Введение

Глубокие нейронные сети в настоящее время становятся одним из самых популярных

методов машинного обучения. Они показывают лучшие результаты по сравнению с

альтернативными методами в таких областях, как распознавание речи, обработка

изображений и т.д. Главной причиной успешного применения глубоких нейронных сетей является то, что сеть

автоматически выделяет из данных важные признаки, необходимые для решения задачи. Это обстоятельство является важным преимуществом нейронных сетей при обработке больших объемов данных.

Рекуррентные нейронные сети- вид нейронных сетей, где связи между элементами образуют направленную последовательность. Благодаря этому появляется возможность обрабатывать серии событий во времени или последовательные пространственные цепочки. В отличие от многослойных перцептронов, рекуррентные сети могут использовать свою внутреннюю память для обработки последовательностей произвольной длины. Поэтому сети RNN применимы в таких задачах, где целостное разбито на определенные сегменты(например, поиск ошибок в логах программ или распознование человеческой речи).

В данном отчете представлен краткий обзор основных видов глубоких рекуррентных нейронных сетей, представленных во фреймворке Keras, а также представлен пример работы рекуррентных нейронных сетей на примере задачи поиска аномалий в логах данных.

Глава 1

Фреймворк Keras

В качестве среды разработки применялся фреймворк Keras. Это фреймворк поддержки глубокого обучения на Python, обеспечивающий удобный способ создания и обучения практически для любых моделей глубокого обучения.

Данный фреймворк обладает следующими ключевыми характеристиками, которые и обусловили выбор данного фреймворка для решения задач:

-возможность выполнять один и тот же код на CPU или на GPU;

-наличие встроенного API, упрощающего разработку прототипов моделей глубокого обучения;

-наличие встроенной поддержки рекуррентных и сверточных сетей;

-наличие поддержки произвольных сетевых архитектур, например, моделей с множественными входами и выходами, совместного использования моделей и слоев и т.д.

Таким образом, Keras подходит для создания практически любых моделей машинного обучения. Он также используется в крупных компаниях, таких как Google, Netflix, Uber для решения широкого круга задач.

Tensorflow

Keras является высокоуровневой библиотекой, не реализующей низкоуровневые операции, такие, как манипуляции с тензорами и дифференцирования. Для решения этих задач используется специализированные библиотеки поддержки тензоров. Keras поддерживает множество таких библиотек. Одной из них является Tensorflow, разработанный в компании Google, являющийся наиболее распространенной, масштабируемой и высококачественной платформой для манипуляций с тензорами. Любой код, использующий Keras, можно запускать с любой из этих библиотек без необходимости менять что-либо в коде. Если одна из библиотек показывает более высокую производительность во время работы, то в процессе разработки можно переключиться на нее.

Рекуррентные нейронные сети (Recurrent Neural Network, RNN) — класс моделей машинного обучения, основанный на использовании предыдущих состояний сети для вычисления текущего [5; 10]. Такие сети удобно применять в тех случаях, когда входные данные задачи представляют собой нефиксированную последовательность значений, как, например, текстовые данные, где текстовый фрагмент представлен нефиксированным количеством предложений, фраз и слов. Каждый символ в тексте, отдельные слова, знаки препинания и даже целые фразы — все это может являться атомарным элементом входной последовательности.

Рекуррентные нейронные сети обрабатывают входные последовательности по одному интервалу за раз, поддерживая состояние(в данной ситуации-это вектор) на всем его протяжении.

Во фреймворке Keras программно реализовано три рекуррентных слоя соответствующих сетей: SimpleRNN,GRU и LSTM . Рассмотрим каждый из рекуррентных слоев по отдельности.

