1 backprop()

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

根据注释可以看出是基于输出层和标签计算多层网络的梯度。之后跳进另外一个函数内。

1.1 calcBackpropGradients

这段代码提供的注释很多，这里直接翻译原有注释

```
/** 计算梯度和偏差. 在一下的两个地方使用:
 * (a) backprop (用于标准的网络结构)
 * (b) backpropGradient (layer类方法, 用于当MultiLayerNetwork类被用作layer的时候)
 * @param epsilon 偏差 (technically errors .* activations). 当withOutputLayer = true时, 不被使用
 * @param withOutputLayer 如果为true: 认为最后一层为输出层, 并且根据标签计算偏差. 在这种情况下输入的epsilon将不会被使用 (可能为null)
 *                        如果为false: 计算反向传播的梯度
 * @return 输入的梯度和偏差 (epsilon)
 */
protected Pair<Gradient, INDArray> calcBackpropGradients(INDArray epsilon, boolean
```

```
    withOutputLayer) {
        if (flattenedGradients == null)
            initGradientsView();
```

在刚开始运行的时候`flattenedGradients`字段为null。之后进入`initGradientsView()`方法。

1.1.1 initGradientsView()

```
/**
 * This method: 用于初始化展平的梯度数据 (用于反向传播)并且对于所有的网络层设置梯度子集
 * 作为一般规则，在通过fit（DataSet）或Fit（DataSetIterator）进行训练时，不需要手动调用它，
 */
public void initGradientsView() {
    if (layers == null)
        init();

    //获取网络层的层数
    int nLayers = layers.length;

    //首先：计算（反向传播）参数的总长度
    int backpropParamLength = 0;
    int[] nParamsPerLayer = new int[nLayers];
    for (int i = 0; i < nLayers; i++) {
        NeuralNetConfiguration conf = layerWiseConfigurations.getConf(i);
        nParamsPerLayer[i] = layers[i].conf().getLayer().initializer().numParams(conf);
        backpropParamLength += nParamsPerLayer[i];
    }
```

1.1.1.1 numParams(NeuralNetConfiguration conf)

```
@Override
public int numParams(NeuralNetConfiguration conf) {
    org.deeplearning4j.nn.conf.layers.FeedForwardLayer layerConf =
                    (org.deeplearning4j.nn.conf.layers.FeedForwardLayer) conf.getLayer();
    int nIn = layerConf.getNIn();
    int nOut = layerConf.getNOut();
    return nIn * nOut + nOut; //weights + bias
}
```

因为需要计算每一层的参数个数，所以需要调用如上的函数，每一个层参数的计算个数方法为`nIn * nOut + nOut`。
然后继续返回到上层函数继续执行。

1.1.1 initGradientsView()

```
//以上计算得出参数的总个数之后，创建ndarray。'f'代表数组的存储顺序，有兴趣可以查阅nd4j官网
flattenedGradients = Nd4j.zeros(new int[] {1, backpropParamLength}, 'f');

int backpropParamsSoFar = 0;
for (int i = 0; i < layers.length; i++) {
    //如果该层参数个数为0则跳过当前层
    if (nParamsPerLayer[i] == 0)
        continue; //This layer doesn't have any parameters...

    //NDArrayIndex.point(0)用于指定是第几行，这里就是指定第0行
    //NDArrayIndex.interval(backpropParamsSoFar, backpropParamsSoFar + nParamsPerLayer[i]) 用于获取列坐标索引
    INDArray thisLayerGradView = flattenedGradients.get(NDArrayIndex.point(0),
```

```
                    NDArrayIndex.interval(backpropParamsSoFar, backpropParamsSoFar + nParamsPerLayer[i]));
        layers[i].setBackpropGradientsViewArray(thisLayerGradView);
        backpropParamsSoFar += nParamsPerLayer[i];
    }
```

刚开始看这段代码的时候，不懂很明白这是要做什么，这时候需要回看到这个函数的目的是什么。这个函数的目的是`initializes the flattened gradients array (used in backprop) and sets the appropriate subset in all layers`
这里的for loop主要是设置各个层的梯度数组。

