, lr * Math.pow(decayRate, iteration));
            break;
        case Inverse:
            conf.setLearningRateByParam(variable,
                            lr / Math.pow((1 + decayRate * iteration), conf.getLrPolicyPower()));
            break;
        case Step:
            conf.setLearningRateByParam(variable,
                            lr * Math.pow(decayRate, Math.floor(iteration / conf.getLrPolicySteps())));
            break;
        case TorchStep:
            if (iteration > 1 && conf.getLrPolicySteps() % iteration == 0)
                conf.setLearningRateByParam(variable, lr * decayRate);
            break;
        case Poly:
            conf.setLearningRateByParam(variable, lr * Math
                            .pow((1 - ((double) iteration) / conf.getNumIterations()), conf.getLrPolicyPower()));
            break;
        case Sigmoid:
            conf.setLearningRateByParam(variable,
                            lr / (1 + Math.exp(-decayRate * (iteration - conf.getLrPolicySteps()))));
            break;
        case Schedule:
            if (conf.getLayer().getLearningRateSchedule().containsKey(iteration))
                conf.setLearningRateByParam(variable, conf.getLayer().getLearningRateSchedule().get(iteration));
            break;
    }
    if (layer.conf().getLayer().getUpdater() == org.deeplearning4j.nn.conf.Updater.NESTEROVS) {
        applyMomentumDecayPolicy(layer, iteration, variable);
    } else if (updaterForVariable.get(variable) != null) {
```

```
updaterForVariable.get(variable).update(conf.getLearningRateByParam(variable));
    }
}
```