1. SimpleRNN

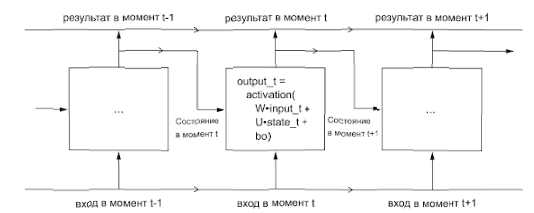
Данный слой является наиболее простым для использования во фреймворке Керас. Он обрабатывает пакеты последовательностей, как и другие слои в этом фреймворке, например, слой Embedding. Это означает, что входные данные у данного слоя будут иметь вид: (размер\_пакета, временные\_интервалы, входные\_призраки). SimpleRNN действует в двух разных режимах:

-возвращает полные последовательности результатов для всех временных интервалов (трехмерный тензор). Для выбора этого режима нужно подать в рекуррентный слой аргумент return\_sequences=true;

-последний результат для каждой входной последовательности (двухмерный тензор с формой(размер\_пакета, входные\_признаки).

На рисунке 1 описан пример ячейки SimpleRNN.

Данный слой является наиболее быстрым в использовании для решения практических задач. Однако он обладает существенной проблемой затухания градиента, что делает невозможной ее использования для больших объемов данных. Суть этой проблемы заключается в том, что по мере увеличения количества слоев нейронная сеть становится необучаемой.

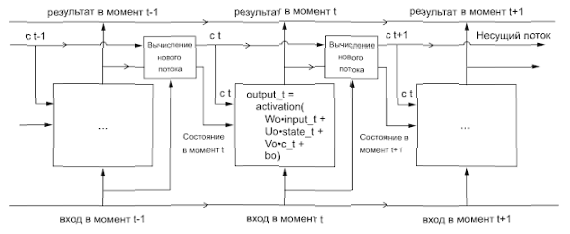


2. LSTM

В основе данного рекуррентного слоя лежит алгоритмы долгой краткосрочной памяти(LSTM), разработанный в 1997 году. Данный алгоритм был разработан для решения проблемы затухания градиента, актуальной для метода SimpleRNN.

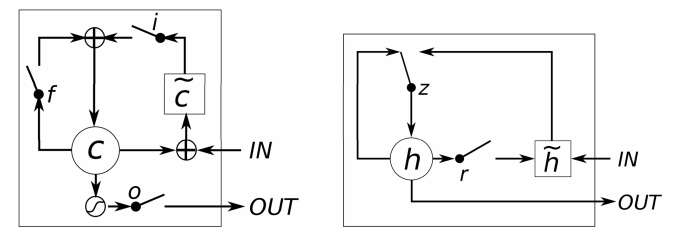
Данный слой является вариацией предыдущего метода, в котором была добавлена поддержка переноса информации через интервалы времени. Вычисление следующего значения в несущем потоке данных основывается на трех фильтрах(входном i, забвения f и выходном o), имеющих форму ячейки SimpleRNN и собственные весовые матрицы. Схема ячейки LSTM описана на рисунке 2.

Модель ЛСТМ, в отличие от модели SimpleRNN, менее подвержено проблеме затухания градиента, но оно более трудоемко в использовании для вычислительных машин. Однако вся сила ЛСТМ проявляется в решении более сложных задач, например, задачи разпознавания образов.



3. GRU

Метод управляемых рекуррентных блоков был разработан в 2014 году. Он основан на том же принципе, что и слои LSTM, но в отличие от этого метода она использует меньше фильтров (используется два фильтра: обновления z и сброса r), что влияет на количестве нелинейностей, которое приходит от входных данных и конечном итоге влияет на порядок вычислений. Основные различия между этими рекуррентными сетями показано на рисунке 3. Метод GRU реализует в первую очередь компромисс между затратностью вычислений и результативной мощностью, что является немаловажным в области машинного обучения.



Решение поставленной задачи

В рамках учебной дисциплины была поставлена задача поиска аномалий на основе логов. Разработанный код должен находить аномалии в этих логах, такие, как нетипичные типы действий, нетипичное время действия и т.д. Поиск аномалий должен производиться с помощью рекуррентных нейронных сетей.

С целью решения данной задачи был разработан программный код с использованием фреймворка Keras и рекуррентных нейронных сетей SimpleRNN, GRU, LSTM. В этой главе будет описан процесс разработки решения для поставленной задачи.

