在反向传播计算完梯度和误差之后，返回到`MultiLayerNetwork.computeGradientAndScore()`方法中继续计算输出函数的偏差

MultiLayerNetwork.computeGradientAndScore()

```
@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), c
    alcL2(true), true);

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

接下来的研究重点

是`((IOutputLayer) getOutputLayer()).computeScore(calcL1(true), calcL2(true), true);`

这是主要是调用输出层的计算分数的方法，其接口为

```
/**
 * Compute score after labels and input have been set.
 *
 * @param fullNetworkL1 L1 regularization term for the entire network
 * @param fullNetworkL2 L2 regularization term for the entire network
 * @param training      whether score should be calculated at train or tes
t time (this affects things like application of
 *                      dropout, etc)
 * @return score (loss function)
 */
double computeScore(double fullNetworkL1, double fullNetworkL2, boolean tr
aining);
```

在输出层设置好标签和输入之后计算损失函数分数，并且使用l1, l2正则化参数。

在进入`computeScore`方法之前，首先要计算得出网络的l1和l2参数。

L1计算

MultiLayerNetwork的方法

整个网络的L1参数需要调用每一层(Layer)计算l1并且进行求和

```
@Override
public double calcL1(boolean backpropParamsOnly) {
    double l1 = 0.0;
    for (int i = 0; i < layers.length; i++) {
        l1 += layers[i].calcL1(backpropParamsOnly);
    }
    return l1;
}
```

而每一个单独Layer计算L1的方式是：

1. 权重L1 = 层设置的权重L1 * 该层权重矩阵的1范式
   （`getParam(DefaultParamInitializer.BIAS_KEY).norm1Number().doubleValue()`该式子是求取权重矩阵的一范式的值）
2. 偏置L1 = 层设置的偏置L1 * 该层偏置矩阵的1范式
   （`getParam(DefaultParamInitializer.BIAS_KEY).norm1Number().doubleValue()`该式子是求取偏置矩阵的一范式的值）
3. 层L1 = 权重L1 + 偏置L1

```
@Override
public double calcL1(boolean backpropParamsOnly) {
    if (!conf.isUseRegularization())
        return 0.0;
    double l1Sum = 0.0;
    if (conf.getL1ByParam(DefaultParamInitializer.WEIGHT_KEY) > 0.0) {
        l1Sum += conf.getL1ByParam(DefaultParamInitializer.WEIGHT_KEY)
                        *
getParam(DefaultParamInitializer.WEIGHT_KEY).norm1Number().doubleValue();
    }
    if (conf.getL1ByParam(DefaultParamInitializer.BIAS_KEY) > 0.0) {
        l1Sum += conf.getL1ByParam(DefaultParamInitializer.BIAS_KEY)
                        * getParam(DefaultParamInitializer.BIAS_KEY).norm1N
umber().doubleValue();
    }
    return l1Sum;
```

```
}
```

L2计算

MultiLayerNetwork的方法

整个网络的L2参数需要调用每一层(Layer)计算L2并且进行求和

```
@Override
public double calcL2(boolean backpropParamsOnly) {
    double l2 = 0.0;
    for (int i = 0; i < layers.length; i++) {
        l2 += layers[i].calcL2(backpropParamsOnly);
    }
    return l2;
}
```

因为矩阵2范式的计算公式不同，为此对应参数的2范式计算方式也有所改变。但是总体思路不变

```
@Override
public double calcL2(boolean backpropParamsOnly) {
    if (!conf.isUseRegularization())
        return 0.0;

    //L2 norm: sqrt( sum_i x_i^2 ) -> want sum squared weights, so l2 norm squared
    double l2Sum = 0.0;
    if (conf.getL2ByParam(DefaultParamInitializer.WEIGHT_KEY) > 0.0) {
        double l2Norm = getParam(DefaultParamInitializer.WEIGHT_KEY).norm2Number().doubleValue();
        l2Sum += 0.5 * conf.getL2ByParam(DefaultParamInitializer.WEIGHT_KEY) * l2Norm * l2Norm;
    }
    if (conf.getL2ByParam(DefaultParamInitializer.BIAS_KEY) > 0.0) {
        double l2Norm = getParam(DefaultParamInitializer.BIAS_KEY).norm2Number().doubleValue();
        l2Sum += 0.5 * conf.getL2ByParam(DefaultParamInitializer.BIAS_KEY) * l2Norm * l2Norm;
    }
    return l2Sum;
}
```

computeScore

在L1， L2参数都计算完成之后继续看如何计算损失函数得分

```
/** Compute score after labels and input have been set.
 * @param fullNetworkL1 L1 regularization term for the entire network
 * @param fullNetworkL2 L2 regularization term for the entire network
 * @param training whether score should be calculated at train or test time (this affects things like application of
 *                 dropout, etc)
 * @return score (loss function)
 */
@Override
public double computeScore(double fullNetworkL1, double fullNetworkL2, boolean training) {

    //首先对当前的输入和标签做检查
    if (input == null || labels == null)
        throw new IllegalStateException("Cannot calculate score without input and labels");
    //初始化L1和L2值
    this.fullNetworkL1 = fullNetworkL1;
    this.fullNetworkL2 = fullNetworkL2;

    //根据输入，使用 y = xw + b做一个变换
    INDArray preOut = preOutput2d(training);

    //获取损失函数
    ILossFunction lossFunction = layerConf().getLossFn();

    //double score = lossFunction.computeScore(getLabels2d(), preOut, layerConf().getActivationFunction(), maskArray, false);

    //调用损失函数的计算损失得分
    double score = lossFunction.computeScore(getLabels2d(), preOut, layerConf().getActivationFn(), maskArray,
                    false);

    //获取的得分加上L1和L2值
    score += fullNetworkL1 + fullNetworkL2;

    //除以miniBatch大小，求取平均值
    score /= getInputMiniBatchSize();

    this.score = score;
```

```
37.
38.         return score;
39.     }
```

lossFunction.computeScore