ИСКУССТВЕННЫЕ НЕЙРОННЫЕ СЕТИ

Лекция 9

Рекуррентные нейронные сети (RNN)

Последовательности, развёртка во времени, BPTT и затухающие градиенты

Преподаватель: Аршын Алтыбай

Кафедра компьютерной инженерии,
Алматинский технологический университет, Алматы, Казахстан

Учебный год: 2026

> **Связь с предыдущими лекциями.** В лекциях 1–6 рассмотрены нейрон, многослойная сеть, функции потерь, алгоритм обратного распространения и оптимизация. Лекции 7–8 посвящены свёрточным сетям и переносу обучения для изображений. Теперь данные имеют естественный *порядок во времени*: каждое наблюдение зависит от предыдущих. Для обработки таких данных вводится рекуррентное состояние [6, 7, 4].

1. Цели и результаты обучения

Цель: понять, почему обычного MLP недостаточно для произвольных последовательностей, вывести уравнения простой рекуррентной сети, получить градиенты методом обратного распространения во времени и реализовать небольшую модель для прогнозирования. По завершении занятия студент должен уметь:

1. различать временные ряды, последовательности токенов и многомерные траектории;
2. объяснять разделение параметров между моментами времени;
3. записывать прямой проход RNN в покомпонентной и матричной формах;
4. вычислять размерности, число параметров и стоимость обработки;
5. выводить рекуррентные формулы BPTT, а не только использовать готовую библиотеку;
6. объяснять затухание и взрыв градиентов через произведения якобианов;

7. различать RNN, LSTM, GRU и Transformer на концептуальном уровне;
8. строить причинно-корректное разбиение временного ряда на обучение и тест.

План: задачи и представление данных; простая RNN; численные расчёты; функции потерь; BPTT; устойчивость; практическая реализация; оценка качества; вопросы и литература.

2. Почему последовательности требуют особой архитектуры?

2.1 Структура последовательности

Пусть дана последовательность $x_1, x_2, \ldots, x_T$, где $x_t \in \mathbb{R}^{d_x}$. Индекс t обозначает позицию в последовательности, чаще всего время. В отличие от обычного набора признаков $\{x_i\}$, перестановка элементов может изменить смысл задачи: предложения «собака укусила человека» и «человек укусил собаку» содержат близкий набор слов, но различаются порядком.

Для временного ряда x_t может быть вектором: температура, влажность и потребление энергии в один час. Для текста x_t — числовое представление слова или подсловного токена. Для аудио x_t — фрейм спектральных признаков. Во всех случаях порядок наблюдений имеет значение [2, 4].

2.2 Четыре схемы отображения

1. **One-to-one**: обычная классификация изображения, одно наблюдение — один ответ.
2. **Many-to-one**: анализ тональности текста, весь текст — одна метка.
3. **One-to-many**: условная генерация последовательности по одному входному вектору.
4. **Many-to-many**: разметка каждого слова или прогноз для каждого шага.

Фактическая архитектура может включать encoder, decoder, pooling или дополнительные выходные слои. Схема описывает формат входа и ответа, а не полностью определяет модель.

> **Принцип причинности** При прогнозировании x_{t+1} разрешено использовать только информацию, доступную к моменту t. Нельзя случайно передать сети будущую часть временного ряда через нормали-

зацию, скользящее окно или перемешанное разбиение.

3. От полносвязного слоя к памяти состояния

Обычная модель $y = f_\theta(x)$ не хранит информацию между двумя независимыми вызовами. Можно объединить последние m наблюдений: $y_t = f(x_{t-m+1}, \dots, x_t)$, но размер входа фиксируется заранее. RNN строит сжатое состояние h_t и повторно использует одну функцию перехода:

$$h_t = \phi(W_{xh}x_t + W_{hh}h_{t-1} + b_h), \quad y_t = W_{hy}h_t + b_y. \tag{1}$$

Здесь h_{t-1} — информация о прошлом, а h_t — обновлённое состояние. Для задач классификации y_t может обозначать *логиты*, а вероятности получаются отдельным применением sigmoid или softmax [6].

