ИСКУССТВЕННЫЕ НЕЙРОННЫЕ СЕТИ

Лекция 10
LSTM и GRU

Управляемая память, математические модели, градиенты и реализация на Python

Преподаватель: Аршын Алтыбай

Кафедра компьютерной инженерии,
Алматинский технологический университет, Алматы, Казахстан

Учебный год: 2026

> **Связь с предыдущими занятиями.** В лекции 9 была получена простая RNN и рассмотрен метод обратного распространения во времени (BPTT). Мы выяснили, что длинная цепочка производных может приводить к затуханию или взрыву градиента. Теперь изучим две архитектуры с *управляемой памятью*: Long Short-Term Memory (LSTM) и Gated Recurrent Unit (GRU). Основные исторические источники: [9, 11, 6].

1. Цель лекции и результаты обучения

Цель — понять, как обучаемые вентили регулируют запись, сохранение и извлечение информации, вывести формулы LSTM и GRU, научиться анализировать их размерности и корректно обучать их на последовательных данных.

После лекции студент сможет:

1. объяснить ограничения простой RNN и смысл дополнительного состояния памяти;
2. записать все уравнения LSTM и GRU с точными размерностями;
3. объяснить назначение forget, input, output, update и reset gates;
4. вручную выполнить один шаг вычисления обеих ячеек;
5. найти количество обучаемых параметров и сравнить сложность архитектур;
6. объяснить перенос градиента через состояние c_t без утверждения, что проблема затухания исчезает полностью;
7. построить корректный эксперимент по прогнозированию временного ряда;

8. обучить, проверить и сравнить LSTM и GRU в PyTorch.

План лекции: ограничения RNN; устройство LSTM; прямой проход и градиенты; устройство GRU; сравнение; работа с батчами, масками и двунаправленностью; NumPy и PyTorch; контрольные вопросы.

2. От простой рекурсии к управляемой памяти

2.1 Краткое повторение простой RNN

Пусть $x_t \in \mathbb{R}^{d_x}$ — вход на шаге t, $h_t \in \mathbb{R}^{d_h}$ — скрытое состояние, h_0 задано. Обычная рекуррентная сеть использует переход

$$h_t = \tanh(W_x x_t + W_h h_{t-1} + b). \tag{1}$$

Вычисление на шаге T зависит от прошлого только через h_{T-1}. Для переноса производной от позднего времени к раннему появляется произведение якобианов:

$$\frac{\partial h_T}{\partial h_k} = J_T J_{T-1} \cdots J_{k+1}, \qquad J_t = \operatorname{diag}(1 - \tanh^2(a_t)) W_h. \tag{2}$$

Если нормы большинства множителей заметно меньше единицы, градиент уменьшается; если больше единицы, он может расти. Это упрощённое объяснение: фактическое поведение зависит также от направления градиента и спектральных свойств матриц [8, 6].

2.2 Идея вентилей

Управляемая сеть должна *отдельно* отвечать на три вопроса: что сохранить из прошлого, какую новую информацию записать, какую часть памяти использовать для ответа. Для этого вычисляются числа от нуля до единицы:

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \qquad \sigma(z) \in (0, 1).$$

Вектор из таких чисел называется *вентилем* (gate). Поэлементное умножение $g \odot v$ позволяет уменьшать или сохранять отдельные компоненты v. Значение $g_i \approx 0$ означает слабую передачу соответствующей компоненты, а $g_i \approx 1$ — сильную.

> **Важное уточнение** Вентиль не является бинарным переключателем. Во время обучения его значения непрерывны и дифференцируемы; он

оптимизируется вместе с остальными весами методом градиентного спуска.

3. Архитектура Long Short-Term Memory

3.1 Откуда появилась LSTM

LSTM была предложена С. Хохрайтером и Ю. Шмидхубером в 1997 году для лучшего сохранения информации на длинных промежутках [9]. Современная распространённая запись содержит три вентиля и две связанные переменные состояния: c_t — *состояние ячейки памяти*, h_t — *скрытое состояние*. В распространённых вариантах уравнений присутствует forget gate, развитый в последующих модификациях [10].

