МІНІСТЕРСТВООСВІТИІ НАУКИУКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ
Навчально-науковий інститут Інформаційних технологій
Кафедра штучного інтелекту

Чичкарьов Є.А.

Методичні рекомендації щодо проведення практичних занять з дисципліни «Сучасні технології програмування в системах зі штучним інтелектом» для студентів спеціальності 122 Комп'ютерні науки всіх форм навчання

Практична робота № 4. Налагодження багатошарових нейронних мереж. Алгоритми оптимізації в TensorFlow/Keras

Розглянуто на засіданні кафедри ШІ
Протокол № 1 від «29» серпня 2024 р

Київ
2024

УДК 004.658

Чичкарьов Є.А. Методичні рекомендації щодо проведення практичних занять з дисципліни «Сучасні технології програмування в системах зі штучним інтелектом» для студентів спеціальності 122 Комп’ютерні науки всіх форм навчання. Практична робота № 4. Налагодження багатошарових нейронних мереж. Алгоритми оптимізації в TensorFlow/Keras. - Київ: ДУТ, 2024. 8 с.

Методичні рекомендації призначені для ознайомлення студентів спеціальності 122 Комп’ютерні науки всіх форм навчання з різними аспектами створення додатків для вирішення задач штучного інтелекту та набуття ними практичного досвіду створення додатків, які використовують технології глибокого навчання і пакету TensorFlow, при виконанні практичних занять з дисципліни «Сучасні технології програмування в системах зі штучним інтелектом».

Рецензенти:

Рекомендовано
на засіданні кафедри штучного інтелекту,
протокол № 1 від 29 серпня 2024 р.

ЗМІСТ

Практична робота №4. Налагодження багатошарових нейронних мереж. Алгоритми оптимізації в TensorFlow/Keras

Мета: *ознайомитися з алгоритмами оптимізації в TensorFlow/Keras, їх використанням для побудови і навчання моделей TensorFlow.*

1 Теоретичний розділ

1.1 Різні алгоритми оптимізації для навчання нейронної мережі

Придатний алгоритм оптимізації може експоненціально скоротити час навчання.

Оптимізатори — це алгоритми або методи, які використовуються для зміни атрибутів вашої нейронної мережі, таких як ваги та швидкість навчання, щоб зменшити втрати.

Алгоритми або стратегії оптимізації відповідають за зменшення втрат і надання максимально точних результатів.

Розглянемо різні види оптимізаторів та їхні переваги:

1.1.1 Градієнтний спуск (Gradient Descent)

Градієнтний спуск — найпростіший алгоритм оптимізації. Це ітеративний алгоритм оптимізації, який використовується для знаходження мінімального значення функції. Загальна ідея полягає в тому, щоб ініціалізувати параметри до випадкових значень, а потім робити невеликі кроки в напрямку «нахилу» на кожній ітерації. Градієнтний спуск широко використовується в контрольованому навчанні, щоб мінімізувати функцію помилок і знайти оптимальні значення для параметрів. Для алгоритмів градієнтного спуску розроблено різні розширення. Деякі з них обговорюються нижче:

Зворотне поширення в нейронних мережах також використовує алгоритм градієнтного спуску.

Градієнтний спуск — це алгоритм оптимізації першого порядку, який залежить від похідної першого порядку функції втрат. Він обчислює, у який бік слід змінити вагові коефіцієнти, щоб функція досягла мінімуму. Через зворотне поширення втрати переносяться з одного шару на інший, а параметри моделі, також відомі як ваги, змінюються залежно від втрат, щоб втрати можна було мінімізувати.

Основна розрахункова формула:

$$\Delta\theta = -\eta\nabla_\theta J(\theta),$$
$$\theta = \theta + \Delta\theta = \theta - \eta\nabla_\theta J(\theta),$$

де $\theta -$ параметри мережі, $J(\theta)$ - цільова функція або функція втрат у разі машинного навчання, а $\eta -$ швидкість навчання, $\nabla_\theta -$ градієнт.

Переваги:

1. Легке обчислення.
2. Легко реалізувати.
3. Легко зрозуміти.

