Нейронные сети

Обработка текстов, лекция 6

Константин Архипенко

arkhipenko@ispras.ru, vk.com/konstantin_arkhipenko

21 октября 2016 г.

Нейронные сети

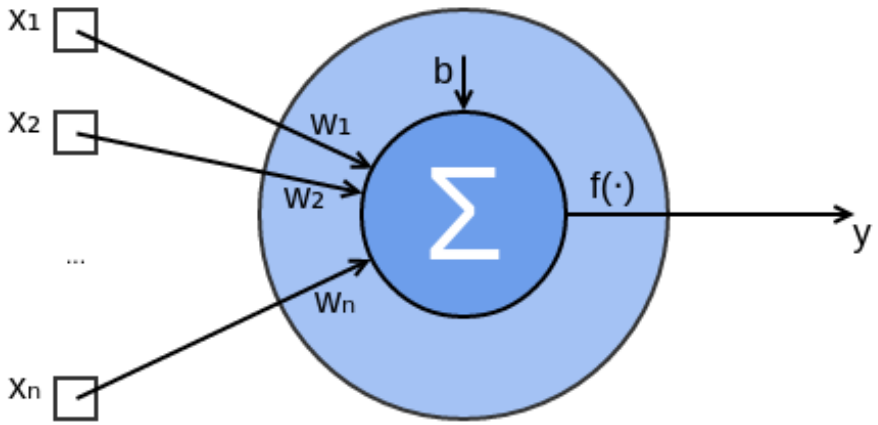
- Мощные алгоритмы машинного обучения, показывающие лучшие результаты во многих задачах обработки изображений, звука, **текстов** и других данных

Популярные архитектуры нейросетей

- Нейронные сети прямого распространения
  - Многослойный персептрон
  - Свёрточные нейронные сети – рассматривать не будем, используются преимущественно для обработки изображений
- Рекуррентные нейронные сети
- Часто нейросети различных архитектур соединяют, образуя более сложные и мощные нейросети

Содержание

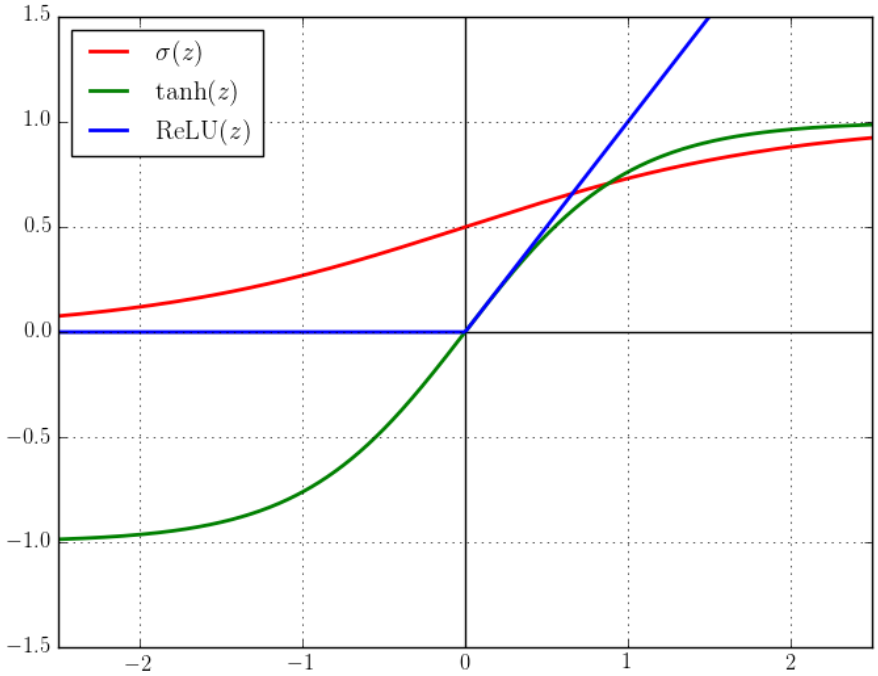
Нейрон



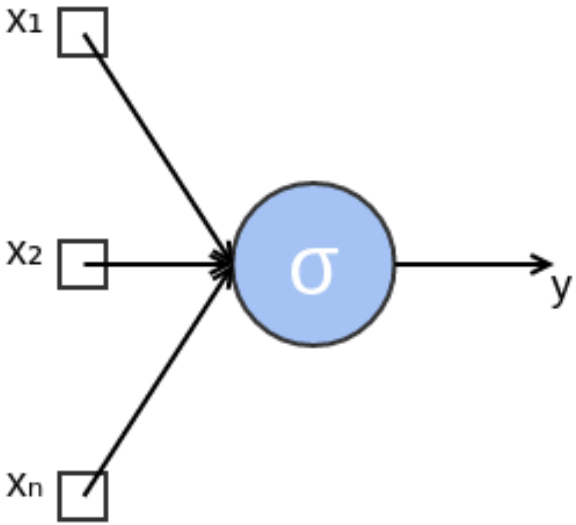
- $y(\mathbf{x}) = f(\langle \mathbf{x}, \mathbf{w} \rangle + b)$
- $f(\cdot)$ – функция активации, обычно нелинейная
- Обучение нейрона есть настройка его параметров $\mathbf{w}$ и b

Функции активации

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

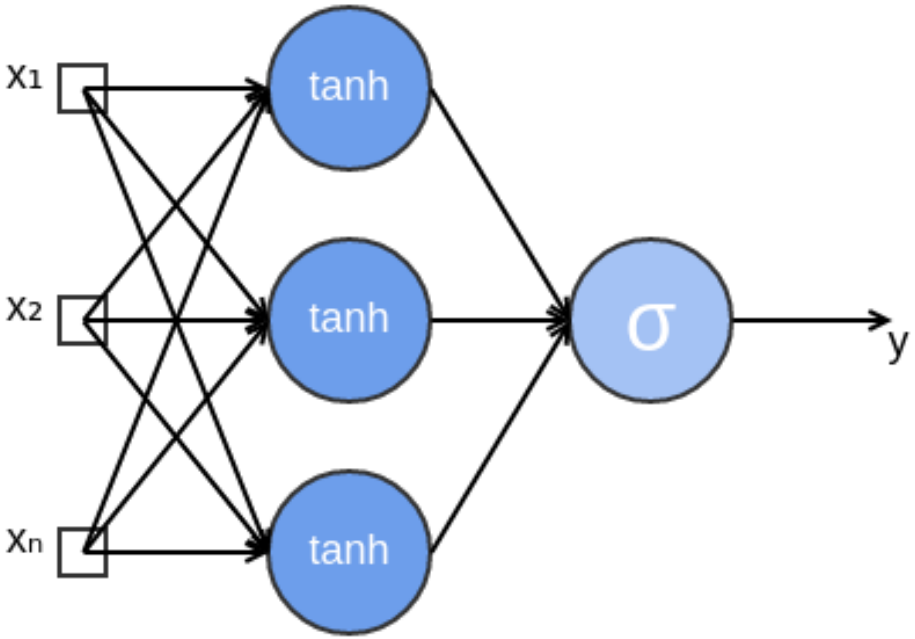


Нейрон с функцией активации $\sigma(z)$



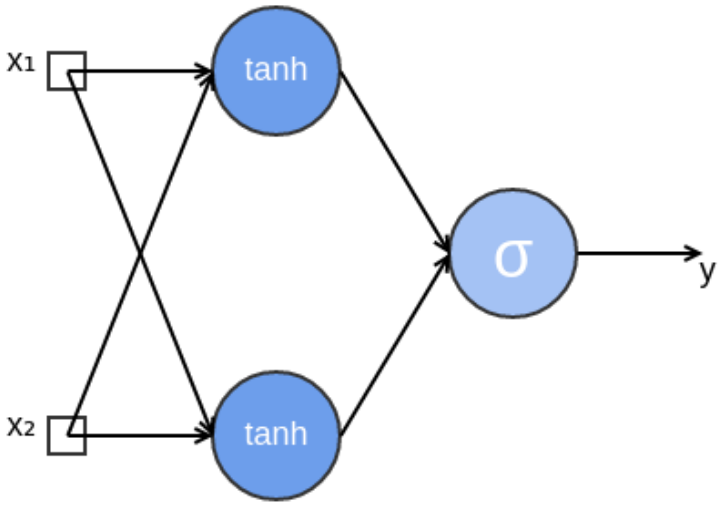
- $y(\mathbf{x}) = \sigma(\langle \mathbf{x}, \mathbf{w} \rangle + b)$
- Получается не что иное, как логистическая регрессия

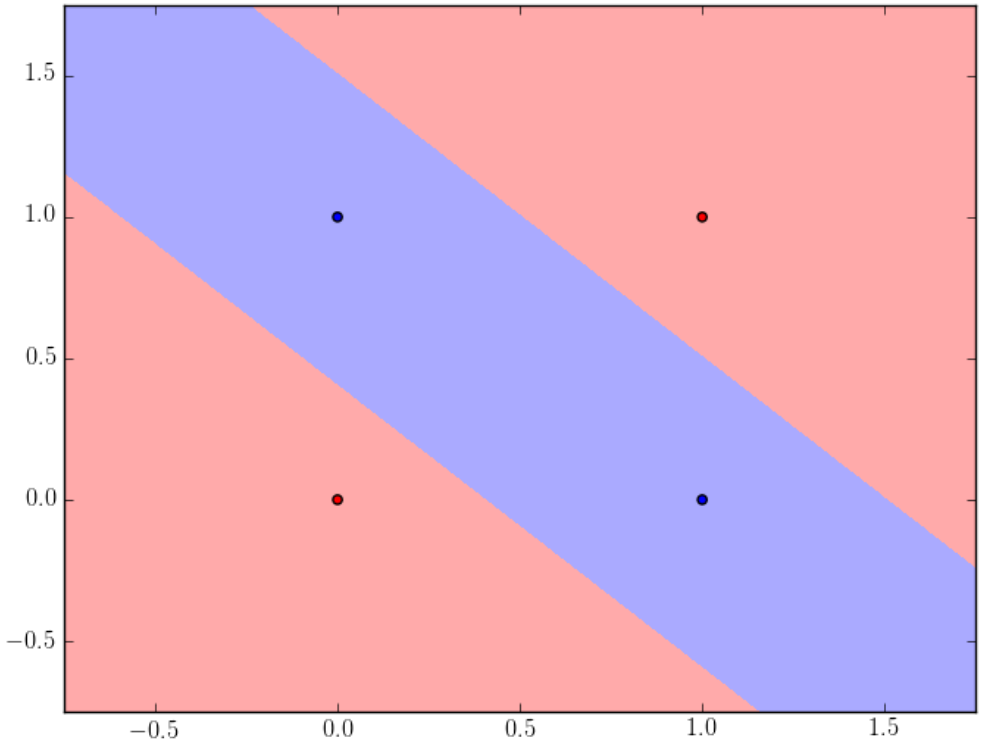
Добавим слой нейронов



- $h_i = \tanh(\langle \mathbf{x}, \mathbf{w_{1_i}} \rangle + b_{1_i})$
  $y = \sigma(\langle \mathbf{h}, \mathbf{w_2} \rangle + b_2)$
- Заведомо мощнее, чем логистическая регрессия