3.1 Размерности и разделение параметров

Для $x_t \in \mathbb{R}^{d_x}$ и $h_t \in \mathbb{R}^{d_h}$:

$$W_{xh} \in \mathbb{R}^{d_h \times d_x}, \qquad W_{hh} \in \mathbb{R}^{d_h \times d_h}, \qquad b_h \in \mathbb{R}^{d_h}.$$

Для выхода из d_y компонент матрица $W_{hy} \in \mathbb{R}^{d_y \times d_h}$. Полное число параметров простой RNN с линейной головой:

$$N = d_h d_x + d_h^2 + d_h + d_y d_h + d_y. \tag{2}$$

Пример. Если $d_x = 3$, $d_h = 8$, $d_y = 2$, то $N = 24 + 64 + 8 + 16 + 2 = 114$. При обработке 10 или 100 временных шагов число *параметров* остаётся равным 114. Однако затраты вычислений и объём промежуточных активаций для BPTT растут с длиной последовательности.

Не путайте *параметры* модели с *узлами вычислительного графа*. Каждая развёрнутая копия RNN во времени использует одни и те же W_{xh}, W_{hh}, b_h; это не T независимых сетей.

4. Математическая развёртка RNN во времени

При $h_0 = 0$ первые три шага равны

$$h_1 = \phi(W_{xh}x_1 + b_h),$$
$$h_2 = \phi(W_{xh}x_2 + W_{hh}h_1 + b_h),$$
$$h_3 = \phi(W_{xh}x_3 + W_{hh}h_2 + b_h).$$

Величина h_3 зависит от всех x_1, x_2, x_3, хотя влияние ранних наблюдений может ослабевать. В вычислительном графе переход $h_{t-1} \mapsto h_t$ повторяется T раз. Этот граф делает возможным применение цепного правила к весам, общим для всех шагов.

4.1 Выбор начального состояния

Чаще берут $h_0 = 0$. Возможны обучаемый вектор h_0 или перенос состояния из предыдущего блока последовательности. Последний вариант требует контролировать границы независимых эпизодов: состояние от одного пациента или документа нельзя бездумно переносить к другому.

4.2 Функции активации

Классическая простая RNN использует tanh, поэтому $h_{t,j} \in (-1, 1)$. Иногда применяется ReLU, но при неудачной инициализации и длинных последовательностях возникают проблемы динамической устойчивости. Функция sigmoid используется для вероятностей или управляющих ворот, однако на скрытом состоянии её ограниченный диапазон и насыщение могут усложнить обучение.

5. Подробный численный пример прямого распространения

Рассмотрим *скалярную* RNN (один вход, один скрытый нейрон):

$$h_t = \tanh(0.5x_t + 0.2h_{t-1}), \qquad h_0 = 0, \qquad (x_1, x_2, x_3) = (1, 0, -1).$$

Тогда $a_1 = 0.5$ и

$$h_1 = \tanh(0.5) \approx 0.462117.$$

Для второго шага:

$$a_2 = 0.2h_1 \approx 0.0924234, \qquad h_2 = \tanh(a_2) \approx 0.092161.$$

Для третьего шага:

$$a_3 = -0.5 + 0.2h_2 \approx -0.481568, \qquad h_3 \approx -0.447498.$$

Последнее значение зависит не только от $x_3 = -1$, но и от предыдущих x_1, x_2 через h_2. Если же заменить h_2 на ноль, ответ будет $\tanh(-0.5) = -0.462117$, то есть контекст действительно влияет на выход.

Пример 1. Ручная развёртка RNN средствами NumPy

```python
import numpy as np
x = np.array([1.0, 0.0, -1.0])
h = 0.0
for t, xt in enumerate(x, start=1):
    a = 0.5 * xt + 0.2 * h
    h = np.tanh(a)
    print(t, f"a={a:.6f}", f"h={h:.6f}")
```

Ожидаемый вывод

```
1 a=0.500000 h=0.462117
2 a=0.092423 h=0.092161
3 a=-0.481568 h=-0.447498
```

6. Функции потерь для последовательностей

6.1 Классификация всей последовательности

Если задана одна метка c для всей последовательности, берут финальное скрытое состояние h_T и логиты $z = Wh_T + b$. Для K классов

$$p_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}, \qquad \mathcal{L} = -\log p_c.$$

Вместо h_T иногда применяют pooling по всем состояниям, внимание или двунаправленный encoder, если задача допускает обработку полной последовательности целиком.