1.1.1.2 flattenedGradients.get()

```
/**
 * Returns a subset of this array based on the specified
 * indexes
 *
 * @param indexes the indexes in to the array
 * @return a view of the array with the specified indices
 */
INDArray get(INDArrayIndex... indexes);
```

这个方法主要是根据给定的数组索引获取数组的子集。

1.1.1.3 point(int point)

```
/**
 * Returns a point index
 * @param point the point index
 * @return the point index based
 * on the specified point
 */
public static INDArrayIndex point(int point) {
    return new PointIndex(point);
}
```

用户返回指定点的索引，在这里使用主要是指定行索引。

1.1.1.4 interval(int begin, int end)

```
/**
 * Generates an interval from begin (inclusive) to end (exclusive)
 *
 * @param begin the begin
 * @param end   the end index
 * @return the interval
 */
public static INDArrayIndex interval(int begin, int end) {
    return interval(begin, 1, end, false);
}
```

生成区间[begin, end)区间内数据的索引，用于获取列索引。

1.1 calcBackpropGradients

```
String multiGradientKey;
//使用初始化之后的flattenedGradients构造梯度类
```

```
Gradient gradient = new DefaultGradient(flattenedGradients);
Layer currLayer;

//计算并应用每个图层的后向梯度
/**
 * 跳过索引的输出层，只是向后循环更新每个层的系数。
 * (当 withOutputLayer == true)
 *
 * 激活为每个图层应用激活函数，并将其设置为下一图层的输入。
 *
 * Typical literature contains most trivial case for the error calculation: wT * weights
 * This interpretation transpose a few things to get mini batch because ND4J is rows vs columns organization for params
 */
int numLayers = getnLayers();
//将梯度存储为列表; used to ensure iteration order in DefaultGradient linked hash map. i.e., layer 0 first instead of output layer
LinkedList<Triple<String, INDArray, Character>> gradientList = new LinkedList<>();
```

在构造梯度类，梯度列表以及获取网络结构的层数之后，继续执行以下语句：

```
int layerFrom;
Pair<Gradient, INDArray> currPair;
//判断是否使用输出层
if (withOutputLayer) {
    //对输出层做类型检查
    if (!(getOutputLayer() instanceof IOutputLayer)) {
        log.warn("Warning: final layer isn't output layer. You cannot use backprop without an output layer.");
        return null;
    }

    //获取输出层
    IOutputLayer outputLayer = (IOutputLayer) getOutputLayer();
    //对标签进行检查
    if (labels == null)
        throw new IllegalStateException("No labels found");
    //设置输出层的标签，用于计算偏差
    outputLayer.setLabels(labels);
    //首先获取输出层的梯度
    currPair = outputLayer.backpropGradient(null);
```

接下来单步进入输出层反向传播梯度的的函数

1.2.1 outputLayer.backpropGradient(INDArray epsilon)

```
 @Override
public Pair<Gradient, INDArray> backpropGradient(INDArray epsilon) {
    Pair<Gradient, INDArray> pair = getGradientsAndDelta(preOutput2d(true)); //Returns Gradient and delta^(this), not Gradient and epsilon^(this-1)
    INDArray delta = pair.getSecond();

    INDArray epsilonNext = params.get(DefaultParamInitializer.WEIGHT_KEY).mmul(delta.transpose()).transpose();
    return new Pair<>(pair.getFirst(), epsilonNext);
}
```

1.2.1.1 preOutput2d(true)

调用链如下

```
protected INDArray preOutput2d(boolean training) {
    return preOutput(training);
}

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;
}
```

因为这里是OutputLayer在调用，这里相当于使用`y = xw + b`计算并得出输出层还未经过激活函数变换的输出。并将计算得出的结果传入到`getGradientsAndDelta(INDArray preOut)`方法中