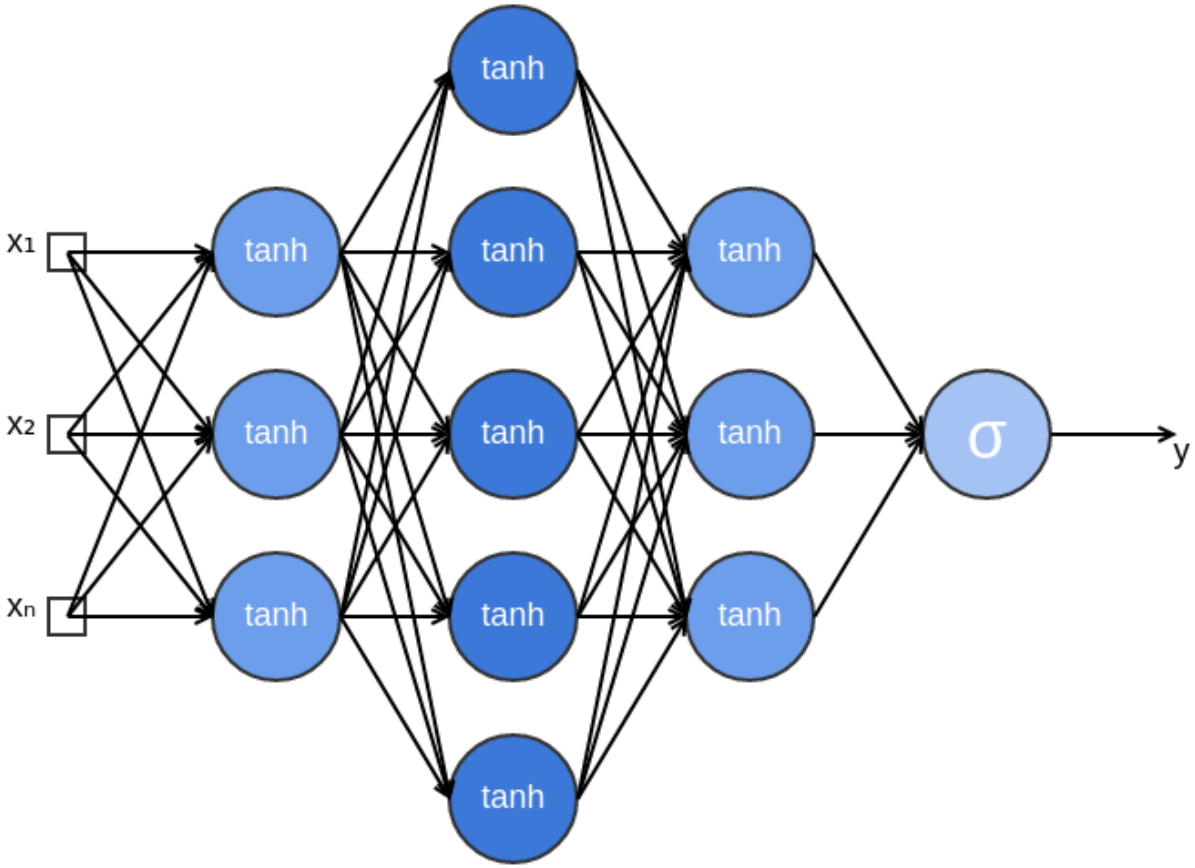
XOR



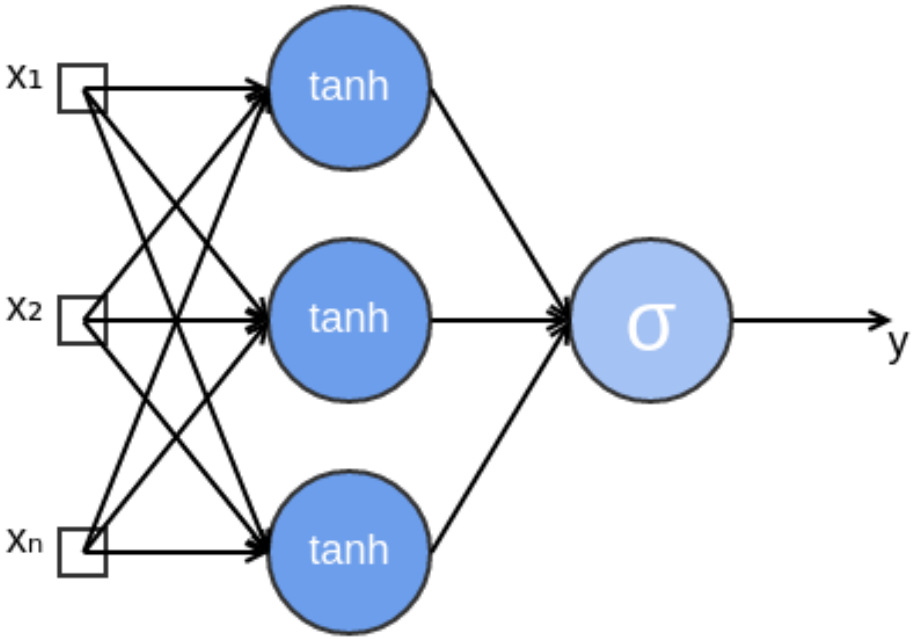


- Линейные классификаторы на такое не способны

Многослойный персептрон

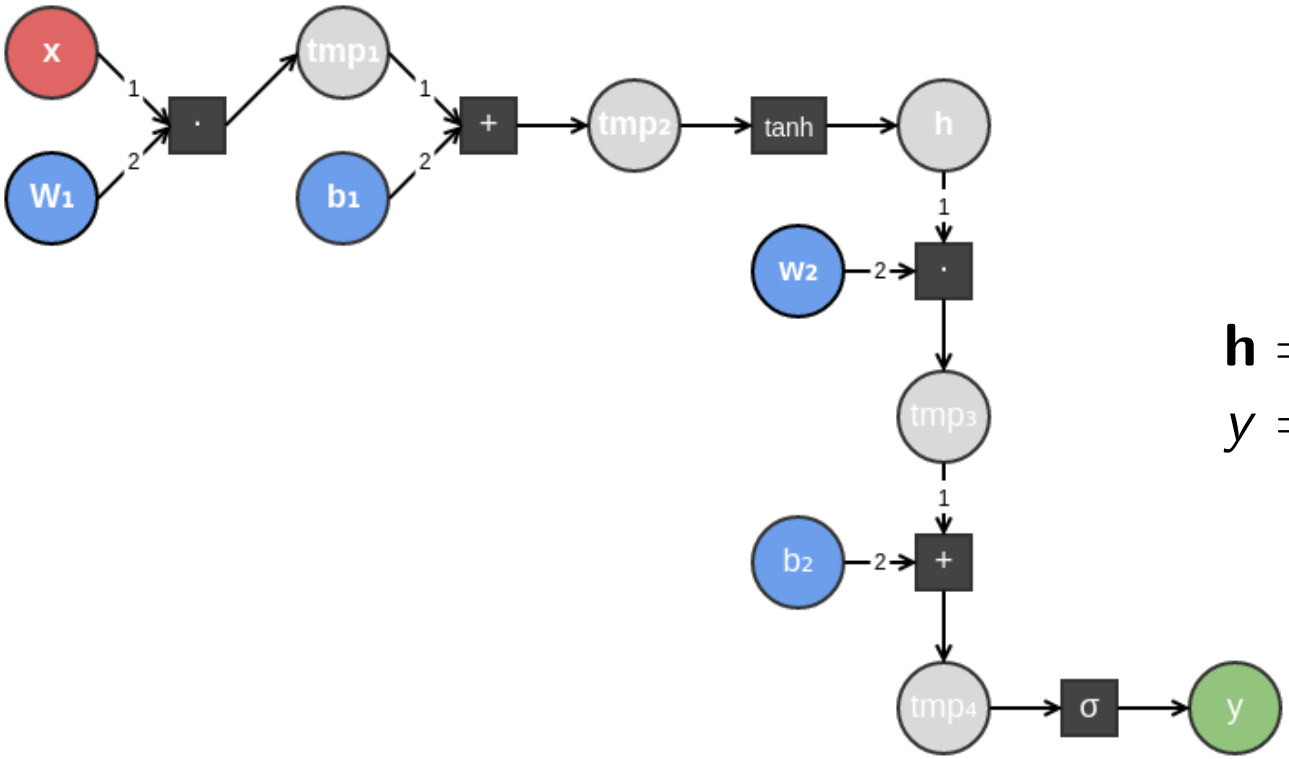


Лаконичная запись происходящего в нейросети



- $\mathbf{h} = \tanh(\mathbf{x}\mathbf{W_1} + \mathbf{b_1})$
  $y = \sigma(\mathbf{h}\mathbf{w_2^T} + b_2)$
- Соглашение: используем векторы-строки; при умножении на матрицу вектор находится слева

Граф вычислений



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

Мультиномиальная логистическая регрессия

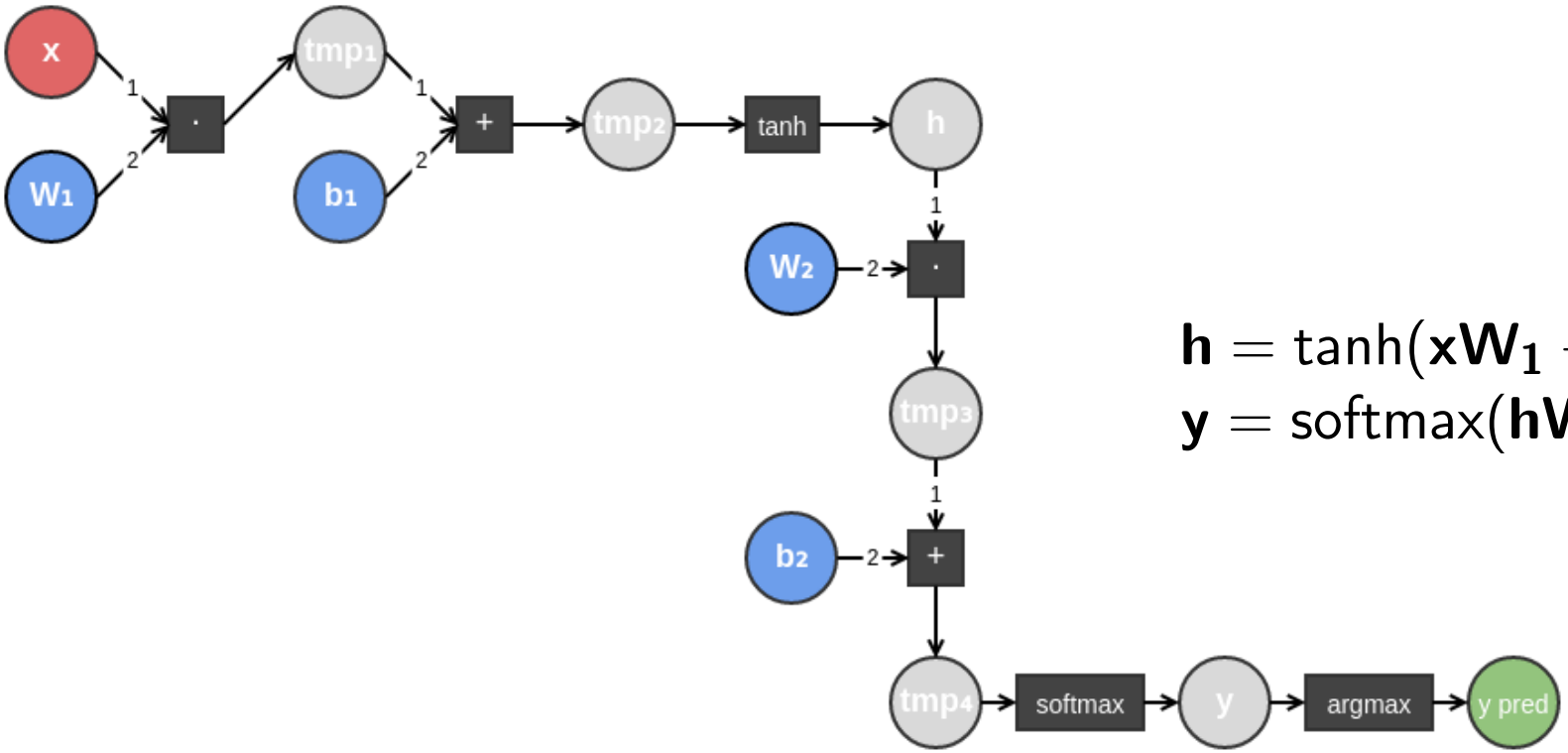
- Softmax:

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

- Аналогично, усилим мультиномиальную логистическую регрессию добавлением слоя:

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

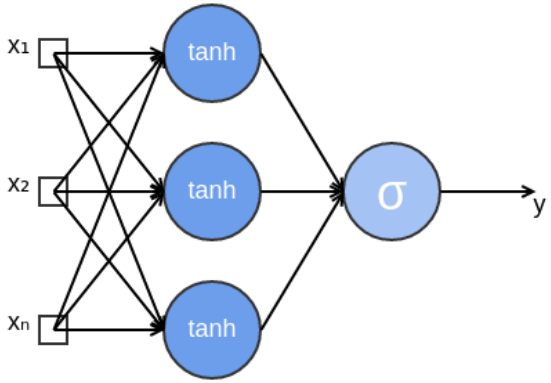
Граф вычислений



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

Векторизация + графы вычислений

- С этого момента мы отказываемся от подобных картинок:



- Дело в том, что softmax нельзя представить такими нейронами
- Будем добиваться максимальной степени векторизации и использовать графы вычислений

Содержание

Как обучается логистическая регрессия

- Вспомним метод максимального правдоподобия:

$$L = -\frac{1}{N}\sum_{n=1}^{N} \log P(y_{true_n}|\mathbf{x_n}) + \lambda\|\mathbf{w}\|^2 \to \min_{\mathbf{w},b}$$

$$P(1|\mathbf{x_n}) = 1 - P(0|\mathbf{x_n}) = y_n = \sigma(\mathbf{x_n}\mathbf{w^T} + b)$$

Кросс-энтропия

- Перепишем $-\log P(y_{true_n}|\mathbf{x_n})$:

$$-\log P(y_{true_n}|\mathbf{x_n}) = -(y_{true_n} \log y_n + (1 - y_{true_n}) \log(1 - y_n))$$

- В правой части находится бинарная кросс-энтропия $\text{binaryCrossentropy}(y_{true_n}, y_n)$:

$$\text{binaryCrossentropy}(t, o) = -(t \log o + (1 - t) \log(1 - o))$$

Как обучается логистическая регрессия

- Таким образом, максимизация правдоподобия есть то же самое, что минимизация кросс-энтропии:

$$L = \frac{1}{N}\sum_{n=1}^{N} \text{binaryCrossentropy}(y_{true_n}, y_n) + \lambda\|\mathbf{w}\|^2 \to \min_{\mathbf{w},b}$$

$$y_n = \sigma(\mathbf{x_n}\mathbf{w}^\mathbf{T} + b)$$

- В случае многоклассовой классификации $\text{binaryCrossentropy}(y_{true_n}, y_n)$ следует заменить на обычную кросс-энтропию $\text{categoricalCrossentropy}(\mathbf{y_{true_n}}, \mathbf{y_n})$

Градиентный спуск

- На каждой итерации t обновляем параметры $\mathbf{w}$ и b:

$$w_i \leftarrow w_i - \alpha_t \frac{\partial L}{\partial w_i} \qquad b \leftarrow b - \alpha_t \frac{\partial L}{\partial b} \qquad \alpha_t > 0$$

- Градиент функции потерь выписывается в явном виде
- А вот найти стационарную точку аналитически не удаётся – поэтому и используется градиентный спуск

Стохастический градиентный спуск (SGD)

- Вычисление градиента L на каждой итерации t дорого, поскольку в L входит сумма по всем элементам выборки
- Вместо этого предлагается для каждой итерации выбрать свой элемент выборки $(\mathbf{x_n}, y_{true_n})$ и вычислить градиент потери L_n только на этом элементе:

$$L_n = \text{binaryCrossentropy}(y_{true_n}, y_n) + \lambda \|\mathbf{w}\|^2$$

$$w_i \leftarrow w_i - \alpha_t \frac{\partial L_n}{\partial w_i} \qquad b \leftarrow b - \alpha_t \frac{\partial L_n}{\partial b} \qquad \alpha_t > 0$$

- α_t зависит только от t и обычно уменьшается со временем для сходимости

SGD с мини-батчами

- Между двумя крайностями есть золотая середина
- Разобьём выборку на мини-батчи (группы одинакового размера), а затем на каждой итерации вычисляем градиент потери L_b на определённом мини-батче b

SGD с мини-батчами

- Градиент L аппроксимируется лучше, чем в обычном SGD – можно двигаться увереннее (использовать бóльшие значения α_t)
- Эффективность почти не теряется, если вычисления в рамках мини-батча можно распараллелить
- Для обучения нейронных сетей на GPU мини-батчи используются почти всегда

Обучение нейронной сети

- Мы подробно рассмотрим процесс обучения следующей нейронной сети для задачи бинарной классификации:

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

- Используем бинарную кросс-энтропию в качестве функции потерь и SGD без мини-батчей
- Нам потребуется ввести понятие якобиана и вспомнить цепное правило

Якобиан

- Пусть $\mathbf{v} = f(\mathbf{u})$. Тогда $\frac{\partial \mathbf{v}}{\partial \mathbf{u}}$ – матрица:

$$(\frac{\partial \mathbf{v}}{\partial \mathbf{u}})_{ij} = \frac{\partial v_i}{\partial u_j}$$

- Мы обобщим понятие якобиана

Якобиан

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

$$(\frac{\partial v}{\partial \mathbf{u}})_i = \frac{\partial v}{\partial u_i}$$

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

$$(\frac{\partial v}{\partial \mathbf{U}})_{ij} = \frac{\partial v}{\partial u_{ij}}$$

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

$$(\frac{\partial \mathbf{v}}{\partial \mathbf{U}})_{ijk} = \frac{\partial v_i}{\partial u_{jk}}$$

