

УДК 004.853

БИБЛИОТЕКА ГЛУБИННОГО ОБУЧЕНИЯ DEEPLARNING4J

А.А. СОЛОВЬЁВ

(Представлено: канд. физ.-мат. наук, доц. Д.Ф. ПАСТУХОВ)

Анализируется библиотека с открытым исходным кодом, распределенная библиотека глубинного обучения для виртуальной машины Java (JVM) *Deeplearning4J (DL4J)* на примере архитектуры сверточной нейронной сети, представленной Алексом Крижевским (AlexKrizhevsky) на конкурсе ImageNet в 2012 году.

Стремительное развитие технологий машинного обучения делает спрос на различные инструменты, позволяющие обучать архитектуры нейронных сетей любой сложности, а также тонко настраивать различные параметры при обучении. Одним из современных инструментов машинного обучения является библиотека DL4J.

DL4J – это первая коммерческая библиотека глубинного обучения с открытым исходным кодом, написанная для Java и Scala [1]. Данная библиотека позволяет строить архитектуры нейронных сетей различной сложности и глубины, тонко настраивать всевозможные параметры обучения. Библиотека предназначена для использования в бизнес-средах на распределенных графических и центральных процессорах, включает в себя реализацию ограниченной машины Больцмана, глубокой сети убеждений, глубокого автокодера, многоуровневого автокодирования и рекурсивной сети нейронных тензоров, word2vec, doc2vec и GloVe. Все эти алгоритмы включают распределенные параллельные версии, которые интегрируются с ApacheHadoop и Spark [2].

Для анализа данной библиотеки рассмотрим архитектуру сверточной нейронной сети AlexNet, которая заняла первое место в конкурсе ImageNet в 2012 году [3].

Программная реализация данной архитектуры сверточной нейронной сети при помощи библиотеки DL4J представлена следующим образом:

```
01: MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
02:   .seed(seed)
03:   .weightInit(WeightInit.DISTRIBUTION)
04:   .dist(new NormalDistribution(dist))
05:   .activation(Activation.RELU)
06:   .updater(Updater.NESTEROVS).momentum(momentum)
07:   .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
08:   .iterations(iterations)
09:   .gradientNormalization(GradientNormalization.RenormalizeL2PerLayer)
10:   .learningRate(learningRate)
11:   .regularization(true).l2(l2)
12:   .list()
13:   .layer(0, new ConvolutionLayer.Builder(kernelSize, stride, padding)
           .nIn(in).nOut(out).build())
14:   .layer(1, new LocalResponseNormalization.Builder().build())
15:   .layer(2, new SubsamplingLayer.Builder(kernelSize, stride).build())
16:   .layer(3, new ConvolutionLayer.Builder(kernelSize, stride, padding)
           .nIn(in).nOut(out).build())
17:   .layer(4, new LocalResponseNormalization.Builder().build())
18:   .layer(5, new SubsamplingLayer.Builder(kernelSize, stride).build())
19:   .layer(6, new ConvolutionLayer.Builder(kernelSize, stride, padding)
           .nIn(in).nOut(out).build())
20:   .layer(7, new ConvolutionLayer.Builder(kernelSize, stride, padding)
           .nIn(in).nOut(out).build())
21:   .layer(8, new ConvolutionLayer.Builder(kernelSize, stride, padding)
           .nIn(in).nOut(out).build())
22:   .layer(9, new SubsamplingLayer.Builder(kernelSize, stride).build())
23:   .layer(10, new DenseLayer.Builder().nOut(out).dropOut(dropOut).build())
```

```
24: .layer(11, new DenseLayer.Builder().nOut(out).dropOut(dropOut).build())
25: .layer(12, new OutputLayer.Builder(LossFunctions.NEGATIVELOGLIKELIHOOD)
    .nOut(out).activation(Activation.SOFTMAX).build())
26: .backprop(true)
27: .setInputType(InputType.convolutional(height, width, channels))
28: .build().
```

Рассмотрим данный программный код подробнее. В первой строке программного кода происходит объявление и инициализация многослойной нейронной сети. Создается объект класса `MultiLayerConfiguration`, которому присваивается значение нового объекта класса `NeuralNetConfiguration`. Данный класс реализует методы конфигурирования нейронных сетей, которые охватывают параметры каждого слоя.

Далее устанавливаются необходимые нам параметры нейронной сети. Метод `seed` принимает в себя вещественное число, которое используется при инициализации весов нейронной сети. Инициализация весов происходит по определенному закону, который устанавливается в методе `weightInit`. Тип закона устанавливается в методе `dist`. В нашем случае инициализация весов происходит по нормально-распределенному закону.

Также необходимо указать функцию активации для скрытых слоев нейронов. Для этого используется метод `activation`, который в качестве параметра принимает вид функции. В настоящее время огромной популярностью пользуется выпрямленная линейная функция активации `RELU`. Использование данной функции приводит к прореживанию весов, что ограждает нейронную сеть от переобучения.

