ELBRUS / BOOT CAMP

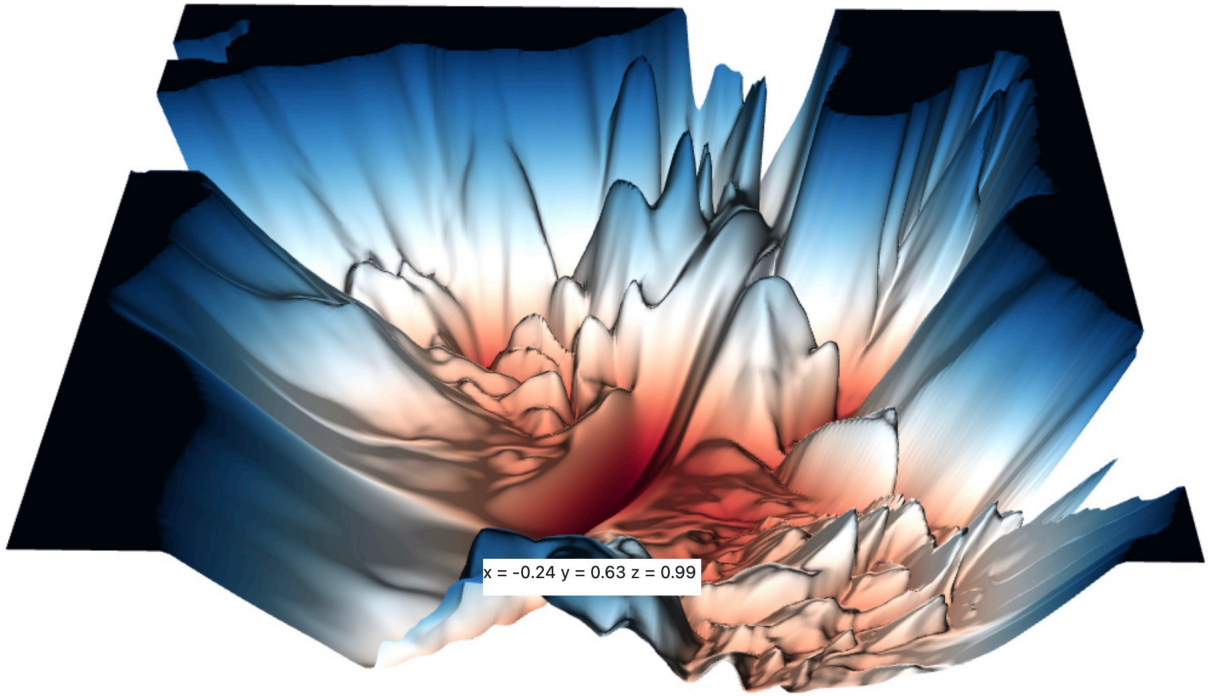
Оптимизация нейронных сетей • Optimizers

Вчера

Эльбрус буткемп

- интуиция нейронных сетей
- базовая архитектура
- функции активации
- функции потерь

Сегодня: оптимизация



> Оптимизация – процесс поиска экстремума (т.е. минимального или максимального значения функции).

Оптимизация функций

Эльбрус буткемп

1. Как правило, в машинном обучении функцию **минимизируют**
    - почему?

Оптимизация функций

Эльбрус буткемп

1. Как правило, в машинном обучении функцию **минимизируют**
    - почему?
2. Один из подходов для решения задачи оптимизации – градиентный спуск.