Цепное правило

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

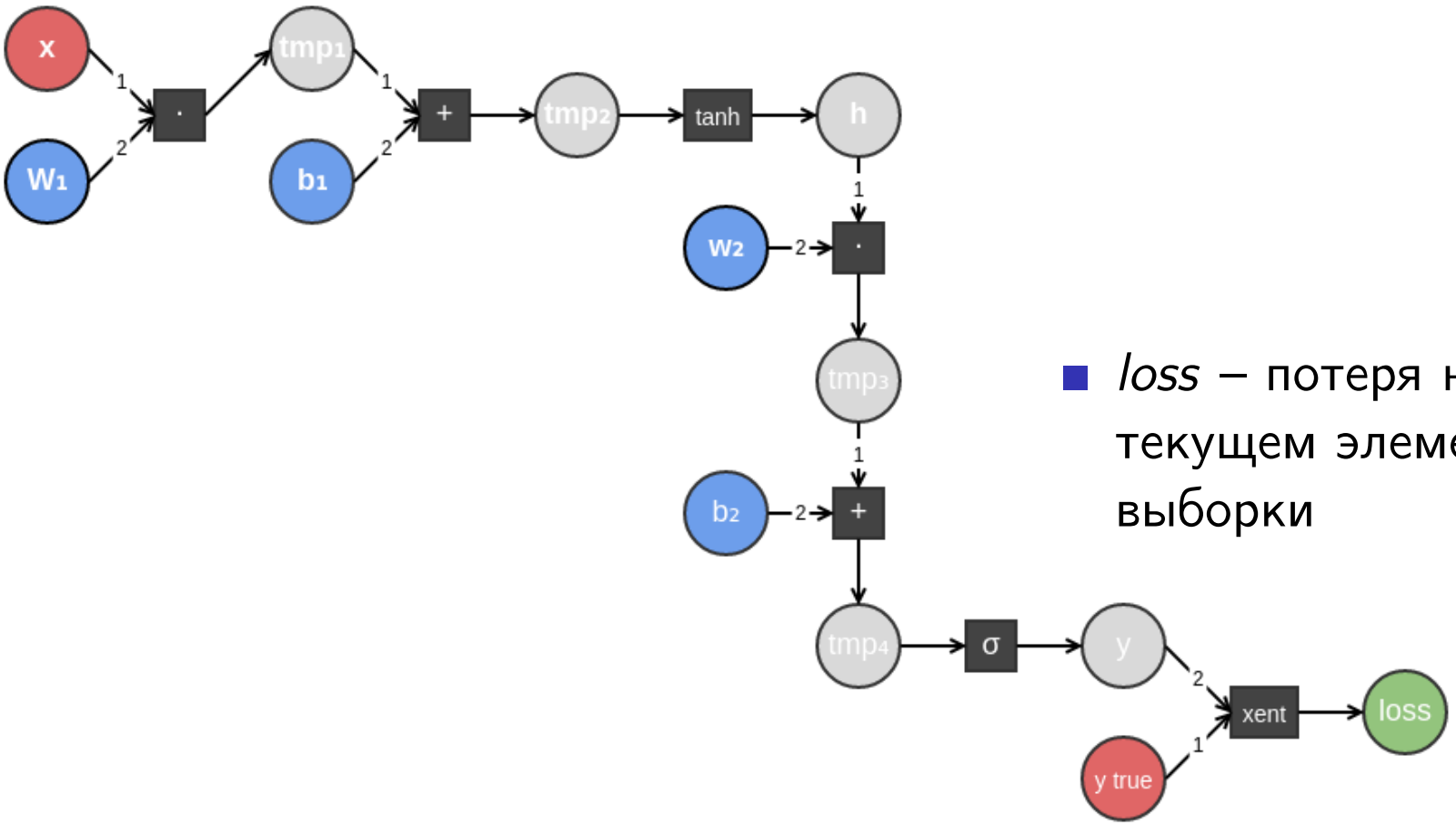
$$\frac{\partial w}{\partial u} = \frac{\partial w}{\partial v}\frac{\partial v}{\partial u}$$

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

$$\frac{\partial w}{\partial \mathbf{U}} = \frac{\partial w}{\partial \mathbf{v}}\frac{\partial \mathbf{v}}{\partial \mathbf{U}}$$

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

Расширим граф вычислений



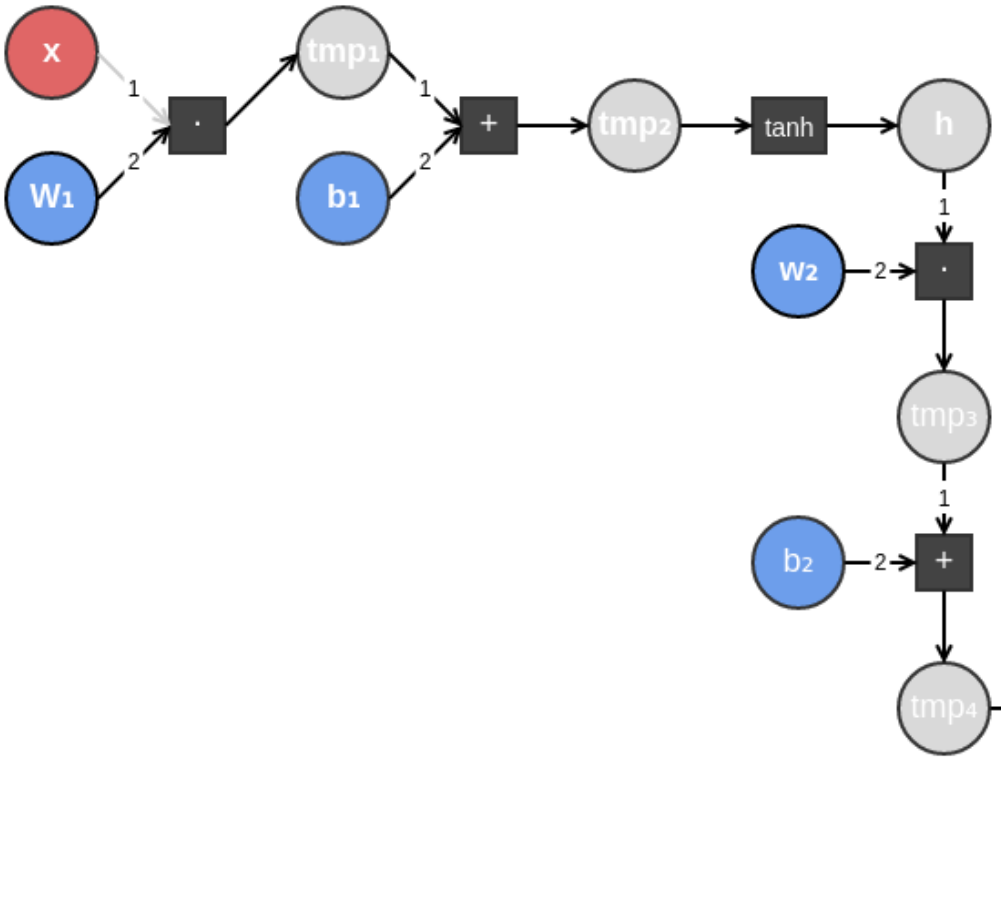
- *loss* – потеря на текущем элементе выборки

Backpropagation

- Наша цель – вычислить градиент *loss* по всем синим вершинам графа (настраиваемым параметрам нейросети), чтобы затем сделать шаг SGD
- Для этого будем идти от вершины *loss* к синим вершинам, вычисляя по цепному правилу градиенты *loss* по круглым вершинам на нашем пути

Backpropagation

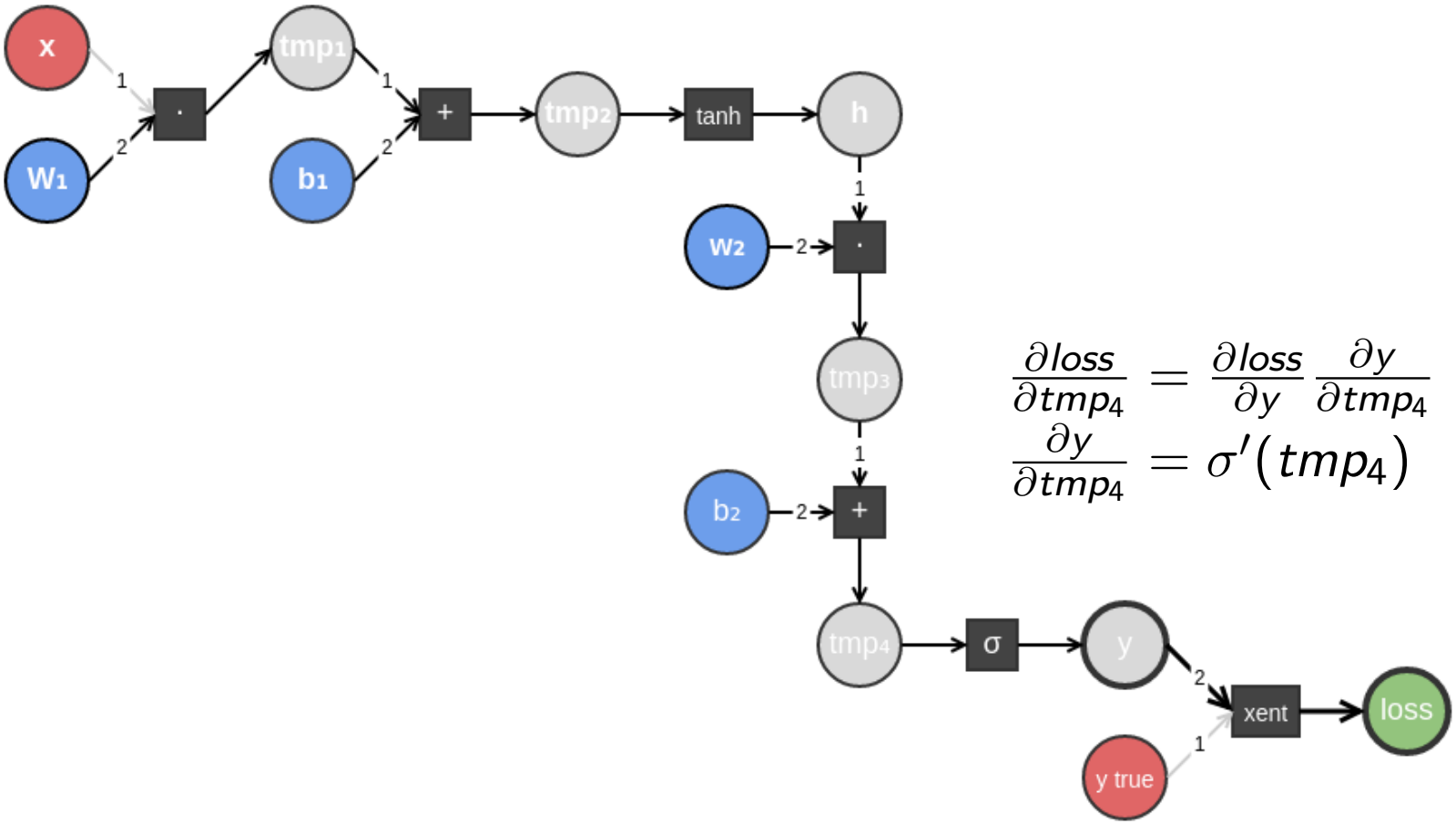
- $loss = -(y_{true} \log y + (1 - y_{true}) \log(1 - y))$



$$\frac{\partial loss}{\partial y} = \frac{1-y_{true}}{1-y} - \frac{y_{true}}{y}$$

Backpropagation

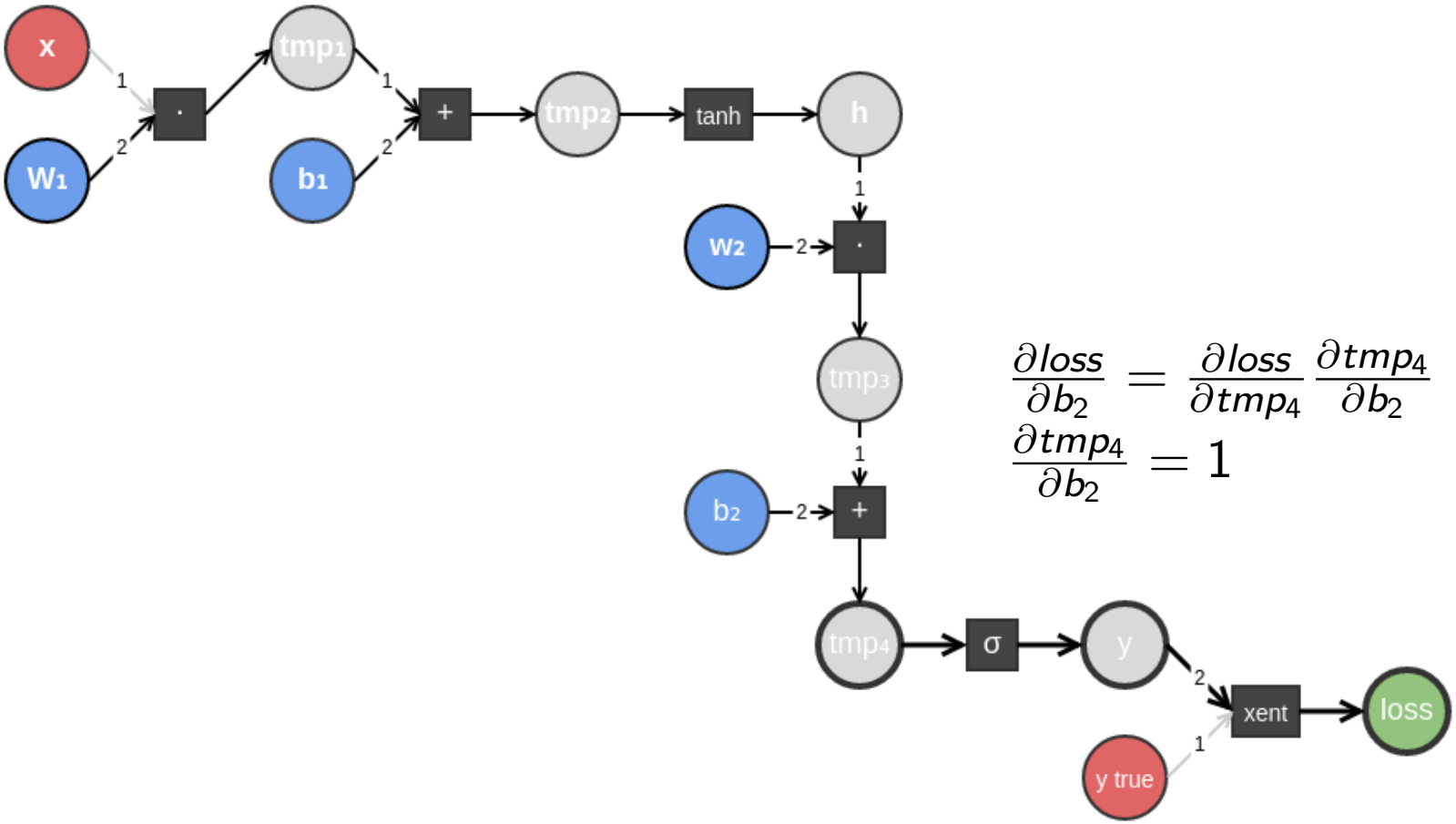
- $y = \sigma(tmp_4)$



$$\frac{\partial loss}{\partial tmp_4} = \frac{\partial loss}{\partial y}\frac{\partial y}{\partial tmp_4}$$
$$\frac{\partial y}{\partial tmp_4} = \sigma'(tmp_4)$$

