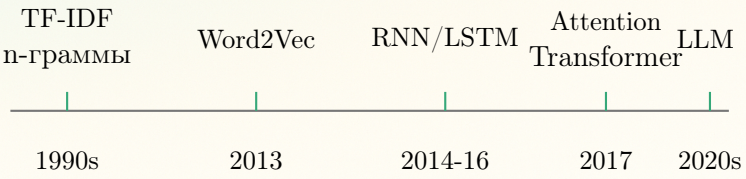
NLP: от Bag-of-Words до Attention

представления текста, Word2Vec, Conv1D, RNN/LSTM/GRU, языковые модели

Цели занятия

- Понять основные задачи NLP и типовые метрики.
- Научиться кодировать текст: one-hot, BoW, TF-IDF.
- Понять идею эмбеддингов и как их учит Word2Vec.
- Увидеть модели для последовательностей: Conv1D, RNN, LSTM, GRU.
- Понять, что такое языковая модель и perplexity.
- Понять мотивацию Seq2Seq и Attention (интуиция).

Timeline: как развивались подходы



- Сдвиг: от ручных признаков к обучаемым представлениям и end-to-end.

Шаг 0: токенизация — что это и зачем

Токенизация — это правило, которое превращает строку в последовательность токенов (единиц, с которыми работает модель).

Пример:

$$\text{“Мама мыла раму.”} \to [\text{мама}, \text{мыла}, \text{раму}, .]$$

- Почему это важно:
  - словарь токенов определяет, что вообще модель умеет различать;
  - токенизация влияет на длину последовательности (а значит на скорость/память);
  - в русском языке много форм слов $\Rightarrow$ выбор токенов особенно критичен.

Вариант 1: токены-слова (word-level)

Идея: токен = слово (обычно после приведения к нижнему регистру).

- Плюсы:
  - токены “понятные человеку”;
  - короткие последовательности (меньше шагов модели).
- Минусы:
  - OOV (out-of-vocabulary): встречаем слово, которого нет в словаре $\Rightarrow$ приходится заменять на <UNK>;
  - в русском: много форм (кошка/кошки/кошке/кошкой/...) $\Rightarrow$ словарь раздувается, редких форм много;
  - опечатки почти всегда превращаются в <UNK>.

Word-level хорошо для простых baselines (BoW/TF-IDF), но плохо масштабируется.

Вариант 2: токены-символы (char-level)

Идея: токен = символ.

$$\text{“кот”} \to [\text{к}, \text{о}, \text{т}]$$

- Плюсы:
  - почти нет OOV: любой текст можно разобрать на символы;
  - модель может учиться морфологии/суффиксам/опечаткам.
- Минусы:
  - последовательности становятся очень длинными;
  - сложнее выучить смысл на далёких расстояниях (много шагов).

Char-level иногда используют для специфических задач, но чаще берут subword.

Вариант 3: subword (BPE / WordPiece)

Идея: токен — не обязательно целое слово, а часть слова. Это позволяет собрать редкие слова из кусочков.

Пример (схематично):

$$\text{“программирование”} \to [\text{программ}, \text{ирова}, \text{ние}]$$

или (другой вариант разбиения):

$$\text{“кошечка”} \to [\text{кош}, \text{ечк}, \text{а}]$$

- Плюсы:
  - почти нет <UNK>: новое слово часто собирается из известных частей;
  - словарь разумного размера (например, 20–50k токенов);
  - хорошо для русского: общие корни/суффиксы повторяются.
- Минусы:
  - токены менее “читаемы” человеком;
  - последовательности длиннее, чем в word-level (но короче, чем в char-level).

Практические детали токенизации (что обычно делают)

- Нормализация: lowercasing, обработка ё/е, пробелов, кавычек.
- Пунктуация: часто выделяют отдельными токенами (. , ! ?).
- Числа/URL: иногда заменяют на специальные токены (<NUM>, <URL>).
- Спец-токены:
  - <PAD> — добивка до одинаковой длины в batch
  - <UNK> — неизвестный токен (стараются избегать)
  - <BOS>/<EOS> — начало/конец последовательности (для генерации)

Как мы будем представлять текст численно

Любая модель в итоге хочет числа. Поэтому мы строим отображение:

$$\text{текст} \;\to\; \text{токены} \;\to\; \text{вектор(ы)} \;\to\; \text{модель}$$

- Для классификации документа часто нужен один вектор на документ.
- Для последовательных моделей (RNN/Transformer) нужны вектора на каждый токен.

