РЕКУРРЕНТНЫЕ НЕЙРОННЫЕ СЕТИ: МОДЕЛИРОВАНИЕ ПОСЛЕДОВАТЕЛЬНОСТЕЙ

Методические указания
к выполнению практических работ
для студентов 2-го курса магистратуры, обучающихся по образовательным программам
направления 09.04.03 «Прикладная информатика» всех форм обучения

Воронеж 2022

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«Воронежский государственный технический университет»

Кафедра инноватики и строительной физики имени профессора И.С. Суровцева

РЕКУРРЕНТНЫЕ НЕЙРОННЫЕ СЕТИ: МОДЕЛИРОВАНИЕ ПОСЛЕДОВАТЕЛЬНОСТЕЙ

Методические указания
к выполнению практических работ
для студентов 2-го курса магистратуры, обучающихся по образовательным программам
направления 09.04.03 «Прикладная информатика» всех форм обучения

Воронеж 2022

УДК 519.85(07)
ББК 22.18.я73

Составители:
П. А. Головинский, А. О. Шаталова, Еникеев Э.И.

РЕКУРРЕНТНЫЕ НЕЙРОННЫЕ СЕТИ: МОДЕЛИРОВАНИЕ ПОСЛЕДОВАТЕЛЬНОСТЕЙ: методические указания к выполнению практических работ по дисциплине «Глубокое обучение с Python» для студентов 2-го курса магистратуры, обучающихся по образовательной программе «Технологии искусственного интеллекта» направления 09.04.03 «Прикладная информатика» /ФГБОУ ВО «Воронежский государственный технический университет»; сост.: П.А. Головинский, А.О. Шаталова, Э.И. Еникеев. - Воронеж, 2022. – 30 с.

Содержат комплекс требований к содержанию, порядку выполнения и оформлению практических работ студентов магистрантов в соответствии с перечнем общекультурных, общепрофессиональных и профессиональных компетенций, осваиваемых в процессе обучения. Описана структура индивидуальных заданий для самостоятельного выполнения по дисциплине «Глубокое обучение с Python».

Предназначены для студентов, обучающихся по образовательной программе «Технологии искусственного интеллекта» направления 09.04.03 «Прикладная информатика» всех форм обучения.

Ил. 5. Библиогр.: 11 назв.

УДК 519.85(07)
ББК 22.18.я73

Печатается по решению редакционно-издательского совета Воронежского государственного технического университета

***Рецензент** – С. А. Баркалов, д.т.н., профессор кафедры управления строительством Воронежского государственного технического университета*

ВВЕДЕНИЕ

В самых различных практически важных задачах встречается необходимость обработки последовательностей того или иного вида данных. Это могут быть текстовые или звуковые фразы из слов, последовательные видеокадры или временные зависимости (временные ряды) котировок валют. В любом случае, нужно уметь найти в таком потоке данных нужную информацию, и дать ее прогноз на ближайшее будущее. Одним из способов встраивания в работу нейронной сети информации о последовательной цепочке данных является образование обратной связи в нейронной сети. Обратная связь может присутствовать в нейронной сети в виде локальной обратной связи, т.е. на уровне одного нейрона, либо в форме глобальной обратной связи, которая охватывает всю сеть.

Сети с одной или несколькими обратными связями называются рекуррентными. Наличие обратных связей позволяет учитывать, наряду с текущей поступающей информацией, результаты обработки данных на предыдущих шагах. Обратная связь дает возможность сделать нейронную сеть более гибкой, но может привести к потере устойчивости. Устойчивость сети означает, что близкие входные данные приводят к близким выходным данным. Если этого свойства нет, то небольшие изменения во входных данных приведут к абсолютно иному результату, что в большинстве случаев является нежелательным эффектом, ограничивая множество возможных рекуррентных сетей только устойчивыми.

Изменяющиеся во времени состояния нейронных систем исследуются методами нейродинамики. Нейродинамические системы описываются дифференциальными или разностными уравнениями. Они учитывают наличие большого количества связанных между собой нейронов, нелинейность их отклика, диссипативность (затухание без внешних воздействий), а также влияние внутренних шумов системы.

1. МОДЕЛЬ ХОПФИЛДА

В отличие от обычной статической модели нейрона, в динамической сети функционально задается не выходной потенциал (сигнал) нейрона, а скорость его изменения или правило изменения на каждом шаге алгоритма. Без внешних связей потенциал убывает со скоростью пропорциональной величине самого потенциала, поэтому он уменьшается по экспоненциальному закону, подобному закону радиоактивного распада. Наличие связей между нейронами и внешних сигналов приводит к сложной динамике, позволяющей производить различную обработку информации. Основой для формирования модели в такой сети является ассоциативная связь. Под ассоциацией мы понимаем установление устойчивой связи между двумя одновременными возбуждениями в сети, обладающими достаточной интенсивностью и длительностью. В качестве примера мы укажем полносвязную динамическую систему нейронов Хопфилда, описываемую системой уравнений

$$\frac{dy_i(t)}{dt} = -y_i(t) + \sigma(u_i) + K_i, u_i = \sum_j w_{ij} y_j \,. \qquad (1)$$

Здесь переменная $y_i(t)$ описывает величину отклика нейрона с номером i. В левой части уравнения (1) стоит скорость изменения этого отклика. В правой части уравнения (1) находятся три слагаемых, первое из которых обеспечивает затухание сигнала в отсутствие внешних связей. Функция $\sigma(u_i)$ учитывает влияние остальных нейронов, поскольку

$$u_i = \sum_j w_{ij} y_j(t)\,, \qquad (2)$$

где симметричные веса $w_{ij} = w_{ji}$ подлежат определению в процессе обучения. Чаще всего в качестве функции активации используют логистическую функцию

$$\sigma(u_i) = \frac{1}{1+\exp(-u_i)}\,. \qquad (3)$$

Последнее слагаемое K_i в правой части уравнения (1) учитывает постоянное смещение. Модель Хопфилда является глобально устойчивой (теорема Коэна-Гроссберга), т.е. в результате динамики она стремится к состоянию в некоторой точке многомерного пространства $\bar{\boldsymbol{y}} = \{\bar{y}_i\}$ при $t \to \infty$.

Вопрос, который возникает при использовании сетей Хопфилда для формирования ассоциативных связей, состоит определении надлежащего варианта обучения, т.е. изменения весов w_{ij}. Один из наиболее простых способов обучения состоит в выборе варианта Хебба, когда усиливаются более

возбужденные при подаче обучающих примеров связи. Для непрерывной модели процесс обучения описывается уравнением

$$\frac{dw_{ij}}{dt} = \eta y_j (1 - w_{ij}) - P \sum_{k \neq j} w_{ik} y_k,\ P << \eta. \quad (4)$$

Параметр $P > 0$ управляет пластичностью системы. Такая схема обучения обеспечивает усиление задействованных синаптических путей вплоть до их насыщения. При использовании сетей очень большого размера время вычислений по схеме уравнения (4) становится критически большим, и для его снижения целесообразно пользоваться механизмом внимания, под которым понимается снижение порогов возбуждения релевантных (соответствующих задаче) нейронных групп и увеличение порогов остальных нейронов.