Backpropagation

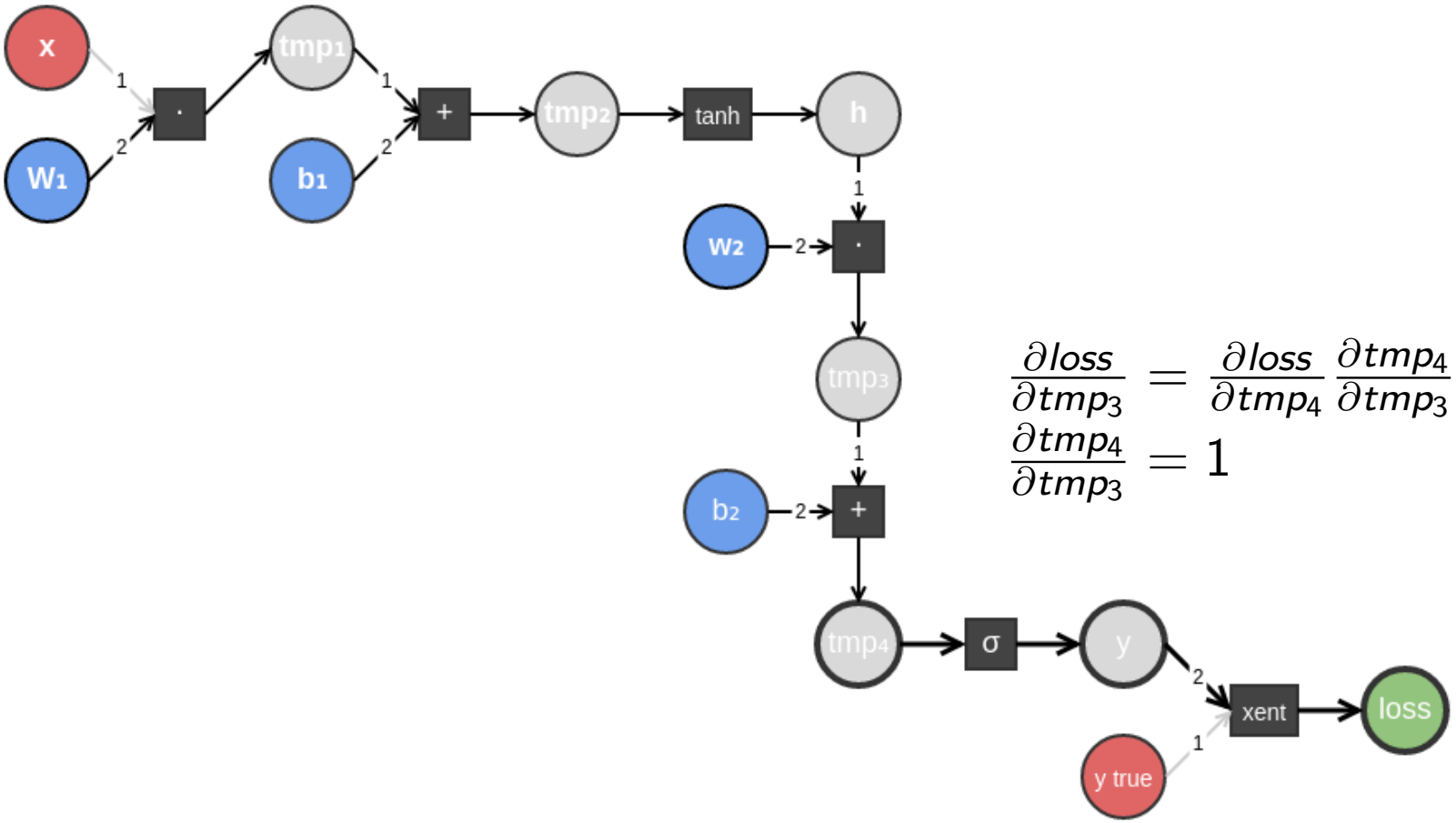
- $tmp_4 = tmp_3 + b_2$



$$\frac{\partial loss}{\partial b_2} = \frac{\partial loss}{\partial tmp_4}\frac{\partial tmp_4}{\partial b_2}$$
$$\frac{\partial tmp_4}{\partial b_2} = 1$$

Backpropagation

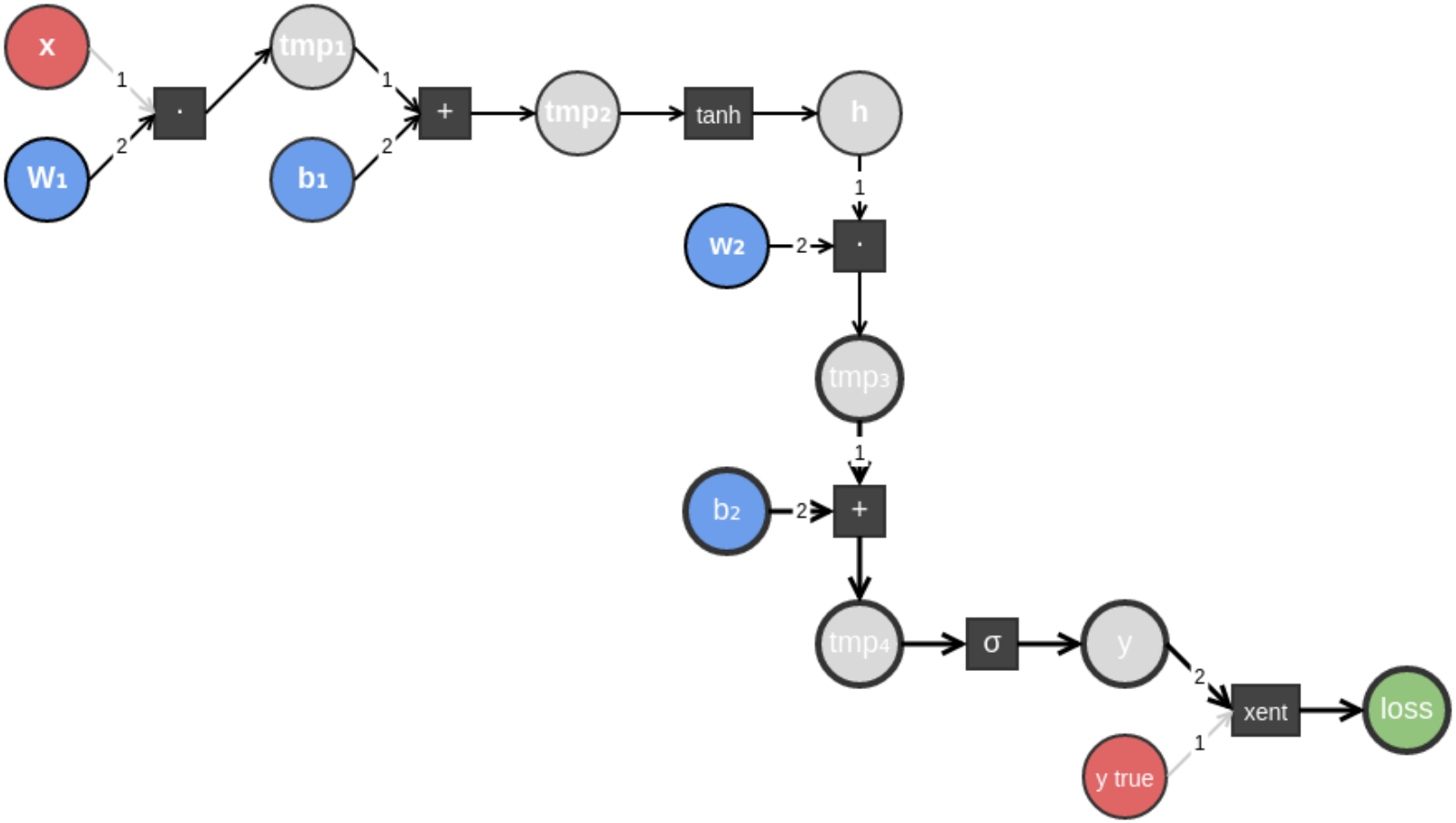
- $tmp_4 = tmp_3 + b_2$



$$\frac{\partial loss}{\partial tmp_3} = \frac{\partial loss}{\partial tmp_4}\frac{\partial tmp_4}{\partial tmp_3}$$
$$\frac{\partial tmp_4}{\partial tmp_3} = 1$$

