

ОБЩИЕ СЮЖЕТЫ DL И RNN

Сергей Николенко

СПбГУ — Санкт-Петербург

29 апреля 2025 г.

Random facts:



- В Японии 29 апреля — День Сёва (день рождения императора Хирохито); он открывает «Золотую неделю» праздников: День Конституции (3 мая), День зелени (День основания государства, 4 мая), Праздник детей (5 мая); жизнь в Японии в эту неделю замирает
- 29 апреля 1627 г. кардинал Ришельё создал Компанию Ста акционеров для колонизации Канады, а 29 апреля 1770 г. Джеймс Кук впервые высадился в Австралии
- 29 апреля 1925 г. Николай Бухарин впервые употребил в докладе на XIV партконференции фразу «генеральная линия партии», а 29 апреля 1933 г. при ЦК ВКП(б) была создана Центральная комиссия по чистке партии
- 29 апреля 1945 г. Адольф Гитлер женился на Еве Браун, а уже на следующий день покончил жизнь самоубийством
- 29 апреля 1961 г. Леонид Рогозов, хирург 6-й Советской антарктической экспедиции, успешно вырезал сам себе аппендикс
- 29 апреля 1967 г. Мухаммед Али был лишён боксёрского титула, потому что накануне отказался от призыва в армию

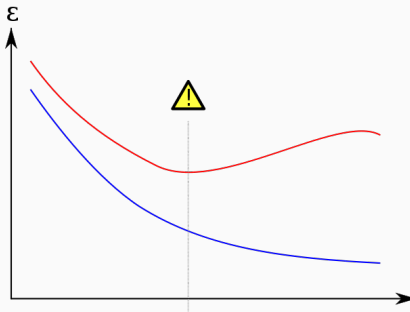
РЕГУЛЯРИЗАЦИЯ В НЕЙРОННЫХ СЕТЯХ

- У нейронных сетей *очень* много параметров.
- Регуляризация совершенно необходима.
- L_2 или L_1 регуляризация ($\lambda \sum_w w^2$ или $\lambda \sum_w |w|$) — это классический метод и в нейронных сетях, *weight decay*.
- Очень легко добавить: ещё одно слагаемое в целевую функцию, и иногда всё ещё полезно.

- Регуляризация есть во всех библиотеках. Например, в *Keras*:
 - `W_regularizer` добавит регуляризатор на матрицу весов слоя;
 - `b_regularizer` — на вектор свободных членов;
 - `activity_regularizer` — на вектор выходов.

РЕГУЛЯРИЗАЦИЯ В НЕЙРОННЫХ СЕТЯХ

- Второй способ регуляризации: *ранняя остановка* (early stopping).
- Давайте просто останавливаться, когда начнёт ухудшаться ошибка на валидационном множестве!
- Тоже есть из коробки в *Keras*, через callbacks.

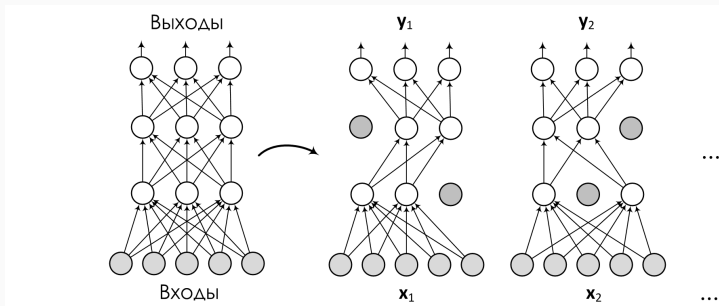


- Третий способ – max-norm constraint.
- Давайте искусственно ограничим норму вектора весов каждого нейрона:

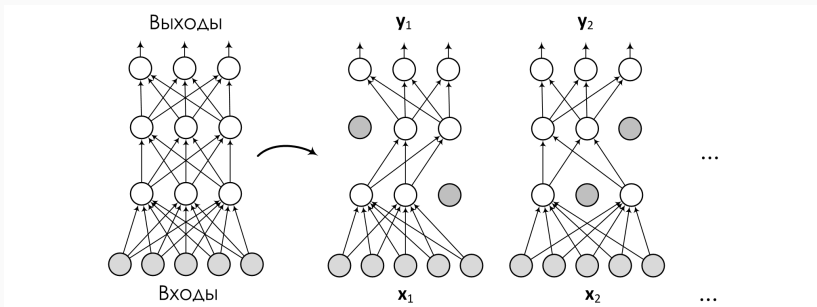
$$\|\mathbf{w}\|^2 \leq c.$$

- Это можно делать в процессе оптимизации: когда $\mathbf{w}$ выходит за шар радиуса c , проецируем его обратно.

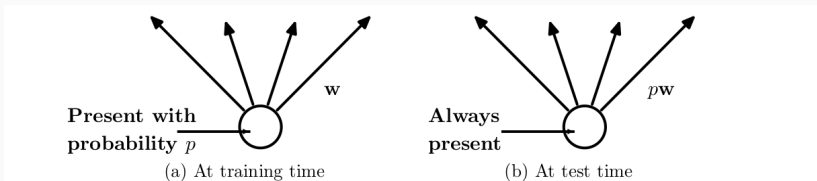
- Но есть и другие варианты.
- *Дропаут* (dropout): давайте просто выбросим некоторые нейроны случайным образом с вероятностью p !
(Srivastava et al., 2014)



- Получается, что мы сэмплируем кучу сетей, и нейрон получает на вход «среднюю» активацию от разных архитектур.
- Технический вопрос: как потом применять? Неужели надо опять сэмплировать кучу архитектур и усреднять?

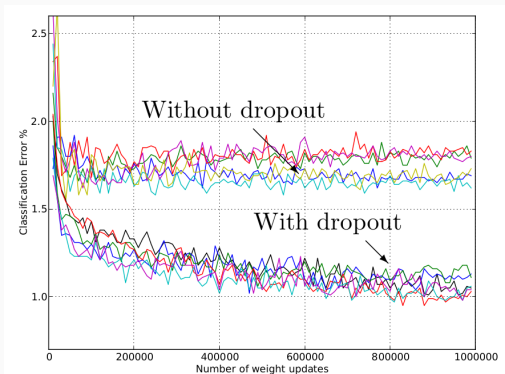


- Чтобы применить обученную сеть, умножим результат на $1/p$, сохраняя ожидание выхода!

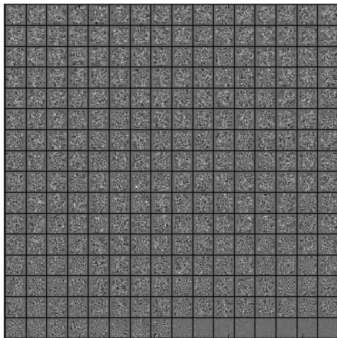


- В качестве вероятности часто можно брать просто $p = \frac{1}{2}$, большой разницы нет.

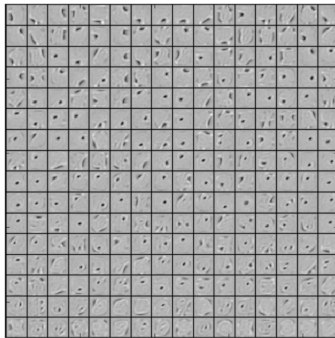
- Dropout улучшает (большие и свёрточные) нейронные сети *очень заметно*... но почему? WTF?



- Идея 1: нейроны теперь должны обучать признаки самостоятельно, а не рассчитывать на других.

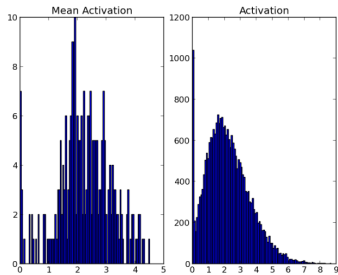


(a) Without dropout

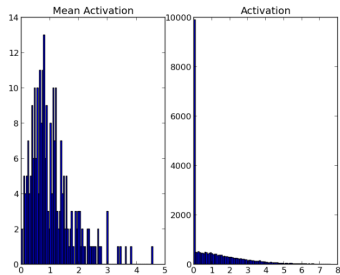


(b) Dropout with $p = 0.5$.

- Аналогично, и разреженность появляется.



(a) Without dropout



(b) Dropout with $p = 0.5$.

- Идея 2: мы как бы усредняем огромное число (2^N) сетей с общими весами, каждую обучая на один шаг.
- Усреднять кучу моделей очень полезно – bootstrapping/xgboost/всё такое...



- Получается, что дропаут – это такой экстремальный бутстреппинг.

- Идея 3: this is just like sex!
- Как работает половое размножение?
- Важно собрать не просто хорошую комбинацию, а *устойчивую* хорошую комбинацию.

BY ADI LIVNAT AND CHRISTOS PAPADIMITRIOU

Sex as an Algorithm

- Идея 4: нейрон посылает активацию a с вероятностью 0.5.
- Но можно наоборот: давайте посылать 0.5 (или 1) с вероятностью a .
- Ожидание то же, дисперсия для маленьких p растёт (что неплохо).
- И у нас получаются стохастические нейроны, которые посылают сигналы случайно – точно как в мозге!
- Улучшение примерно то же, как от dropout, но нужно меньше коммуникации между нейронами (один бит вместо float).
- Т.е. стохастические нейроны в мозге работают как дропаут-регуляризатор!
- Возможно, именно поэтому мы умеем задумываться.

