Semestrální práce – Project 5: Machine Learning

4IZ431 - Umělá inteligence 1

https://inst.eecs.berkeley.edu/~cs188/sp20/project5/

https://github.com/itsDaiton/cs188-machine-learning

Question 1: Perceptron

První otázkou tohoto projektu bylo implementovat binární perceptron. Jelikož se jedná o klasifikaci binární, tak výstup perceptronu nabývá hodnot buďto -1 nebo 1.

Prvním úkolem bylo implementovat metodu ***run(self, x)***, která vypočítá skalární součin dvou vektorů – hodnoty vstupního neuronu a jeho odpovídající váhy.

```
# Výpočet skalárního součinu vstupních neuronů a jejich vah
return nn.DotProduct(x, self.get_weights())
```

Druhým úkolem bylo implementovat metodu predikce ***get_prediction(self, x)***, která by měla vrátit hodnotu 1 v případě, že výsledek skalárního součinu byl kladný nebo hodnotu -1 v opačném případě. Zavoláme tedy implementovanou metodu na výpočet skalárního součinu a převedeme ji na skalár, jelikož předchozí metoda vrací objekt *nn.DotProduct* ze zdrojové knihovny autorů projektu. Pokud je výsledek kladný, vrátíme 1, jinak vrátíme -1.

```
# Klasifikace vstupního neuronu x - pokud je výsledek skalárního součinu kladný, tak 1, jinak -1
if nn.as_scalar(self.run(x)) >= 0:
    return 1
else:
    return -1
```

Posledním úkolem této otázky bylo napsat metodu ***train(self)***, která bude perceptron trénovat na dostupných datech. Úkolem je opakovaně procházet dataset a provádět aktualizace na špatně klasifikovaných trénovacích případech. Kód opakovaně kontroluje jednotlivé trénovací případy a jestliže se výsledná hodnota predikce nerovná předpokládanému výsledku (labelu), tak se použije metoda *nn.Parameter.update*, která upraví hodnotu vah. Cyklus končí v případě, že jsou všechny příklady správně klasifikovány.

```
# Trénování perceptronu - pro vstupní data která se špatně klasifikují se upravují váhy dokud se všechna data neklasifikují správně
missclassified = True
while missclassified:
    missclassified = False
    for x, y in dataset.iterate_once(1):
        if self.get_prediction(x) != nn.as_scalar(y):
            nn.Parameter.update(self.w, x, nn.as_scalar(y))
            missclassified = True
```

Question 2: Non-linear Regression

Cílem této otázky je natrénovat neuronovou síť, která bude aproximovat funkci *sin(x)* na intervalu [−2π, 2π]. Prvním úkolem je implementovat metodu ***__init__(self)***, ve které je potřeba vytvořit vlastní architekturu neuronové sítě. Moje architektura má následující parametry:

- Vstupní vrstva – 1 neuron
- Skrytá vrstva č. 1 – 200 neuronů

- Skrytá vrstva č. 2 – 100 neuronů
- Výstupní vrstva – 1 neuron
- Velikost dávky – 100
- Rychlost učení – 0,02

Jedná se o plně propojenou (dopřednou) neuronovou síť. V kódu lze vidět propojení jednotlivých vrstev a také přidání biasů do odpovídacích vrstev. Váhy a biasy jednotlivých vrstev jsem ještě seskupil do pole pro jednoduší práci.

```
# Propojení vstupní vrstvy s první skrytou vrstvou
self.w1 = nn.Parameter(1, 200)
self.b1 = nn.Parameter(1, 200)

# Propojení první skryté vrstvy s druhou skrytou vrstvou
self.w2 = nn.Parameter(200, 100)
self.b2 = nn.Parameter(1, 100)

# Propojení druhé skryté vrstvy s výstupní vrstvou
self.w3 = nn.Parameter(100, 1)
self.b3 = nn.Parameter(1, 1)

# Seskupení všech vah a biasů do jedné proměnné
self.hyperparameters = [self.w1, self.b1, self.w2, self.b2, self.w3, self.b3]

# Zbylé hyperparametry - velikost dávky a rychlost učení
self.batch_size = 100
self.learning_rate = 0.02
```

