

# Методы машинного обучения

Шорохов С.Г.

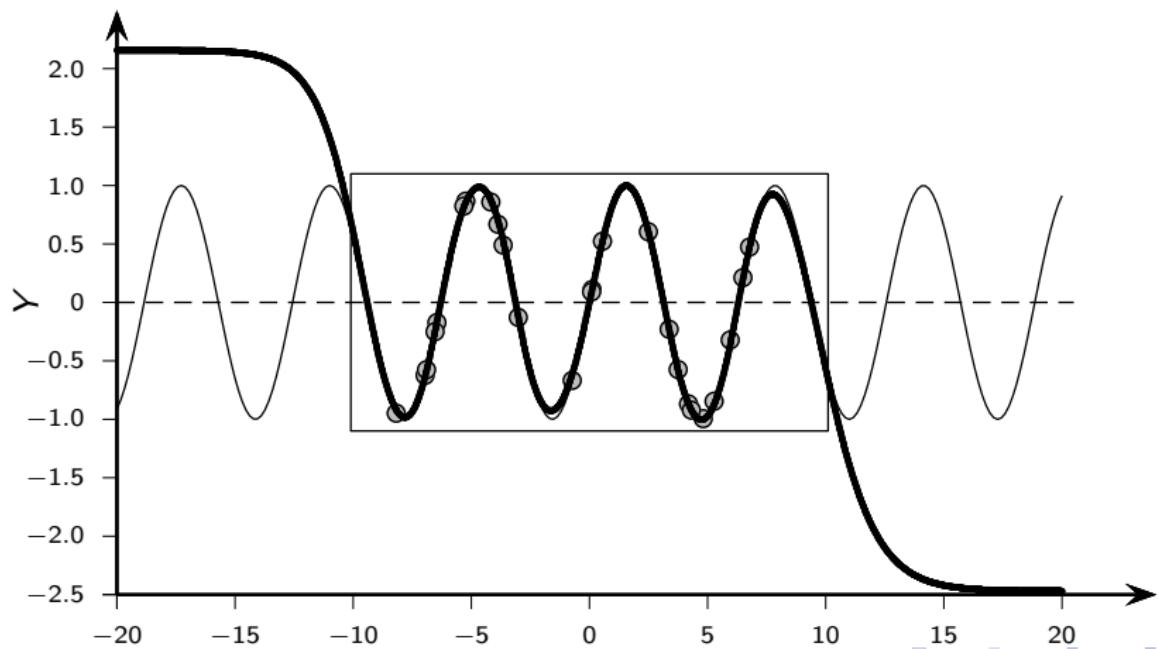
кафедра математического моделирования и искусственного интеллекта

Лекция 6. Рекуррентные нейронные сети



# Результат обучения сети MLP по синусоиде

Нейронная сеть MLP не может корректно прогнозировать функцию синуса за пределами диапазона обучения.





# Рекуррентные нейронные сети

Многослойные персептроны (MLP) представляют собой нейронные сети с прямой связью, в которых информация движется только в одном направлении, а именно от входного слоя к выходному через скрытые слои.

Напротив, **рекуррентные нейронные сети** (Recurrent Neural Networks, RNN) содержат петли обратной связи между двумя (или более) слоями, что делает их идеальными для обучения на основе входных данных в форме последовательностей. RNN обучаются, разворачивая (или развертывая) рекуррентные соединения, в результате чего получаются глубокие сети, оптимальные параметры которых можно определить с помощью алгоритма обратного распространения ошибки (backpropagation) и градиентных методов.

Задача сети RNN состоит в том, чтобы аппроксимировать функцию, которая предсказывает целевую выходную последовательность  $\mathcal{Y} = \langle \mathbf{y}_1, \dots, \mathbf{y}_\tau \rangle$ ,  $\mathbf{y}_t \in \mathbb{R}^p$  по заданной входной последовательности  $\mathcal{X} = \langle \mathbf{x}_1, \dots, \mathbf{x}_\tau \rangle$ ,  $\mathbf{x}_t \in \mathbb{R}^d$ . То есть прогнозируемые сетью RNN выходные данные (результат)  $\mathbf{o}_t \in \mathbb{R}^p$  для входных данных  $\mathbf{x}_t \in \mathbb{R}^d$  должны быть аналогичны или близки к целевому отклику  $\mathbf{y}_t \in \mathbb{R}^p$  для каждого момента времени  $t \in \{1, \dots, \tau\}$ .



# Применение сетей RNN

Нейронные сети RNN изначально предложены для обработки последовательностей данных, таких, как серии событий во времени, последовательные пространственные данные или последовательности слов (фразы), когда важен порядок следования объектов в последовательности.

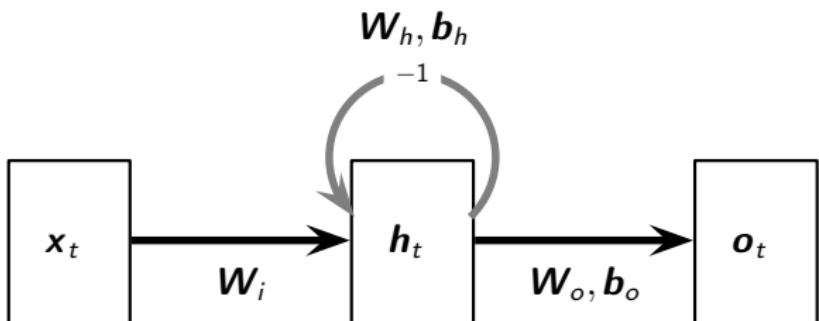
- Обработка текста на естественном языке (последовательности слов):
  - анализ текста
  - автоматический перевод
- Обработка аудио (последовательности звуков):
  - автоматическое распознавание речи
- Обработка видео (последовательности кадров):
  - прогнозирование следующего кадра на основе предыдущих
  - распознавание эмоций, жестов
- Обработка временных рядов (изменения параметров во времени):
  - погодные данные
  - анализ данных с финансовых рынков
- Управление динамическими системами (последовательности состояний динамических систем)

# Скрытые состояния RNN

Чтобы изучить зависимости между элементами входной последовательности, сеть RNN поддерживает последовательность  $t$ -мерных векторов **скрытых состояний** (скрытых векторов)  $\mathbf{h}_t \in \mathbb{R}^m$ , где  $\mathbf{h}_t$  фиксирует основные характеристики входных последовательностей до момента времени  $t$ . Скрытый вектор  $\mathbf{h}_t$  в момент времени  $t$  зависит от входного вектора  $\mathbf{x}_t \in \mathbb{R}^d$  в момент времени  $t$  и предыдущих скрытых векторов состояния  $\mathbf{h}_{t-1} \in \mathbb{R}^m$  для момента времени  $t - 1$  и вычисляется следующим образом:

$$\mathbf{h}_t = f^h (\mathbf{W}_i^T \mathbf{x}_t + \mathbf{W}_h^T \mathbf{h}_{t-1} + \mathbf{b}_h),$$

где  $f^h$  — функция активации скрытых состояний (обычно  $\tanh$  или  $\text{ReLU}$ ).





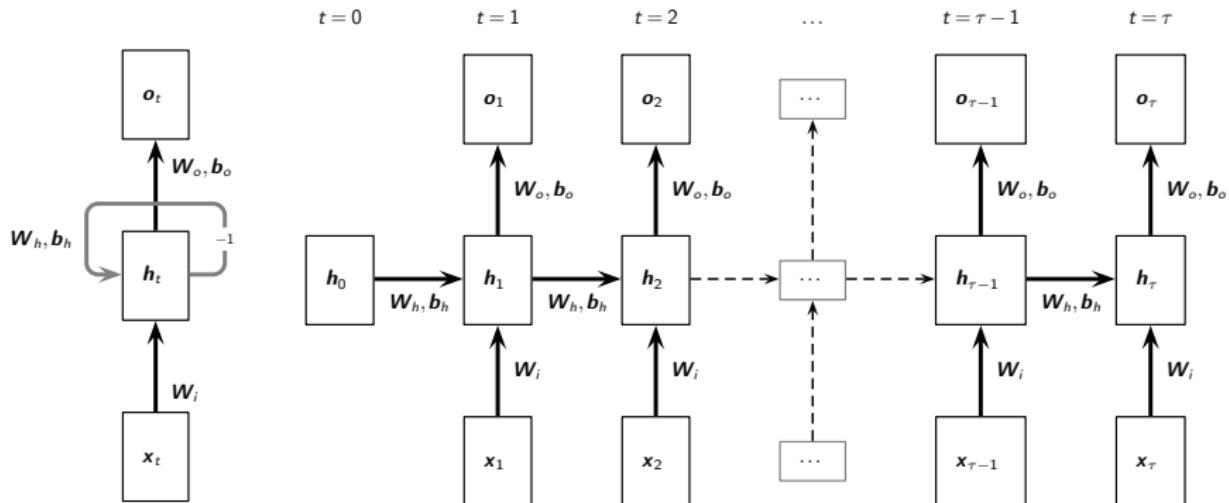
Важно отметить, что в нейронных сетях RNN весовые матрицы и векторы смещения **не зависят от времени  $t$** . Например, для скрытого слоя одна и та же матрица весов  $\mathbf{W}_h \in \mathbb{R}^{m \times m}$  и вектор смещения  $\mathbf{b}_h \in \mathbb{R}^m$  используются и обновляются при обучении модели на всех временных шагах  $t$ .

Это пример совместного использования параметров или привязки весов между различными слоями или компонентами нейронной сети. Аналогично, матрица входных весов  $\mathbf{W}_i \in \mathbb{R}^{d \times m}$  (смещение для входного слоя предполагается нулевым), матрица выходных весов  $\mathbf{W}_o \in \mathbb{R}^{m \times p}$  и соответствующий вектор смещения  $\mathbf{b}_o \in \mathbb{R}^p$  являются общими для всех моментов времени.

Это значительно уменьшает количество параметров, которые должны быть обучены сетью RNN, при этом предполагается, что все существенные характеристики последовательностей в обучающих данных могут быть зафиксированы за счет выбора этих общих параметров.

# Развертка сети RNN во времени

На рисунке ниже цикл обратной связи развернут для  $\tau$  временных шагов. Здесь также явно наблюдается распределение параметров сети RNN во времени, поскольку все весовые матрицы и векторы смещения не зависят от  $t$ .





# Ошибка (потери) при обучении сети RNN

Обучающие данные для сети RNN представляют собой набор  $\mathbf{D} = \{\mathcal{X}_i, \mathcal{Y}_i\}_{i=1}^n$ , состоящий из  $n$  входных последовательностей  $\mathcal{X}_i$  и соответствующих целевых последовательностей-откликов  $\mathcal{Y}_i$  с длиной последовательности, равной  $\tau$ . По заданным парам  $(\mathcal{X}, \mathcal{Y}) \in \mathbf{D}$  с  $\mathcal{X} = \langle \mathbf{x}_1, \dots, \mathbf{x}_\tau \rangle$  и  $\mathcal{Y} = \langle \mathbf{y}_1, \dots, \mathbf{y}_\tau \rangle$  нейронная сеть RNN должна обновлять параметры модели  $\mathbf{W}_i$ ,  $\mathbf{W}_h$ ,  $\mathbf{b}_h$ ,  $\mathbf{W}_o$ ,  $\mathbf{b}_o$  для входного, скрытого и выходного слоев, чтобы научиться предсказывать соответствующую выходную последовательность  $\mathcal{O} = \langle \mathbf{o}_1, \dots, \mathbf{o}_\tau \rangle$ .