Градиентный спуск

Эльбрус буткемп

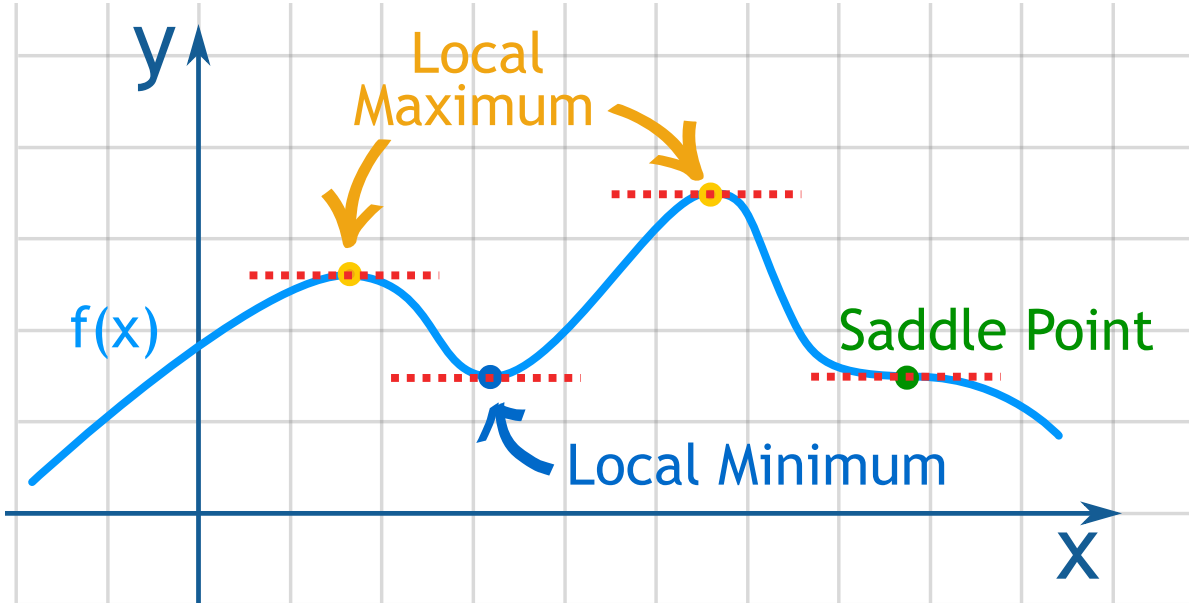
Сформулировать его можно так:

> Будем из текущих значений параметров вычитать произведение частной производной и learning rate до тех пор, пока не придем в какой-нибудь минимум

Градиентный спуск: проблемы

Эльбрус буткемп

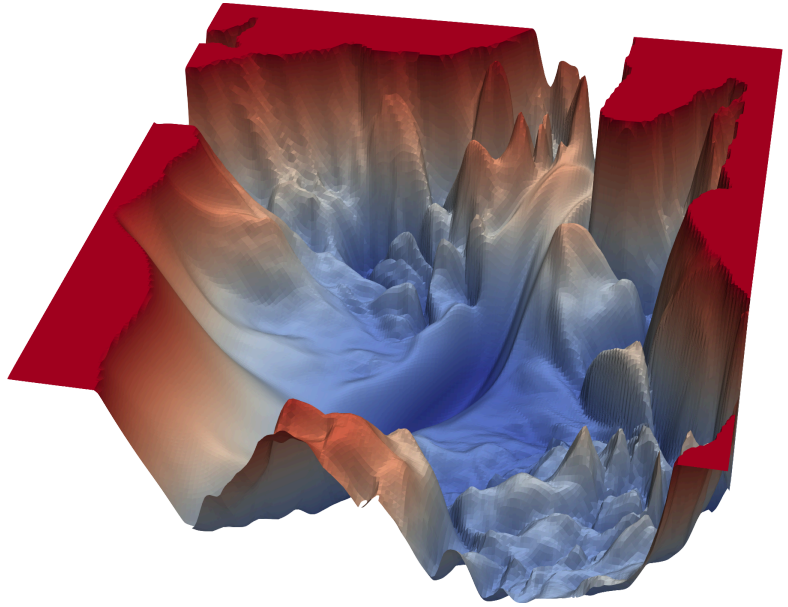
Он застревает в локальных минимумах и седловых точках



Градиентный спуск: проблемы

Эльбрус буткемп

Поверхность функции ошибки может быть очень сложной: плато, резкие «обрывы» и пр.