6.2 Предсказание на каждом шаге

Если на каждом шаге задана цель r_t, используют среднюю функцию потерь

$$\mathcal{L} = \frac{1}{\sum_{t=1}^{T} m_t} \sum_{t=1}^{T} m_t \ell(y_t, r_t), \tag{3}$$

где $m_t \in \{0, 1\}$ — маска действительных позиций, не заполненных padding. Для регрессии применяют MSE/MAE, для классов — cross-entropy. Формула требует хотя бы одного ненулевого элемента маски.

6.3 Teacher forcing

В обучении autoregressive decoder иногда на следующий шаг передают истинный предыдущий токен, а не собственное предсказание сети. Это ускоряет обучение, но различие между обучением и генерацией может приводить к накоплению ошибок. При использовании teacher forcing важно явно описывать, какая информация была доступна в момент выдачи каждого предсказания.

7. Обратное распространение во времени

7.1 Что означает Backpropagation Through Time?

Backpropagation Through Time — применение обратного распространения к графу RNN, развёрнутому по шагам $t = 1, \dots, T$ [6, 7]. Пусть

$$a_t = W_{xh} x_t + W_{hh} h_{t-1} + b_h, \qquad h_t = \phi(a_t),$$

и общая функция потерь имеет вид $\mathcal{L} = \sum_{t=1}^{T} \ell_t(h_t)$, причём ℓ_t может быть нулевой для шагов без непосредственной цели.

Обозначим $g_t = \nabla_{h_t} \ell_t(h_t)$ — *локальный* вклад ошибки на шаге t, и $\delta_t = \nabla_{a_t} \mathcal{L}$. Благодаря цепному правилу:

$$\boxed{\delta_t = \Big(g_t + W_{hh}^{\mathsf{T}} \delta_{t+1} \Big) \odot \phi'(a_t), \qquad \delta_{T+1} = 0.} \tag{4}$$

Знак $\odot$ обозначает покомпонентное умножение. Смысл уравнения: градиент в момент t складывается из ошибки текущего шага и влияния *всех будущих шагов*, проходящего через h_t.

7.2 Градиенты общих параметров

Поскольку одни и те же веса используются на каждой позиции, градиенты по параметрам являются суммами по времени:

$$\frac{\partial \mathcal{L}}{\partial W_{xh}} = \sum_{t=1}^{T} \delta_t x_t^{\mathsf{T}}, \tag{5}$$

$$\frac{\partial \mathcal{L}}{\partial W_{hh}} = \sum_{t=1}^{T} \delta_t h_{t-1}^{\mathsf{T}}, \tag{6}$$

$$\frac{\partial \mathcal{L}}{\partial b_h} = \sum_{t=1}^{T} \delta_t. \tag{7}$$

Для линейной выходной головы $y_t = W_{hy} h_t + b_y$ соответствующие градиенты также суммируются по времени, но в g_t необходимо предварительно учесть производную выходной функции потерь.

> **Ключевое различие** Forward pass последовательно вычисляет $h_1, \dots, h_T$. BPTT идёт в обратном направлении: $\delta_T, \dots, \delta_1$. При этом параметры разделяются по времени, поэтому их градиенты *суммируются*, а не рассчитываются независимо для каждой копии сети.

8. Ручной расчёт градиента BPTT

Возьмём линейную активацию $\phi(a) = a$ только для прозрачного ручного расчёта:

$$h_t = w x_t + u h_{t-1}, \qquad x_1 = 1, \quad x_2 = 2, \quad h_0 = 0, \quad w = 0.5, \quad u = 0.1.$$

Тогда $h_1 = 0.5$ и $h_2 = 0.5 \cdot 2 + 0.1 \cdot 0.5 = 1.05$. Пусть цель $r = 1$ и потеря только на последнем шаге:

$$\mathcal{L} = \frac{1}{2}(h_2 - r)^2 = \frac{1}{2}(0.05)^2 = 0.00125.$$

Локальный градиент $\delta_2 = h_2 - r = 0.05$. Для первого шага $\delta_1 = u\delta_2 = 0.005$. Следовательно,

$$\frac{\partial \mathcal{L}}{\partial w} = \delta_1 x_1 + \delta_2 x_2 = 0.005 \cdot 1 + 0.05 \cdot 2 = 0.105,$$

$$\frac{\partial \mathcal{L}}{\partial u} = \delta_1 h_0 + \delta_2 h_1 = 0 + 0.05 \cdot 0.5 = 0.025.$$

При шаге $\eta = 0.1$ веса станут $w_{\text{new}} = 0.4895$, $u_{\text{new}} = 0.0975$. Заметьте, что $\partial \mathcal{L}/\partial w$ содержит два слагаемых: изменение w воздействует на h_2 непосредственно и опосредованно через h_1.

