Allegro Summer Experience - Data Science

Rafał Kornel

Abstrakt

Celem zadania rekrutacyjnego było napisanie oraz wytrenowanie modelu regresji logistycznej algorytmem Stochastic Gradient Descend with momentum, bez używania gotowych bibliotek do sieci neuronowych, która miałaby kategoryzować ręcznie pisane cyfry, jako liczby pierwsze $(2, 3, 5, 7)$ lub liczby złożone $(4, 6, 8, 9)$. Zbiorem, na którym został wykonany projekt, był zbiór MNIST, zawierający ponad 60000 skategoryzowanych obrazków rozmiaru 28x28 pixeli, zawierających cyfry od 0 do 9. W modelu zostało przyjęte 0 - liczba pierwsza, oraz 1 - liczba złożona. Całość została napisana w języku Python (wersji 3.7), z wykorzystaniem bibliotek: Numpy, Matplotlib, Pickle, Sympy. Przeprowadzone obliczenia oraz eksperymenty pozwoliły wyznaczyć hiperparametry:

$$\text{Iteracje } I - 100$$

$$\text{Rozmiar batch'u } N - 300$$

$$\text{Tempo uczenia } \alpha - 3$$

$$\text{Współczynnik } \beta \text{ momentum} - 0.9$$

$$\text{Struktura sieci} - 784 \to 64 \to 10 \to 5 \to 1$$

$$\text{Funkcja aktywująca} - \text{Sigmoid (funkcja logistyczna)}$$

Trafność modelu została oszacowana na:

$$\text{Trafność} = 0.96423589$$

Wprowadzenie teoretyczne

Model regresji liniowej jest używany powszechnie, gdy zadaniem jest klasyfikacja, a wynikiem jest zmienna binarna - taka, która przyjmuje jedną z dwóch wartości. Dla uproszczenia przyjmuje się, że zmienna zwrócona przez sieć przyjmuje wartości z przedziału $[0, 1]$. Na tę konkretną sieć neuronową można patrzeć jako na funkcję

$$N(\mathbf{x}_i)_{\mathbf{W},\mathbf{B}} : \mathbb{R}^n \to [0, 1]$$

co oznacza że przyjmuje ona wektor $\mathbf{x}_i$, oraz zależy od parametrów $\mathbf{W}$ oraz $\mathbf{B}$, przy czym są to w ogólności macierze, bądź struktury złożone z macierzy. Przez $\mathbf{W}$ rozumie się wagi sieci, zatem macierze reprezentujące wagi pomiędzy poszczególnymi warstwami sieci, a $\mathbf{B}$ - bias'y, czyli wyrazy wolne. Aby wytrenować sieć została używa funkcja kosztu:

$$C(N(\mathbf{x}_i), y_i) = -y \cdot \log(N(\mathbf{x}_i)) - (1 - y) \cdot \log(1 - N(\mathbf{x}_i))$$

gdzie przez $N(\mathbf{x}_i)$ należy rozumieć wynik działania sieci na wektor $\mathbf{x}_i$. W rzeczywistej implementacji, wyraz $N(\mathbf{x}_i)$ będzie równy funkcji sigmoidalnej

$$\sigma(\mathbf{x}) = \frac{1}{1 + e^{-x}},$$

która jest traktowana jako funkcja aktywująca neurony w sieci. Do trenowania zastosowany został algrytm SGD with momentum, który w implementacji jest równy następującym równaniom:

$$\delta^L = \nabla_a C \odot \sigma'(z^L), \tag{1}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l), \tag{2}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j, \tag{3}$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j, \tag{4}$$

