Нейронные сети

Обработка текстов, лекция 6

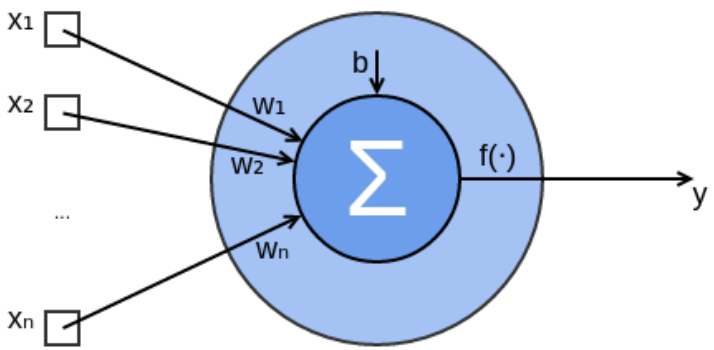
18.10.2017

Константин Архипенко

Содержание

Содержание

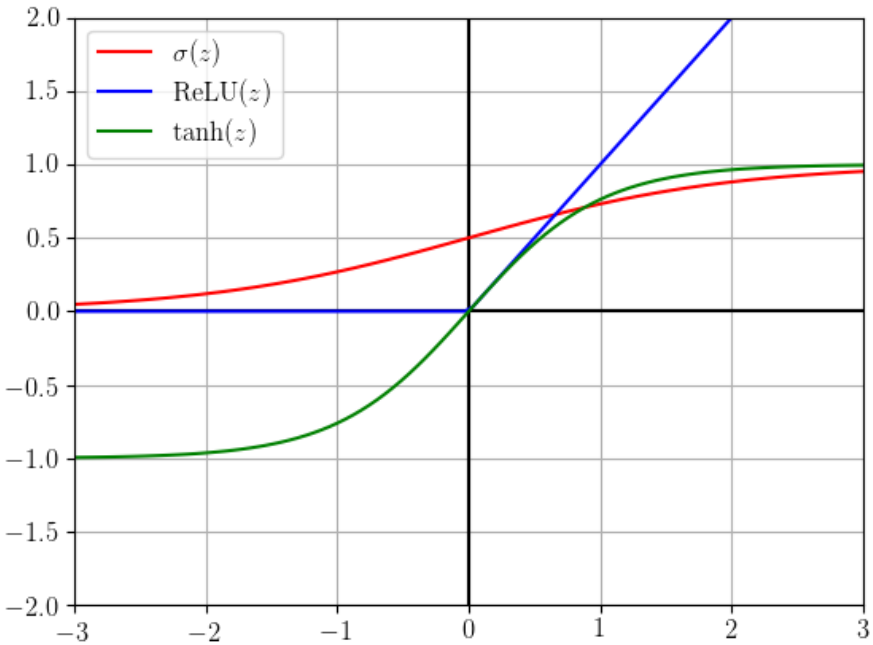
Нейрон



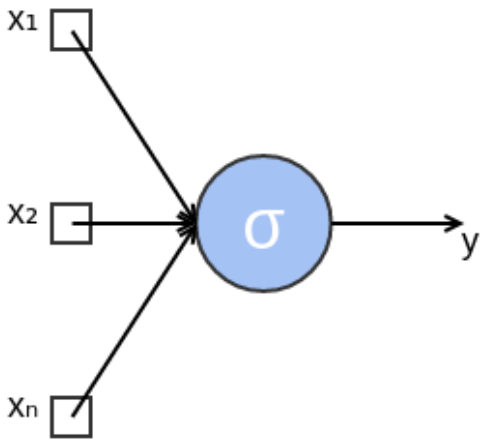
- $y(\boldsymbol{x}) = f(\boldsymbol{x}\boldsymbol{w}^T + b) = f(\sum_i x_i w_i + b)$
- $f(\cdot)$ – функция активации, чаще нелинейная
- Обучение нейрона есть настройка его параметров $\boldsymbol{w}$ и b

Функции активации

- $\sigma(z) = \frac{1}{1+e^{-z}}$
- $\mathrm{ReLU}(z) = \max(0, z)$
- $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\sigma(2z) - 1$
- Бывают и другие

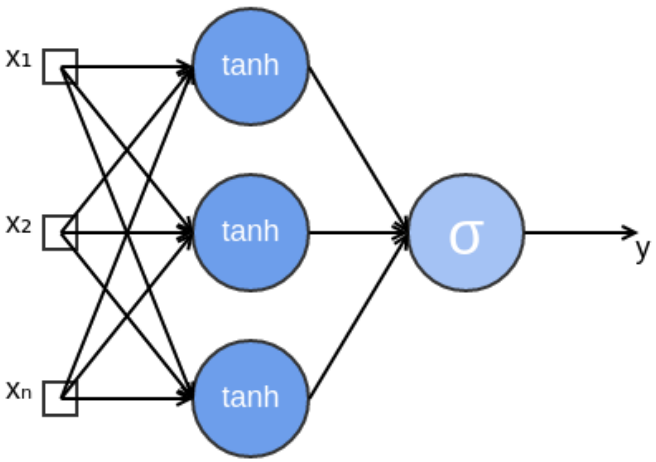


Нейрон с функцией активации $\sigma(z)$



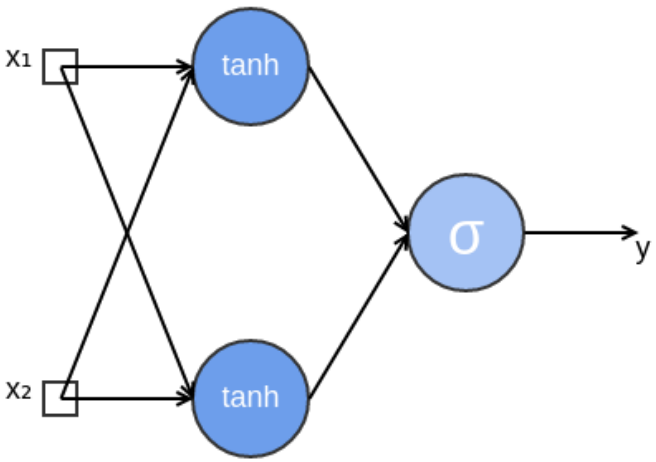
- $y(\boldsymbol{x}) = \sigma(\boldsymbol{x}\boldsymbol{w}^T + b)$
- Получается не что иное, как логистическая регрессия

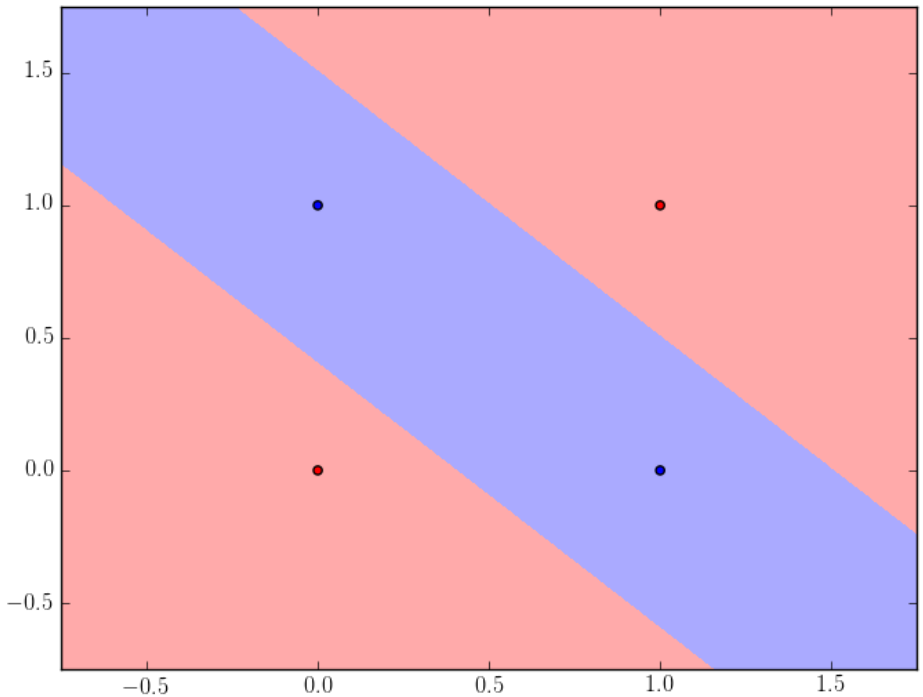
Добавим слой нейронов



- $\boldsymbol{h} = \tanh(\boldsymbol{x}\boldsymbol{W_1} + \boldsymbol{b_1})$
  $y = \sigma(\boldsymbol{h}\boldsymbol{w_2}^T + b_2)$
- Заведомо более мощная модель, чем логистическая регрессия

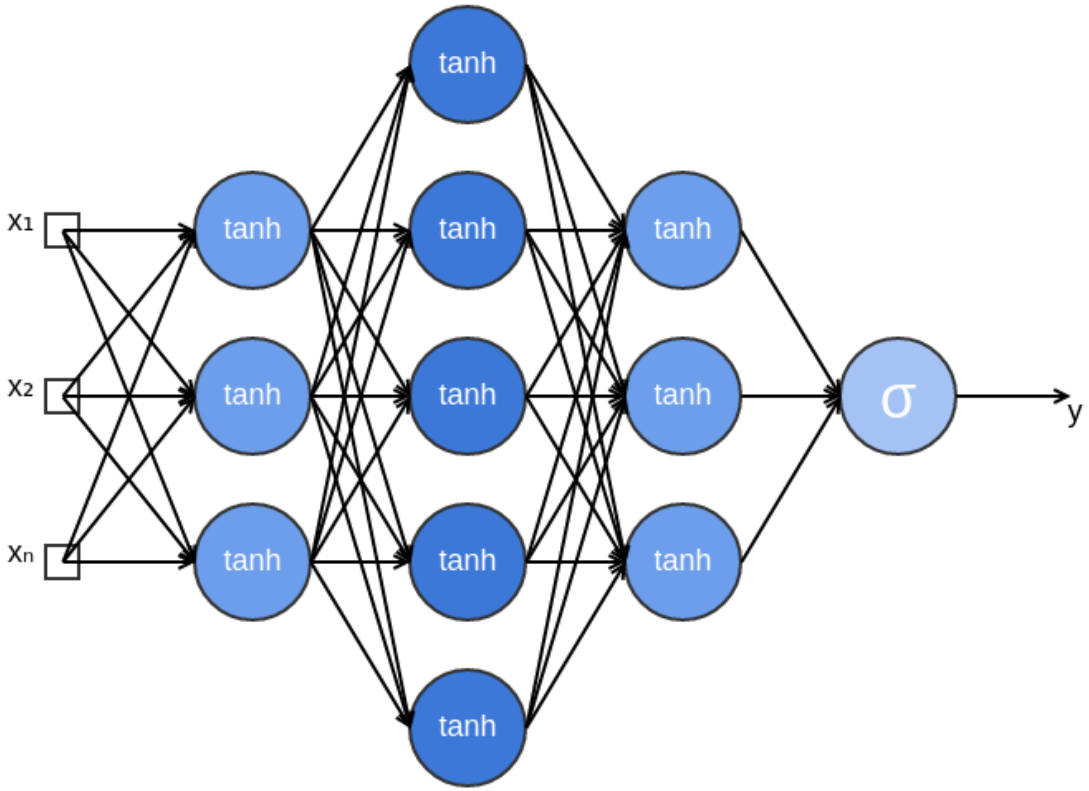
XOR





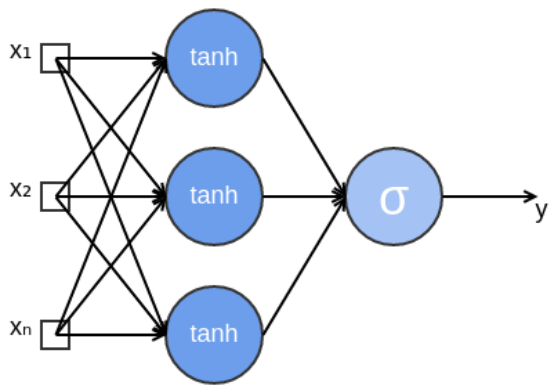
- Линейные классификаторы на такое не способны

Multilayer perceptron (MLP)



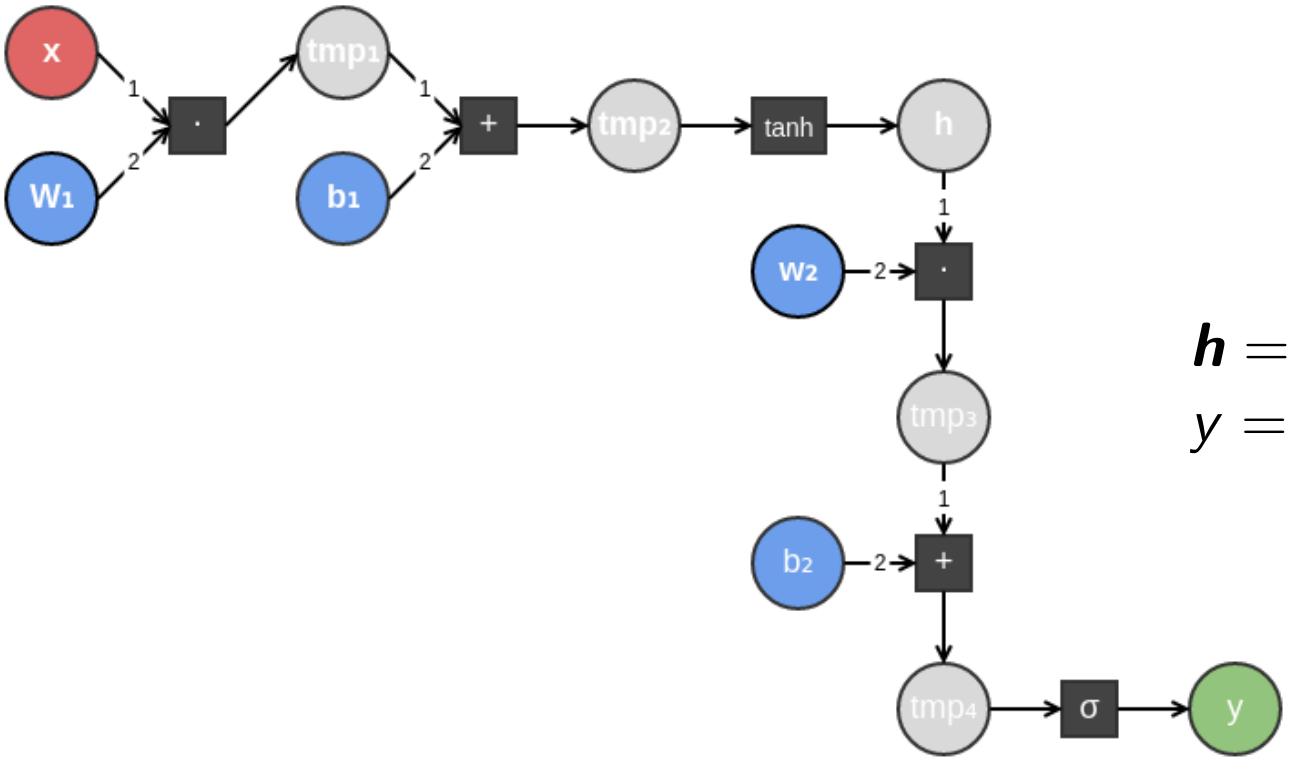
Графы вычислений

- Вместо подобных картинок



для визуализации архитектуры нейросетей ~~можно~~ нужно использовать более компактные и удобные графы вычислений, максимально отражающие факт векторизации

Пример графа вычислений



$$\boldsymbol{h} = \tanh(\boldsymbol{x}\boldsymbol{W_1} + \boldsymbol{b_1})$$
$$y = \sigma(\boldsymbol{h}\boldsymbol{w_2}^T + b_2)$$

- Входные, промежуточные и выходные вершины, операции и настраиваемые параметры

Мультиномиальная логистическая регрессия

- Вероятность класса i:

