WOJSKOWA AKADEMIA TECHNICZNA

im. Jarosława Dąbrowskiego

WYDZIAŁ CYBERNETYKI

Sprawozdanie

Zaawansowane metody uczenia maszynowego

Deep learning - Wprowadzenie

Autor:

Karol Baranowski

Prowadzący:

mgr inż. Przemysław Czuba

Spis treści

Zadanie

Zaimplementuj regresję logistyczną do klasyfikacji kotów.

Uzupełnione fragmenty kodu

Autor w celu sprawniejszego testowania uzupełnionych funkcji rozwiązał zadanie w Jupyer Notebook i załączył je w formacie w formacie .ipynb (python notebook).

Inicjalizacja zerami

```
def initialize_with_zeros(dim):
    """
    This function creates a vector of zeros of shape (dim, 1) for w and initializes b to 0.

    Argument:
    dim -- size of the w vector we want (or number of parameters in this case)

    Returns:
    w -- initialized vector of shape (dim, 1)
    b -- initialized scalar (corresponds to the bias)
    """

    ### KOD ### (~1 linijka)
    w = np.zeros((dim, 1)) # dim rows 1 column
    b = 0
    ### KOD ###

    assert (w.shape == (dim, 1))
    assert (isinstance(b, float) or isinstance(b, int))

    return w, b

dim = 2 # 2 rows
w, b = initialize_with_zeros(dim)
print("w = " + str(w))
print("b = " + str(b))
```

```
w = [[0.]
 [0.]]
b = 0
```

Powyżej w funkcji *initalize_with_zeros* zainicjalizowano wektor wag o długości podanej w parametrze *dim* (technicznie to macierz z jedną kolumną i *dim* wierszami). Bias *b* ustawiono jako wartość 0. Pierwsze dane nie muszą być przesunięte o bias. Zarówno wagi jak i bias będą liczone w przyszłych iteracjach.

Propagacja

```
In [100]:
def propagate(w, b, X, Y):
    """
    Implement the cost function and its gradient for the propagation explained above
    Arguments:
    w -- weights, a numpy array of size (num_px * num_px * 3, 1)
    b -- bias, a scalar
    X -- data of size (num_px * num_px * 3, number of examples)
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat) of size (1, number of examples)
    Return:
    cost -- negative log-likelihood cost for logistic regression
    dw -- gradient of the loss with respect to w, thus same shape as w
    db -- gradient of the loss with respect to b, thus same shape as b
    Tips:
    - Write your code step by step for the propagation. np.log(), np.dot()
    """
    m = X.shape[1]
    # FORWARD PROPAGATION (FROM X TO COST)
    ### KOD ### (≈ 2 linijki)
    # compute activation
    z = np.dot(w.T, X) + b
    a = sigmoid(z)
    # compute cost
    loss = - Y * np.log(a) - (1 - Y) * np.log(1 - a)
    cost = (1 / m) * np.sum(loss)
    ### KOD ###
    # BACKWARD PROPAGATION (TO FIND GRAD)
    ### KOD ### (≈ 2 linijki)
    # db
    dz = a - Y
    db = (1 / m) * np.sum(dz)
    # dw
    dw = (1 / m) * np.dot(X, dz.T)
    ### KOD ###
    assert (db.dtype == float)
    assert(dw.shape == w.shape)
    cost = np.squeeze(cost)
    assert (cost.shape == ())
    grads = {"dw": dw,
             "db": db}
    return grads, cost
w, b, X, Y = np.array([[1], [2]]), 2, np.array([[1, 2], [3, 4]]), np.array([[1, 0]])
grads, cost = propagate(w, b, X, Y)
print("dw = " + str(grads["dw"]))
print("db = " + str(grads["db"]))
print("cost = " + str(cost))
```

```
dw = [[0.99993216]
 [1.99980262]]
db = 0.49993523062470574
cost = 6.000064773192205
```

W powyżej funkcji należało uzupełnić kod dla *forward propagation* wedle wzorów:

Loss function dla pojedynczego przykładu $x^{(i)}$:

$$z^{(i)} = w^T x^{(i)} + b$$

$$\hat{y}^{(i)} = a^{(i)} = sigmoid(z^{(i)})$$

$$\mathcal{L}(a^{(i)}, y^{(i)}) = -y^{(i)}\log(a^{(i)}) - (1 - y^{(i)})\log(1 - a^{(i)})$$

Cost function:

$$J = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)})$$

z jest wynikiem mnożenia wektora transponowanego wag *w* i wektora danych wejściowych *X*, i jest on przesunięty o wartość biasu *b*. *w* jest transponowany aby udało się mnożenie macierzowe *np.dot()* tzn. „wiersz przez kolumnę". *z* określa aktywację neuronu na wyjściu natomiast użycie na niej funkcji *sigmoid* zwraca znormalizowaną wartość spomiędzy 0 a 1 zapisaną w zmiennej *a*. Funkcja *sigmoid* dla ujemnych wartości dąży do 0, dla pozytywnych dąży do 1, natomiast w okolicach zera rośnie gwałtownie. Dalej liczona jest funkcja straty i jej wyniki zapisywane w wektorze *loss* (dla wszystkich neuronów) określająca (z użyciem logarytmów) różnicę danych wyjściowych i wyliczonej aktywacji. Następnie jest wyliczana średnia arytmetyczna z elementów wektora *loss* i zapisywana jako wartość zmiennej *cost*.

Kod dla backward propagation przekazuje jak należy zmienić poszczególne parametry (poprzez wartości ich pochodnych, które określają czy funkcja rośnie, czy maleje w zależności od zmian zadanego parametru i jak bardzo) aby zminimalizować funkcję kosztu w następnej iteracji za pomocą metody *gradient descent*, która znajduje minimum lokalne dla zadanej funkcji. Kod należało uzupełnić wedle wzorów:

$$dZ = A - Y$$

$$dW = \frac{1}{m} X dZ^T$$

$$db = \frac{1}{m} sum(dZ)$$

Wyliczone pochodne *dw* (wektor) i *db* (liczba) wraz wartością kosztu są zwracane na końcu funkcji, a jej działanie prezentują wyliczone wartości dla przykładowych wartości *w, b, X, Y*.

Optymalizacja

```python
def optimize(w, b, X, Y, num_iterations, learning_rate, print_cost=True):
    """
    This function optimizes w and b by running a gradient descent algorithm
    Arguments:
    w -- weights, a numpy array of size (num_px * num_px * 3, 1)
    b -- bias, a scalar
    X -- data of shape (num_px * num_px * 3, number of examples)
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat), of shape (1, number of examples)
    num_iterations -- number of iterations of the optimization loop
    learning_rate -- learning rate of the gradient descent update rule
    print_cost -- True to print the loss every 100 steps
    Returns:
    params -- dictionary containing the weights w and bias b
    grads -- dictionary containing the gradients of the weights and bias with respect to the cost function
    costs -- list of all the costs computed during the optimization, this will be used to plot the learning curve.

    Tips:
    You basically need to write down two steps and iterate through them:
        1) Calculate the cost and the gradient for the current parameters. Use propagate().
        2) Update the parameters using gradient descent rule for w and b.
    """
    costs = []
    for i in range(num_iterations):
        grads, cost = propagate(w, b, X, Y)
        # Retrieve derivatives from grads
        dw = grads["dw"]
        db = grads["db"]
        # gradient update rule (≈ 2 lines of code)
        ### KOD ###
        w = w - learning_rate * dw
        ### KOD ###
        b = b - learning_rate * db
        # Record the costs
        if i % 100 == 0:
            costs.append(cost)
        # Print the cost every 100 training examples
        if print_cost and i % 100 == 0:
            print("Cost after iteration %i: %f" % (i, cost))
    params = {"w": w,
              "b": b}
    grads = {"dw": dw,
             "db": db}
    return params, grads, costs
```

Optymalizacja funkcji kosztu polega na aktualizowaniu wartości wektora *w* i biasu *b* poprzez dodanie/odjęcie odpowiadającej wartości pochodnej pomnożonej przez *learning_rate* wedle wzorów:

$$w = w - \alpha \times dw, \quad b = b - \alpha \times db$$

Całe postępowanie należy powtórzyć “wybraną” liczbę iteracji. *learning_rate* (α we wzorze) mnożony z pochodną określa o ile przesuwamy się podczas jednej iteracji, a znak pochodnej kierunek przesunięcia (należy dodać gdy pochodna ujemna i odjąć gdy dodatnia).