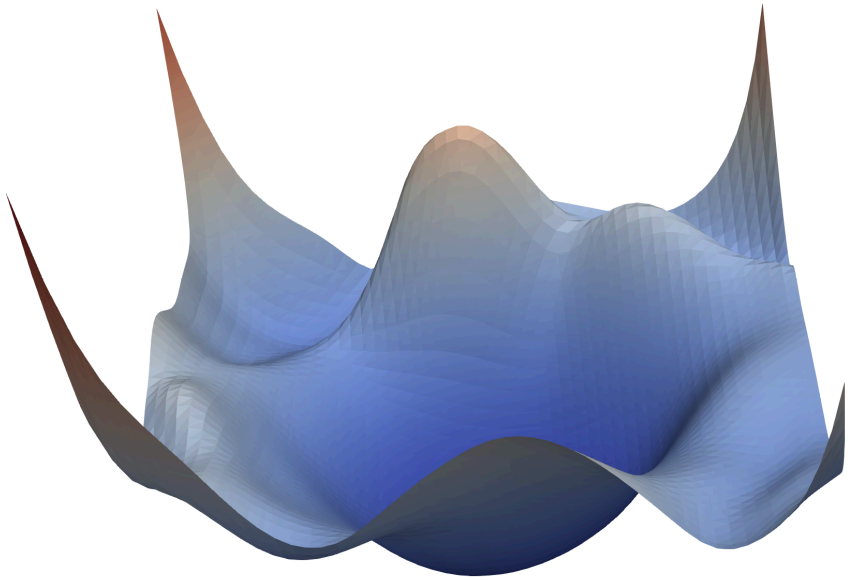


©Visualizing the Loss Landscape of Neural Nets

Градиентный спуск: проблемы

Эльбрус буткемп

Сложно подобрать скорость обучения: слишком большой lr может заставить алгоритм расходиться (или просто перепрыгнуть минимум), слишком маленький замедлит процесс оптимизации



©Visualizing the Loss Landscape of Neural Nets

Градиентный спуск

Эльбрус буткемп

Чтобы избежать перечисленных (и еще нескольких проблем), используют разные трюки и модификации.

- Nesterov Accelerated Gradient
- Adagrad
- RMSProp / Adadelta
- Adam
- в библиотеке `torch`: `'ASGD', 'Adadelta', 'Adagrad', 'Adam', 'AdamW', 'Adamax', 'LBFGS', 'NAdam', 'Optimizer', 'RAdam', 'RMSprop', 'Rprop', 'SGD', 'SparseAdam'`

Градиентный спуск · Общая постановка

Эльбрус буткемп

$$w_{t+1} = w_t - \lambda \nabla_{w_t} L(w_t)$$

- w_i - текущее значение параметров
- λ - learning rate
- $\nabla_{w_t} L(w_t)$ - производные по параметрам w
- t - текущий шаг итерации

```
for t in range(steps):
  dw = gradient(loss, data, w)
  w = w - lr * dw
```

Batch gradient descent

Эльбрус буткемп

- Но объектов для предсказания несколько (десятков, сотен, тысяч), на них на всех мы ошибаемся, а значит, каждый из объектов вносит некоторый вклад в вычисление производной.
- Тогда будем считать градиенты для всех объектов в выборке и просто усреднять:

$$\nabla_{w_{t+1}} L(x_1, y_1; w_t^1)$$
$$\nabla_{w_{t+1}} L(x_2, y_2; w_t^2)$$
$$\dots$$
$$\nabla_{w_{t+1}} L(x_N, y_N; w_t^N)$$

$$\Longrightarrow \nabla_{w_{t+1}} = \frac{1}{N} \sum_{i=1}^{N} \nabla_{w_t} L(x_i, y_i; w_t)$$

Batch gradient descent

Эльбрус буткемп

- Но объектов для предсказания несколько (десятков, сотен, тысяч), на них на всех мы ошибаемся, а значит, каждый из объектов вносит некоторый вклад в вычисление производной.
- Тогда будем считать градиенты для всех объектов в выборке и просто усреднять:

$$\nabla_{w_{t+1}} = \frac{1}{N}\sum_{i=1}^{N}\nabla_{w_t} L(x_i, y_i; w_t)$$

- Но если объектов в выборке очень много (N слишком велико), то не хватит памяти.