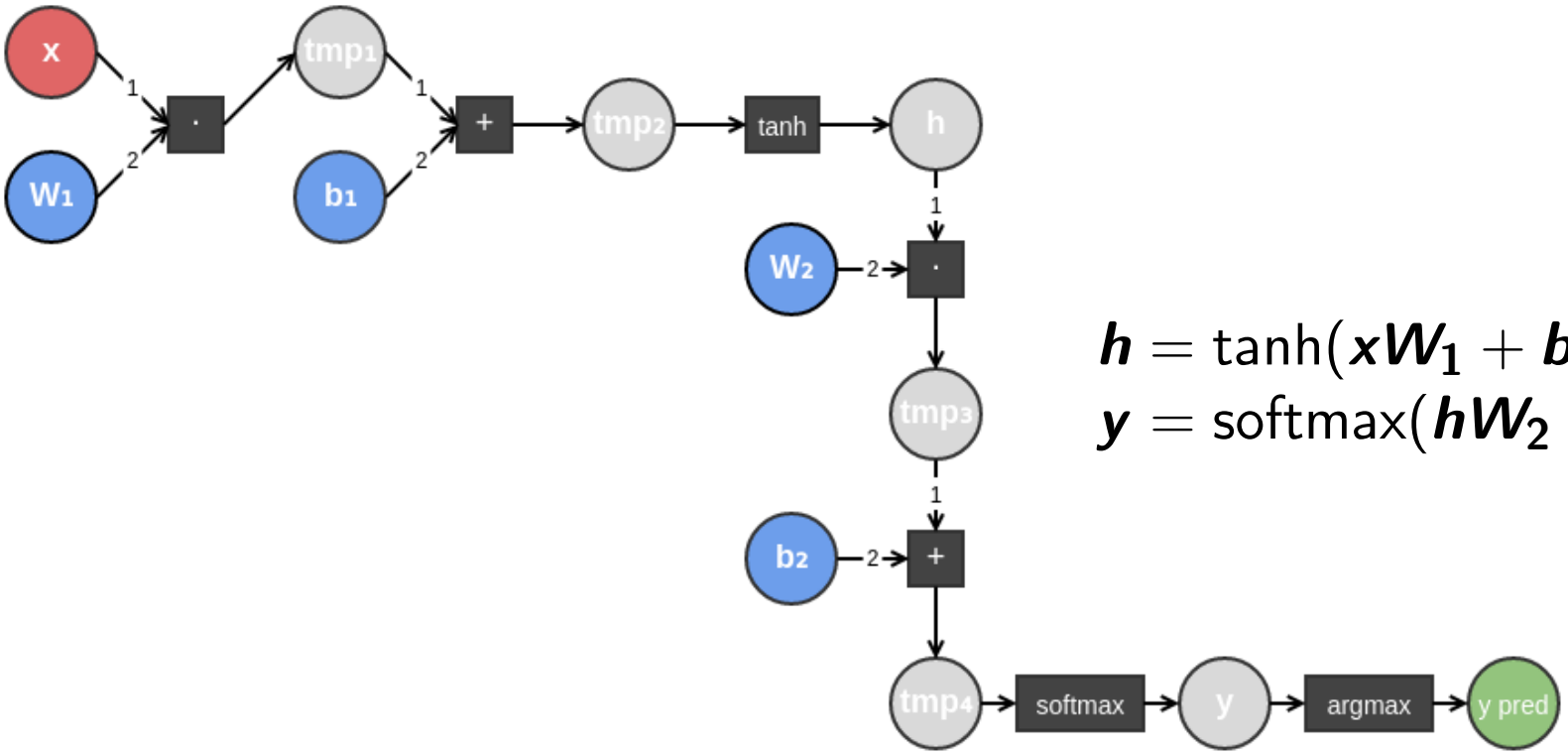
$$\text{softmax}(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

- Аналогичным образом можно усилить модель добавлением слоя:

$$\boldsymbol{h} = \tanh(\boldsymbol{x}\boldsymbol{W_1} + \boldsymbol{b_1})$$

$$\boldsymbol{y} = \text{softmax}(\boldsymbol{h}\boldsymbol{W_2} + \boldsymbol{b_2})$$

Граф вычислений



$$\boldsymbol{h} = \tanh(\boldsymbol{x}\boldsymbol{W_1} + \boldsymbol{b_1})$$
$$\boldsymbol{y} = \mathrm{softmax}(\boldsymbol{h}\boldsymbol{W_2} + \boldsymbol{b_2})$$

- После предсказания вектора $\boldsymbol{y}$ вероятностей классов выбирается наиболее подходящий класс $y_{pred} = \arg\max_i y_i$

Содержание

Обучение логистической регрессии

- Метод максимального правдоподобия:

$$\mathcal{L}(\boldsymbol{X}, \boldsymbol{y_{true}}, \boldsymbol{w}, b) = -\frac{1}{N}\sum_{n=1}^{N} \log P(y_{true}^{[n]} \mid \boldsymbol{x^{[n]}}) + \frac{\lambda}{2}\|\boldsymbol{w}\|^2 \to \min_{\boldsymbol{w}, b}$$

$$P(1 \mid \boldsymbol{x^{[n]}}) = 1 - P(0 \mid \boldsymbol{x^{[n]}}) = y^{[n]} = \sigma(\boldsymbol{x^{[n]}}\boldsymbol{w}^T + b)$$

Бинарная кросс-энтропия

- Перепишем $-\log P(y_{true}^{[n]} \mid \boldsymbol{x}^{[n]})$:

$$-\log P(y_{true}^{[n]} \mid \boldsymbol{x}^{[n]}) = -(y_{true}^{[n]} \log y^{[n]} + (1 - y_{true}^{[n]}) \log(1 - y^{[n]}))$$

- В правой части находится бинарная кросс-энтропия $\text{BCE}(y_{true}^{[n]}, y^{[n]})$
- Таким образом, максимизация правдоподобия есть то же самое, что минимизация кросс-энтропии:

$$\mathcal{L} = \frac{1}{N} \sum_{n=1}^{N} \text{BCE}(y_{true}^{[n]}, y^{[n]}) + \frac{\lambda}{2} \|\boldsymbol{w}\|^2 \to \min_{\boldsymbol{w}, b}$$

$$y^{[n]} = \sigma(\boldsymbol{x}^{[n]} \boldsymbol{w}^T + b)$$

Мультиномиальная логистическая регрессия

- Для многоклассовой классификации бинарная кросс-энтропия заменяется на обычную:

$$\mathcal{L} = \frac{1}{N}\sum_{n=1}^{N} \text{CrossEntropy}(\boldsymbol{y}_{\boldsymbol{true}}^{[n]}, \boldsymbol{y}^{[n]}) + \frac{\lambda}{2}\|\boldsymbol{W}\|_F^2 \to \min_{\boldsymbol{W},\boldsymbol{b}}$$

$$\boldsymbol{y}^{[n]} = \text{softmax}(\boldsymbol{x}^{[n]}\boldsymbol{W} + \boldsymbol{b})$$

$$\text{CrossEntropy}(\boldsymbol{y}_{\boldsymbol{true}}^{[n]}, \boldsymbol{y}^{[n]}) = -\boldsymbol{y}_{\boldsymbol{true}}^{[n]} \log(\boldsymbol{y}^{[n]})^T$$

- Здесь $\boldsymbol{y}_{\boldsymbol{true}}^{[n]}$ – вектор из нулей, кроме единицы в позиции i, соответствующей истинному классу объекта $\boldsymbol{x}^{[n]}$ (one-hot encoding)

Градиентный спуск

- Рассматриваем бинарную логистическую регрессию
- Для минимизации $\mathcal{L}$ есть много методов (включая методы второго порядка), наиболее простым и пригодным для больших данных является градиентный спуск
- На каждой итерации t обновляем параметры $\boldsymbol{w}$ и b ($\alpha_t > 0$):

$$w_i \leftarrow w_i - \alpha_t \nabla_{w_i} \mathcal{L}$$

$$b \leftarrow b - \alpha_t \nabla_b \mathcal{L}$$

- Градиент $\mathcal{L}$ выписывается в явном виде, но найти стационарную точку аналитически не получится

Стохастический градиентный спуск (SGD)

- Вычисление градиента $\mathcal{L}$ на каждой итерации t дорого, поскольку в него входит сумма по всем объектам выборки
- Вместо этого предлагается на каждой итерации брать случайный объект $(\boldsymbol{x}^{[n]}, y^{[n]}_{true})$ выборки и считать градиент потери $\mathcal{L}^{[n]}$ только на этом объекте:

$$\mathcal{L}^{[n]} = \text{BCE}(y^{[n]}_{true}, y^{[n]}) + \frac{\lambda}{2}\|\boldsymbol{w}\|^2$$

$$w_i \leftarrow w_i - \alpha_t \nabla_{w_i} \mathcal{L}^{[n]}$$

$$b \leftarrow b - \alpha_t \nabla_b \mathcal{L}^{[n]}$$

- Есть стратегии уменьшения α_t с теоретическими гарантиями сходимости, но им следуют далеко не всегда

SGD с минибатчами

- Разобьем выборку на минибатчи (группы обычно одинакового размера), а затем будем на каждой итерации брать минибатч b и считать градиент потери $\mathcal{L}^{[b]}$:

$$\mathcal{L}^{[b]} = \frac{1}{|b|} \sum_{n \in b} \text{BCE}(y_{true}^{[n]}, y^{[n]}) + \frac{\lambda}{2} \|\boldsymbol{w}\|^2$$

- По сравнению с простым SGD появляется возможность распараллеливания вычислений
- Для обучения нейронных сетей на GPU минибатчи используются почти всегда

Обучение нейронной сети

- Рассмотрим подробно обучение нейронной сети для бинарной классификации:

$$\boldsymbol{h} = \tanh(\boldsymbol{x}\boldsymbol{W_1} + \boldsymbol{b_1})$$
$$y = \sigma(\boldsymbol{h}\boldsymbol{w_2}^T + b_2)$$

- Используем бинарную кросс-энтропию в качестве функции потерь и SGD
- Опустим для простоты L_2-регуляризацию (но она очень желательна!) и пока забудем о минибатчах
- Потребуется ввести понятие якобиана и вспомнить цепное правило

Якобиан

- Пусть $\boldsymbol{v} = f(\boldsymbol{u})$. Тогда $\nabla_{\boldsymbol{u}} \boldsymbol{v}$ – матрица:

$$(\nabla_{\boldsymbol{u}} \boldsymbol{v})_{ij} = \nabla_{u_j} v_i$$

- Обобщим понятие якобиана

Якобиан

- Пусть $f : \mathbb{R}^m \to \mathbb{R}$, $v = f(\boldsymbol{u})$. Тогда $\nabla_{\boldsymbol{u}} v$ – вектор:

$$(\nabla_{\boldsymbol{u}} v)_i = \nabla_{u_i} v$$

- Пусть $f : \mathbb{R}^{m \times n} \to \mathbb{R}$, $v = f(\boldsymbol{U})$. Тогда $\nabla_{\boldsymbol{U}} v$ – матрица:

$$(\nabla_{\boldsymbol{U}} v)_{ij} = \nabla_{u_{ij}} v$$

- Пусть $f : \mathbb{R}^{m \times n} \to \mathbb{R}^n$, $\boldsymbol{v} = f(\boldsymbol{U})$. Тогда $\nabla_{\boldsymbol{U}} \boldsymbol{v}$ – 3-мерный тензор:

$$(\nabla_{\boldsymbol{U}} \boldsymbol{v})_{ijk} = \nabla_{u_{jk}} v_i$$

Цепное правило

- Если $w = g(v)$, $v = f(u)$, а функции дифференцируемы, то

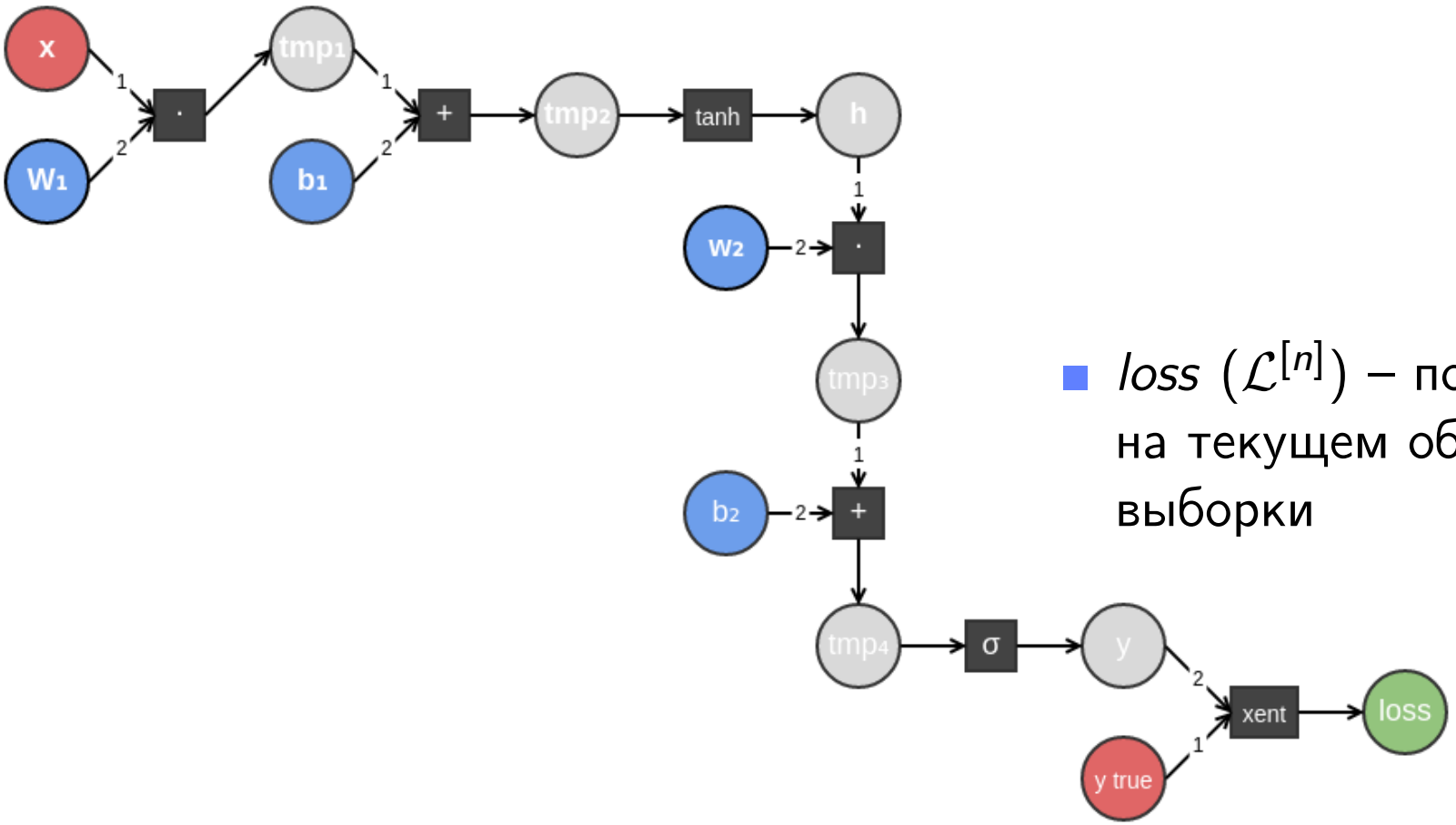
$$\nabla_u w = \nabla_v w \nabla_u v$$

- Справедливо и для наших якобианов
- Например, если $w = g(\boldsymbol{v})$, $\boldsymbol{v} = f(\boldsymbol{U})$, то

$$\nabla_{\boldsymbol{U}} w = \nabla_{\boldsymbol{v}} w \nabla_{\boldsymbol{U}} \boldsymbol{v}$$

- В последнем равенстве вектор-строка умножается на 3-мерный тензор – получается матрица