Дискретная версия сети Хопфилда описывается уравнениями

$$y_i(t+1) = \sigma(u_i(t)), \quad (5)$$

а обучение такой модели осуществляется на основе правила

$$w_{ij}(t+1) - w_{ij}(t) = \eta y_j (1 - w_{ij}) - P \sum_{k \neq j} w_{ik} y_k,\ P << \eta. \quad (6)$$

К сожалению, у нейронной сети Хопфилда есть ряд недостатков. Важно, что она имеет относительно небольшой объём памяти, величину которого можно оценить выражением $M \approx \frac{N}{2 \ln N}$. Попытка записи большего числа образов приводит к тому, что нейронная сеть перестаёт их распознавать. Кроме того, достижение устойчивого состояния не гарантирует правильный ответ сети. Это происходит из-за того, что сеть может сойтись к так называемым ложным аттракторам (точками притяжения), склеенным из фрагментов различных образов.

Модель Хопфилда позволяет реализовывать автоассоциативную память, т.е. восстанавливать многомерный сигнал по его приближенному значению. Другим применением сети в форме уравнения (5) является кластеризация данных, поскольку любые исходные данные после ряда описываемых им итераций сходятся к некоторому числу конечных состояний. В результате, последовательная обработка различных сигналов такой нейронной сетью приведет к автоматическому разбиению их на кластеры. Доказано также сильное утверждение о том, что любую машину Тьюринга (любой компьютер) можно имитировать рекуррентной сетью, построенной на нейронах с сигмоидальной функцией активации. Мы изложим основные идеи рекуррентных сетей и приведем простой пример кода. Для более глубокого изучения рекомендуется литература [1-8]. Представленный в методических указаниях код взят из [9,10].

Для запуска кода из учебного пособия и собственных программ на Python предлагается использовать Colaboratory [11]. Данная интерактивная среда (блокнот Colab) позволяет выполнять код Python в браузере. Блокноты Colab

будут храниться на вашем Google Диске. Вы сможете открыть к ним доступ иным лицам, разрешив им просматривать или даже редактировать документ, а также оставлять комментарии.

2. АРХИТЕКТУРА РЕКУРРЕНТНЫХ СЕТЕЙ

В качестве строительного блока рекуррентных сетей можно использовать многослойный персептрон. Поскольку многослойный персептрон содержит несколько слоев, то можно замкнуть связи между различными слоями для построения рекуррентного отображения между входными и выходными последовательностями. Мы рассмотрим несколько типовых архитектур рекуррентных нейронных сетей на базе многослойного персептрона. На рис. 1 показана блочная диаграмма сети, которая называется моделью в пространстве состояний. Нейроны скрытого слоя замкнуты на входной слой через банк единичных задержек. Входной слой сети состоит из объединения узлов обратной связи и узлов источника, по которым поступают внешние сигналы. Количество единичных задержек, используемых для замыкания выхода скрытого слоя на входной слой, определяет порядок модели.

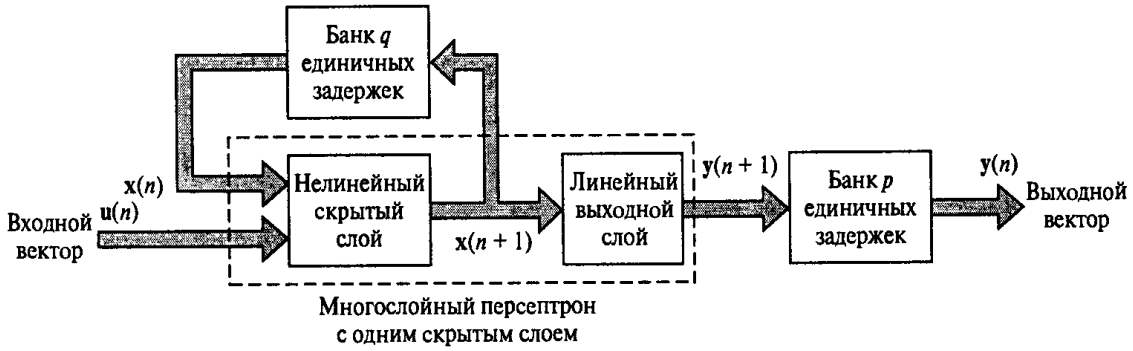

Рис.1. Модель в пространстве состояний

Обозначим символом $\boldsymbol{u}(n)$ вектор входных сигналов длиной m в момент времени n, а $\boldsymbol{x}(n)$ – вектор выходных сигналов скрытого слоя в тот же момент времени. Тогда динамику этой модели можно описать системой уравнений

$$\boldsymbol{x}(n+1) = \boldsymbol{\sigma}(\boldsymbol{x}(n), \boldsymbol{u}(n)), \tag{7}$$

$$\boldsymbol{y}(n) = \boldsymbol{C}\boldsymbol{x}(n), \tag{8}$$

где $\boldsymbol{\sigma}(\cdot,\cdot)$ - нелинейные функции активации, описывающие отклик скрытого слоя, $\boldsymbol{C}$ - матрица синаптических весов, характеризующая связь сигналов скрытого слоя $\boldsymbol{x}$ и выходного слоя $\boldsymbol{y}$. Скрытый слой в этой модели нелинеен, а выходной – линеен.

Похожей архитектурой обладает сеть Эльмана, показанная на рис. 2. В ней отсутствует банк единичных задержек, а выходной слой может быть нелинейным.

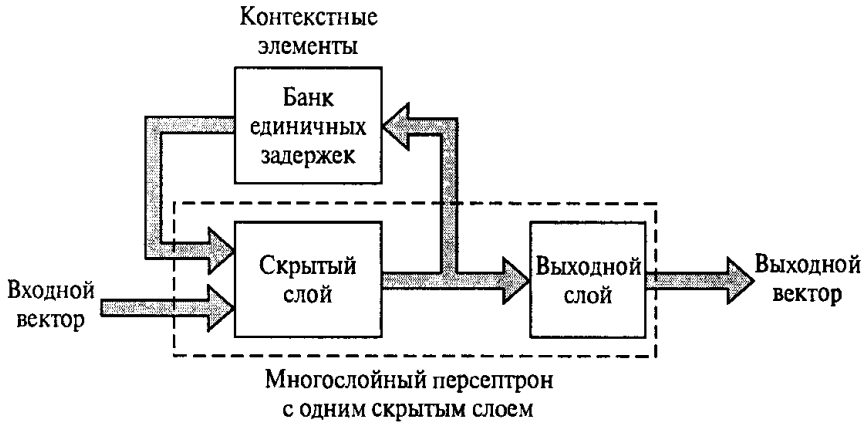

Рис. 2. Простая рекуррентная сеть

Еще одной близкой по архитектуре является рекуррентный многослойный персептрон, показанный на рис. 3.