Недоліки :

1. Може зупинятися на локальних мінімумах.
2. Ваги змінюються після обчислення градієнта для всього набору даних. Таким чином, якщо набір даних занадто великий, для зближення до мінімумів можуть знадобитися значны ресурси.
3. Потрібен великий обсяг пам’яті для обчислення градієнта на всьому наборі даних.

1.1.2 Стохастичний градієнтний спуск (Stochastic Gradient Descent)

Стохастичний градієнтний спуск (SGD) — це варіант алгоритму градієнтного спуску , який використовується для оптимізації моделей машинного навчання . Він усуває обчислювальну неефективність традиційних методів градієнтного спуску при роботі з великими наборами даних у проектах машинного навчання.

У SGD замість використання всього набору даних для кожної ітерації вибирається лише один випадковий приклад навчання (або невелика партія) для обчислення градієнта та оновлення параметрів моделі. Цей випадковий вибір вводить випадковість у процес оптимізації, тому термін «стохастичний» у стохастичному градієнтному спуску

Перевагою використання SGD є його обчислювальна ефективність, особливо при роботі з великими наборами даних. Завдяки використанню одного прикладу або невеликої партії обчислювальна вартість однієї ітерації значно знижується порівняно з традиційними методами градієнтного спуску, які потребують обробки всього набору даних..

$\theta = \theta - \alpha \cdot \nabla J(\theta; x(i); y(i))$, де $\{x(i), y(i)\}$ – навчальні приклади .

Оскільки параметри моделі часто оновлюються, параметри мають високу дисперсію та флуктуації функцій втрат при різних інтенсивностях.

Переваги:

1. Часте оновлення параметрів моделі, отже, скорочує час.
2. Вимагає менше пам’яті, оскільки не потрібно зберігати значення функцій втрат.
3. Може отримати нові мінімуми.

Недоліки :

1. Висока різниця в параметрах моделі.
2. Може стріляти навіть після досягнення глобальних мінімумів.
3. Щоб отримати таку саму конвергенцію, що й градієнтний спуск, потрібно повільно зменшувати значення швидкості навчання.

1.1.3 Покращення алгоритму стохастичного градієнтного спуску

Імпульс (Momentum)

Метод імпульсу: цей метод використовується для прискорення алгоритму градієнтного спуску шляхом врахування експоненціально зваженого середнього градієнтів. Використання середніх значень змушує алгоритм наближатися до мінімумів швидше, оскільки градієнти до незвичайних напрямків скасовуються. Псевдокод для методу моменту наведено нижче.

```
V = 0
for each iteration i:
   compute dW
   V = β V + (1 - β) dW
   W = W - α V
```

V і **dW** аналогічні швидкості та прискоренню відповідно. α – це швидкість навчання, а β – це аналог імпульсу, який зазвичай підтримується на рівні 0,9. Фізичне тлумачення полягає в тому, що швидкість м'яча, що котиться вниз, нарощує імпульс відповідно до напрямку схилу (градієнта) пагорба і, отже, сприяє кращому прибуттю м'яча з мінімальним значенням (у нашому випадку – з мінімальними втратами).

Імпульс був винайдений для зменшення високої дисперсії в SGD і пом'якшення конвергенції. Це прискорює конвергенцію у відповідному напрямку та зменшує флуктуацію у невідповідному напрямку. У цьому методі використовується ще один гіперпараметр, відомий як імпульс, що нижче символізується «α».

Запишемо формули, що задають цей підхід:

$$v(t) = \alpha v(t-1) + \eta \nabla_\theta J(\theta),$$
$$\theta(t+1) = \theta(t) - v(t),$$

де $\theta -$ параметри мережі, $J(\theta)$ - цільова функція або функція втрат у разі машинного навчання, а $\eta -$ швидкість навчання, $\nabla_\theta -$ градієнт, $\alpha \in [0,1] -$ коефіцієнт, потрібний для збереження історії значень середнього градієнту (зазвичай вибирається близьким до 0.9), $v(t)$ - це накопичене середнє значення градієнтів на кроці t.

Імпульс **γ** зазвичай встановлюється рівним 0,9 або подібним значенням.

Переваги використання імпульсу:

1. Зменшує коливання і високу дисперсію параметрів.
2. Алгоритм оптимізації збигається швидше, ніж градієнтний спуск.