Для обучения сети RNN вычисляется **ошибка** (error) или **потери** (loss) для прогнозируемых векторов отклика  $\mathbf{o}_t$  и фактических векторов отклика  $\mathbf{y}_t$  по всем временным шагам  $t = \overline{1, \tau}$ . Например, квадратичная ошибка (для задачи регрессии) задается как

$$\mathcal{E}_{\mathcal{X}} = \sum_{t=1}^{\tau} \mathcal{E}_{\mathbf{x}_t} = \frac{1}{2} \sum_{t=1}^{\tau} \|\mathbf{y}_t - \mathbf{o}_t\|^2 = \frac{1}{2} \sum_{t=1}^{\tau} \sum_{i=1}^p (y_{ti} - o_{ti})^2$$



# Ошибка (потери) при обучении сети RNN

С другой стороны, если на выходном слое используется функция активации softmax (решается задача многоклассовой классификации), то в качестве ошибки (потерь) применяется функция кросс-энтропии, заданная как

$$\mathcal{E}_{\mathcal{X}} = \sum_{t=1}^{\tau} \mathcal{E}_{\mathbf{x}_t} = - \sum_{t=1}^{\tau} \sum_{i=1}^p y_{ti} \ln(o_{ti}),$$

где  $\mathbf{y}_t = (y_{t1}, y_{t2}, \dots, y_{tp})^T \in \mathbb{R}^p$  и  $\mathbf{o}_t = (o_{t1}, o_{t2}, \dots, o_{tp})^T \in \mathbb{R}^p$ .

Обучая сеть на входной последовательности длины  $\tau$ , мы вначале развертываем сеть RNN на  $\tau$  шагов, отслеживая параметры сети, которые могут быть обучены при помощи стандартных прямого прохода и обратного распространения ошибки с учетом связей между слоями.



## Прямой проход по сети RNN (по времени)

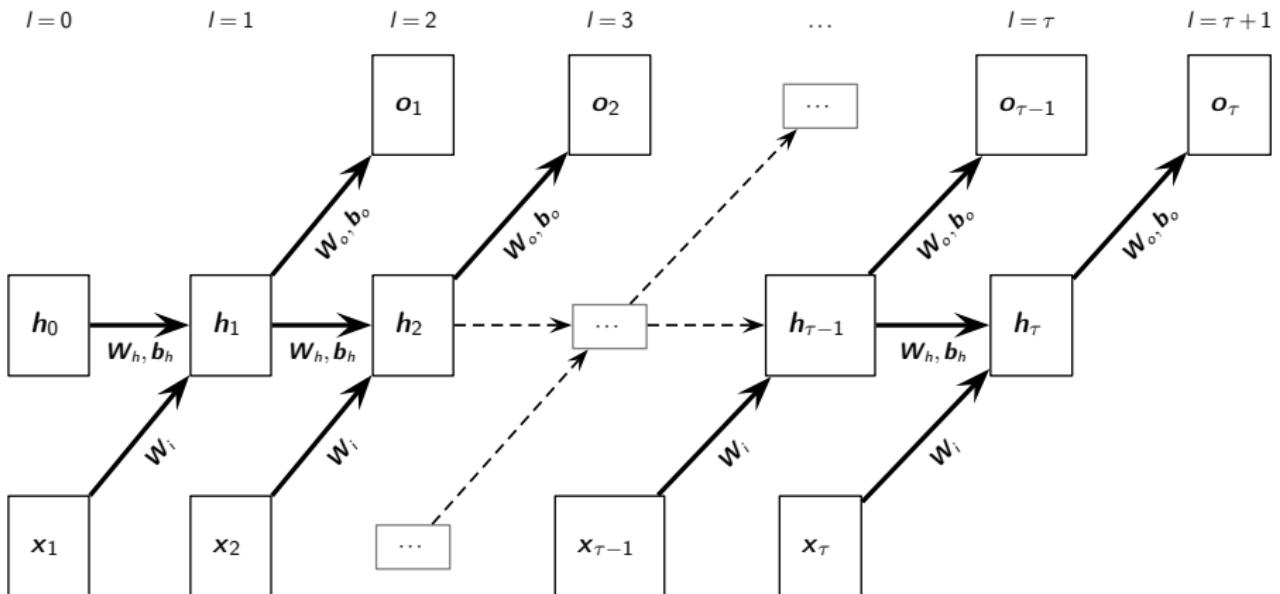
Процесс **прямого прохода** (feed forward) начинается в момент времени  $t = 0$  с входным начальным скрытым вектором состояния  $\mathbf{h}_0$ , который обычно устанавливается равным 0 или может быть задан на предыдущем шаге прогнозирования.

Для заданного текущего набора параметров сети RNN выход  $\mathbf{o}_t$  прогнозируется для каждого временного шага  $t = 1, 2, \dots, \tau$  по формуле

$$\begin{aligned}\mathbf{o}_t &= f^0(\mathbf{W}_0^T \mathbf{h}_t + \mathbf{b}_0) = f^0\left(\mathbf{W}_0^T \underbrace{f^h(\mathbf{W}_i^T \mathbf{x}_t + \mathbf{W}_h^T \mathbf{h}_{t-1} + \mathbf{b}_h)}_{\mathbf{h}_t} + \mathbf{b}_0\right) = \dots = \\ &= f^0\left(\mathbf{W}_0^T f^h\left(\mathbf{W}_i^T \mathbf{x}_t + \mathbf{W}_h^T \left(\dots \underbrace{f^h(\mathbf{W}_i^T \mathbf{x}_1 + \mathbf{W}_h^T \mathbf{h}_0 + \mathbf{b}_h)}_{\mathbf{h}_1} + \dots\right) + \mathbf{b}_h\right) + \mathbf{b}_0\right)\end{aligned}$$

Сеть RNN неявно делает прогноз для каждого префикса входной последовательности  $\{\mathbf{x}_t\}_{t=1}^\tau$ , поскольку выход  $\mathbf{o}_t$  зависит от всех предыдущих входных векторов  $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_t$ , но не от каких-либо будущих входных векторов  $\mathbf{x}_{t+1}, \dots, \mathbf{x}_\tau$ .

# Сеть RNN: шаги прямого прохода





# Градиент ошибки на выходных нейронах

Как только выходная последовательность  $\mathcal{O} = \langle \mathbf{o}_1, \mathbf{o}_2, \dots, \mathbf{o}_\tau \rangle$  сгенерирована, можно вычислить ошибку прогнозирования  $\mathcal{E}_{\mathcal{X}}$ , используя функцию квадратичной ошибки или кросс-энтропии. Ошибка прогнозирования  $\mathcal{E}_{\mathcal{X}}$ , в свою очередь, может быть использована для вычисления градиентов, которые распространяются обратно от выходного слоя к входному слою для каждого временного шага.

Обозначим через  $\mathcal{E}_{\mathbf{x}_t}$  ошибку (потери) для вектора  $\mathbf{x}_t$  из входной последовательности  $\mathcal{X} = \langle \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_\tau \rangle$ . Если  $\mathbf{o}_t = (o_{t1}, o_{t2}, \dots, o_{tp})^T \in \mathbb{R}^p$  – это  $p$ -мерный выходной вектор в момент времени  $t$ , то  $\mathbf{o}_t = f^o(\mathbf{net}_t^o)$ , где  $\mathbf{net}_t^o = \mathbf{W}_o^T \mathbf{h}_t + \mathbf{b}_o = (\text{net}_{t1}^o, \text{net}_{t2}^o, \dots, \text{net}_{tp}^o)^T$  – вектор чистых значений (взвешенная сумма входов) на выходных нейронах  $\mathbf{o}_t$  в момент времени  $t$  (до применения функции активации  $f^o$ ).

Определим вектор  $\delta_t^o$  как чистый вектор градиента ошибки для выходного вектора  $\mathbf{o}_t$ , т. е. вектор производных от функции ошибки  $\mathcal{E}_{\mathbf{x}_t}$  по отношению к чистым значениям на нейронах  $\mathbf{o}_t$ , заданный как

$$\delta_t^o = \left( \frac{\partial \mathcal{E}_{\mathbf{x}_t}}{\partial \text{net}_{t1}^o}, \frac{\partial \mathcal{E}_{\mathbf{x}_t}}{\partial \text{net}_{t2}^o}, \dots, \frac{\partial \mathcal{E}_{\mathbf{x}_t}}{\partial \text{net}_{tp}^o} \right).$$



## Градиент ошибки на скрытых нейронах

Аналогично, пусть  $\delta_t^h$  обозначает чистый вектор градиента ошибки для нейронов скрытого состояния  $\mathbf{h}_t$  в момент времени  $t$

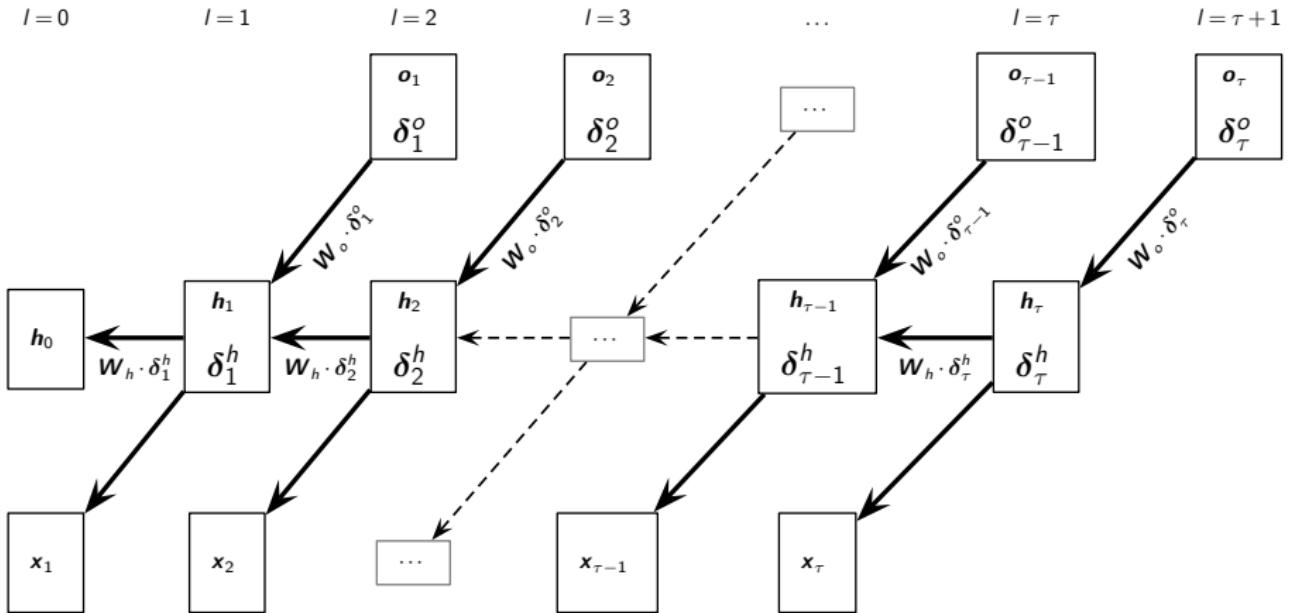
$$\delta_t^h = \left( \frac{\partial \mathcal{E}_{\mathbf{x}_t}}{\partial \text{net}_{t1}^h}, \frac{\partial \mathcal{E}_{\mathbf{x}_t}}{\partial \text{net}_{t2}^h}, \dots, \frac{\partial \mathcal{E}_{\mathbf{x}_t}}{\partial \text{net}_{tp}^h} \right),$$

где  $\mathbf{h}_t = (h_{t1}, h_{t2}, \dots, h_{tm})^T \in \mathbb{R}^m$  –  $m$ -мерный вектор скрытого состояния в момент времени  $t$ , а  $\text{net}_{ti}^h$  – чистое значение на скрытом нейроне  $h_{ti}$  в момент времени  $t$ .

Ключевым шагом в обратном распространении ошибки является вычисление градиентов ошибки в обратном порядке, начиная с выходных нейронов и заканчивая входными нейронами через скрытые нейроны.

Для шага обратного распространения проще рассматривать сеть RNN в виде отдельных слоев с учетом зависимостей вместо развертки по времени.