3.2 Уравнения одной ячейки

Для $x_t \in \mathbb{R}^{d_x}$, $h_{t-1}, c_{t-1} \in \mathbb{R}^{d_h}$ запишем:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f), \quad \text{вентиль забывания}, \tag{3}$$

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i), \quad \text{вентиль записи}, \tag{4}$$

$$\widetilde{c}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c), \quad \text{кандидат памяти}, \tag{5}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \widetilde{c}_t, \quad \text{обновление памяти}, \tag{6}$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o), \quad \text{вентиль вывода}, \tag{7}$$

$$h_t = o_t \odot \tanh(c_t), \quad \text{скрытое состояние}. \tag{8}$$

Матрицы $W_\bullet \in \mathbb{R}^{d_h \times d_x}$, $U_\bullet \in \mathbb{R}^{d_h \times d_h}$, смещения $b_\bullet \in \mathbb{R}^{d_h}$. Произведение $\odot$ выполняется *покомпонентно*, не как обычное матричное умножение. Порядок вычисления o_t и кандидата не влияет на результат: оба используют старое h_{t-1}.

Смысл двух состояний. c_t служит отдельным каналом обновления памяти, тогда как h_t — доступное снаружи представление ячейки. В LSTM-слое PyTorch оба имеют размерность d_h, но концептуально играют разные роли.

3.3 Понимание каждого вентиля на примере

Представим, что сеть читает предложение *«Вчера погода была солнечной. Сегодня она стала ...»*. Для выбора окончания полезно помнить признаки текущей темы (погода), но не обязательно сохранять каждое слово. Вентиль f_t может сохранить компоненты, связанные с темой, i_t — записать сведения о смене дня, а o_t — открыть нужную часть памяти для предсказания. Это *интерпретационная аналогия*, а не утверждение, что отдельный нейрон всегда кодирует строго один понятный человеку признак.

3.4 Геометрический смысл обновления

Формула (6) является суммой двух управляемых каналов:

$$\underbrace{f_t \odot c_{t-1}}_{\text{сохраняемая память}} + \underbrace{i_t \odot \widetilde{c}_t}_{\text{новая информация}}.$$

Поскольку нет обязательного внешнего ограничения $f_t + i_t = 1$, компонента памяти c_t может по модулю превышать единицу. Внешнее скрытое состояние h_t ограничивается $\tanh(c_t)$ и вентилем вывода. Именно поэтому нельзя отождествлять c_t с обычной tanh-активацией.

4. Подробный численный пример: один шаг LSTM

Пусть размерность каждого состояния равна единице, предыдущие значения $c_{t-1} = 0.40$, $h_{t-1} = 0.10$. Допустим, после линейных преобразований получены предактивации

$$a_f = 0, \qquad a_i = \ln 3, \qquad a_c = \operatorname{artanh}(0.5), \qquad a_o = \ln 3.$$

Поскольку $\sigma(0) = 0.5$, $\sigma(\ln 3) = 0.75$ и $\tanh(\operatorname{artanh}(0.5)) = 0.5$, получаем

$$\begin{aligned} f_t &= 0.5, \quad i_t = 0.75, \quad \widetilde{c}_t = 0.5, \quad o_t = 0.75, \\ c_t &= 0.5 \cdot 0.4 + 0.75 \cdot 0.5 = 0.575, \\ h_t &= 0.75 \tanh(0.575) \approx 0.38931. \end{aligned}$$

Здесь старая память даёт вклад 0.2, новая — 0.375. Выход не равен 0.575, потому что LSTM отдельно фильтрует память вентилем вывода.

Пример 1. Проверка ручного расчёта LSTM на NumPy

```python
import numpy as np
sigmoid = lambda z: 1.0 / (1.0 + np.exp(-z))
c_prev = 0.40
f = sigmoid(0.0)
i = sigmoid(np.log(3.0))
g = np.tanh(np.arctanh(0.5))
o = sigmoid(np.log(3.0))
c = f * c_prev + i * g
h = o * np.tanh(c)
print('f, i, candidate, o =', f, i, g, o)
print(f'c={c:.6f}, h={h:.6f}')
```

Ожидаемый вывод

```
f, i, candidate, o = 0.5 0.75 0.5 0.75
c=0.575000, h=0.389314
```

5. Почему LSTM помогает передавать градиент?

