Neura - uživatelská dokumentace

1. Úvod

Tato práce se věnuje tvorbě knihovny na neuronové sítě, kterou bude možné použít k aproximaci některých jednoduchých matematických funkcí. Implementuje základní stavební kameny neuronových sítí a pokládá základ, na kterém je možné stavět další prvky. Knihovna je celá vytvořená za pomocí jediné externí knihovny a to numpy. Díky tomu je kód velice snadný na pochopení, ale zároveň toto rozhodnutí snižuje rychlost celé knihovny.

Knihovna celá běží na CPU a nepodporuje GPU akceleraci pro jednodušší kód. Celý kód dodržuje názvovou konvenci z PEP 8.

2. Rozdělení

Kód je strukturovaný jako python knihovna, přičemž veškeré jeho funkcionality jsou rozděleny do pěti python souborů a to:

- activation functions,
- layers,
- loss_functions,
- optimizers,
- network.

Toto rozdělení bylo zvoleno pro zvýšení přehlednosti kódu. Mimo jiné se v knihovně také nachází již předpřipravené příklady jejího použití.

3. Aktivační funkce

Knihovna neura nakládá s aktivačními funkcemi jako se součástmi jednotlivých vrstev a je tedy nutné je přiložit jako argument k vytvoření vrstvy. Vždy je nutné vytvořit instanci aktivační funkce.

Neura implementuje dvě aktivační funkce, LeakyReLU a Sigmoid, přičemž vytváří abstraktní třídu AbstractActivationFunction pro snadnou implementaci dalších funkcí. Aktivační funkce se nachází v souboru activation_functions.py a je tedy nutné je importovat následujícím způsobem:

```
from neura.activation_functions import Sigmoid,LeakyReLU

gradient_of_negative_input = 0.1

sigmoid_functon = Sigmoid()
leakyReLU = LeakyReLU(gradient_of_negative_input)

ReLU = LeakyReLU(0)
```

4. Loss funkce

Loss funkce jsou nedílnou součástí neuronových sítí v knihovně neura a jejich instance je tedy nutné použít jako argument při vytváření nového objektu třídy Network.
V knihovně jsou implementovány dvě loss funkce: Binary-cross-entropy a Mean square error. Mean square error byla vybrána pro jeji schopnost dobře aproximovat matematické funkce. Binary-cross-entropy byla vybrána pro binární klasifikaci.

Loss funkce lze importovat následujícím způsobem:

```
from neura.loss_functions import MSE, BinaryCrossEntropy
```

5. Vrstvy

Neura v sobě má pouze dense layer, což je nejzákladnější typ vrstvy. Jako argumenty k vytvoření potřebuje velikost vstupu, velikost výstupu a aktivační funkci.

Vrstva se nachází v souboru layers.py a lze ji importovat a poté vytvořit následovně:

```
from neura.layers import DenseLayer
from neura.activation_functions import Sigmoid

input_size = 1
output_size = 10
activation_function = Sigmoid()

new_layer = DenseLayer(input_size, output_size, activation_function)
```

Vrstva má dvě hlavní funkce: forward a backpropagate. Funkce forward nechá vrstvou projít daný vstup podle již popsaného procesu, zatímco funkce backpropagate počítá gradienty. Obě tyto funkce během učení automaticky využívá třída Network.

Třída přijímá do funkce forward argumenty ve formátu numpy pole, ve kterém je první dimenzí počet vstupů a druhou je velikost jednoho vstupu. Do funkce backward naopak přijímá numpy pole, ve kterém je první dimenzí počet vstupů a druhou dimenzí je velikost jednoho výstupu.

6. Optimizéry

Optimizéry zprostředkovávají aplikaci změn jednotlivých parametrů na základě gradientů. V knihovně neura se nachází dva optimizéry:

- Stochastický gradient descent (SGD) je základním optimizerem, který pouze přičte opačný gradient vynásobený learning ratem k parametrům. Rychlost učení lze tedy pomocí learning ratu regulovat.
- Stochastický gradient descent používající momentum (MomentumSGD) je vylepšená verze původního SGD, která přidává ještě další hyperparameter momentum, který lze měnit pro regulaci učení. Momentum do jisté míry

přesouvá rychlost předešlého kroku učení do dalšího kroku. Je vhodné jej zvolit v hodnotách v intervalu <0;1>.

Optimizéry se nachází v souboru optimizers a lze je importovat a inicializovat následujícím způsobem:

```
from neura.optimizers import SGD, MomentumSGD

learning_rate = 10**(-3)
momentum = 0.9

SGD_optimizer = SGD(learning_rate)
momentumSGD_optimizer = MomentumSGD(learning_rate, momentum)
```

7. Network

Network je poslední struktura, která je v knihovně implementována. Slouží k uspořádání všech prvků, samotnému tréninku, testování a případné predikci. Třída se nachází v souboru network.py a pro její inicializaci vyžaduje instanci loss funkce a optimizeru. Pomocí funkce add_layer je do ní možné přidávat vrstvy.

```
from neura.optimizers import MomentumSGD
from neura.layers import DenseLayer

layer = DenseLayer(input_size,output_size,activation_function)

network = Network(MSE(), MomentumSGD(learning_rate, momentum))
network.add_layer(layer)
```

Funkce fit zprostředkovává celý proces učení. Přijímá jako argumenty x_data, což jsou vstupní data, y_data, což jsou data, která se síť snaží predikovat (labely), epochs, což je počet epoch neboli cyklů, batch_size, což je množství dat, po kterém optimizér aktualizuje parametry a epoch_info, což je počet epoch, po kterých se do konzole vypíše chybovost sítě. Dále přijímá ještě parametr training_percentage, který určuje, jaká část dat bude použita k tréninku a jaká k vyhodnocení přesnosti sítě. Posledním parametrem je bool parametr shuffle, díky kterému může síť promíchat data před tréninkem.

```
network.fit(x_data,y_data,epochs,batch_size, epoch_info,
training_percentage, shuffle)
```

Síť dále poskytuje možnost testovat model pomocí funkce test, která přijímá pouze parametry x a y, kde x jsou vstupní data a y jsou labely.

```
network.test(x,y)
```

Síť lze dále používat k predikci dalších hodnot pomocí funkce predict.

```
network.predict(x)
```

Vytrénovaný model je možné uložit pomocí funkce save, která jako argument přijímá název, pod kterým bude uložena. Přípona uloženého modelu je .pkl.

```
network.save(filename)
```

K opětovnému načtení je určena funkce load. Ta po jejím zavolání vytvoří novou instanci třídy Network, a proto se jedná o statickou funkci, která není volána na objekt, ale na třídu.

```
network = Network.load(filename)
```