# Сеть RNN: шаги обратного прохода





# Вычисление чистых градиентов

При обратном распространении ошибки изменяется направление потока вычислений для вычисления чистых градиентов  $\delta_t^o$  и  $\delta_t^h$ . Чистый вектор градиента на выходе  $\mathbf{o}_t$  можно вычислить следующим образом:

$$\delta_t^o = \partial \mathbf{f}^o \odot \partial \mathcal{E}_{\mathbf{x}_t},$$

где  $\odot$  – поэлементное произведение или произведение Адамара,  $\mathbf{f}^o$  – функция активации в выходном слое. С другой стороны, чистые градиенты в каждом из скрытых слоев должны учитывать входящие чистые градиенты от  $\mathbf{o}_t$  и от  $\mathbf{h}_{t+1}$ :

$$\delta_t^h = \partial \mathbf{f}^h \odot ((\mathbf{W}_o \delta_t^o) + (\mathbf{W}_h \delta_{t+1}^h)),$$

где  $\mathbf{f}^h$  – функция активации в выходном слое. Обратите внимание, что  $\mathbf{h}_t$  зависит только от  $\mathbf{o}_t$ , т.е.  $\delta_{t+1}^h = 0$ . Чистые градиенты не нужно вычислять для  $\mathbf{h}_0$  или для любого из входных нейронов  $\mathbf{x}_t$ , поскольку они являются листовыми узлами в графе обратного распространения.



# Пример RNN для грамматики Ребера

Используем сеть RNN для изучения грамматики Артура Ребера (1967), которая порождается соответствующим конечным автоматом.

Пусть  $\Sigma = \{B, E, P, S, T, V, X\}$  обозначает алфавит, состоящий из семи символов, и пусть  $\$$  обозначает терминальный символ.

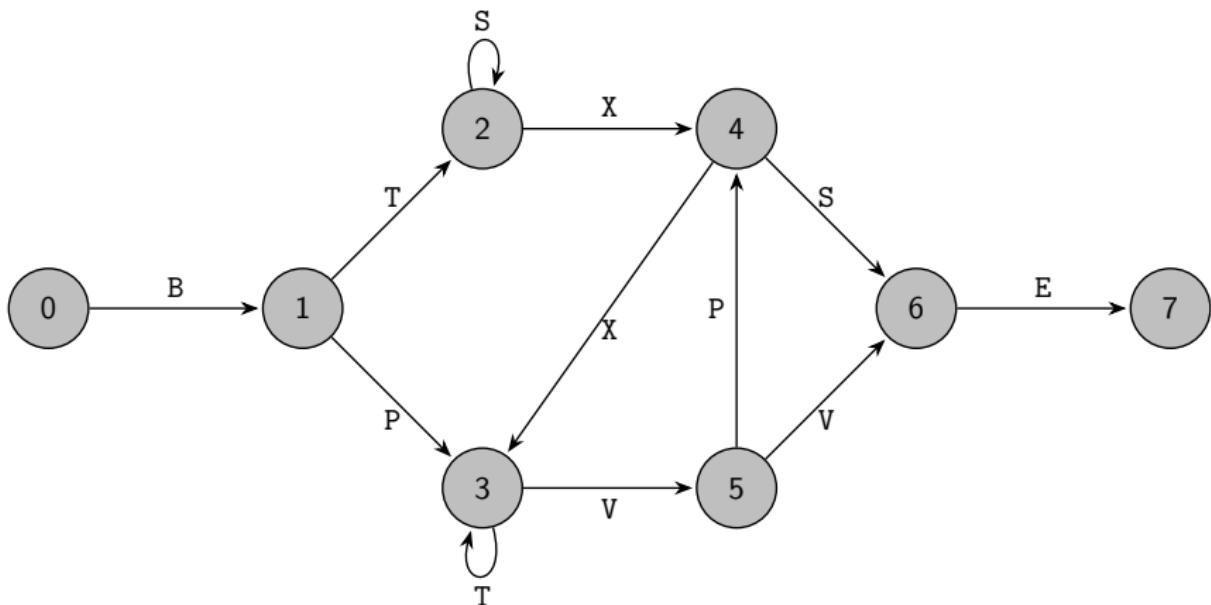
Начиная с начального узла автомата, можно генерировать строки, которые соответствуют грамматике Ребера, включая в строки символы на проходящих ребрах. Если из узла автомата есть два перехода в другие узлы, то каждый из них может быть выбран с равной вероятностью.

Задача сети RNN состоит в том, чтобы научиться предсказывать следующий символ для каждой позиции в заданной последовательности Ребера. Для обучения сети RNN генерируем последовательности Ребера при помощи конечного автомата.

# Конечный автомат грамматики Ребера



Последовательность  $\langle B, T, S, S, X, X, T, V, V, E \rangle$  является допустимой последовательностью Ребера (с соответствующей последовательностью состояний автомата  $\langle 0, 1, 2, 2, 2, 4, 3, 3, 5, 6, 7 \rangle$ ). С другой стороны, последовательность  $\langle B, P, T, X, S, E \rangle$  не является допустимой последовательностью Ребера, так как нет ребер из состояния 3 с символом  $X$ .





# Отклик для последовательности Ребера

Пусть  $S_{\mathcal{X}} = \langle s_1, s_2, \dots, s_7 \rangle$  — последовательность Ребера. Соответствующий истинный отклик (выходные данные)  $\mathcal{Y}$  затем задается как набор следующих символов от каждого из ребер, выходящих из состояния, соответствующего каждой позиции в  $S_{\mathcal{X}}$ .

Например, рассмотрим последовательность Ребера  $S_{\mathcal{X}} = \langle B, P, T, V, V, E \rangle$  с последовательностью состояний  $\pi = \langle 0, 1, 3, 3, 5, 6, 7 \rangle$ . Затем желаемая выходная последовательность задается как  $S_{\mathcal{Y}} = \{P \mid T, T \mid V, T \mid V, P \mid V, E, \$\}$ , где  $\$$  — терминальный символ. Здесь  $P \mid T$  означает, что следующим символом может быть либо  $P$ , либо  $T$ . Видим, что  $S_{\mathcal{Y}}$  содержит последовательность возможных следующих символов из каждого из состояний в  $\pi$  (исключая начальное состояние 0).



# Двоичное кодирование символов алфавита

Чтобы сгенерировать обучающие данные для сети RNN, нужно преобразовать символьные строки Ребера в числовые векторы посредством двоичного кодирования:

$B$	$(1, 0, 0, 0, 0, 0, 0)^T$
$E$	$(0, 1, 0, 0, 0, 0, 0)^T$
$P$	$(0, 0, 1, 0, 0, 0, 0)^T$
$S$	$(0, 0, 0, 1, 0, 0, 0)^T$
$T$	$(0, 0, 0, 0, 1, 0, 0)^T$
$V$	$(0, 0, 0, 0, 0, 1, 0)^T$
$X$	$(0, 0, 0, 0, 0, 0, 1)^T$
$\$$	$(0, 0, 0, 0, 0, 0, 0)^T$

Терминальный символ  $\$$  не является частью алфавита, и его кодировка состоит из нулей.

Наконец, чтобы закодировать возможные следующие символы, следуем аналогичному двоичному кодированию с 1 в столбце, соответствующем разрешенным символам.

Например, выбор  $P \mid T$  кодируется как  $(0, 0, 1, 0, 1, 0, 0)^T$ . Таким образом, последовательность Ребера  $S_{\mathcal{X}}$  и желаемая выходная последовательность  $S_{\mathcal{Y}}$  кодируются как:

	$\mathcal{X}$						$\mathcal{Y}$					
$\Sigma$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$y_1$	$y_2$	$y_3$	$y_4$	$y_5$	$y_6$
	B	P	T	V	V	E	P T	T V	T V	P V	E	\$
B	1	0	0	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	1	0	0	0	0	1	0
P	0	1	0	0	0	0	1	0	0	1	0	0
S	0	0	0	0	0	0	0	0	0	0	0	0
T	0	0	1	0	0	0	1	1	1	0	0	0
V	0	0	0	1	1	0	0	1	1	1	0	0
X	0	0	0	0	0	0	0	0	0	0	0	0



# Обучение сети RNN грамматике Ребера

Для обучения сети генерируем  $n = 400$  последовательностей Ребера с минимальной длиной 30. Максимальная длина последовательности составляет  $\tau = 52$ . Каждая из этих последовательностей Ребера используется для создания обучающей пары  $(\mathcal{X}, \mathcal{Y})$ , как описано выше. Затем обучаем сеть RNN с  $m = 4$  скрытыми нейронами, используя активацию  $\tanh$ . Размеры входного и выходного слоев определяются размерностью кодирования, а именно  $d = 7$  и  $p = 7$ . Используем активацию сигмоидой в выходном слое, рассматривая каждый нейрон как независимый. Используем в качестве функции ошибки бинарную кросс-энтропию.

Сеть RNN обучается для  $r = 10000$  эпох, используя размер шага градиента  $\eta = 1$  и весь набор из 400 входных последовательностей в качестве размера пакета. Модель сети RNN отлично обучается на тренировочных данных, не допуская ошибок в предсказании набора возможных следующих символов.



Далее тестируем модель сети RNN на 100 ранее неизвестных последовательностях Ребера (минимальная длина 30, как и раньше). Сеть RNN не делает ошибок в тестовых последовательностях.

С другой стороны, также обучена сеть MLP с одним скрытым слоем, размер которого варьировался от 4 до 100. Даже после  $r = 10000$  эпох сеть MLP не может правильно предсказать ни одну из выходных последовательностей. Она делает в среднем 2.62 ошибки на последовательность как для обучающих, так и для тестовых данных.

Увеличение количества эпох или количества скрытых слоев не улучшает производительность сети MLP.



# Двунаправленные сети RNN (BRNN)

Сеть RNN использует скрытое состояние  $\mathbf{h}_t$ , которое зависит от предыдущего скрытого состояния  $\mathbf{h}_{t-1}$  и текущих входных данных  $\mathbf{x}_t$  в момент времени  $t$ . Другими словами, сеть RNN смотрит только на информацию из прошлого.

**Двунаправленная сеть RNN** (Bidirectional RNN, BRNN) расширяет модель сети RNN, чтобы также включать информацию из будущего.

