Machine Learning, Hand in 1

Adam Petro (au679613), Jakub Krzyzynski (au679615), Anton D. Lautrup (au590794)

Department of Computer Science
Aarhus University
Aarhus, Denmark

October 2020

Part I: Logistic Regression

Summary and Results

We tweak the parameters; *learning rate* `lr`, *mini-batch size* `batch_size`, and the number of `epochs`, and recieve a test score of just above 95% accuracy out of sample, which must be considered pretty good. The parameters we used to achieve this were `lr=1.7, batch_size=4, epochs=300`, with resulting in sample score at 0.952495 and test score 0.950665. The figure (1) obtained from `logistic_test` is provided on the next page. Note that the cost goes up a little in one of the first epochs, this is probably due to the high learning rate overshooting one initial local minima.

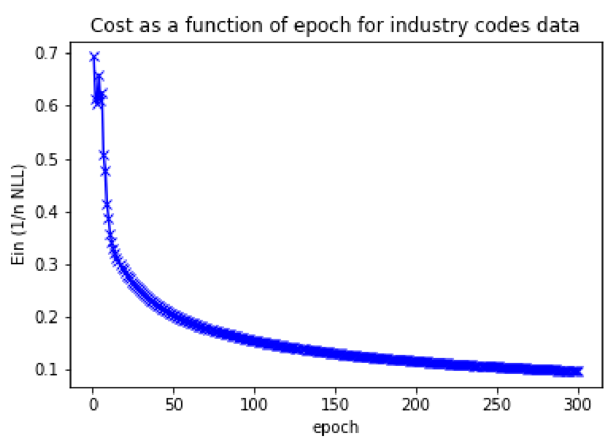

Figure 1: The parameters on the figure were `lr=1.7, batch_size=4, epochs=300`, the test score = 0.950665 and in sample score = 0.952495

Actual Code

First is the `cost_grad` implementation, where we loop over the size of the data, and for each datapoint, calculate the log of the cost function and the gradient, and put them in lists `c`, and `g`. Finally, we evaluate the average on both lists, by summation and division by the number of elements, before we return the `cost` and `grad` as outputs. The formula we modelled were (3.9) and the result of Exercise 3.7 from [LFD]

```
def cost_grad(self, X, y, w):
    cost = 0;grad = np.zeros(w.shape);
    c=[];g=[];
    logi = lambda a: 1/(1+np.exp(-a))

    for i in range(len(y)):
        c.append(np.log(1+np.exp(np.dot(X[i],w.T)*-y[i])))
        g.append((-y[i]*X[i])*logi(-y[i]*np.dot(w.T,X[i])))

    cost = (1/len(y))*np.sum(c)
    grad = (1/len(y))*np.sum(g,axis=0)
    assert grad.shape == w.shape
    return cost, grad
```

Next is the `Fit` function, first we stick together the `X` and `y` lists, to keep track of `X,y` pairs during permutation, and we calculate a number `n_b`, that defines the amount of data in each mini-batch. Then, in a loop over the epochs, we retrieve a random permutation of the `X_y`, reseperate the `X`s and ys, and inputs the first `n_b` elements of both into the `cost_grad` function. Finally we take the resulting gradient to calculate a new `w`, and append the cost to the `history` list.

```
def fit(self, X, y, w=None, lr=0.1, batch_size=16, epochs=10):
    if w is None: w = np.zeros(X.shape[1])
    history = []
    X_y = np.c_[X,y]
    n_b = int(np.floor(len(X_y)/batch_size))

    for i in range(epochs):
        np.random.permutation(X_y)

        temp_x = X_y[:,:-1]
        temp_y = X_y[:,-1]
        cost,grad = self.cost_grad(temp_x[:n_b],(temp_y[:n_b]),w)
        w = w-lr*grad

        history.append(cost)
    self.w = w
    self.history = history
```

More detailed description can be found in comments in the actual code. Those have been omitted here for clarity purposes.

Theory

Running time

We read data of size $n \times d$ in each epoch for *epochs* number of epochs. Therefore the running time is

$$\boldsymbol{epochs * n * d}$$

Sanity Check

Logistic regression does not use pixel locality, since it does not use any convolutions or pooling layers. It treats all pixels separately and does not gather any context information from nearby pixels. Therefore by shuffling pixels around, the classifier quality will be the same as without shuffling.

Linear Separability

There are infinitely many separators on linearly separable data and logistic regressions prefers larger weights. This means that it would not converge on any fixed solution but instead keep choosing larger weights.