Пример 2. Проверка BPTT посредством конечных разностей

```python
import numpy as np

def loss(w, u):
    h1 = w * 1.0
    h2 = w * 2.0 + u * h1
    return 0.5 * (h2 - 1.0)**2

eps = 1e-6
w, u = 0.5, 0.1
num_dw = (loss(w+eps,u)-loss(w-eps,u))/(2*eps)
num_du = (loss(w,u+eps)-loss(w,u-eps))/(2*eps)
print(round(num_dw, 6), round(num_du, 6))
```

Ожидаемый вывод

```
0.105 0.025
```

9. Почему градиенты затухают или взрываются?

Для фиксированной траектории скрытых состояний якобиан перехода имеет вид

$$J_t = \frac{\partial h_t}{\partial h_{t-1}} = \operatorname{diag}(\phi'(a_t))W_{hh}. \tag{8}$$

Влияние раннего состояния h_k на позднее h_T выражается произведением

$$\frac{\partial h_T}{\partial h_k} = J_T J_{T-1} \cdots J_{k+1}.$$

Для операторной нормы справедлива оценка

$$\left\| \frac{\partial h_T}{\partial h_k} \right\| \le \prod_{t=k+1}^{T} \|J_t\|. \tag{9}$$

Если все нормы ограничены числом $\rho < 1$, верхняя оценка убывает как ρ^{T-k}. Это *достаточное условие* затухания. Большие нормы отдельных матриц допускают рост, но сами по себе не доказывают взрыв градиента: важны направления векторов и произведение матриц. Для tanh производная не превышает 1 по модулю и близка к нулю при насыщении [8, 9].

9.1 Скалярная иллюстрация

Если $h_t = uh_{t-1}$, то $\partial h_T/\partial h_0 = u^T$. При $u = 0.5$ и $T = 20$ получаем $0.5^{20} \approx 9.54 \cdot 10^{-7}$. При $u = 1.5$ и $T = 20$ имеем $1.5^{20} \approx 3325.26$. В первом случае ранний сигнал почти исчезает, во втором резко увеличивается.

9.2 Что помогает на практике?

1. **Gradient clipping:** ограничивает норму градиента при больших значениях, но не исправляет затухание.
2. **Ортогональная инициализация W_{hh}:** может стабилизировать локальное распространение сигнала.
3. **Gated architectures:** LSTM и GRU создают управляемые пути передачи состояния.
4. **Truncated BPTT:** ограничивает длину развёртки и затраты памяти, но отсекает часть дальних зависимостей.
5. **Нормализация и контроль learning rate:** помогают уменьшить нестабильность оптимизации.

10. Усечённое BPTT и обработка длинных рядов

При полном BPTT граф содержит все T переходов. Это может быть дорого по памяти. Truncated BPTT разбивает длинную последовательность на блоки длины K и периодически отключает историю градиентов через `detach()`. Числовое состояние можно передавать дальше, но производные через границу блока не протекают.

> Разбиение на блоки с `detach()` *меняет используемый градиент*: оно не эквивалентно точному полному BPTT. Если целевая зависимость находится значительно дальше K шагов, информация для её освоения может быть потеряна.

11. Сравнение RNN, LSTM и GRU

Простая RNN использует одно состояние h_t и одну операцию $\phi(Wx+Uh+b)$. LSTM вводит отдельно скрытое состояние h_t и ячейку памяти c_t, а также входные, забывающие и выходные ворота [9]. GRU объединяет часть управляющих механизмов, используя reset/update gates [10].