Нейронная сеть BRNN поддерживает вектор **обратного скрытого состояния**  $\mathbf{b}_t \in \mathbb{R}^m$ , который зависит от следующего скрытого обратного состояния  $\mathbf{b}_{t+1}$  и текущего входного значения  $\mathbf{x}_t$ . Выходные данные в момент времени  $t$  являются функцией как  $\mathbf{h}_t$ , так и  $\mathbf{b}_t$ . В частности, прямые и обратные скрытые векторы состояния вычисляются следующим образом:

$$\mathbf{h}_t = f^h \left( \mathbf{W}_{ih}^T \mathbf{x}_t + \mathbf{W}_h^T \mathbf{h}_{t-1} + \mathbf{b}_h \right),$$

$$\mathbf{b}_t = f^b \left( \mathbf{W}_{ib}^T \mathbf{x}_t + \mathbf{W}_b^T \mathbf{b}_{t+1} + \mathbf{b}_b \right),$$

где  $\mathbf{W}_{ih}$ ,  $\mathbf{W}_h$ ,  $\mathbf{b}_h$ ,  $\mathbf{W}_{ib}$ ,  $\mathbf{W}_b$ ,  $\mathbf{b}_b$  – веса и смещения нейронной сети



Выходные данные в момент времени  $t$  вычисляются только тогда, когда доступны как  $\mathbf{h}_t$ , так и  $\mathbf{b}_t$ , и задаются как

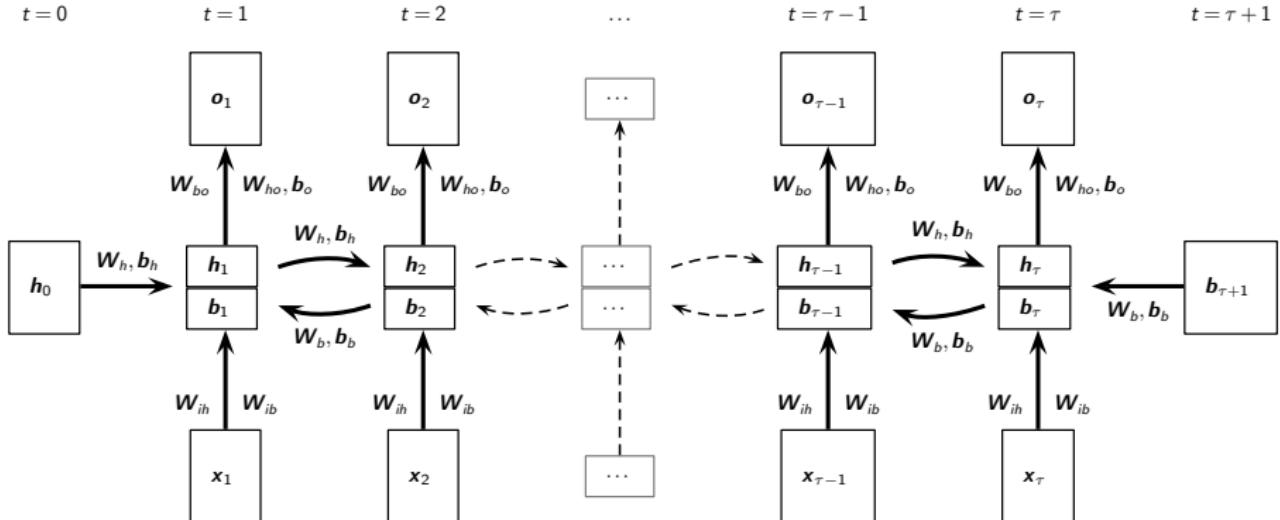
$$\mathbf{o}_t = f^o (\mathbf{W}_{ho}^T \mathbf{h}_t + \mathbf{W}_{bo}^T \mathbf{b}_t + \mathbf{b}_o)$$

Понятно, что сетям BRNN нужен полный набор входных данных, прежде чем они смогут вычислить выходные данные.

Также можно рассматривать сети BRNN как имеющие два набора входных последовательностей, а именно прямую входную последовательность  $\mathcal{X} = \langle \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_\tau \rangle$  и обратную входную последовательность  $\mathcal{X}^r = \langle \mathbf{x}_\tau, \mathbf{x}_{\tau-1}, \dots, \mathbf{x}_1 \rangle$  с соответствующими скрытыми состояниями  $\mathbf{h}_t$  и  $\mathbf{b}_t$ , которые вместе определяют выход  $\mathbf{o}_t$ .

Таким образом, сеть BRNN состоит из двух «сложенных» сетей RNN с независимыми скрытыми слоями, которые совместно определяют выход.

# Сети BRNN: развертка во времени





Одной из проблем при обучении RNN является их чувствительность либо к исчезающему, либо взрывающемуся градиенту. Например, рассмотрим задачу вычисления вектора чистого градиента  $\delta_t^h$  для скрытого слоя в момент времени  $t$ , заданного как

$$\delta_t^h = \partial \mathbf{f}^h \odot ((\mathbf{W}_o \delta_t^o) + (\mathbf{W}_h \delta_{t+1}^h))$$

Предположим для простоты, что используется линейная функция активации, т. е.  $\partial \mathbf{f}^h = 1$ , и проигнорируем вектор чистого градиента для выходного слоя, сосредоточившись только на зависимости от скрытых слоев. Тогда для входной последовательности длины  $\tau$  имеем

$$\delta_t^h = \mathbf{W}_h \delta_{t+1}^h = \mathbf{W}_h (\mathbf{W}_h \delta_{t+2}^h) = \mathbf{W}_h^2 \delta_{t+2}^h = \dots = \mathbf{W}_h^{\tau-t} \delta_\tau^h$$

Можно заметить, что чистый градиент в момент времени  $\tau$  влияет на чистый градиент в момент времени  $t < \tau$  в зависимости от значений  $\mathbf{W}_h^{\tau-t}$  (степени матрицы скрытых весов  $\mathbf{W}_h$ ).



# Спектральный радиус матрицы скрытых весов

Пусть спектральный радиус матрицы  $\mathbf{W}_h$ , определяемый как модуль наибольшего собственного значения матрицы  $\mathbf{W}_h$ , задан как  $|\lambda_1|$ .

Получается, что если  $|\lambda_1| < 1$ , то  $\|\mathbf{W}_h^k\| \rightarrow 0$  при  $k \rightarrow \infty$ , то есть при обучении на длинных последовательностях градиенты исчезают.

С другой стороны, если  $|\lambda_1| > 1$ , то по крайней мере один элемент  $\mathbf{W}_h^k$  становится неограниченным, и, таким образом,  $\|\mathbf{W}_h^k\| \rightarrow \infty$  при  $k \rightarrow \infty$ , то есть градиенты взрываются при обучении на длинных последовательностях.

Ясно, что чистые градиенты масштабируются в соответствии с собственными значениями матрицы  $\mathbf{W}_h$ . Следовательно, если  $|\lambda_1| < 1$ , то  $|\lambda_1|^k \rightarrow 0$  при  $k \rightarrow \infty$ , а поскольку  $|\lambda_1| \geq |\lambda_i|$  для всех  $i = 1, 2, \dots, m$ , то обязательно также  $|\lambda_i|^k \rightarrow 0$ . То есть градиенты исчезают.

С другой стороны, если  $|\lambda_1| > 1$ , то  $|\lambda_1|^k \rightarrow \infty$  при  $k \rightarrow \infty$ , и градиенты взрываются.

Поэтому, чтобы ошибка не исчезала и не взрывалась, спектральный радиус матрицы  $\mathbf{W}_h$  должен оставаться равным 1 или очень близким к нему.



# Проблема исчезающих/взрывающихся градиентов

Проблема исчезающих (затухающих) градиентов (vanishing gradient) состоит в том, что по мере продвижения от выходного слоя к входному слою в алгоритме обратного распространения градиенты становятся все меньше и меньше и приближаются к нулю, что в конечном итоге оставляет веса начальных слоев почти неизменными. В результате градиентный спуск не сходится к оптимальной точке.

Признаки исчезающих градиентов:

- Веса последних слоев изменяются существенно, тогда как веса начальных слоев не сильно изменяются (или не изменяются совсем).
- Веса модели могут стать равным нулю во время обучения.
- Модель обучается очень медленно, иногда обучение останавливается на очень ранней стадии сразу после нескольких итераций.

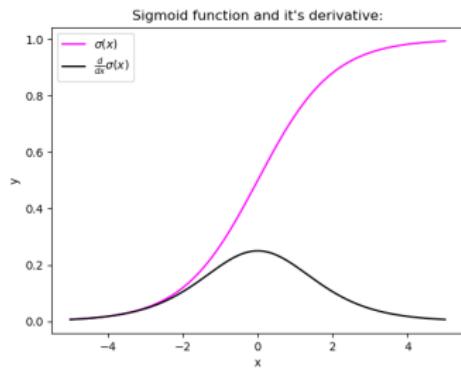
Напротив, проблема взрывающихся градиентов (exploding gradient) состоит в том, что градиенты в алгоритме обратного распространения становятся все больше и больше по мере продвижения к входному слою и это приводит к очень большим обновлениям весов и заставляет градиентный спуск расходиться.

Признаки взрывающихся градиентов:

- Наблюдается экспоненциальный рост параметров модели.
- Веса модели могут стать равными NaN (Not-a-Number) во время обучения.

Некоторые функции активации, такие как сигмоида, имеют очень большую разницу между дисперсией входных и выходных данных, а именно, они сжимают входные данные большего диапазона в меньшей диапазон в интервале  $[0,1]$ .

График сигмоиды показывает, что для больших входных данных (отрицательных или положительных) сигмоида стремится к 0 или 1 с производной, очень близкой к нулю. Таким образом, у алгоритма обратного распространения практически нет градиента для обратного распространения по слоям нейронной сети, и существующие небольшие остаточные градиенты продолжают уменьшаться по мере продвижения к начальным слоям.



Точно так же в некоторых случаях начальным весам нейронной сети могут соответствовать большие значения потерь и большие значения градиента. Далее большие значения градиента приводят к большим обновлениям весов сети и, как следствие, к нестабильности сети. Иногда веса могут стать настолько большими, что они переполняются и приводят к появлению значений NaN.



# Решение проблемы градиентов

## 1. Правильная инициализация весов

Исследователь Xavier Glorot с соавторами установили (2011), что для правильно-го прохождения градиента между слоями:

- Дисперсия выходов каждого слоя должна быть равна дисперсии его входов.
- Градиенты должны иметь одинаковую дисперсию до и после прохождения через слой в обратном направлении.

Хотя невозможно, чтобы оба условия выполнялись для любого слоя в сети кроме случая, когда количество входов в слой ( $\text{fan}_{in}$ ) будет равно количеству нейронов в слое ( $\text{fan}_{out}$ ), был предложен следующий компромисс (инициализация Xavier (по имени автора) или инициализация Glorot (по его фамилии)): случайным образом инициализировать веса для каждого слоя в сети со следующими вероятностными распределениями ( $\text{fan}_{avg} = \frac{1}{2} (\text{fan}_{in} + \text{fan}_{out})$ ):

- нормальное распределение со средним 0 и дисперсией  $\sigma^2 = \frac{1}{\text{fan}_{avg}}$
- равномерное распределение от  $-r$  до  $r$ , где  $r = \sqrt{\frac{3}{\text{fan}_{avg}}} = \sqrt{3}\sigma$

В качестве альтернативы для функции активации ReLU и ее вариантов можно использовать инициализацию He (Kaiming He et al. 2016), в которой  $\sigma^2 = \frac{1}{\text{fan}_{in}}$ .

По умолчанию Keras использует инициализацию Glorot с равномерным распределением.



# Решение проблемы градиентов

## 2. Использование функций активации без насыщения

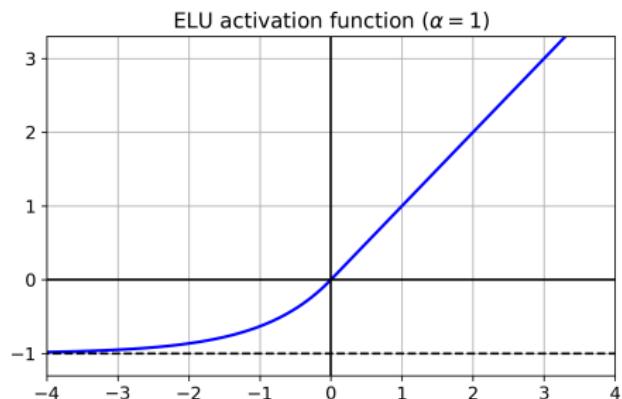
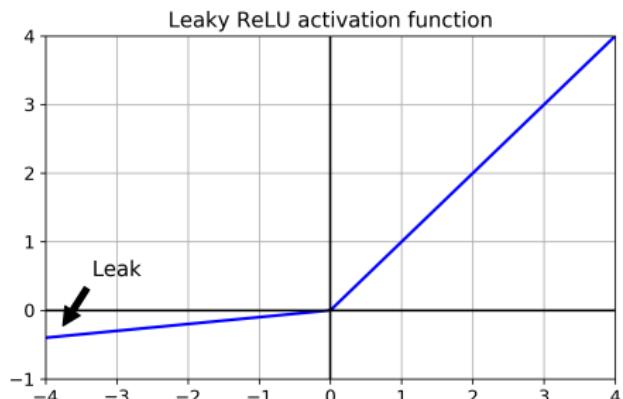
**Насыщение** (saturation) сигмоиды (наличие горизонтальной асимптоты) для больших значений входных данных (отрицательных или положительных) является основной причиной исчезновения градиентов, что делает сигмоиду нерекомендуемой для использования в скрытых слоях сети.

