

```
1 #%% md
2 # Atividade 02: Implementando uma Rede Neural
3
4 Nesta atividade, você irá treinar uma rede neural com
   camadas completamente conectadas para realizar
   classificação de imagens, e irá testá-la utilizando o
   dataset CIFAR-10.
5
6 Nesta atividade, você irá:
7
8 - testar uma **função de perda** (**loss function**)
   para uma rede neural de duas camadas
9 - testar a avaliação de um **gradiente analítico**
10 - **verificar a implementação** utilizando gradiente
    numérico
11 - **treinar** uma rede em um pequeno problema por
    meio de **SGD**
12 - **treinar e depurar** uma rede em um conjunto de **dados reais**
13 - usar um conjunto de validação para **ajustar
    hiperparâmetros**
14 - **visualizar** os pesos finais que foram obtidos
15
16 #%% md
17 ## Preparação para usar o *Colaboratory*
18
19 Caso você deseje usar o *Colaboratory*, execute os
   comandos da célula a seguir para conseguir acessar os
   arquivos em uma conta do **Google Drive**.
20
21 #%%
22 #from google.colab import drive
23 #drive.mount('/content/drive/')
24 #%% md
25 Além disso, você talvez deseje trabalhar um
   subdiretório (ou pasta) específico, por exemplo `Pratica2`. Esse diretório (ou pasta) já deve ter
   sido criado e conter o ***notebook*** dessa prática.
26
27 Sendo assim, utilize as instruções na próxima célula
   para alterar o diretório corrente para esse
```

```
27 subdiretório (alterando o nome do subdiretório, se
   for o caso).
28 #%%
29 #import os
30 #os.chdir("drive/My Drive/Pratica2")
31 #%% md
32 ## Obtendo o conjunto de dados CIFAR-10
33
34 Nesta atividade, você irá desenvolver uma rede neural
   para realizar classificação de imagens e irá testá-
   la utilizando o dataset CIFAR-10.
35
36 Para tanto, você deve obter as imagens dessa base de
   dados executando a célula a seguir.
37
38 **OBS: Você só precisar realizar esta etapa uma vez
   !** Caso você já a tenha feito anteriormente e está
   retornando para continuar a execução dessa prática,
   não será necessário que se faça a recuperação das
   imagens novamente. Para verificar se os dados da base
   estão disponíveis, basta checar a existência do
   subdiretório `cifar-10-batches-py` no diretório
   corrente.
39 #%%
40 #!wget http://www.cs.toronto.edu/~kriz/cifar-10-
   python.tar.gz
41 #!tar -xzvf cifar-10-python.tar.gz
42 #!rm cifar-10-python.tar.gz
43 #%% md
44 ## Código de inicialização e configuração básica
45
46 Execute a célula a seguir para garantir a importação
   de alguns recursos básicos, bem como a inicialização/
   configuração para exibição correta de gráficos.
47 #%%
48 # Algum código de inicialização
49
50 import numpy as np
51 import matplotlib.pyplot as plt
52 from past.builtins import xrange
53
```

```
54 from __future__ import print_function
55
56 %matplotlib inline
57 plt.rcParams['figure.figsize'] = (10.0, 8.0)
58 plt.rcParams['image.interpolation'] = 'nearest'
59 plt.rcParams['image.cmap'] = 'gray'
60
61 %load_ext autoreload
62 %autoreload 2
63
64 def rel_error(x, y):
65     """ retorna erro relativo """
66     return np.max(np.abs(x - y) / (np.maximum(1e-8,
67         np.abs(x) + np.abs(y))))
67 #%%
68 ## Código para Classe implementando Rede Neural com
69 ## 02 Camadas
70
71 Nesta atividade será utilizada a classe `TwoLayerNet`
72 ` para representar instâncias de uma rede neural.
73 Esta classe se encontra definida dentro da célula
74 abaixo .
75
76 Os parâmetros da rede serão armazenados na variável
77 de instância `self.params` que é um dicionário em que
78 as chaves são os nomes (*strings*) de cada parâmetro
79 e os valores são **arrays numpy**.
80
81
82 #%%
83
84 class TwoLayerNet(object):
85     """
86     A two-layer fully-connected neural network. The net
87     has an input dimension of
88      $N$ , a hidden layer dimension of  $H$ , and performs
89     classification over  $C$  classes.
90     We train the network with a softmax loss function
91     and L2 regularization on the
92     weight matrices. The network uses a ReLU
93     nonlinearity after the first fully
94     connected layer.
```

```

83
84  In other words, the network has the following
     architecture:
85
86  input - fully connected layer - ReLU - fully
     connected layer - softmax
87
88  The outputs of the second fully-connected layer
     are the scores for each class.
89  """
90
91  def __init__(self, input_size, hidden_size,
92               output_size, std=1e-4):
93      """
94          Initialize the model. Weights are initialized to
         small random values and
94          biases are initialized to zero. Weights and
         biases are stored in the
95          variable self.params, which is a dictionary with
         the following keys:
96
97          W1: First layer weights; has shape (D, H)
98          b1: First layer biases; has shape (H,)
99          W2: Second layer weights; has shape (H, C)
100         b2: Second layer biases; has shape (C,)
101
102        Inputs:
103          - input_size: The dimension D of the input data.
104          - hidden_size: The number of neurons H in the
             hidden layer.
105          - output_size: The number of classes C.
106          """
107        self.params = {}
108        self.params['W1'] = std * np.random.randn(
109            input_size, hidden_size)
109        self.params['b1'] = np.zeros(hidden_size)
110        self.params['W2'] = std * np.random.randn(
111            hidden_size, output_size)
111        self.params['b2'] = np.zeros(output_size)
112
113    def loss(self, X, y=None, reg=0.0):

