**Introduction to Machine Learning (Spring 2019)**

**Homework #1 (Due date: April 8)**

**Student ID 2014313873**

**Name 하승준**

**Instruction:** We provide all codes and datasets in Python. Please write your code to complete two models: linear regression and logistic regression. Besides, please measure the performance for each model.

1. **[30 pts]** Implementation
2. **[Linear regression]** Implement training and evaluation function in ‘models/LinearRegression.py’ (‘train’ and ‘eval’ respectively).

Train :

np.random.seed() #For random seed.

lst = np.zeros((x.shape[0],1)) #For random election

y = y.reshape(x.shape[0],1) #Make y two dimension array.

for epoch in range(epochs):

for j in range(x.shape[0]): #Random Election Part

lst[j] = j

np.random.shuffle(lst)

temp\_x = np.zeros((batch\_size, x.shape[1]))

temp\_y = np.zeros((batch\_size, 1))

for i in range(int(x.shape[0]/batch\_size)):

t = i\*10

for j in range(batch\_size):

temp\_x[j] = x[int(lst[t+j])]

temp\_y[j] = y[int(lst[t+j])]

y\_expected = np.dot(temp\_x, self.W)

grad = (-2.0/batch\_size)\*np.dot(np.transpose(temp\_x),(temp\_y-y\_expected))

self.W = optim.update(self.W, grad, lr)

start = int((x.shape[0]/batch\_size))\*batch\_size #If x.shape[0] % batch\_size != 0

finish = x.shape[0] #Do last iteration

remain=finish-start

if remain!=0:

temp\_x = np.zeros((remain, x.shape[1]))

temp\_y = np.zeros((remain, 1))

for j in range(remain):

temp\_x[j] = x[int(lst[start+j])]

temp\_y[j] = y[int(lst[start+j])]

y\_expected = np.dot(temp\_x, self.W)

grad = (-2.0/remain)\*np.dot(np.transpose(temp\_x),(temp\_y-y\_expected))

self.W = optim.update(self.W, grad, lr)

y = y.reshape(-1)

tmp = y-self.eval(x).reshape(-1)

final\_loss=np.dot(tmp,tmp)/x.shape[0] #Calculate final\_loss after all iter finished.

Eval :

pred = np.dot(x, self.W)

1. **[Logistic regression]** Implement training and evaluation function in ‘models/LogisticRegression.py’ (‘train’ and ‘eval’ respectively).

Train :

#Similar with Linear Regression

np.random.seed()

lst = np.zeros((x.shape[0], 1))

y = y.reshape(x.shape[0],1)

for epoch in range(epochs):

for j in range(x.shape[0]):

lst[j] = j

np.random.shuffle(lst)

temp\_x = np.zeros((batch\_size, x.shape[1]))

temp\_y = np.zeros((batch\_size, 1))

for i in range(x.shape[0]//batch\_size):

t = i\*batch\_size

for j in range(batch\_size):

temp\_x[j] = x[int(lst[t+j])]

temp\_y[j] = y[int(lst[t+j])]

y\_expected = self.\_sigmoid(np.dot(temp\_x, self.W))

grad = (1.0/batch\_size)\*np.dot(np.transpose(temp\_x), y\_expected-temp\_y)

self.W = optim.update(self.W, grad, lr)

start = (x.shape[0]//batch\_size)\*batch\_size

finish = x.shape[0]

remain = finish-start

if remain != 0 :

temp\_x = np.zeros((remain, x.shape[1]))

temp\_y = np.zeros((remain, 1))

for j in range(remain):

temp\_x[j] = x[int(lst[start+j])]

temp\_y[j] = y[int(lst[start+j])]

y\_expected = self.\_sigmoid(np.dot(temp\_x, self.W))

grad = (1.0/remain)\*np.dot(np.transpose(temp\_x), y\_expected-temp\_y)

self.W = optim.update(self.W, grad, lr)

h = np.dot(x, self.W)

final\_loss = -(1.0/x.shape[0])\*np.sum(h\*(y-1.0) - np.log(1.0+np.exp(-h))) #To avoid division by zero.

Eval :

last = self.\_sigmoid(np.dot(x, self.W))

pred = (last >= threshold).astype(int)

1. **[Optimization]** Implement SGD, Momentum, RMS Prop optimizers in ‘optim/Optmizer.py’. Training should be based on the minibatch, not the whole data.

SGD :

Pass

updated\_weight = w - lr\*grad

Momentum :

self.gamma = gamma

self.v = 0.0

self.v = self.gamma \* self.v + lr\*grad

updated\_weight = w - self.v

RMSProp :

self.gamma = gamma

self.epsilon = epsilon

self.G = 0.0

self.G = self.gamma\*self.G + (1.0-self.gamma)\*grad\*\*2

updated\_weight = w - (lr\*grad)/(np.sqrt(self.G+self.epsilon))

NOTE: You should write your codes in ‘EDIT HERE’ signs. It is not recommended to edit other parts. Once you complete your implementation, run the main codes to check if it is done correctly (‘linear\_main.py’ for Linear Regression and ‘logistic\_main.py’ for Logistic Regression).