Таким образом, чтобы решить проблему насыщения функций активации, таких как сигмоида и гиперболический тангенс, следует использовать другие функции без насыщения, такие как ReLu и его альтернативы. К сожалению, функция ReLu также не является идеальным выбором для промежуточных слоев сети, так как она страдает от проблемы, известной как умирание нейронов, когда некоторые нейроны продолжают выдавать 0 в качестве выходных данных по мере обучения сети.

Некоторые популярные альтернативы функции ReLU, которые смягчают проблему исчезающих градиентов при использовании в качестве активации для промежуточных слоев сети – это функции активации LReLU, PReLU, ELU, SELU.

# Решение проблемы градиентов

## 2. Использование функций активации без насыщения



$$\text{LReLU}_{\alpha}(z) = \max(\alpha z, z), \alpha > 0$$

В частности,  $\alpha = 0.01$  (small leak),  
 $\alpha = 0.2$  (huge leak)

$$\text{ELU}_{\alpha}(z) = \begin{cases} \alpha (\exp(z) - 1), & z < 0 \\ z, & z \geq 0 \end{cases}$$



# Решение проблемы градиентов

## 3. Пакетная нормализация (Batch Normalization)

Пакетная нормализация слоя состоит в применении следующих формул:

$$\begin{aligned}\boldsymbol{\mu}_B &= \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)} \Rightarrow \boldsymbol{\sigma}_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (\mathbf{x}^{(i)} - \boldsymbol{\mu}_B)^2 \Rightarrow \\ &\Rightarrow \hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^2 + \varepsilon}} \Rightarrow \mathbf{z}^{(i)} = \gamma \odot \hat{\mathbf{x}}^{(i)} + \beta, i = \overline{1, m_B}\end{aligned}$$

где:

- $\boldsymbol{\mu}_B$  – вектор входных средних, оцененный по всему мини-пакету  $B$
- $\boldsymbol{\sigma}_B^2$  – вектор входных дисперсий, также оцениваемый по мини-пакету
- $m_B$  – число точек в мини-пакете  $B$
- $\hat{\mathbf{x}}^{(i)}$  – вектор центрированных и нормализованных данных для точки  $i$
- $\gamma$  – вектор параметров масштабирования выходных данных слоя
- операция  $\odot$  представляет собой поэлементное умножение
- $\beta$  – вектор параметров сдвига (смещения) выходных данных слоя
- $\varepsilon$  – малое число, призванное избежать деления на ноль (обычно  $10^{-5}$ )
- $\mathbf{z}^{(i)}$  – результат пакетной нормализации вектора  $\mathbf{x}^{(i)}$  из мини-пакета



# Решение проблемы градиентов

## 3. Пакетная нормализация (Batch Normalization)

Пакетная нормализация нейронной сети заключается в добавлении операции нормализации в модель непосредственно перед или после функции активации каждого скрытого слоя. Эта операция центрирует и нормализует каждый вход, затем масштабирует и сдвигает результат, используя два новых вектора параметров на слой: один для масштабирования ( $\gamma$ ), другой для сдвига ( $\beta$ ). Другими словами, операция позволяет модели узнать оптимальный масштаб и среднее значение входных данных каждого слоя. При пакетной нормализации не нужно стандартизировать обучающую выборку.

Итак, в ходе обучения нейронной сети пакетная нормализация стандартизует входные данные, потом масштабирует и сдвигает их.

Оказалось, что пакетная нормализация существенно улучшает обучение глубоких нейронных сетей. Снижается острота проблемы исчезающих градиентов, уменьшается зависимость от метода инициализации весов, возникает возможность использовать больший шаг обучения, что ускоряет процесс обучения. наконец, в качестве бонуса пакетная нормализация действует как регуляризатор, снижая необходимость применения других методов регуляризации.



# Решение проблемы градиентов

## 4. Обрезка градиентов (Gradient Clipping)

Другим популярным методом смягчения проблемы взрывающихся градиентов является **обрезка градиентов** во время обратного распространения, чтобы они никогда не превышали некоторый порог. Этот метод чаще всего используется в рекуррентных нейронных сетях, поскольку пакетную нормализацию сложно использовать для этой архитектуры нейронных сетях. Для других архитектур нейронных сетей обычно достаточно пакетной нормализации.

Обрезка градиента может производиться по отдельным значениям (clipvalue) и по норме (clipnorm) градиента. Пороговое значение (threshold) градиента является настраиваемым гиперпараметром. При обрезке градиента по значениям оптимизатор обрезает каждую компоненту градиента, которая выходит за диапазон, задаваемый порогом. При этом направление градиента может измениться. При обрезке по норме все компоненты градиента будут пропорционально уменьшены, если норма  $L_2$  градиента превышает пороговое значение.



# Управляемые (gated) сети RNN

Для решения проблемы исчезающих градиентов в сетях RNN могут быть использованы нейроны-вентили (gate neurons), управляющие доступом к скрытым состояниям.

Рассмотрим  $m$ -мерный вектор скрытого состояния  $\mathbf{h}_t \in \mathbb{R}^m$  в момент времени  $t$ . В обычной RNN этот вектор обновляется следующим образом:

$$\mathbf{h}_t = f^h \left( \mathbf{W}_i^T \mathbf{x}_t + \mathbf{W}_h^T \mathbf{h}_{t-1} + \mathbf{b}_h \right)$$

Пусть  $\mathbf{g} \in \{0, 1\}^m$  — бинарный вектор. Если мы возьмем поэлементное произведение  $\mathbf{g}$  и  $\mathbf{h}_t$ , а именно,  $\mathbf{g} \odot \mathbf{h}_t$ , то элементы  $\mathbf{g}$  действуют как вентили, которые позволяют либо сохранить соответствующий элемент  $\mathbf{h}_t$ , либо обнулить его, то есть выбрать элементы  $\mathbf{h}_t$  для запоминания или забывания.

Однако для обратного распространения нам нужны дифференцируемые вентили, для которых мы используем сигмоидную активацию нейронов-вентилей, чтобы их значение лежало в диапазоне  $[0, 1]$ . Подобно логическим вентилям, такие нейроны позволяют полностью запомнить входные данные, если значение равно 1, или забыть, если значение равно 0. Кроме того, они обеспечивают взвешенную память, позволяющую частично запомнить элементы  $\mathbf{h}_t$  для значений от 0 до 1.



# Пример дифференцируемого вентиля

Рассмотрим скрытый вектор состояния

$$\mathbf{h}_t = (-0.94, 1.05, 0.39, 0.97, 0.90)^T$$

и вектор логических вентилей  $\mathbf{g}$ :

$$\mathbf{g} = (0, 1, 1, 0, 1)^T$$

Их поэлементное произведение дает

$$\mathbf{g} \odot \mathbf{h}_t = (0, 1.05, 0.39, 0, 0.90)^T$$

Первый и четвертый элементы были «забыты». Теперь рассмотрим дифференцируемый вектор вентилей с элементами от 0 до 1:

$$\mathbf{g} = (0.1, 0, 1., 0.9, 0.5)^T$$

Поэлементное произведение  $\mathbf{g}$  и  $\mathbf{h}_t$  дает

$$\mathbf{g} \odot \mathbf{h}_t = (-0.094, 0, 0.39, 0.873, 0.45)^T$$

Тогда только часть значений скрытого вектора  $\mathbf{h}_t$  сохраняется в памяти.



Чтобы увидеть, как работают управляемые (gated) нейроны, рассмотрим RNN с [вентилем забывания](#).

В обычной сети RNN с активацией  $\tanh$  (гиперболический тангенс), вектор скрытых состояний обновляется безусловно следующим образом:

$$\mathbf{h}_t = \tanh(\mathbf{W}_i^T \mathbf{x}_t + \mathbf{W}_h^T \mathbf{h}_{t-1} + \mathbf{b}_h)$$

Вместо прямого обновления  $\mathbf{h}_t$  будем использовать нейроны с вентилями забывания, чтобы контролировать, какую часть предыдущего вектора скрытых состояний следует забыть при вычислении его нового значения, а также контролировать, как его обновлять с учетом новых входных данных  $\mathbf{x}_t$ .

Пусть даны входные данные  $\mathbf{x}_t$  и предыдущие скрытые состояния  $\mathbf{h}_{t-1}$ , тогда сначала вычисляем [вектор обновления](#)  $\mathbf{u}_t \in \mathbb{R}^m$  следующим образом:

$$\mathbf{u}_t = \tanh(\mathbf{W}_u^T \mathbf{x}_t + \mathbf{W}_{hu}^T \mathbf{h}_{t-1} + \mathbf{b}_u)$$

Вектор  $\mathbf{u}_t$  по сути является немодифицированным вектором скрытого состояния, как в обычной сети RNN.



# Вентиль забывания в управляемой сети RNN

Используя векторный вентиль забывания  $\phi_t \in \mathbb{R}^m$ , можно вычислить новый вектор скрытого состояния  $\mathbf{h}_t$  следующим образом:

$$\mathbf{h}_t = \phi_t \odot \mathbf{h}_{t-1} + (1 - \phi_t) \odot \mathbf{u}_t$$

Очевидно, что новый вектор скрытого состояния  $\mathbf{h}_t$  сохраняет часть предыдущего значения вектора скрытого состояния  $\mathbf{h}_{t-1}$  и (дополнительную) часть значений вектора обновления  $\mathbf{u}_t$ . Заметим, что если  $\phi_t = 0$ , т. е. если мы хотим полностью забыть предыдущие скрытые состояния, то  $1 - \phi_t = 1$ , что означает, что скрытые состояния будут полностью обновляться на каждом временном шаге, как и в обычной сети RNN. Тогда выходной вектор  $\mathbf{o}_t$  вычисляется как

$$\mathbf{o}_t = f^o (\mathbf{W}_o^T \mathbf{h}_t + \mathbf{b}_o)$$

Вектор вентиля забывания  $\phi_t$  зависит от вектора предыдущего скрытого состояния  $\mathbf{h}_{t-1}$  и новых входных данных  $\mathbf{x}_t$ , поэтому

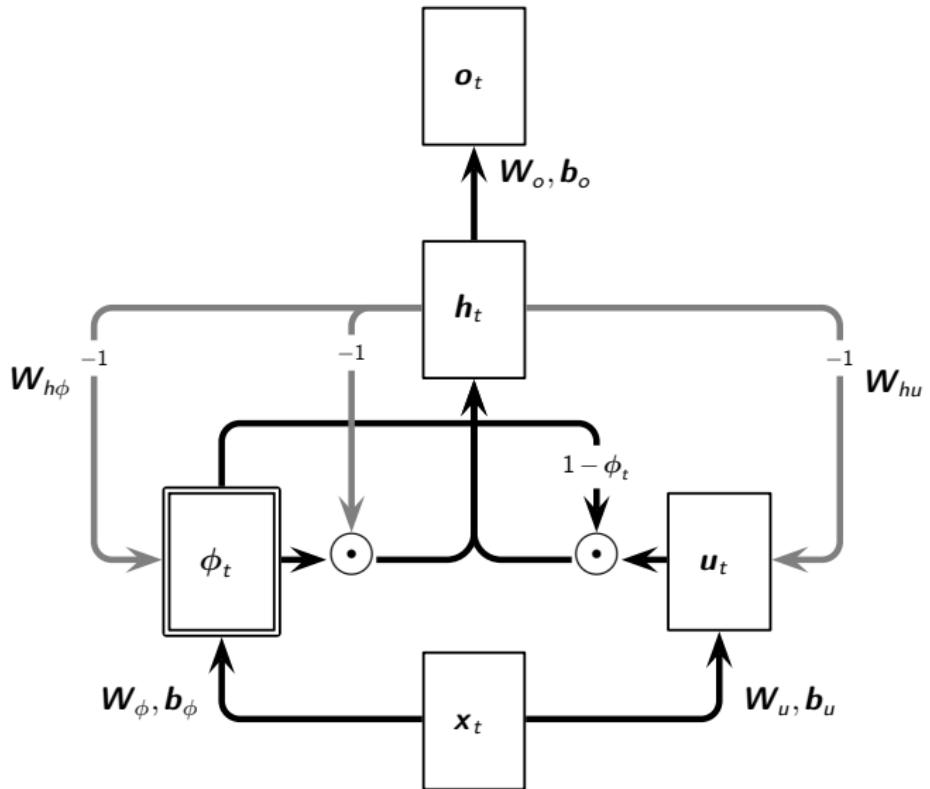
$$\phi_t = \sigma (\mathbf{W}_\phi^T \mathbf{x}_t + \mathbf{W}_{h\phi}^T \mathbf{h}_{t-1} + \mathbf{b}_\phi),$$