Predykcja

```python
def predict(w, b, X):
    '''
    Predict whether the label is 0 or 1 using learned logistic regression parameters (w, b)

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px * 3, 1)
    b -- bias, a scalar
    X -- data of size (num_px * num_px * 3, number of examples)

    Returns:
    Y_prediction -- a numpy array (vector) containing all predictions (0/1) for the examples in X
    '''

    m = X.shape[1]
    Y_prediction = np.zeros((1, m))
    w = w.reshape(X.shape[0], 1)
    # Compute vector "A" (a in my case) predicting the probabilities of a cat being present in the picture
    ### KOD ### (≈ 1 line of code)
    a = sigmoid(np.dot(w.T, X) + b)
    ### KOD ###
    for i in range(a.shape[1]):
    # Convert probabilities A[0,i] to actual predictions p[0,i]
    ### KOD ### (≈ 4 lines of code)
        if a[0][i] > 0.5:
            Y_prediction[0][i] = 1
        else:
            Y_prediction[0][i] = 0
    ### KOD ###

    assert (Y_prediction.shape == (1, m))

    return Y_prediction
```

W funkcji *predict* konwertowany jest (po wszystkich iteracjach minimalizowania funkcji kosztu i zmianach parametrów) wynik poszczególnych aktywacji na binarne wartości 0 lub 1 w zależności czy uzyskał wartość większą od 0.5 czy mniejszą. W pętli for *a.shape[1]* określa ilość neuronów, a *a[0][i]* (macierz jednowierszowa) wartość aktywacji iterowanego neuronu.

Model

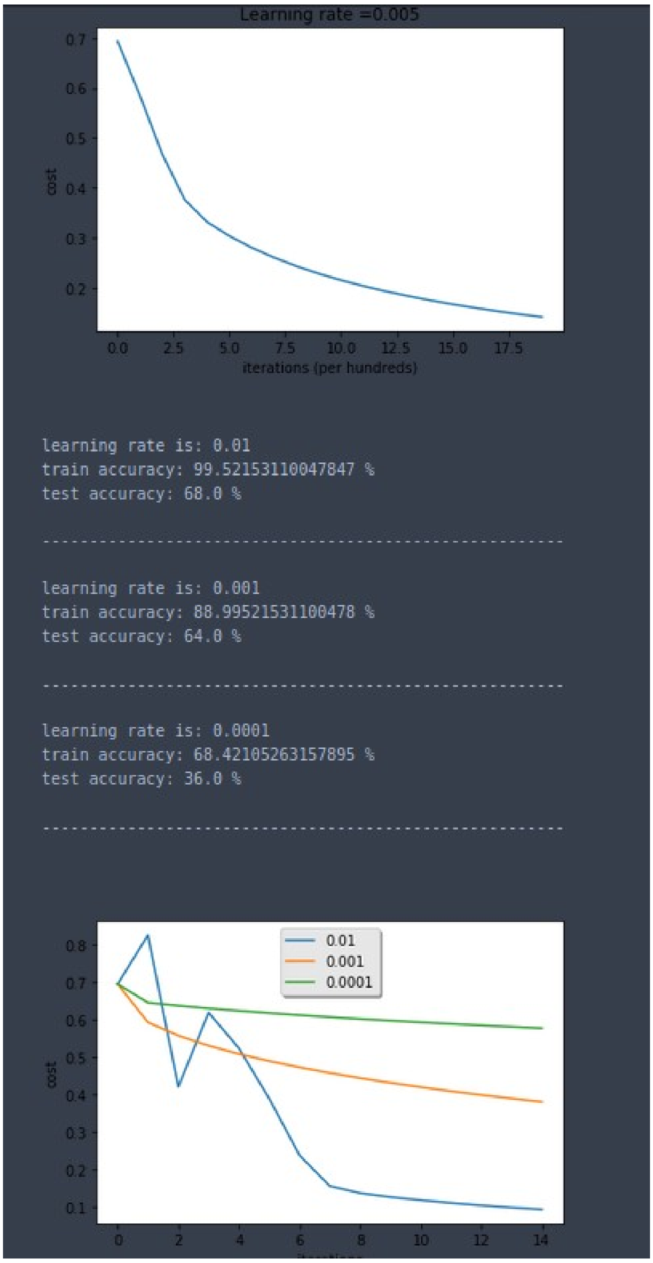
```
### KOD ###

# initialize parameters with zeros (≈ 1 line of code)
w, b = initialize_with_zeros(X_train.shape[0])
# Gradient descent (≈ 1 line of code)
params, grads, costs = optimize(w, b, X_train, Y_train, num_iterations, learning_rate, print_cost)
# Retrieve parameters w and b from dictionary "parameters"
w = params["w"]
b = params["b"]
# Predict test/train set examples (≈ 2 lines of code)
Y_prediction_test = predict(w, b, X_test)
Y_prediction_train = predict(w, b, X_train)
### KOD ###

# Print train/test Errors
print("train accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_train - Y_train)) * 100))
print("test accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_test - Y_test)) * 100))
```