Недоліки :

1. Додається ще один гіперпараметр, який потрібно вибрати вручну і акуратно.

Таким чином, при використанні SGD з імпульсом (або SGD with momentum) кожної нової ітерації оптимізації використовується ковзне середнє

значення градієнту. Рух у напрямку середнього з врахуванням минулих градієнтів додає в алгоритм оптимізації ефект імпульсу, що дозволяє скоригувати напрямок чергового кроку щодо історично домінуючого напряму. Для цього достатньо використовувати наближене ковзне середнє і зберігати всі попередні значення градієнтів, для обчислення «чесного середнього».

Прискорений градієнт Нестерова

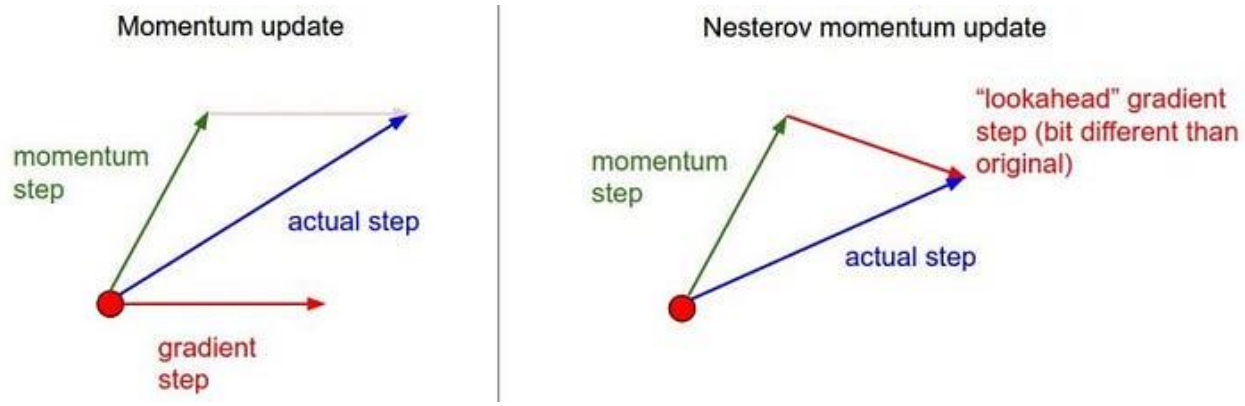
Імпульс може бути хорошим методом, але якщо імпульс занадто високий, алгоритм може пропустити локальні мінімуми та може продовжувати зростати. Отже, для вирішення цієї проблеми був розроблений алгоритм NAG. Це метод погляду вперед.

Accelerated gradient Нестерова відрізняється від методу з імпульсом, його особливістю є обчислення градієнта при оновленні v(t)у чудовій точці. Ця точка береться попереду у напрямку руху накопиченого градієнта:

$$v(t) = \alpha v(t-1) + \eta \nabla_\theta J\big(\theta - \alpha v(t-1)\big),$$
$$\theta(t+1) = \theta(t) - v(t),$$

На зображенні зображені відмінності цих двох методів.

Червоним вектором на першій частині зображено напрям градієнта в поточній точці простору параметрів, такий градієнт використовується стандартному SGD. На другій частині червоний вектор ставить градієнт зрушений на накопичене середнє. Зеленими векторами обох частинах виділено імпульси, накопичені градієнти.



Усі типи градієнтного спуску мають деякі проблеми:

1. Вибір оптимального значення швидкості навчання. Якщо швидкість навчання надто мала, для зближення градієнтного спуску може знадобитися багато років.
2. Майте постійну швидкість навчання для всіх параметрів. Можливо, є деякі параметри, які ми не хочемо змінювати з такою ж швидкістю.
3. Може потрапити в пастку на місцевих мінімумах.