1. **[30 pts]** Experimental results
2. **[Linear Regression]** For ‘Graduate’ and ‘Concrete’ dataset, adjust the number of training epochs and learning rate to minimize RMSE. Report your best results for each optimizer.   
   (Batch size = 10, epsilon = 0.01, gamma = 0.9)

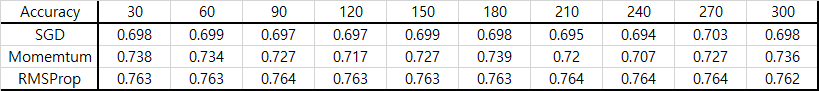
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Dataset** | **Optimizer** | **# of epochs** | **Learning rate** | **RMSE** |
| **Graduate** | SGD | 500 | 0.0005 | 0.08 |
| Momentum | 500 | 0.0005 | 0.08 |
| RMSProp | 500 | 0.0005 | 0.08 |
| **Concrete** | SGD | 40000 | 0.0005 | 11.65 |
| Momentum | 10000 | 0.0005 | 11.58 |
| RMSProp | 10000 | 0.0005 | 11.63 |

1. **[Logistic Regression]** For ‘Titanic’ and ‘Digit’ dataset, adjust the number of training epochs and learning rate to maximize accuracy. Report your best results for each optimizer.  
   (Batch size = 10, epsilon = 0.01, gamma = 0.9)

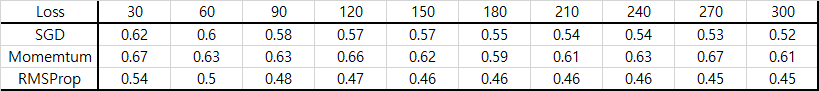
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Dataset** | **Optimizer** | **# of epochs** | **Learning rate** | **Acc.** |
| **Titanic** | SGD | 300 | 0.0005 | 0.833 |
| Momentum | 300 | 0.0005 | 0.833 |
| RMSprop | 3000 | 0.0005 | 0.852 |
| **Digit** | SGD | 600 | 0.001 | 0.994 |
| Momentum | 300 | 0.001 | 0.994 |
| RMSprop | 300 | 0.001 | 0.994 |

(c) **[Optimization]** For ‘Titanic’ dataset, execute the logistic regression with three optimization methods. Given the following parameter settings, draw two plots : a plot whose x-axis and y-axis are epochs and accuracy, and a plot whose x-axis and y-axis are epochs and cross-entropy loss. Explain which optimization method shows the best accuracy.

|  |  |
| --- | --- |
| **Parameter Settings** | |
| Batch size | 10 |
| Learning rate | 0.0005 |
| Epsilon | 0.01 |
| Gamma | 0.9 |
| # of Epochs | 30, 60, 90, …, 300 |



Accuracy(100 data’s average)



Loss(100 data’s average)

10개의 Batch를 고르는 것을 Epoch마다 완전 Random하게 해줬습니다. Sequential Batch기법을 사용하여 데이터의 앞에부터 10개씩 Batch를 뽑아서 시작하면, 첫 10개의 데이터가 먼저 학습되기 때문에 뒤로갈수록 그 10개의 경향성이 짙어질 것입니다. 이로 인해 특정 환경에서는 잘 작동할 수 있지만(초반에 존재하는 데이터가 모든 데이터의 경향성을 보여줄 수 있는 경우), 보편적인 경우에 최적의 학습을 못할 수도 있다고 생각이 들었습니다. 따라서 Epoch마다 첫 10개의 Batch를 Shuffle을 통하여 완전히 Random하게 뽑아줬으며, 이를 통해 Random 환경에서 학습을 하는 상황을 가정할 수 있었습니다. Random Batch를 사용할 경우, 1회 시행할 때마다 Accuracy가 다르게 나오는데, 이 때문에 정량적인 비교가 어려워 100번의 시행을 한 후 평균값을 통하여 그래프를 작성하였습니다.

다음으로 Data set에 대한 분석입니다. Titanic Dataset은 7가지의 독립 변수로 이루어져 있으며, 총 데이터의 개수는 779개로 Train을 하고, 108개로 Test를 하는 Dataset입니다. 7개의 독립변수들의 학습률에 따라서 최종적으로 Accuracy가 결정기 때문에, RMSProp같이 Weight에대해 Running 중 Learning Rate를 조절하는 Optimizer가 잘 작동할 가능성이 높다고 생각됩니다.