- Идея 5: dropout — это специальная форма априорного распределения.
- Это очень полезный взгляд, с ним победили dropout в рекуррентных сетях.
- Но для этого нужно сначала поговорить о нейронных сетях по-байесовски...
- Вернёмся к этому, если будет время.

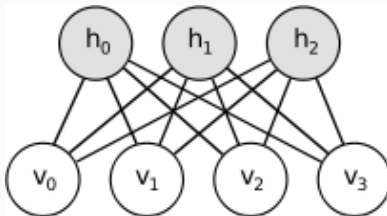
- Итого получается, что dropout – это очень крутой метод.



- Но и он сейчас отходит на второй план из-за нормализации по мини-батчам и новых версий градиентного спуска.
- О них чуть позже, а сначала об инициализации весов.

ИНИЦИАЛИЗАЦИЯ ВЕСОВ

- Революция глубокого обучения началась с *предобучением без учителя* (unsupervised pretraining).
- Главная идея: добраться до хорошей области пространства весов, затем уже сделать fine-tuning градиентным спуском.
- Ограниченные машины Больцмана (restricted Boltzmann machines):

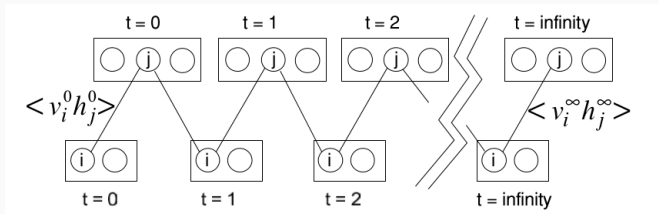


- Это ненаправленная графическая модель, задающая распределение

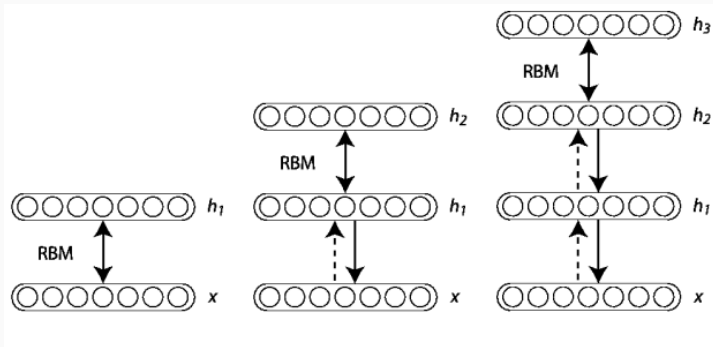
$$p(\mathbf{v}) = \sum_{\mathbf{h}} p(\mathbf{v}, \mathbf{h}) = \frac{1}{Z} \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}, \text{ где}$$

$$E(\mathbf{v}, \mathbf{h}) = -\mathbf{b}^\top \mathbf{v} - \mathbf{c}^\top \mathbf{h} - \mathbf{h}^\top W \mathbf{v}.$$

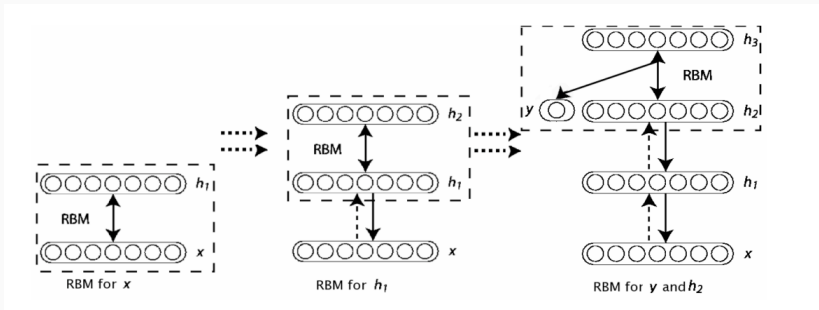
- Обучают алгоритмом Contrastive Divergence (приближение к сэмплированию по Гиббсу).



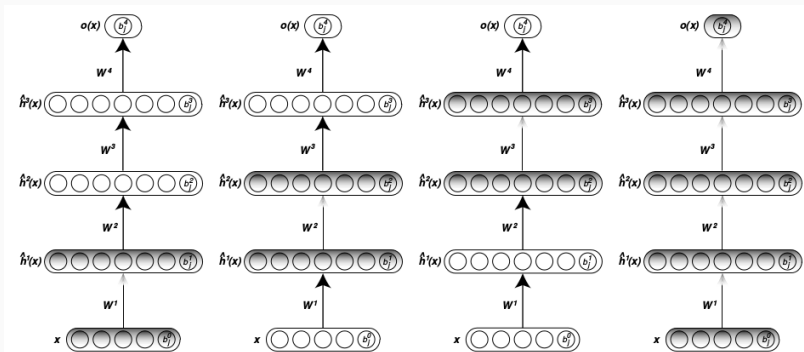
- Из RBM можно сделать глубокие сети, поставив одну на другую:



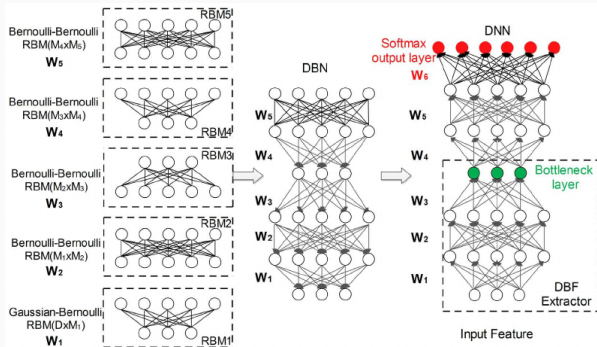
- И вывод можно вести последовательно, уровень за уровнем:



- А потом уже дообучать градиентным спуском (fine-tuning).



- Этот подход привёл к прорыву в распознавании речи.



- Но обучать глубокие сети из RBM довольно сложно, они хрупкие, и вычислительно тоже нелегко.
- И сейчас уже не очень-то и нужны сложные модели вроде RBM для того, чтобы попасть в хорошую начальную область.
- Инициализация весов — важная часть этого.

- *Xavier initialization* (Glorot, Bengio, 2010).
- Рассмотрим простой линейный нейрон:

$$y = \mathbf{w}^\top \mathbf{x} + b = \sum_i w_i x_i + b.$$

- Его дисперсия равна

$$\begin{aligned} \text{Var}[y_i] &= \text{Var}[w_i x_i] = \mathbb{E}[w_i^2 x_i^2] - (\mathbb{E}[w_i x_i])^2 = \\ &= \mathbb{E}[x_i]^2 \text{Var}[w_i] + \mathbb{E}[w_i]^2 \text{Var}[x_i] + \text{Var}[w_i] \text{Var}[x_i]. \end{aligned}$$

- Его дисперсия равна

$$\begin{aligned}\text{Var}[y_i] &= \text{Var}[w_i x_i] = \mathbb{E}[w_i^2 x_i^2] - (\mathbb{E}[w_i x_i])^2 = \\ &= \mathbb{E}[x_i]^2 \text{Var}[w_i] + \mathbb{E}[w_i]^2 \text{Var}[x_i] + \text{Var}[w_i] \text{Var}[x_i].\end{aligned}$$

- Для нулевого среднего весов

$$\text{Var}[y_i] = \text{Var}[w_i] \text{Var}[x_i].$$

- И если w_i и x_i инициализированы независимо из одного и того же распределения,

$$\text{Var}[y] = \text{Var}\left[\sum_{i=1}^{n_{\text{out}}} y_i\right] = \sum_{i=1}^{n_{\text{out}}} \text{Var}[w_i x_i] = n_{\text{out}} \text{Var}[w_i] \text{Var}[x_i].$$

- Иначе говоря, дисперсия на выходе пропорциональна дисперсии на входе с коэффициентом $n_{\text{out}} \text{Var}[w_i]$.

- До (Glorot, Bengio, 2010) стандартным способом инициализации было

$$w_i \sim U \left[-\frac{1}{\sqrt{n_{\text{out}}}}, \frac{1}{\sqrt{n_{\text{out}}}} \right].$$

- См., например, *Neural Networks: Tricks of the Trade*.
- Так что с дисперсиями получается

$$\text{Var}[w_i] = \frac{1}{12} \left(\frac{1}{\sqrt{n_{\text{out}}}} + \frac{1}{\sqrt{n_{\text{out}}}} \right)^2 = \frac{1}{3n_{\text{out}}}, \text{ и}$$

$$n_{\text{out}} \text{Var}[w_i] = \frac{1}{3},$$

и после нескольких уровней сигнал совсем умирает;
аналогичный эффект происходит и в backprop.

- Инициализация Ксавье — давайте попробуем уменьшить изменение дисперсии, т.е. взять

$$\text{Var}[w_i] = \frac{2}{n_{\text{in}} + n_{\text{out}}};$$

для равномерного распределения это

$$w_i \sim U \left[-\frac{\sqrt{6}}{\sqrt{n_{\text{in}} + n_{\text{out}}}}, \frac{\sqrt{6}}{\sqrt{n_{\text{in}} + n_{\text{out}}}} \right].$$

- Но это работает только для симметричных активаций, т.е. не для ReLU...