После в методе `updater` указывается механизм обновления весов. В большинстве случаев хорошим выбором является использование алгоритма оптимизации стохастического градиентного спуска `NESTEROVS` [1]. Метод `momentum` является дополнительным методом, который отвечает за то, насколько быстро алгоритм оптимизации сходится в оптимальной точке.

Метод `iterations` принимает целое число, которое указывает на количество итераций для одной эпохи обучения сети.

Градиентная нормализация (`gradientNormalization`) применяется во избежание проблем малых и больших градиентов.

В методе `learningRate` указывается скорость обучения нейронной сети. Данный параметр является одним из важнейших при обучении сети. Высокая скорость обучения сети означает, что в обновлениях веса будут делаться крупные шаги, поэтому образцу может потребоваться меньше времени, чтобы набрать оптимальный набор весов. Но слишком высокая скорость обучения может привести к крупным и недостаточно точным скачкам, которые помешают достижению оптимальных показателей.

В следующем методе мы указываем нашей сети, что будем использовать регуляризацию.

Регуляризация помогает избежать переобучение нейронной сети. В нашем случае мы используем `L2`-регуляризацию. Регуляризация `L2` накапливает большие весовые коэффициенты сети и предотвращает слишком большой вес. Общие значения для `L2`-регуляризации составляют от $1e-3$ до $1e-6$.

Функция `list` позволяет создать многослойную нейронную сеть. В архитектуре рассматриваемой сети находится: 13 слоев, из которых 5 сверточных слоев; 3 слоя субдискретизации; 2 нормализации; 2 полносвязных слоя и выходной слой.

Метод `layer` принимает два параметра – номер слоя и объект класса `Layer`. Рассматриваемые далее классы слоев наследуются от класса `Layer`.

Класс `ConvolutionLayer` реализует метод `builder`, в котором устанавливаются такие параметры, как размер ядра свертки, шаг свертки и отступ. Также класс реализует методы `nIn` и `nOut`, в которых устанавливаются количество входных и выходных нейронов; метод `build` – конфигурирует заданные параметры.

Класс `SubsamplingLayer` также реализует метод `builder`, в котором указываются размер ядра и шаг субдискретизации. Данный класс поддерживает следующие типы субдискретизации: `MAX`, `AVG`, `NON`. Слои данного типа предназначены для уменьшения размерности карт признаков, полученных после свертки. Уменьшение размерности карт признаков приводит к сокращению параметров, используемых при обучении, без существенной потери важной информации.

Класс `LocalResponseNormalization` реализует алгоритм нормализации данных.

Класс `DenseLayer` реализует функционал полносвязного слоя прямого распространения. Также в данном классе реализован метод `dropOut`, который принимает вещественное значение, указывающее на процент нейронов, которые будут отброшены во время обучения [4]. Данный параметр позволяет сети не переобучаться.

Класс `OutputLayer` реализует функционал вывода данных с различными объективными совпадениями для разных целей. Данный класс производит классификацию, а также регрессию. Метод `Builder`

данного класса принимает в качестве аргумента некоторую функцию. В нашем случае это функция потерь отрицательного логарифмического правдоподобия.

Метод `backprop` реализует механизм обратного распространения ошибки, в качестве параметра принимает значение типа `bool`, которое указывает, использовать ли данный метод при обучении сети.

В методе `setInputType` устанавливается тип нейронной сети (в нашем случае сверточная), а также размер и количество каналов входных изображений.

Метод `build` собирает описанную выше нейронную сеть.

Заключение

В данной работе были рассмотрены лишь некоторые возможности библиотеки DL4J и проанализированы основные моменты при построении сверточной нейронной сети. Достоинствами данной библиотеки являются хорошая документация классов, методов и алгоритмов, огромное количество всевозможных методов обучения нейронных сетей, а также то, что данная библиотека написана для виртуальной машины Java. В библиотеке содержится большое количество методов конфигурирования нейронных сетей, однако основополагающими методами являются методы инициализации весов (`weightInit`), установки функции активации (`activation`) и скорости обучения сети (`learningRate`), а также конфигурирования слоев (`layer`) и способа их обучения. Также стоит отметить, что DL4J – это библиотека с открытым исходным кодом, что позволяет свободно использовать и распространять исходные коды с использованием данной библиотеки.

ЛИТЕРАТУРА

1. Библиотека глубинного обучения DeepLearning4J [Электронный ресурс]. – Режим доступа: <https://deeplearning4j.org/>. – Дата доступа: 23.09.2017.
2. Библиотека с открытым исходным кодом DeepLearning4J [Электронный ресурс]. – Режим доступа: <https://ru.wikipedia.org/wiki/Deeplearning4j/>. – Дата доступа: 24.09.2017.
3. Krizhevsky, A. Imagenet classification with deep convolutional neural networks / Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton // NIPS. – 2012. – 1106–1114 p.
4. Dropout: A Simple Way to Prevent Neural Networks from Overfitting / Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever // Journal of Machine Learning Research. – 2014. – (15). – P. 1929–1958.