5.1 Локальная производная вдоль памяти

Рассмотрим *прямой* путь по ребру $c_{t-1} \to c_t$ при фиксированных текущих значениях вентилей. Из (6) следует:

$$\left.\frac{\partial c_t}{\partial c_{t-1}}\right|_{f_t, i_t, \widetilde{c}_t \text{ fixed}} = \operatorname{diag}(f_t). \tag{9}$$

Вдоль такого пути за k шагов возникает произведение $f_t \odot f_{t-1} \odot \cdots$, а не обязательное произведение множителей $\tanh'$ и рекуррентных матриц. Если значения forget gate близки к единице, этот канал может передавать сигнал на большее число шагов [9, 6].

Что часто объясняют неправильно Нельзя утверждать, что *полный* градиент LSTM всегда равен произведению forget gates: вентили и скрытые состояния зависят друг от друга, поэтому существуют дополнительные пути производной. Кроме того, $f_t < 1$ в обычной сигмоиде: при достаточно большой длине последовательности даже прямой путь может затухать. LSTM *смягчает*, но не устраняет все проблемы обучения длинным зависимостям.

5.2 Численное сравнение каналов

Если коэффициент перехода обычной линейной рекурсии равен 0.5, то через 20 шагов его вклад равен $0.5^{20} \approx 9.54 \cdot 10^{-7}$. В LSTM при $f_t = 0.99$ вдоль прямого канала памяти получаем $0.99^{20} \approx 0.8179$. Это лишь иллюстрация конкретных *путей* графа, а не гарантии успешного обучения любой LSTM.

5.3 Инициализация forget gate

На практике положительное начальное смещение forget gate иногда помогает сберегать память в начале обучения. Однако выбор смещений зависит от используемого фреймворка и постановки. При работе с готовой библиотекой нельзя предполагать, что все начальные значения одинаковы: их следует проверять и фиксировать в описании эксперимента.

6. Параметры и вычислительная сложность LSTM

6.1 Подсчёт параметров

В учебной записи четыре аффинных преобразования: для $f, i, \widetilde{c}, o$. Каждое использует $d_h d_x$ входных весов, d_h^2 рекуррентных весов и d_h смещений. Поэтому для одного слоя LSTM:

$$N_{\text{LSTM,one bias}} = 4d_h(d_x + d_h + 1). \tag{10}$$

Например, при $d_x = 6$, $d_h = 16$ получаем $4 \cdot 16 \cdot (6 + 16 + 1) = 1472$ параметра.

Особенность PyTorch. В `torch.nn.LSTM` для каждого слоя и направления хранятся *два* вектора смещений (`bias_ih` и `bias_hh`). Тогда число параметров стандартного однослойного однонаправленного LSTM без проекций равно

$$N_{\text{LSTM,PyTorch}} = 4d_h(d_x + d_h + 2). \tag{11}$$

При тех же размерах это 1536. Два смещения могут быть объединены алгебраически в одно, поэтому число хранимых параметров библиотеки отличается от компактной учебной формулы. Для многослойной сети необходимо учитывать изменившуюся размерность входов верхних слоёв [13].

6.2 Временные и пространственные затраты

При длине последовательности T основные умножения матриц масштабируются примерно как $O(T(d_x d_h + d_h^2))$ с постоянным множителем порядка четырёх для четырёх групп вентилей. При BPTT приходится сохранять промежуточные результаты или пересчитывать их. Разделение параметров сохраняет количество весов независимым от T, но не делает стоимость вычисления независимой от длины.

7. GRU: более компактная рекуррентная ячейка

7.1 Мотивация

GRU (Gated Recurrent Unit) появилась в работе K. Чо и соавторов в контексте encoder–decoder архитектуры [11]. В отличие от LSTM, типичная GRU имеет *одно* состояние h_t и два вентиля: update gate z_t и reset gate r_t. В ней нет отдельной пары (c_t, h_t) и отдельного output gate.

7.2 Одна из распространённых математических записей GRU

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z), \tag{12}$$

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r), \tag{13}$$

$$\widetilde{h}_t = \tanh\big(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h\big), \tag{14}$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \widetilde{h}_t. \tag{15}$$

Здесь $z_t \approx 1$ способствует сохранению старого состояния, а $z_t \approx 0$ — переходу к кандидату. Reset gate контролирует влияние предыдущего состояния при построении кандидата.

> **Соглашения различаются.** В некоторых учебниках формулу записывают как $h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \widetilde{h}_t$: тогда семантика z_t относительно сохранения/обновления противоположна. Обе записи допустимы при согласованном определении вентиля. Кроме того, PyTorch вычисляет кандидат после применения reset gate к *рекуррентному аффинному выражению*, а не до умножения на U_h; см. раздел ниже [14].