Граф вычислений нейронной сети



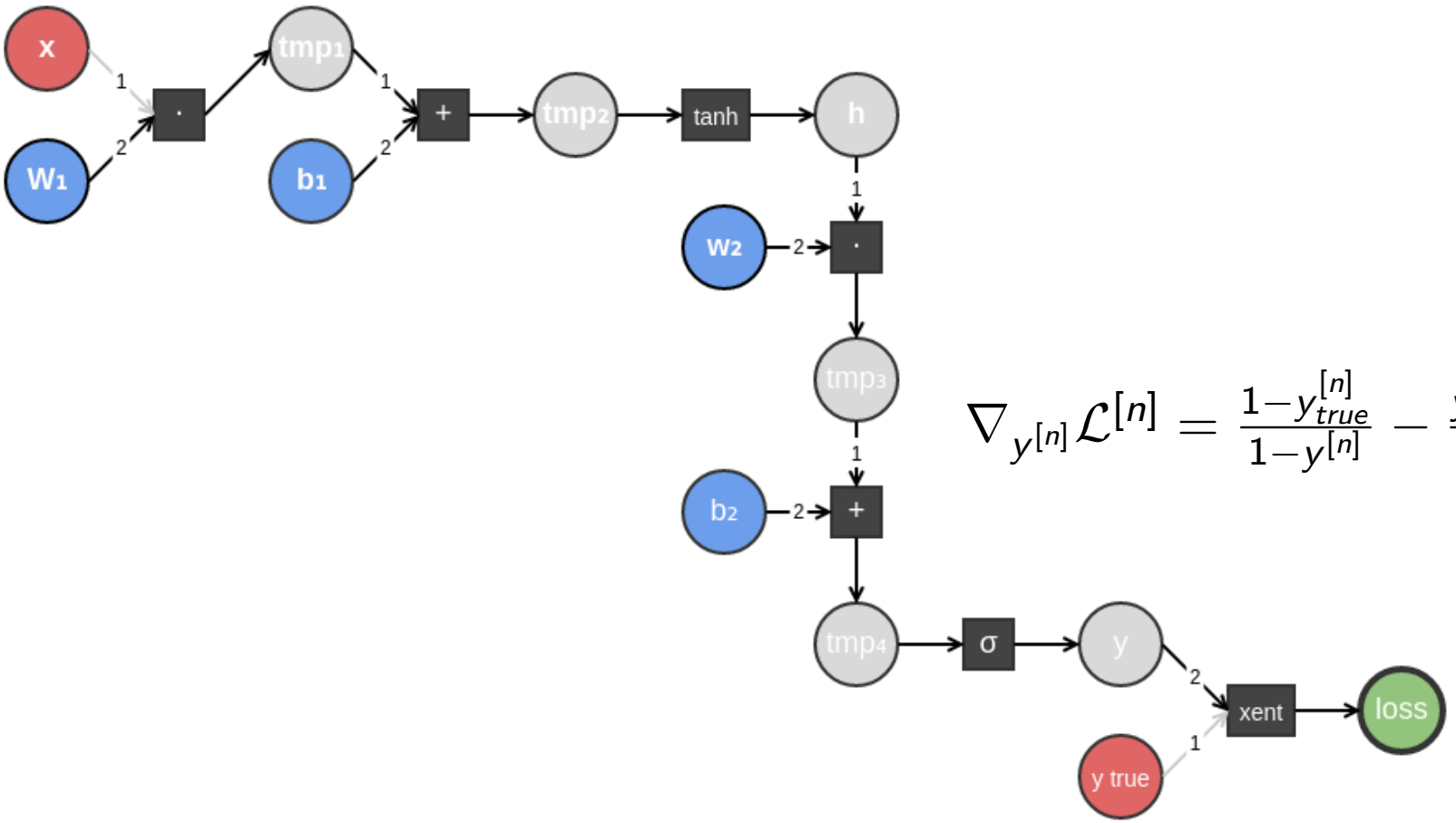
- *loss* ($\mathcal{L}^{[n]}$) – потеря на текущем объекте выборки

Backpropagation

- Пусть мы вычислили значения вершин от входных к выходным на текущем объекте выборки
- Наша цель – вычислить градиент $\mathcal{L}^{[n]}$ по всем настраиваемым параметрам нейросети, чтобы затем сделать шаг SGD
- Для этого будем идти назад от вершины *loss* к вершинам-параметрам, вычисляя по цепному правилу градиенты $\mathcal{L}^{[n]}$ по круглым вершинам на нашем пути

Backpropagation

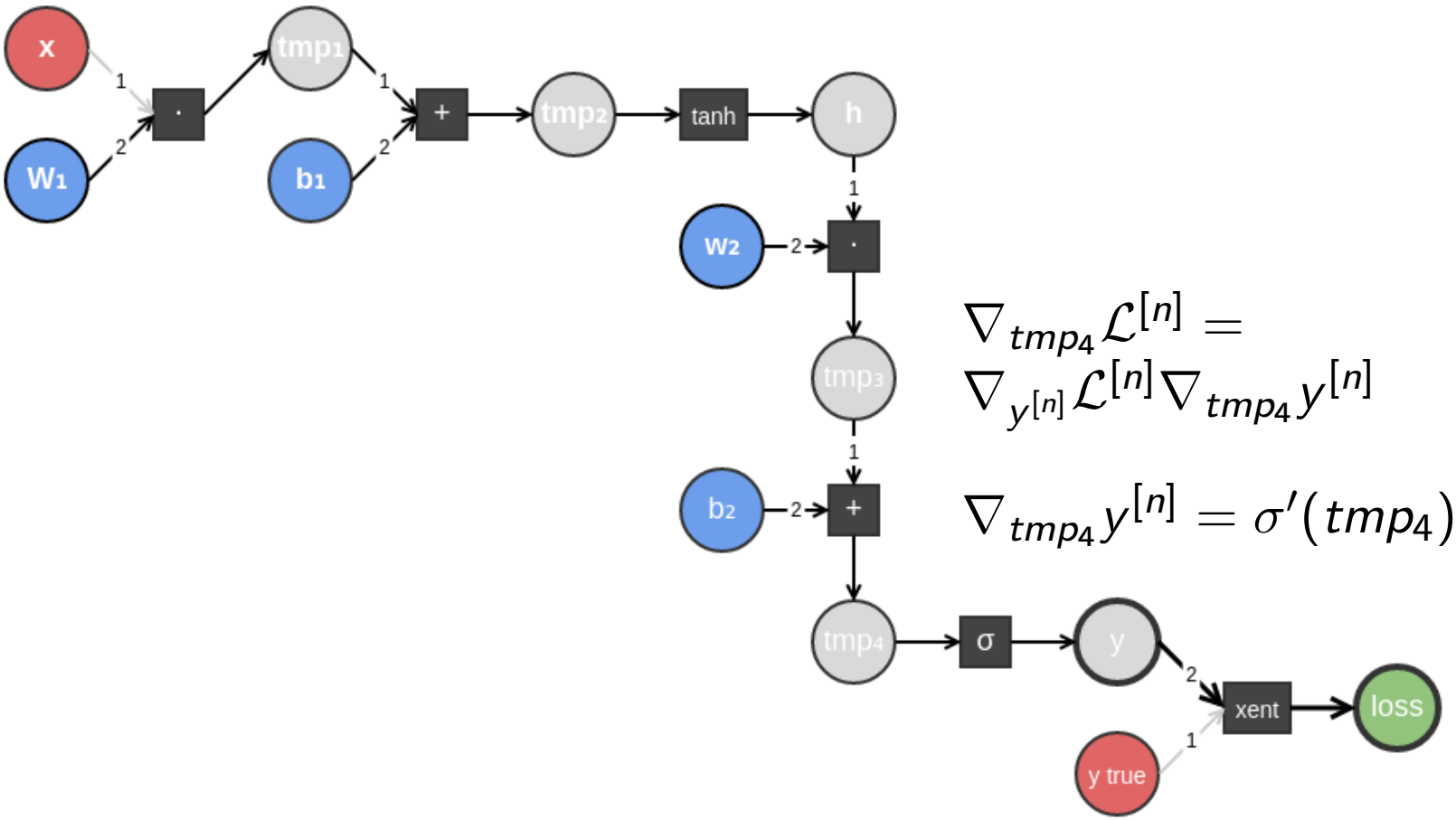
- $\mathcal{L}^{[n]} = -(y_{true}^{[n]} \log y^{[n]} + (1 - y_{true}^{[n]}) \log(1 - y^{[n]}))$



$$\nabla_{y^{[n]}} \mathcal{L}^{[n]} = \frac{1 - y_{true}^{[n]}}{1 - y^{[n]}} - \frac{y_{true}^{[n]}}{y^{[n]}}$$

Backpropagation

- $y = \sigma(tmp_4)$

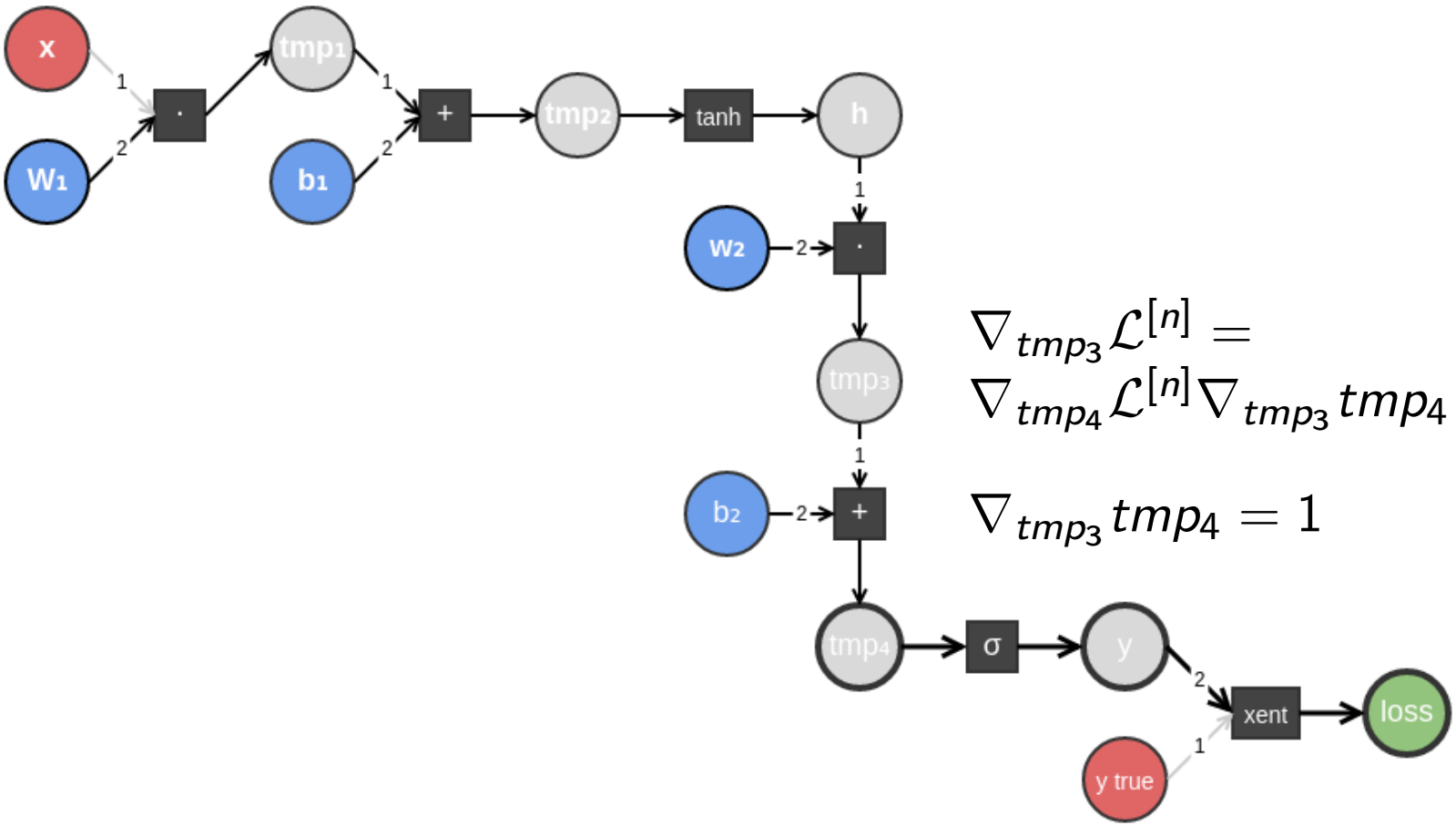


$$\nabla_{tmp_4} \mathcal{L}^{[n]} = \nabla_{y^{[n]}} \mathcal{L}^{[n]} \nabla_{tmp_4} y^{[n]}$$

$$\nabla_{tmp_4} y^{[n]} = \sigma'(tmp_4)$$

Backpropagation

- $tmp_4 = tmp_3 + b_2$

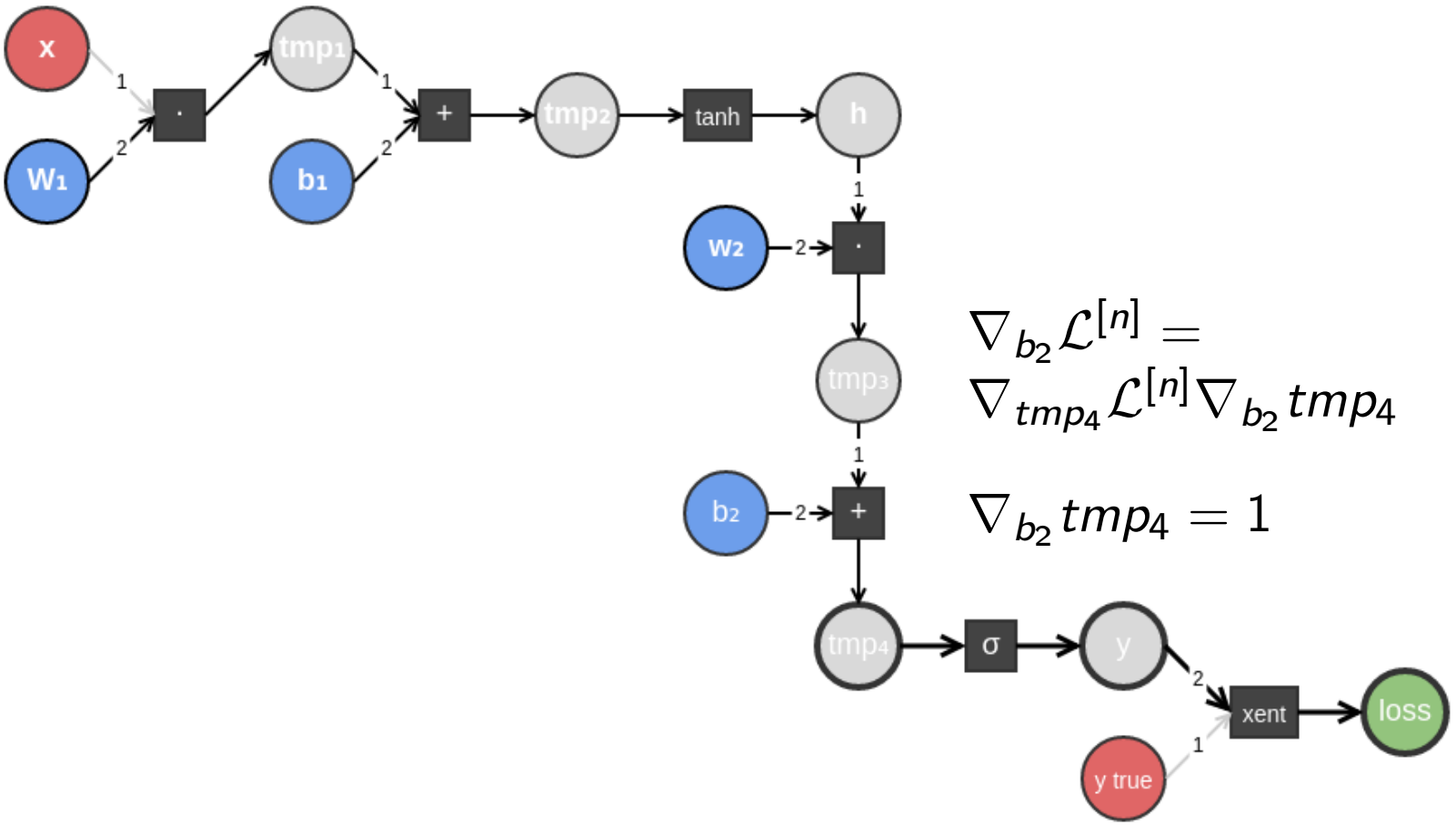


$$\nabla_{tmp_3} \mathcal{L}^{[n]} = \nabla_{tmp_4} \mathcal{L}^{[n]} \nabla_{tmp_3} tmp_4$$