1.2.1.2 getGradientsAndDelta(INDArray preOut)

```
/** Returns tuple: {Gradient,Delta,Output} given preOut */
private Pair<Gradient, INDArray> getGradientsAndDelta(INDArray preOut) {
    //首先获取当前层的损失函数
    ILossFunction lossFunction = layerConf().getLossFn();
    //获取2维的列表。（主要是针对RNN CNN这种网络，因为他们的数据组成方式是3d或者4d，需要转化为2d之后才能残油矩阵运算）
    INDArray labels2d = getLabels2d();
    //判断两个矩阵的形状，进行一个检验
    if (labels2d.size(1) != preOut.size(1)) {
        throw new DL4JInvalidInputException("Labels array numColumns (size(1) = " + labels2d.size(1)
                        + ") does not match output layer" + " number of outputs (nOut = " + preOut.size(1) + ")");
    }
    //传入标签，输出层的输出，激活函数和掩码，利用损失函数来计算偏差
    INDArray delta = lossFunction.computeGradient(labels2d, preOut,
```

```
layerConf().getActivationFn(), maskArray);
```

1.2.1.2.1 lossFunction.computeGradient

```
@Override
public INDArray computeGradient(INDArray labels, INDArray preOutput, IActivation activationFn
, INDArray mask) {
    INDArray gradients = super.computeGradient(labels, preOutput, activationFn, mask);
    return gradients.divi(labels.size(1));
}
```

之后调用父类的`computeGradient()`方法来计算梯度

```
@Override
public INDArray computeGradient(INDArray labels, INDArray preOutput, IActivation activationFn
, INDArray mask) {
    //因为前面获取的preOutput只是 y = xw + b部分，尚未经过激活函数的变化
    //所以这里先 复制一份preOutput，然后经过激活函数的变换获得output
    INDArray output = activationFn.getActivation(preOutput.dup(), true);

    //这里先计算两个矩阵之间的差距， 然后再*2
    //这里是因为当前的损失函数类型为LossMSE()， 所以需要对
    INDArray dLda = output.subi(labels).muli(2);


    //损失函数的权重为null，为此不进行改步操作
    if (weights != null) {
        dLda.muliRowVector(weights);
    }

    //如若使用掩码，则与掩码进行计算
    f(mask != null && LossUtil.isPerOutputMasking(dLda, mask)){
        //For *most* activation functions: we don't actually need to mask dL/da in addition t
o masking dL/dz later
        //but: some, like softmax, require both (due to dL/dz_i being a function of dL/da_j,
for i != j)
        //We could add a special case for softmax (activationFn instanceof ActivationSoftmax)
but that would be
        // error prone - but buy us a tiny bit of performance
        LossUtil.applyMask(dLda, mask);
    }

    //根据激活函数计算梯度
    INDArray gradients = activationFn.backprop(preOutput, dLda).getFirst();
```

这里会调用激活函数的backprop，主要是用于求其在激活函数之后的梯度。
因为我这里最后一层的函数为IDENTITY，本身对输入不会做任何变换，所以直接返回本身。

```
@Override
public Pair<INDArray, INDArray> backprop(INDArray in, INDArray epsilon) {
    return new Pair<>(epsilon, null);
}
```

之后调用Pair.getFirst()，就是为了获取激活函数求导之后的epsilon

```
    //Loss function with masking
    if (mask != null) {
```

```
            LossUtil.applyMask(gradients, mask);
        }

        return gradients;
    }
```

在这个函数最后调用掩码进行计算，然后这个函数到这里执行完毕，返回上层函数。

1.2.1.2.1 lossFunction.computeGradient

```
        INDArray gradients = super.computeGradient(labels, preOutput, activationFn, mask);
        return gradients.divi(labels.size(1));
    }
```

调用`gradients.divi(labels.size(1))`，梯度除以labels的第二个维度。一般情况下为最后一层的神经元个数。然后返回到`getGradientsAndDelta(INDArray preOut)`方法中。