Свойство	Vanilla RNN	LSTM	GRU
Состояния	h_t	h_t, c_t	h_t
Ворота	Нет	Три группы	Две группы
Число параметров	Обычно меньше	Обычно больше	Часто между RNN и LSTM
Длинные зависимости	Труднее обучать	Специальный путь памяти	Управляемое обновление
Выбор	Короткие ряды, учебные примеры	Сложная память и временные ряды	Компактная альтернатива LSTM

11.1 Основные формулы LSTM

Для краткости объединим вход x_t и предыдущее состояние h_{t-1} в вектор $z_t = [x_t; h_{t-1}]$:

$$i_t = \sigma(W_i z_t + b_i), \qquad f_t = \sigma(W_f z_t + b_f),$$
$$o_t = \sigma(W_o z_t + b_o), \qquad \widetilde{c}_t = \tanh(W_c z_t + b_c),$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \widetilde{c}_t, \qquad h_t = o_t \odot \tanh(c_t).$$

Если f_t близко к единице, состояние c_t может сохраняться через многие шаги. Однако LSTM не гарантирует идеальную память: качество зависит от задачи, данных и обучения. Более подробное устройство LSTM/GRU станет предметом следующего занятия.

12. Как устроены тензоры в PyTorch

Модуль `torch.nn.RNN(input_size, hidden_size, num_layers, batch_first=True)` принимает вход формы (B, T, d_x), где B — размер пакета, T — число шагов, d_x — число признаков. При однонаправленной сети выход имеет форму (B, T, d_h), а финальные состояния слоёв — (L, B, d_h), где L — число рекуррентных слоёв. Для двунаправленной сети соответствующие размерности учитывают два направления.

Пример 3. Проверка размерностей PyTorch RNN

```python
import torch
from torch import nn

torch.manual_seed(42)
model = nn.RNN(input_size=3, hidden_size=8,
               num_layers=2, batch_first=True)
x = torch.randn(4, 10, 3)  # B=4, T=10, features=3
sequence_states, final_states = model(x)
print(tuple(sequence_states.shape))
print(tuple(final_states.shape))
```

Ожидаемый вывод

```
(4, 10, 8)
(2, 4, 8)
```

Важная техническая деталь Для `nn.RNN` с `nonlinearity='tanh'` параметры PyTorch организованы по слоям и направлениям. Реализация может содержать *два отдельных вектора смещения* для вход-

ного и рекуррентного преобразований, поэтому реальное число параметров `nn.RNN` следует подсчитывать через `sum(p.numel() for p in model.parameters())`; формула (2) относится к одной учебной ячейке с единым смещением.

13. Практика: прогноз следующего значения временного ряда

13.1 Корректная постановка эксперимента

Возьмём синтетический ряд $s_t = \sin(0.08t) + 0.25\sin(0.19t)$. Из окна длины $W = 20$ модель будет предсказывать s_{t+W}. Разделим исходную временную ось на три последовательные области: обучение, валидация и тест. Для демонстрации используем RNN с 16 скрытыми нейронами и линейный выход.

13.2 Полный воспроизводимый пример

Следующий код не скачивает внешние данные и может быть выполнен на CPU. Для независимого теста окна строятся отдельно внутри каждой временной области, поэтому окна обучения не включают будущие значения из валидации или теста.

Пример 4. Обучение простой RNN для прогноза ряда

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

torch.manual_seed(7)
np.random.seed(7)
t = np.arange(900, dtype=np.float32)
s = np.sin(0.08*t) + 0.25*np.sin(0.19*t)
train, valid, test = s[:600], s[600:750], s[750:]
mean, std = float(train.mean()), float(train.std())

def windows(part, length=20):
    values = (part - mean) / std
    X = np.stack([values[i:i+length]
                  for i in range(len(values)-length)])
    y = values[length:]
    return (torch.tensor(X[:,:,None], dtype=torch.float32),
            torch.tensor(y[:,None], dtype=torch.float32))

```

Пример 4 (продолжение)