Druhým úkolem bylo implementovat metodu ***run(self, x)***, která provede predikci modelu. Výpočet spočívá v tom, že nejdříve se vypočte lineární kombinace vstupu a vah, k tomuto výsledku ještě připočteme odpovídající bias a z tohoto čísla následně vypočteme výstupní aktivační hodnotu neuronu pomocí aktivační funkce ReLU. Důležité je říci, že aktivační funkci ReLU již nepoužíváme při samotné klasifikaci na výstupní vrstvě, protože ReLU ořezává záporné hodnoty *(dead neuron problem)*. To znamená, že v případě, když budeme mít záporný výstup, tak ReLU vždy klasifikuje tuto hodnotu jako 0. Může tedy dojít ke zkreslení a také k horší přesnosti modelu. Z tohoto důvodu se ReLU na výstupní vrstvě nepoužívá. Provedeme tedy pouze lineární kombinaci bez použití aktivační funkce. Výstupem metody je predikce modelu.

```
# Výpočet výstupní aktivační hodnoty pro první skrytou vrstvu pomocí aktivační funkce ReLU
output1 = nn.ReLU(nn.AddBias(nn.Linear(x, self.w1), self.b1))

# Výpočet výstupní aktivační hodnoty pro druhou skrytou vrstvu pomocí aktivační funkce ReLU
output2 = nn.ReLU(nn.AddBias(nn.Linear(output1, self.w2), self.b2))

# Výpočet výstupní aktivační hodnoty pro výstupní vrstvu
# Zde se již funkce ReLU nepoužívá, protože chceme na výstupu i záporné hodnoty, které ReLU ořezává
output3 = nn.AddBias(nn.Linear(output2, self.w3), self.b3)

# Výsledná predikce modelu
return output3
```

Třetím úkolem bylo implementovat metodu ***get_loss(self, x, y)***, která vypočítá ztrátovou funkci predikce. Použijeme ztrátovou funkci z knihovny nn, Square Loss (střední kvadratická chyba), která se často používá při regresních úlohách.

```python
# Výpočet ztrátové funkce - rozdíl mezi získanou hodnotou a předpokládanou hodnotou
return nn.SquareLoss(self.run(x), y)
```

Posledním úkolem bylo implementovat metodu ***train(self, dataset)***, která vytvořenou neuronovou síť natrénuje. V zadání je určeno, že výsledná ztrátová funkce by měla být 0.02 a nižší. Nejdříve si inicializujeme ztrátovou funkci na kladné nekonečno, abychom ji mohli postupně snižovat. Cyklus trénování tedy nastavíme tak, aby neustále běžel, dokud je ztrátová funkce větší než 0.02. Samotné trénování má následující postup:

1. Provedeme predikci modelu a vypočítáme ztrátovou funkci z predikce
2. Vypočteme gradienty ztráty vzhledem k jednotlivým parametrům modelu
3. Iterujeme skrze gradienty a zpětně aktualizujeme hodnoty vah a biasů v síti dle hodnot gradientu a rychlosti učení (backpropagation algoritmus)

```python
# Prvotní incilializace ztrátové funkce na nekonečno, jelikož chceme v rámci trénování minimalizovat tuto hodnotu
loss = float('inf')
# Trénování modelu - dokud není ztrátová funkce menší než 0.02, tak se bude model neustále trénovat
while loss > 0.02:
    # Iterace přes dávku dat z datasetu
    for x, y in dataset.iterate_once(self.batch_size):
        loss = self.get_loss(x, y)
        # Výpočet gradientů pro jednotlivé hyperparametry
        gradients = nn.gradients(loss, self.hyperparameters)
        # Převod ztrátové funkce zpět na skalární hodnotu
        loss = nn.as_scalar(loss)
        # Backpropagation - zpětná aktualizace vah a biasů pro celou síť
        for i in range(len(gradients)):
            self.hyperparameters[i].update(gradients[i], -self.learning_rate)
```