1.2.1.2 getGradientsAndDelta(INDArray preOut)

```
        //初始化新的梯度类
        Gradient gradient = new DefaultGradient();

        //获取权重梯度视图
        INDArray weightGradView = gradientViews.get(DefaultParamInitializer.WEIGHT_KEY);
        //获取偏重梯度视图
        INDArray biasGradView = gradientViews.get(DefaultParamInitializer.BIAS_KEY);

        //Equivalent to:  weightGradView.assign(input.transpose().mmul(delta));
        //相当于更新权重的梯度
        Nd4j.gemm(input, delta, weightGradView, true, false, 1.0, 0.0);

        //对权重梯度进行赋值，初始值为 delta的第0行之和
        biasGradView.assign(delta.sum(0));

        //将权重的梯度放入到初始化之后的梯度类中
        gradient.gradientForVariable().put(DefaultParamInitializer.WEIGHT_KEY, weightGradView);
        gradient.gradientForVariable().put(DefaultParamInitializer.BIAS_KEY, biasGradView);

        //返回梯度和delta
        return new Pair<>(gradient, delta);
    }
```

返回到上层函数`backpropGradient(INDArray epsilon)`中

1.2.1 outputLayer.backpropGradient(INDArray epsilon)

```
        //获取delta
        INDArray delta = pair.getSecond();
        //误差 = (w * delta^T) ^ T
        INDArray epsilonNext =
    params.get(DefaultParamInitializer.WEIGHT_KEY).mmul(delta.transpose()).transpose();
        return new Pair<>(pair.getFirst(), epsilonNext);
    }
```

1.1 calcBackpropGradients

然后返回上层函数继续执行

```
        //获取到了当前层的Pair<Gradient, INDArray>
        currPair = outputLayer.backpropGradient(null);

        //遍历梯度map里面的  权重梯度以及偏置梯度
        for (Map.Entry<String, INDArray> entry : currPair.getFirst().gradientForVariable().entrySet()) {
            //获取原始的名称，基本为"W", "b"
            String origName = entry.getKey();
            //然后根据当前所在的层数进行拼装，比如变成"1_W", "1_b"
            multiGradientKey = String.valueOf(numLayers - 1) + "_" + origName;

            //然后构建三元组，  字符串新名称，  梯度INDArray，以及展平之后的梯度INDArray
            //添加到链表的最后
            gradientList.addLast(new Triple<>(multiGradientKey, entry.getValue(),
                            currPair.getFirst().flatteningOrderForVariable(origName)));
        }

        //判断是否有输入预处理操作
        if (getLayerWiseConfigurations().getInputPreProcess(numLayers - 1) != null)
            currPair = new Pair<>(currPair.getFirst(),
                            this.layerWiseConfigurations.getInputPreProcess(numLayers - 1)
                                            .backprop(currPair.getSecond(), getInputMiniBatchSize()));

        //numLayers - 1为输出层，且输出层的梯度和误差以及计算好了
        //所以layerFrom为 numLayers - 2
        layerFrom = numLayers - 2;
    } else {

        //如果无输出层，则从numLayers - 1开始
        currPair = new Pair<>(null, epsilon);
        layerFrom = numLayers - 1;
    }


    //根据前面计算的梯度来进行反向传播
    // Calculate gradients for previous layers & drops output layer in count
    for (int j = layerFrom; j >= 0; j--) {
        //获取当前网络层
        currLayer = getLayer(j);
        //如果当前层是FrozenLayer，终止反向传播
        if (currLayer instanceof FrozenLayer)
            break;

        //根据上一层的误差来重新计算梯度和误差便于继续反向传播
        currPair = currLayer.backpropGradient(currPair.getSecond());

        //新建三元组子列表
        LinkedList<Triple<String, INDArray, Character>> tempList = new LinkedList<>();

        //遍历梯度和误差
        for (Map.Entry<String, INDArray> entry : currPair.getFirst().gradientForVariable().entrySet()) {
            String origName = entry.getKey();
            multiGradientKey = String.valueOf(j) + "_" + origName;
            tempList.addFirst(new Triple<>(multiGradientKey, entry.getValue(),
                            currPair.getFirst().flatteningOrderForVariable(origName)));
        }

```

```
        //加入到gradientList的前面
        for (Triple<String, INDArray, Character> triple : tempList)
            gradientList.addFirst(triple);

        //Pass epsilon through input processor before passing to next layer (if applicable)
        if (getLayerWiseConfigurations().getInputPreProcess(j) != null)
            currPair = new Pair<>(currPair.getFirst(),
    getLayerWiseConfigurations().getInputPreProcess(j)
                            .backprop(currPair.getSecond(), getInputMiniBatchSize()));
    }

    //Add gradients to Gradients (map), in correct order
    //把所有梯度以正确的顺序加入到Gradients (map)中
    for (Triple<String, INDArray, Character> triple : gradientList) {
        gradient.setGradientFor(triple.getFirst(), triple.getSecond(), triple.getThird());
    }

    //返回当前的梯度，和误差
    return new Pair<>(gradient, currPair.getSecond());
```