Stochastic gradient descent

Эльбрус буткемп

Тогда можно идти по каждому элементу датасета и делать эти же вычисления по очереди для каждого объекта:

$$w_{t+1} = w_t - \lambda \nabla_{w_t} L(x_i, y_i; w_t)$$

```
for t in range(steps):
  for example in data:
    dw = gradient(loss, example, w)
    w = w - lr * dw
```

- `lr` здесь и далее в коде – это λ
- Градиент вычисляется для каждого элемента
- Часто обновляем веса
- Используем мало памяти

Minibatch GD

Эльбрус буткемп

Идти по каждому элементу отдельно долго, по всем элементам разом – накладно по ресурсам, тогда можно разделить датасет на batches («батчи») и итерироваться по ним:

$$w_{t+1} = w_t - \lambda \nabla_{w_t} L(x_{i:i+m}, y_{i:i+m}; w_t)$$

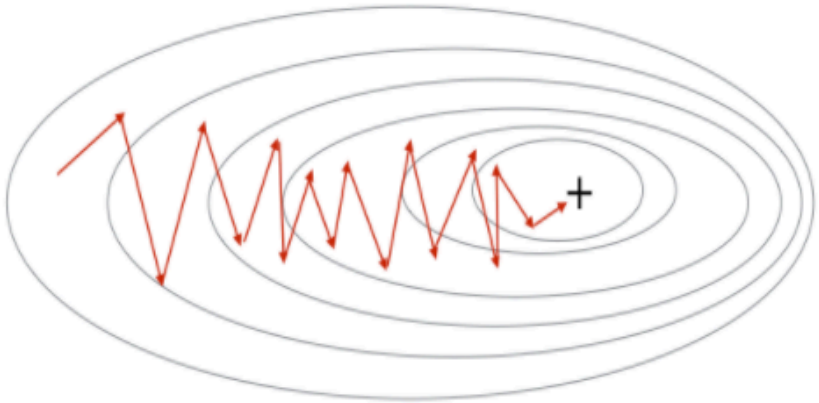
```
for t in range(steps):
  for mini_batch in get_batch(data, batch_size):
    dw = gradient(loss, mini_batch, w)
    w = w - lr * dw
```

❗ Размер батча чаще всего параметр: обычно это 32-64-128-256-512 – все зависит от объема памяти

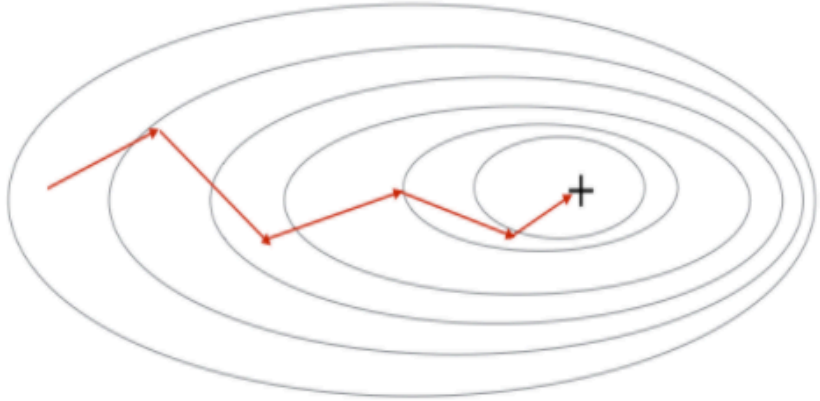
Картинка, которая всё объясняет

Эльбрус буткемп

Stochastic Gradient Descent



Mini-Batch Gradient Descent



Minibatch SGD + Momentum

Эльбрус буткемп

$$v_{t+1} = \rho v_t + \nabla_{w_t} L(x, y; w_t)$$

$$w_{t+1} = w_t - \lambda v_{t+1}$$

```
for t in range(steps):
    dw = gradient(loss, w)
    v = rho * v + dw
    w = w - lr * v
```