Преобразование данных

В качестве лог-файла был взят файл “http.csv”, содержащий в себе следующие поля:

1. Id-идентификатор компьютера
2. Date-дата посещения веб-страницы
3. User-идентификатор пользователя системы.
4. url-сайт, на который зашел пользователь.
5. Activity-тип действия
6. Content-содержание веб-страницы.

Из этих параметров на аномалию в системе влияет совокупность данных, состоящая из пользователя системы, сайта, на который он зашел и типа действия. Поэтому при считывании данных из файла требуется выбрать нужные нам поля, в данном случае-user,url,activity.

Поле url требуется преобразовать таким образом, чтобы в поле оставалось только значение домена, на который заходит пользователь системы.

Для обучения системы требуется произвести манипуляцию данными таким образом, чтобы они были разделены на два тензора: входной и целевой. Входным тензором являются данные, на которых происходит дальнейшее обучение нейронных сетей. Целевым тензором является переменная с некоторым количеством классов.

В данном случае за целевой тензор была взята совокупность полей user и url для каждой строки файла, предварительно объединенная в единую комбинацию load-data, а за целевой тензор- поле activity, которое может принимать одно из трех значений: WWWVisit, WWWUpload или WWWDownload.

Программный код для преобразования данных описан ниже.

def get\_url(url):

return urlparse(url).hostname

df = pd.read\_csv("r6.2/http.csv",nrows=10000,usecols=["user", "url", "activity"],)

df['url'] = df['url'].apply(get\_url)

df['activity'] = df['activity'].apply(delete\_slash)

df['load-data'] = df['user']+" "+ df['url']

del(df['url'])

del(df['user'])

ds = df.sample(frac=1).values

print(ds)

X=ds[:,1]

Y=ds[:,0]

Преобразование строк в кодовые значения

Модель обучения принимает тензоры в виде значений типа float. Однако наши данные представлены в виде значений строкового типа. Для решения данной проблемы воспользуемся встроенной утилитой фреймворка Keras под названием Tokenizer. Данная утилита использует прямое кодирование слов(one-hot encoding). Этот метод состоит в присваивании каждому слову уникального целочисленного индекса i с последующим его преобразованием в бинарный веткор размера словаря, все элементы которого содержат нули, кроме i-го элемента, которому присваивается 1. Для каждого слова присваивается определенный индекс, который зависит от количества вхождений слова в список. В частности, наиболее часто встречающееся слово будет иметь индекс 1, а наименее встречающееся слово-индекс N, где N-количество слов в словаре. Для получения индекса всех слов пользуемся методом fit\_on\_texts утилиты Tokenizer, для преобразования строк в списки целочисленных элементов- методом texts\_to\_sentences.

Для нормализации и преобразования входного тензора в тип с плавающей точкой воспользуемся методом pad\_sequences утилиты sequence, которая дополняет нулями слева до определённого значения количество элементов каждого участка тензора, а также преобразует данные в нужный вид для обучения.

Для векторизации целевого тензора воспользуемся методом to\_categorical утилиты np\_utils, позволяющей преобразовать данные списка на N+1 элементов, где N-максимальное значение преобразуемого списка.

t = Tokenizer()

t2 = Tokenizer()

t.fit\_on\_texts(X)

encoded = t.texts\_to\_sequences(X)

num\_words = len(t.word\_index)+1

max\_len=10

encoded = sequence.pad\_sequences(encoded , maxlen=max\_len, dtype='double')

t2.fit\_on\_texts(Y)

encoded2 = t2.texts\_to\_sequences(Y)

encoded2 = [item for sublist in encoded2 for item in sublist]

nb\_classes=4

encoded2=np\_utils.to\_categorical(encoded2, nb\_classes)

Обучение

Для обучения нейронных сетей требуется в начале разделить данные на тренировочные и тестовые. Система обучается на тренировочных данных, а затем проверяет на тестовых данных процент точности.

Для построения нейронных сетей применим встроенную во фреймворк Keras модель Sequential, которая основана на предположении, что нейронная сеть имеет только один вход и один выход и состоит из линейного стека слоев.