Gdzie wskaźnik górny L oznacza ostatni indeks w strukturze sieci, wartości z indeksem l przebiegają pozostałe warstwy sieci, $\nabla_a C$ to gradient funkcji kosztu po wielkościach a, która jest równa $a^l_j = \sigma(z^l_j) = \sigma(\Sigma_k w^l_{jk} a^{l-1}_k + b^l_j)$ lub wektorowo $\mathbf{a}^l = \sigma(\mathbf{w}^l \cdot \mathbf{a}^{l-1} + \mathbf{b}^l)$, zatem a odpowiadają wartościom neuronów, zaś $z^l_j = \Sigma_k w^l_{jk} \cdot a^{l-1}_k + b^l_j$ lub wektorowo $\mathbf{z}^l = \mathbf{w}^l \cdot \mathbf{a}^{l-1} + \mathbf{b}^l$, jest po prostu liniową kombinacją neuronów poprzedniej warstwy, z wagami oraz biasem. Dalej $\odot$ jest to produkt Hadamard'a, czyli w języku numpy operacja np.array * np.array (czyli wymnożenie wektorów wartość po wartości). Wielkość δ jest tutaj wporwadzona, aby ułatwić implementację. Podane równania zostały zaczerpnięte z [1]. Na podstawie powyższych równań (1-4) można obliczyć gradient funkcji kosztu względem wartości wag oraz bias'ów. Następnie został wykorzystany algorytm momentum, dzięki któremu model szybciej zbiega do optymalnego minimum. Matematycznie wyraża się on następująco: na podstawie (3) oraz (4) obliczamy o ile powinny zmienić się wagi oraz biasy, w^l_{jk} oraz b^l_j, przy klasycznym SGD zmodyfikowalibyśmy je o:

$$w^l_{jk} = w^l_{jk} - \frac{\alpha}{N} dw^l_{jk}, \qquad dw^l_{jk} = \frac{\partial C}{\partial w^l_{jk}},$$

$$b^l_j = b^l_j - \frac{\alpha}{N} db^l_j, \qquad db^l_j = \frac{\partial C}{\partial b^l_j},$$

gdzie α jest tempem uczenia (długością kroku), a N jest liczbą danych w batch'u. Jednak wprowadzenie momentum modyfikuje te równania, mianowicie wprowadzane są dwie wielkości:

$$V_{w^l_{jk}} = \beta V_{w^l_{jk}} + (1-\beta) dw^l_{jk},$$

$$V_{b^l_j} = \beta V_{b^l_j} + (1-\beta) db^l_j,$$

przy czym dla pierwszej iteracji macierze V_{w^l} i V_{b^l} są wypełnione zerami, i mają kształt kolejno $\mathbf{w}$ oraz $\mathbf{b}$. Korzystając z tych wielkości modyfikuje się wagi i bias'y w sposób następujący:

$$w^l_{jk} = w^l_{jk} - \frac{\alpha}{N} V_{w^l_{jk}},$$

$$b^l_j = b^l_j - \frac{\alpha}{N} V_{b^l_j}.$$

Momentum efektywnie uwzględnia "historię" wielkości, na której jest stosowany, w naszym przypadku gradient. Pozwala szybciej znaleźć szukane minimum parametrów.

Struktura kodu

Program został podzielony na 3 pliki źródłowe:

1. main.py - kod odpowiedzialny za uruchomienie algorytmu.
2. model.py - plik zawierający klasę Model, która jest odpowiedzialna za działanie algorytmu.
3. parser.py - odpowiada za poprawne czytanie plików (były to pliki binarne), odkodowanie ich oraz przechowywanie danych w jednym obiekcie.

Parser

Klasa Parser posiada (poza konstruktorem) metodę *parse_files(features_filename, labels_filename)*, która otwiera podane jako argumenty pliki oraz zwraca pythonowe listy zawierające informacje o obrazkach (macierze numpy o wartościach z przedziału $[0, 1]$, odpowiadające szarości pikseli) oraz klasyfikacje MNIST (listy 10-elementowe, odpowiadające wartości liczby, np [1, 0, 0, 0, 0, 0, 0, 0, 0, 0] -> jest to zero). Obiekty klasy Parser posiadają pole *data*, które zawiera tuple - pary obrazek - opis. Jest to główna struktura przechowująca dane.