$$\nabla_{tmp_3} tmp_4 = 1$$

Backpropagation

- $tmp_4 = tmp_3 + b_2$

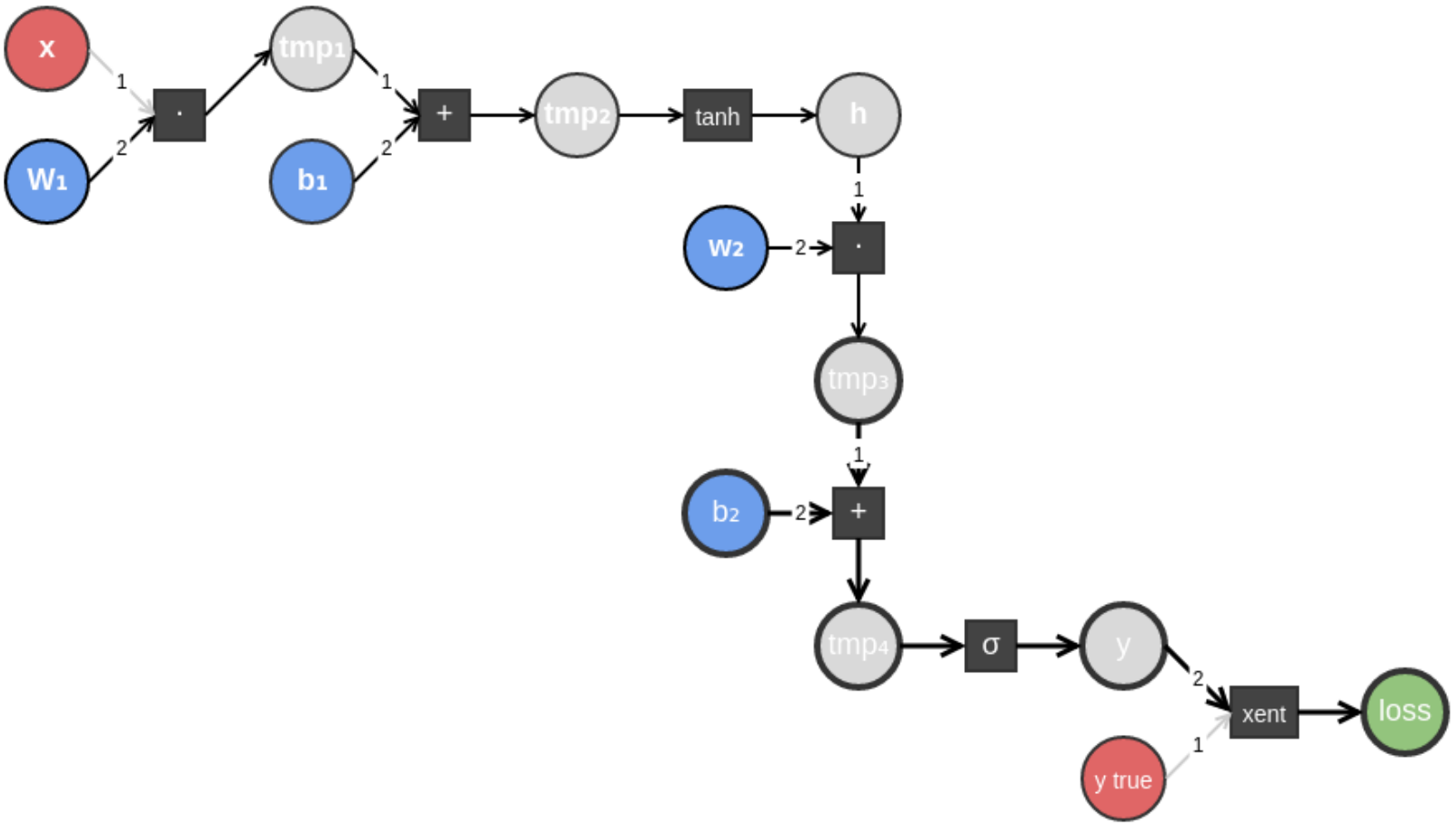


$$\nabla_{b_2} \mathcal{L}^{[n]} = \nabla_{tmp_4} \mathcal{L}^{[n]} \nabla_{b_2} tmp_4$$

$$\nabla_{b_2} tmp_4 = 1$$

Backpropagation

- И так далее



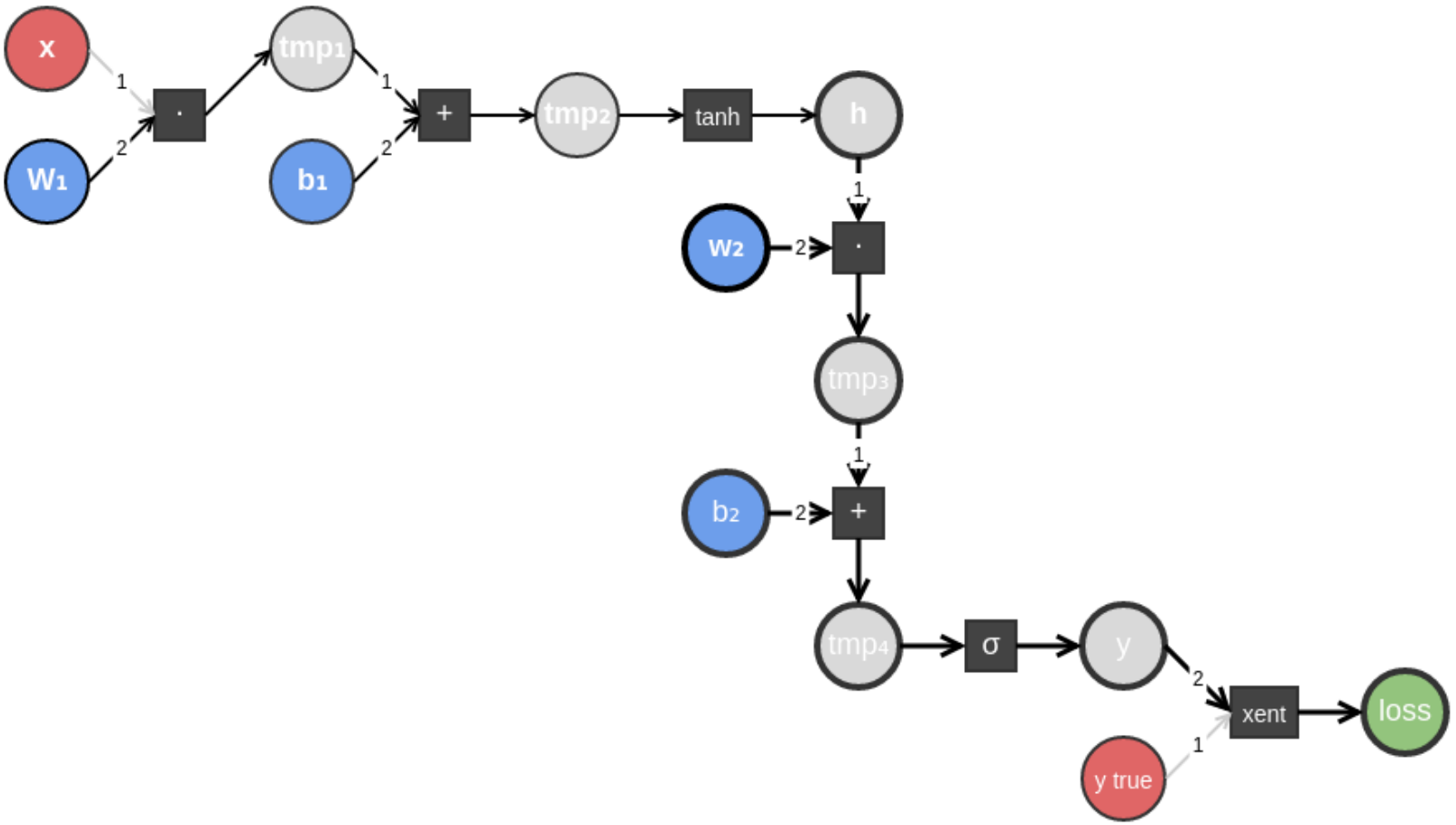
Автоматическое дифференцирование

- Вершина tmp_3 знает градиент $\nabla_{tmp_3}\mathcal{L}^{[n]}$ и отправляет его операции-предку
- Прямоугольник-операция, получив градиент по tmp_3, умножает его на свои якобианы $\nabla_{\boldsymbol{h}} tmp_3$ и $\nabla_{\boldsymbol{w_2}} tmp_3$ (подставляя текущие значения вершин) и отправляет результаты вершинам $\boldsymbol{h}$ и $\boldsymbol{w_2}$ соответственно:

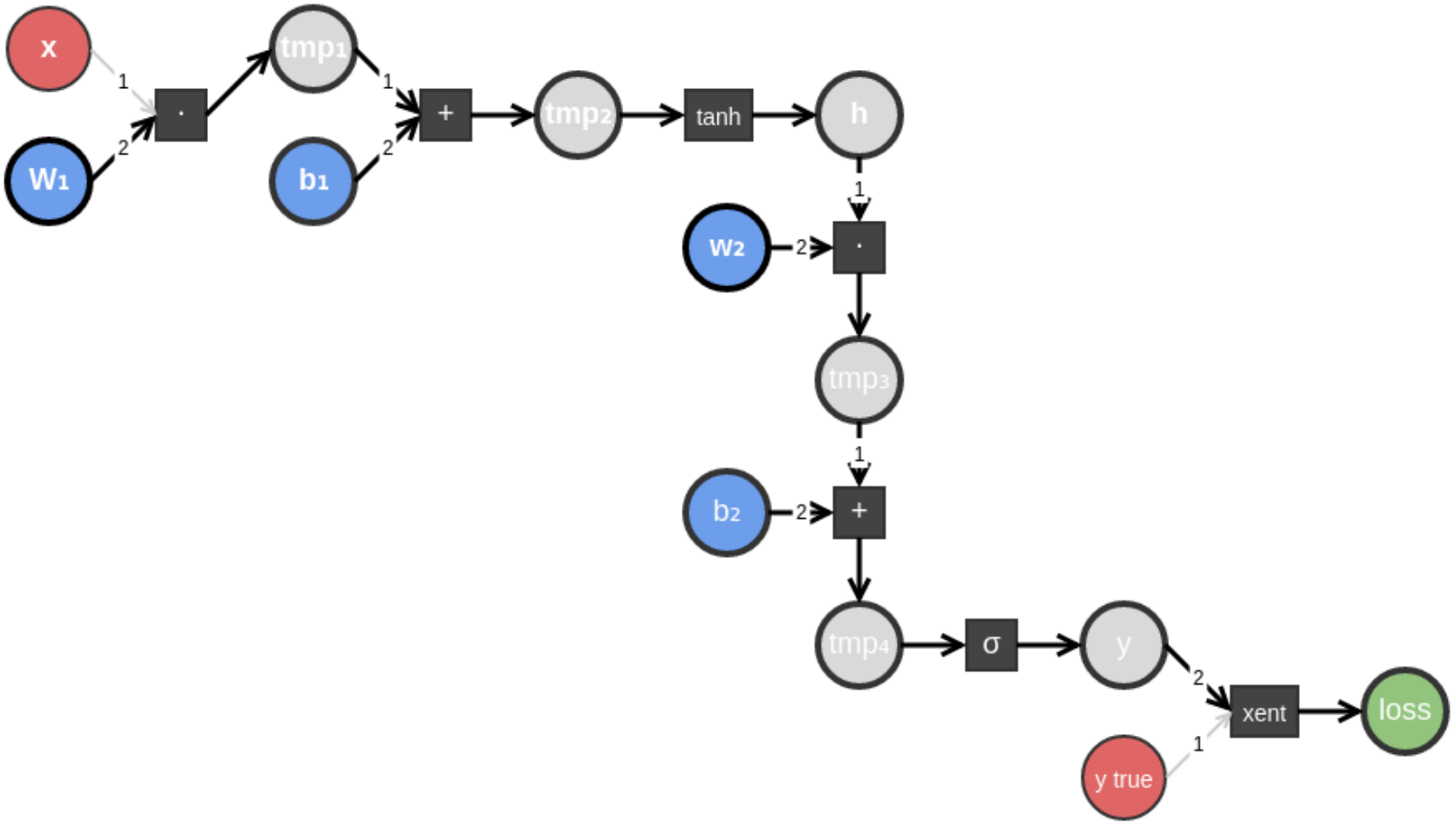
$$\nabla_{\boldsymbol{h}}\mathcal{L}^{[n]} = \nabla_{tmp_3}\mathcal{L}^{[n]}\nabla_{\boldsymbol{h}} tmp_3 = \nabla_{tmp_3}\mathcal{L}^{[n]}\boldsymbol{w_2}$$

$$\nabla_{\boldsymbol{w_2}}\mathcal{L}^{[n]} = \nabla_{tmp_3}\mathcal{L}^{[n]}\nabla_{\boldsymbol{w_2}} tmp_3 = \nabla_{tmp_3}\mathcal{L}^{[n]}\boldsymbol{h}$$

Backpropagation



Backpropagation



Минибатчи

- На входе теперь не вектор $\boldsymbol{x}^{[n]}$, а матрица $\boldsymbol{X}^{[b]}$ с равным размеру минибатча количеством строк
- $\boldsymbol{tmp_1}$, $\boldsymbol{tmp_2}$, $\boldsymbol{h}$ превращаются в матрицы
- tmp_3, tmp_4 – в векторы
- $y^{[n]}$ и $y^{[n]}_{true}$ – в векторы $\boldsymbol{y}^{[b]}$ и $\boldsymbol{y}^{[b]}_{\boldsymbol{true}}$
- К графу добавляется новая выходная вершина – усредненная потеря $\mathcal{L}^{[b]}$ на минибатче
- В итоге распараллеливается не только предсказание нейросети, но и backpropagation
- Минибатчи используются и на тестовых данных после обучения

Общая схема обучения нейросети

- Параметры нейросети инициализируются случайными малыми значениями (например, Glorot uniform[1]), смещения $\boldsymbol{b}$ – обычно нулями
- Каждую эпоху обучающая выборка случайным образом разбивается на минибатчи
- Для каждого минибатча вычисляют градиент потери и обновляют параметры
- Обучение заканчивается по достижении определенного числа эпох или при прекращении снижения функции потерь на валидационной выборке

[1] Glorot & Bengio "Understanding the difficulty of training deep feedforward neural networks" (2010), `http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf`

Оптимизация функции потерь

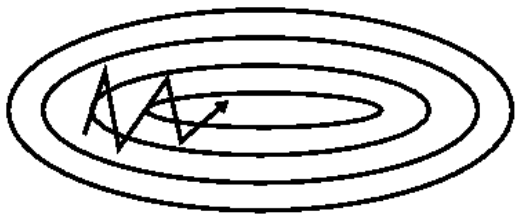
- В отличие от логистической регрессии, обучение нейросети есть оптимизация очень сложной функции с большим количеством локальных минимумов
- Применять SGD становится трудно
- Рассмотрим улучшения SGD для более быстрого нахождения более оптимальных параметров