Question 3: Digit Classification

Cílem této otázky je natrénovat síť pro klasifikaci ručně psaných číslic z datasetu MNIST. Každá číslice má 28x28 pixelů, budeme tedy potřebovat 784 vstupů. Jedná se o vícetřídní klasifikaci, jelikož budeme klasifikovat do jedné z deseti tříd – číslice 0 až 9. Postup řešení otázky je velmi podobný otázce 2. Nejdříve je potřeba vytvořit si architekturu sítě v metodě ***__init__(self)***. V případě této úlohy jsou zvolil větší architekturu:

- Vstupní vrstva – 784 neuronů
- Skrytá vrstva č. 1 – 256 neuronů
- Skrytá vrstva č. 2 – 128 neuronů
- Skrytá vrstva č. 3 – 64 neuronů
- Výstupní vrstva – 10 neuronů
- Velikost dávky – 40
- Rychlost učení – 0,1

Velikost dávky a rychlost učení jsem v průběhu implementace měnil, při výše uvedených hodnotách síť dosahovala nejlepších výsledků.

```python
# Propojení vstupní vrstvy (28*28 pixelů) s první skrytou vrstvou
self.w1 = nn.Parameter(784, 256)
self.b1 = nn.Parameter(1, 256)

# Propojení první skryté vrstvy s druhou skrytou vrstvou
self.w2 = nn.Parameter(256, 128)
self.b2 = nn.Parameter(1, 128)

# Propojení druhé skryté vrstvy s výstupní vrstvou
self.w3 = nn.Parameter(128, 64)
self.b3 = nn.Parameter(1, 64)

# Propojení třetí skryté vrstvy s výstupní vrstvou (10 tříd)
self.w4 = nn.Parameter(64, 10)
self.b4 = nn.Parameter(1, 10)

# Seskupení všech vah a biasů do jedné proměnné
self.hyperparameters = [self.w1, self.b1, self.w2, self.b2, self.w3, self.b3, self.w4, self.b4]

# Zbylé hyperparametry - velikost dávky a rychlost učení
self.batch_size = 40
self.learning_rate = 0.1
```

Metoda ***run(self, x)*** pro provedení predikce je totožná jako v předchozí otázce, jen bylo potřebovat přidat výpočet aktivační hodnoty na třetí skryté vrstvě. Opět nepoužíváme ReLU při finální klasifikaci číslice (v tomto případě je to i uvedené v zadání).

```python
# Výpočet výstupní aktivační hodnoty pro první skrytou vrstvu pomocí aktivační funkce ReLU
output1 = nn.ReLU(nn.AddBias(nn.Linear(x, self.w1), self.b1))

# Výpočet výstupní aktivační hodnoty pro druhou skrytou vrstvu pomocí aktivační funkce ReLU
output2 = nn.ReLU(nn.AddBias(nn.Linear(output1, self.w2), self.b2))

# Výpočet výstupní aktivační hodnoty pro třetí skrytou vrstvu pomocí aktivační funkce ReLU
output3 = nn.ReLU(nn.AddBias(nn.Linear(output2, self.w3), self.b3))

# Výpočet výstupní aktivační hodnoty pro výstupní vrstvu, zde bez ReLU (dle zadání)
output4 = nn.AddBias(nn.Linear(output3, self.w4), self.b4)

# Výsledná predikce modelu
return output4
```

Následuje metoda ***get_loss(self, x, y)***, kde tentokrát použijeme ztrátovou funkci *nn.SoftmaxLoss*, jelikož se jedná o vícetřídní klasifikaci (dle zadání).

```python
# Výpočet ztrátové funkce pro vícetřídní klasifikaci
return nn.SoftmaxLoss(self.run(x), y)
```