Начнём с самого простого: кодируем слово и потом соберём из слов документ.

One-hot: слово как номер (ID) в словаре

Пусть словарь V содержит все токены, например:

$$\mathrm{V} = \{\text{мама}, \text{мыла}, \text{раму}, \text{кот}, \text{стол}\}, \quad |\mathrm{V}| = 5$$

One-hot — это вектор длины $|\mathrm{V}|$, где стоит 1 на позиции слова:

$$\text{one-hot}(\mathrm{w}) \in \mathbb{R}^{|\mathrm{V}|}$$

слово	мама	мыла	раму	кот	стол
one-hot(мыла)	0	1	0	0	0
one-hot(кот)	0	0	0	1	0

- Плюс: это корректный способ “пронумеровать” слова.
- Минус: никакой семантики: “кот” и “стол” так же “далеки”, как “кот” и “кошка”.

Почему one-hot неудобен напрямую

- Размерность равна $|V|$. В реальных задачах $|V|$ легко $50\,000$ и больше.
- Вектор разреженный: почти все координаты $0 \Rightarrow$ хранить/умножать неэффективно.
- Главное: похожесть слов не закодирована.

Поэтому one-hot редко используют как “финальные признаки”, но он полезен как ступень к BoW и эмбеддингам.

Bag of Words (BoW): документ как сумма one-hot

Идея: документ — это мешок слов (порядок игнорируем).
Если документ d содержит токены $\mathrm{w}_1, \ldots, \mathrm{w}_\mathrm{T}$, то BoW-вектор можно получить так:

$$\phi(\mathrm{d}) = \sum_{\mathrm{t}=1}^{\mathrm{T}} \text{one-hot}(\mathrm{w}_\mathrm{t})$$

То есть $\phi_\mathrm{i}(\mathrm{d})$ = сколько раз встречилось i-е слово словаря.

- BoW переводит документ в один вектор $\in \mathbb{R}^{|\mathrm{V}|}$.
- Отлично подходит для линейных моделей (логрег, линейный SVM).

BoW на примере: два документа, один словарь

Пусть словарь $V = \{\text{кот}, \text{рыба}, \text{ест}, \text{спит}\}$.

$$d_1 = \text{“кот ест рыбу”}, _2 = \text{“кот спит”}$$

документ	кот	рыба	ест	спит
$\phi(d_1)$	1	1	1	0
$\phi(d_2)$	1	0	0	1

- Плюсы: просто и работает.
- Минусы: порядок потерян: “кот не ест” и “кот ест” становятся слишком похожими.

Ещё два минуса BoW (важные на практике)

- Частотные слова доминируют: “и”, “в”, “на”, “это” встречаются почти везде.
- Длины документов разные: длинный текст имеет больше счётчиков, даже если смысл тот же.

TF-IDF решает первую проблему (а нормировка/TF решают вторую).

TF-IDF: как “приглушить” частотные слова

Идея:

- слово важно, если оно частое в документе (TF),
- но редкое в корпусе (IDF).

Один из стандартных вариантов:

$$\mathrm{tf}(w, d) = \frac{\#(w \text{ в } d)}{|d|} \qquad \mathrm{idf}(w) = \log \frac{N}{\mathrm{df}(w) + 1}$$

$$\mathrm{tfidf}(w, d) = \mathrm{tf}(w, d) \cdot \mathrm{idf}(w)$$

Смысл: “стоп-слова” получают маленький вес, информативные — большой.

Мини-пример TF-IDF (с числами, без магии)

Корпус из N = 3 документов:

- d_1: “кот ест рыбу”
- d_2: “кот спит”
- d_3: “рыба вкусная”

Посчитаем df (в скольких документах встречается слово):

$$\mathrm{df}(\text{кот}) = 2, \quad \mathrm{df}(\text{рыба}) = 2, \quad \mathrm{df}(\text{спит}) = 1$$

Тогда (примерно, натуральный логарифм):

$$\mathrm{idf}(\text{кот}) = \log \frac{3}{2+1} = \log 1 = 0$$

$$\mathrm{idf}(\text{спит}) = \log \frac{3}{1+1} = \log 1.5 \approx 0.405$$

Вывод: “кот” почти не помогает различать документы (встречается часто), “спит” — помогает больше.