Part II: Softmax

Summary and Results

In this script we implemented the Softmax activation function, with a learning algorithm, to determine weights. Adjusting parameters `lr, batch_size, epochs` we obtain reasonable results in the initial test on the wine data in the `softmax_test` code. We succeeded in obtaining a test score of 91.1111%, and in sample score of 96.2406% with parameters `lr=0.01, batch_size=10, epochs=600`. The plot 2 is provided below.

Proceeding with the next task, of applying the script to data from the MNIST, handwritten digit database, we just use the provided parameters `lr=0.42, batch_size=666, epochs=100`, and receive test accuracy of 67.6668% with in sample score 67.79%. These results are "okayish", but further work on the parameters would probably increase the accuracy.

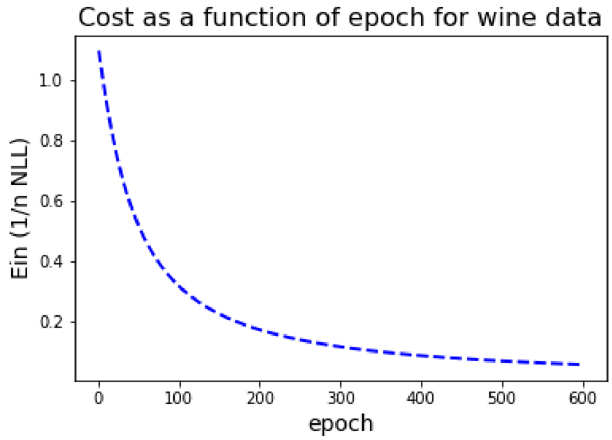

Figure 2: Plot generated from the wine data. The parameters on the figure were `lr=0.01, batch_size=10, epochs=600`, the test score = 0.911111 and in sample score = 0.962406

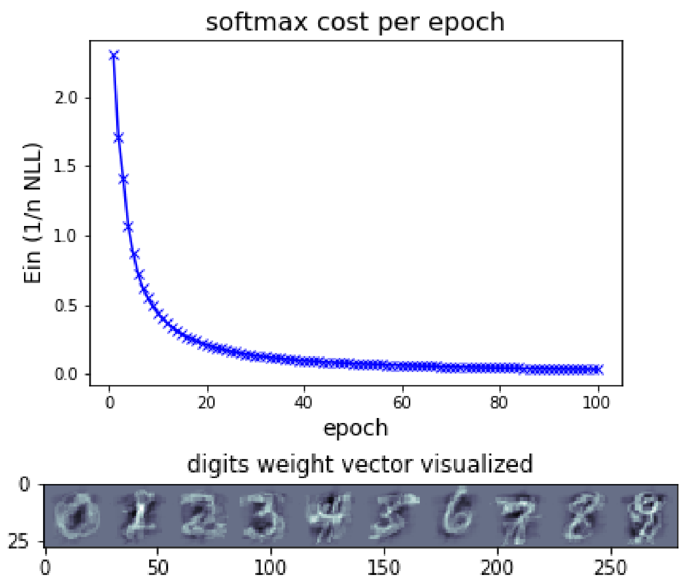

Figure 3: Top: Plot generated from the MNIST data. The parameters on the figure were `lr=0.42, batch_size=666, epochs=100`, the test score = 0.676668 and in sample score = 0.6779.
Bottom: Graphical representation of the weights associated with each of the handwritten digits.

Actual Code

We will only include the `cost_grad` in this section because the `fit` is mechanically identical to the one we used in the `logistic_regression` script.

```
def cost_grad(self, X, y, W):
    cost = np.nan
    grad = np.zeros(W.shape)*np.nan
    Yk = (one_in_k_encoding(y, self.num_classes))
    softmax_matrix = softmax(np.dot(X,W))

    cost = (-1/len(y))*np.sum(Yk*np.log(softmax_matrix))
    grad = (-(1/len(y))*np.dot(X.T,(Yk-softmax_matrix)))
    return cost, grad
```

This time we calculate everything in single steps, rather than using loops. First we map `y` to `Yk` by using the `one_in_k_encoding` function, this turns the `y`s into a clever matrix form. Next we calculate the `softmax_matrix` because we use it twice, and finally, we calculate the cost and the gradient, i accordance with the note on Softmax

Theory

This fit complexity is equivalent to fit function from logistic regression, so we have to multiply *epochs* times running time of `cost_grad`, so final complexity is

$$\boldsymbol{O(epochs * n * d * K)}$$