```
Xtr, ytr = windows(train)
Xva, yva = windows(valid)
Xte, yte = windows(test)
loader = DataLoader(TensorDataset(Xtr,ytr),
                    batch_size=32, shuffle=True)

class SequenceForecaster(nn.Module):
    def __init__(self):
        super().__init__()
        self.rnn = nn.RNN(1, 16, batch_first=True)
        self.head = nn.Linear(16, 1)

    def forward(self, x):
        states, _ = self.rnn(x)
        return self.head(states[:, -1, :])

net = SequenceForecaster()
optimizer = torch.optim.Adam(net.parameters(), lr=0.005)
criterion = nn.MSELoss()

for epoch in range(80):
    net.train()
    for xb, yb in loader:
        optimizer.zero_grad()
        loss = criterion(net(xb), yb)
        loss.backward()
        nn.utils.clip_grad_norm_(net.parameters(), max_norm=1.0)
        optimizer.step()
    if (epoch+1) % 20 == 0:
        net.eval()
        with torch.no_grad():
            val_mae = torch.mean(torch.abs(net(Xva)-yva)).item()
        print(f'epoch={epoch+1}, val_MAE_scaled={val_mae:.4f}')

net.eval()
with torch.no_grad():
    pred = net(Xte).squeeze().numpy()*std + mean
    actual = yte.squeeze().numpy()*std + mean
print('Test MAE:', float(np.mean(np.abs(pred-actual))))
```

Значения MAE зависят от версии библиотек, платформы и случайных деталей оптимизации; поэтому здесь приводится исполняемый код, а не выдуманная заранее точная метрика. При защите лабораторной работы требуется показать *фактический* график потерь и тестовую MAE из собственного запуска.

13.3 С чем сравнивать нейросеть?

Простейший baseline — прогноз «следующее значение равно последнему наблюдению в окне»: $\widehat{s}_{t+1} = s_t$. Более сильный baseline может использовать скользящее среднее или авторегрессионную модель. Если RNN не превосходит простой baseline на будущих значениях, её усложнение не обосновано.

14. Оценка качества прогнозирования

Для тестовых пар $(y_i, \hat{y}_i)$ определим

$$\mathrm{MAE} = \frac{1}{n}\sum_i |y_i - \hat{y}_i|, \qquad \mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_i (y_i - \hat{y}_i)^2}.$$

MAE проще интерпретировать в единицах исходного ряда; RMSE сильнее штрафует крупные ошибки. MAPE не следует применять без оговорок, если истинные значения близки к нулю: относительная ошибка может становиться нестабильной.

14.1 Какие разбиения нельзя использовать?

1. **Случайное перемешивание временных окон** между train и test часто создаёт перекрывающиеся участки и завышает качество.
2. **Нормализация по всем данным**, включая тест, приводит к утечке статистики будущего.
3. **Выбор лучшей эпохи по тестовой ошибке** превращает тест в валидацию и искажает итоговую оценку.

Для реальных данных иногда используют rolling-origin или expanding-window validation: модель обучается на прошлом и проверяется на следующем временном отрезке.

15. Двунаправленные RNN и ограничения причинности

Двунаправленная сеть формирует состояния из прямого прохода $(x_1 \to x_T)$ и обратного $(x_T \to x_1)$. Итоговое представление может иметь вид $[\overrightarrow{h}_t; \overleftarrow{h}_t]$. Это полезно, если вся последовательность доступна заранее (например, разметка готового предложения). Однако при *онлайн-прогнозе будущего* обратный проход через будущие наблюдения недопустим: он нарушает причинность.

16. RNN и Transformer: где используются сегодня?

Transformer применяет механизм внимания для моделирования зависимостей без традиционного последовательного рекуррентного перехода [11]. В отличие от стандартной RNN, многие вычисления по позициям Transformer допускают параллелизацию при обучении. Вместе с тем RNN/GRU/LSTM сохраняют практическую ценность для небольших моделей, потоковых сигналов и задач, где важны ограниченная память и инкрементальная обработка. Нет универсального правила, что одна архитектура всегда лучше другой.

17. Типичные ошибки студентов и способы проверки

1. **Путаются оси:** подавать (T, B, d_x) при `batch_first=True`; проверяйте формы тензоров перед обучением.
2. **Забывают `zero_grad()`:** PyTorch по умолчанию накапливает градиенты между шагами.
3. **Неправильно выбирают выход:** для many-to-one берут последний валидный шаг, а не произвольную позицию padding.