Na závěr je potřeba implementovat metodu ***train(self, dataset)***, která bude opět velmi podobná jako v předchozí otázce. Nyní je v zadání určeno, že síť by se měla trénovat do té doby, dokud nedosáhne přesnosti cca. 97 – 98% na validační množině. Implementace je tedy velmi obdobná, avšak cyklus se bude opakovat, dokud nebude potřebná přesnost 98%. Při skončení každé iterace datasetu se zkontroluje validační přesnost a pokud není na požadované hodnotě, trénování pokračuje.

```python
validation_accuracy = 0
# Trénování modelu - dokud není přesnost modelu na validačních datech větší než 0.98, tak se bude model neustále trénovat
while validation_accuracy <= 0.98:
    # Iterace přes dávku dat z datasetu
    for x, y in dataset.iterate_once(self.batch_size):
        loss = self.get_loss(x, y)
        # Výpočet gradientů pro jednotlivé hyperparametry
        gradients = nn.gradients(loss, self.hyperparameters)
        # Backpropagation - zpětná aktualizace vah a biasů pro celou síť
        for i in range(len(gradients)):
            self.hyperparameters[i].update(gradients[i], -self.learning_rate)
    # Výpočet přesnosti modelu na validačních datech
    validation_accuracy = dataset.get_validation_accuracy()
```

Question 4: Language Identification

V poslední otázce tohoto projektu je úkolem vytvořit síť, která na základě textu zjistí, v jakém jazyce byl napsán. Dataset obsahuje slova v pěti jazycích a cílem modelu je identifikovat v jakém jazyce bylo slovo napsáno. Slova jsou složena z různého počtu písmen, tudíž je potřeba vytvořit architekturu, která toto zvládne. Vstupem do sítě budou písmena x_0 až x_n. Tento problém lze vyřešit pomocí rekurentních neuronových sítí (RNN), kde výstup jedné vrstvy slouží jako vstup pro další vrstvu. Vstupem do sítě bude počet unikátních písmen, které tvoří abecedu daného datasetu – 47 (získáno ze zadání). Výstupem bude klasifikace do jednoho z jazyků, v datasetu jich je celkem 5. Nejprve tedy vytvoříme architekturu neurnové sítě v metodě ***__init__(self)***. Síť má následující parametry:

- Vstupní vrstva – 47 neuronů
- Rekurzivní skrytá podsít – 256 neuronů
- Výstupní vrstva – 5 neuronů
- Velikost dávky – 100
- Rychlost učení – 0,15

```python
# Propojení vstupní vrstvy (47 znaků) s první skrytou vrstvou (zpracování prvního písmena slova)
self.w1 = nn.Parameter(self.num_chars, 256)
self.b1 = nn.Parameter(1, 256)

# Rekurzivní skrytá podsíť pro zpracování dalších písmen slova (tento kód se rekurzivně volá v cyklu níže)
self.w2 = nn.Parameter(256, 256)
self.b2 = nn.Parameter(1, 256)

# Poslední vrstva pro vytvoření predikce modelu (klasifikace do 5 jazyků)
self.w3 = nn.Parameter(256, len(self.languages))
self.b3 = nn.Parameter(1, len(self.languages))

# Seskupení všech vah a biasů do jedné proměnné
self.hyperparameters = [self.w1, self.b1, self.w2, self.b2, self.w3, self.b3]

# Zbylé hyperparametry - velikost dávky a rychlost učení
self.batch_size = 100
self.learning_rate = 0.15
```

Dále implementujeme metodu ***run(self, xs)***. Nejdříve se vypočte aktivační hodnota pro první slovo jako v klasické dopředné síti. Dále napojíme výstup této vrstvy na skrytou podsíť, kterou budeme rekurzivně volat dle počtu písmen ve slově. Výsledkem je spojení prvního slova s druhým do společného vektoru – toto opakujeme, dokud neprojdeme všechna písmena slova. V cyklu lze vidět, že iterujeme přes proměnnou xs (písmena slova), ale začínáme až od druhého. Vždy sečteme lineární kombinace předchozího písmena s aktuálním písmenem, přidáme bias a vypočteme aktivační hodnotu. Pokud máme celé slovo zpracované, pošleme ho na výstupní vrstvu, kde opět spočítáme lineární kombinaci a provedeme finální predikci (o jaký jazyk se jedná).

```python
# Výpočet výstupní aktivační hodnoty pro první písmeno slova (stejná architekrura jako v předchozích dopředných sítích)
h = nn.ReLU(nn.AddBias(nn.Linear(xs[0], self.w1), self.b1))

# Dále musíme spojit výstup předchozího kroku s dalším písmenem slova (napojení na skrytou podsíť)
# Výsledek je vektor prvních dvou písmen slova
# Tento proces poté opakujeme pro všechna písmena slova, dokud nezpracujeme celé slovo (rekuzrivní volání v cyklu)
# Důležité je, že výstup z předchozího kroku je vždy vstupem pro další krok (skrytý stav [h])
for x in xs[1:]:
    h = nn.ReLU(nn.AddBias(nn.Add(nn.Linear(x, self.w1), nn.Linear(h, self.w2)), self.b1))
# Výsledná predikce modelu
return nn.AddBias(nn.Linear(h, self.w3), self.b3)
```