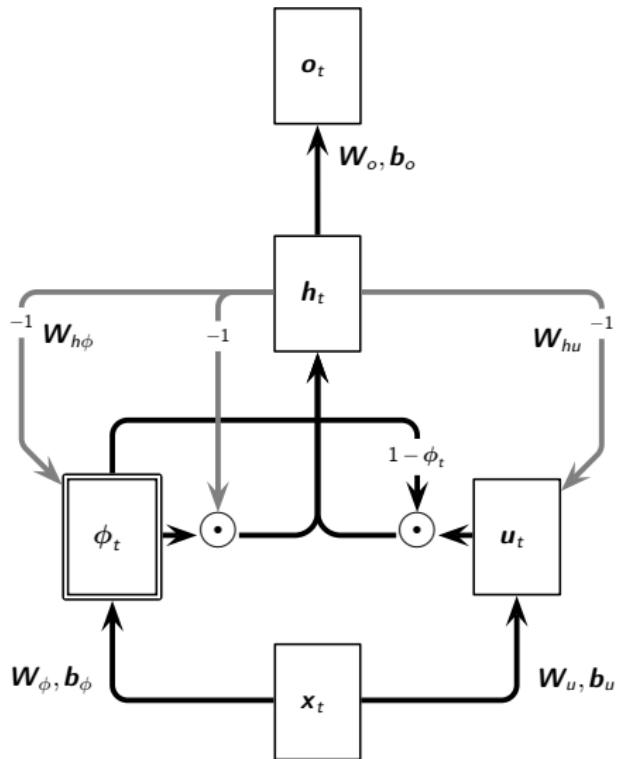
где используется функция активации сигмоида, чтобы гарантировать, что значения находятся в диапазоне  $[0, 1]$ .

# Сеть RNN с вентилем забывания

Повторяющиеся соединения нейронов показаны серым цветом, вентиль забывания  $\phi_t$  показан двойной линией.

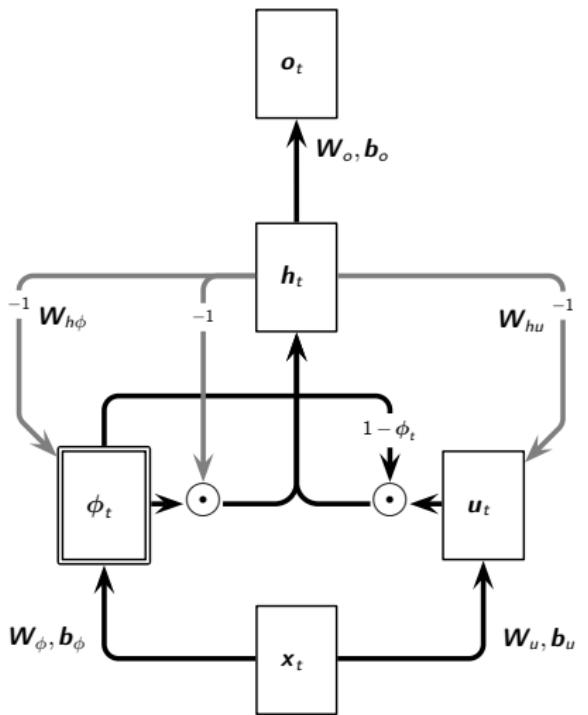
Символ  $\odot$  обозначает поэлементное произведение.





Итак, вектор вентиляй забывания  $\phi_t$  – это слой, зависящий от  $h_{t-1}$  и  $x_t$ . Соединения нейронов забывания с входными нейронами и нейронами скрытых состояний являются полносвязными и задаются соответствующими весовыми матрицами  $W_\phi, W_{h\phi}$  и вектором смещения  $b_\phi$ . Выход слоя забывания  $\phi_t$  нужен, чтобы модифицировать предыдущий слой скрытого состояния  $h_{t-1}$ , и, следовательно, как  $\phi_t$ , так и  $h_{t-1}$  поступают в новый слой поэлементного произведения, обозначенный на рисунке символом  $\odot$ .

## Сеть RNN со слоем вентиляй забывания



Наконец, выходные данные этого слоя поэлементного произведения используются в качестве входных данных для нового скрытого слоя  $h_t$ , который также получает входные данные от другого поэлементного вентиля, который вычисляет выходные данные  $u_t$  и  $1 - \phi_t$ . Таким образом, в отличие от обычных слоев, которые полностью связаны и имеют матрицу весов и вектор смещения между слоями, связи между  $\phi_t$  и  $h_t$  через поэлементный слой взаимно-однозначны, а веса фиксированы на значении 1 со смещением 0. Точно так же связи между  $u_t$  и  $h_t$  через другой поэлементный слой также являются взаимно однозначными, с фиксированным весом 1 и смещением 0.



## Пример обновления вектора скрытых состояний

Пусть  $t = 5$ . Предположим, что предыдущий вектор скрытых состояний и вектор обновления заданы следующим образом:

$$\mathbf{h}_{t-1} = (-0.94, 1.05, 0.39, 0.97, 0.90)^T, \mathbf{u}_t = (0.5, 2.5, -1.0, -0.5, 0.8)^T$$

Пусть вентиль забывания и его дополнение заданы следующим образом:

$$\phi_t = (0.9, 1, 0, 0.1, 0.5)^T, 1 - \phi_t = (0.1, 0, 1, 0.9, 0.5)^T$$

Затем новый вектор скрытых состояний вычисляется как взвешенная сумма предыдущего вектора скрытых состояний и вектора обновления:

$$\begin{aligned}\mathbf{h}_t &= \phi_t \odot \mathbf{h}_{t-1} + (1 - \phi_t) \odot \mathbf{u}_t = \\ &= (0.9, 1, 0, 0.1, 0.5)^T \odot (-0.94, 1.05, 0.39, 0.97, 0.90)^T + \\ &\quad + (0.1, 0, 1, 0.9, 0.5)^T \odot (0.5, 2.5, -1.0, -0.5, 0.8)^T = \\ &= (-0.846, 1.05, 0, 0.097, 0.45)^T + (0.05, 0, -1.0, -0.45, 0.4)^T = \\ &= (-0.796, 1.05, -1.0, -0.353, 0.85)^T\end{aligned}$$



# Градиенты в сети RNN с вентилями забывания

Чистые градиенты на выходах вычисляются с учетом частных производных функции активации ( $\partial \mathbf{f}^o$ ) и функции ошибки ( $\partial \mathcal{E}_{\mathbf{x}_t}$ ):

$$\delta_t^o = \partial \mathbf{f}^o \odot \partial \mathcal{E}_{\mathbf{x}_t}$$

Для других слоев мы можем перевернуть все стрелки, чтобы определить зависимости между слоями. Чистый градиент  $\delta_{ti}^u$  в нейроне слоя обновления  $i$  в момент  $t$  равен:

$$\delta_{ti}^u = \frac{\partial \mathcal{E}_{\mathbf{x}_t}}{\partial \text{net}_{ti}^h} \frac{\partial \text{net}_{ti}^h}{\partial u_{ti}} \frac{\partial u_{ti}}{\partial \text{net}_{ti}^u} = \delta_{ti}^u (1 - \phi_{ti}) (1 - u_{ti}^2),$$

где  $\frac{\partial \text{net}_{ti}^h}{\partial u_{ti}} = \frac{\partial}{\partial u_{ti}} (\phi_{ti} h_{t-1,i} + (1 - \phi_{ti}) u_{ti}) = (1 - \phi_{ti})$  и слой обновления использует функцию активации  $\tanh$ .

По всем нейронам мы получаем суммарный градиент в точке  $\mathbf{u}_t$  следующим образом:

$$\delta_t^u = \delta_t^h \odot (1 - \phi_t) \odot (1 - \mathbf{u}_t \odot \mathbf{u}_t)$$



# Градиенты в сети RNN с вентилями забывания

Чистый градиент  $\delta_{ti}^\phi$  в нейроне вентиля забывания  $i$  в момент времени  $t$  задается как

$$\delta_{ti}^\phi = \frac{\partial \mathcal{E}_{\mathbf{x}_t}}{\partial \text{net}_{ti}^\phi} = \frac{\partial \mathcal{E}_{\mathbf{x}_t}}{\partial \text{net}_{ti}^h} \frac{\partial \text{net}_{ti}^h}{\partial \phi_{ti}} \frac{\partial \phi_{ti}}{\partial \text{net}_{ti}^\phi} = \delta_{ti}^h (h_{t-1,i} - u_{ti}) \phi_{ti} (1 - \phi_{ti})$$

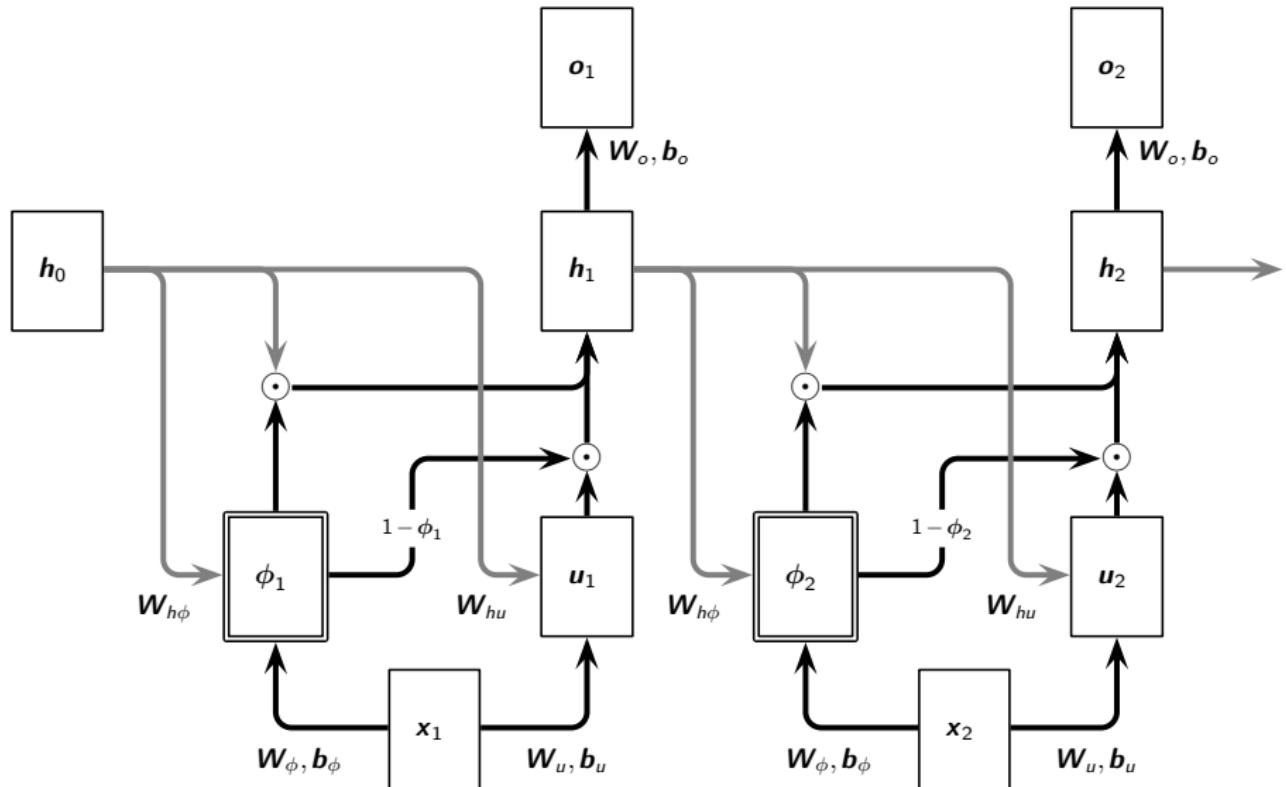
По всем нейронам мы получаем суммарный градиент при  $\phi_t$  следующим образом:

$$\delta_t^\phi = \delta_t^h \odot (\mathbf{h}_{t-1} - \mathbf{u}_t) \odot \phi_t \odot (1 - \phi_t)$$