7.3 Что делает reset gate?

Если r_t близок к нулю, кандидат $\widetilde{h}_t$ меньше зависит от прошлого в учебной записи (14). Это полезно, когда текущий вход указывает на резкую смену режима последовательности. При r_t около единицы кандидат учитывает предыдущий контекст. При этом итоговое состояние может сохранить прошлое напрямую через update gate z_t.

8. Ручной численный пример GRU

Возьмём одномерную модель, $h_{t-1} = 0.4$, $z_t = 0.75$, $r_t = 0.5$, $\widetilde{h}_t = -0.2$. Эти значения могут получиться после соответствующих аффинных преобразований и функций активации. По (15)

$$h_t = 0.75 \cdot 0.4 + (1 - 0.75) \cdot (-0.2) = 0.30 - 0.05 = 0.25.$$

Значит, из-за большого z_t итоговая память всё ещё существенно определяется предыдущим состоянием. Если бы $z_t = 0.1$, то $h_t = 0.1 \cdot 0.4 + 0.9 \cdot (-0.2) = -0.14$.

Пример 2. Проверка обновления GRU

```python
import numpy as np
h_prev = 0.4
h_candidate = -0.2
for z in (0.75, 0.10):
    h = z * h_prev + (1 - z) * h_candidate
    print(f'z={z:.2f} -> h={h:.2f}')
```

Ожидаемый вывод

```
z=0.75 -> h=0.25
z=0.10 -> h=-0.14
```

9. GRU в PyTorch и отличие от учебной записи

В PyTorch используются следующие формулы для одного слоя [14]:

$$
\begin{aligned}
r_t &= \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{t-1} + b_{hr}),\\
z_t &= \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{t-1} + b_{hz}),\\
n_t &= \tanh\big(W_{in}x_t + b_{in} + r_t \odot (W_{hn}h_{t-1} + b_{hn})\big),\\
h_t &= (1 - z_t) \odot n_t + z_t \odot h_{t-1}.
\end{aligned}
$$

Отличие содержится в положении r_t: обычно $U_h(r_t \odot h_{t-1}) \neq r_t \odot (U_h h_{t-1})$, поскольку матричное умножение и покомпонентное масштабирование не коммутируют. Поэтому не следует ожидать полного совпадения параметров и численных результатов между произвольной самописной учебной GRU и `nn.GRU`. Однако обе реализации выражают ту же основную идею управляемого перехода состояния.

9.1 Количество параметров GRU

Учебная ячейка содержит три группы весов и смещений:

$$N_{\text{GRU,one bias}} = 3d_h(d_x + d_h + 1). \tag{16}$$

Стандартная реализация PyTorch использует два вектора смещений для каждой группы:

$$N_{\text{GRU,PyTorch}} = 3d_h(d_x + d_h + 2).$$

При $d_x = 6$, $d_h = 16$: $N_{\text{GRU,PyTorch}} = 3 \cdot 16 \cdot 24 = 1152$, меньше, чем 1536 для LSTM той же ширины.

10. Сравнение RNN, LSTM и GRU

Критерий	Простая RNN	LSTM	GRU
Состояния	h_t	h_t и c_t	h_t
Вентили	нет	три основных	два
Группы аффинных преобразований	1	4	3
Длинные зависимости	часто трудно	управляемый канал памяти	управляемое сохранение состояния
Параметры (типичная ячейка)	меньше	больше	между RNN и LSTM
Характерная задача	небольшие последовательности	зависимости с длительным контекстом	компактные последовательные модели

Ни один вариант не гарантирует лучшей точности *для всех* данных. Влияние архитектуры нужно измерять на отложенной выборке, учитывая одинаковые условия обучения, количество параметров, вычислительный бюджет и корректную процедуру оценки [4, 3].

11. Многослойные и двунаправленные сети

11.1 Stacked LSTM/GRU

Многослойная архитектура передаёт последовательность скрытых состояний слоя ℓ в следующий слой $\ell+1$. Если у первого слоя вход имеет размер d_x, а скрытая ширина равна d_h, то на следующем однонаправленном слое вход имеет размер d_h. При двунаправленности — обычно $2d_h$.