Model

Klasa model, poza konstruktorem:

```
class Model():
    def __init__(self, layers, activation_function, cost_function, filename=""):

        self.layers = layers
        self.act = activation_function
        self.cost = cost_function
        self.act_der = lambdify(x, diff(self.act(x), x))

        if filename != "":
            with open(filename, "rb") as f:
                self.l = pickle.load(f)
                self.b = pickle.load(f)
                self.z = pickle.load(f)
                self.d = pickle.load(f)
                self.w = pickle.load(f)
        else:
            self.l = np.array([ np.array([ 0.0 for _ in range(i) ]) for i in layers ])
            self.b = np.array([ np.array([random.random() for x in range(len(self.l[i])) ]) \
                                                    for i in range(len(self.l))])
            self.z = copy.deepcopy(self.l)
            self.d = copy.deepcopy(self.z)
            self.w = [ np.random.uniform(low=-1.0, high=1.0, size=(len(self.l[i+1]), \
                                len(self.l[i]))) for i in range(len(self.l) - 1) ]
```

który definiuje następujące pola:

1. layers - struktura sieci
2. act - funkcja aktywująca
3. cost - funkcja kosztu
4. act_der - pochodna funkcji aktywującej (jedyne miejsce w którym używa została bibliotek sympy)
5. l - wartości neuronów (np.array wektorów)
6. b - wartości biasów (np.array wektorów)
7. w - wartości wag (lista macierzy np.array)
8. z - wartości neuronów **przed** zastosowaniem funkcji aktywującej (np.array wektorów)
9. d - macierz odpowiadająca δ z równania (1) oraz (2) (np.array wektorów)

posiada następujące funkcje:

1. validate - liczy celność modelu oraz średnią wartość funkcji kosztu, na danym batch'u danych validacyjnych
2. predict - zwraca przewidywaną wartość y dla wektora $\mathbf{x}$

3. backpropagation - metoda licząca poprawki do wag oraz biasów, które należy wprowadzić aby zbliżyć się do minimum
4. train - metoda realizująca równania (1-4), razem z wyżej wspomnianą metodą backpropagation, oraz SGD w. momentum
5. serialize - zapisuje wszystkie macierze do pliku binarnego, aby utworzyć model z gotowego pliku, należy w konstruktorze przekazać mu zmienną *filename* odpowiadająca nazwie pliku

Klasa Model jest główną "maszynką", realizującą algorytm.

Main

Główny plik programu, *main.py*, odpowiada za sklejenie wszystkich mniejszych komponentów programu w całość. Po preambule importującej niezbędne biblioteki następuje definicja funkcji:

```
def sigmoid(x): return 1/(1+np.e**(-x))
def cost(y, label): return -label*np.log(sigmoid(y)) - (1-label)*np.log(1-sigmoid
    (y))
```

są to omówione wcześniej funkcje sigmoid, oraz funkcja kosztu.

```
mnist_train = Parser('train-images-idx3-ubyte', 'train-labels-idx1-ubyte')
mnist_valid = Parser('t10k-images.idx3-ubyte', 't10k-labels.idx1-ubyte')
```

Następuje tu zdefiniowanie obiektów klasy Parser, przechowujących dane w odpowiednich formatach. *mnist_train* to dane treningowe, zaś *mnist_valid* przechowuje zbiór walidacyjny.

```
network          = [784, 64, 10, 5, 1]
batch            = 300
learning_rate    = 3
iterations       = 100
momentum         = 0.9
```

W tym miejscu definiowane są hiperparametry, użyte dalej do wywołania funkcji trenującej sieć.

```
test_model = Model(network, sigmoid, cost)
test_model.train(mnist_train.data, batch, learning_rate, iterations, momentum,
    valid=mnist_valid.data)

v, c = test_model.validate(mnist_valid.data)
print(f"Accuracy and avg. cost of model over whole validation set: {v}, {c}")
```

Ten kawałek odpowiada za stworzenie modelu, wytrenowanie go, oraz policzenie trafności i średniej wartości funkcji kosztu na całym zbiorze walidacyjnym. Warto zwrócić uwagę, że obecność zmiennej *valid=mnist_valid.data* gwarantuje walidacje modelu po każdej iteracji, na podzbiorze zbioru walidacyjnego, o rozmiarze N (patrz metoda Model.train). Program następnie wypisuje policzoną estymacje modelu, oraz średnią wartość funkcji kosztu, po całym zbiorze walidacyjnym.