Backpropagation

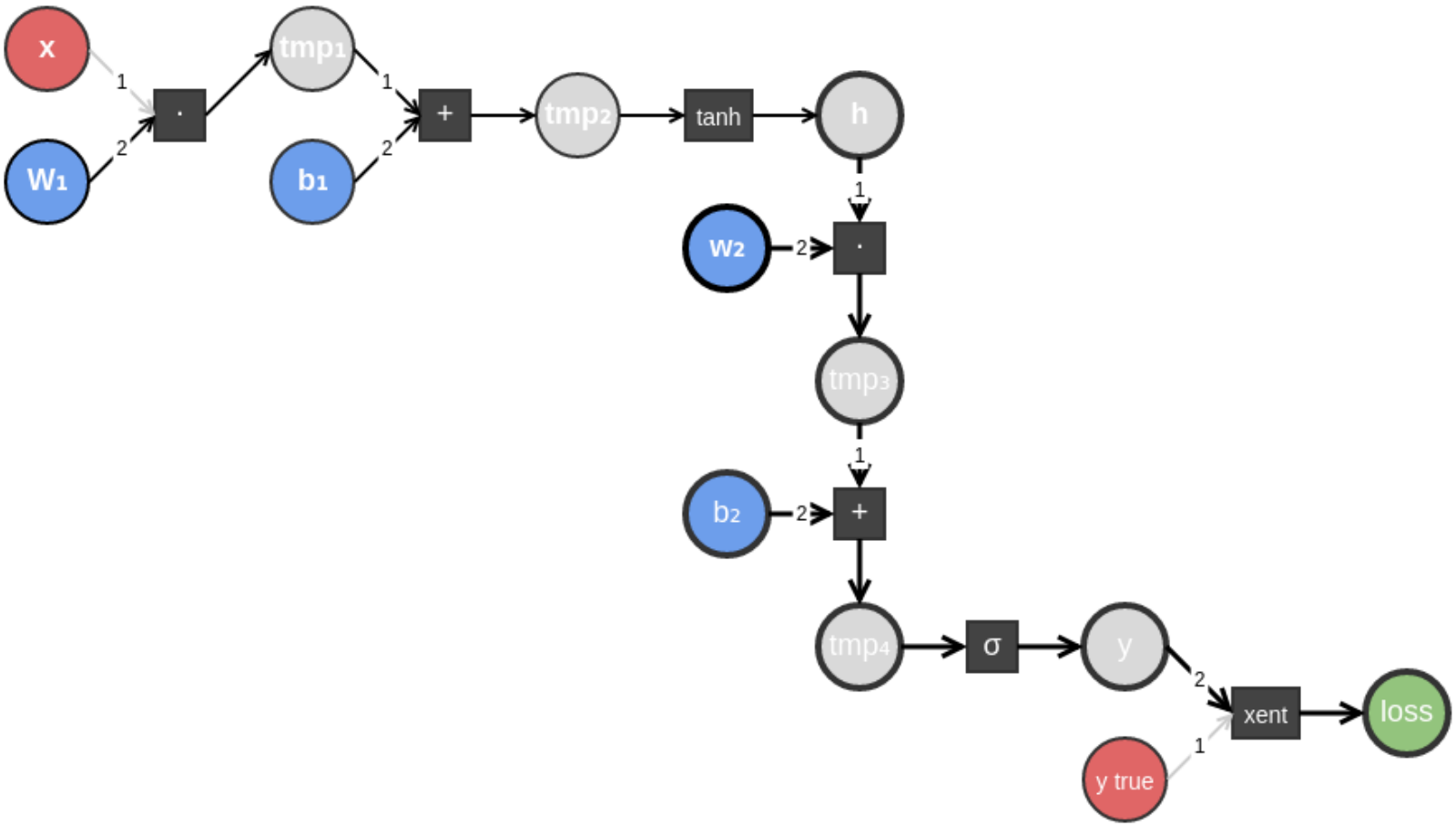
- И так далее



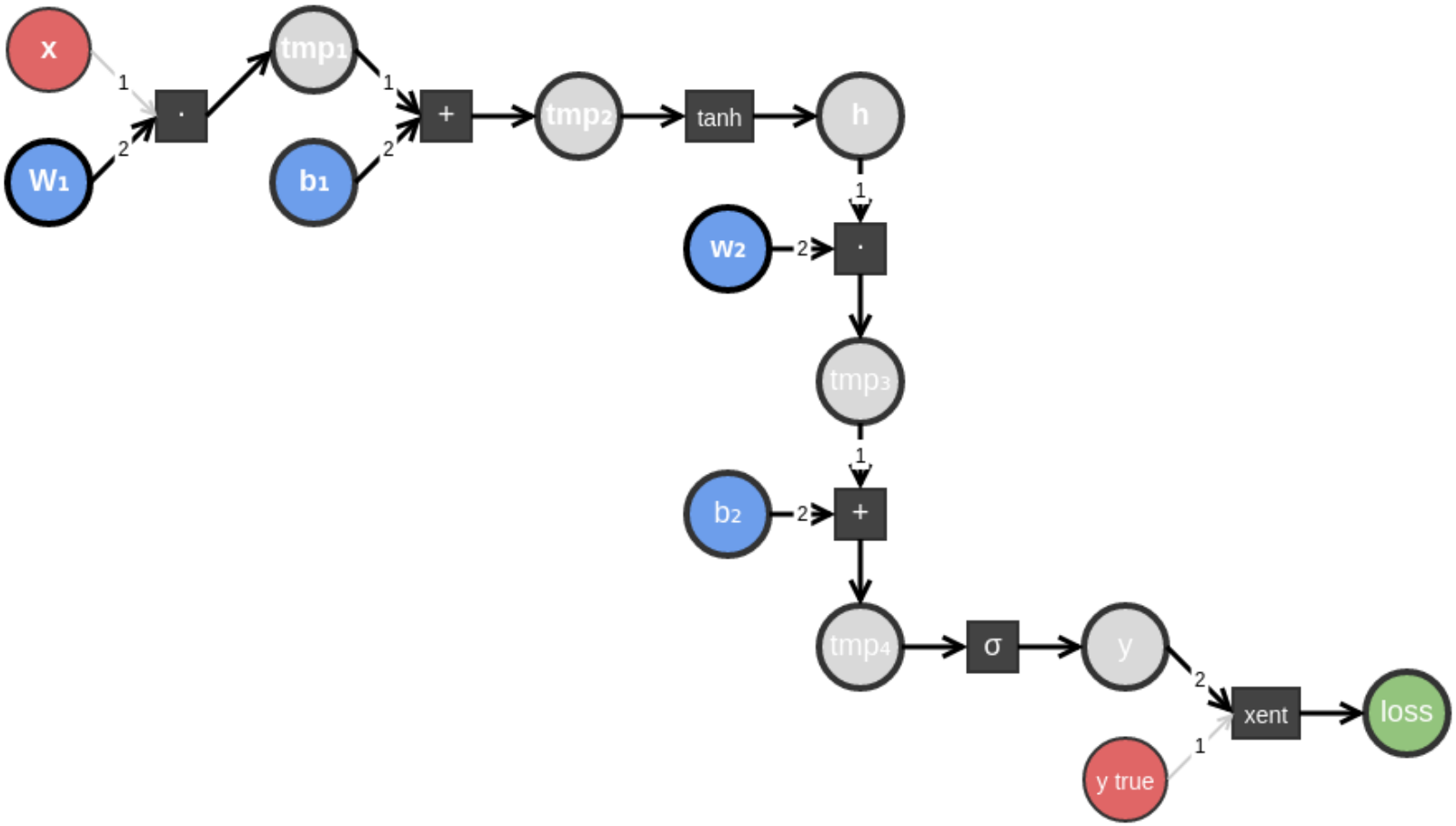
Как реализуется backpropagation

- Вершина tmp_3 знает градиент $\frac{\partial loss}{\partial tmp_3}$ и посылает эту информацию прямоугольнику-предку
- Прямоугольник (операция скалярного произведения) получает эту информацию. Он знает, как считать якобианы $\frac{\partial tmp_3}{\partial \mathbf{h}}$ и $\frac{\partial tmp_3}{\partial \mathbf{w_2}}$
- Но сами якобианы он никогда не считает
- Ему достаточно лишь умножать свой градиент $\frac{\partial loss}{\partial tmp_3}$ на эти якобианы, одновременно подставляя текущие значения $\mathbf{h}$ и $\mathbf{w_2}$
- Готовые градиенты $\frac{\partial loss}{\partial \mathbf{h}}$ и $\frac{\partial loss}{\partial \mathbf{w_2}}$ отправляются вершинам $\mathbf{h}$ и $\mathbf{w_2}$ соответственно

Backpropagation



Backpropagation



Обучение нейросети с мини-батчами

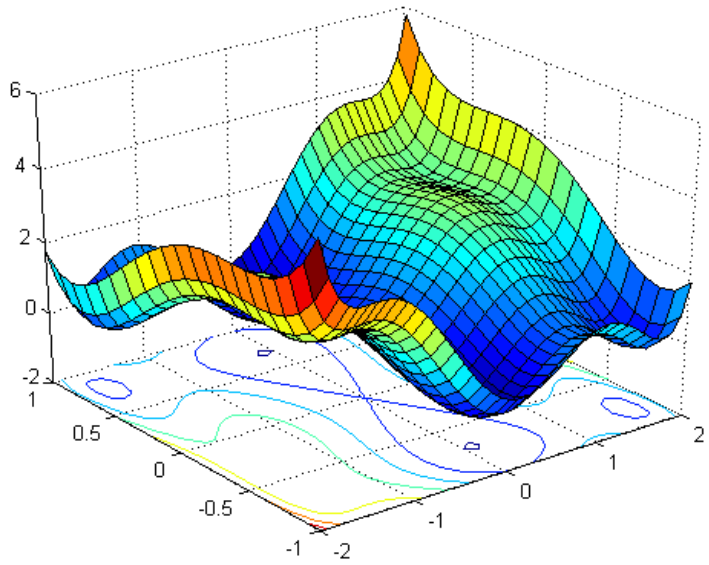
- Подаём на вход не вектор $\mathbf{x}$, а матрицу $\mathbf{X}$ с равным размеру мини-батча количеством строк
- $\mathbf{tmp_1}$, $\mathbf{tmp_2}$, $\mathbf{h}$ превращаются в матрицы
- tmp_3, tmp_4, y, y_{true}, $loss$ – в векторы
- Потери на элементах мини-батча нужно усреднить – добавляем зелёную вершину $avgLoss$ и минимизируем её
- В итоге мы распараллеливаем не только вычисление ответа нейросети, но и backpropagation

Обучение нейросети с мини-батчами

- На практике процесс обучения нейросети состоит из некоторого количества эпох
- Во время каждой эпохи обучающую выборку случайным образом разбивают на мини-батчи
- Для каждого мини-батча в рамках определённой эпохи вычисляют градиент потери на этом мини-батче и обновляют настраиваемые параметры
- Мини-батчи используются и при работе нейросети на тестовой выборке

Модификации SGD

- В отличие от логистической регрессии, обучение нейросети есть оптимизация очень сложной функции с большим количеством локальных минимумов
- Применять SGD становится трудно
- Рассмотрим некоторые модификации SGD, призванные облегчить поиск оптимальных параметров

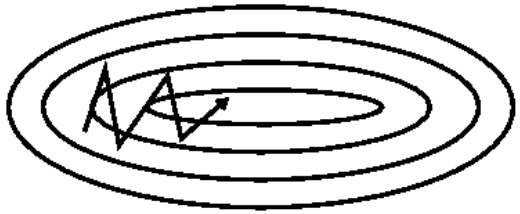


Модификации SGD: Момент

- Обозначим через θ совокупность настраиваемых параметров нейросети
- $\mathbf{v_t} = \gamma \mathbf{v_{t-1}} - \alpha_t \frac{\partial loss}{\partial \theta}|_{\theta=\theta_t}$
  $\theta_{t+1} = \theta_t + \mathbf{v_t}$
- $0 < \gamma < 1$
- По функции потерь катится мячик, набирая скорость на спусках и проскакивая неглубокие локальные минимумы
- Нестеров:
  $\mathbf{v_t} = \gamma \mathbf{v_{t-1}} - \alpha_t \frac{\partial loss}{\partial \theta}|_{\theta=\theta_t+\mathbf{v_{t-1}}}$
  $\theta_{t+1} = \theta_t + \mathbf{v_t}$

- Без момента:

- С моментом:



Модификации SGD: RMSProp

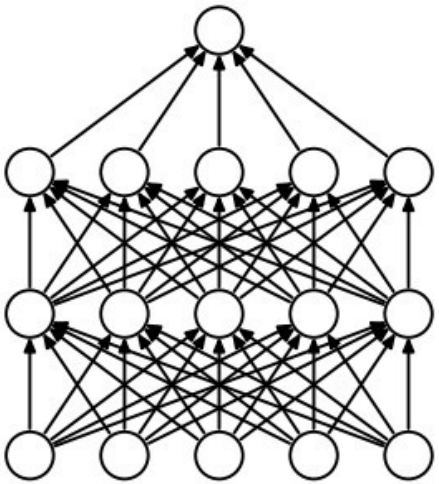
- $\mathbf{g_t} = \frac{\partial loss}{\partial \theta}|_{\theta=\theta_t}$
  $\mathbf{E[g^2]_t} = 0.9\mathbf{E[g^2]_{t-1}} + 0.1\mathbf{g_t^2}$
  $\boldsymbol{\alpha_t} = \frac{\eta}{\sqrt{\mathbf{E[g^2]_t} + \varepsilon}}$
  $\theta_{t+1} = \theta_t - \boldsymbol{\alpha_t} \odot \mathbf{g_t}$
- $\odot$ – поэлементное умножение
- Поддерживает примерно одинаковую скорость
- Помогает выбираться из узких глубоких локальных минимумов
- Требует дополнительной памяти
- См. также AdaDelta, AdaGrad, Adam

Dropout

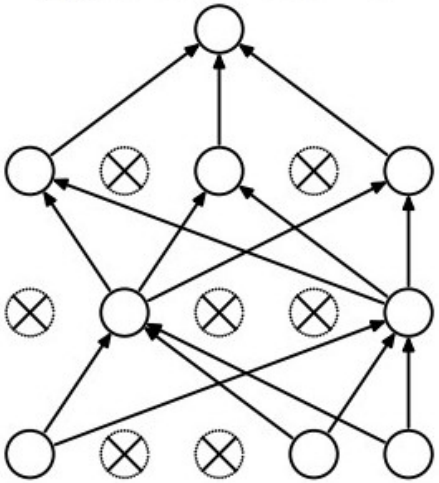
- Каждый элемент тензора обнуляется с вероятностью p либо умножается на $\frac{1}{p}$ с вероятностью $1 - p$
- Пример:

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

- Помогает не переобучаться
- При работе нейросети на тестовой выборке dropout выключают



(a) Standard Neural Net

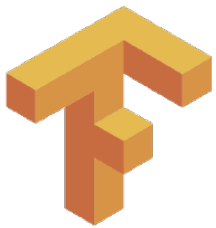


(b) After applying dropout.

Содержание

Библиотеки глубокого обучения

- Theano – низкоуровневая; строит и оптимизирует графы вычислений, считает в них градиенты; остальное пользователь делает сам
- TensorFlow
- Keras – высокоуровневая; использует Theano или TensorFlow в качестве backend; простая в освоении; по сравнению с Theano обладает меньшей гибкостью



Keras: пример

- Классифицируем новости на 46 тем
- Пусть есть словарь из `max_words` слов
- Каждую новость представим в виде бинарного вектора из `max_words` элементов (присутствует ли слово в тексте новости)
- Разобьём новости на обучающую и тестовую части; построим соответствующие матрицы `X_train` и `X_test` из вышеупомянутых бинарных векторов

Keras: reuters_mlp.py

```
model = Sequential()
model.add(Dense(512, input_shape=(max_words,)))
model.add(Activation("relu"))
model.add(Dropout(0.5))
model.add(Dense(46))
model.add(Activation("softmax"))

model.compile(loss="categorical_crossentropy",
              optimizer="adam",
              metrics=["accuracy"])

history = model.fit(X_train, Y_train,
                    nb_epoch=5, batch_size=32,
                    verbose=1, validation_split=0.1)
score = model.evaluate(X_test, Y_test,
                       batch_size=32, verbose=1)
```

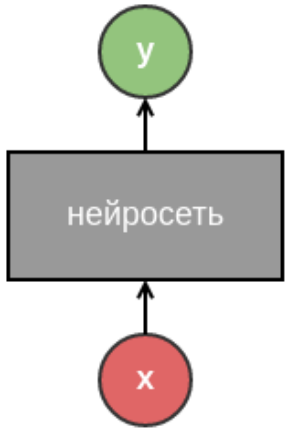
Советы

- Для задач классификации используйте кросс-энтропию
- Или hinge loss, если кажется, что максимизация правдоподобия является не лучшей идеей. При этом последний слой нейросети следует делать линейным
- Не забывайте использовать `class_weight` в случае несбалансированных классов
- Для задач регрессии чаще всего используют среднеквадратичную ошибку (MSE) и линейный последний слой
- Используйте модификации SGD и dropout
- Часто полезен ранний останов (early stopping) – обучение прекращается, если значение функции потерь на валидационной выборке перестаёт уменьшаться

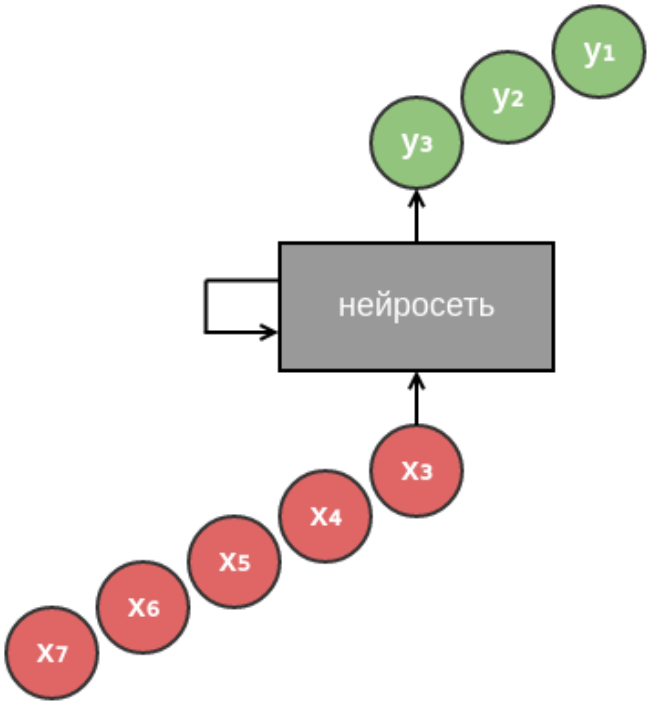
Содержание

Нейронная сеть прямого распространения

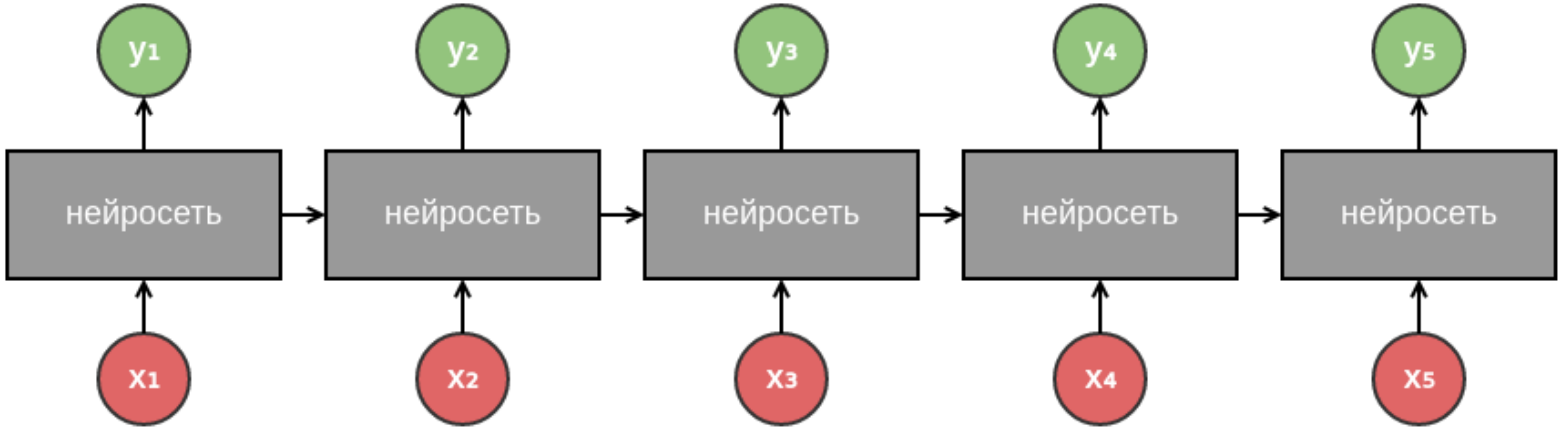
- Вход имеет фиксированное число элементов; то же самое для выхода
- А как быть с последовательностями переменной длины, например, с предложениями на русском языке, в которых имеются тесные взаимосвязи между соседними словами?