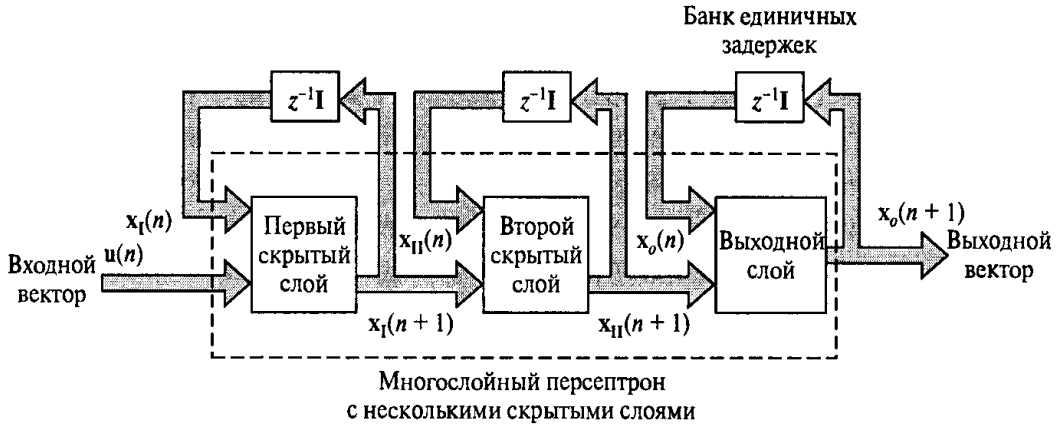

Рис. 3. Рекуррентный многослойный персептрон

Этот персептрон имеет один или несколько скрытых слоев. Каждый слой замкнут сам на себя собственной обратной связью.

Для обучения рекуррентных сетей применяется алгоритм обратного распространения во времени. Он является расширением стандартного алгоритма обратного распространения и может быть получен путем развертывания временных операций сети в виде многослойной сети прямого распространения, топология которой расширяется на один слой для каждого шага времени.

3. СЕТЬ LSTM

LSTM-сеть (сеть долгой краткосрочно памяти) представляет собой искусственную нейронную сеть, содержащую LSTM-модули вместо или в

дополнение к другим модулям сети. LSTM-модуль является рекуррентным и способен запоминать значения, как на короткие, так и на длинные промежутки времени. LSTM-блоки содержат три или четыре «вентиля», которые используются для контроля потоков информации на входах и на выходах памяти данных блоков. Эти вентили реализованы в виде логистической функции для вычисления значения в интервале [0, 1]. Умножение на это значение используется для частичного допуска или запрещения потока информации внутрь и наружу памяти. Например, «входной вентиль» контролирует меру вхождения нового значения в память, а «вентиль забывания» контролирует меру сохранения значения в памяти. «Выходной вентиль» контролирует меру того, в какой степени значение, находящееся в памяти, используется при расчёте выходной функции активации для блока. В некоторых реализациях входной вентиль и вентиль забывания реализуются в виде единого вентиля. Старое значение следует забывать тогда, когда появится новое значение, достойное запоминания.

Веса в LSTM блоке (ячейке) используются для задания направления действия вентилей. Эти веса определены для значений, которые подаются в блок. Таким образом, LSTM-блок определяет, как распоряжаться памятью ячейки через функции ее значений, и настройка весов позволяет обучить LSTM-блок минимизировать ошибку выхода. LSTM-блоки обычно обучают при помощи метода обратного распространения ошибки во времени.

LSTM сети широко используются в системах распознавания речи, в автоматических переводчиках и торговых приложениях. Для понимания работы ячейки LSTM нам, прежде всего, потребуется понять принцип ее работы. Кроме вектора входов $\boldsymbol{x}(t)$ и вектора выходов $\boldsymbol{y}(t)$ в момент времени *t*, ячейка имеет внутреннее состояние, определяемое набором параметров памяти в виде вектора $\boldsymbol{c}(t)$. Внутри ячейки имеется три гейта (вентиля), которые управляют амплитудой забывания состояния, амплитудой его обновления и амплитудой выхода. Для срабатывания ячейки в ней прежде всего нужно определить ее обновленное состояние и на основе полученного значения вычислить выходное значение вектора.

Мы рассмотрим только один из вариантов LSTM. Поскольку ячейка работает рекуррентно, то будем считать, что в некоторый момент времени *t* заданы значения выходов $\boldsymbol{y}(t)$ и состояние ячейки $\boldsymbol{c}(t)$. Вектор-кандидат на новое значение ячейки при подаче на вход нового сигнала $\boldsymbol{x}(t+1)$ вычисляется по формуле

$$\boldsymbol{c}'(t+1) = \tanh(\boldsymbol{W}_{xc}\boldsymbol{x}(t+1) + \boldsymbol{W}_{yc}\boldsymbol{y}(t) + \boldsymbol{b}_c). \qquad (9)$$

Однако новое значение ячейки составляет из старого значения $\boldsymbol{c}(t)$ и вектора-кандидата на новое значение $\boldsymbol{c}'(t)$ в виде линейной комбинации

$$\boldsymbol{c}(t+1) = \boldsymbol{f}(t+1) * \boldsymbol{c}(t+1) + \boldsymbol{i}(t+1) * \boldsymbol{c}'(t+1). \qquad (10)$$

Символ $*$ обозначает точечное произведение векторов, т.е., например, выражение

$\boldsymbol{f}(t+1) * \boldsymbol{c}(t+1)$ обозначает вектор с координатами $f_j(t+1) * c_j(t+1)$. Сами веса $\boldsymbol{f}$, $\boldsymbol{i}$, определяющие забывание и запоминание, и амплитуда выхода вычисляются с помощью сигмоидальной функции активации по формулам

$$\boldsymbol{i}(t+1) = \sigma(\boldsymbol{W}_{xi}\boldsymbol{x}(t+1) + \boldsymbol{W}_{yi}\boldsymbol{y}(t) + \boldsymbol{b}_i), \quad (11)$$

$$\boldsymbol{f}(t+1) = \sigma(\boldsymbol{W}_{xf}\boldsymbol{x}(t+1) + \boldsymbol{W}_{yf}\boldsymbol{y}(t) + \boldsymbol{b}_f), \quad (12)$$

$$\boldsymbol{\theta}(t+1) = \sigma(\boldsymbol{W}_{x\theta}\boldsymbol{x}(t+1) + \boldsymbol{W}_{y\theta}\boldsymbol{y}(t) + \boldsymbol{b}_\theta). \quad (13)$$

Формулы (11) имеют вид обычных нелинейных нейронов, отклик которых зависит от значения входных сигналов $\boldsymbol{x}$ в момент времени t+1 и значения выходного сигнала $\boldsymbol{y}$ в предшествующий момент времени t.