Исходная модель, на которой проходит обучение, состоит из трех слоев:

1. Слой Embedding. Данный слой является слоем весов обучения. Его лучше всего воспринимать как словарь, который отображает целочисленные индексы, обозначающие конкретные слова, в плотные векторы. Слой принимает минимум два аргумента: количество возможных токенов, например, количество слов в словаре и размерность пространства. Также можно подать на вход слою дополнительные параметры, например, длину входного пространства.
2. Слой нейронной сети. Данный слой реализует одну из рекуррентных сетей, в частности, SimpleRNN, LSTM и GRU. Как уже было описано ранее, на вход данные слои принимают три параметра: размер пакета, временные интервалы, входные признаки. Последний параметр является обязательным для ввода в каждой из моделей.
3. Слой Dense организован как простой стек полносвязных слоев с различными операциями активации. Аргумент N, передаваемый каждому слою Dense-это число скрытых нейронов слоя. Скрытый нейрон-это измерение в пространстве представлений слоя. Наличие N скрытых нейронов означает, что весовая матрица W будет иметь форму (input\_dimension, N): скалярное произведение на W спроецирует входные данные в 16-мерное пространство представлений. Затем будет произведено сложение с вектором смещений b, а также выполнена операция активации для последнего уровня, которая устанавливает эффективные ограничения на результат сети. Так, чтобы решить задачу, бинарной классификации, нужно выбрать функцию активации sigmoid, а для задачи многоклассовой однозначной спецификации- функцию softmax. Решаемая нами задача является задачей бинарной классификации.

После построения модели мы проводим ее компиляцию. Для компиляции модели требуется выбрать функцию потерь и оптимизатор. Функция потерь — это величина, которую требуется свести к минимуму в ходе обучения, поэтому она должна представлять собой меру успеха для решаемой задачи. Так как решаемая задача является задачей бинарной классификации и результатом работы сети является вероятность, то предпочтительнее всего использовать функцию потерь binary\_crossentropy. Можно выбрать также функцию mean\_squared\_error, однако она дает менее качественные результаты в случае нахождения вероятностей, в отличие от метода перекрестной энтропии.

Оптимизатор определяет точный способ использования градиента потерь для изменения параметров. Например, оптимизатор RMSProp реализует метод градиентного спуска с импульсом. Для решения данной задачи мы применили оптимизатор Adam, также основанный на градиентном спуске.

После компиляции модели можно начинать обучение сети. В случае использования библиотеки Keras достаточно вызвать метод fit сети-он пытается адаптировать модель под обучающие данные. В процессе обучения отображаются две величины: потери сети на обучающих данных и точность сети на обучающих данных. Например, для построенной модели LSTM точность составила 98 %. После мы проверяем контрольный набор:

scores = model.evaluate(X\_test,Y\_test, batch\_size=32)

print("Accuracy of LSTM: %.2f%%" % (scores[1]\*100))

Точность на контрольном наборе составляет 96.82 %, что отличается от тренировочного набора, что обусловлено переобучением данных, когда модели машинного обучения показывают худшую точность на новом наборе данных по сравнении с тренировочным.

После тестовой проверки сохраняем модель. Архитектура построенной модели сохраняется в файл формата .json, а весы построенной рекуррентной сети- в формате HDF5(.h5).

Ниже приведен пример кода для построения модели, содержащей в себе слой реккурентной нейронной сети типа LSTM.

embedding\_vecor\_length = 10

nb\_classes=4

model = Sequential()

model.add(Embedding(num\_words, embedding\_vecor\_length, input\_length=10))

model.add(LSTM(num\_of\_cell))

model.add(Dense(nb\_classes, activation='softmax'))

model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])

print(model.summary())

model.fit(X\_train,Y\_train,nb\_epoch=200,batch\_size=128)

scores = model.evaluate(X\_test,Y\_test, batch\_size=32)

print("Accuracy of LSTM: %.2f%%" % (scores[1]\*100))

#saving the model

model\_json = model.to\_json()

with open("lstm-model.json", "w") as json\_file:

json\_file.write(model\_json)

# serialize weights to HDF5

model.save\_weights("lstm-model.h5")

print("Saved model to disk")

В таблице 1 показана точность рекуррентных нейронных сетей на обучающих и тестовых данных.