- Можем «выскочить» из локального минимума
- Можем немного «сгладить» градиенты, потому что импульс будет «подталкивать» нас в нужную сторону
- Если поверхность функции потерь гладкая и хорошая, то еще и быстрее придем к минимуму
- Но добавляется гиперпараметр – ρ надо выбирать самим

Nesterov Accelerated Gradient

Эльбрус буткемп

$$v_{t+1} = \rho v_t + \lambda \nabla_{w_t} L(x, y; w_t - \rho v_t)$$

$$w_{t+1} = w_t - v_{t+1}$$

```
for t in range(steps):
    dw = gradient(loss, w)
    v = r * v + lr * dw
    w = w - v
```

- Более совершенная реализация импульса
- Реализуется параметром оптимизатора `nesterov=True`

Adagrad • Adaptive gradient

Эльбрус буткемп

> Если параметр принадлежит цепочке часто активирующихся нейронов, его постоянно дёргают туда-сюда, а значит сумма быстро накапливается.

Это хорошо или плохо?

Adagrad · Adaptive gradient

Эльбрус буткемп

Можно адаптировать `lr` для таких параметров:

$$g_t = \nabla_{w_t} L(x, y; w_t)$$

$$G_{t+1} = G_t + g_t^2$$

$$w_{t+1} = w_t - \frac{\lambda}{\sqrt{G_{t+1} + \epsilon}} g_t$$

```
for t in range(steps):
  dw = gradient(loss, w)
  sq_grads += dw**2
  adapt_lr = lr / (sqrt(sq_grads) + e)
  w = w - adapt_lr * dw
```

🔥 `torch.optim.Adagrad()`

RMSprop

Эльбрус буткемп

$$v_t = \delta \mathbf{E}[v_{t-1}] + (1-\delta)(\nabla_{w_t} L(x, y; w_t))^2$$

$$w_{t+1} = w_t - \frac{\lambda}{\sqrt{v_t + \epsilon}}(\nabla_{w_t} L(x, y; w_t))$$

```
for t in range(steps):
  dw = gradient(loss, w)
  sq_grads = delta*sq_grads + (1-delta) * dw**2
  adapt_lr = lr / (sqrt(sq_grads)+e)
  w = w -  adapt_lr * sqrt(sq_grads)
```

RMSprop - root mean squared propagation

🔥 `torch.optim.RMSprop()`

Основные идеи

Эльбрус буткемп

- адаптивный learning rate
- импульс (momentum)

Что же с ними сделать?

Adam · импульс + адаптивный lr

Эльбрус буткемп

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1)(\nabla_{w_t} L(x, y; w_t))$$
$$v_{t+1} = \beta_2 v_t + (1 - \beta_2)(\nabla_{w_t} L(x, y; w_t))^2$$

> Сначала m_t, v_t будут почти нулевыми и будут долго накапливаться, поэтому их искусственно увеличивают на (почти) случайные величины.

$$\hat{m}_{t+1} = \frac{m_t}{1 - \beta_1} \quad \hat{v}_{t+1} = \frac{v_t}{1 - \beta_2}$$

Итоговая формула обновления весов:

$$w_{t+1} = w_t - \frac{\lambda}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