Финальный выходной сигнал вычисляется на основании нового состояния ячейки $\boldsymbol{c}(t+1)$ и управляющего параметра выходного слоя $\boldsymbol{\theta}$ с помощью соотношения

$$\boldsymbol{y}(t+1) = \boldsymbol{\theta}(t+1) * \tanh(\boldsymbol{c}(t+1)). \quad (14)$$

Всюду в уравнениях (9)-(14) для сокращения записи использованы матричные обозначения. Это значит, что выражение $\boldsymbol{W}\boldsymbol{x}$ обозначает произведение матрицы $\boldsymbol{W}$ на вектор $\boldsymbol{x}$. При выполнении этой операции снова получается вектор. Применение скалярной функции σ к вектору означает ее действие последовательно на каждую компоненту, т.е. $\boldsymbol{h} = \sigma(\boldsymbol{b})$ эквивалентно выражению $h_j = \sigma(b_j)$.

4. ВОЗМОЖНОСТИ PYTORCH

Разминка: numpy

Перед тем, как представить PyTorch, мы сначала реализуем нейронную сеть с помощью numpy. Numpy позволяет предоставить объект n-мерного массива и множество функций для управления этими массивами. Numpy – это общая структура для научных вычислений, в которой нет ничего о графах вычислений, глубоком обучении или градиентах.

Можно использовать numpy, чтобы подогнать полином третьего порядка к синусоидальной функции, вручную реализовав прямой и обратный проходы по сети с помощью операций numpy:

```
# -*- coding: utf-8 -*-
import numpy as np
import math

# Create random input and output data
x = np.linspace(-math.pi, math.pi, 2000)
y = np.sin(x)

# Randomly initialize weights
a = np.random.randn()
b = np.random.randn()
```

```
c = np.random.randn()
d = np.random.randn()

learning_rate = 1e-6
for t in range(2000):
    # Forward pass: compute predicted y
    # y = a + b x + c x^2 + d x^3
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

    # Compute and print loss
    loss = np.square(y_pred - y).sum()
    if t % 100 == 99:
        print(t, loss)

    # Backprop to compute gradients of a, b, c, d with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_a = grad_y_pred.sum()
    grad_b = (grad_y_pred * x).sum()
    grad_c = (grad_y_pred * x ** 2).sum()
    grad_d = (grad_y_pred * x ** 3).sum()

    # Update weights
    a -= learning_rate * grad_a
    b -= learning_rate * grad_b
    c -= learning_rate * grad_c
    d -= learning_rate * grad_d

print(f'Result: y = {a} + {b} x + {c} x^2 + {d} x^3')
```

Numpy – отличный фреймворк, но он не может использовать графические процессоры для ускорения вычислений. Для современных глубоких нейронных сетей графические процессоры часто обеспечивают ускорение в 50 раз или больше, поэтому numpy недостаточно для современного глубокого обучения.

PyTorch: тензоры

Представим кратко фундаментальную концепцию PyTorch: тензоры. PyTorch:тензоры концептуально идентично массиву numpy. Тензор – это *n*-мерный массив, а PyTorch предоставляет множество функций для работы с этими тензорами. Тензорные системы могут отслеживать вычислительный график и градиенты, и они полезны как универсальный инструмент для научных вычислений. В отличие от numpy, PyTorch:тензоры может использовать графические процессоры для ускорения вычислений. Чтобы запустить PyTorch Tensor на GPU, просто нужно указать правильное устройство.

Используем тензоры PyTorch, чтобы подогнать полином третьего порядка к синусоидальной функции. Как и в приведенном выше примере numpy, вручную реализуется прямой и обратный проходы по сети:

```
# -*- coding: utf-8 -*-

import torch
import math


dtype = torch.float
device = torch.device("cpu")
# device = torch.device("cuda:0") # Uncomment this to run on GPU

# Create random input and output data
x = torch.linspace(-math.pi, math.pi, 2000, device=device, dtype=dtype)
y = torch.sin(x)

# Randomly initialize weights
a = torch.randn((), device=device, dtype=dtype)
b = torch.randn((), device=device, dtype=dtype)
c = torch.randn((), device=device, dtype=dtype)
d = torch.randn((), device=device, dtype=dtype)

learning_rate = 1e-6
for t in range(2000):
    # Forward pass: compute predicted y
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

    # Compute and print loss
    loss = (y_pred - y).pow(2).sum().item()
    if t % 100 == 99:
        print(t, loss)

    # Backprop to compute gradients of a, b, c, d with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_a = grad_y_pred.sum()
    grad_b = (grad_y_pred * x).sum()
    grad_c = (grad_y_pred * x ** 2).sum()
    grad_d = (grad_y_pred * x ** 3).sum()

    # Update weights using gradient descent
    a -= learning_rate * grad_a
    b -= learning_rate * grad_b
    c -= learning_rate * grad_c
    d -= learning_rate * grad_d


print(f'Result: y = {a.item()} + {b.item()} x + {c.item()} x^2 + {d.item()} x^3')
```

PyTorch: Тензоры и автоматическое вычисление градиентов (дифференциорование)

В приведенных выше примерах пришлось вручную программировать как прямой, так и обратный проход нейронной сети. Реализация обратного прохода вручную не представляет особого труда для небольшой двухуровневой сети, но может быстро стать сложной задачей для больших многоуровневых сетей. Для облегчения работы можно использовать автоматическое дифференцирование для вычисления обратных проходов в нейронных сетях. Пакет autograd в PyTorch обеспечивает требуемую функциональность. При использовании autograd прямой проход сети определит вычислительный граф; узлы в графе будут тензорами, а ребра будут функциями, которые производят выходные тензоры из входных тензоров. Обратное распространение по этому графу позволяет легко вычислять градиенты. Выглядит сложно, но на практике это довольно просто. Каждый тензор представляет узел в вычислительном графе. Если x – это тензор, для которого x.requires_grad = True, то x.grad – это другой тензор, содержащий градиент x относительно некоторого скалярного значения. Здесь мы используем PyTorch Tensors и autograd, чтобы описать синусоидальную волну с помощью полинома третьего порядка, и обратный проход по сети реализуется следующим образом:

```
# -*- coding: utf-8 -*-
import torch
import math

dtype = torch.float
device = torch.device("cpu")
# device = torch.device("cuda:0")  # Uncomment this to run on GPU

# Create Tensors to hold input and outputs.
# By default, requires_grad=False, which indicates that we do not need to
# compute gradients with respect to these Tensors during the backward pass.
x = torch.linspace(-math.pi, math.pi, 2000, device=device, dtype=dtype)
y = torch.sin(x)

# Create random Tensors for weights. For a third order polynomial, we need
# 4 weights: y = a + b x + c x^2 + d x^3
# Setting requires_grad=True indicates that we want to compute gradients with
# respect to these Tensors during the backward pass.
a = torch.randn((), device=device, dtype=dtype, requires_grad=True)
b = torch.randn((), device=device, dtype=dtype, requires_grad=True)
c = torch.randn((), device=device, dtype=dtype, requires_grad=True)
d = torch.randn((), device=device, dtype=dtype, requires_grad=True)

learning_rate = 1e-6
for t in range(2000):
    # Forward pass: compute predicted y using operations on Tensors.
```

```
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

    # Compute and print loss using operations on Tensors.
    # Now loss is a Tensor of shape (1,)
    # loss.item() gets the scalar value held in the loss.
    loss = (y_pred - y).pow(2).sum()
    if t % 100 == 99:
        print(t, loss.item())

    # Use autograd to compute the backward pass. This call will compute the
    # gradient of loss with respect to all Tensors with requires_grad=True.
    # After this call a.grad, b.grad. c.grad and d.grad will be Tensors holding
    # the gradient of the loss with respect to a, b, c, d respectively.
    loss.backward()

    # Manually update weights using gradient descent. Wrap in torch.no_grad()
    # because weights have requires_grad=True, but we don't need to track this
    # in autograd.
    with torch.no_grad():
        a -= learning_rate * a.grad
        b -= learning_rate * b.grad
        c -= learning_rate * c.grad
        d -= learning_rate * d.grad

        # Manually zero the gradients after updating weights
        a.grad = None
        b.grad = None
        c.grad = None
        d.grad = None

print(f'Result: y = {a.item()} + {b.item()} x + {c.item()} x^2 + {d.item()} x^3')
```