Учитывая все слои, включая выходные, забывающие, обновляемые и поэлементные слои, полный вектор чистого градиента на скрытом слое в момент времени  $t$  задается как:

$$\delta_t^h = \mathbf{W}_o \delta_t^o + \mathbf{W}_{h\phi} \delta_{t+1}^\phi + \mathbf{W}_{hu} \delta_{t+1}^u + (\delta_{t+1}^h \odot \phi_{t+1})$$

# Развернутая сеть RNN с вентилями забывания





# Сети долгой краткосрочной памяти LSTM

Сети долгой краткосрочной памяти LSTM (Long Short-Term Memory) используют векторы дифференцируемых вентиляй для управления вектором скрытого состояния  $\mathbf{h}_t$ , а также другим вектором  $\mathbf{c}_t \in \mathbb{R}^m$ , называемым вектором внутренней памяти. В частности, сети LSTM используют три вектора вентиляй: вектор входных вентиляй  $\kappa_t \in \mathbb{R}^m$ , вектор вентиляй забывания  $\phi_t \in \mathbb{R}^m$  и вектор выходных вентиляй  $\omega_t \in \mathbb{R}^m$ .

Как и обычная сеть RNN, сеть LSTM также поддерживает вектор скрытого состояния для каждого временного шага. Однако содержимое скрытого вектора выборочно копируется из вектора внутренней памяти через выходной вентиль, при этом внутренняя память обновляется через входной вентиль, а ее части забываются через вентиль забывания.

Каждый из векторов вентиляй концептуально играет разную роль в сети LSTM. Вектор входных вентиляй  $\kappa_t$  определяет, какая часть входного вектора через вектор обновления  $u_t$  может влиять на вектор памяти  $\mathbf{c}_t$ .

Вектор вентиляй забывания  $\phi_t$  определяет, какую часть предыдущего вектора памяти следует забыть, и, наконец, вектор выходных вентиляй  $\omega_t$  определяет, какая часть состояния памяти сохраняется для скрытого состояния.

# Схема сети LSTM

На каждом временном шаге  $t$  три вектора вентиляй обновляются следующим образом:

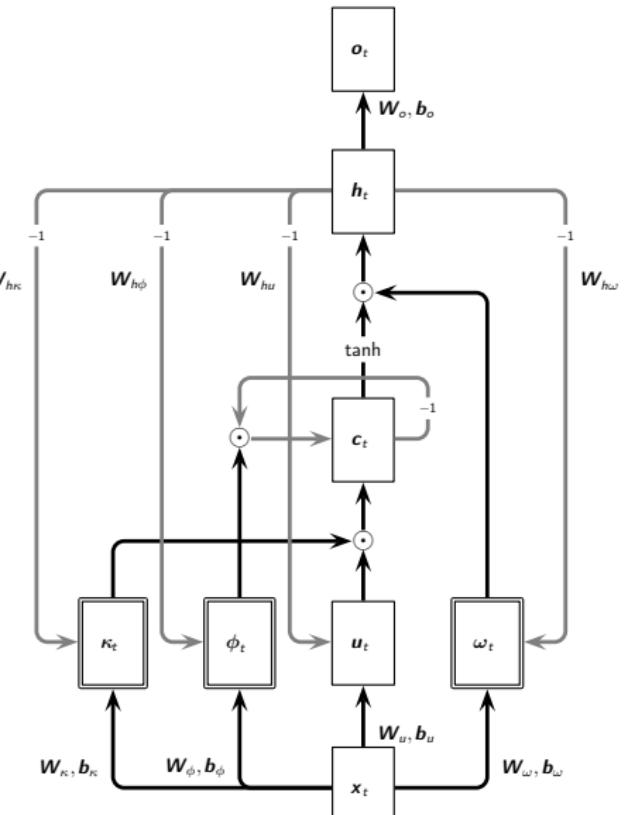
$$\kappa_t = \sigma (\mathbf{W}_\kappa^T \mathbf{x}_t + \mathbf{W}_{h\kappa}^T \mathbf{h}_{t-1} + \mathbf{b}_\kappa)$$

$$\phi_t = \sigma (\mathbf{W}_\phi^T \mathbf{x}_t + \mathbf{W}_{h\phi}^T \mathbf{h}_{t-1} + \mathbf{b}_\phi)$$

$$\omega_t = \sigma (\mathbf{W}_\omega^T \mathbf{x}_t + \mathbf{W}_{h\omega}^T \mathbf{h}_{t-1} + \mathbf{b}_\omega)$$

Повторяющиеся соединения показаны серым цветом, вентили показаны двойной линией.

Символ  $\odot$  обозначает поэлементное произведение.





# Скрытые нейроны сети LSTM

Когда даны векторы  $\mathbf{x}_t$  и  $\mathbf{h}_{t-1}$ , сеть LSTM вычисляет  $\mathbf{u}_t$  после применения функции активации  $\tanh$ :

$$\mathbf{u}_t = \tanh(\mathbf{W}_u^T \mathbf{x}_t + \mathbf{W}_{hu}^T \mathbf{h}_{t-1} + \mathbf{b}_u)$$

Затем LSTM вычисляет внутреннюю память и скрытые векторы состояния:

$$\mathbf{c}_t = \kappa_t \odot \mathbf{u}_t + \phi_t \odot \mathbf{c}_{t-1}$$

$$\mathbf{h}_t = \omega_t \odot \tanh(\mathbf{c}_t)$$

Вектор  $\mathbf{c}_t$  зависит от  $\mathbf{u}_t$  и  $\mathbf{c}_{t-1}$ . Однако вектор  $\kappa_t$  контролирует степень, в которой  $\mathbf{u}_t$  влияет на  $\mathbf{c}_t$ , а вентиль забывания  $\phi_t$  контролирует, какая часть предыдущей памяти будет забыта.

Вектор  $\mathbf{h}_t$  зависит от вектора внутренней памяти  $\mathbf{c}_t$ , активированного при помощи  $\tanh$ , но  $\omega_t$  контролирует, какая часть внутренней памяти отражается в скрытом состоянии.



# Выходные нейроны сети LSTM

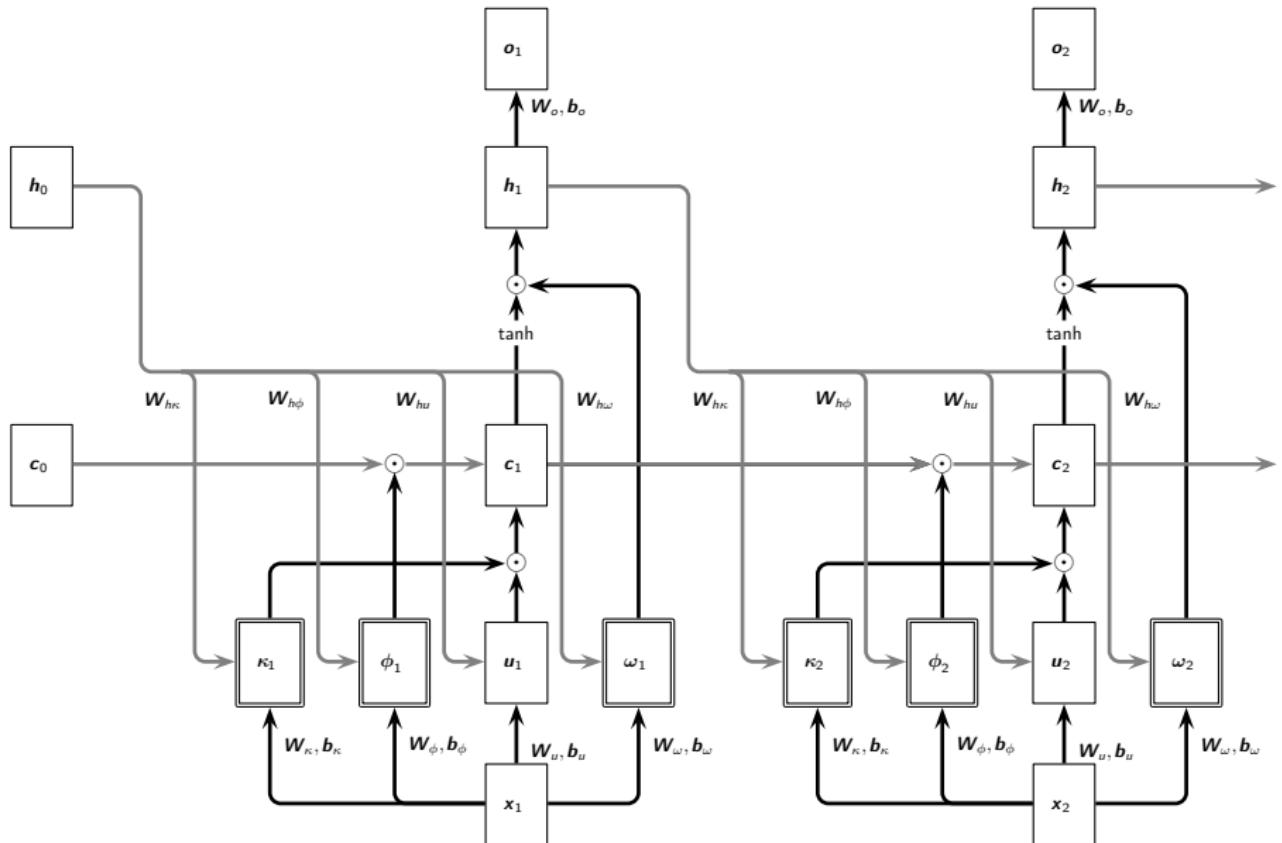
Наконец, выход нейронной сети  $\mathbf{o}_t$  получается путем применения функции активации выходных нейронов  $f^o$  к линейной комбинации значений нейронов скрытого состояния:

$$\mathbf{o}_t = f^o \left( \mathbf{W}_o^T \mathbf{h}_t + \mathbf{b}_o \right)$$

Сети LSTM обычно могут обрабатывать длинные последовательности, поскольку чистые градиенты для состояний внутренней памяти не исчезают в течение длительных временных шагов. Это связано с тем, что по построению состояние памяти  $\mathbf{c}_{t-1}$  в момент времени  $t - 1$  связано с состоянием памяти  $\mathbf{c}_t$  в момент времени  $t$  через неявные веса, фиксированные на 1, и смещения, фиксированные на 0, с линейной активацией.

Это позволяет ошибке перемещаться по времененным шагам, не исчезая и не взрываясь.