Что TF-IDF всё равно не умеет

- Порядок слов: “не хорошо” vs “хорошо” (в простом BoW/TF-IDF).
- Контекст/многозначность: “лук” (овощ) и “лук” (оружие) одинаковы.
- Семантика: “кот” не ближе к “кошка”, чем к “стол” в пространстве one-hot/BoW.

Нужны плотные представления: чтобы “похожие слова были рядом”. Это и есть эмбеддинги.

Эмбеддинги: делаем пространство, где “похожее рядом”

Эмбеддинг — вектор небольшой размерности:

$$e(w) \in \mathbb{R}^d, \quad d \ll |V|$$

- Хотим, чтобы “кот” и “кошка” были близко (например, по косинусной близости).
- Такие вектора удобно подавать в нейросети (RNN/CNN/Transformer).

Дальше: как обучать E? Один из классических способов — Word2Vec.

Эмбеддинг через one-hot: умножение на матрицу

Идея: one-hot — это “указатель на слово”, а матрица эмбеддингов E хранит вектора слов.

$$\mathrm{v}(\mathrm{w}) \in \mathbb{R}^{|\mathrm{V}|}, \in \mathbb{R}^{|\mathrm{V}| \times \mathrm{d}}, (\mathrm{w}) = \mathrm{v}(\mathrm{w})^{\top} \mathrm{E} \in \mathbb{R}^{\mathrm{d}}$$

Как это работает (на примере):
Пусть $|\mathrm{V}| = 5,\ \mathrm{d} = 3$, а слово w имеет индекс 2, тогда

$$\mathrm{v}(\mathrm{w}) = (0, 1, 0, 0, 0)$$

Тогда произведение $\mathrm{v}(\mathrm{w})^{\top}\mathrm{E}$ просто выбирает 2-ю строку матрицы E.

Матрица эмбеддингов E:

$$\mathrm{E} = \begin{pmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{pmatrix} (\mathrm{w}) = (\text{2-я строка})$$

На практике обычно не перемножают: делают lookup (взять строку по индексу), но смысл тот же.

Дистрибутивная гипотеза: от смысла к задаче обучения

> “Слова похожи, если встречаются в похожих контекстах.”

- Пример идеи: если рядом со словом часто встречаются “мурчит, хвост, корм”, то это, вероятно, что-то про животных.
- “кот” и “кошка” будут иметь похожие контексты $\Rightarrow$ хотим, чтобы их вектора были близко.

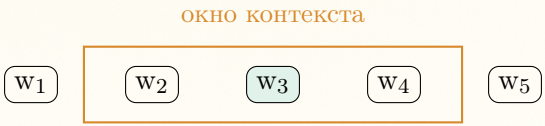
План: превратить это в обучающую задачу (предсказывать контекст по слову или наоборот).

Окно контекста: из текста делаем обучающие примеры

Берём окно размера c. Для каждого слова w_t считаем контекстом соседей:

$$\{w_{t-c}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+c}\}$$

Главная мысль: из одного предложения получаем много пар (центр, контекст).



центр w_3, контекст: w_1, w_2, w_4, w_5

Дальше: как именно формулируем задачу — CBOW или Skip-gram.

CBOW и Skip-gram: две постановки одной идеи

CBOW:
контекст $\rightarrow$ центр

- вход: слова вокруг
- выход: центральное слово
- часто быстрее, хорошо на частых словах

Skip-gram:
центр $\rightarrow$ контекст

- вход: центральное слово
- выход: слова вокруг
- часто лучше на редких словах

В дальнейшем будем мыслить Skip-gram: из центра предсказываем контекст.

Что именно учит Word2Vec (Skip-gram): “скор” пары слов

У каждого слова есть вектор. Для пары (центр w, контекст c) считаем скор:

$$s(w, c) = e(w)^\top e(c)$$

Интуиция:

- если w и c часто встречаются рядом $\Rightarrow$ хотим $s(w, c)$ большим
- если пара случайная $\Rightarrow$ хотим $s(w, c)$ маленьким

Осталось понять, как превратить “большой/маленький скор” в функцию потерь.

Negative Sampling: учим отличать настоящие пары от случайных

Для каждой положительной пары (w, c) берём k отрицательных $(w, n_1), \ldots, (w, n_k)$.
Вероятность “пара настоящая”:

$$P(\text{real} \mid w, t) = \sigma(s(w, t))$$

Loss для одного центра w:

$$L = -\log \sigma(s(w, c)) \; - \; \sum_{j=1}^{k} \log \sigma(-s(w, n_j))$$

- Почему быстрее, чем softmax: мы считаем всего $k + 1$ скоров, а не $|V|$.
- На практике k часто 5–20.