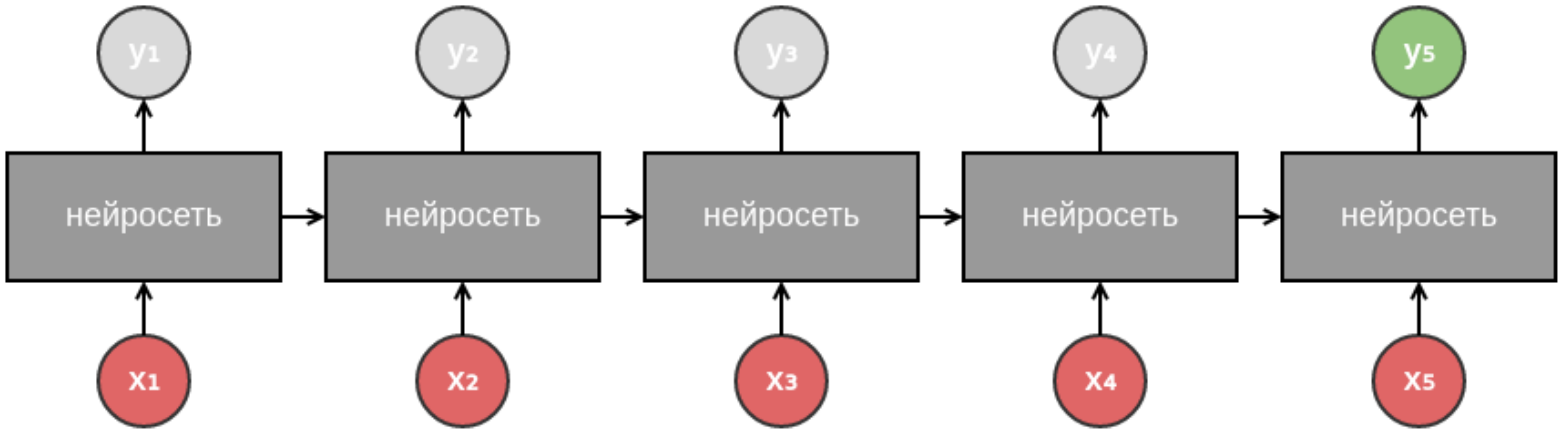
Рекуррентная нейронная сеть



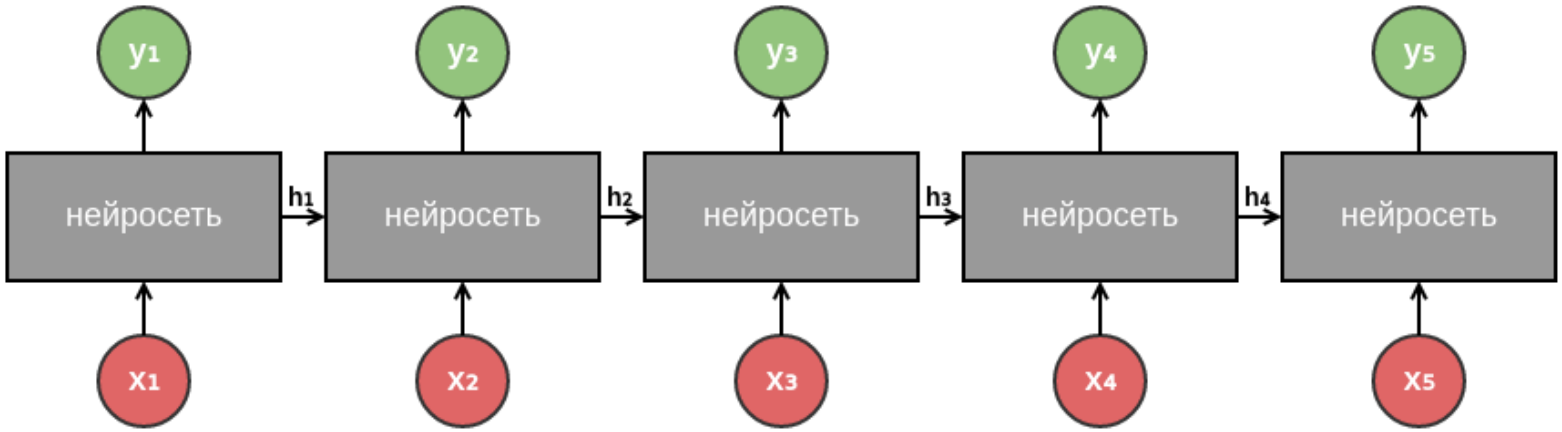
Представленная в другом виде



Если нас интересует только последний выход



Рекуррентная нейронная сеть Элмана



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

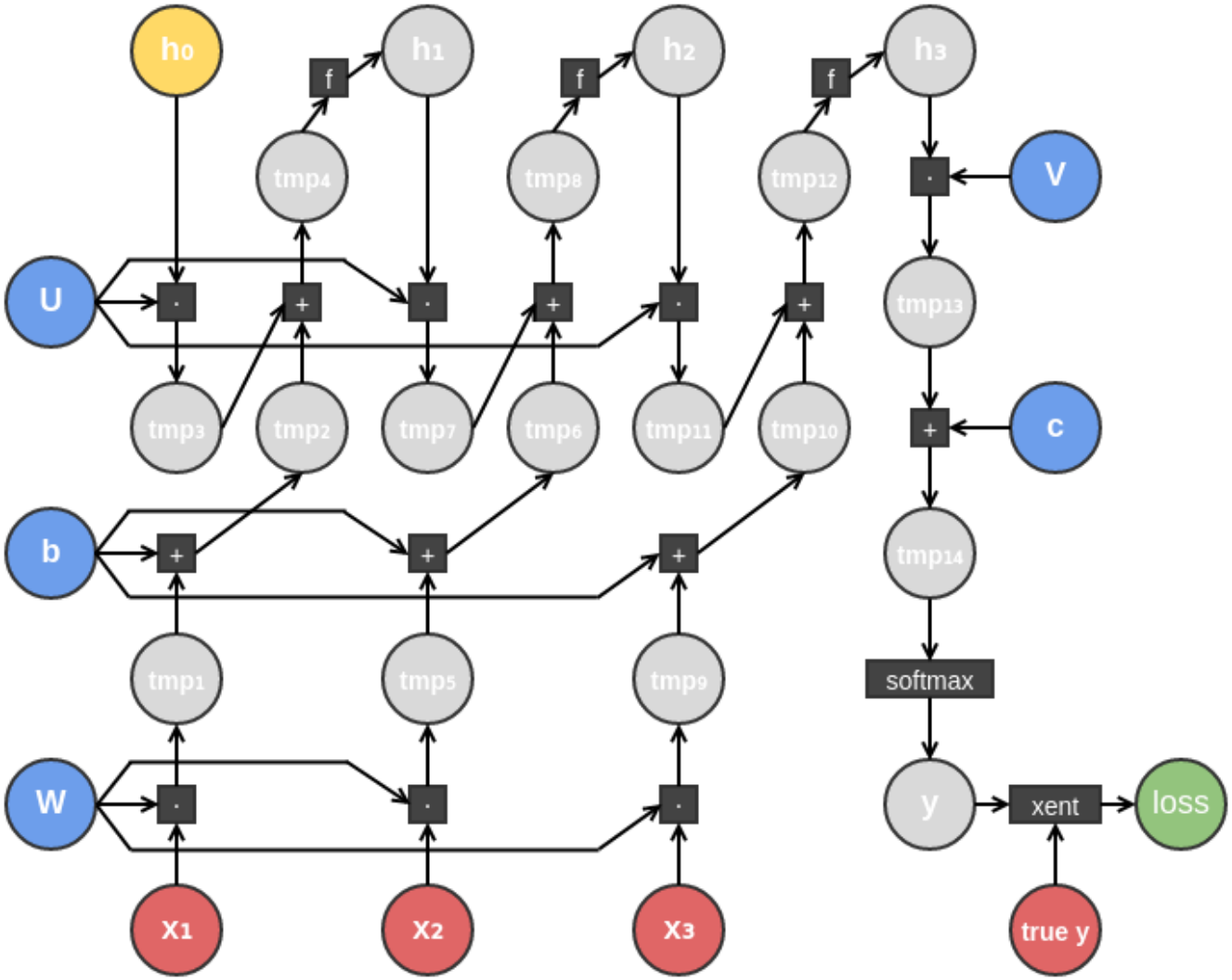
Задача анализа тональности твитов

- Классифицируем твиты на три класса – негативные, нейтральные и позитивные
- Возьмём обученный на соцсетях word2vec; будем представлять твит в виде последовательности $(\mathbf{x_t})_{t=1}^{T}$ векторов его слов
- Решим задачу с помощью рекуррентной нейросети Элмана:

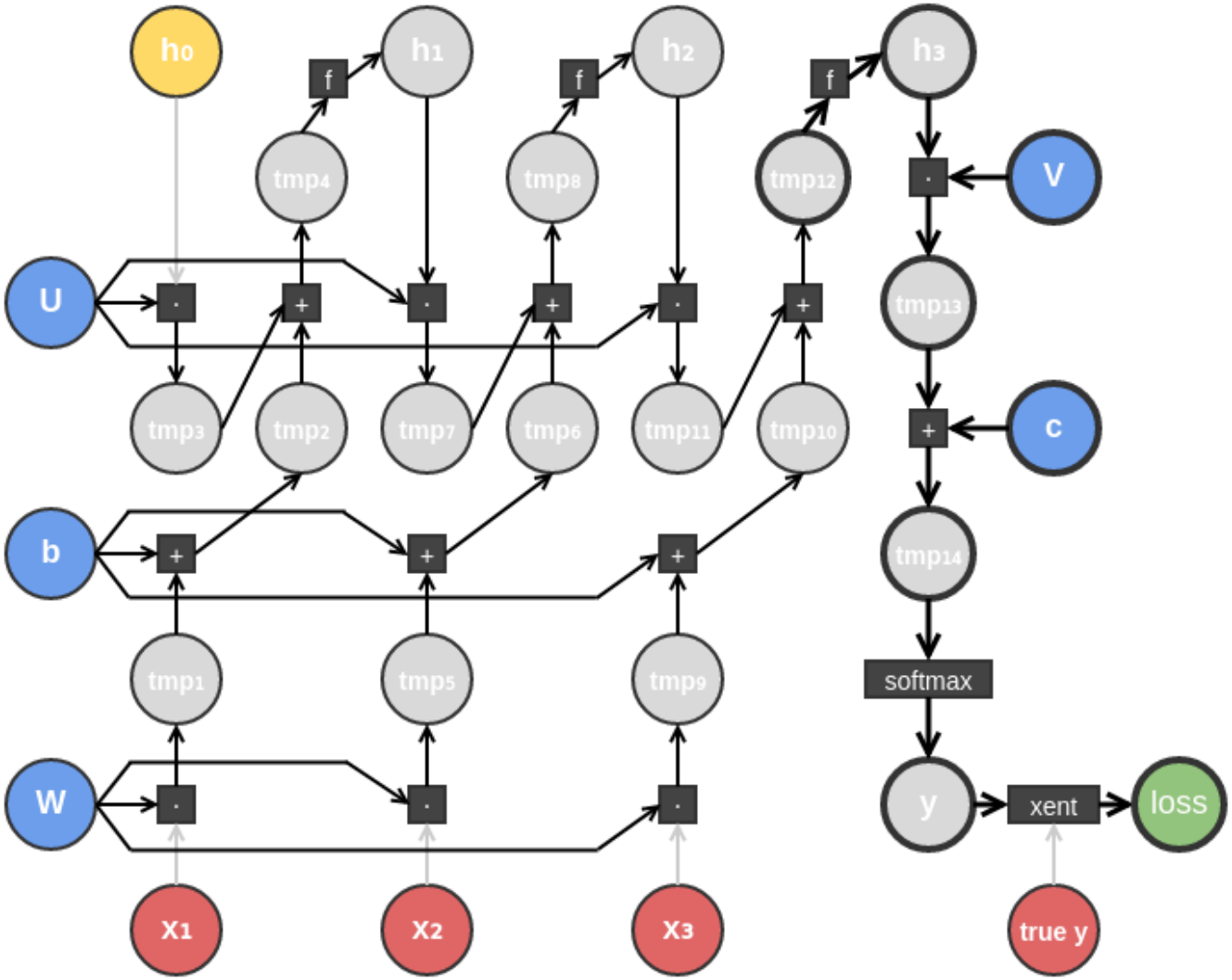
$$\mathbf{h_t} = f(\mathbf{x_t}\mathbf{W} + \mathbf{h_{t-1}}\mathbf{U} + \mathbf{b})$$
$$\mathbf{y} = \text{softmax}(\mathbf{h_T}\mathbf{V} + \mathbf{c})$$
$$\mathbf{h_0} = \mathbf{0}$$