RMSProp (running mean square)

RMSprop: RMSprop був запропонований Джеффрі Гінтоном з Університету Торонто. Інтуїція передбачає застосування методу експоненціально зваженого середнього до другого моменту градієнтів (dW2). Псевдокод для цього такий:

```
S = 0
for each iteration i
   compute dW
   S = β S + (1 - β) dW2
   W = W - α dW/√S + ε
```

Метод імпульсів із поправкою Нестерова – це досить очевидні евристики. Надалі були запропоновані складніші та витонченіші підходи. Наприклад, досить часто під час навчання нейронних мереж використовують оптимізатор RMSProp. Його метою є не лише згладжування псевдоградієнта у стохастичних алгоритмах, а й нормалізація швидкості зміни вектора ваги:

$$\omega = \left[\omega_1, \ldots, \omega_n\right]^T$$

Про що тут йдеться? Часто в завданнях багатовимірної оптимізації, коли число вагових коефіцієнтів, що підбираються, не 2-3, а десятки, сотні, тисячі і більше, приватні похідні за функцією втрат від кожного з них можуть сильно відрізнятися:

$$\frac{dL_i(\omega)}{d\omega} = \begin{bmatrix} \partial L_i(\omega_1)/\partial\omega_1 \\ \cdots \\ \partial L_i(\omega_n)/\partial\omega_n \end{bmatrix}$$

А це, у свою чергу, призводить до сильної зміни одних вагових коефіцієнтів і практично залишає без зміни інші:

$$\omega^{(t+1)} = \omega^{(t)} - \eta_t \cdot \nabla L_i\left(\omega^{(t)}\right)$$

Так ось, щоб нормалізувати швидкість адаптації ваг вектора, Джефрі Хінтон на своєму курсі з Deep Learning (глибоке навчання) запропонував робити наступне перенормування кроку навчання:

$$G = \alpha G + (1-\alpha)\nabla L_i\left(\omega^{(t)}\right) \odot \nabla L_i\left(\omega^{(t)}\right)$$

$$\omega^{(t+1)} = \omega^{(t)} - \eta \cdot \frac{\nabla L_i\left(\omega^{(t)}\right)}{\sqrt{G} + \varepsilon}$$

(тут поділ векторів у другій формулі виконується поелементно). Спочатку ми схожим чином обчислюємо експоненційну ковзну середню, але не за градієнтами, а за квадратами градієнтів (тут - поелементне множення відповідних компонентів векторів).

А потім нормуємо поточний вектор градієнта на корінь квадратний із усереднених (за експонентом) квадратів градієнтів на останніх ітераціях. (Тут $\epsilon > 0$ - невелика константа для виключення поділу на нуль). В результаті, кожен компонент вектора градієнта $\nabla L_i\left(\omega^{(t)}\right)$ буде поділений на пропорційне значення вектора $\sqrt{G} + \epsilon_i$, таким чином, дещо нормований. Завдяки цьому вирівнюється швидкість зміни коефіцієнтів у векторі ω: значення з великими градієнтами починають змінюватися дещо повільніше, а значення з малими градієнтами – дещо швидше. У результаті часто дозволяє збільшити збіжність псевдоградієнтних методів.

Адаград

Одним із недоліків усіх описаних оптимізаторів є те, що швидкість навчання є постійною для всіх параметрів і для кожного циклу. Цей оптимізатор змінює швидкість навчання. Він змінює швидкість навчання **«η»** для кожного параметра та на кожному кроці **«t».** Це алгоритм оптимізації другого порядку. Він працює на основі похідної функції помилок.

Похідна функції втрат для заданих параметрів у заданий момент часу t.

Основним принципом AdaGrad є масштабування швидкості навчання для кожного параметра відповідно до загального квадрата градієнтів, які спостерігаються під час навчання.

Етапи алгоритму такі:

Крок 1: Ініціалізація змінних

- Ініціалізуємо параметри θ і малу константу є, щоб уникнути ділення на нуль.
- Ініціалізуємо змінну G суми квадратів градієнтів нулями, яка має ту саму форму, що й θ.

Крок 2: Обчислення градієнту

Обчислюємо градієнт функції втрат $\nabla_\theta J(\theta)$,

Крок 3: Оновлення суми квадратів градієнту

Оновіть суму квадратів градієнту G для кожного параметра з індексом i:
$G[i] \mathrel{+}= (\nabla_\theta J(\theta[i]))^2$

Крок 4: Оновлення параметрів

Оновлюємо кожен параметр, використовуючи швидкість адаптивного навчання:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

$$g_{t,i} = \nabla_\theta J\left(\theta_{t,i}\right)$$

У наведених вище рівняннях η позначає швидкість навчання, а $\nabla_\theta J(\theta[i])$ представляє градієнт функції втрат відносно параметра θ[i].