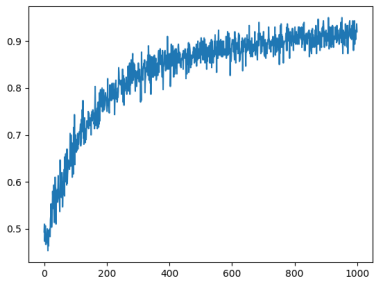
```
 name = f"{str(network).replace(' ', '')}_b{batch}_lr{learning_rate}_i{iterations
    }_m{momentum}_acc{v:.5}"
with open(name, "wb") as f:
    model.serialize(f)

#recovered_model = Model(network, sigmoid, cost, filename=name)
```

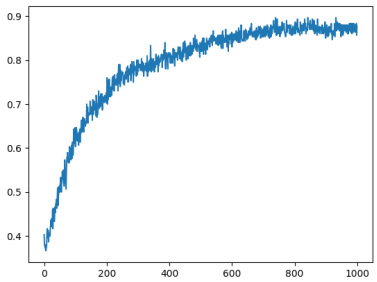
Ostatni fragment kodu zapisuje model do pliku o nazwie w zmiennej *name*. W komentarzu jest pokazany przykład, jak tworzyć obiekty modelu z zapisanego wcześniej pliku. Ważny jest fakt, że nadal trzeba podać w konstruktorze strukture sieci, oraz funkcje aktywacyjną i kosztu.

Eksperymenty, wyniki

Na modelu przprowadzone zostały wielokrotne testy. Wcześniej wspomniane fragmenty kodu i metody pozwoliły śledzić poprawność przewidywań sieci. Klasyfikacja modelu (z racji że sieć zwraca liczbę z przedziału $[0, 1]$ została rozwiązana w następujący sposób: jeśli model zwróci liczbę powyżej 0.5, uznajemy że przewiduje wartość 1, jeżeli poniżej uznajemy 0. Pozwoliło to na wprowadzenie oceny modelu, jako liczbę poprawnie sklasyfikowanych wejść dzieloną przez liczbę danych. Pierwsza ważna konkluzja: algorytm momentum znacznie poprawia stabilność modelu podczas trenowania. Poniżej przedstawione są dwa wykresy trafności modelu, na nieoptymalnych parametrach.



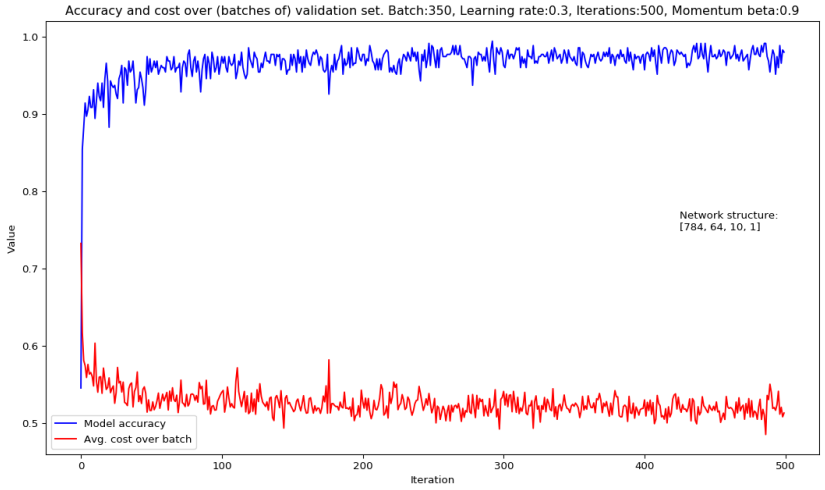
(a) Wykres trafności modelu dla nieoptymalnych parametrów, brak momentum ($\beta = 0$)