11.2 Bidirectional

Bidirectional LSTM/GRU анализирует последовательность слева направо и справа налево, после чего объединяет результаты. Для классификации *уже полностью доступного* текста это может быть полезно. Но для онлайн-прогноза будущих значений обратное направление, использующее будущие наблюдения, нарушало бы причинность.

11.3 Dropout между слоями

В `nn.LSTM(..., num_layers=2, dropout=0.2)` аргумент `dropout` относится к выходам промежуточных рекуррентных слоёв (не последнего) в многослойной конструкции; он не означает автоматический dropout на всех внутренних переходах отдельной ячейки. Для одного слоя такой параметр не обеспечивает ту же регуляризацию. Семантику аргумента следует уточнять по документации [13].

12. Как подавать данные в PyTorch

Пусть используется `batch_first=True`. Тогда вход имеет форму (B, T, d_x), где B — число последовательностей в пакете, T — их длина.

Для однонаправленной однослойной LSTM:

$$\texttt{output} \in \mathbb{R}^{B \times T \times d_h}, \quad \texttt{h_n}, \texttt{c_n} \in \mathbb{R}^{1 \times B \times d_h}.$$

Для GRU вместо пары возвращается `h_n`. При наличии нескольких слоёв/направлений первая размерность конечных состояний равна $D \cdot L$, где $D = 1$ или 2, L — число слоёв. `batch_first=True` *не изменяет* порядок размерностей `h_n` и `c_n`. Для bidirectional-моделей последний элемент массива `output[:, -1, :]` не тождествен простому объединению двух финальных состояний из `h_n`; направление и временная позиция имеют значение.

Пример 3. Проверка формы тензоров и количества параметров

```python
import torch
B, T, dx, dh = 4, 12, 6, 16
x = torch.randn(B, T, dx)
lstm = torch.nn.LSTM(dx, dh, batch_first=True)
gru = torch.nn.GRU(dx, dh, batch_first=True)
y_l, (h_l, c_l) = lstm(x)
y_g, h_g = gru(x)
print(y_l.shape, h_l.shape, c_l.shape)
print(y_g.shape, h_g.shape)
print('LSTM:', sum(p.numel() for p in lstm.parameters()))
print('GRU:', sum(p.numel() for p in gru.parameters()))
```

Ожидаемый вывод

```
torch.Size([4, 12, 16]) torch.Size([1, 4, 16]) torch.Size([1, 4, 16])
torch.Size([4, 12, 16]) torch.Size([1, 4, 16])
LSTM: 1536
GRU: 1152
```

13. Практическая задача: прогноз следующего значения временного ряда

13.1 Постановка

Дан одномерный временной ряд $s_0, \dots, s_{N-1}$. По окну длины m необходимо спрогнозировать следующее значение:

$$(s_{t-m}, \dots, s_{t-1}) \mapsto \widehat{s}_t.$$

Для простоты воспользуемся синтетическим рядом с синусоидальными компонентами и шумом. Важно соблюдать порядок времени: ранние наблюдения служат обучением, более поздние — валидацией, самые поздние — тестом. Масштабирование оценим *только по обучающему интервалу*. Для наглядности код ниже использует обычное окно без перекрытия целевых временных индексов между разными выборками.

13.2 Подготовка воспроизводимого эксперимента

Создаём 360 наблюдений и выделяем первые 240 для обучения, следующие 60 для валидации, последние 60 для теста. Стандартизация использует среднее и стандартное отклонение первых 240 значений. Для каждого раздела строим окна, не содержащие целевых ответов из другого раздела. Чтобы тест имел достаточный контекст, в его входные окна допустимо включать предшествующие наблюдения, *уже известные к моменту прогноза*; здесь для прозрачности используются только окна внутри соответствующего раздела.

Пример 4. Данные, разбиение и скользящие окна

```
import numpy as np
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

np.random.seed(7)
torch.manual_seed(7)
t = np.arange(360, dtype=np.float32)
series = (np.sin(0.07*t) + 0.35*np.sin(0.19*t)
          + 0.05*np.random.randn(len(t))).astype(np.float32)
train_raw = series[:240]
val_raw = series[240:300]
test_raw = series[300:]
mean = float(train_raw.mean())
std = float(train_raw.std()) + 1e-8
train = (train_raw - mean) / std
```