Как выглядит обучение Word2Vec (по шагам)

Шаги (Skip-gram + Negative Sampling):

1. Берём центр w и настоящий контекст c из окна $\Rightarrow$ положительная пара.
2. Берём k случайных слов $n_j \Rightarrow$ отрицательные пары.
3. Хотим, чтобы
$$s(w, c) \text{ рос}, \qquad s(w, n_j) \text{ уменьшались}.$$
4. Делаем шаг SGD и обновляем вектора только этих слов (это быстро).

Именно так “таблица эмбеддингов” E становится осмысленной по корпусу текстов.

Negative Sampling на примере

Пусть центр: кот. Настоящий контекст из окна: ест. Отрицательные слова: случайные.

пара	метка	что хотим от score
(кот, ест)	1	$s(\text{кот}, \text{ест})$ увеличить
(кот, таблица)	0	$s(\text{кот}, \text{таблица})$ уменьшить
(кот, молекула)	0	$s(\text{кот}, \text{молекула})$ уменьшить

Так мы учим “кот” быть ближе к словам из своего контекста и дальше от случайных.

Что получаем на выходе Word2Vec (и ограничения)

- Матрицу эмбеддингов E: вектор для каждого слова.
- Как использовать:
  - как входные признаки для моделей последовательностей (RNN/CNN/Transformer)
  - как простой baseline для документа: среднее по словам $\frac{1}{T}\sum_t e(w_t)$
- Ограничения:
  - один вектор на слово $\Rightarrow$ не различает смыслы по контексту (“лук”)

Контекстные эмбеддинги (BERT/Transformer) решают это, но это следующая глава.

Conv1D для текста: что подаём на вход

После Word2Vec (или другой embedding-матрицы) предложение — это последовательность векторов:

$$[w_1, \dots, w_T] \Rightarrow [e_1, \dots, e_T], \quad e_t \in \mathbb{R}^d$$

Можно думать, что это матрица:

$$X \in \mathbb{R}^{T \times d} \quad (\text{длина } T \times \text{размер эмбеддинга } d)$$

- по строкам — токены (позиции в тексте)
- по столбцам — координаты эмбеддинга

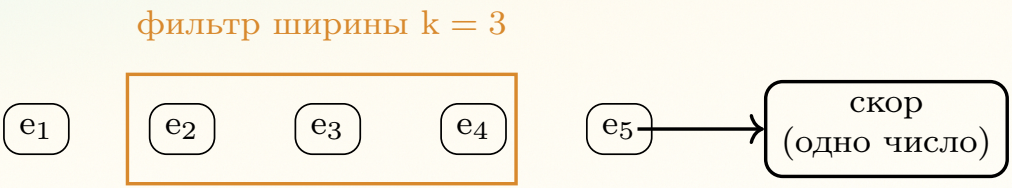
Conv1D будет “скользить” по оси времени $t = 1..T$, как по строкам матрицы.

Conv1D: фильтр как детектор n-грамм

Фильтр ширины k смотрит на k соседних токенов (пример: $k = 3$):

$$(e_t, e_{t+1}, e_{t+2})$$

и выдаёт число — насколько похоже на выученный шаблон.



окно (e_2, e_3, e_4): “нашли ли шаблон?”

- Один фильтр может выучить шаблон вроде “не очень”, другой — “совсем плохо”.

Feature map и max-pooling: “был ли шаблон где-то в тексте?”

Если мы двигаем фильтр по всем позициям, получаем последовательность скоров:

$$s_1, s_2, \ldots, s_{T-k+1}$$

Это называется feature map. Дальше часто делают:

$$\max(s_1, \ldots, s_{T-k+1})$$

- Max-pooling отвечает: “нашёлся ли шаблон где-нибудь?”
- Для классификации это удобно: текст может быть длинным, но важно наличие фразы/паттерна.

Плюс Conv1D: быстро и параллельно. Минус: контекст ограничен шириной окна.

Зачем RNN, если есть Conv1D?

Conv1D хорошо ловит локальные шаблоны (n-граммы), но:

- “если в начале было не, а в конце плохо” — далеко друг от друга;
- иногда смысл зависит от порядка и дальнего контекста.