Momentum

- Пусть $\boldsymbol{\theta}$ – совокупность настраиваемых параметров нейросети
- $\boldsymbol{v_t} = \gamma \boldsymbol{v_{t-1}} - \alpha_t \nabla_{\boldsymbol{\theta}} \mathcal{L}^{[b]}\big|_{\boldsymbol{\theta}=\boldsymbol{\theta_{t-1}}}$
  $\boldsymbol{\theta_t} = \boldsymbol{\theta_{t-1}} + \boldsymbol{v_t}$
- По функции потерь катится мяч, ускоряясь на спусках
- Чуть более умная версия мяча (Nesterov accelerated gradient):
  $\boldsymbol{v_t} = \gamma \boldsymbol{v_{t-1}} - \alpha_t \nabla_{\boldsymbol{\theta}} \mathcal{L}^{[b]}\big|_{\boldsymbol{\theta}=\boldsymbol{\theta_{t-1}}+\gamma \boldsymbol{v_{t-1}}}$

- SGD

- SGD + momentum



Sebastian Ruder “An overview of gradient descent optimization algorithms”, http://ruder.io/optimizing-gradient-descent

RMSprop

- Хотелось бы, чтобы скорость $\boldsymbol{v}$ для каждого из направлений была примерно одинаковой
- Решение $\boldsymbol{v_t} = -\alpha_t \frac{\nabla_{\boldsymbol{\theta}} \mathcal{L}^{[b]}}{|\nabla_{\boldsymbol{\theta}} \mathcal{L}^{[b]}| + \varepsilon}\Big|_{\boldsymbol{\theta}=\boldsymbol{\theta_{t-1}}}$ не очень подходит: представим два минибатча с противоположно направленными градиентами разной величины
- Возьмем скользящее среднее:

$$\boldsymbol{a_t} = 0.9\boldsymbol{a_{t-1}} + 0.1|\nabla_{\boldsymbol{\theta}} \mathcal{L}^{[b]}\big|_{\boldsymbol{\theta}=\boldsymbol{\theta_{t-1}}}| \qquad \boldsymbol{v_t} = -\alpha_t \frac{\nabla_{\boldsymbol{\theta}} \mathcal{L}^{[b]}}{\boldsymbol{a_t} + \varepsilon}\Big|_{\boldsymbol{\theta}=\boldsymbol{\theta_{t-1}}}$$

- Еще лучше: корень (R) из скользящего среднего (M) квадратов (S):

$$\boldsymbol{a_t} = 0.9\boldsymbol{a_{t-1}} + 0.1(\nabla_{\boldsymbol{\theta}} \mathcal{L}^{[b]}\big|_{\boldsymbol{\theta}=\boldsymbol{\theta_{t-1}}})^2 \qquad \boldsymbol{v_t} = -\alpha_t \frac{\nabla_{\boldsymbol{\theta}} \mathcal{L}^{[b]}}{\sqrt{\boldsymbol{a_t}} + \varepsilon}\Big|_{\boldsymbol{\theta}=\boldsymbol{\theta_{t-1}}}$$

Лекция Geoffrey Hinton: `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

RMSprop

- Использование корня из среднего квадратов имеет связь с кривизной функции потерь по каждому из направлений
- Проблема разной величины градиентов по разным направлениям решается изящнее, скорость не становится слишком большой
- Небольшой недостаток: для хранения скользящего среднего квадратов нужна дополнительная память (на GPU)
- Если заменить градиент в числителе на его скользящее среднее, получим Adam[2]. Это не RMSprop + momentum

[2] Kingma & Ba "Adam: A Method for Stochastic Optimization" (2014), https://arxiv.org/abs/1412.6980

Проблема выбора α_t

- Частично решается методами RMSprop и Adam (SGD + momentum – в меньшей степени)
- Некоторые исследователи[3] рекомендуют обучать нейросеть некоторое количество эпох (фиксированное или до прекращения снижения значения функции потерь) с фиксированным α_t, а затем многократно уменьшать (до 10 раз)

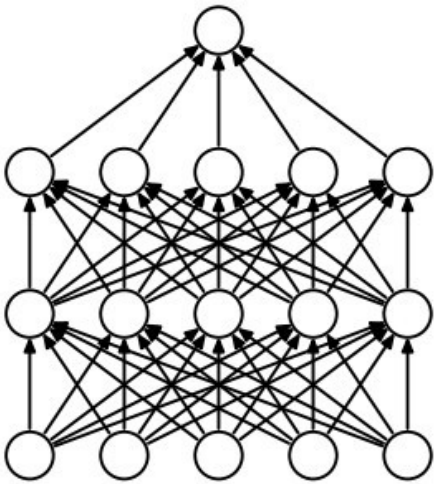
[3] He et al. "Deep Residual Learning for Image Recognition" (2015), https://arxiv.org/abs/1512.03385

Dropout

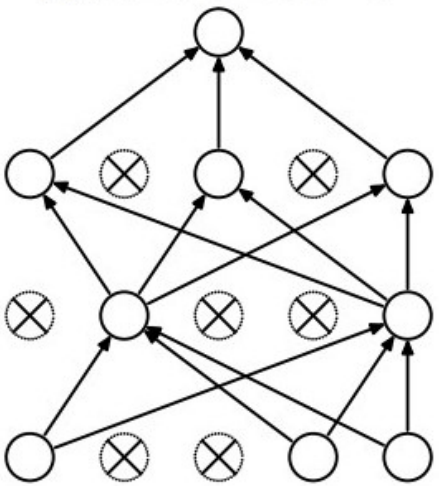
- Каждый элемент тензора обнуляется с вероятностью p либо умножается на $\frac{1}{p}$ с вероятностью $1 - p$
- Пример:

$$\tilde{\boldsymbol{h}} = \tanh(\boldsymbol{x}\boldsymbol{W_1} + \boldsymbol{b_1})$$
$$\boldsymbol{h} = \text{dropout}(\tilde{\boldsymbol{h}})$$
$$y = \sigma(\boldsymbol{h}\boldsymbol{w_2}^T + b_2)$$

- Более эффективным обобщением является Gaussian dropout (с умножением на нормальный шум с матожиданием 1 вместо Бернулли)



(a) Standard Neural Net



(b) After applying dropout.

Dropout

- Помогает от переобучения
- При выводе (работе на тестовых данных) dropout выключают
- Сейчас на практике чаще используют различные виды нормализации (batch/layer/weight) и другие изощренные способы регуляризации (zoneout[4] и т. п.)

[4] Krueger et al. "Zoneout: Regularizing RNNs by Randomly Preserving Hidden Activations" (2016), https://arxiv.org/abs/1606.01305

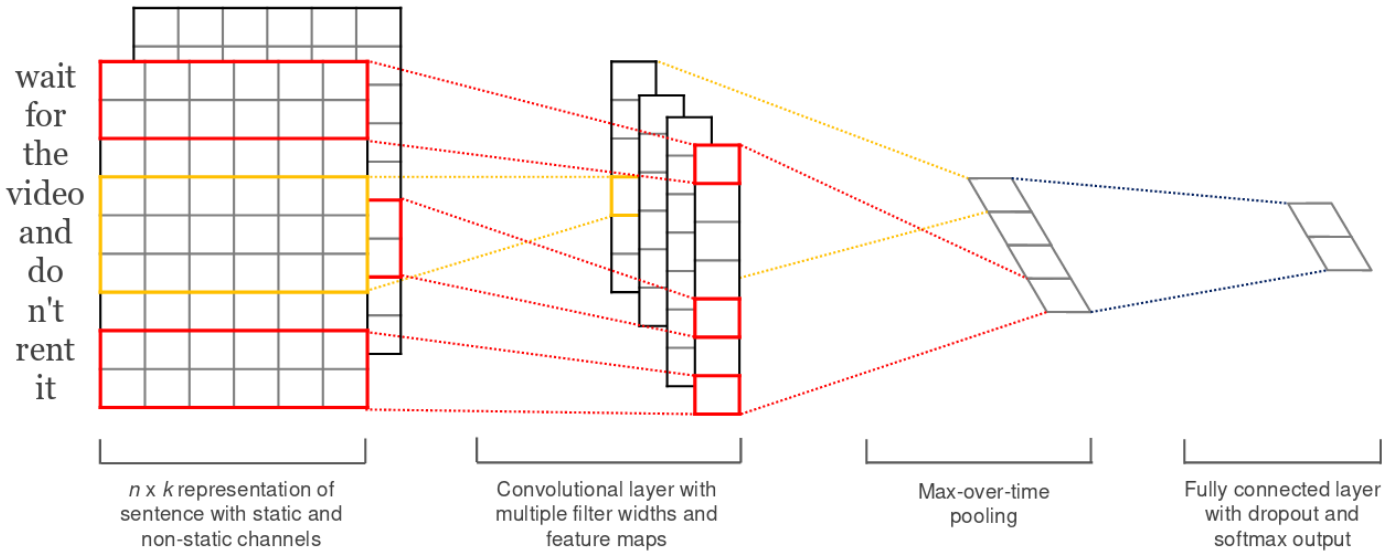
Содержание

Сверточные нейронные сети

- Изначально использовались в задачах компьютерного зрения и широко используются сейчас (например, многочисленные варианты ResNet)
- Иногда помогают и при обработке текстов
- Рассмотрим одно из возможных применений сверточных сетей для классификации коротких текстов (предложений)[5]

[5] Kim "Convolutional Neural Networks for Sentence Classification" (2014), http://www.aclweb.org/anthology/D14-1181

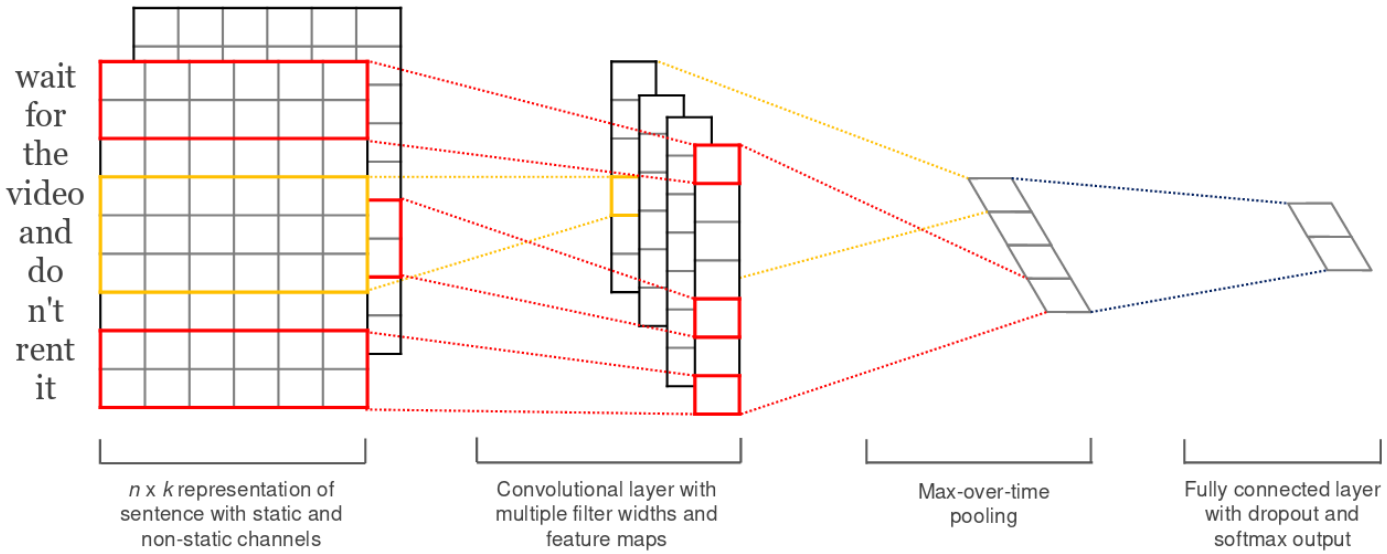
Классификация текстов с помощью CNN



- На вход подается матрица векторных представлений (например, word2vec[6], где векторные представления могут строиться автоматически по большому неразмеченному корпусу текстов) слов предложения

[6] Mikolov et al. "Distributed Representations of Words and Phrases and their Compositionality" (2013), https://arxiv.org/abs/1310.4546

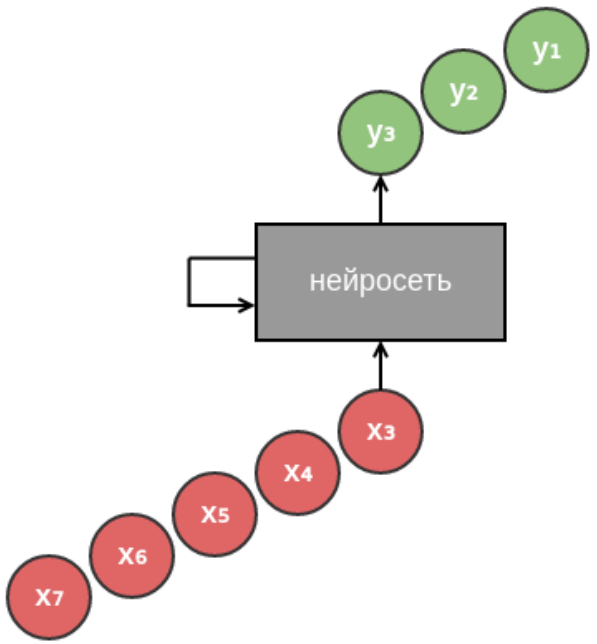
Классификация текстов с помощью CNN



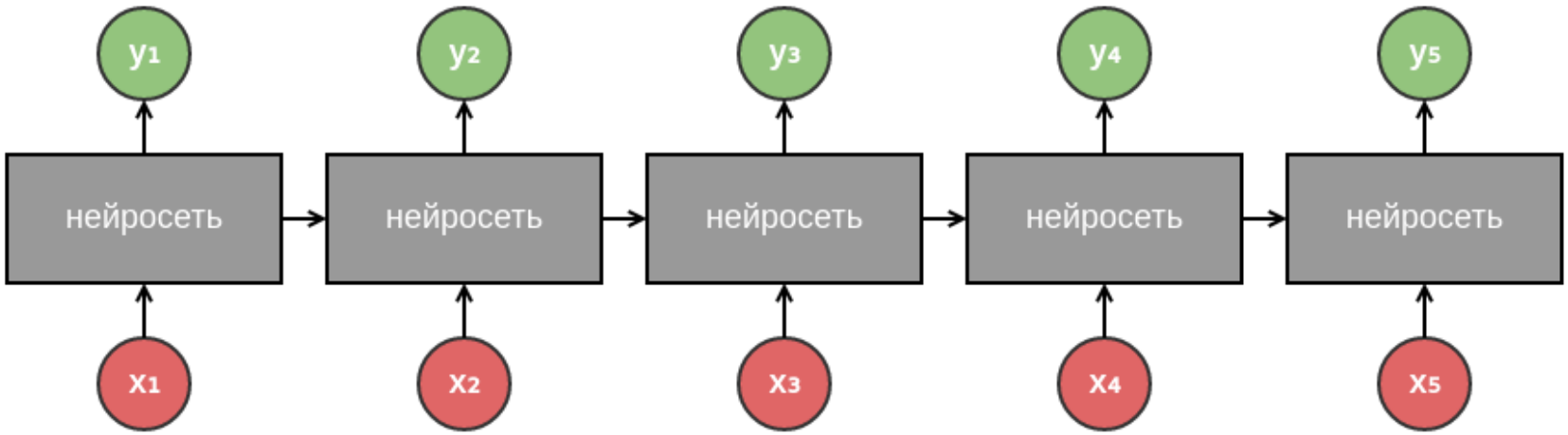
- Свертка – скалярное умножение фрагмента входа на обучаемый (тем же backpropagation) фильтр
- Используются фильтры с разной шириной по словам, pooling осуществляется выбором максимального значения

Рекуррентные нейронные сети: мотивация

- Обработка последовательных данных: предложений на естественном языке, временных рядов и т. п.
- Различная длина последовательностей
- Взаимосвязи между элементами последовательностей, возможно, далеко отстоящими
- Рекуррентная нейросеть (RNN) аккумулирует информацию о предыдущих элементах подаваемой на вход последовательности