Пример 4 (продолжение)

```
val = (val_raw - mean) / std
test = (test_raw - mean) / std

window = 12
def build_windows(arr, m=12):
    xs, ys = [], []
    for j in range(m, len(arr)):
        xs.append(arr[j-m:j, None])
        ys.append(arr[j])
    return (torch.tensor(np.stack(xs), dtype=torch.float32),
            torch.tensor(np.array(ys)[:, None], dtype=torch.float32))
Xtr, Ytr = build_windows(train, window)
Xva, Yva = build_windows(val, window)
Xte, Yte = build_windows(test, window)
loader = DataLoader(TensorDataset(Xtr, Ytr),
                    batch_size=32, shuffle=True)
print(Xtr.shape, Xva.shape, Xte.shape)
```

Ожидаемый вывод

```
torch.Size([228, 12, 1]) torch.Size([48, 12, 1]) torch.Size([48, 12, 1])
```

13.3 Одна универсальная модель для LSTM и GRU

Рекуррентный encoder преобразует окно в скрытое представление, а линейная *голова* переводит его в прогноз вещественного значения. Параметр `cell` управляет выбором архитектуры.

Пример 5. Архитектура модели

```
class Forecaster(nn.Module):
    def __init__(self, cell='LSTM', hidden=24):
        super().__init__()
        rnn_class = nn.LSTM if cell == 'LSTM' else nn.GRU
        self.rnn = rnn_class(input_size=1, hidden_size=hidden,
                             batch_first=True)
        self.head = nn.Linear(hidden, 1)

    def forward(self, x):
        _, state = self.rnn(x)
        h = state[0] if isinstance(state, tuple) else state
        return self.head(h[-1])

lstm_model = Forecaster('LSTM')
gru_model = Forecaster('GRU')
print('LSTM model ready:', isinstance(lstm_model, nn.Module))
print('GRU model ready:', isinstance(gru_model, nn.Module))
```

Ожидаемый вывод

```
LSTM model ready: True
GRU model ready: True
```

13.4 Обучение и Early Stopping

Для регрессии минимизируем среднюю квадратичную ошибку MSE $= n^{-1}\sum_{j=1}^{n}(\widehat{y}_j - y_j)^2$. В процессе обучения отслеживаем качество *на validation*, сохраняя веса лучшей эпохи. Тест используется только после окончания настройки.

Пример 6. Обучение с выбором лучшей эпохи

```
import copy

def fit_model(model, epochs=80, patience=12):
    opt = torch.optim.Adam(model.parameters(), lr=0.01)
    loss_fn = nn.MSELoss()
    best_loss = float('inf')
    best_state = None
    wait = 0
    for epoch in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            pred = model(xb)
            loss = loss_fn(pred, yb)
            loss.backward()
            nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            opt.step()
        model.eval()
        with torch.no_grad():
            valid = loss_fn(model(Xva), Yva).item()
        if valid < best_loss - 1e-6:
            best_loss = valid
            best_state = copy.deepcopy(model.state_dict())
            wait = 0
        else:
            wait += 1
            if wait >= patience:
                break
    model.load_state_dict(best_state)
    return best_loss

for cell in ('LSTM', 'GRU'):
    torch.manual_seed(7)
    net = Forecaster(cell)
    val_mse = fit_model(net)
    net.eval()
    with torch.no_grad():
```

Пример 6 (продолжение)

```
        test_mae = torch.mean(torch.abs(net(Xte)-Yte)).item()*std
    print(cell, f'val_MSE={val_mse:.4f}',
          f'test_MAE_original_units={test_mae:.4f}')
```

Численные значения метрик зависят от версии PyTorch, оборудования, порядка операций и других деталей. Поэтому здесь намеренно не приводится выдуманный «ожидаемый» результат обучения. Студент должен запустить код, сохранить собственный журнал и интерпретировать полученные числа.

13.5 Честное сравнение с наивным прогнозом

Прежде чем объявить нейросеть полезной, сравним её с прогнозом *последнее наблюдение равно следующему*:

$$\widehat{s}_t^{\text{naive}} = s_{t-1}.$$

Пример 7. Простой baseline на тестовом интервале

```
naive = Xte[:, -1, 0:1]
naive_mae = torch.mean(torch.abs(naive-Yte)).item() * std
print(f'Naive test MAE: {naive_mae:.4f}')
```

Все значения стандартизированы одинаково, а умножение ошибки на `std` возвращает её в исходные единицы. Даже если LSTM/GRU уменьшит train loss, это само по себе не доказывает, что она превосходит baseline на будущем интервале.