PyTorch: определение новых функций автоматического дифференцирования

Внутри себя каждый примитивный оператор автоматического дифференцирования представляет собой две функции, которые работают с тензорами. Функция forward (прямое распространение) вычисляет выходные тензоры из входных тензоров. Функция backward (обратное распространение) получает производную выходных тензоров относительно некоторой скалярной величины и вычисляет градиент входных тензоров относительно той же скалярной величины. В PyTorch можно легко задать свой собственный оператор автоматического дифференцирования, определив подкласс torch.autograd.Function и реализовав функции для прямого и обратного направления. Затем можно использовать новый оператор автоматического дифференцирования, создав экземпляр и вызвав его как функцию, передав тензоры, содержащие входные данные. В этом примере теперь определим модель как $y = a + bP_3(c + dx)$, где $P_3(z) = (5z^3 - 3z)/2$ – так называемый полином Лежандра третьей степени,

вместо $y = a + bx + cx^2 + dx^3$. Напишем собственную функцию автоматического дифференцирования для прямого и обратного вычисления $P_3(z)$ и используем ее для реализации данной модели:

```
# -*- coding: utf-8 -*-
import torch
import math


class LegendrePolynomial3(torch.autograd.Function):
    """
    We can implement our own custom autograd Functions by subclassing
    torch.autograd.Function and implementing the forward and backward passes
    which operate on Tensors.
    """

    @staticmethod
    def forward(ctx, input):
        """
        In the forward pass we receive a Tensor containing the input and return
        a Tensor containing the output. ctx is a context object that can be used
        to stash information for backward computation. You can cache arbitrary
        objects for use in the backward pass using the ctx.save_for_backward method.
        """
        ctx.save_for_backward(input)
        return 0.5 * (5 * input ** 3 - 3 * input)

    @staticmethod
    def backward(ctx, grad_output):
        """
        In the backward pass we receive a Tensor containing the gradient of the loss
        with respect to the output, and we need to compute the gradient of the loss
        with respect to the input.
        """
        input, = ctx.saved_tensors
        return grad_output * 1.5 * (5 * input ** 2 - 1)


dtype = torch.float
device = torch.device("cpu")
# device = torch.device("cuda:0")  # Uncomment this to run on GPU

# Create Tensors to hold input and outputs.
# By default, requires_grad=False, which indicates that we do not need to
# compute gradients with respect to these Tensors during the backward pass.
x = torch.linspace(-math.pi, math.pi, 2000, device=device, dtype=dtype)
y = torch.sin(x)

# Create random Tensors for weights. For this example, we need
# 4 weights: y = a + b * P3(c + d * x), these weights need to be initialized
```

```
# not too far from the correct result to ensure convergence.
# Setting requires_grad=True indicates that we want to compute gradients with
# respect to these Tensors during the backward pass.
a = torch.full((), 0.0, device=device, dtype=dtype, requires_grad=True)
b = torch.full((), -1.0, device=device, dtype=dtype, requires_grad=True)
c = torch.full((), 0.0, device=device, dtype=dtype, requires_grad=True)
d = torch.full((), 0.3, device=device, dtype=dtype, requires_grad=True)

learning_rate = 5e-6
for t in range(2000):
    # To apply our Function, we use Function.apply method. We alias this as 'P3'.
    P3 = LegendrePolynomial3.apply

    # Forward pass: compute predicted y using operations; we compute
    # P3 using our custom autograd operation.
    y_pred = a + b * P3(c + d * x)

    # Compute and print loss
    loss = (y_pred - y).pow(2).sum()
    if t % 100 == 99:
        print(t, loss.item())

    # Use autograd to compute the backward pass.
    loss.backward()

    # Update weights using gradient descent
    with torch.no_grad():
        a -= learning_rate * a.grad
        b -= learning_rate * b.grad
        c -= learning_rate * c.grad
        d -= learning_rate * d.grad

        # Manually zero the gradients after updating weights
        a.grad = None
        b.grad = None
        c.grad = None
        d.grad = None

print(f'Result: y = {a.item()} + {b.item()} * P3({c.item()} + {d.item()} x)')
```

PyTorch: nn

Вычислительные графы и автоматическое вычисление градиента – мощный подход для определения сложных операторов и автоматического получения производных, однако для больших нейронных сетей необработанный автоградиент может быть слишком низкоуровневым. При построении нейронных сетей часто желательно распределение вычислений по слоям, некоторые из которых имеют обучаемые параметры, оптимизируемые во

время обучения. В TensorFlow такие пакеты, как Keras, TensorFlow-Slim и TFLearn, предоставляют абстракции более высокого уровня над необработанными вычислительными графами, которые полезны для построения нейронных сетей. В PyTorch пакет nn служит той же цели. Пакет nn определяет набор модулей, которые примерно эквивалентны уровням нейронной сети. Модуль получает входные тензоры и вычисляет выходные тензоры, но также может хранить внутреннее состояние, такое как тензоры, содержащие обучаемые параметры. Пакет nn также определяет набор полезных функций потерь, которые обычно используются при обучении нейронных сетей.