Následuje spočtení ztrátové funkce identicky jako v předchozí otázce, protože se jedná opět o vícetřídní klasifikaci.

```python
# Výpočet ztrátové funkce - opět používáme SoftmaxLoss, jelikož se jedná o vícetřídní klasifikaci
return nn.SoftmaxLoss(self.run(xs), y)
```

Na závěr je potřeba síť natrénovat. Proces je totožný jako u dvou předchozích otázek. V tomto případě je ale v zadání uvedeno, že potřebná validační přesnost má být alespoň 81%. Takto v celku špatná přesnost se vyžaduje, protože v datech je velké množství šumu (dle zadání). Nastavil jsem tedy, aby cyklus proběhl 20x (20 epoch). Konec trénování je v tomhle případě fixní, skončí vždy po 20 epochách. Moje architektura dosahovala potřebné přesnosti při průměrně 16-20 epochách. Po skončení se vypíše přesnost a pokud není dostatečná, autograder řešení zamítne.

```python
# Trénování modelu - 20 epoch trénování
# V zadání totiž autoři píší, že jejich referenční model dosahuje přesnosti cca. 0.89 na validačních datech po 10 - 20 epochách
# Trénování tedy neskončí dříve jako u předchozích modelů, ale vždy až po 20 epochách
# Po skončení trénování se vypíše přesnost modelu na validačních datech a autograder i tak vyhodnotí úspěšnost
for i in range(20):
    # Iterace přes dávku dat z datasetu
    for x, y in dataset.iterate_once(self.batch_size):
        loss = self.get_loss(x, y)
        # Výpočet gradientů pro jednotlivé hyperparametry
        gradients = nn.gradients(loss, self.hyperparameters)
        # Backpropagation - zpětná aktualizace vah a biasů pro celou síť
        for i in range(len(gradients)):
            self.hyperparameters[i].update(gradients[i], -self.learning_rate)
```

Výsledky implementace

Question 1

```
Question q1
===========
*** q1) check_perceptron
Sanity checking perceptron...
Sanity checking perceptron weight updates...
Sanity checking complete. Now training perceptron
*** PASS: check_perceptron

### Question q1: 6/6 ###

Finished at 14:34:15

Provisional grades
==================
Question q1: 6/6
------------------
Total: 6/6
```

Question 2

```
Question q2
===========
*** q2) check_regression
Your final loss is: 0.018305
*** PASS: check_regression

### Question q2: 6/6 ###

Finished at 15:20:49

Provisional grades
==================
Question q2: 6/6
------------------
Total: 6/6
```

Question 3

```
Question q3
===========
*** q3) check_digit_classification
Your final test set accuracy is: 97.560000%
*** PASS: check_digit_classification

### Question q3: 6/6 ###

Finished at 15:49:55

Provisional grades
==================
Question q3: 6/6
------------------
Total: 6/6
```

Question 4

```
Your final test set accuracy is: 86.800000%
*** PASS: check_lang_id

### Question q4: 7/7 ###

Finished at 16:13:13

Provisional grades
==================
Question q4: 7/7
------------------
Total: 7/7
```

Autograder problém (bug)

Autograder mi individuálně vyhodnotí řešení všech otázek jako správné, ale submission_autograder mi vyhodnotí řešení otázky 1 jako špatné při celkovém hodnocení.

```
  Question q1..................... 0/6
  Question q2..................... 6/6
  Question q3..................... 6/6
  Question q4..................... 7/7
Generating submission token..................................................... DONE
```