Идея RNN: читать текст слева направо и хранить состояние-память.
RNN строит представление префикса: “что мы уже прочитали и что это значит”.

RNN: формула и смысл состояния

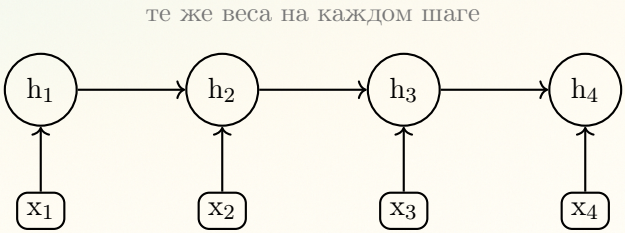
На шаге t получаем вход x_t (эмбеддинг токена) и прошлую память h_{t-1}:

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

- x_t — текущий токен
- h_{t-1} — “что мы помним про всё до $t-1$”
- h_t — обновлённая память

Важно: одни и те же матрицы W_x, W_h используются на каждом шаге.

RNN: развертка по времени (unrolling)



- Это одна и та же “ячейка”, применённая много раз.
- Поэтому RNN работает с любой длиной текста.

Как используют RNN в задачах (два режима)

- Many-to-one (классификация текста): берём последнее состояние

$$h_T \to \hat{y}$$

- Many-to-many (разметка токенов, NER): предсказываем на каждом шаге

$$h_t \to \hat{y}_t$$

В обоих случаях RNN полезна тем, что учитывает порядок токенов.

Почему RNN трудно учить на длинных текстах

При backprop через время градиент “идёт назад” через много шагов.
Интуиция без математики:

- если на каждом шаге мы умножаем градиент примерно на 0.9, то через 50 шагов получаем $0.9^{50} \approx 0.005 \Rightarrow$ почти ноль (затухание);
- если умножаем примерно на 1.1, то $1.1^{50} \approx 117 \Rightarrow$ взрыв.

Формально там стоит произведение матриц/производных, но смысл тот же: много раз перемножили.

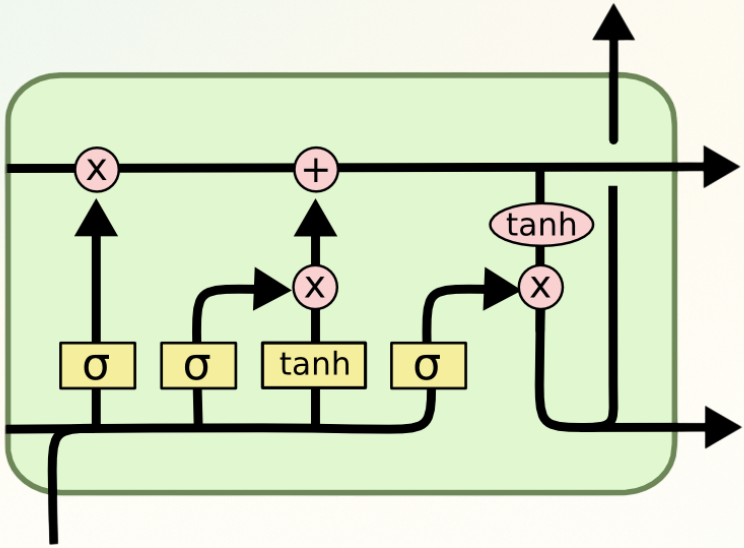
Практические “пластыри” для RNN

- Gradient clipping: ограничиваем норму градиента (боремся со взрывом).
- правильная инициализация, нормализации.
- Главное решение: использовать LSTM/GRU, где есть специальный путь памяти.

LSTM: память + гейты (зачем придумали)

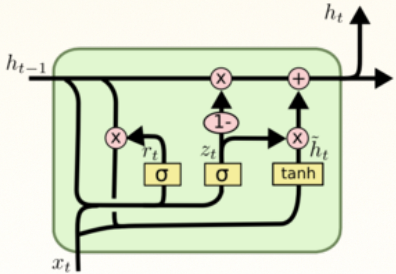
- Обычная RNN плохо держит долгие зависимости из-за затухания/взрыва градиента.
- В LSTM добавили отдельную память c_t и гейты (значения от 0 до 1), которые управляют потоком информации.
- Интуиция гейтов:
  - forget: что забыть из прошлого
  - input: что записать из текущего
  - output: что выдать наружу

Идея: есть “магистраль памяти” c_t, по которой информация может течь дальше.