В этом примере пакет nn используется для реализации сетью полиномиальной модели:

```
# -*- coding: utf-8 -*-
import torch
import math


# Create Tensors to hold input and outputs.
x = torch.linspace(-math.pi, math.pi, 2000)
y = torch.sin(x)

# For this example, the output y is a linear function of (x, x^2, x^3), so
# we can consider it as a linear layer neural network. Let's prepare the
# tensor (x, x^2, x^3).
p = torch.tensor([1, 2, 3])
xx = x.unsqueeze(-1).pow(p)

# In the above code, x.unsqueeze(-1) has shape (2000, 1), and p has shape
# (3,), for this case, broadcasting semantics will apply to obtain a tensor
# of shape (2000, 3)

# Use the nn package to define our model as a sequence of layers. nn.Sequential
# is a Module which contains other Modules, and applies them in sequence to
# produce its output. The Linear Module computes output from input using a
# linear function, and holds internal Tensors for its weight and bias.
# The Flatten layer flatens the output of the linear layer to a 1D tensor,
# to match the shape of `y`.
model = torch.nn.Sequential(
    torch.nn.Linear(3, 1),
    torch.nn.Flatten(0, 1)
)

# The nn package also contains definitions of popular loss functions; in this
# case we will use Mean Squared Error (MSE) as our loss function.
loss_fn = torch.nn.MSELoss(reduction='sum')

learning_rate = 1e-6
```

```
for t in range(2000):

    # Forward pass: compute predicted y by passing x to the model. Module objects
    # override the __call__ operator so you can call them like functions. When
    # doing so you pass a Tensor of input data to the Module and it produces
    # a Tensor of output data.
    y_pred = model(xx)

    # Compute and print loss. We pass Tensors containing the predicted and true
    # values of y, and the loss function returns a Tensor containing the
    # loss.
    loss = loss_fn(y_pred, y)
    if t % 100 == 99:
        print(t, loss.item())

    # Zero the gradients before running the backward pass.
    model.zero_grad()

    # Backward pass: compute gradient of the loss with respect to all the learnable
    # parameters of the model. Internally, the parameters of each Module are stored
    # in Tensors with requires_grad=True, so this call will compute gradients for
    # all learnable parameters in the model.
    loss.backward()

    # Update the weights using gradient descent. Each parameter is a Tensor, so
    # we can access its gradients like we did before.
    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad

# You can access the first layer of `model` like accessing the first item of a list
linear_layer = model[0]

# For linear layer, its parameters are stored as `weight` and `bias`.
print(f'Result: y = {linear_layer.bias.item()} + {linear_layer.weight[:, 0].item()} x +
{linear_layer.weight[:, 1].item()} x^2 + {linear_layer.weight[:, 2].item()} x^3')
```

PyTorch: optim

До этого момента мы обновляли модели вручную, изменяя тензоры, содержащие обучаемые параметры, с помощью torch.no_grad (). Это легкая задача для простых алгоритмов оптимизации, таких как стохастический градиентный спуск, но на практике часто обучаются нейронные сети с помощью сложных оптимизаторов, таких как AdaGrad, RMSProp, Adam и т. д. Пакет optim в PyTorch предоставляет реализации часто используемых алгоритмов для соответствующих задач оптимизации.

В следующем примере используется пакет nn для определения модели, используя алгоритм RMSprop, предоставляемый пакетом optim:

```
# -*- coding: utf-8 -*-
import torch
import math


# Create Tensors to hold input and outputs.
x = torch.linspace(-math.pi, math.pi, 2000)
y = torch.sin(x)

# Prepare the input tensor (x, x^2, x^3).
p = torch.tensor([1, 2, 3])
xx = x.unsqueeze(-1).pow(p)

# Use the nn package to define our model and loss function.
model = torch.nn.Sequential(
    torch.nn.Linear(3, 1),
    torch.nn.Flatten(0, 1)
)
loss_fn = torch.nn.MSELoss(reduction='sum')

# Use the optim package to define an Optimizer that will update the weights of
# the model for us. Here we will use RMSprop; the optim package contains many other
# optimization algorithms. The first argument to the RMSprop constructor tells the
# optimizer which Tensors it should update.
learning_rate = 1e-3
optimizer = torch.optim.RMSprop(model.parameters(), lr=learning_rate)
for t in range(2000):
    # Forward pass: compute predicted y by passing x to the model.
    y_pred = model(xx)

    # Compute and print loss.
    loss = loss_fn(y_pred, y)
    if t % 100 == 99:
        print(t, loss.item())

    # Before the backward pass, use the optimizer object to zero all of the
    # gradients for the variables it will update (which are the learnable
    # weights of the model). This is because by default, gradients are
    # accumulated in buffers( i.e, not overwritten) whenever .backward()
    # is called. Checkout docs of torch.autograd.backward for more details.
    optimizer.zero_grad()

    # Backward pass: compute gradient of the loss with respect to model
    # parameters
    loss.backward()

    # Calling the step function on an Optimizer makes an update to its
    # parameters
    optimizer.step()
```

```
linear_layer = model[0]
print(f'Result: y = {linear_layer.bias.item()} + {linear_layer.weight[:, 0].item()} x +
{linear_layer.weight[:, 1].item()} x^2 + {linear_layer.weight[:, 2].item()} x^3')
```

PyTorch: пользовательские модули nn

Иногда могут потребоваться модели более сложные, чем последовательность существующих модулей. Для этих случаев можно определить свои собственные модули, создав подкласс nn.Module и определить процедуру, которая получает входные тензоры и производит выходные тензоры с использованием других модулей или других операций автоградиента над тензорами.

В этом примере мы реализуем наш полином третьего порядка как собственный подкласс Module:

```
# -*- coding: utf-8 -*-
import torch
import math


class Polynomial3(torch.nn.Module):
    def __init__(self):
        """
        In the constructor we instantiate four parameters and assign them as
        member parameters.
        """
        super().__init__()
        self.a = torch.nn.Parameter(torch.randn(()))
        self.b = torch.nn.Parameter(torch.randn(()))
        self.c = torch.nn.Parameter(torch.randn(()))
        self.d = torch.nn.Parameter(torch.randn(()))

    def forward(self, x):
        """
        In the forward function we accept a Tensor of input data and we must return
        a Tensor of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Tensors.
        """
        return self.a + self.b * x + self.c * x ** 2 + self.d * x ** 3

    def string(self):
        """
        Just like any class in Python, you can also define custom method on PyTorch modules
        """
        return f'y = {self.a.item()} + {self.b.item()} x + {self.c.item()} x^2 + {self.d.item()} x^3'


# Create Tensors to hold input and outputs.
```

```
x = torch.linspace(-math.pi, math.pi, 2000)
y = torch.sin(x)

# Construct our model by instantiating the class defined above
model = Polynomial3()

# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the nn.Linear
# module which is members of the model.
criterion = torch.nn.MSELoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=1e-6)
for t in range(2000):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x)

    # Compute and print loss
    loss = criterion(y_pred, y)
    if t % 100 == 99:
        print(t, loss.item())

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

print(f'Result: {model.string()}')
```

PyTorch: поток управления + распределение веса

В качестве примера динамических графиков и распределения весов возьмем довольно условную модель: полином третьего-пятого порядка, который на каждом прямом проходе выбирает случайное число от 3 до 5 и использует это количество порядков, многократно беря одни и те же веса для вычисления моделей четвертого и пятого порядка. Для этой модели возьмем обычное управление потоком Python для реализации цикла, и реализуем распределение весов, повторно используя один и тот же параметр несколько раз при определении прямого прохода.