(b) Wykres trafności modelu dla nieoptymalnych parametrów, z algorytmem momentum ($\beta = 0.95$)

Rys. 1: Porównanie modelu bez algorytmu momentum, oraz z

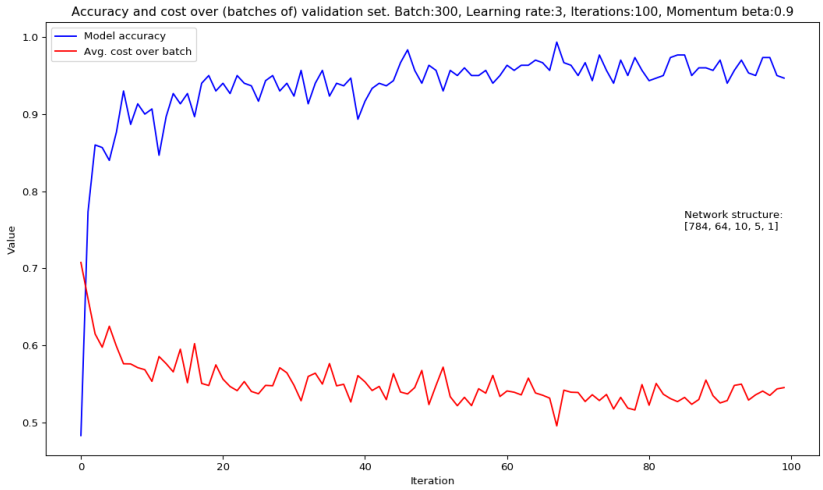
Widać od razu, iż brak algorytmu momentum powoduje większe oscylacje modelu. Kolejną bardzo istotną konkluzją, podczas szukania odpowiednich hiperparametrów jest fakt, że zbieżność modelu zwiększyła się niewiarygodnie dużo, po zwiększeniu pierwszej ukrytej warstwy. Jak widać na [Rys.1], algorytm uzyskał poziom celności w okolicy 90% dopiero po kilkuset iteracjach. Na poniższym rysunku [Rys.2] znajduje się wykres trafności dla algorytmu, po zwiększeniu pierwszej ukrytej warstwy z 10 do 64.



Rys. 2: Trafność algorytmu dla bardziej optymalnych wartości hiperparametrów, wraz ze zwiększoną siecią.

W tym wypadku parametry nie są optymalne, ponieważ sieć bardzo szybko zbiegła do satysfakcjonującego poziomu wiarygodności.

Ostatecznie, najbardziej optymalne parametry zostały ustalone w takiej postaci, jak w abstrakcie. Wykres trafności modelu wygląda następująco:



Rys. 3: Trafność algorytmu dla parametrów ustalonych jako najbardziej optymalne. Model bardzo szybko osiąga akceptowalną skuteczność, a po nasyceniu nieznacznie oscyluje.

Dla takich parametrów, skuteczność modelu została oszacowana (metodą omówioną powyżej), na:

$$\text{Trafność} = 0.96423589$$

Jest to wynik bardzo satysfakcjonujący. Potwierdza on, iż wytrenowana na zbiorze treningowym sieć potrafi dobrze rozpoznać dane o podobnych własnościach, pochodzących z odseparowanego zbioru (zbioru walidacyjnego). Ta konkluzja kończy omówienie doświadczenia. Cały kod znajdować się będzie na repozytorium w serwisie github. Ponadto, plik wytrenowanego, ostatecznego modelu został nazwany *final.model*, a także załączony razem z kodem i sprawozdaniem.

References

[1] http://neuralnetworksanddeeplearning.com/chap2.html