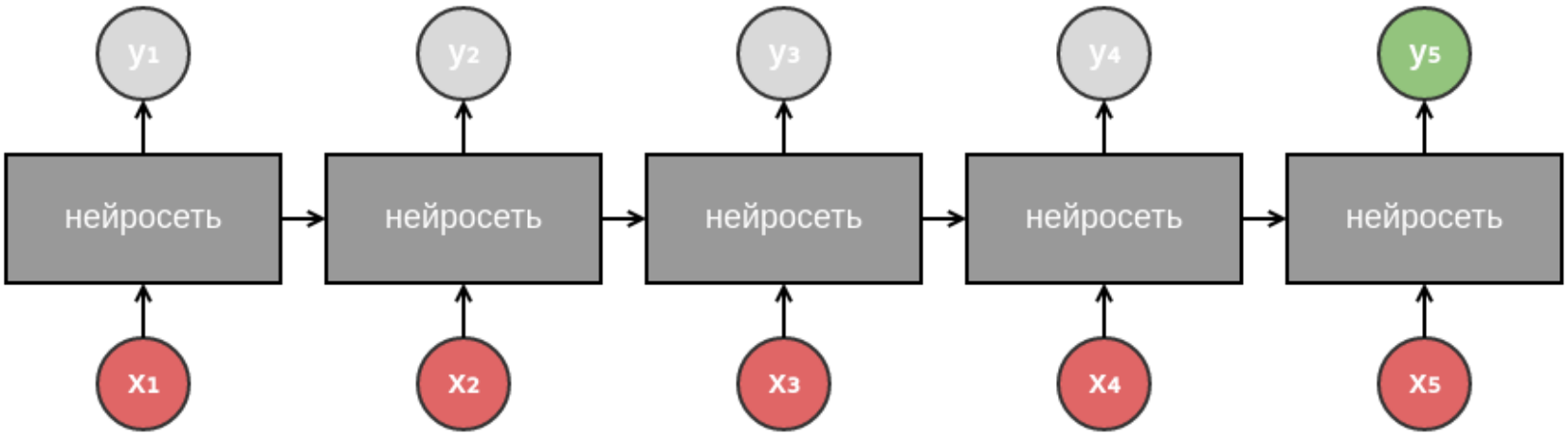


RNN, представленная в другом виде



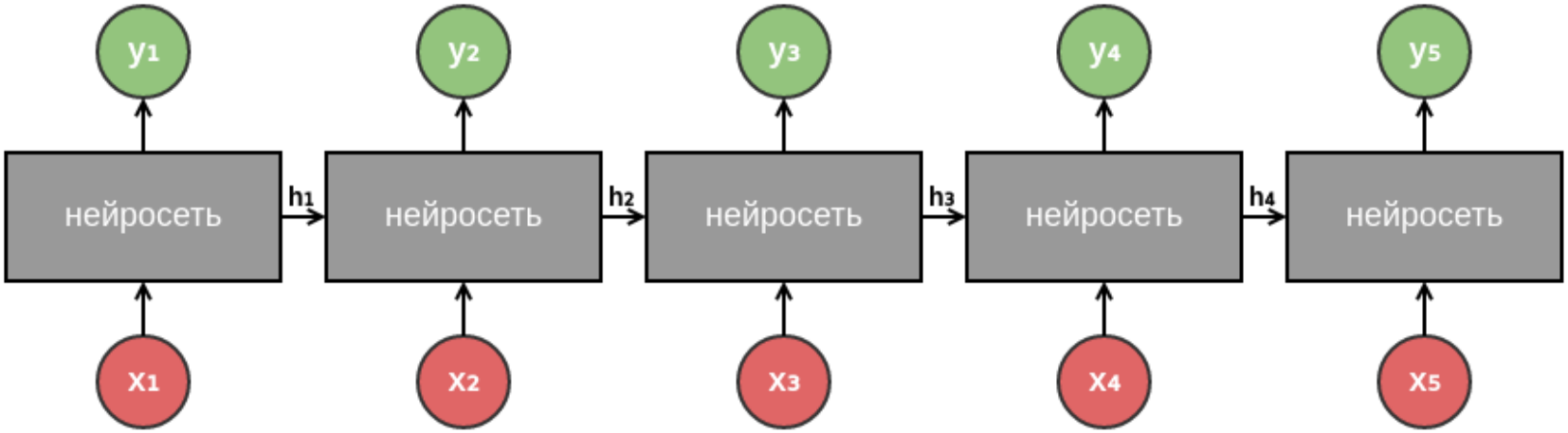
- Можно использовать как все элементы последовательности выходов (например, в задаче определения частей речи)...

RNN, представленная в другом виде



- так и только последний, игнорируя и не вычисляя градиент потери по остальным (например, в задаче классификации предложений)

Простая RNN[7]



- $\boldsymbol{h_t} = f_h(\boldsymbol{x_t}\boldsymbol{W} + \boldsymbol{h_{t-1}}\boldsymbol{U} + \boldsymbol{b})$ – скрытое состояние
  $\boldsymbol{y_t} = f_y(\boldsymbol{h_t}\boldsymbol{V} + \boldsymbol{c})$
  $\boldsymbol{h_0} = 0$

[7] Elman "Finding Structure in Time" (1990), http://psych.colorado.edu/~kimlab/Elman1990.pdf

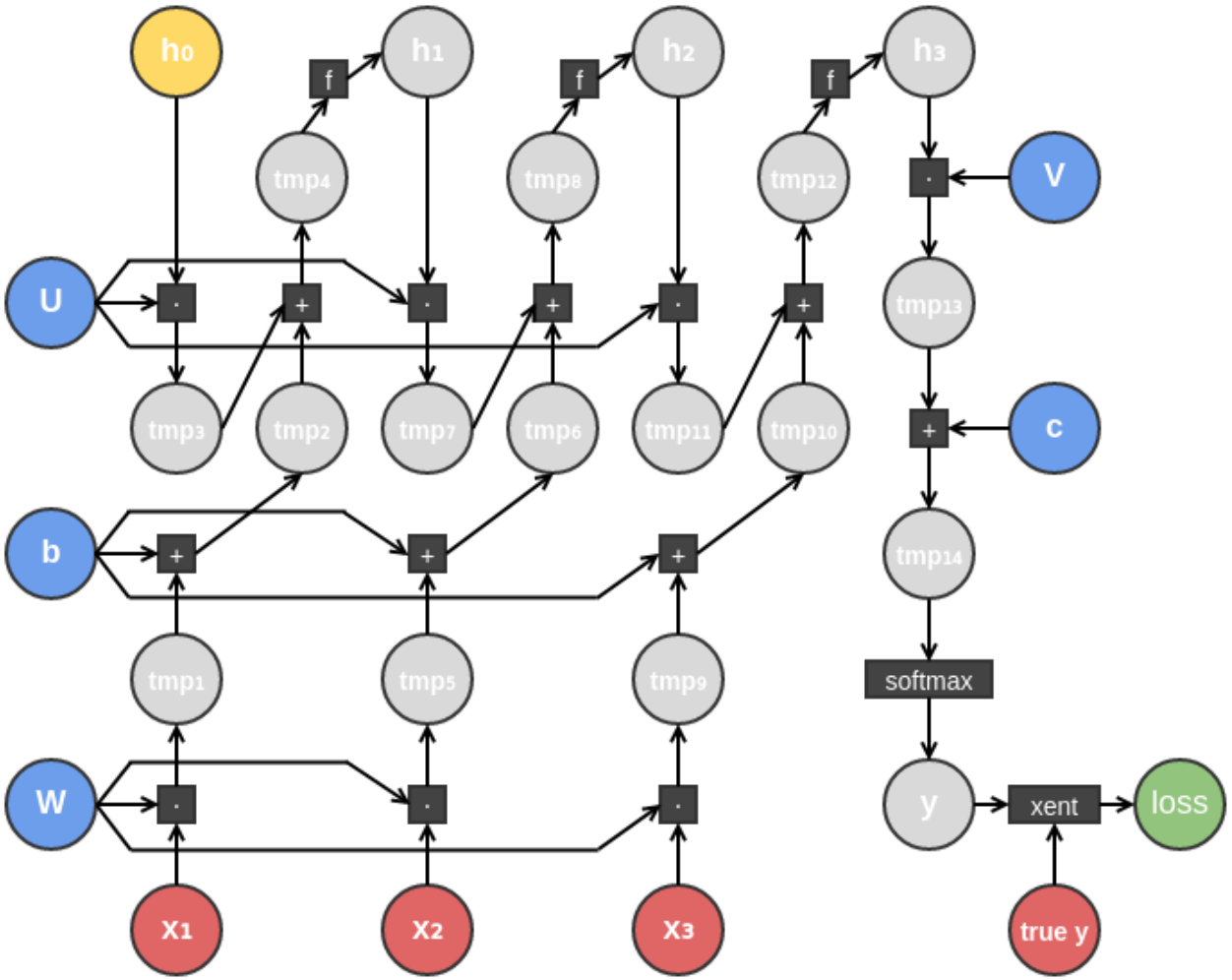
Вычисление градиента в RNN

- Пусть мы хотим классифицировать короткие тексты
- Возьмем обученный word2vec и представим каждый текст как матрицу $T \times D$, где T – количество слов, D – размерность word2vec
- Используем RNN:

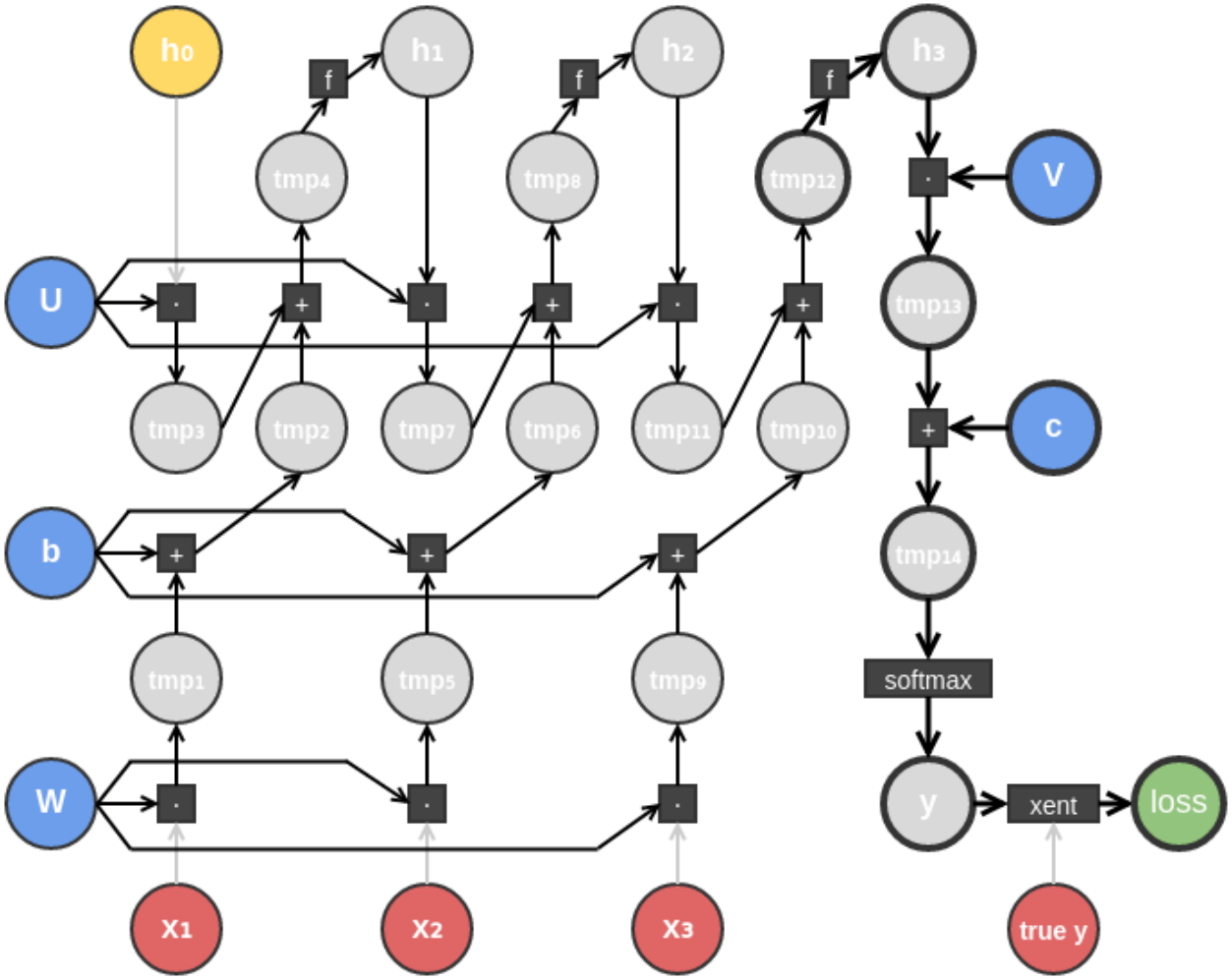
$$\boldsymbol{h_t} = \tanh(\boldsymbol{x_t}\boldsymbol{W} + \boldsymbol{h_{t-1}}\boldsymbol{U} + \boldsymbol{b}), t = 1 \dots T$$
$$\boldsymbol{y} = \text{softmax}(\boldsymbol{h_T}\boldsymbol{V} + \boldsymbol{c}) - \text{вероятности классов}$$
$$\boldsymbol{h_0} = \boldsymbol{0}$$