Переваги :

1. Швидкість навчання змінюється для кожного циклу навчання.
2. Не потрібно вручну регулювати швидкість навчання.
3. Можливість тренуватися на розріджених даних.

Недоліки :

1. Обчислювально дорогий метод, оскільки необхідно обчислити похідну другого порядку.
2. Швидкість навчання завжди знижується, що призводить до повільного навчання.

AdaDelta

Алгоритм **AdaDelta** - це розширення **AdaGrad**, яке намагається усунути проблему *занепаду швидкості навчання*.

В Adadelta нам не потрібно встановлювати швидкість читання за замовчуванням, оскільки ми беремо ефективну швидкість минулих кроків до поточного градієнта.

З алгоритмом Adagrad виникають три основні проблеми.

- Швидкість навчання монотонно знижується.
- Рівень навчання під час пізнього періоду навчання дуже низький.
- його потрібно встановити, виконавши початкову глобальну швидкість навчання.

Adadelta є розширенням Adagrad і також намагається надмірно зменшити швидкість навчання Adagrad.

Замість накопичення всіх попередньо зведених у квадрат градієнтів ***Adadelta*** обмежує вікно накопичених попередніх градієнтів деяким фіксованим розміром **w**. У цьому випадку використовується експоненціальне ковзне середнє, а не сума всіх градієнтів.

$$\mathbf{E[g^2](t)=\gamma \cdot E[g^2](t-1)+(1-\gamma)\cdot g^2(t)}$$

Ми встановлюємо значення **γ** таким же, як і момент імпульсу, приблизно 0,9.

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2,$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t$$

Оновіть параметри

Переваги :

- Тепер швидкість навчання не падає і навчання не припиняється.

Недоліки :

- Обчислювально дорого.

Адам

Adam, що розшифровується як Adaptive Moment Estimation, - це адаптивний алгоритм швидкості навчання, призначений для підвищення швидкості навчання в глибоких нейронних мережах і швидкого досягнення конвергенції.

Він налаштовує швидкість навчання кожного параметра на основі історії градієнта, і це налаштування допомагає нейронній мережі ефективно навчатися в цілому.

Адам (Adaptive Moment Estimation) працює з імпульсами першого та другого порядку. Інтуїція Адама полягає в тому, що ми не хочемо котитися так швидко лише тому, що можемо перестрибнути мінімум, ми хочемо трохи зменшити швидкість для ретельного пошуку. На додаток до зберігання експоненціально спадного середнього минулих квадратичних градієнтів, наприклад **AdaDelta**, ***Адам*** також зберігає експоненціально спадне середнє минулих градієнтів **M(t).**

M(t) і V(t) – це значення першого моменту, який є ***середнім***, і другого моменту, який є ***нецентрованою дисперсією*** градієнтів відповідно.

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}.$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}.$$

Перший і другий порядок імпульсу

Тут ми беремо середнє **M(t)** і **V(t),** щоб **E[m(t)]** дорівнював **E[g(t)]**, де **E[f(x)]** є очікуваним значенням **f(x)**.

Щоб оновити параметр:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t.$$

Оновлення параметрів:

Значення для β_1 становлять 0,9, 0,999 для β_2 та (10^{-8}) для '**ϵ**'.

Adam Optimization: Алгоритм оптимізації Адама включає метод імпульсу та RMSprop разом із корекцією зсуву. Псевдокод для цього підходу такий:

V = 0

S = 0

for each iteration i

compute dW
$V = \beta_1 S + (1 - \beta_1)\, dW$
$S = \beta_2 S + (1 - \beta_2)\, dW^2$
$V = V/\{1 - \beta_1^i\}$
$S = S/\{1 - \beta_2^i\}$

$$W = W - \alpha \frac{V}{\sqrt{S + \varepsilon}}$$

Kingma and Ba, автори Adam, рекомендували наступні значення для гіперпараметрів..