- Используем categoricalCrossentropy

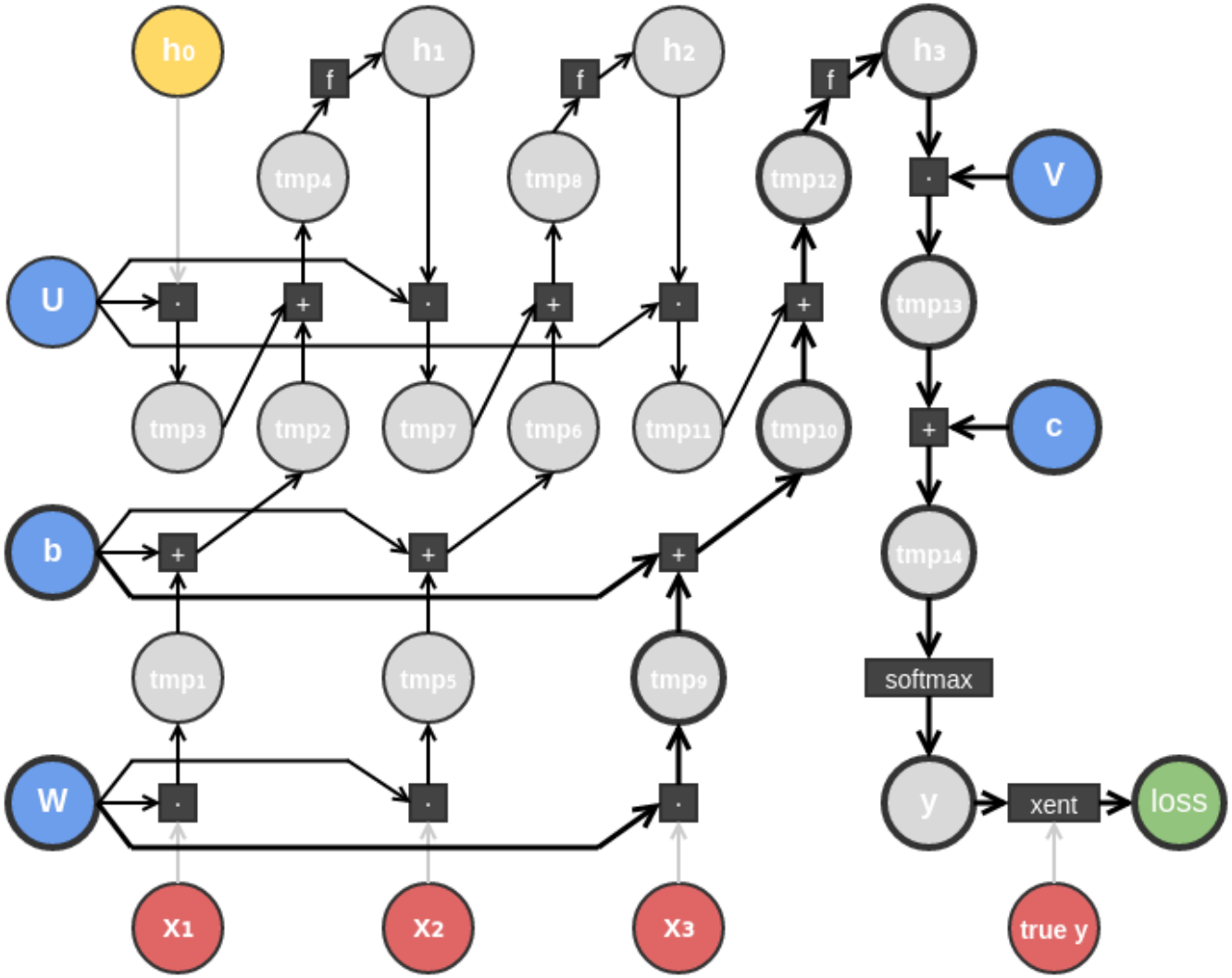
Вычисление градиента в рекуррентной нейросети



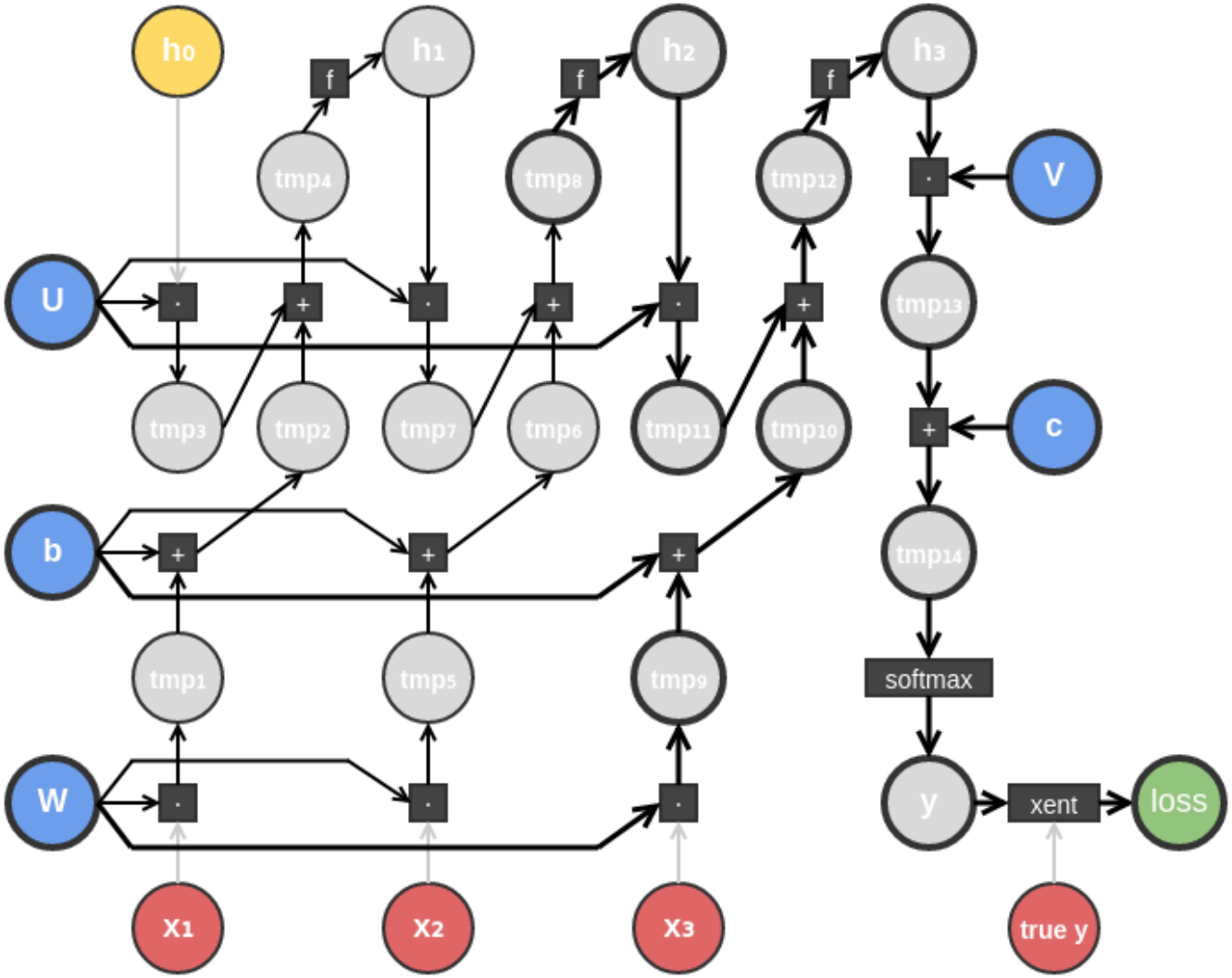
Вычисление градиента в рекуррентной нейросети



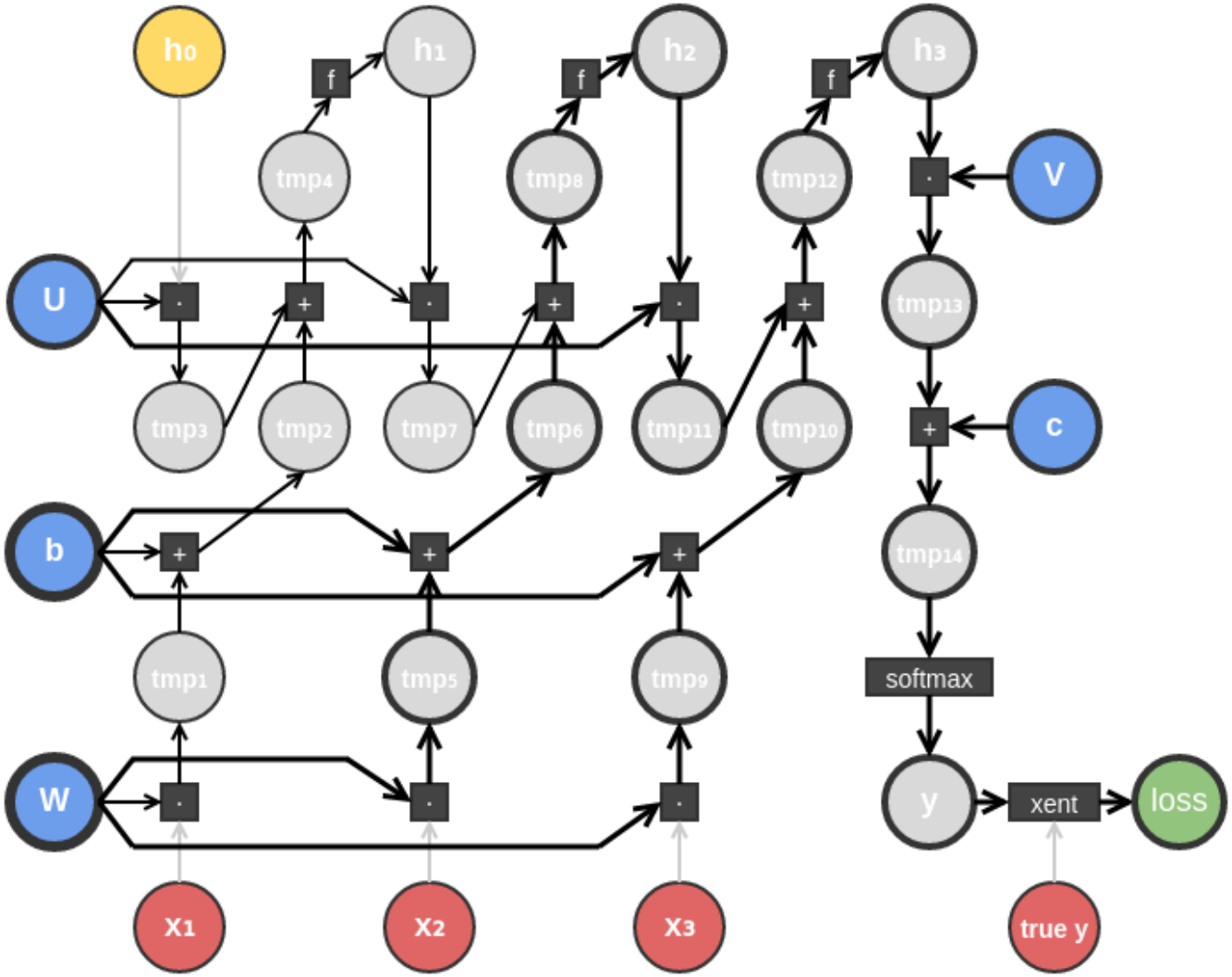
Вычисление градиента в рекуррентной нейросети



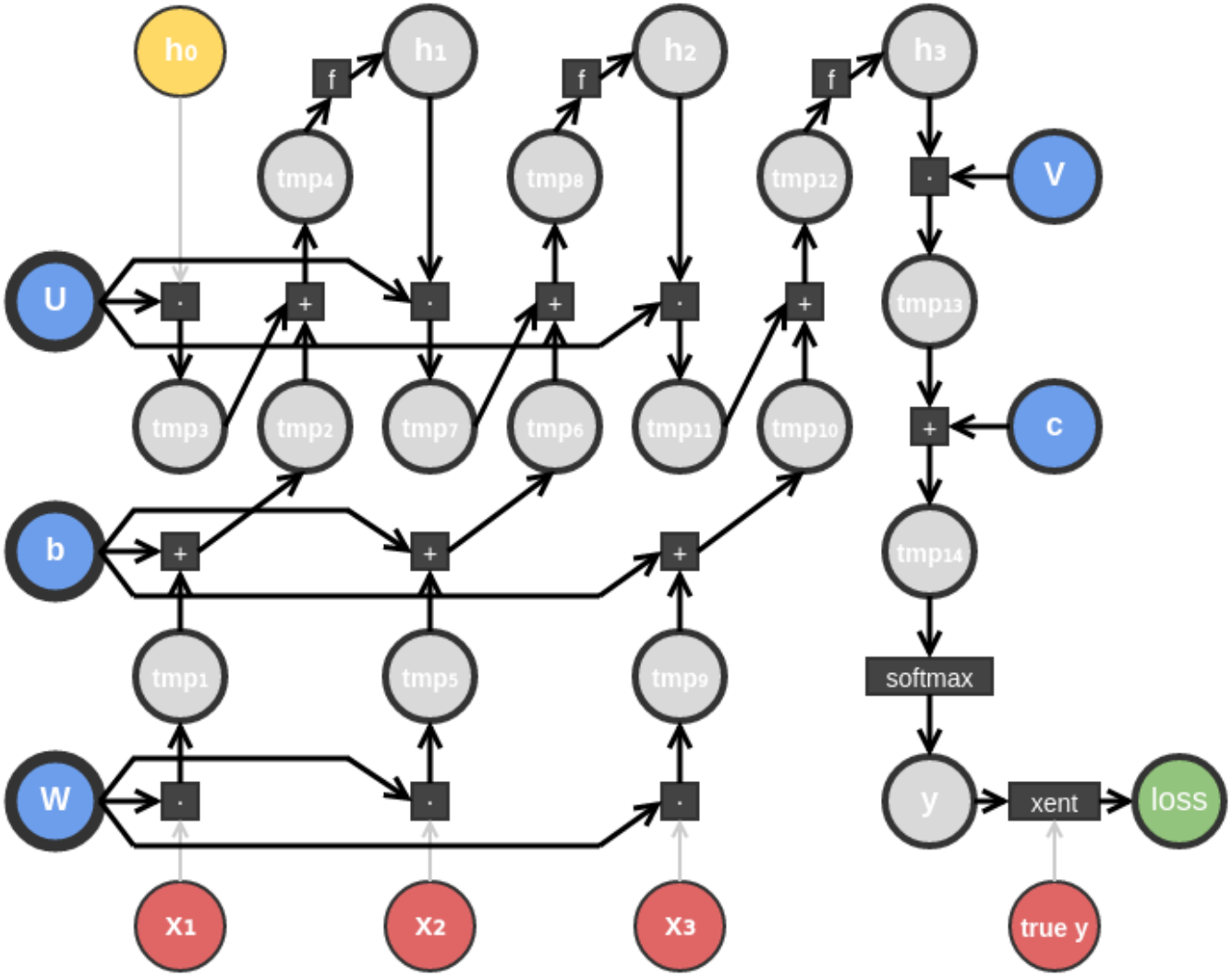
Вычисление градиента в рекуррентной нейросети