1 backprop()

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

对于当前的`MultiLayerNetwork`类设置成员变量的值。

Pair数据结构源码解读

```
package org.deeplearning4j.berkeley;

/**
 * A generic-typed pair of objects.
 * @author Dan Klein
 */
public class Pair<F, S> implements Serializable, Comparable<Pair<F, S>> {
    static final long serialVersionUID = 42;

    F first;
    S second;

    public F getFirst() {
        return first;
    }

    public S getSecond() {
        return second;
    }

    public void setFirst(F pFirst) {
        first = pFirst;
```

```
    }

    public void setSecond(S pSecond) {
        second = pSecond;
    }

    public Pair<S, F> reverse() {
        return new Pair<>(second, first);
    }
}
```

Gradient

```
package org.deeplearning4j.nn.gradient;

/**
 * Generic gradient
 *
 * @author Adam Gibson
 */
public interface Gradient extends Serializable {

    /**
     * Gradient look up table
     *
     * @return the gradient look up table
     */
    Map<String, INDArray> gradientForVariable();

    /**
     * The full gradient as one flat vector
     *
     * @return
     */
    INDArray gradient(List<String> order);

    /**
     * The full gradient as one flat vector
     *
     * @return
     */
    INDArray gradient();

    /**
     * Clear residual parameters (useful for returning a gradient and then clearing old objec
ts)
     */
    void clear();

    /**
     * The gradient for the given variable
     *
     * @param variable the variable to get the gradient for
     * @return the gradient for the given variable or null
     */
    INDArray getGradientFor(String variable);

    /**
     * Update gradient for the given variable
```

```
     *
     * @param variable the variable to get the gradient for
     * @param gradient the gradient values
     * @return the gradient for the given variable or null
     */
    INDArray setGradientFor(String variable, INDArray gradient);

    /**
     * Update gradient for the given variable; also (optionally) specify the order in which t
he array should be flattened
     * to a row vector
     *
     * @param variable        the variable to get the gradient for
     * @param gradient        the gradient values
     * @param flatteningOrder the order in which gradients should be flattened (null ok - def
ault)
     * @return the gradient for the given variable or null
     */
    INDArray setGradientFor(String variable, INDArray gradient, Character flatteningOrder);

    /**
     * Return the gradient flattening order for the specified variable, or null if it is not
explicitly set
     * @param variable    Variable to return the gradient flattening order for
     * @return            Order in which the specified variable's gradient should be
flattened
     */
    Character flatteningOrderForVariable(String variable);

}
```

DefaultGradient

```
package org.deeplearning4j.nn.gradient;

/**
 * Default gradient implementation. Basically lookup table
 * for ndarrays
 *
 * @author Adam Gibson
 */

public class DefaultGradient implements Gradient {
    public static final char DEFAULT_FLATTENING_ORDER = 'f';
    private Map<String, INDArray> gradients = new LinkedHashMap<>();
    private Map<String, Character> flatteningOrders;
    private INDArray flattenedGradient;

    public DefaultGradient() {}

    public DefaultGradient(INDArray flattenedGradient) {
        this.flattenedGradient = flattenedGradient;
    }

    @Override
    public Map<String, INDArray> gradientForVariable() {
        return gradients;
    }