$\alpha = 0.001$
$\beta 1 = 0.9$
$\beta 2 = 0.999$
$\varepsilon = 10^{-8}$

Переваги :

1. Метод досить швидкий і швидко сходиться.
2. Виправляє зникаючу швидкість навчання, високу дисперсію.
3. Низькі вимоги до пам’яті: Адам вимагає лише двох додаткових змінних для кожного параметра, що робить його ефективним з використанням пам’яті.
4. Надійний до вибору гіперпараметрів: Адам відносно нечутливий до вибору гіперпараметрів, що робить його простим у використанні на практиці.

Недоліки :
Обчислювально дорогий.

1.2 Використання оптимізаторів в Keras

Під час навчання нейронної мережі її ваги спочатку ініціалізуються випадковим чином, а потім вони оновлюються в кожну епоху таким чином, щоб підвищити загальну точність мережі. У кожну епоху вихід навчальних даних порівнюється з фактичними даними за допомогою функції втрат, щоб обчислити помилку, а потім вага відповідно оновлюється. Але як ми знаємо, як оновити вагу, щоб підвищити точність?

По суті, це задача оптимізації, метою якої є оптимізація функції втрат і досягнення ідеальних ваг. Метод, який використовується для оптимізації, відомий як **оптимізатор**. Gradient Descent є найбільш відомим, але є багато інших оптимізаторів, які використовуються для практичних цілей, і всі вони доступні в Keras.

Keras надає API для різних реалізацій оптимізаторів. У Keras наявні такі типи оптимізаторів:

- SGD
- RMSprop

- Adam
- Adadelta
- Adagrad
- Adamax
- Nadam
- Ftrl
- Kepac

Для використання оптимізаторів в Keras є два основні шляхи:

1. Створити екземпляр оптимізатора в Keras і вказати його під час компіляції моделі. Приклад:

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential()
model.add(layers.Dense(64, kernel_initializer='uniform', input_shape=(10,)))
model.add(layers.Activation('softmax'))

opt = keras.optimizers.Adam(learning_rate=0.01)
model.compile(loss='categorical_crossentropy', optimizer=opt)
```

2. Безпосередньо вказати ідентифікатор оптимізатора (у вигляді рядку) під час компіляції моделі. Приклад:

```
# pass optimizer by name: default parameters will be used
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

1.3 Особливості використання оптимізаторів Keras

Нижче обговорено різні типи оптимізаторів у Keras та особливості їх використанням, а також переваги та недоліки.

1.3.1 Оптимізатор Keras SGD (стохастичний градієнтний спуск)

Оптимізатор SGD використовує градієнтний спуск разом із імпульсом. У цьому типі оптимізатора для обчислення градієнта використовується підмножина пакетів.

Синтаксис SGD у Keras

```
tf.keras.optimizers.SGD
(learning_rate=0.01, momentum=0.0, nesterov=False, name="SGD", **kwargs)
```

Приклад Keras SGD

Тут оптимізатор SGD імпортовано з бібліотеки Keras . Ми також вказали швидкість навчання в цьому прикладі.

```
import numpy as np
import tensorflow as tf
opt = tf.keras.optimizers.SGD(learning_rate=0.1)
var = tf.Variable(1.0)
loss = lambda: (var ** 2)/2.0       # d(loss)/d(var1) = var1
step_count = opt.minimize(loss, [var]).numpy()
# Step is `- learning_rate * grad`
var.numpy()
```

Результат:

0.9

1.3.2 Керас Оптимізатор RMSProp (середньоквадратичний розповсюдження)

Метою оптимізатора RMSProp є забезпечення постійного зменшення середнього квадрата градієнтів. А по-друге, також виконується ділення градієнта на корінь середнього.

Синтаксис Keras RMSProp

```
tf.keras.optimizers.RMSprop(learning_rate=0.001, rho=0.9,
momentum=0.0, epsilon=1e-07, centered=False, name="RMSprop",**kwargs)
```

Приклад Кераса RMSProp (середньоквадратичний розповсюдження)

У наступному прикладі показано, як реалізовано алгоритм середньоквадратичного розповсюдження в бібліотеці keras

```
opt = tf.keras.optimizers.RMSprop(learning_rate=0.1)
var1 = tf.Variable(10.0)
loss = lambda: (var1 ** 2) / 2.0    # d(loss) / d(var1) = var1
step_count = opt.minimize(loss, [var1]).numpy()
var1.numpy()
```

Результат:

9.683772

1.3.3 Keras Adam Optimizer (оцінка адаптивного моменту)

Оптимізатор adam використовує алгоритм adam, у якому для виконання процесу оптимізації використовується метод стохастичного градієнтного спуску. Він ефективний у використанні та споживає дуже мало пам’яті. Це доречно у випадках, коли для використання доступна величезна кількість даних і параметрів.