# Развернутая сеть LSTM





# Обучение сетей LSTM

Во время обратного распространения вектор чистого градиента на выходном слое в момент времени  $t$  вычисляется путем рассмотрения частных производных функции активации  $\partial \mathbf{f}^o$  и функции ошибки  $\partial \mathcal{E}_{\mathbf{x}_t}$ , следующим образом:

$$\delta_t^o = \partial \mathbf{f}^o \odot \partial \mathcal{E}_{\mathbf{x}_t},$$

где мы предполагаем, что выходные нейроны независимы. Таким образом, вектор чистого градиента  $\delta_t^c$  в точке  $\mathbf{c}_t$  определяется как:

$$\delta_t^c = \delta_t^h \odot \omega_t \odot (1 - \mathbf{c}_t \odot \mathbf{c}_t) + \delta_{t+1}^c \odot \phi_t$$

Входной вентиль также имеет только одно входящее ребро в обратном распространении от  $\mathbf{c}_t$  через поэлементное умножение  $\kappa_t \odot \mathbf{u}_t$  с активацией сигмоидой. Аналогичным образом, как описано выше для  $\delta_t^o$ , чистый градиент  $\delta_t^\kappa$  на входном вентиле  $\kappa_t$  равен:

$$\delta_t^\kappa = \delta_t^c \odot \mathbf{u}_t \odot (1 - \kappa_t) \odot \kappa_t$$



# Обучение сетей LSTM

Те же рассуждения применимы к вектору обновления  $\mathbf{u}_t$ , который также имеет входящее ребро от  $\mathbf{c}_t$  через  $\boldsymbol{\kappa}_t \odot \mathbf{u}_t$  с активацией  $\tanh$ , поэтому вектор чистого градиента  $\delta_t^u$  на слое обновления равен

$$\delta_t^u = \delta_t^c \odot \boldsymbol{\kappa}_t \odot (1 - \mathbf{u}_t \odot \mathbf{u}_t)$$

Аналогично, при обратном распространении есть одно входящее соединение с выходным вентилем от  $\mathbf{h}_t$  через  $\boldsymbol{\omega}_t \odot \tanh(\mathbf{c}_t)$  с активацией сигмоидой, поэтому

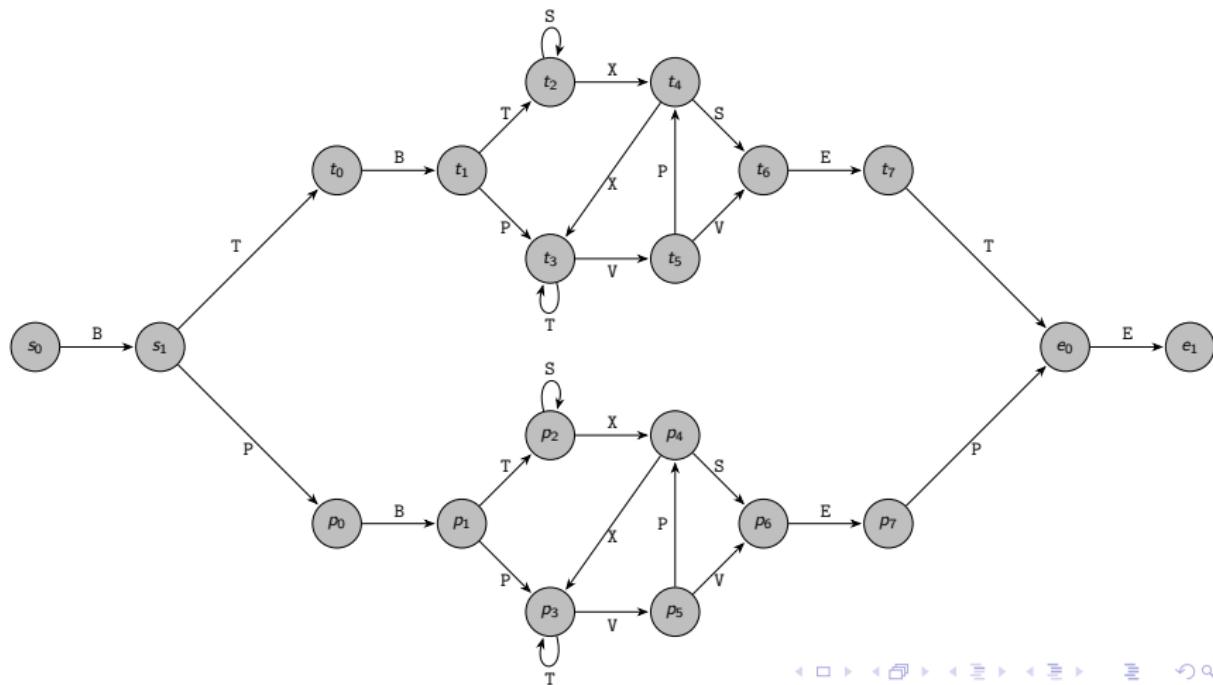
$$\delta_t^\omega = \delta_t^h \odot \tanh(\mathbf{c}_t) \odot (1 - \boldsymbol{\omega}_t) \odot \boldsymbol{\omega}_t$$

Наконец, чтобы вычислить чистые градиенты на скрытом слое, обратите внимание, что градиенты возвращаются к  $\mathbf{h}_t$  из следующих слоев:  $\mathbf{u}_{t+1}$ ,  $\boldsymbol{\kappa}_{t+1}$ ,  $\boldsymbol{\varphi}_{t+1}$ ,  $\boldsymbol{\omega}_{t+1}$  и  $\mathbf{o}_t$ . Следовательно, вектор чистого градиента на векторе скрытого состояния  $\delta_t^h$  задается как

$$\delta_t^h = \mathbf{W}_o \delta_t^o + \mathbf{W}_{h\kappa} \delta_{t+1}^\kappa + \mathbf{W}_{h\varphi} \delta_{t+1}^\varphi + \mathbf{W}_{h\omega} \delta_{t+1}^\omega + \mathbf{W}_{hu} \delta_{t+1}^u$$

# Сеть LSTM для грамматики Ребера

Мы используем сеть LSTM для обучения встроенной грамматике Ребера, которая генерируется соответствующим автоматом. Этот автомат содержит две копии автоматов Ребера.





# Последовательности в грамматике Ребера

Из состояния  $s_1$  верхний автомат достигается путем следования по ребру с меткой  $T$ , тогда как нижний автомат достигается через ребро с меткой  $P$ . Состояния верхнего автомата помечены как  $t_0, t_1, \dots, t_7$ , состояния нижних автоматов помечаются как  $p_0, p_1, \dots, p_7$ . Наконец, обратите внимание, что в состояние  $e_0$  можно попасть либо из верхнего, либо из нижнего автомата, следуя по ребрам, помеченным  $T$  и  $P$  соответственно.

Первым символом всегда является  $B$ , а последним символом всегда является  $E$ . Однако важным моментом является то, что второй символ всегда совпадает с предпоследним символом, и поэтому любая модель обучения последовательностям должна изучать эту долгосрочную зависимость. Например, следующая допустимая вложенная последовательность Ребера:

$$S_{\mathcal{X}} = \langle B, T, B, T, S, S, X, X, T, V, V, E, T, E \rangle$$

Задача сети LSTM — научиться предсказывать следующий символ для каждой из позиций в заданной встроенной последовательности Ребера.



Для обучения мы генерируем  $n = 400$  встроенных последовательностей Ребера с минимальной длиной 40 и преобразуем их в обучающие пары  $(\mathcal{X}, \mathcal{Y})$  с использованием двоичного кодирования. Максимальная длина последовательности составляет  $\tau = 64$ .

С учетом длины последовательностей, используется сеть LSTM с  $m = 20$  скрытых нейронов (меньшие значения  $m$  либо требуют больше эпох для обучения, либо имеют проблемы с обучением сети). Размеры входного и выходного слоев определяются размерностью кодирования, а именно  $d = 7$  и  $p = 7$ . Мы используем активацию сигмоидой на выходном слое, рассматривая каждый нейрон как независимый.

Наконец, мы используем в качестве функции ошибки бинарную кросс-энтропию. Сеть LSTM обучена для  $r = 10000$  эпох (при размере шага  $\eta = 1$  и размере батча 400). Сеть отлично обучается на имеющихся данных, не допуская ошибок в предсказании набора возможных следующих символов.



Мы тестируем модель сети LSTM на 100 ранее недоступных встроенных последовательностях Ребера (минимальная длина 40, как и раньше). Обученная сеть LSTM не делает ошибок в тестовых последовательностях. В частности, она изучила зависимость между вторым символом и предпоследним символом, которые всегда должны совпадать.

Встроенная грамматика Ребера была выбрана, поскольку у сетей RNN возникают проблемы с изучением зависимостей с большим диапазоном. Используя сеть RNN с  $m = 60$  скрытых нейронов и  $r = 25000$  эпох с размером шага  $\eta = 1$ , сеть RNN может идеально обучиться на тренировочных последовательностях. То есть она не делает ошибок ни в одной из 400 тренировочных последовательностей.

Однако на тестовых данных эта сеть RNN ошибается в 40 из 100 тестовых последовательностей. На самом деле в каждой из этих тестовых последовательностей сеть допускает ровно одну ошибку – она не может правильно предсказать предпоследний символ. Эти результаты показывают, что, хотя сеть RNN способна «запоминать» долгосрочную зависимость в обучающих данных, она не может полностью обобщать невидимые тестовые последовательности.



# Применение сетей LSTM

Эффективность сетей LSTM была продемонстрирована при выполнении многих задач, связанных с обработкой последовательностей, таких как генерация текста, генерация рукописного текста, перевод последовательности в последовательность, оценка компьютерных программ, создание подписей к изображениям, создание исходного кода и т.п.

В обучении с подкреплением LSTM являются наиболее эффективными моделями для последовательностей, например, модель AlphaStar для StarCraft II, модель OpenAI Five для Dota 2.

Сети LSTM превосходно справляются с изучением абстракций, то есть умело извлекают семантическую информацию и сохраняют ее в своих ячейках памяти.

Сети LSTM доминировали в задачах генерации текста до появления архитектуры трансформеров в 2017 году, в частности, они использовались в первых больших языковых моделях (LLM).



# Ограничения сетей LSTM

Несмотря на свои успехи, LSTM имеют три основных ограничения:

- ❶ Сложность пересмотра решения о сохранении во внутренней памяти. Например, в задаче поиска ближайшего соседа для заданного вектора последовательность должна последовательно сканироваться в поисках наиболее похожего вектора, чтобы получить прикрепленное к нему значение в конце обработки последовательности, при этом сеть LSTM неохотно обновляет сохраненное ранее значение, когда найден более похожий вектор.
- ❷ Ограниченные возможности хранения, т. е. информация должна быть сжата в состояния скалярных ячеек. Поэтому LSTM хуже работает с редкими токенами из-за ограниченной емкости внутренней памяти.
- ❸ Отсутствие распараллеливаемости из-за смешивания памяти, т. е. скрытых связей между скрытыми состояниями от одного временного шага к другому, которые обеспечивают последовательную обработку.

Эти ограничения LSTM проложили путь к появлению трансформеров в языковом моделировании.



# Архитектура xLSTM

Создатель оригинальной архитектуры LSTM Зепп Хохрайтер (с соавторами) 7 мая 2024 г. представил первое за 27 лет обновление архитектуры под названием xLSTM. Новая версия LSTM может конкурировать с трансформерами как по производительности, так и по масштабируемости.

Сеть xLSTM состоит из двух ключевых подсетей – mLSTM и sLSTM. В mLSTM память представлена не скалярной величиной, а матрицей, что расширяет возможности хранения информации и параллелизации обучения. В sLSTM добавлен новый метод смешивания памяти, благодаря которому дополнительно улучшается производительность модели. Сигмоидная функция активации заменена на экспоненциальную, благодаря чему алгоритм может более гибко управлять своей памятью.

В задаче языкового моделирования сеть xLSTM, обученная на 15 млрд токенов, продемонстрировала превосходство над трансформерами, архитектурами на основе самовнимания и рекуррентными нейросетями. В тестах её производительность оказалась сопоставима с производительностью сети GPT-3, имеющей 350 миллионов параметров.