- ...до работы (He et al., 2015). Вернёмся к

$$\text{Var} [w_i x_i] = \mathbb{E} [x_i]^2 \text{Var} [w_i] + \mathbb{E} [w_i]^2 \text{Var} [x_i] + \text{Var} [w_i] \text{Var} [x_i]$$

- Мы теперь можем обнулить только второе слагаемое:

$$\text{Var} [w_i x_i] = \mathbb{E} [x_i]^2 \text{Var} [w_i] + \text{Var} [w_i] \text{Var} [x_i] = \text{Var} [w_i] \mathbb{E} [x_i^2], \text{ и}$$

$$\text{Var} [y^{(l)}] = n_{\text{in}}^{(l)} \text{Var} [w^{(l)}] \mathbb{E} \left[(x^{(l)})^2 \right].$$

- Мы теперь можем обнулить только второе слагаемое:

$$\text{Var} [y^{(l)}] = n_{\text{in}}^{(l)} \text{Var} [w^{(l)}] \mathbb{E} \left[(x^{(l)})^2 \right].$$

- Предположим, что $x^{(l)} = \max(0, y^{(l-1)})$, и у $y^{(l-1)}$ симметричное распределение вокруг нуля. Тогда

$$\mathbb{E} \left[(x^{(l)})^2 \right] = \frac{1}{2} \text{Var} [y^{(l-1)}], \quad \text{Var} [y^{(l)}] = \frac{n_{\text{in}}^{(l)}}{2} \text{Var} [w^{(l)}] \text{Var} [y^{(l-1)}].$$

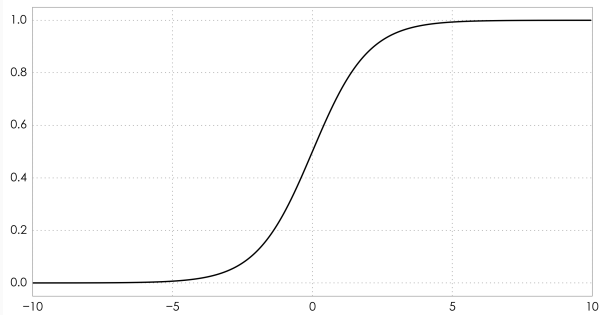
- И это приводит к формуле для дисперсии активации ReLU; теперь нет никакого n_{out} :

$$\text{Var}[w_i] = 2/n_{\text{in}}^{(l)}.$$

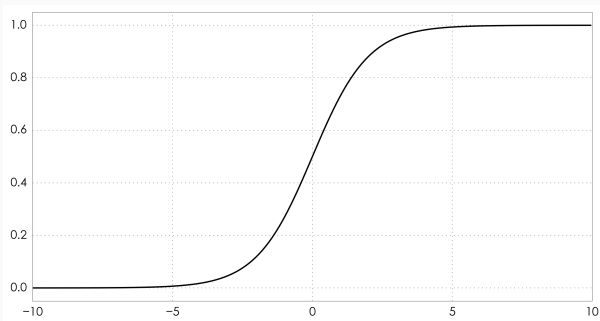
- Кстати, равномерную инициализацию делать не обязательно, можно и нормальное распределение:

$$w_i \sim N\left(0, \sqrt{2/n_{\text{in}}^{(l)}}\right).$$

- Кстати, о (Glorot, Bengio, 2010) – ещё одна важная идея.
- Эксперименты показали, что $\sigma(x) = \frac{1}{1+e^{-x}}$ работает в глубоких сетях довольно плохо.



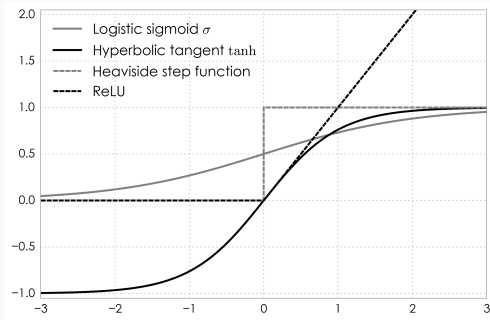
- *Насыщение*: если $\sigma(x)$ уже «обучилась», т.е. даёт большие по модулю значения, то её производная близка к нулю и «поменять мнение» трудно.



- Но ведь другие тоже насыщаются? В чём разница?

- Рассмотрим последний слой сети $h(W\mathbf{a} + \mathbf{b})$, где $\mathbf{a}$ — выходы предыдущего слоя, $\mathbf{b}$ — свободные члены, h — функция активации последнего уровня, обычно softmax .
- Когда мы начинаем оптимизировать сложную функцию потерь, поначалу выходы $\mathbf{h}$ не несут полезной информации о входах, ведь первые уровни ещё не обучены.
- Тогда неплохим приближением будет константная функция, выдающая средние значения выходов.
- Это значит, что $h(W\mathbf{a} + \mathbf{b})$ подберёт подходящие свободные члены $\mathbf{b}$ и постарается обнулить слагаемое $W\mathbf{h}$, которое поначалу скорее шум, чем сигнал.

- Иначе говоря, в процессе обучения мы постараемся привести выходы предыдущего слоя к нулю.
- Здесь и проявляется разница: у $\sigma(x) = \frac{1}{1+e^{-x}}$ область значений $(0, 1)$ при среднем $\frac{1}{2}$, и при $\sigma(x) \rightarrow 0$ будет и $\sigma'(x) \rightarrow 0$.
- А у $\tanh$ наоборот: когда $\tanh(x) \rightarrow 0$, $\tanh'(x)$ максимальна.

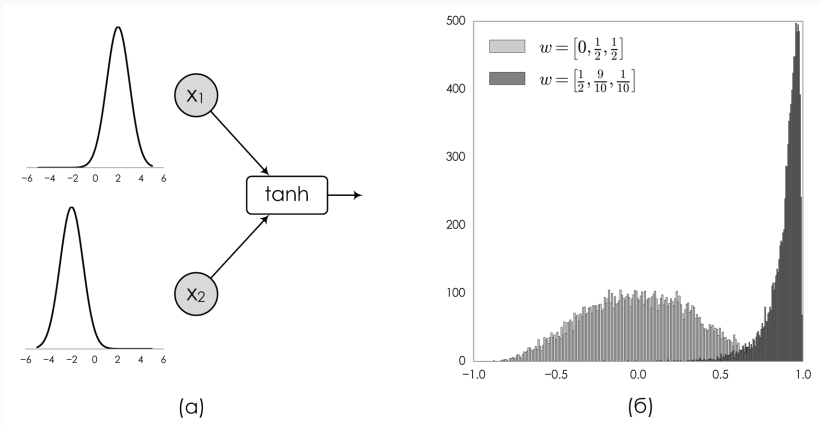


НОРМАЛИЗАЦИЯ ПО МИНИ-БАТЧАМ

- Ещё одна важная проблема в глубоких сетях: *внутренний сдвиг переменных* (internal covariate shift).
- Когда меняются веса слоя, меняется распределение его выходов.
- Это значит, что следующему уровню придётся всё начинать заново, он же не ожидал таких входов, не видел их раньше!
- Более того, нейроны следующего уровня могли уже и насытиться, и им теперь сложно быстро обучиться заново.
- Это серьёзно мешает обучению.

НОРМАЛИЗАЦИЯ ПО МИНИ-БАТЧАМ

- Вот характерный пример:



- Что делать?

НОРМАЛИЗАЦИЯ ПО МИНИ-БАТЧАМ

- Можно пытаться нормализовать входы каждого уровня.
- Не работает: рассмотрим для простоты уровень с одним только bias b и входами $\mathbf{u}$:

$$\hat{\mathbf{x}} = \mathbf{x} - \mathbb{E}[\mathbf{x}], \text{ где } \mathbf{x} = \mathbf{u} + b.$$

- На следующем шаге градиентного спуска получится $b := b + \Delta b...$
- ...но $\hat{\mathbf{x}}$ не изменится:

$$\mathbf{u} + b + \Delta b - \mathbb{E}[\mathbf{u} + b + \Delta b] = \mathbf{u} + b - \mathbb{E}[\mathbf{u} + b].$$

- Так что всё обучение сведётся к тому, что b будет неограниченно расти — не очень хорошо.

- Можно пытаться добавить нормализацию как отдельный слой:

$$\hat{\mathbf{x}} = \text{Norm}(\mathbf{x}, X).$$

- Это лучше, но теперь этому слою на входе нужен весь датасет X !
- И на шаге градиентного спуска придётся вычислить $\frac{\partial \text{Norm}}{\partial \mathbf{x}}$ и $\frac{\partial \text{Norm}}{\partial X}$, да ещё и матрицу ковариаций

$$\text{Cov}[\mathbf{x}] = \mathbb{E}_{\mathbf{x} \in X} [\mathbf{x} \mathbf{x}^\top] - \mathbb{E}[\mathbf{x}] \mathbb{E}[\mathbf{x}]^\top.$$

- Это точно не работает.

- Решение в том, чтобы нормализовать каждый вход отдельно, и не по всему датасету, а по текущему кусочку; это и есть *нормализация по мини-батчам* (batch normalization).
- После нормализации по мини-батчам получим

$$\hat{x}_k = \frac{x_k - \mathbb{E}[x_k]}{\sqrt{\text{Var}[x_k]}},$$

где статистики подсчитаны по текущему мини-батчу.

- Ещё одна проблема: теперь пропадают нелинейности!
- Например, σ теперь практически всегда близка к линейной.