x
+
tanh
x
x
σ
σ
tanh
σ

GRU: упрощённая версия LSTM

- GRU решает ту же проблему, что и LSTM: лучше переносит информацию на большие расстояния.
- Отличие: в GRU меньше гейтов и нет отдельного c_t как в LSTM (упрощённая схема).
- Практический эффект:
  - меньше параметров ⇒ часто быстрее обучается
  - качество часто сопоставимо с LSTM



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Сравнение: RNN vs GRU vs LSTM

Модель	Память	Парам.	Обучение	Когда использовать
RNN	слабая	мало	трудно на длинных	короткие последовательности, простые задачи
GRU	хорошая	средне	стабильнее RNN	базовый выбор: проще и быстрее, часто достаточно
LSTM	очень хорошая	больше	стабильно	когда важны долгие зависимости и есть ресурсы

- GRU/LSTM добавляют управляемую память и уменьшают проблемы градиента.
- Сейчас часто используют Transformer, но RNN-семейство всё ещё встречается.

Языковая модель (LM): что это

Языковая модель — это модель, которая умеет оценивать вероятность текста и предсказывать следующий токен.

$$P(w_1, \ldots, w_T) = \prod_{t=1}^{T} P(w_t \mid w_{<t})$$

- На каждом шаге модель выдаёт распределение вероятностей по словарю:

$$P(\cdot \mid w_{<t})$$

- Задача: по префиксу “Мама мыла ...” предсказать, что дальше вероятно “раму”.

Зачем LM: примеры применения

- Автодополнение: клавиатура/IDE/поиск
- Генерация текста: чат-боты, письма, истории
- Оценка “естественности”: сравнить две фразы и выбрать более вероятную
- Основа LLM: GPT-подобные модели — это большие LM

Если умеем хорошо предсказывать следующий токен, то можем генерировать, выбирая токены по вероятностям.

n-граммные модели: идея

Марковское приближение:

$$P(w_t \mid w_{<t}) \approx P(w_t \mid w_{t-n+1}, \ldots, w_{t-1})$$

- Bigram (n=2): следующий токен зависит только от предыдущего
- Trigram (n=3): зависит от двух предыдущих

То есть мы искусственно ограничиваем “память” модели длиной $n - 1$.

Как n-граммы оценивают вероятность (через счётчики)

Идея: частота ≈ вероятность.
Для биграмм:

$$P(w_t \mid w_{t-1}) \approx \frac{\#(w_{t-1}, w_t)}{\#(w_{t-1})}$$

- $\#(w_{t-1}, w_t)$ — сколько раз встречалась пара подряд
- $\#(w_{t-1})$ — сколько раз встречалось слово w_{t-1}

Это просто статистика по корпусу, без нейросетей.

Плюсы и минусы n-грамм

Плюсы:

- очень просто и быстро
- легко интерпретировать (счётчики)

Минусы:

- разреженность: многие n-граммы не встречались $\Rightarrow$ вероятность 0
- дальний контекст: модель “не помнит” дальше $n - 1$ токенов

Сглаживания (Laplace, Kneser–Ney) уменьшают нули, но дальний контекст всё равно ограничен.

Нейросетевая LM: что предсказываем на каждом шаге

Пусть x_t — эмбеддинг токена w_t. Модель строит состояние h_t и предсказывает следующий токен:

$$h_t = \mathrm{RNN}(h_{t-1}, x_t)$$

$$\hat{p}_{t+1} = \mathrm{softmax}(W h_t + b)$$

- $\hat{p}_{t+1}$ — вектор вероятностей размера $|V|$
- самый вероятный токен: $\arg\max \hat{p}_{t+1}$

Как обучают нейросетевую LM (идея loss)

Для каждого шага t у нас есть “правильный” следующий токен w_{t+1}. Мы хотим, чтобы модель дала ему большую вероятность.

$$L = -\sum_{t=1}^{T-1} \log P(w_{t+1} \mid w_{\le t})$$

- Это cross-entropy по предсказанию следующего токена.
- Обучаем градиентным спуском (backprop through time).

Нейросетевая LM может учитывать контекст “дольше”, чем n-граммы (хотя RNN не идеальны).