Таблица 1

Результат тестирования рекуррентных нейронных сетей для тестовых данных(10000 данных, 200 проходов)

|  |  |  |
| --- | --- | --- |
|  | Точность на обучающих данных | Точность на тестовых данных |
| LSTM | 98.0 % | 96.82 % |
| SimpleRNN | 98.05 % | 96.77 % |
| GRU | 98.08 % | 96.90 % |

В случае увеличения количества данных точность сети может увеличиться, однако для всех методов возможна проблема затухания градиента(в первую очередь, для метода SimpleRNN, но это также касается и двух других методов).

Количество данных и проходов было выбрано целесообразно этой задаче с целью демонстрации различных подходов рекуррентных нейронных сетей. В случае большего количества целевых данных рекомендуется использование рекуррентных сетей LSTM и GRU.

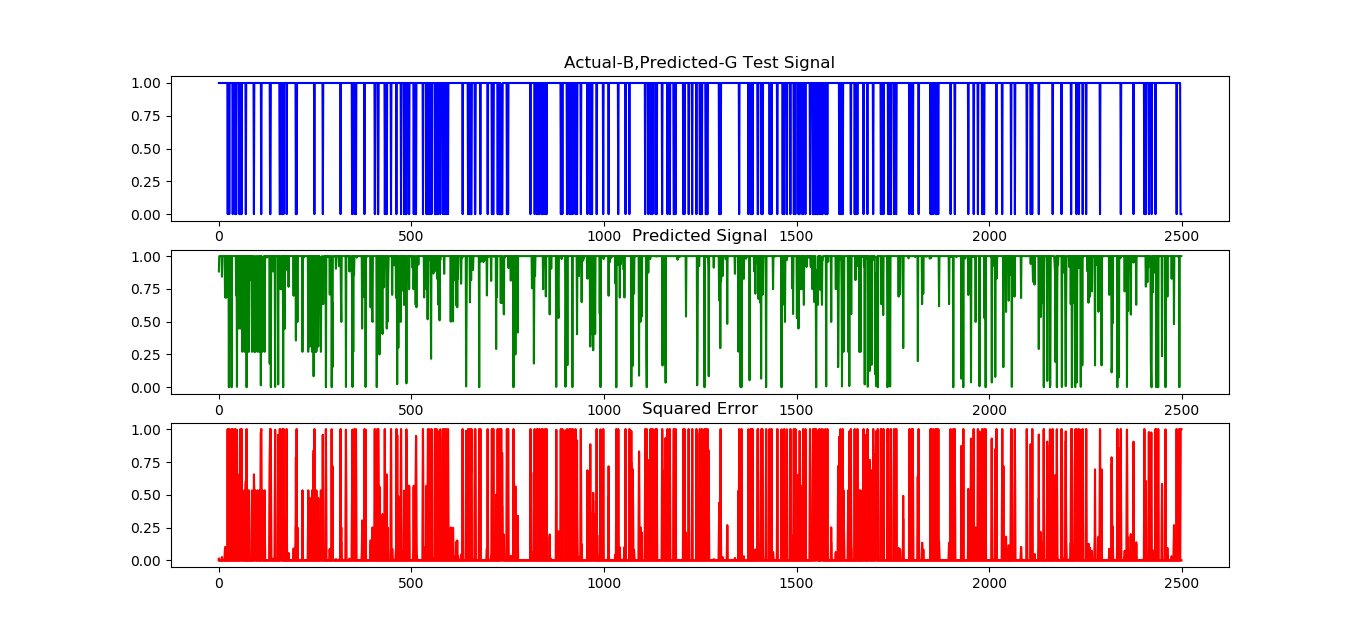
Тестирование данных на конкретных примерах

Теперь требуется произвести тестирование данных на конкретных примерах. В качестве тестовых данных был взят специально написанный лог, содержащий в себе примеры очевидных аномалий, такие, как несколько скачиваний подряд или загрузка контента на сайт, с которого данный пользователь обычно скачивает информацию.