- Чтобы это исправить, нужно добавить гибкости уровню batchnorm.
- В частности, нужно разрешить ему обучаться иногда *ничего не делать* со входами.
- Так что вводим дополнительные параметры (сдвиг и растяжение):

$$y_k = \gamma_k \hat{x}_k + \beta_k = \gamma_k \frac{x_k - \mathbb{E}[x_k]}{\sqrt{\text{Var}[x_k]}} + \beta_k.$$

- γ_k и β_k — это новые переменные, тоже будут обучаться градиентным спуском, как веса.

НОРМАЛИЗАЦИЯ ПО МИНИ-БАТЧАМ

- И ещё добавим ϵ в знаменатель, чтобы на ноль не делить.
- Теперь мы можем формально описать слой батч-нормализации — для очередного мини-батча $B = \{\mathbf{x}_1, \dots, \mathbf{x}_m\}$:

- вычислить базовые статистики по мини-батчу

$$\mu_B = \frac{1}{m} \sum_{i=1}^m \mathbf{x}_i, \quad \sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (\mathbf{x}_i - \mu_B)^2,$$

- нормализовать входы

$$\hat{\mathbf{x}}_i = \frac{\mathbf{x}_i - \mu_b}{\sqrt{\sigma_B^2 + \epsilon}},$$

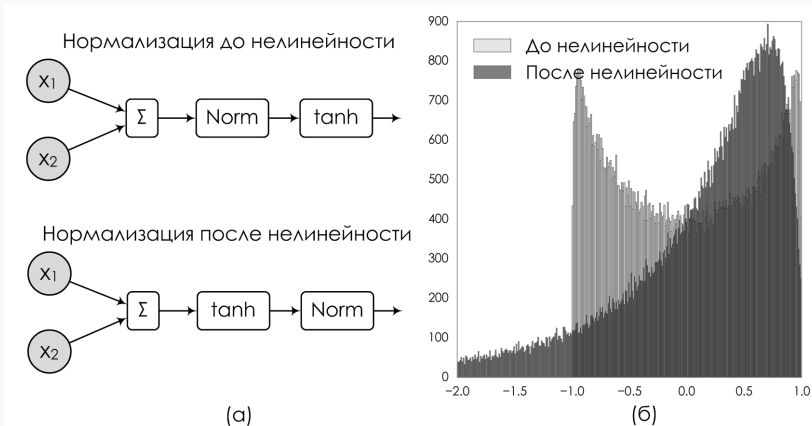
- вычислить результат

$$\mathbf{y}_i = \gamma \mathbf{x}_i + \beta.$$

- Через всё это совершенно стандартным образом пропускаются градиенты, в том числе по γ и β .

НОРМАЛИЗАЦИЯ ПО МИНИ-БАТЧАМ

- Последнее замечание: важно, куда поместить слой batchnorm.
- Можно до, а можно после нелинейности.



Что BN на самом деле делает

- Изначально идея была в том, чтобы сократить internal covariate shift.
- Но выяснилось, что от этого эффективность BN не особенно зависит, но BN помогает с регуляризацией, делает дропаут ненужным и т.д. Возможные причины:
 - BN сглаживает градиенты, уменьшает липшицеву константу (Santurkar et al., 2019);
 - BN разделяет обучение направления и длины векторов весов, что улучшает обучение;
 - есть результаты о том, что BN помогает добиться линейной сходимости в обычном градиентном спуске.

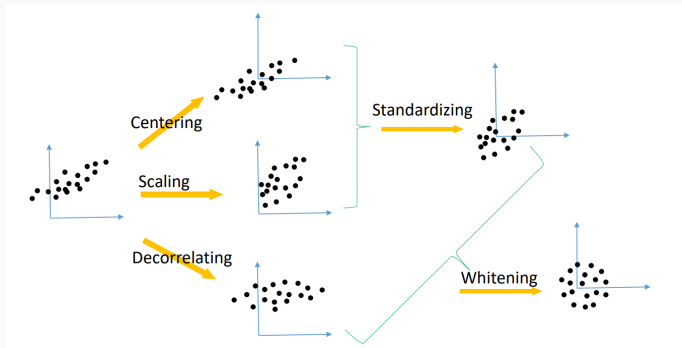
- Нормализация по мини-батчам сейчас стала фактически стандартом.
- Очередная очень крутая история, примерно как дропаут.
- Но мысль идёт и дальше:
 - (Laurent et al., 2016): BN не помогает рекуррентным сетям;
 - (Cooijmans et al., 2016): recurrent batch normalization;
 - (Salimans and Kingma, 2016): нормализация весов для улучшения обучения — давайте добавим веса как

$$\mathbf{h}_i = f \left(\frac{\gamma}{\|\mathbf{w}_i\|} \mathbf{w}_i^\top \mathbf{x} + b_i \right);$$

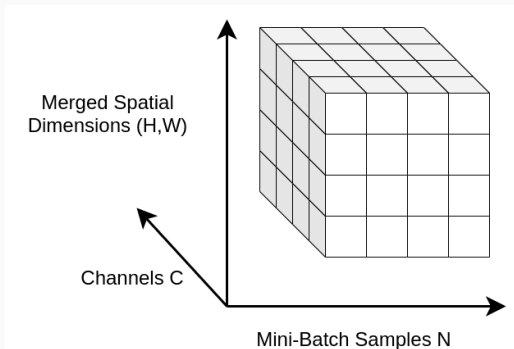
тогда мы будем перемасштабировать градиент, стабилизировать его норму и приближать его матрицу ковариаций к единичной, что улучшает обучение.

- Давайте разберёмся в ландшафте методов нормализации

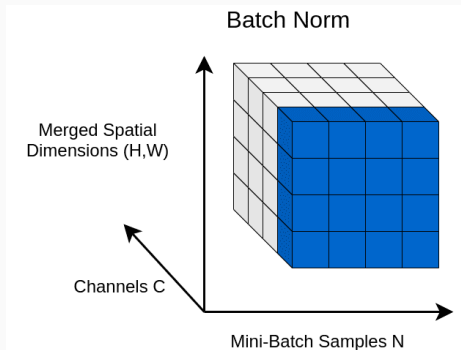
- В целом все эти методы используют одни и те же операции:



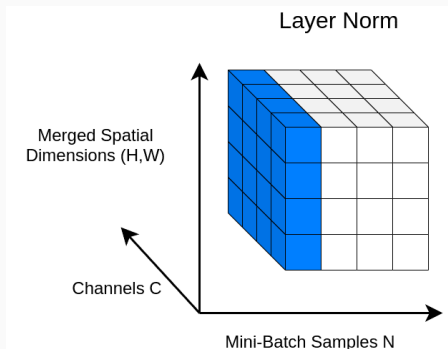
- И различаются набором операций и тем, по каким размерностям они применяются:



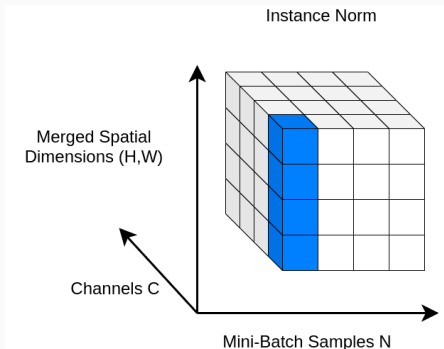
- Batch norm – по мини-батчу (часто используется для изображений):



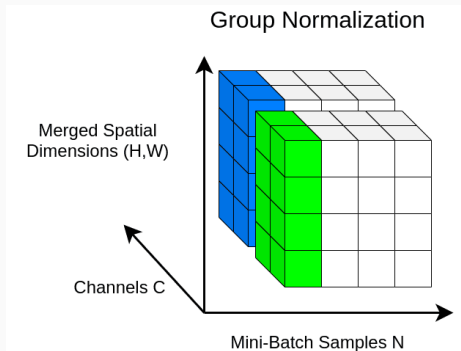
- Layer norm – по каналам и по размерности, т.е. по всем выходам слоя (используется в основном для векторов, в частности в трансформерах):



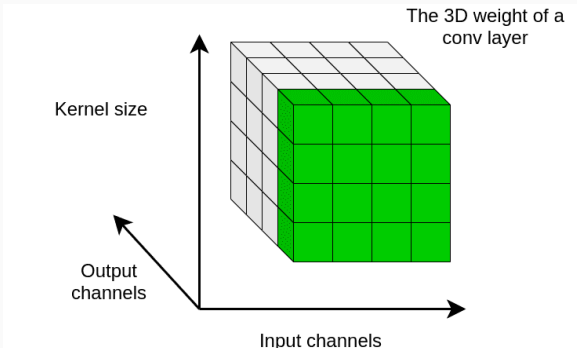
- Instance norm – только по размерности, т.е. каналы убираем (используется, например, в AdaIN для переноса стиля):



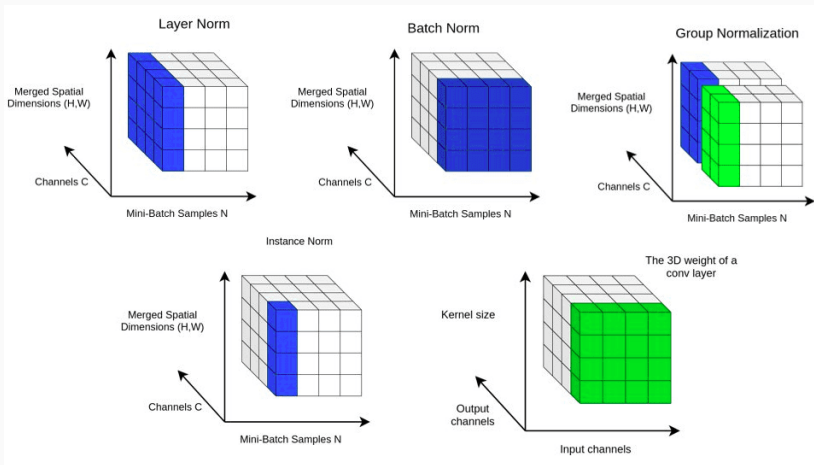
- Group norm – можно разделить каналы на группы:



- Weight standardization – усредним веса, а не активации, отдельно для каждого канала, т.е. в том числе нормализуем градиенты при backpropagation (group norm + weight standardization использовались в BiT – Big Transfer):



- Итого:



- Но вообще это, как всегда, большая наука (Huang et al., 2020):

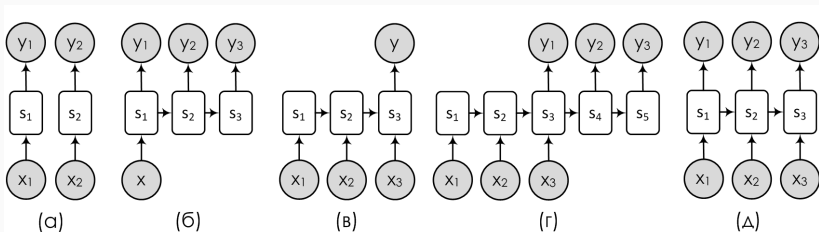
Method	NAP	NOP	NRR	Published In
Local contrast normalization [108]	$\Pi_{IN}(\mathbf{X}) \in \mathbb{R}^{md \times hw}$; local spatial positions	Standardizing	No	ICCV, 2009
Local response normalization [75]	$\Pi_{PN}(\mathbf{X}) \in \mathbb{R}^{mhw \times d}$; local channels	Scaling	No	NeurIPS, 2012
Batch normalization (BN) [8]	$\Pi_{BN}(\mathbf{X}) \in \mathbb{R}^{d \times mhw}$	Standardizing	Learnable $\gamma, \beta \in \mathbb{R}^d$	ICML, 2015
Mean-only BN [21]	$\Pi_{BN}(\mathbf{X}) \in \mathbb{R}^{d \times mhw}$	Centering	No	NeurIPS, 2016
Layer normalization (LN) [20]	$\Pi_{LN}(\mathbf{X}) \in \mathbb{R}^{m \times dhw}$	Standardizing	Learnable $\gamma, \beta \in \mathbb{R}^d$	Arxiv, 2016
Instance normalization (IN) [67]	$\Pi_{IN}(\mathbf{X}) \in \mathbb{R}^{md \times hw}$	Standardizing	Learnable $\gamma, \beta \in \mathbb{R}^d$	Arxiv, 2016
L^p -Norm BN [90]	$\Pi_{BN}(\mathbf{X}) \in \mathbb{R}^{d \times mhw}$	Standardizing with L^p -Norm divided	Learnable $\gamma, \beta \in \mathbb{R}^d$	Arxiv, 2016
Divisive normalization [76]	$\Pi_{LN}(\mathbf{X}) \in \mathbb{R}^{m \times dhw}$; local spatial positions and channels	Standardizing	Learnable $\gamma, \beta \in \mathbb{R}^d$	ICLR, 2017
Conditional IN [68]	$\Pi_{IN}(\mathbf{X}) \in \mathbb{R}^{md \times hw}$	Standardizing	Side information	ICLR, 2017
Dynamic LN [101]	$\Pi_{LN}(\mathbf{X}) \in \mathbb{R}^{m \times dhw}$	Standardizing	Generated $\gamma, \beta \in \mathbb{R}^d$	INTERSPEECH, 2017
Conditional BN [107]	$\Pi_{BN}(\mathbf{X}) \in \mathbb{R}^{d \times mhw}$	Standardizing	Side information	NeurIPS, 2017
Pixel normalization [72]	$\Pi_{PN}(\mathbf{X}) \in \mathbb{R}^{mhw \times d}$	Scaling	No	ICLR, 2018
Decorrelated BN [34]	$\Pi_{BN}(\mathbf{X}) \in \mathbb{R}^{d \times mhw}$	ZCA whitening	Learnable $\gamma, \beta \in \mathbb{R}^d$	CVPR, 2018
Group normalization (GN) [22]	$\Pi_{GN}(\mathbf{X}) \in \mathbb{R}^{mg \times shw}$	Standardizing	Learnable $\gamma, \beta \in \mathbb{R}^d$	ECCV, 2018

- Правда большая:

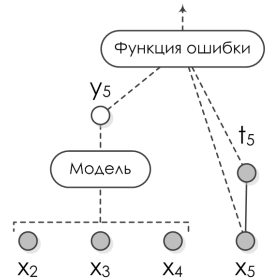
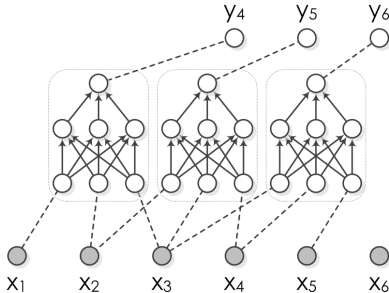
Adaptive IN [69]	$\Pi_{IN}(\mathbf{X}) \in \mathbb{R}^{md \times hw}$	Standardizing	Generated $\gamma, \beta \in \mathbb{R}^d$	ECCV, 2018
L^1 -Norm BN [91], [92]	$\Pi_{BN}(\mathbf{X}) \in \mathbb{R}^{d \times mhw}$	Standardizing with L^1 -Norm divided	Learnable $\gamma, \beta \in \mathbb{R}^d$	NeurIPS, 2018
Whitening and coloring BN [79]	$\Pi_{BN}(\mathbf{X}) \in \mathbb{R}^{d \times mhw}$	CD whitening	Color transformation	ICLR, 2019
Generalized BN [93]	$\Pi_{BN}(\mathbf{X}) \in \mathbb{R}^{d \times mhw}$	General standardizing	Learnable $\gamma, \beta \in \mathbb{R}^d$	AAAI, 2019
Iterative normalization [81]	$\Pi_{BN}(\mathbf{X}) \in \mathbb{R}^{d \times mhw}$	ZCA whitening by Newton's iteration	Learnable $\gamma, \beta \in \mathbb{R}^d$	CVPR, 2019
Instance-level meta normalization [103]	$\Pi_{LN}(\mathbf{X})/\Pi_{IN}(\mathbf{X})/\Pi_{GN}(\mathbf{X})$	Standardizing	Learnable & generated $\gamma, \beta \in \mathbb{R}^d$	CVPR, 2019
Spatially adaptive denormalization [109]	$\Pi_{BN}(\mathbf{X}) \in \mathbb{R}^{d \times mhw}$	Standardizing	Generated $\gamma, \beta \in \mathbb{R}^{d \times h \times w}$	CVPR, 2019
Position normalization (PN) [71]	$\Pi_{PN}(\mathbf{X}) \in \mathbb{R}^{mhw \times d}$	Standardizing	Learnable $\gamma, \beta \in \mathbb{R}^d$	NeurIPS, 2019
Root mean square LN [97]	$\Pi_{LN}(\mathbf{X}) \in \mathbb{R}^{m \times dhw}$	Scaling	Learnable $\gamma \in \mathbb{R}^d$	NeurIPS, 2019
Online normalization [99]	$\Pi_{LN}(\mathbf{X}) \in \mathbb{R}^{m \times dhw}$	Scaling	No	NeurIPS, 2019
Batch group normalization [73]	$\Pi_{BGN}(\mathbf{X}) \in \mathbb{R}^{gmg \times smshw}$	Standardizing	Learnable $\gamma, \beta \in \mathbb{R}^d$	ICLR, 2020
Instance enhancement BN [105]	$\Pi_{BN}(\mathbf{X}) \in \mathbb{R}^{d \times mhw}$	Standardizing	Learnable & generated $\gamma, \beta \in \mathbb{R}^d$	AAAI, 2020
PowerNorm [96]	$\Pi_{BN}(\mathbf{X}) \in \mathbb{R}^{d \times mhw}$	Scaling	Learnable $\gamma, \beta \in \mathbb{R}^d$	ICML, 2020
Local context normalization [74]	$\Pi_{GN}(\mathbf{X}) \in \mathbb{R}^{mg \times shw}$; local spatial positions and channels	Standardizing	Learnable $\gamma, \beta \in \mathbb{R}^d$	CVPR, 2020
Filter response normalization [100]	$\Pi_{IN}(\mathbf{X}) \in \mathbb{R}^{md \times hw}$	Scaling	Learnable $\gamma, \beta \in \mathbb{R}^d$	CVPR, 2020
Attentive normalization [106]	$\Pi_{BN}(\mathbf{X})/\Pi_{IN}(\mathbf{X})/\Pi_{LN}(\mathbf{X})/\Pi_{GN}(\mathbf{X})$	Standardizing	Generated $\gamma, \beta \in \mathbb{R}^d$	ECCV, 2020

РЕКУРРЕНТНЫЕ НЕЙРОННЫЕ СЕТИ

- Последовательности: текст, временные ряды, речь, музыка...
- Есть разные виды задач, основанных на последовательностях:

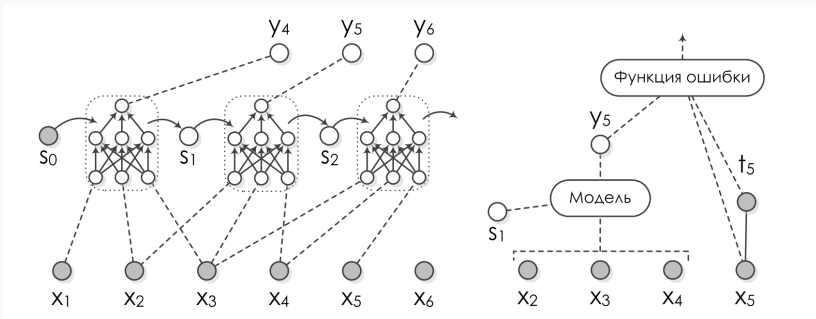


- Как применить к последовательности нейронную сеть?
- Можно использовать скользящее окно:



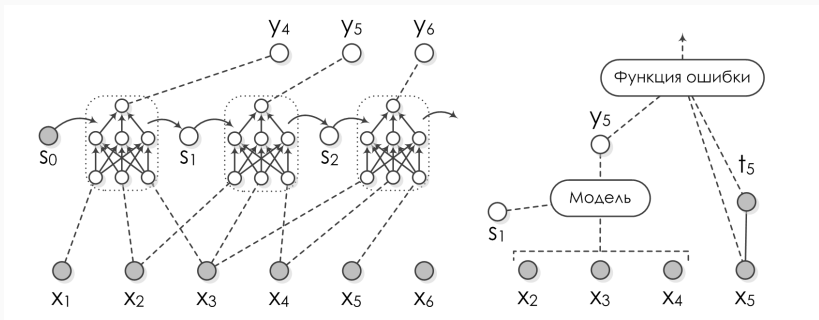
ПОСЛЕДОВАТЕЛЬНОСТИ

- ...но ещё лучше будет сохранять какое-нибудь скрытое состояние и обновлять его каждый раз.
- Это в точности идея *рекуррентных нейронных сетей* (recurrent neural networks, RNN).



- Но как теперь делать backpropagation? Получается, что в графе вычислений теперь циклы:

$$s_i = h(x_i, x_{i+1}, x_{i+2}, s_{i-1}).$$



- Это же ужасно, и всё сломалось?..

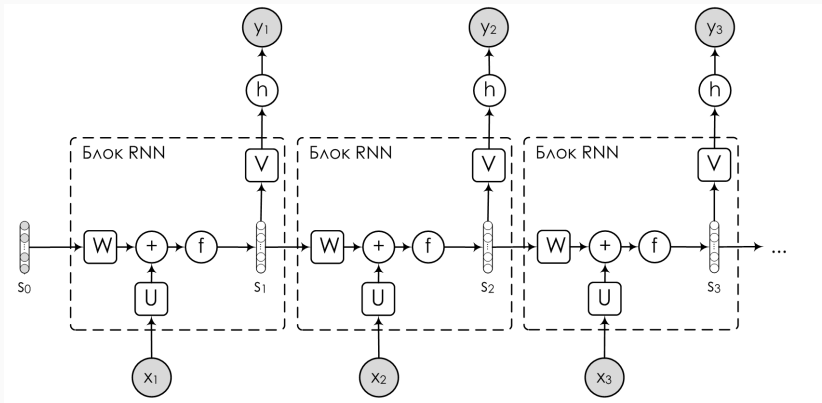
- ...да нет, конечно. Можно “развернуть” циклы обратно:

$$\begin{aligned}y_6 &= f(x_3, x_4, x_5, s_2) = f(x_3, x_4, x_5, h(x_2, x_3, x_4, s_1)) = \\&= f(x_3, x_4, x_5, h(x_2, x_3, x_4, h(x_1, x_2, x_3, s_0)))\end{aligned}$$

- Так что формально проблемы нет.
- Но масса проблем в реальности: получается, что рекуррентная сеть – это такая *очень* глубокая сеть с кучей общих весов...

Простая RNN

- “Простая” RNN:



- Формально:

$$\mathbf{a}_t = \mathbf{b} + W\mathbf{s}_{t-1} + U\mathbf{x}_t,$$

$$\mathbf{s}_t = f(\mathbf{a}_t),$$

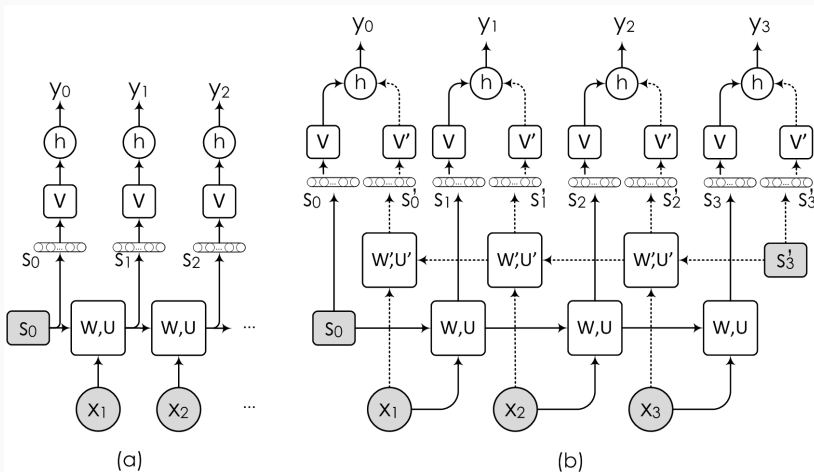
$$\mathbf{o}_t = \mathbf{c} + V\mathbf{s}_t,$$

$$\mathbf{y}_t = h(\mathbf{o}_t),$$

где f – рекуррентная нелинейность, h – функция выхода.

ДВУНАПРАВЛЕННАЯ RNN

- Иногда нужен контекст с обеих сторон:



- Формально:

$$\mathbf{s}_t = \sigma(\mathbf{b} + W\mathbf{s}_{t-1} + U\mathbf{x}_t),$$

$$\mathbf{s}'_t = \sigma(\mathbf{b}' + W'\mathbf{s}'_{t+1} + U'\mathbf{x}_t),$$

$$\mathbf{o}_t = \mathbf{c} + V\mathbf{s}_t + V'\mathbf{s}'_t,$$

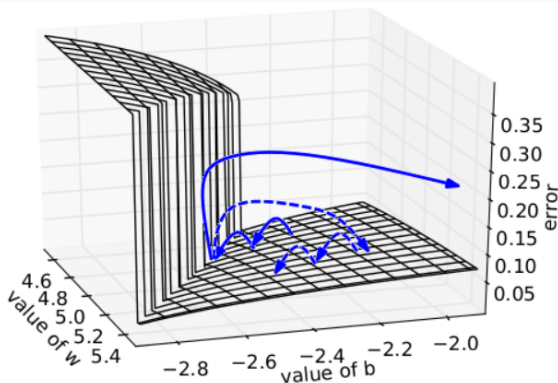
$$\mathbf{y}_t = h(\mathbf{o}_t).$$

- И это, конечно, обобщается на любой другой тип конструкций.

- Две проблемы:
 - взрывающиеся градиенты (exploding gradients);
 - затухающие градиенты (vanishing gradients).
- Надо каждый раз умножать на одну и ту же W , и норма градиента может расти или убывать экспоненциально.
- Взрывающиеся градиенты: надо каждый раз умножать на W , и норма градиента может расти экспоненциально.
- Что делать?

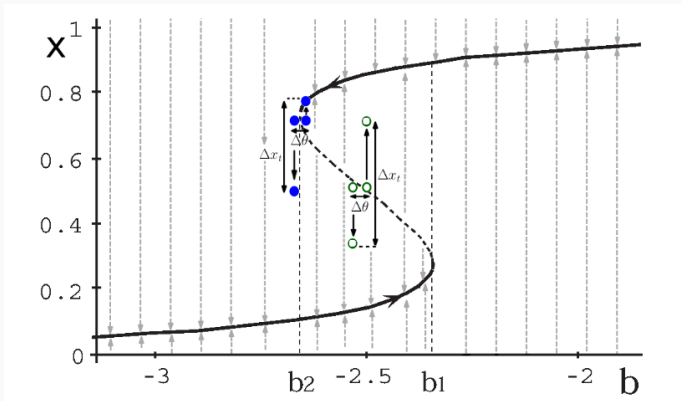
- Да просто обрезать градиенты, ограничить сверху, чтобы не росли.
- Два варианта – ограничить общую норму или каждое значение:
 - `sgd = optimizers.SGD(lr=0.01, clipnorm=1.)`
 - `sgd = optimizers.SGD(lr=0.01, clipvalue=.05)`

- (Pascanu et al., 2013) – вот что будет происходить:



ВЗРЫВАЮЩИЕСЯ ГРАДИЕНТЫ

- Там же объясняется, откуда возьмется такие перепады: есть точки бифуркации у RNN.



КАРУСЕЛЬ КОНСТАНТНОЙ ОШИБКИ: LSTM и GRU

- Затухающие градиенты: надо каждый раз умножать на W .
- Поэтому не получается долгосрочную память реализовать.