14. Типичные ошибки в практике

1. **Путаница h_t и c_t.** В LSTM это два разных состояния; в GRU отдельного c_t нет.
2. **Неверное толкование update gate.** В зависимости от соглашения z_t может означать преимущественное сохранение старой памяти либо её замену кандидатом.
3. **Несоответствие NumPy и PyTorch GRU.** В библиотечной формуле reset gate может стоять после рекуррентного преобразования.
4. **Ошибки размерностей.** При `batch_first=True` вход имеет (B, T, d_x), конечные состояния — (DL, B, d_h).
5. **Неправильное разделение временного ряда.** Случайное перемешивание окон перед train/test split может переносить сведения о буду-

щем в обучение.

6. **Нормализация по всем данным.** На будущей тестовой области нельзя оценивать mean/std для обучения.
7. **Сравнение моделей с сильно разным бюджетом.** Лучше учитывать не только ширину, но и число параметров, время обучения и фиксированные условия эксперимента.
8. **Неверное использование теста.** Тестовые метрики не должны управлять выбором эпохи или гиперпараметров.
9. **Скрытые состояния без detach.** При обучении на последовательных фрагментах перенос графа между пакетами может вызывать нежелательное накопление градиентов и памяти.
10. **Утверждение «LSTM всегда лучше GRU».** Универсального превосходства нет; вывод обоснован только экспериментом для конкретной задачи.

15. Современные альтернативы и области применения

LSTM/GRU применяются к потокам датчиков, временным рядам, аудио, тексту и последовательностям действий. Хотя Transformer часто используется для больших языковых моделей благодаря механизму внимания и удобству параллельного обучения [12], рекуррентные модели продолжают быть полезными для небольших задач, ограниченных ресурсов и потоковой обработки. Модели временных рядов также следует сравнивать с классическими статистическими подходами и простыми базовыми прогнозами.

16. Итоги лекции

- LSTM использует отдельные состояния памяти c_t и вывода h_t, управляемые вентилями забывания, записи и вывода.
- GRU объединяет механизмы памяти в одном скрытом состоянии с reset/update gates.
- Прямой путь обновления памяти может облегчить перенос градиента, но не гарантирует сохранения любых длинных зависимостей.
- Учебные формулы GRU и реализация PyTorch отличаются порядком некоторых операций; сравнение требует осторожности.
- При одинаковых d_x, d_h стандартная GRU обычно содержит меньше параметров, чем LSTM.

- Корректное сравнение требует временного разбиения, валидации, baseline и контроля утечки данных.

17. Контрольные вопросы

1. Почему обучение простой RNN затрудняется при длинных последовательностях?
2. Что означает произведение якобианов во времени?
3. Каков математический смысл сигмоидного вентиля?
4. Почему значения gate не следует считать жёсткими бинарными переключателями?
5. Что называется состоянием ячейки c_t в LSTM?
6. Чем состояние c_t отличается от h_t?
7. Запишите формулу forget gate и объясните обозначения.
8. Запишите формулу input gate.
9. Что называется кандидатом состояния памяти?
10. Выведите формулу обновления c_t.
11. Какой смысл имеет output gate?
12. Почему скрытое состояние h_t не обязательно равно c_t?
13. Может ли модуль компоненты c_t превышать единицу?
14. Выполните численный расчёт c_t и h_t для заданных вентилей.
15. Какова локальная производная прямого пути $c_{t-1} \mapsto c_t$?
16. Почему полный градиент LSTM не сводится к произведению forget gates?
17. Как меняется прямой градиентный канал при $f_t = 0.99$?
18. Перечислите размерности четырёх матриц входных весов LSTM.
19. Выведите формулу числа параметров учебной LSTM.
20. Почему подсчёт числа параметров PyTorch отличается от учебного?
21. Что означают update gate и reset gate GRU?
22. Почему в стандартной GRU отсутствует отдельное состояние c_t?
23. Запишите полный цикл обновления GRU.
24. В чём состоят альтернативные соглашения о смысле z_t?
25. Чем PyTorch GRU отличается от записи с $U_h(r_t \odot h_{t-1})$?
26. Подсчитайте число параметров GRU с $d_x = 4$, $d_h = 10$.
27. Каковы различия между многослойной и двунаправленной RNN?
28. Почему bidirectional architecture может нарушить причинность прогноза?
29. Какие формы имеют `output`, `h_n`, `c_n` при `batch_first=True`?