Вычисление градиента в рекуррентной нейросети



Вычисление градиента в рекуррентной нейросети



Вычисление градиента в рекуррентной нейросети

- Вершина **W** получает части своего градиента от трёх вершин – $\mathbf{tmp_1}$, $\mathbf{tmp_5}$ и $\mathbf{tmp_9}$; эти части суммируются
- Аналогично для вершин **U** и **b**

Рекуррентная нейросеть с мини-батчами

- Проблема: признаки элементов выборки имеют разную длину
- Чтобы сформировать мини-батч, придётся дополнить нулевыми векторами признаки его элементов так, чтобы они имели одинаковую длину
- Keras поддерживает **маскирование** – при достижении таких нулевых векторов прекращается обновление скрытого состояния и выхода
- В Theano матрицу-маску можно создать самому
- Совокупность признаков элементов мини-батча, подаваемая на вход нейросети, есть 3-мерный тензор

Содержание

Затухающий градиент

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

$$\frac{\partial \mathbf{h_5}}{\partial \mathbf{x_5}} = diag(f_h'(\mathbf{x_5}\mathbf{W} + \mathbf{h_4}\mathbf{U} + \mathbf{b}))\mathbf{W^T}$$

$$\frac{\partial \mathbf{h_5}}{\partial \mathbf{x_2}} = \frac{\partial \mathbf{h_5}}{\partial \mathbf{h_4}} \frac{\partial \mathbf{h_4}}{\partial \mathbf{h_3}} \frac{\partial \mathbf{h_3}}{\partial \mathbf{h_2}} diag(f_h'(\mathbf{x_2}\mathbf{W} + \mathbf{h_1}\mathbf{U} + \mathbf{b}))\mathbf{W^T}$$

Затухающий градиент

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

$$\frac{\partial \mathbf{h_t}}{\partial \mathbf{h_{t-1}}} = diag(f_h'(\mathbf{x_t}\mathbf{W} + \mathbf{h_{t-1}}\mathbf{U} + \mathbf{b}))\mathbf{U^T}$$

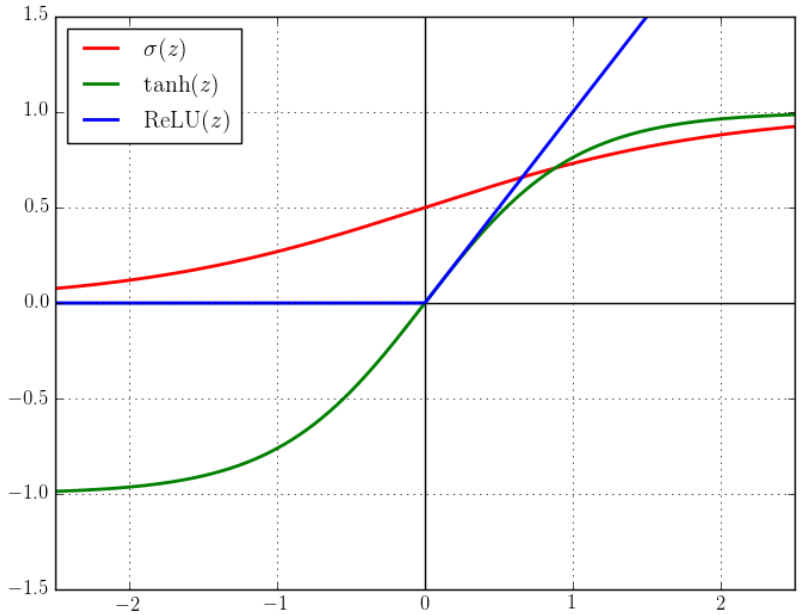
- Если $|f_h'(\cdot)| < \gamma$ и $\|\mathbf{U}\| < \frac{1}{\gamma}$, то элементы матрицы $\frac{\partial \mathbf{h_5}}{\partial \mathbf{x_2}}$ будут близки к нулю
- Нейросеть начнёт страдать маразмом: на её ответ в момент времени $t = 5$ то, что подавалось на вход при $t = 2$, уже почти не влияет

Затухающий градиент

- Проблема затухающего градиента проявляется и в многослойных персептронах
- Для её избежания нужно грамотно выбирать функции активации и использовать правильную инициализацию настраиваемых параметров нейросети
- Вместо нейросети Элмана следует использовать менее подверженные этой проблеме рекуррентные нейросети
- Вместо затухания градиент может и взрываться

Функции активации

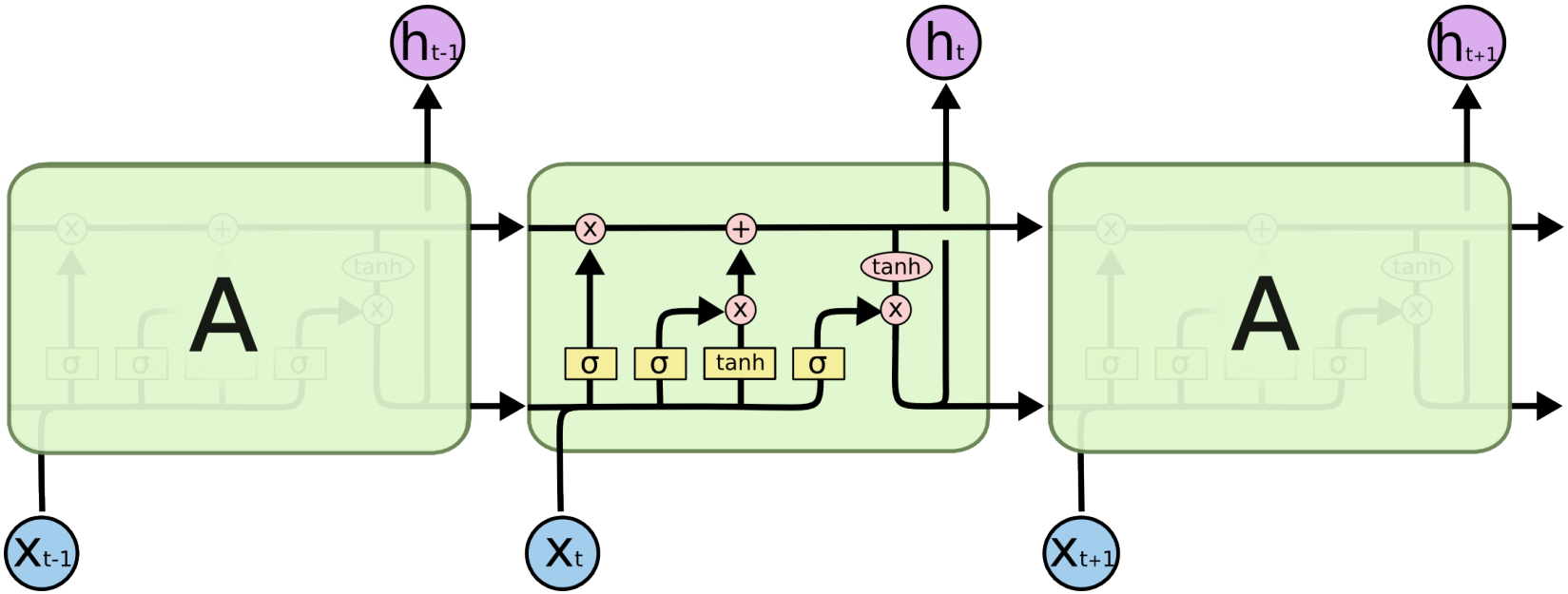
- $\sigma(z)$, чья производная ограничена $\frac{1}{4}$, больше всех провоцирует затухание градиента



Инициализация

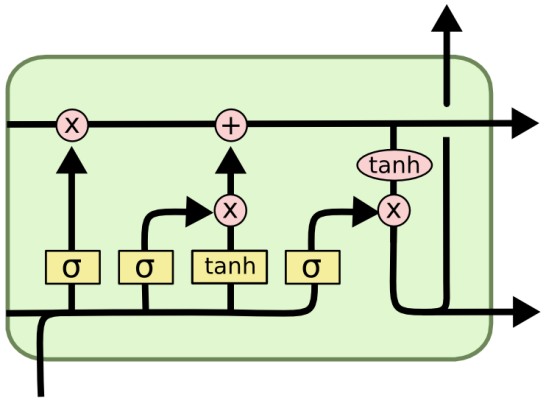
- В случае обычного слоя $\mathbf{y} = f(\mathbf{x}\mathbf{W} + \mathbf{b})$ для инициализации $\mathbf{W}$ чаще всего используется равномерное распределение (например, Glorot uniform)
- Рекуррентные матрицы ($\mathbf{U}$) любят инициализировать случайными ортогональными матрицами, уменьшая вероятность затухания или взрыва градиента
- Смещения ($\mathbf{b}$) инициализируются нулями, за редкими исключениями

Long Short-Term Memory



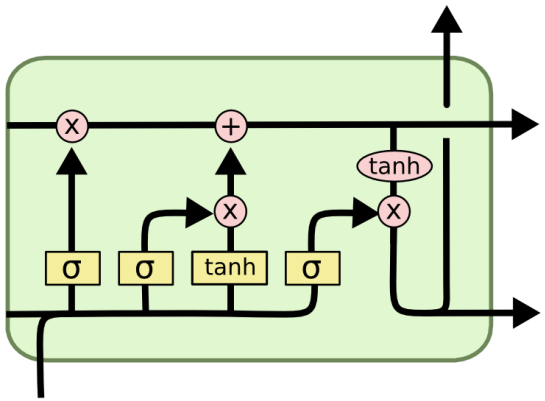
Long Short-Term Memory

- $\mathbf{f_t} = \sigma(\mathbf{x_t W_f} + \mathbf{h_{t-1} U_f} + \mathbf{b_f})$
  $\mathbf{i_t} = \sigma(\mathbf{x_t W_i} + \mathbf{h_{t-1} U_i} + \mathbf{b_i})$
  $\tilde{\mathbf{c}}_\mathbf{t} = \tanh(\mathbf{x_t W_c} + \mathbf{h_{t-1} U_c} + \mathbf{b_c})$
  $\mathbf{o_t} = \sigma(\mathbf{x_t W_o} + \mathbf{h_{t-1} U_o} + \mathbf{b_o})$
  $\mathbf{c_t} = \mathbf{f_t} \odot \mathbf{c_{t-1}} + \mathbf{i_t} \odot \tilde{\mathbf{c}}_\mathbf{t}$
  $\mathbf{h_t} = \mathbf{o_t} \odot \tanh(\mathbf{c_t})$
- LSTM обладает дополнительным скрытым состоянием $\mathbf{c}$



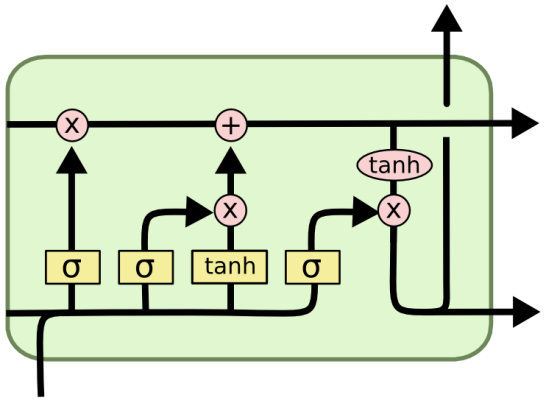
Long Short-Term Memory

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



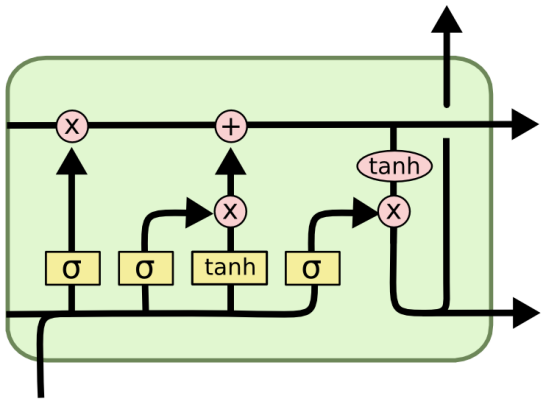
Long Short-Term Memory

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



Long Short-Term Memory

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



LSTM и проблема затухающего градиента

$$\frac{\partial \mathbf{c_t}}{\partial \mathbf{c_{t-1}}} = diag(\mathbf{f_t})$$

- Для тех моментов времени, когда нейросеть не хотела забывать (значение forget gate почти равно единице), градиент почти не затухает
- Наша нейросеть больше не страдает маразмом и может запоминать важную информацию на много шагов во времени
- Смещение $\mathbf{b_f}$ forget gate желательно инициализировать не нулевым вектором, а, например, вектором из единиц

Содержание

Keras: анализ тональности

- Классифицируем рецензии на IMDb на негативные и позитивные
- Рецензию представляем в виде последовательности индексов её слов в словаре, который имеет размер `max_features`
- Индексы в словаре начинаются с единицы; ноль будем использовать для выравнивания

Keras: imdb_lstm.py

```
X_train = sequence.pad_sequences(X_train, maxlen=80)
X_test = sequence.pad_sequences(X_test, maxlen=80)

model = Sequential()
model.add(Embedding(max_features, 128, dropout=0.2))
model.add(LSTM(128, dropout_W=0.2, dropout_U=0.2))
model.add(Dense(1))
model.add(Activation("sigmoid"))

model.compile(loss="binary_crossentropy", optimizer="adam",
              metrics=["accuracy"])

model.fit(X_train, y_train, batch_size=32,
          nb_epoch=15, validation_data=(X_test, y_test))
score, acc = model.evaluate(X_test, y_test, batch_size=32)
```

Слой Embedding

- Специальный слой для перевода индексов слов в словаре в векторные представления этих слов
- Экономит память
- Можно инициализировать векторами word2vec и сделать необучаемым

Приложения рекуррентных нейронных сетей

- Анализ тональности (sentiment analysis)
- Моделирование языка, генерация текста
- Машинный перевод
- Определение частей речи и синтаксический анализ
- Чат-боты (см. Luka)
- Распознавание и генерация рукописного текста
- Генерация текстовых описаний изображений и видео
- ...

Ключевые слова для гуглежа

keras
keras examples
word2vec
gated recurrent unit
colah's blog

deep learning book
awesome rnn
theano
tensorflow
sebastian ruder
wildml

attention mechanism
batch normalization
layer normalization
awesome tensorflow
leavesbreathe tensorflow