Keras Adam Optimizer є найпопулярнішим і широко використовуваним оптимізатором для навчання нейронної мережі.

Синтаксис Adam Optimizer

```
tf.keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9 beta_2=0.999, epsilon=1e-07,amsgrad=False, name="Adam",**kwargs)
```

Приклад Keras Adam Optimizer

Наступний фрагмент коду показує приклад **adam optimizer**.

```
opt = tf.keras.optimizers.Adam(learning_rate=0.1)
var1 = tf.Variable(10.0)
loss = lambda: (var1 ** 2)/2.0      # d(loss)/d(var1) == var1
step_count = opt.minimize(loss, [var1]).numpy()
# The first step is `-learning_rate*sign(grad)`
var1.numpy()
```

Результат:

9.9

1.3.4 Керас Оптимізатор Adadelta

Оптимізатор Adadelta використовує адаптивну швидкість навчання з методом стохастичного градієнтного спуску. У цьому методі пропонується використовувати експоненційну середню суми квадратів градієнтів у знаменнику формули зміни вагових коефіцієнтів. У формулі оновлення використано експоненційне середнє із суми квадратів попередніх змін аналізованого параметра. Такий підхід дозволяє повністю відмовитися від коефіцієнта навчання у формулі оновлення вагових коефіцієнтів та побудувати максимально адаптивний алгоритм навчання. Але ціною за це будуть додаткові ітерації обчислень і виділення додаткової пам'яті для зберігання іншого значення в кожному нейроні.

Синтаксис звертання до оптимізатору Adadelta

```
tf.keras.optimizers.Adadelta(
learning_rate=0.001, rho=0.95, epsilon=1e-07, name="Adadelta",**kwargs)
```

1.3.5 Керас Оптимізатор Adagrad

Керас Adagrad optimizer має швидкість навчання, яка використовує певні параметри. В AdaGrad швидкість навчання ділиться на квадратний корінь із суми квадратів градієнтів за всі попередні ітерації навчання. Такий підхід дозволяє знизити швидкість навчання параметрів, що часто оновлюються. Основний недолік даного методу випливає з його формули - сума квадратів градієнтів може тільки зростати і, як наслідок, швидкість навчання прагне "0". Що, зрештою, призведе до зупинки навчання.

Синтаксис Keras Адаград

```
tf.keras.optimizers.Adagrad(learning_rate=0.001,
initial_accumulator_value=0.1,epsilon=1e-07, name="Adagrad", **kwargs)
```

2 Завдання

1. Дослідити вплив на результати навчання вибір алгоритму оптимізації (у відповідності до наданого зразка).
2. Додати 1, 2, 4 приховані шари, спробувати змінити кількість нейронів в цих шарах. Знову перевірити, як змінюються результати навчання для різних оптимізаторів, спробувати змінити їх налаштування.
3. Створити власний набір даних для мультикласової класифікації, перевірити поведінку оптимізаторів на новому датасеті.
4. Завантажити датасет (для вирішення задач класифікації) з сайту https://archive.ics.uci.edu/, створити класифікатор для нього, спробуваути порівняти декілька класфікаторів.

3 Контрольні питання

1. Що таке оптимізатор, навіщо він потрібен прі вирішенні задач машинного навчання?
2. Які оптимізатори використовуються в Keras/TensorFlow?
3. Опішить алгоритм градієнтного пошуку мінімуму.
4. Опішить алгоритм SGD для пошуку мінімуму.
5. Опішить алгоритм RMSProp для пошуку мінімуму.
6. Опішить алгоритм AdaGrad для пошуку мінімуму.
7. Опішить алгоритм AdaDelta для пошуку мінімуму.
8. Опішить алгоритм Adam для пошуку мінімуму.
9. Як налаштувати поведінку оптимізатора в Keras/TensorFlow?