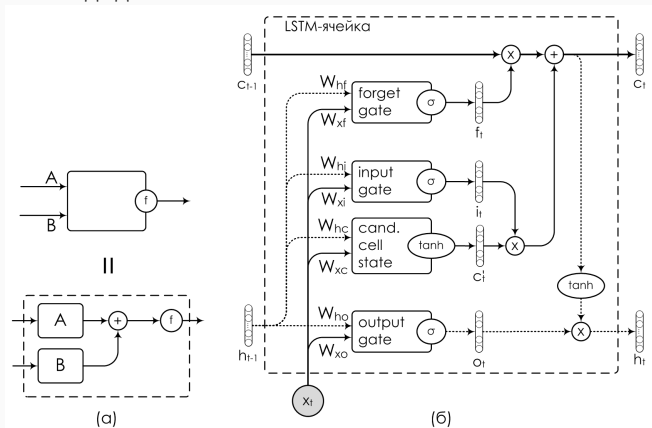
- А хочется. Что делать?..

- Базовую идею мы уже видели в ResNet: надо сделать так, чтобы градиент проходил.
- В RNN это называется «карусель константной ошибки» (constant error carousel).



- Идея из середины 1990-х (Шмидхубер): давайте составлять RNN из более сложных частей, в которых будет прямой путь для градиентов, и память будет контролироваться явно.

- LSTM (long short-term memory). “Ванильный” LSTM: $\mathbf{c}_t$ – состояние ячейки памяти, $\mathbf{h}_t$ – скрытое состояние.
- Input gate и forget gate определяют, надо ли менять $\mathbf{c}_t$ на нового кандидата в состояния ячейки.



- Формально:

$$\begin{aligned}
 \mathbf{c}'_t &= \tanh(W_{xc}\mathbf{x}_t + W_{hc}\mathbf{h}_{t-1} + \mathbf{b}_{c'}) && \text{candidate cell state} \\
 \mathbf{i}_t &= \sigma(W_{xi}\mathbf{x}_t + W_{hi}\mathbf{h}_{t-1} + \mathbf{b}_i) && \text{input gate} \\
 \mathbf{f}_t &= \sigma(W_{xf}\mathbf{x}_t + W_{hf}\mathbf{h}_{t-1} + \mathbf{b}_f) && \text{forget gate} \\
 \mathbf{o}_t &= \sigma(W_{xo}\mathbf{x}_t + W_{ho}\mathbf{h}_{t-1} + \mathbf{b}_o) && \text{output gate} \\
 \mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{c}'_t, && \text{cell state} \\
 \mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t) && \text{block output}
 \end{aligned}$$

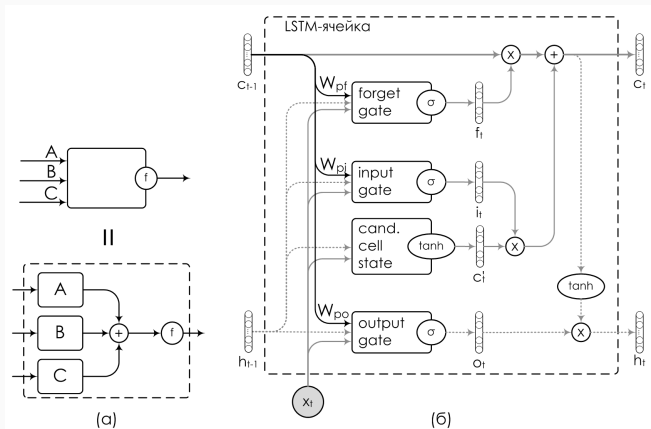
- Так что LSTM может контролировать состояние ячейки при помощи скрытого состояния и весов.
- Например, если forget gate закрыт ($\mathbf{f}_t = 1$), то получится карусель константной ошибки: $\mathbf{c}_t = \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{c}'_t$, и $\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} = 1$.
- Важно инициализировать $\mathbf{b}_f$ большим, чтобы forget gate был закрыт поначалу.

- LSTM был создан в середине 1990-х (Hochreiter and Schmidhuber, 1995; 1997).
- В полностью современной форме в (Gers, Schmidhuber, 2000).
- Проблема: хотим управлять $\mathbf{c}$, но гейты его не получают! Они видят только $\mathbf{h}_{t-1}$, а это

$$\mathbf{h}_{t-1} = \mathbf{o}_{t-1} \odot \tanh(\mathbf{c}_{t-1}).$$

- Так что если output gate закрыт, то поведение LSTM вообще от состояния ячейки не зависит.
- Нехорошо. Что делать?..

- ...конечно, добавить ещё несколько матриц! (peepholes)



- Формально:

$$\mathbf{i}_t = \sigma (W_{xi}\mathbf{x}_t + W_{hi}\mathbf{h}_{t-1} + W_{pi}\mathbf{c}_{t-1} + \mathbf{b}_i)$$

$$\mathbf{f}_t = \sigma (W_{xf}\mathbf{x}_t + W_{hf}\mathbf{h}_{t-1} + W_{pf}\mathbf{c}_{t-1} + \mathbf{b}_f)$$

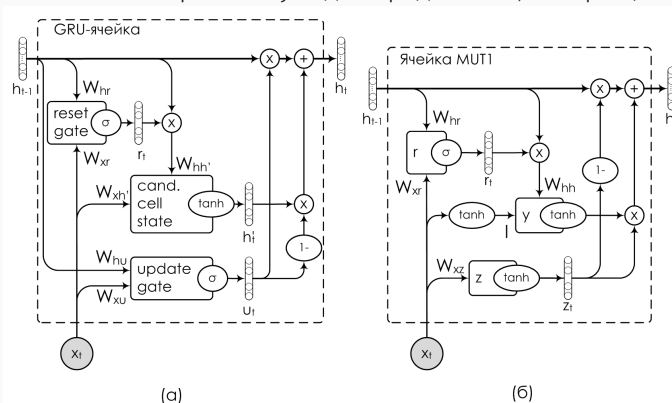
$$\mathbf{o}_t = \sigma (W_{xo}\mathbf{x}_t + W_{ho}\mathbf{h}_{t-1} + W_{po}\mathbf{c}_{t-1} + \mathbf{b}_o)$$

- Видно, что тут есть огромное поле для вариантов LSTM: можно удалить любой гейт, любую замочную скважину, поменять функции активации...
- Как выбрать?

- «LSTM: a Search Space Odyssey» (Greff et al., 2015).
- Большое экспериментальное сравнение.
- В честности, некоторые куда более простые архитектуры (без одного из гейтов!) не сильно проигрывали «ванильному» LSTM.
- И это приводит нас к...



- ...Gated Recurrent Units (GRU; Cho et al., 2014).
- В GRU тоже есть прямой путь для градиентов, но проще.



- Формально:

$$\mathbf{u}_t = \sigma(W_{xu}\mathbf{x}_t + W_{hu}\mathbf{h}_{t-1} + \mathbf{b}_u)$$

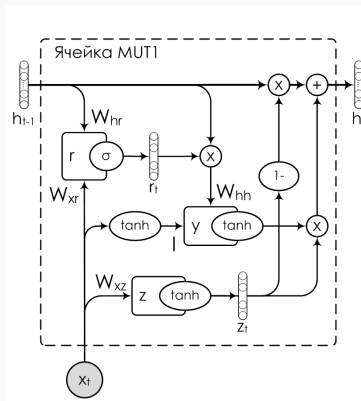
$$\mathbf{r}_t = \sigma(W_{xr}\mathbf{x}_t + W_{hr}\mathbf{h}_{t-1} + \mathbf{b}_r)$$

$$\mathbf{h}'_t = \tanh(W_{xh'}\mathbf{x}_t + W_{hh'}(\mathbf{r}_t \odot \mathbf{h}_{t-1}))$$

$$\mathbf{h}_t = (1 - \mathbf{u}_t) \odot \mathbf{h}'_t + \mathbf{u}_t \odot \mathbf{h}_{t-1}$$

- Теперь есть update gate и reset gate, нет разницы между $\mathbf{c}_t$ и $\mathbf{h}_t$.
- Меньше матриц (6, а не 8 или 11 с замочными скважинами), меньше весов, но только чуть хуже LSTM работает.
- Так что можно больше GRU поместить, и сеть станет лучше.

- Другие варианты тоже есть.
- (Józefowicz, Zaremba, Sutskever, 2015): огромное сравнение, выращивали архитектуры эволюционными методами.
- Три новых интересных архитектуры; например:



Долгосрочная память в базовых RNN

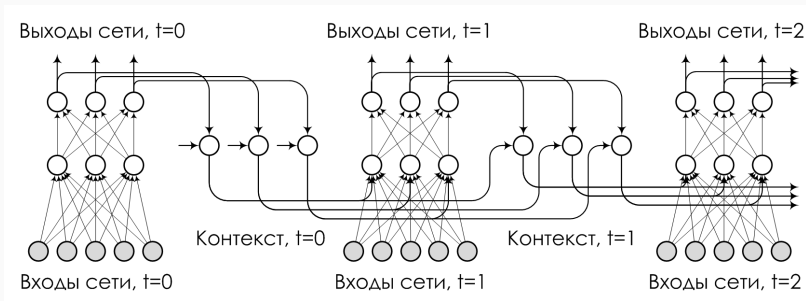
- Следующая идея о том, как добавить долгосрочную память.
- Начнём опять с простой RNN:

$$\mathbf{s}_t = f(U\mathbf{x}_t + W\mathbf{s}_{t-1} + \mathbf{b}), \quad \mathbf{y}_t = h(U\mathbf{s}_t + \mathbf{c}).$$

- Проблема с градиентами в том, что мы умножаем на W , и градиенты либо взрываются, либо затухают.
- Давайте вернёмся к истории RNN...