30. Почему normalization выполняется по обучающей части временного ряда?
31. Зачем необходим baseline при сравнении LSTM и GRU?
32. Почему train MSE недостаточно для оценки полезности модели?
33. Какие последствия имеет подбор гиперпараметров по тесту?
34. Когда LSTM предпочтительнее GRU и почему нельзя ответить универсально?
35. Какие свойства Transformer отличают её от рекуррентных сетей?

18. Задания для самостоятельной работы

1. **Ручной расчёт LSTM.** Для $c_{t-1} = 0.6$, $f_t = 0.8$, $i_t = 0.3$, $\widetilde{c}_t = -0.5$, $o_t = 0.9$ вычислите c_t и h_t.
2. **Ручной расчёт GRU.** Для $h_{t-1} = 0.2$, $z_t = 0.7$, $\widetilde{h}_t = -0.4$ найдите новый h_t в соглашении данной лекции.
3. **Размерности.** При $d_x = 8$, $d_h = 32$ рассчитайте число параметров стандартных LSTM и GRU в PyTorch.
4. **Градиентный путь.** Сравните 0.9^{10}, 0.9^{50}, 0.99^{50}, 0.999^{50} и объясните границы интерпретации.
5. **Реализация.** Напишите одношаговую LSTM на NumPy и проверьте вручную рассчитанные значения.
6. **Сравнение.** Обучите LSTM и GRU с одинаковым скрытым размером и затем с примерно одинаковым числом параметров. Объясните разницу двух способов сравнения.
7. **Эксперимент по памяти.** Постройте синтетическую задачу копирования сигнала через заданную задержку и сравните качество при задержках 5, 15 и 30 шагов.
8. **Временной ряд.** Измените длину окна m в практическом коде на 6, 12 и 24; проанализируйте ошибку и скорость обучения.
9. **Регуляризация.** Добавьте второй рекуррентный слой и dropout; сравните кривые обучения с однослойной моделью.
10. **Отчёт и защита.** Подготовьте 5–7 слайдов: цель, данные, архитектура, параметры, baseline, итоговые метрики, ошибки и выводы.

Требования к отчёту: постановка задачи, математическая модель, код, параметры обучения, временное разбиение, таблица метрик, график истинного и прогнозируемого ряда, обсуждение ограничений.

19. Литература

Список литературы

[1] Траск Э. *Грокаем глубокое обучение* (Grokking Deep Learning). Manning, 2019.

[2] Рашид Т. *Создаём нейронную сеть* (Make Your Own Neural Network). 2016.

[3] Géron A. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. 3rd ed. O'Reilly, 2022.

[4] Chollet F. *Deep Learning with Python*. 2nd ed. Manning, 2021.

[5] Stevens E., Antiga L., Viehmann T. *Deep Learning with PyTorch*. Manning, 2020.

[6] Goodfellow I., Bengio Y., Courville A. *Deep Learning*. MIT Press, 2016. https://www.deeplearningbook.org/

[7] Nielsen M. A. *Neural Networks and Deep Learning*. Determination Press, 2015. http://neuralnetworksanddeeplearning.com/

[8] Bengio Y., Simard P., Frasconi P. Learning Long-Term Dependencies with Gradient Descent Is Difficult. *IEEE Transactions on Neural Networks* 5(2), 157–166 (1994). https://doi.org/10.1109/72.279181

[9] Hochreiter S., Schmidhuber J. Long Short-Term Memory. *Neural Computation* 9(8), 1735–1780 (1997). https://doi.org/10.1162/neco.1997.9.8.1735

[10] Gers F. A., Schmidhuber J., Cummins F. Learning to Forget: Continual Prediction with LSTM. *Neural Computation* 12(10), 2451–2471 (2000). https://doi.org/10.1162/089976600300015015

[11] Cho K. et al. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. *EMNLP*, 2014. https://aclanthology.org/D14-1179/

[12] Vaswani A. et al. Attention Is All You Need. *NeurIPS*, 2017. https://arxiv.org/abs/1706.03762

[13] PyTorch Documentation. *torch.nn.LSTM*. https://docs.pytorch.org/docs/stable/generated/torch.nn.LSTM.html

[14] PyTorch Documentation. *torch.nn.GRU*. https://docs.pytorch.org/docs/stable/generated/torch.nn.GRU.html