W funkcji model należało uzupełnić kod o wywołanie wcześniej opisanych funkcji w odpowiedniej kolejności z danymi treningowymi i testowymi dla rozpoznawania kotów. Uzupełniono wektor wag ilością zer odpowiadającą 64* 64*3 neuronom (px * px * kolory) oraz ustawiono bias na 0. Następnie wywołano funkcję optymalizującą, zwracającą dopasowane parametry *w* i *b* na podstawie treningowych danych wejściowych i wyjściowych. W jej środku określoną liczbę razy wywoływano funkcję realizującą propagację i propagację wsteczną. Ostatecznie wyliczono wartości wyjściowego neuronu dla danych wejściowych treningowych i testowych, aby je porównać po znormalizowaniu.

```
Cost after iteration 0: 0.693147
Cost after iteration 100: 0.584508
Cost after iteration 200: 0.466949
Cost after iteration 300: 0.376007
Cost after iteration 400: 0.331463
Cost after iteration 500: 0.303273
Cost after iteration 600: 0.279880
Cost after iteration 700: 0.260042
Cost after iteration 800: 0.242941
Cost after iteration 900: 0.228004
Cost after iteration 1000: 0.214820
Cost after iteration 1100: 0.203078
Cost after iteration 1200: 0.192544
Cost after iteration 1300: 0.183033
Cost after iteration 1400: 0.174399
Cost after iteration 1500: 0.166521
Cost after iteration 1600: 0.159305
Cost after iteration 1700: 0.152667
Cost after iteration 1800: 0.146542
Cost after iteration 1900: 0.140872
train accuracy: 99.04306220095694 %
test accuracy: 70.0 %
```

Wnioski

Cel zadania został osiągnięty. Sieć zinterpretowała 50 obrazków testowych w 70% poprawnie po przeszkoleniu przez 209 obrazków treningowych (interpretacja na pierwszym wykresie). Działanie sieci można by poprawić dodając dane treningowe, dodając ukryte warstwy, dodając ilość iteracji i modyfikując parametr *learning_rate*.



Z powyżej wygenerowanych wykresów widać, że dobór *learning_rate* przekłada się na stopień dokładności predykcji sieci. Gdy jest on zbyt mały

(0.0001) należałoby powtórzyć propagację o wiele większą ilość iteracji aby zminimalizować funkcję kosztu, więc algorytm jest wtedy za wolny. Gdy jest on zbyt duży, koszt rośnie i maleje naprzemiennie o duże wartości w jednej iteracji i możliwe jest „przeskoczenie" minimum.