4. **Передают граф между пакетами:** при stateful learning состояние следует отсоединять, когда это предусмотрено схемой обучения.
5. **Неверно нормализуют данные:** mean и std нужно оценивать только на train.
6. **Путают предсказание на один шаг с рекурсивным прогнозом:** при многократном применении модели ошибка накапливается.
7. **Сравнивают только train loss:** качество необходимо оценивать на будущей тестовой области.

18. Краткое резюме

- RNN моделирует последовательность через скрытое состояние и общие параметры перехода.
- При развёртке во времени получается вычислительный граф, к которому применяется BPTT.
- Градиенты общих весов суммируются по временным шагам.
- Произведения якобианов объясняют трудности с длинными зависимостями.
- LSTM и GRU предлагают управляемые пути передачи информации.
- Корректное временное разбиение и простой baseline необходимы для честного эксперимента.

19. Контрольные вопросы

1. Что называют последовательностью в машинном обучении?
2. Как порядок токенов влияет на смысл текста?
3. Приведите по одному примеру many-to-one и many-to-many.
4. Почему обычная MLP со скользящим окном не всегда заменяет RNN?

5. Запишите уравнение простой RNN и объясните каждый член.
6. Почему параметры RNN не зависят от числа временных шагов?
7. Определите размерности W_{xh} и W_{hh}.
8. Выведите формулу числа параметров однослойной учебной RNN.
9. Какую роль играет h_0?
10. Чем логиты отличаются от вероятностей?
11. В чём смысл развёртки вычислительного графа?
12. Что представляет собой BPTT?
13. Объясните рекуррентное уравнение для δ_t.
14. Почему градиент W_{hh} суммируется по шагам?
15. Чем отличается gradient checking от обычного backpropagation?
16. Что такое якобиан перехода RNN?
17. Почему возможно затухание градиента при длинных последовательностях?
18. Всегда ли норма $\|W_{hh}\| > 1$ гарантирует взрыв градиента?
19. Для чего используют gradient clipping?
20. Что меняется при truncated BPTT?
21. Какие состояния поддерживает LSTM?
22. Что обозначают input, forget и output gates?
23. Чем GRU отличается от простой RNN?
24. Почему двунаправленная RNN не подходит для обычного причинного онлайн-прогноза?
25. Каково назначение аргумента `batch_first=True`?
26. Что означают формы выходных тензоров в `nn.RNN`?
27. Почему normalization должна обучаться только по train?
28. Как обнаружить утечку данных во временном ряду?
29. Зачем нужен наивный baseline?
30. В чём концептуальное различие RNN и Transformer?

20. Задания для самостоятельной работы

1. **Ручной расчёт (базовый).** Для $h_t = \tanh(0.3x_t + 0.4h_{t-1})$, $h_0 = 0$ и $x = (1, -1, 2)$ вычислите все h_t с точностью до 10^{-4}.
2. **Параметры сети.** Подсчитайте параметры учебной однослойной RNN при $d_x = 5$, $d_h = 12$, $d_y = 3$. Затем сравните с реальным `nn.RNN` PyTorch и объясните разницу.
3. **BPTT (повышенный).** Для двухшаговой линейной RNN самостоятельно выведите $\partial\mathcal{L}/\partial w$ и сравните с конечными разностями.
4. **Программирование.** Реализуйте forward pass скалярной RNN средствами NumPy и сравните с собственным ручным вычислением.
5. **Эксперимент.** Обучите RNN с $d_h = 8, 16, 32$ на синтетическом ряде; сравните валидационную MAE и время обучения.
6. **Оптимизация.** Сравните обучение с clipping и без него при двух learning rate; проанализируйте нормы градиентов.
7. **Baseline.** Сравните RNN с прогнозом последнего значения и со скользящим средним.
8. **Аналитическое задание.** Объясните, при каких условиях двунаправленная RNN допустима для анализа текста и недопустима для прогноза в реальном времени.
9. **Самостоятельный проект.** Возьмите собственный временной ряд, подготовьте временное разбиение, обучите модель и предоставьте график «истина–прогноз».
10. **Подготовка к следующей лекции.** Сопоставьте простую RNN с формулами ячейки LSTM и объясните роль состояния c_t.

Форма отчёта: постановка задачи; описание данных и разбиения; параметры сети; графики train/validation loss; метрики baseline и RNN на тесте; три главных вывода.

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

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