Adam · импульс + адаптивный lr

Эльбрус буткемп

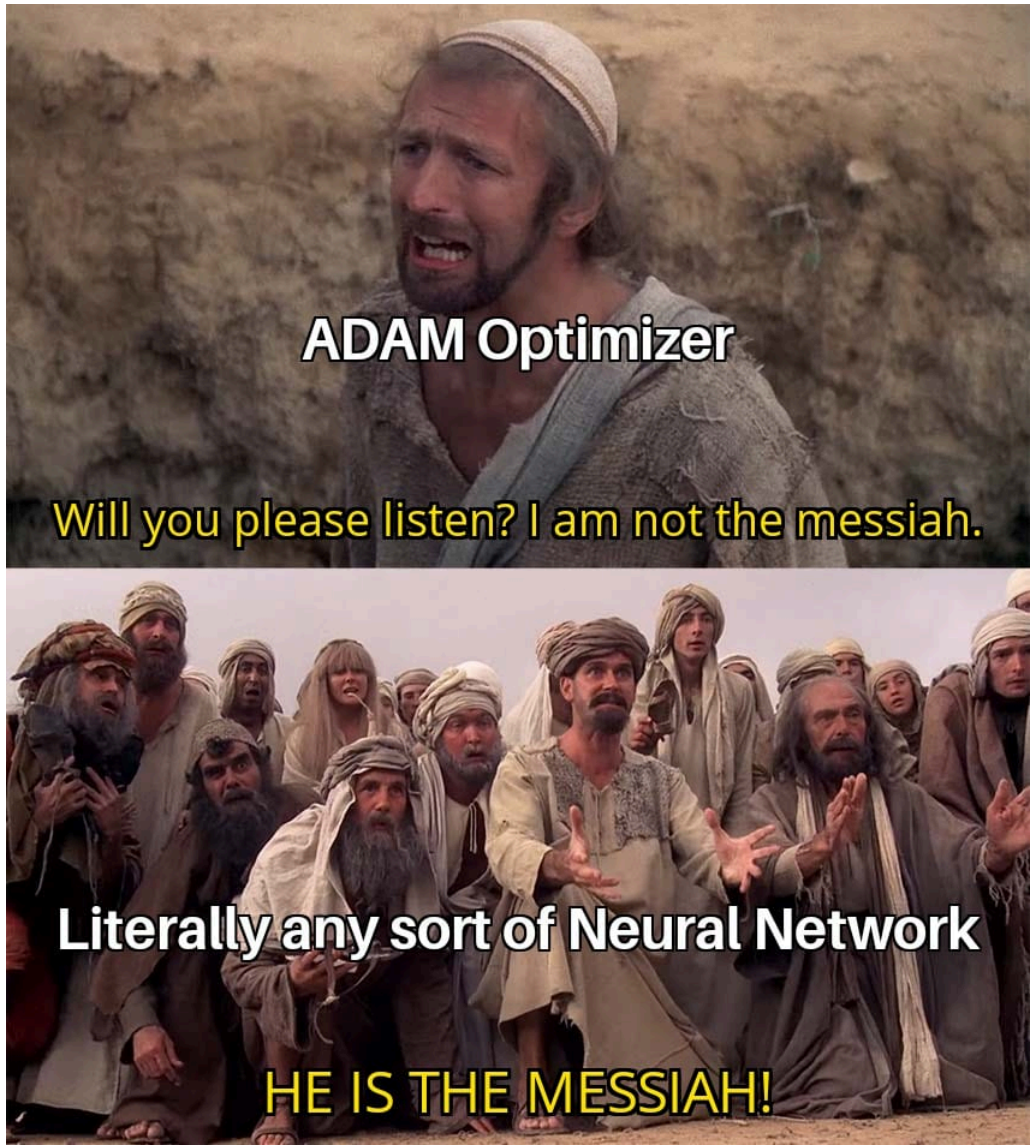
$$m_{t+1} = \beta_1 m_t + (1-\beta_1)(\nabla_{w_t} L(x, y; w_t))$$

$$v_{t+1} = \beta_2 v_t + (1-\beta_2)(\nabla_{w_t} L(x, y; w_t))^2$$

$$w_{t+1} = w_t - \frac{\lambda}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