Реализуем эту модель как подкласс Module:

```
# -*- coding: utf-8 -*-
import random
import torch
import math


class DynamicNet(torch.nn.Module):
    def __init__(self):
```

```
        """
        In the constructor we instantiate five parameters and assign them as members.
        """
        super().__init__()
        self.a = torch.nn.Parameter(torch.randn(()))
        self.b = torch.nn.Parameter(torch.randn(()))
        self.c = torch.nn.Parameter(torch.randn(()))
        self.d = torch.nn.Parameter(torch.randn(()))
        self.e = torch.nn.Parameter(torch.randn(()))

    def forward(self, x):
        """
        For the forward pass of the model, we randomly choose either 4, 5
        and reuse the e parameter to compute the contribution of these orders.

        Since each forward pass builds a dynamic computation graph, we can use normal
        Python control-flow operators like loops or conditional statements when
        defining the forward pass of the model.

        Here we also see that it is perfectly safe to reuse the same parameter many
        times when defining a computational graph.
        """
        y = self.a + self.b * x + self.c * x ** 2 + self.d * x ** 3
        for exp in range(4, random.randint(4, 6)):
            y = y + self.e * x ** exp
        return y

    def string(self):
        """
        Just like any class in Python, you can also define custom method on PyTorch modules
        """
        return f'y = {self.a.item()} + {self.b.item()} x + {self.c.item()} x^2 + {self.d.item()} x^3 +
{self.e.item()} x^4 ? + {self.e.item()} x^5 ?'


# Create Tensors to hold input and outputs.
x = torch.linspace(-math.pi, math.pi, 2000)
y = torch.sin(x)

# Construct our model by instantiating the class defined above
model = DynamicNet()

# Construct our loss function and an Optimizer. Training this strange model with
# vanilla stochastic gradient descent is tough, so we use momentum
criterion = torch.nn.MSELoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=1e-8, momentum=0.9)
for t in range(30000):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x)
```

```
    # Compute and print loss
    loss = criterion(y_pred, y)
    if t % 2000 == 1999:
        print(t, loss.item())

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

print(f'Result: {model.string()}')
```

5. LSTM С ИСПОЛЬЗОВАНИЕМ PYTORCH

Обратимся к рассмотрению примера реализации LSTM с использованием PyTorch. Перед рассмотрением примера обратим внимание на некоторые детали. LSTM PyTorch предполагает, что входные данные являются трехмерными тензорами. Важна семантика осей этих тензоров. Первая ось – это сама последовательность данных, вторая индексирует экземпляры в мини-пакете, а третья индексирует элементы ввода. Мы не обсуждаем мини-пакетирование, поэтому просто проигнорируем это и примем, что будет только одно измерение на второй оси.

Рассмотрим прогнозирование временного ряда авиаперевозок по месяцам с помощью сети LSTM в пакете PyTorch. Прежде всего, нужно загрузить данные в виде файла из набора данных на сайте:

```
#!wget https://raw.githubusercontent.com/jbrownlee/Datasets/master/airline-passengers.csv
```

Листинг программы приведен в Приложении.

Для начала работы нужно загрузить используемые программы из библиотеки (Library).

Затем считываем данные и отображаем их на графике (Data Plot), как показано на рис. 4.

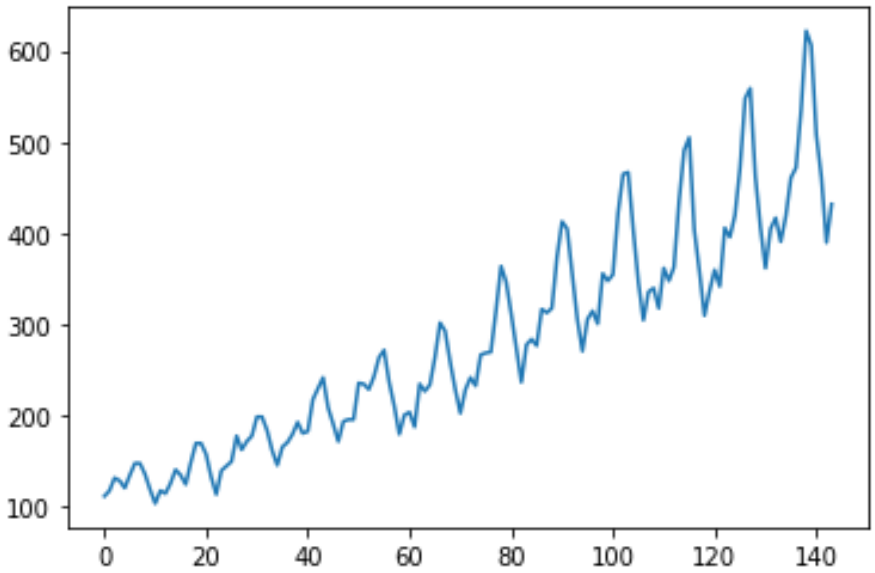

Рис. 4. Статистические данные об авиаперевозках по месяцам

Задаем параметры данных и обучения (Dataloading).

Задаем модуль LSTM (Modul).

Проводим обучение сети (Training).

Проверяем качество результата обучения сети на контрольной последовательности (Testing for Airplane Passengers Dataset) и представляем результаты, как показано на рис. 5.

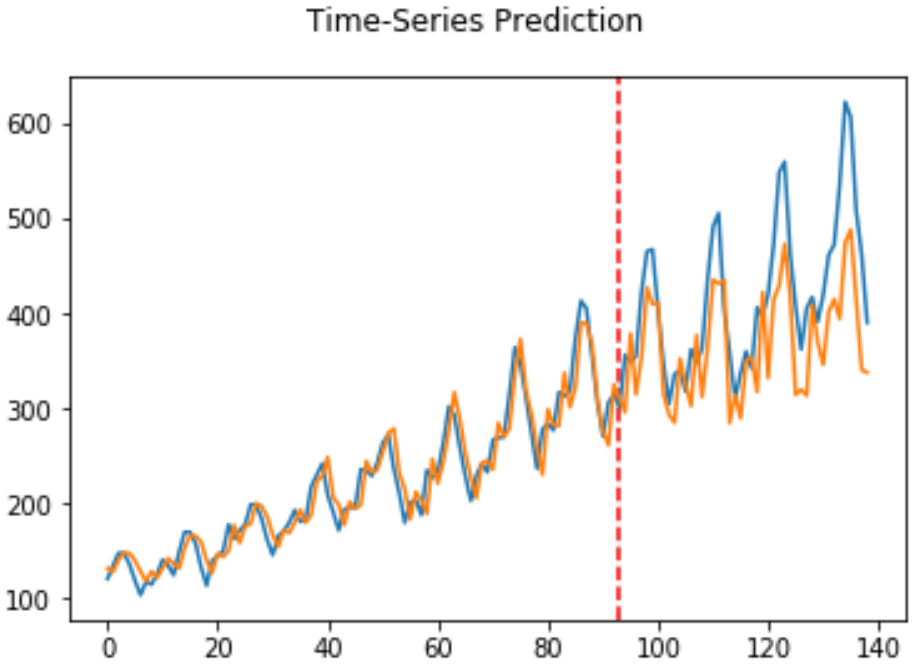

Рис. 5. Прогноз временного ряда

Полученные результаты анализируем и делаем вывод о качестве обучения нейронной сети.

6. ПРАКТИЧЕСКИЕ ЗАДАНИЯ

Практические задания направлены на получения навыков работы с фреймворком PyTorch. Перед началом работы убедитесь, что у вас установлены пакеты numpy, matplotlib, pandas, torch, torch.nn .