각각의 Optimizer에 대해 간단하게 설명하자면, SGD는 평범한 경사하강법을 의미합니다. Momentum은 이 전 Gradient의 방향성을 Gamma만큼 유지하는 경사하강법이며 SGD에 존재하는 Oscilation을 해결하기 위해 등장하였습니다. 이는 SGD에 관성을 주어 경향성을 유지하는 기법으로 SGD보다 좀 더 빠른 수렴을 할 수 있도록 해줍니다. 마지막으로 RMSProp은 Adagrad에서 확장된 개념입니다. 먼저 Adagrad는 지금까지 많이 변화한 Weight들의 Step size는 작게하고, 많이 변화하지 않은 Weight들의 Step size는 크게하자는 취지에서 등장하였습니다. 그러나 Adagrad의 Weight를 구하는 식에서 Epoch를 진행할 때 Epoch+1번째의 Weight를 구하기 위해서 Epoch번째의 Gradient를 제곱할 때 해당 값이 무한정 커질 수도 있고, 그 값이 너무 작아져서 학습이 제대로 되지 않을 수 있는 단점이 있었습니다. 이에 RMSProp은 Gradient제곱 값에 대해 지수평균을 적용하였으며, 이 과정에서 Gamma값이 사용됩니다. (Epsilon은 제곱된 Gradient를 Root를 씌워서 Learning\_Rate\*(Epoch’s Gradient)에 나눠주는데, 이 과정에서 부동소숫점 연산을 하기 떄문에 일정 값 밑으로 내려가서 0으로 나눠지는 경우를 막기위해 더해주는 값 입니다.)

SGD, Momentum, RMSProp 중 RMSProp이 가장 적은 Loss와 높은 Accuracy를 보여주고 있습니다. 그리고 또한 Accuracy의 분포도 상당히 밀집되고 안정적입니다.(다음 페이지에 Epoch별 100개의 Accuracy Data에 대한 그래프를 첨부하겠습니다.) 이는 RMSProp이 Epoch가 진행되면 진행될수록 Weight간의 학습률을 고려하여 충분히 학습된 것은 그 학습률을 낮추고, 덜 학습된 것은 그 학습률을 높여 Weight간의 학습률에 차등을 둬서 학습하기 때문에 안정적이며, 높은 학습률을 보인다고 해석할 수 있습니다. 즉, Titanic Dataset에서 7가지의 독립변수가 모두 같은 비중이 아닐 것이며, 학습되는 Batch\_Data의 순서에 따라서도 다른 학습률 및 경향을 보일 것입니다. 이를 Momentum이나 SGD에서는 7개의 독립변수에 대해서 Weight를 모두 차등없이 학습시킬 것이며 이는 Loss를 최저점을 지나치고 다시 돌아오고를 반복하게 만들 수 있습니다. 이는 Random Batch이기 때문에 그 불안정성이 더 짙어보일 수 있습니다. 그러나 RMSProp에서는 학습률이 높다고 생각되는 Weight에 대해서는 그 학습률을 낮춰서 Loss가 최저점이 되는 점을 지나치지않도록 조정해줍니다. 그렇기 때문에 Random Batch에서도 안정적인 학습을 할 수 있으며, 이에 따라 7개의 독립변수 Weight들이 모두 충분하게 학습되기 때문에 Loss가 낮고, Evaluation에 대해서 Accuracy도 또한 높게 나올 수 있습니다.

다음 페이지부터 나오는 10개의 그래프는 100개의 데이터들의 분포도를 나타낸 것 입니다. 그래프는 Epoch별로 10개 존재하며, 가로축은 1~100까지의 데이터갯수를 의미하며, 세로축은 Accuracy를 의미합니다. 즉, 각 Epoch에서 Optimizer에 따른 Accuracy분포도를 나타낸 것으로 해석할 수 있습니다. 먼저 RMSProp이 전체적으로 모든 Data들이 안정적으로 높은 Accuracy를 보인다는 점을 알 수 있습니다. 특이한 점은 RMSProp이 나머지 두 Optimizer에 비해 모든 데이터에서 항상 Accuracy가 높은 것은 아닌데, 이는 Epoch가 작기 때문에 RMSProp가 가중치를 조절하다보니 SGD와 Momentum에 비해 충분한 학습이 되지 못하기 떄문이며, SGD와 Momentum에서도 초반부터 학습이 잘 이뤄지면 충분히 RMSProp처럼 학습을 할 수 있기 때문입니다. 만약 Epoch가 높아서 충분학 학습이 보장된다면, SGD와 Momentum이 RMSProp에 비해 높은 Accuracy를 가지는 빈도는 낮아질 것입니다.(Epoch 10000~40000사이에서 500번 돌려본 결과 거의 모든 Data에서 RMSProp이 높거나 같은 Accuracy를 보여줬습니다.) 그리고 다른 특이한 점은 SGD, Momentum의 경우 Accuracy가 상당히 낮게 나오는 경우도 존재했습니다. 이또한 작은 Epoch때문에 제대로 학습이 이뤄지지 않기 때문에 발생하는 현상이라고 생각하며, SGD보다 Momentum의 경우게 그 현상이 빈번하였습니다. 이는 학습 관성률이 Momentum이 SGD에 비해 높기 때문에(0.9의 관성율) Loss가 최저점이 되는 점을 지나치고 학습이 종료되어 제대로 된 학습이 이뤄지지 않기 때문이라 생각됩니다.(현재는 Momentum의 Loss가 SGD보다 높은 것이 뚜렷하게 보이지만 Epoch 10000~40000사이에서 500번정도 돌려본 결과 그 차이가 줄어들어 거의 비슷한 Loss를 보여주고 있습니다.)