- И кросс-энтропию в соответствии с принципом максимального правдоподобия

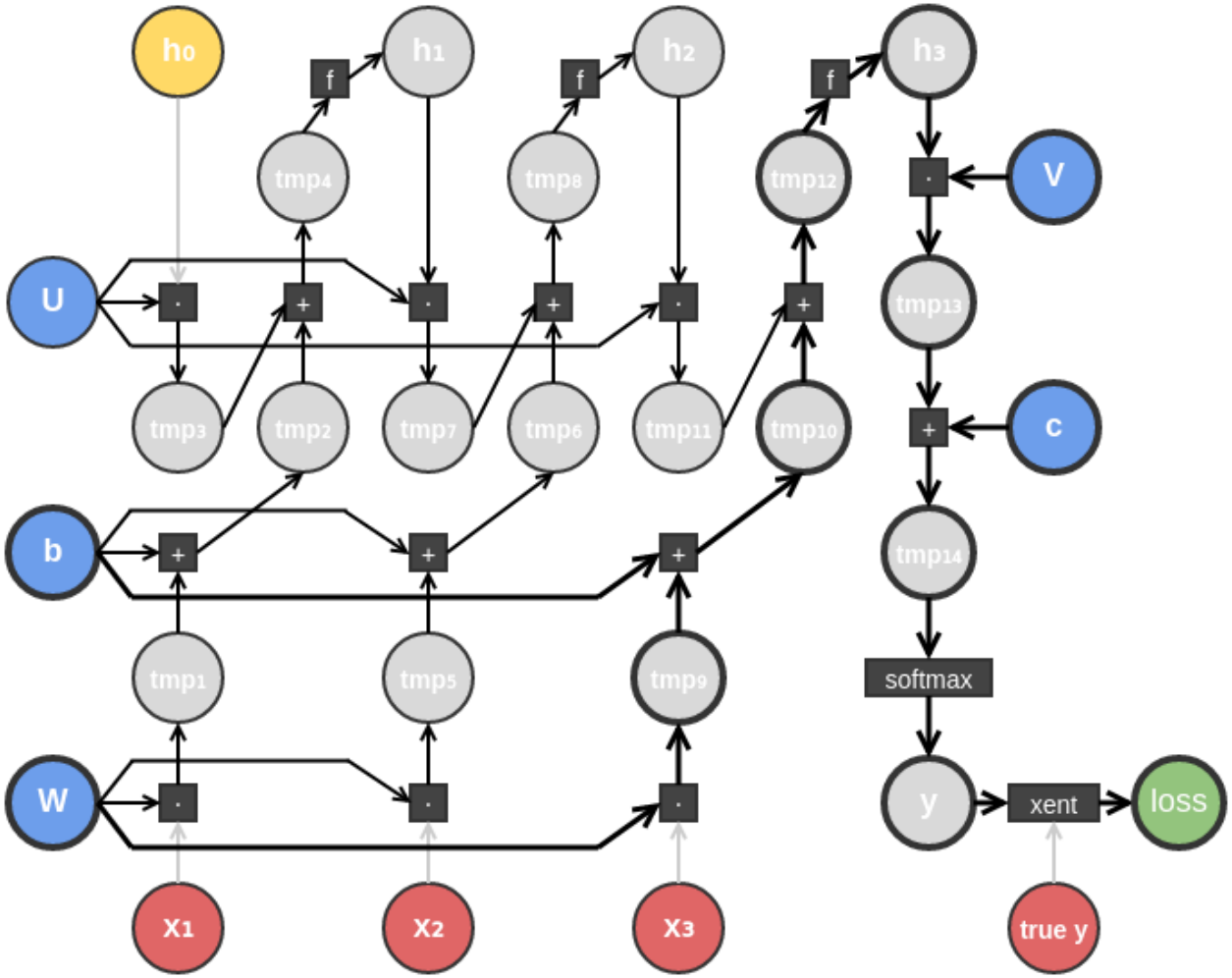
Вычисление градиента в RNN



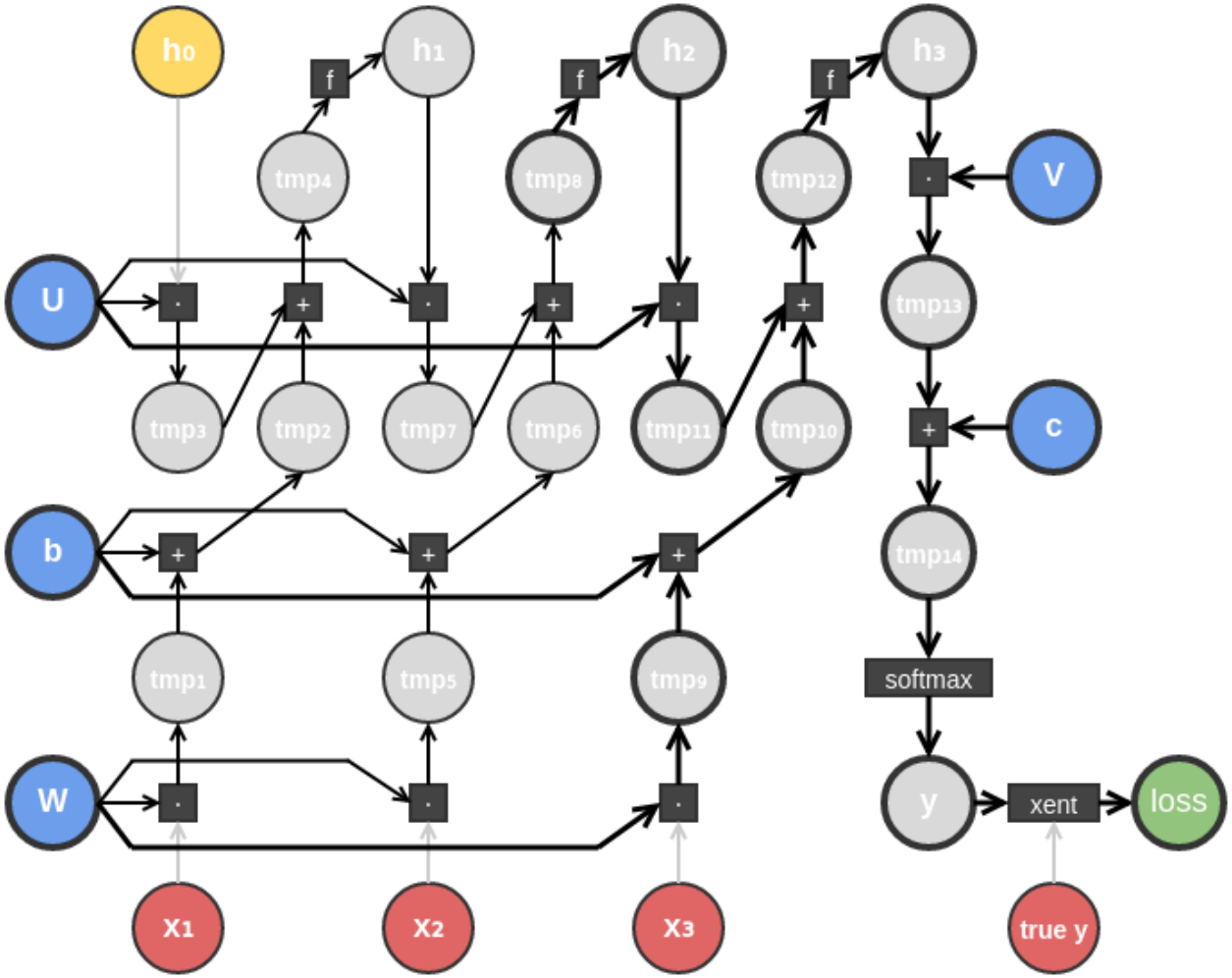
Вычисление градиента в RNN



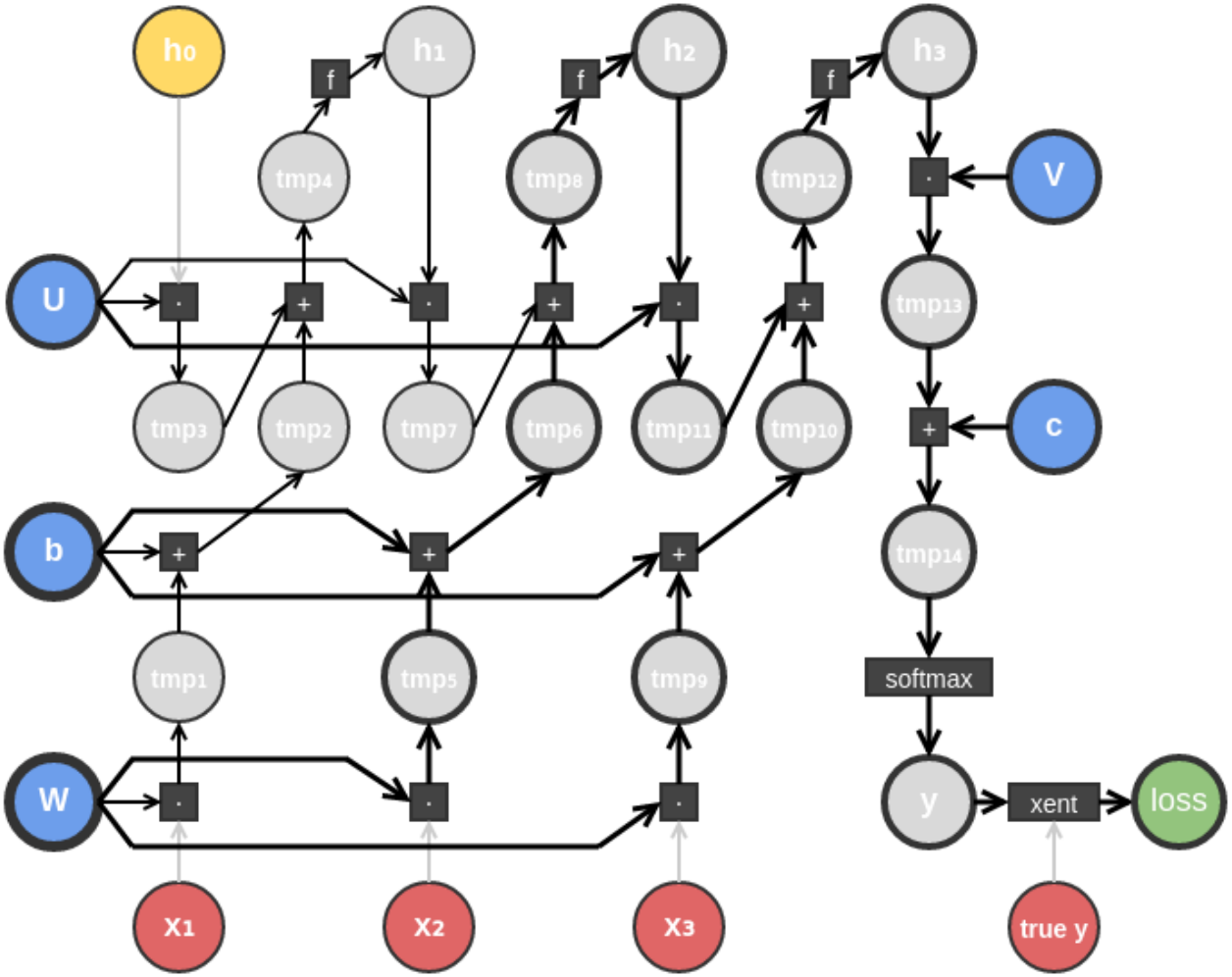
Вычисление градиента в RNN



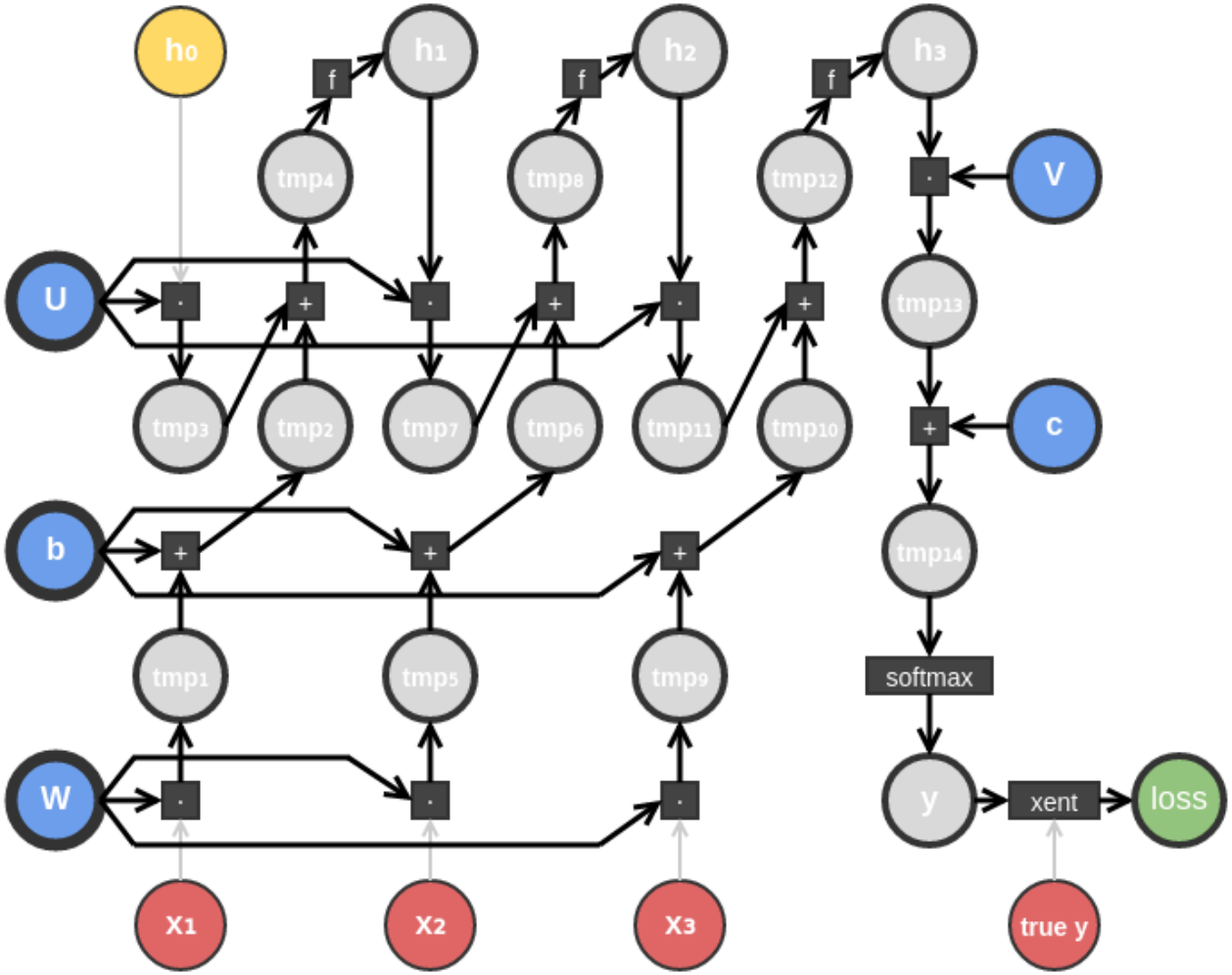
Вычисление градиента в RNN



Вычисление градиента в RNN



Вычисление градиента в RNN



Вычисление градиента в RNN

- Вершина $\boldsymbol{W}$ получает части градиента потери по себе от трех вершин $\boldsymbol{tmp_1}$, $\boldsymbol{tmp_5}$ и $\boldsymbol{tmp_9}$ – эти части суммируются
- Аналогично для вершин $\boldsymbol{U}$ и $\boldsymbol{b}$

RNN и минибатчи

- Последовательности на входе RNN имеют разную длину
- Чтобы сформировать минибатч (3-мерный тензор), придется выровнять (padding) последовательности среди всех объектов минибатча
- При этом выходы RNN для частей входов, относящихся к выравниванию, следует проигнорировать

Затухание (vanishing) градиента в RNN

- $\boldsymbol{h_t} = f_h(\boldsymbol{x_t}\boldsymbol{W} + \boldsymbol{h_{t-1}}\boldsymbol{U} + \boldsymbol{b})$
  $\boldsymbol{y_t} = f_y(\boldsymbol{h_t}\boldsymbol{V} + \boldsymbol{c})$
  $\boldsymbol{h_0} = 0$

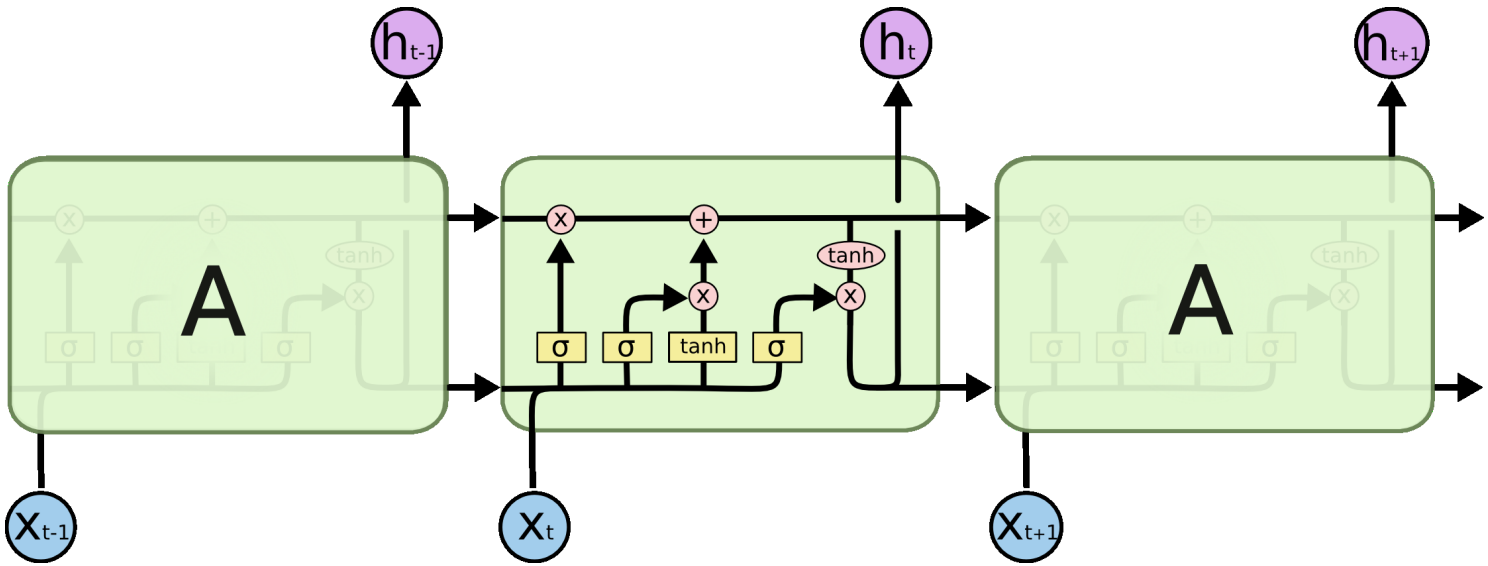
$$\nabla_{\boldsymbol{x_5}} \boldsymbol{h_5} = \text{diag}(f_h'(\boldsymbol{x_5}\boldsymbol{W} + \boldsymbol{h_4}\boldsymbol{U} + \boldsymbol{b}))\boldsymbol{W}^T$$

$$\nabla_{\boldsymbol{x_2}} \boldsymbol{h_5} = \nabla_{\boldsymbol{h_4}} \boldsymbol{h_5} \nabla_{\boldsymbol{h_3}} \boldsymbol{h_4} \nabla_{\boldsymbol{h_2}} \boldsymbol{h_3} \cdot \text{diag}(f_h'(\boldsymbol{x_2}\boldsymbol{W} + \boldsymbol{h_1}\boldsymbol{U} + \boldsymbol{b}))\boldsymbol{W}^T$$

$$\nabla_{\boldsymbol{h_{t-1}}} \boldsymbol{h_t} = \text{diag}(f_h'(\boldsymbol{x_t}\boldsymbol{W} + \boldsymbol{h_{t-1}}\boldsymbol{U} + \boldsymbol{b}))\boldsymbol{U}^T$$