Для того, чтобы привести лог к виду, требуемому для использования в модели, надо выполнить манипуляции с данным логом (см. пункты «Преобразования данных» и «Преобразование строк в кодовые значения» отчета). Перед запуском функции предсказания значений действия пользователя в системе требуется загрузить модель из файлов, в которых она хранится.

Для предсказания вероятности конкретного результата используется функция predict. Эта функция возвращает вероятности для каждого конкретного значения класса целевого тензора. Например, если количество классов целевого тензора равно 4, то результатом функции predict станет лист размером Nx4, где N-количество целевых данных.

Для каждого класса строим график с помощью модуля matplotlib. Данный график состоит из трех элементов: графика вероятности, предсказанной моделью, графика реальной модели и графика среднеквадратичной ошибки для данного класса. С помощью этого графика можно определить, какой класс наиболее вероятен для данного Пример подобного графика для класса WWWVisit описан на рисунке 4.



Для нахождения аномалий был написан специальный алгоритм. На вход подаются списки вероятностей, предсказанных моделью и список реальных вероятностей. Для каждого элемента двух списков находится индекс максимального элемента. В случае, если индексы максимальных элементов соответствующих элементов из двух списков не совпадает, то алгоритм выявляет аномалию.

В случае нахождения аномалии она заносится в словарь, где ключом является комбинация кодировок слов целевого тензора, а значением- количество аномалий, выявленных для комбинации пользователя и сайта, который он посещал.

Таблица 2 показывает количество аномалий, найденных для каждой из рекуррентной сети.

Исходя из этой таблицы, мы можем сделать вывод, что наиболее удачно находят аномалии рекуррентные сети типа GRU. Нейронная сеть SimpleRNN хоть и нашла примерно столько же аномалий, сколько и GRU, но ее предсказания были неточными в большинстве случаев, что делает ее менее способной к нахождению реальных аномалий, нежели сети LSTM и GRU.

Таблица 2

Количество аномалий в логах, найденных с помощью рекуррентных сетей

|  |  |
| --- | --- |
| LSTM | 305 |
| SimpleRNN | 378 |
| GRU | 373 |

Заключение

В результате работы были изучены основные виды рекуррентные сети, используемые в работе фреймворка Keras, а также приведены основные преимущества различных типов рекуррентных нейронных сетей на примере конкретной задачи.

В настоящее время рекуррентные нейронные сети переживают бум своего развития. Основными трендами стали сочетание различных архитектур и применение наработок из других областей для их улучшения. Из примеров можно назвать уже упомянутые нейросети от Google, в которых используют методы, взятые из работ по обучению с подкреплением, нейронные машины Тьюринга, алгоритмы оптимизации вроде Batch Normalization и многое другое, — все это вместе заслуживает отдельной статьи. В целом отметим, что хотя рекуррентные сети не привлекли столь же широкого внимания, как сверточные нейросети, это объясняется лишь тем, что объекты и задачи, с которыми работают рекуррентные сети, не так бросаются в глаза, как приложения, основанные на сверточных сетях, например, DeepDream или Prisma.

Список использованных источников

1. Будыльский Д. В. GRU и LSTM: современные рекуррентные нейронные сети // Молодой ученый. — 2015. — №15. — С. 51-54. — URL https://moluch.ru/archive/95/21426/ (дата обращения: 22.05.2018).
2. Шолле, Ф. Глубокое обучение на Python // СПб.- 2018. – 400 с.
3. LSTM: A Search Space Odyssey / K. Greff [et al.] // arXiv preprint arXiv:1503.04069. — 2015.
4. Multi-resolution linear prediction based features for audio onset detection with bidirectional LSTM neural networks / E. Marchi [et al.] // Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on. — IEEE. 2014. — Pp. 2164–2168.