Perplexity: как оценивают языковую модель

Сначала считаем среднюю отрицательную лог-вероятность (энтропию на данных):

$$H = -\frac{1}{T}\sum_{t=1}^{T} \log P(w_t \mid w_{<t})$$

Перплексия:

$$ppl = \exp(H)$$

- меньше ppl $\Rightarrow$ модель лучше предсказывает текст
- ppl удобна как “одна цифра” качества LM

Интерпретация perplexity: “сколько вариантов в среднем”

- Если ppl $= 2$, это похоже на ситуацию: “в среднем модель выбирает между двумя равновероятными токенами”.
- Если ppl $= 100$, модель очень не уверена: “как будто 100 вариантов”.

Это не буквальное число вариантов, но интуиция очень полезна.

Seq2Seq: когда вход и выход — последовательности

Пример: перевод

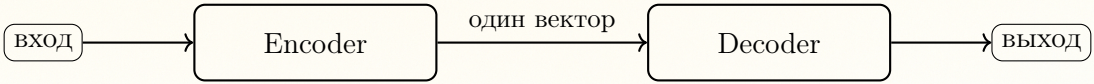
$$[\text{I, love, cats}] \rightarrow [\text{Я, люблю, кошек}]$$

- Encoder читает входную последовательность и строит внутреннее представление.
- Decoder генерирует выход по одному токену, опираясь на это представление.

Проблема классического Seq2Seq: “бутылочное горлышко”

Если encoder сжимает всю фразу в один вектор, то для длинных предложений это сложно:

- в начале мог быть важный факт, но к концу decoder его “не видит”
- особенно тяжело с длинными зависимостями и деталями



“всё сжали в один вектор” — теряем детали

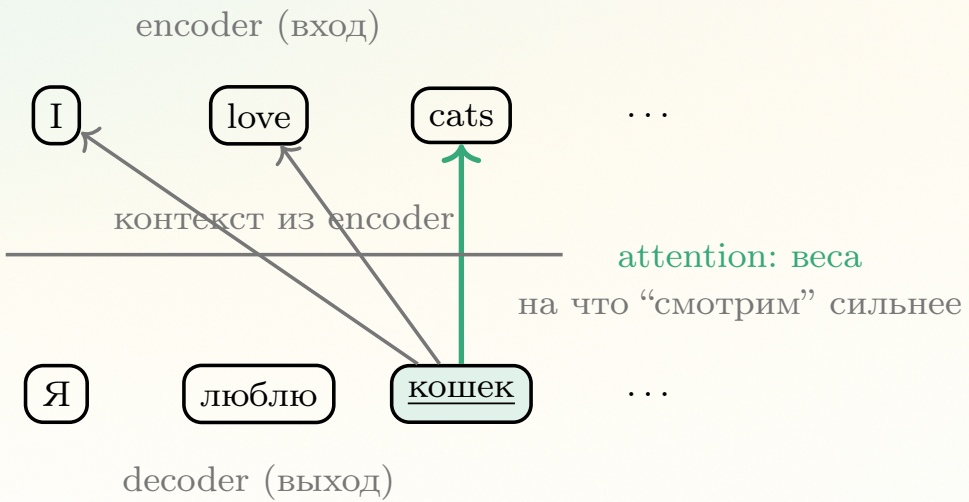
Attention: идея “подсматривать в нужные места”

Вместо одного вектора encoder выдаёт последовательность состояний $h_1, \ldots, h_T$.
Decoder на шаге t выбирает, на какие h_i смотреть сильнее:

- получаются веса внимания (attention weights)
- строится контекст как взвешенная сумма состояний encoder

Интуиция: “переводя слово сейчас, смотрим на нужное слово во входе”.

Attention: decoder-токен “смотрит” на encoder-токены



Мини-вопросы (олимпиадный стиль)

1. Почему BoW не различает “не хорошо” и “хорошо”? Как это исправить?
2. Почему полный softmax в Word2Vec дорогой? Что делает negative sampling?
3. Почему у RNN затухает/взрывается градиент (интуитивно через произведение)?
4. Что измеряет perplexity и почему меньше — лучше?

Итоги

- One-hot/BoW/TF-IDF — простые представления, но без контекста.
- Word2Vec учит семантические вектора через предсказание контекста.
- Conv1D ловит локальные шаблоны (n-граммы) быстро.
- RNN умеет учитывать порядок, но страдает от затухания/взрыва градиентов.
- LSTM/GRU добавляют управляемую память.
- Языковая модель оценивает P(текст), perplexity — мера качества.
- Seq2Seq + Attention — шаг к Transformer.

Границы моего языка означают границы моего мира.

— Людвиг Витгенштейн


The End