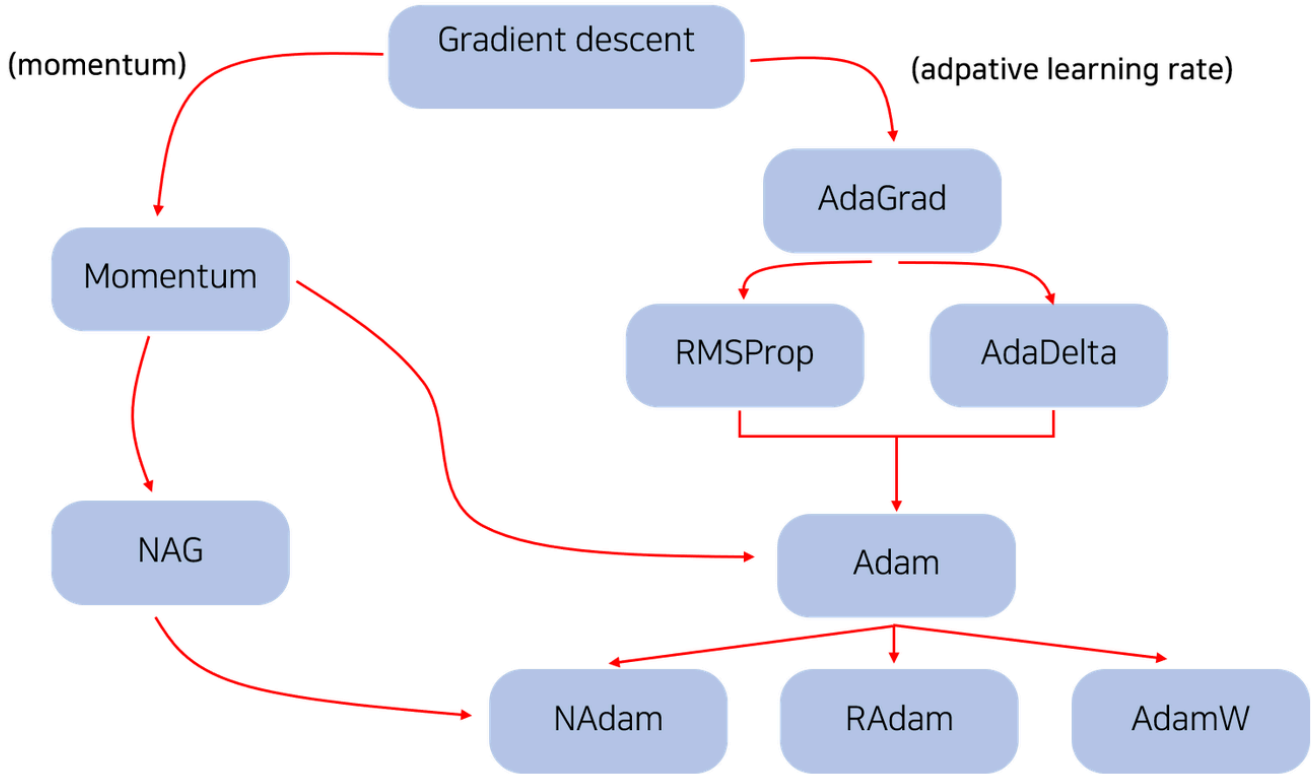
```
for t in range(steps):
    dw = gradient(loss, w)
    m = b1 * m + (1-b1) * dw
    v = b2 * v + (1-b2) * dw**2
    w = w - lr*m / (v.sqrt()+e)
```

Параметры β по умолчанию: (`default: beta1=0.9, beta2=0.999`)



© из какого-то твита

Итоги


Gradient descent
(momentum)
(adpative learning rate)
AdaGrad
Momentum
RMSProp
AdaDelta
NAG
Adam
NAdam
RAdam
AdamW

Итоги

Эльбрус буткемп

- Для решения оптимизационных проблем (= обучения моделей) используют подходы, основанные на градиентных методах
- Две основные идеи:
  - Импульс (momentum, Nesterov)
  - Адаптивный lr (Adagrad, RMSprob)
- Различных видов оптимизаторов много, какой и когда работает лучше - гиперпараметр
  - Обычно в статьях указано, какой алгоритм и с какими параметрами запускали – на это стоит ориентироваться
- Самый популярный - Adam

📝 Методы оптимизации нейронных сетей

Дополнительно

Эльбрус буткемп

- https://www.youtube.com/watch?v=DwKC5S7MceU
- https://habr.com/ru/post/318970/
- https://www.kdnuggets.com/2020/12/optimization-algorithms-neural-networks.html
- https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6
- https://ruder.io/optimizing-gradient-descent/index.html#gradientdescentoptimizationalgorithms