```

```
    @Override
    public INDArray gradient(List<String> order) {
        List<INDArray> toFlatten = new ArrayList<>();
        if (flatteningOrders == null) {
            for (String s : order) {
                if (!gradients.containsKey(s))
                    continue;
                toFlatten.add(gradients.get(s));
            }
        } else {
            for (String s : order) {
                if (!gradients.containsKey(s))
                    continue;
                if (flatteningOrders.containsKey(s) && flatteningOrders.get(s) !=
DEFAULT_FLATTENING_ORDER) {
                    //Arrays with non-default order get flattened to row vector first, then ev
erything is flattened to f order
                    //TODO revisit this, and make more efficient
                    toFlatten.add(Nd4j.toFlattened(flatteningOrders.get(s), gradients.get(s)))
;
                } else {
                    toFlatten.add(gradients.get(s));
                }
            }
        }
        return Nd4j.toFlattened(DEFAULT_FLATTENING_ORDER, toFlatten);
    }

    private void flattenGradient() {
        if (flatteningOrders != null) {
            //Arrays with non-default order get flattened to row vector first, then
everything is flattened to f order
            //TODO revisit this, and make more efficient
            List<INDArray> toFlatten = new ArrayList<>();
            for (Map.Entry<String, INDArray> entry : gradients.entrySet()) {
                if (flatteningOrders.containsKey(entry.getKey())
                                && flatteningOrders.get(entry.getKey()) !=
DEFAULT_FLATTENING_ORDER) {
                    //Specific flattening order for this array, that isn't the default
                    toFlatten.add(Nd4j.toFlattened(flatteningOrders.get(entry.getKey()), entry
.getValue()));
                } else {
                    //default flattening order for this array
                    toFlatten.add(entry.getValue());
                }
            }
            flattenedGradient = Nd4j.toFlattened(DEFAULT_FLATTENING_ORDER, toFlatten);
        } else {
            //Standard case: flatten all to f order
            flattenedGradient = Nd4j.toFlattened(DEFAULT_FLATTENING_ORDER, gradients.values())
;
        }
    }

    @Override
    public INDArray gradient() {
        if (flattenedGradient != null)
            return flattenedGradient;
        flattenGradient();
        return flattenedGradient;
    }

```

```
    @Override
    public void clear() {
        gradients.clear();
    }

    @Override
    public INDArray getGradientFor(String variable) {
        return gradients.get(variable);
    }

    @Override
    public INDArray setGradientFor(String variable, INDArray newGradient) {
        INDArray last = gradients.put(variable, newGradient);
        // TODO revisit whether setGradientFor should update the gradient that can be pulled from this object in any form - currently does not update flattened
        // use of unitialized var for flattengradient in backprop is generating an error in gradient calc if bellow is used
        //        flattenGradient();
        return last;
    }

    @Override
    public INDArray setGradientFor(String variable, INDArray gradient, Character flatteningOrder) {
        INDArray last = setGradientFor(variable, gradient);

        if (flatteningOrder != null) {
            if (flatteningOrders == null)
                flatteningOrders = new LinkedHashMap<>();
            flatteningOrders.put(variable, flatteningOrder);
        }
        return last;
    }

    @Override
    public Character flatteningOrderForVariable(String variable) {
        if (flatteningOrders == null)
            return null;
        return flatteningOrders.get(variable);
    }


    @Override
    public String toString() {
        return "DefaultGradient{" + "gradients=" + gradients + (flatteningOrders != null ? flatteningOrders : "") + '}';
    }
}
```

Nd4j.gemm

```
 /** Matrix multiply: Implements c = alpha*op(a)*op(b) + beta*c where op(X) means transpose X (or not)
 * depending on setting of arguments transposeA and transposeB.<br>
 * Note that matrix c MUST be fortran order, have zero offset and have c.data().length == c.length().
 * An exception will be thrown otherwise.<br>
 * Don't use this unless you know about level 3 blas and NDArray storage orders.
 * @param a First matrix
```

```
 * @param b Second matrix
 * @param c result matrix. Used in calculation (assuming beta != 0) and result is stored in this. f order,
 *          zero offset and length == data.length only
 * @param transposeA if true: transpose matrix a before mmul
 * @param transposeB if true: transpose matrix b before mmul
 * @return result, i.e., matrix c is returned for convenience
 */
public static INDArray gemm(INDArray a, INDArray b, INDArray c, boolean transposeA, boolean transposeB,
                double alpha, double beta) {
    getBlasWrapper().level3().gemm(a, b, c, transposeA, transposeB, alpha, beta);
    return c;
}
```