- Если $|f_h'(\cdot)|\|\boldsymbol{U}\|$ существенно меньше 1, то произойдет затухание градиента, и RNN не сможет запоминать информацию на долгое время (градиент по давним входам почти нулевой)
- Также опасна обратная ситуация (взрыв градиента)

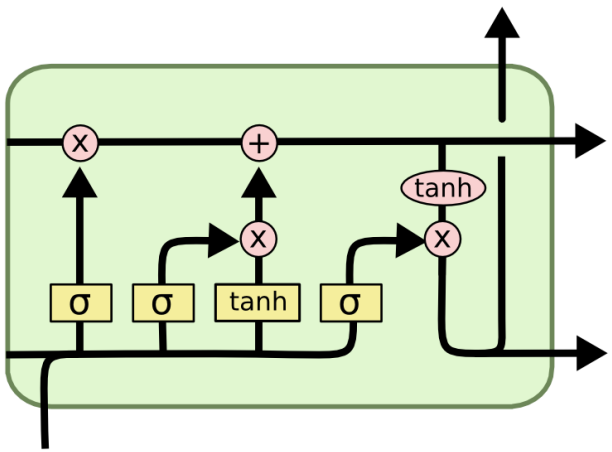
Long Short-Term Memory



Christopher Olah "Understanding LSTM Networks",
http://colah.github.io/posts/2015-08-Understanding-LSTMs

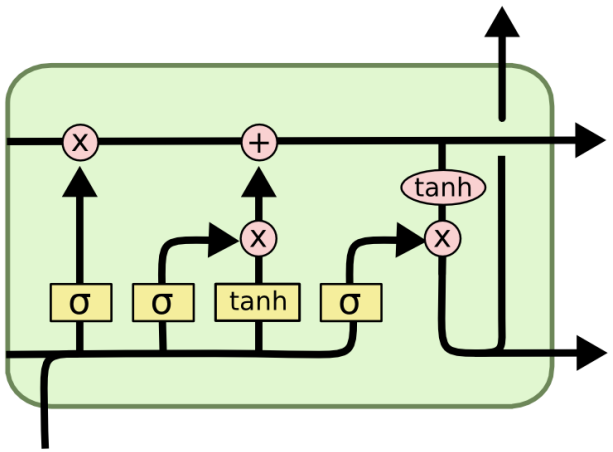
Long Short-Term Memory

- $\boldsymbol{f_t} = \sigma(\boldsymbol{x_t}\boldsymbol{W_f} + \boldsymbol{h_{t-1}}\boldsymbol{U_f} + \boldsymbol{b_f})$
  $\boldsymbol{i_t} = \sigma(\boldsymbol{x_t}\boldsymbol{W_i} + \boldsymbol{h_{t-1}}\boldsymbol{U_i} + \boldsymbol{b_i})$
  $\tilde{\boldsymbol{c}}_t = \tanh(\boldsymbol{x_t}\boldsymbol{W_c} + \boldsymbol{h_{t-1}}\boldsymbol{U_c} + \boldsymbol{b_c})$
  $\boldsymbol{o_t} = \sigma(\boldsymbol{x_t}\boldsymbol{W_o} + \boldsymbol{h_{t-1}}\boldsymbol{U_o} + \boldsymbol{b_o})$
  $\boldsymbol{c_t} = \boldsymbol{f_t} \odot \boldsymbol{c_{t-1}} + \boldsymbol{i_t} \odot \tilde{\boldsymbol{c}}_t$
  $\boldsymbol{h_t} = \boldsymbol{o_t} \odot \tanh(\boldsymbol{c_t})$
- LSTM обладает дополнительным скрытым состоянием $\boldsymbol{c}$, а состояние $\boldsymbol{h}$ одновременно является выходом



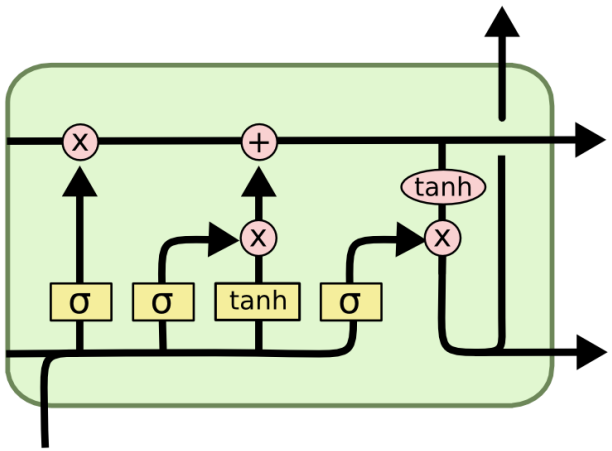
Long Short-Term Memory

- $\boldsymbol{f_t} = \sigma(\boldsymbol{x_t W_f} + \boldsymbol{h_{t-1} U_f} + \boldsymbol{b_f})$
  $\boldsymbol{i_t} = \sigma(\boldsymbol{x_t W_i} + \boldsymbol{h_{t-1} U_i} + \boldsymbol{b_i})$
  $\tilde{\boldsymbol{c}}_t = \tanh(\boldsymbol{x_t W_c} + \boldsymbol{h_{t-1} U_c} + \boldsymbol{b_c})$
  $\boldsymbol{o_t} = \sigma(\boldsymbol{x_t W_o} + \boldsymbol{h_{t-1} U_o} + \boldsymbol{b_o})$
  $\boldsymbol{c_t} = \boldsymbol{f_t} \odot \boldsymbol{c_{t-1}} + \boldsymbol{i_t} \odot \tilde{\boldsymbol{c}}_t$
  $\boldsymbol{h_t} = \boldsymbol{o_t} \odot \tanh(\boldsymbol{c_t})$
- Forget gate: с учетом $\boldsymbol{x_t}$ определяет, какую информацию из $\boldsymbol{c_{t-1}}$ стоит сохранить



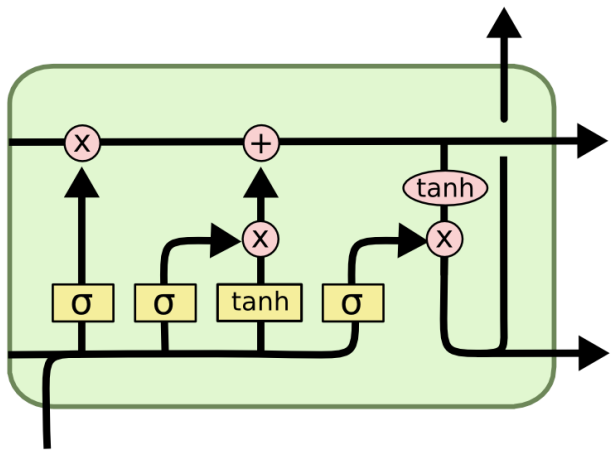
Long Short-Term Memory

- $\boldsymbol{f_t} = \sigma(\boldsymbol{x_t W_f} + \boldsymbol{h_{t-1} U_f} + \boldsymbol{b_f})$
  $\boldsymbol{i_t} = \sigma(\boldsymbol{x_t W_i} + \boldsymbol{h_{t-1} U_i} + \boldsymbol{b_i})$
  $\tilde{\boldsymbol{c}}_t = \tanh(\boldsymbol{x_t W_c} + \boldsymbol{h_{t-1} U_c} + \boldsymbol{b_c})$
  $\boldsymbol{o_t} = \sigma(\boldsymbol{x_t W_o} + \boldsymbol{h_{t-1} U_o} + \boldsymbol{b_o})$
  $\boldsymbol{c_t} = \boldsymbol{f_t} \odot \boldsymbol{c_{t-1}} + \boldsymbol{i_t} \odot \tilde{\boldsymbol{c}}_t$
  $\boldsymbol{h_t} = \boldsymbol{o_t} \odot \tanh(\boldsymbol{c_t})$
- Input gate и cell gate: определяют важность информации из $\boldsymbol{x_t}$ и преобразуют важную информацию для сохранения в состоянии $\boldsymbol{c}$



Long Short-Term Memory

- $\boldsymbol{f_t} = \sigma(\boldsymbol{x_t W_f} + \boldsymbol{h_{t-1} U_f} + \boldsymbol{b_f})$
  $\boldsymbol{i_t} = \sigma(\boldsymbol{x_t W_i} + \boldsymbol{h_{t-1} U_i} + \boldsymbol{b_i})$
  $\tilde{\boldsymbol{c}}_t = \tanh(\boldsymbol{x_t W_c} + \boldsymbol{h_{t-1} U_c} + \boldsymbol{b_c})$
  $\boldsymbol{o_t} = \sigma(\boldsymbol{x_t W_o} + \boldsymbol{h_{t-1} U_o} + \boldsymbol{b_o})$
  $\boldsymbol{c_t} = \boldsymbol{f_t} \odot \boldsymbol{c_{t-1}} + \boldsymbol{i_t} \odot \tilde{\boldsymbol{c}}_t$
  $\boldsymbol{h_t} = \boldsymbol{o_t} \odot \tanh(\boldsymbol{c_t})$
- Output gate: определяет, какую информацию из обновленного состояния $\boldsymbol{c}$ выдать в качестве ответа



LSTM и проблема нестабильного градиента

$$\nabla_{\boldsymbol{c}_{t-1}} \boldsymbol{c}_t = \operatorname{diag}(\boldsymbol{f}_t)$$

- Теперь нейросеть может запоминать информацию надолго в состоянии $\boldsymbol{c}$
- Смещение $\boldsymbol{b}_f$ forget gate желательно инициализировать не вектором из нулей, а вектором из единиц
- Для уменьшения вероятности дестабилизации градиента можно инициализировать матрицы $\boldsymbol{U}$ случайными ортогональными матрицами, также очень полезно делать gradient clipping

Содержание

Библиотеки ~~глубинного~~ глубокого обучения

- Theano – некогда популярная, дала развитие современным библиотекам. Поддержка прекращена
- TensorFlow (Google) – сейчас наиболее известная, пригодная для промышленного использования, сложна в освоении и поддержке
- PyTorch – более простая и гибкая, чем TF, активно развивается, пока есть проблемы с промышленным использованием и распределенным обучением очень больших моделей

PYTORCH

Недостатки TensorFlow

- В TensorFlow граф вычислений компилируется целиком в эффективный машинный код (использующий GPU)
- Появляются проблемы[8] с реализацией некоторых моделей, например, TreeLSTM над деревьями синтаксического разбора произвольной глубины
- Запущенная модель кажется отрезанной от пользователя, с ней трудно контактировать на языке Python
- PyTorch с самого начала лишена этих недостатков

[8] Решаются использованием далеко не самого изящного решения TensorFlow Fold

Пример кода на PyTorch

- Задача с SentiRuEval-2016: классифицируем твиты о мобильных операторах на негативные, нейтральные и позитивные
- Имеем матрицу `X`, содержащую выровненные признаки (индексы слов в словаре) для всех объектов обучающей выборки
- Также имеем вектор `y` меток классов этих объектов и вектор `lengths` с длиной признаков (количеством слов без выравнивания) для каждого объекта
- Пусть у нас уже есть массив numpy `w2v` векторных представлений word2vec

Класс LSTM

```
import torch
import torch.nn as nn
from torch.autograd import Variable
from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence


class LSTM(nn.Module):
    def __init__(self, input_size, output_size, num_layers):
        super().__init__()
        self.lstm = nn.LSTM(input_size, output_size, num_layers)
        self.output_size, self.num_layers = output_size, num_layers

    def forward(self, inputs, lengths):
        """
        :param inputs: torch Variable of size (seq_len, batch, input_size)
        :param lengths: torch LongTensor of size (batch,)
        :return: torch Variable of size (batch, output_size)
        """
        inputs_packed = pack_padded_sequence(inputs, list(lengths))
        h0 = Variable(torch.zeros(
            self.num_layers, inputs.size(1), self.output_size)).cuda()
        c0 = Variable(torch.zeros(
            self.num_layers, inputs.size(1), self.output_size)).cuda()
        outputs_packed, _ = self.lstm(inputs_packed, (h0, c0))
        outputs = pad_packed_sequence(outputs_packed)[0]
        return outputs[lengths - 1, range(outputs.size(1))]
```

Класс SentiNet

```
class SentiNet(nn.Module):
    def __init__(self, w2v, rnn, num_classes):
        super().__init__()
        self.rnn = rnn
        self.fc = nn.Linear(rnn.output_size, num_classes)
        self.softmax = nn.Softmax()
        self.cuda()
        self.emb = nn.Embedding(w2v.size(0), w2v.size(1))
        self.emb.weight = nn.Parameter(w2v)
        self.emb.weight.requires_grad = False

    def forward(self, inputs, lengths):
        """
        :param inputs: torch Variable of size (seq_len, batch)
        :param lengths: torch LongTensor of size (batch,)
        :return: torch Variable of size (batch, num_classes)
        """
        inputs_emb = self.emb(inputs).cuda()
        return self.fc(self.rnn(inputs_emb, lengths.cuda()))

    def predict(self, inputs, lengths):
        return self.softmax(self.forward(inputs, lengths)).cpu()

    def parameters(self):
        return filter(lambda p: p.requires_grad, super().parameters())
```

Обучение нейросети

```
vocab_size, vector_size = w2v.shape
seq_len, num_objects = X.shape
num_lstm_layers = 2
num_classes = 3
batch_size = 64

rnn = LSTM(vector_size, vector_size, num_lstm_layers)
net = SentiNet(torch.from_numpy(w2v), rnn, num_classes)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=.001)

for epoch in range(10):
    optimizer.zero_grad()
    for i in range(num_objects // batch_size):
        X_batch = Variable(torch.from_numpy(X[:, i * batch_size: (i + 1) * batch_size]))
        y_batch = Variable(torch.from_numpy(y[i * batch_size: (i + 1) * batch_size])).cuda()
        l_batch = torch.LongTensor(lengths[i * batch_size: (i + 1) * batch_size])
        outputs = net(X_batch, l_batch)
        loss = criterion(outputs, y_batch)
        loss.backward()
        optimizer.step()

y_pred = net.predict(Variable(torch.from_numpy(X_test)), torch.LongTensor(l_test))\
    .data.numpy().argmax(axis=1)
from sklearn.metrics import f1_score
print(f1_score(y_test, y_pred, average='macro'))
```

Приложения deep learning в обработке языка

- Определение эмоциональной окраски, сарказма и т. п.
- Синтаксический анализ текстов
- Более продвинутый информационный поиск (Яндекс “Королев”)
- Машинный перевод (Google Translate)
- Чат-боты и голосовые помощники (Replika, Samsung Bixby, Алиса)
- ?

Ключевые слова для поиска в Google

pytorch
pytorch tutorials
torchvision
tensorflow
fasttext
syntaxnet
acl 2017

deep learning book
arxiv sanity
attention mechanism
variational autoencoder
layer normalization
simple recurrent unit
selu activation