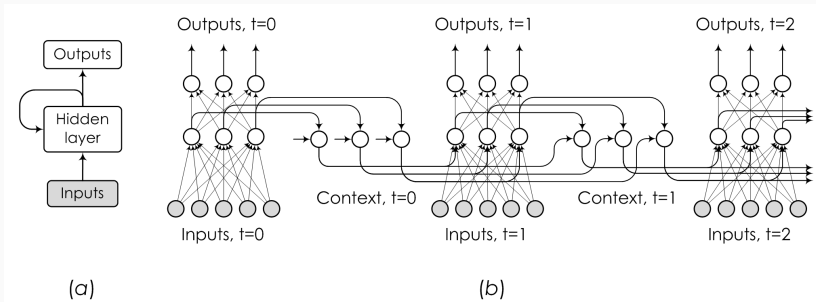


- Сеть Джордана (середина 1980-х):



- Считается первой успешной RNN.

- Сеть Элмана (Elman; конец 1980-х):



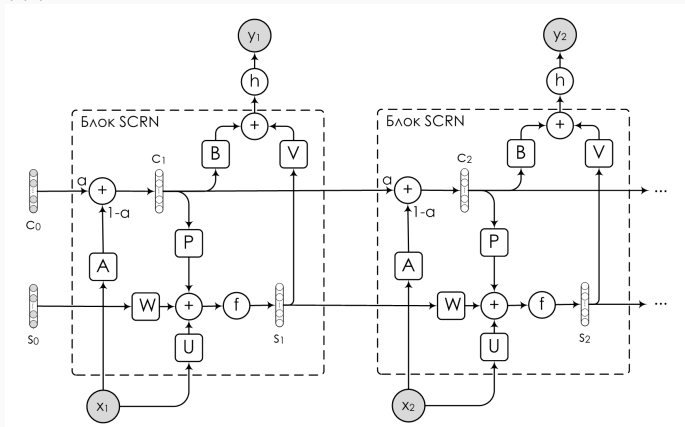
- Разница в том, что нейроны контекста $\mathbf{c}_t$ получают входы со скрытого уровня, а не выходов.
- И нет никаких весов от предыдущих $\mathbf{c}_{t-1}$! То есть веса фиксированы и равны 1.

- Это приводит к хорошим долгосрочным эффектам, потому что нет нелинейности между последовательными шагами, и карусель константной ошибки получается по определению:

$$\mathbf{c}_t = \mathbf{c}_{t-1} + U\mathbf{x}_t.$$

- Идея: можно зафиксировать градиенты, используя единичную матрицу весов вместо обучаемой W .
- Долгосрочная память тут есть... но обучать очень трудно, потому что градиенты надо возвращать к началу последовательности.

- (Mikolov et al., 2014): Structurally Constrained Recurrent Network (SCRN).
- Сочетание двух идей – s_t с W и c_t с диагональной матрицей рекуррентных весов.



- Формально:

$$\mathbf{c}_t = (1 - \alpha) A\mathbf{x}_t + \alpha\mathbf{c}_{t-1},$$

$$\mathbf{s}_t = f(P\mathbf{c}_t + U\mathbf{x}_t + W\mathbf{s}_{t-1}),$$

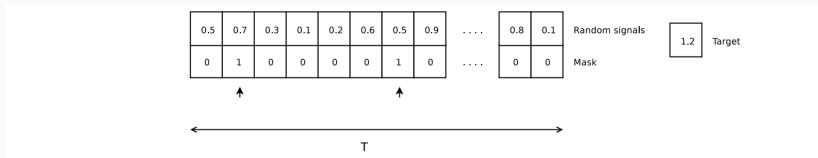
$$\mathbf{y}_t = h(V\mathbf{s}_t + B\mathbf{s}_t).$$

- SCRN – это просто обычный RNN, где $\mathbf{s}_t$ и $\mathbf{c}_t$ в одном векторе, и матрица рекуррентных весов имеет вид

$$W = \begin{pmatrix} R & P \\ 0 & \alpha\mathbf{I} \end{pmatrix},$$

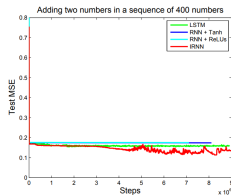
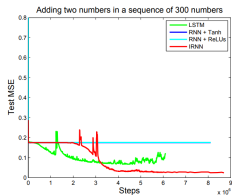
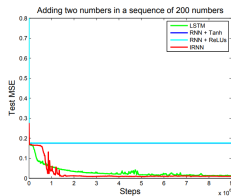
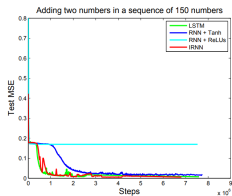
Инициализация RNN с ReLU

- (Le et al., 2015): как правильно инициализировать рекуррентные веса
- IRNN – составим рекуррентные веса с ReLU-активациями и инициализируем единичной матрицей; похоже на SCRNN, но ещё проще
- Пример игрушечной задачи для long-range dependencies:

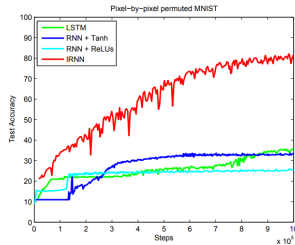
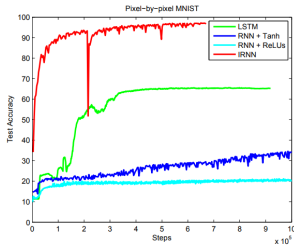


Инициализация RNN с RELU

- И получается хорошо:



- А ещё pixel-by-pixel MNIST:



- Альтернатива: давайте просто регуляризуем W так, чтобы $\det W = 1$.
- Мягкая регуляризация (Pascanu et al., 2013):

$$\Omega = \sum_k \Omega_k = \sum_k \left(\left\| \frac{\frac{\partial E}{\partial \mathbf{s}_{k+1}} \frac{\partial \mathbf{s}_{k+1}}{\partial \mathbf{s}_k}}{\frac{\partial E}{\partial \mathbf{s}_{k+1}}} \right\| - 1 \right)^2.$$

- Жёсткая регуляризация – сделаем W автоматически унитарной (Arjovsky et al., 2015):

$$W = D_3 R_2 F^{-1} D_2 \Pi R_1 F D_1,$$

где D – диагональные матрицы, F – преобразование Фурье, R – отражения, Π – перестановка.

- Кстати, и параметров меньше: теперь только $O(n)$ вместо $O(n^2)$.

- И ещё более простой трюк: давайте правильно инициализируем W (Le, Jaitly, Hinton, 2015).
- Рассмотрим RNN с ReLU-активациями на рекуррентных весах (перед $\mathbf{h}$).
- Тогда если W_{hh} – единичная матрица и $\mathbf{b}_h = 0$, скрытое состояние не изменится, градиент протечёт насквозь.
- Давайте так и инициализируем! Часто приводит к серьёзным улучшениям.

Что делать с RNN на практике

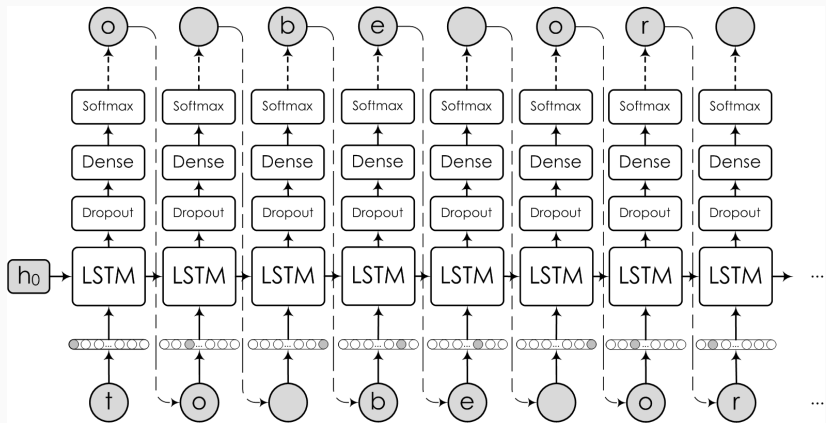
- RNN имеют довольно простую общую структуру: уровни LSTM или GRU.
- Все они выдают последовательность выходов, кроме, возможно, верхнего.
- Дропаут и batchnorm между слоями, а на рекуррентных связях надо аккуратно (потом поговорим).
- Слоёв немного; больше 3-4 трудно, 7-8 сейчас максимум.

- Важный трюк: *skip-layer connections*, как residual, только проще. Добавляем выходы предыдущих слоёв «через один» или «через два», просто конкатенацией.
- RNN, сохраняющие состояние: состояния с одного мини-батча переиспользуются как начальные для следующего. Градиенты в BPTT останавливаются, но состояния остаются.

ПРИМЕР: ПОРОЖДЕНИЕ ТЕКСТА С RNN

- Языковые модели – это естественное прямое приложение к NLP.
- Первая идея – давайте просто обучим последовательность слов через RNN/LSTM.
- О языке будем говорить позже, а пока любопытно, что можно обучить RNN порождать интересные последовательности даже просто символ за символом.
- Karpathy, «The Unreasonable Effectiveness of Neural Networks»; знаменитый пример из (Sutskever et al. 2011):
The meaning of life is the tradition of the ancient human reproduction: it is less favorable to the good boy for when to remove her bigger...
- Это, конечно, всего лишь эффекты краткосрочной памяти, никакого «понимания».

SIMPLE LSTM-BASED ARCHITECTURE



СПАСИБО!

Спасибо за внимание!