6.1. Рассмотреть аппроксимацию периодической функции вида $A\cos x + B\sin x$ на интервале $(-\pi, \pi)$, используя модель в виде полинома третьей степени $y = a + bx + cx^2 + dx^3$. Параметры A и B исходной функции задаются преподавателем.

6.2. Для обучения модели использовать модуль PyTorch, указанный преподавателем.

6.3. Построить график исходной зависимости и обученной модели. На основе графических данных сравнить полученные результаты.

6.4. Рассмотреть работу алгоритма LSTM с использованием PyTorch, на основе программы, приведенной в Приложении.

6.5. Рассмотреть практически путем расчетов влияние на точность результата длины обучающей выборки. Сделать выводы по результату исследования.

6.6. Провести обучение, меняя количество эпох и скорость обучения. Сравнить результаты и сделать вывод о влиянии выбранных параметров на результат.

6.7. Предоставить преподавателю отчет о выполненных заданиях в Word.

БИБЛИОГРАФИЧЕСКИЙ СМПИСОК

1. С. Хайкин. Нейронные сети. – М.: Изд. дом «Вильямс», 2006.
2. С. Осовский. Нейронные сети для обработки информации. – М.: Финансы и статистика, 2002.
3. Я. Гудфеллоу, И. Бенджио, А. Курвилль. Глубокое обучение. – М.: ДМК «Пресс», 2018.
4. О. Жерон. Прикладное машинное обучение с помощью Scikit-Learn и TensorFlow. – СПб: ООО «Диалектика», 2020.
5. С. Николенко, А. Кадурин, Е. Архангельская. Глубокое обучение. – СПб: «Питер», 2018.
6. N.D. Lewis. Deep Time Series Forecasting with Python: An Intuitive Introduction to Deep Learning for Applied Time Series Modeling, CreateSpace Independent Publishing Platform. 2016.
7. Yu, Y., Si, X., Hu, C., & Zhang, J. (2019). *A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures. Neural Computation, 1–36.* doi:10.1162/neco_a_01199.
8. Van Houdt, G., Mosquera, C. & Nápoles, G. A review on the long short-term memory model. *Artif. Intel.l Rev*. **53,** 5929–5955 (2020). https://doi.org/10.1007/s10462-020-09838-1.
9. **https://pytorch.org**
10. **https://colab.research.google.com/github/dlmacedo/starter-academic/blob/master/content/courses/deeplearning/notebooks/pytorch/Time_Series_Prediction_with_LSTM_Using_PyTorch.ipynb#scrollTo=35ndYIwIKteS**
11. **https://colab.research.google.com/notebooks/intro.ipynb**

ОГЛАВЛЕНИЕ

ПРИЛОЖЕНИЕ 1

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import torch
import torch.nn as nn
from torch.autograd import Variable
from sklearn.preprocessing import MinMaxScaler

training_set = pd.read_csv('airline-passengers.csv')
#training_set = pd.read_csv('shampoo.csv')
training_set = training_set.iloc[:,1:2].values
#plt.plot(training_set, label = 'Shampoo Sales Data')
plt.plot(training_set, label = 'Airline Passangers Data')
plt.show()
def sliding_windows(data, seq_length):
    x = []
    y = []

    for i in range(len(data)-seq_length-1):
        _x = data[i:(i+seq_length)]
        _y = data[i+seq_length]
        x.append(_x)
        y.append(_y)

    return np.array(x),np.array(y)

sc = MinMaxScaler()
training_data = sc.fit_transform(training_set)
seq_length = 4
x, y = sliding_windows(training_data, seq_length)

train_size = int(len(y) * 0.67)
test_size = len(y) - train_size

dataX = Variable(torch.Tensor(np.array(x)))
dataY = Variable(torch.Tensor(np.array(y)))
trainX = Variable(torch.Tensor(np.array(x[0:train_size])))
trainY = Variable(torch.Tensor(np.array(y[0:train_size])))
testX = Variable(torch.Tensor(np.array(x[train_size:len(x)])))
testY = Variable(torch.Tensor(np.array(y[train_size:len(y)])))

class LSTM(nn.Module):

    def __init__(self, num_classes, input_size, hidden_size, num_layers):
        super(LSTM, self).__init__()

        self.num_classes = num_classes
```

```
        self.num_layers = num_layers
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.seq_length = seq_length
        self.lstm = nn.LSTM(input_size=input_size, hidden_size=hidden_size,
                            num_layers=num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        h_0 = Variable(torch.zeros(
            self.num_layers, x.size(0), self.hidden_size))
        c_0 = Variable(torch.zeros(
            self.num_layers, x.size(0), self.hidden_size))
        # Propagate input through LSTM
        ula, (h_out, _) = self.lstm(x, (h_0, c_0))
        h_out = h_out.view(-1, self.hidden_size)
        out = self.fc(h_out)

        return out

num_epochs = 2000
learning_rate = 0.01
input_size = 1
hidden_size = 2
num_layers = 1
num_classes = 1
lstm = LSTM(num_classes, input_size, hidden_size, num_layers)
criterion = torch.nn.MSELoss()    # mean-squared error for regression
optimizer = torch.optim.Adam(lstm.parameters(), lr=learning_rate)
#optimizer = torch.optim.SGD(lstm.parameters(), lr=learning_rate)

# Train the model
for epoch in range(num_epochs):
    outputs = lstm(trainX)
    optimizer.zero_grad()
    # obtain the loss function
    loss = criterion(outputs, trainY)
    loss.backward()
    optimizer.step()
    if epoch % 100 == 0:
      print("Epoch: %d, loss: %1.5f" % (epoch, loss.item()))
      lstm.eval()
train_predict = lstm(dataX)

data_predict = train_predict.data.numpy()
dataY_plot = dataY.data.numpy()
data_predict = sc.inverse_transform(data_predict)
dataY_plot = sc.inverse_transform(dataY_plot)
plt.axvline(x=train_size, c='r', linestyle='--')
plt.plot(dataY_plot)
```

```
plt.plot(data_predict)
plt.suptitle('Time-Series Prediction')
plt.show()
```

РЕКУРРЕНТНЫЕ НЕЙРОННЫЕ СЕТИ: МОДЕЛИРОВАНИЕ ПОСЛЕДОВАТЕЛЬНОСТЕЙ

Методические указания
к выполнению практических работ
для студентов 2-го курса магистратуры, обучающихся по образовательным программам
направления 09.04.03 «Прикладная информатика» всех форм обучения

Составители: Головинский Павел Абрамович,
Шаталова Ангелина Олеговна,
Еникеев Эльдар Ильясович

Подписано в печать 00.00.2022 Формат 60x84 1/16. Уч.-изд. л. 0,0.
Усл. печ. л. 0.0. Бумага для множительных аппаратов. Тираж 50 экз. Заказ №___.

ФГБОУ ВО «Воронежский государственный технический университет»
394026 г. Воронеж, Московский проспект, 14

Участок оперативной полиграфии издательство ВГТУ
394026 г. Воронеж, Московский проспект, 14