```

```

114     """
115     Compute the loss and gradients for a two layer
116     fully connected neural
117     network.
118
119     Inputs:
120         - X: Input data of shape (N, D). Each X[i] is a
121             training sample.
122         - y: Vector of training labels. y[i] is the
123             label for X[i], and each y[i] is
124             an integer in the range 0 <= y[i] < C. This
125             parameter is optional; if it
126             is not passed then we only return scores, and
127             if it is passed then we
128             instead return the loss and gradients.
129         - reg: Regularization strength.
130
131     Returns:
132         If y is None, return a matrix scores of shape (N
133         , C) where scores[i, c] is
134         the score for class c on input X[i].
135
136         If y is not None, instead return a tuple of:
137             - loss: Loss (data loss and regularization loss
138                 ) for this batch of training
139                 samples.
140             - grads: Dictionary mapping parameter names to
141                 gradients of those parameters
142                 with respect to the loss function; has the
143                 same keys as self.params.
144
145     # Unpack variables from the params dictionary
146     W1, b1 = self.params['W1'], self.params['b1']
147     W2, b2 = self.params['W2'], self.params['b2']
148     N, D = X.shape
149
150     # Compute the forward pass
151
152     layer1 = X.dot(W1) + b1          # Forward 1st layer
153     layer1[layer1<0] = 0            # ReLU
154     layer2 = layer1.dot(W2) + b2    # Forward 2nd layer

```



```

179     db2 = dLi.sum(axis=1)
180     dW2 = dLi.dot(layer1).T + reg*W2
181
182     dLayer1 = dLi.T.dot(W2.T)
183     dLayer1[layer1<=0] = 0
184
185     db1 = dLayer1.sum(axis=0)
186     dW1 = dLayer1.T.dot(X).T + reg*W1
187
188     assert db2.shape==b2.shape
189     assert dW2.shape==W2.shape
190     assert db1.shape==b1.shape
191     assert dW1.shape==W1.shape
192
193     grads = {
194         'b2': db2,
195         'W2': dW2,
196         'b1': db1,
197         'W1': dW1
198     }
199
200     return loss, grads
201
202     def train(self, X, y, X_val, y_val,
203                 learning_rate=1e-3, learning_rate_decay=
204                 0.95,
205                 reg=5e-6, num_iters=100,
206                 batch_size=200, verbose=False):
207         """
208             Train this neural network using stochastic
209             gradient descent.
210
211             Inputs:
212                 - X: A numpy array of shape (N, D) giving
213                     training data.
214                 - y: A numpy array f shape (N,) giving training
215                     labels; y[i] = c means that
216                         X[i] has label c, where 0 <= c < C.
217                 - X_val: A numpy array of shape (N_val, D)
218                     giving validation data.
219                 - y_val: A numpy array of shape (N_val,) giving

```

```
214 validation labels.  
215     - learning_rate: Scalar giving learning rate for  
optimization.  
216     - learning_rate_decay: Scalar giving factor used  
to decay the learning rate  
217         after each epoch.  
218     - reg: Scalar giving regularization strength.  
219     - num_iters: Number of steps to take when  
optimizing.  
220     - batch_size: Number of training examples to use  
per step.  
221     - verbose: boolean; if true print progress  
during optimization.  
222     """  
223     num_train = X.shape[0]  
224     iterations_per_epoch = max(num_train /  
batch_size, 1)  
225  
226     # Use SGD to optimize the parameters in self.  
model  
227     loss_history = []  
228     train_acc_history = []  
229     val_acc_history = []  
230  
231     idxItems = range(num_train)  
232  
233     for it in xrange(num_iters):  
234         X_batch = None  
235         y_batch = None  
236  
237         selectIdx = np.random.choice(idxItems, size=  
batch_size, replace=True)  
238         X_batch = X[selectIdx,:]  
239         y_batch = y[selectIdx]  
240  
241         # Compute loss and gradients using the current  
minibatch  
242         loss, grads = self.loss(X_batch, y=y_batch,  
reg=reg)  
243         loss_history.append(loss)  
244
```

```

245     for paramName in self.params:
246         self.params[paramName] -= learning_rate *
247             grads[paramName]
248
249         if verbose and it % 100 == 0:
250             print('iteration %d / %d: loss %f' % (it,
251                 num_iters, loss))
252
253             # Every epoch, check train and val accuracy
254             # and decay learning rate.
255             if it % iterations_per_epoch == 0:
256                 # Check accuracy
257                 train_acc = (self.predict(X_batch) ==
258                     y_batch).mean()
259                 val_acc = (self.predict(X_val) == y_val).
260                     mean()
261
262                 train_acc_history.append(train_acc)
263                 val_acc_history.append(val_acc)
264
265             # Decay learning rate
266             learning_rate *= learning_rate_decay
267
268             return {
269                 'loss_history': loss_history,
270                 'train_acc_history': train_acc_history,
271                 'val_acc_history': val_acc_history,
272             }
273
274     def predict(self, X):
275
276         """
277             Use the trained weights of this two-layer
278             network to predict labels for
279             data points. For each data point we predict
280             scores for each of the C
281             classes, and assign each data point to the class
282             with the highest score.
283
284             Inputs:
285             - X: A numpy array of shape (N, D) giving N D-
286                 dimensional data points to
287                 classify.

```

```
277
278     Returns:
279     - y_pred: A numpy array of shape (N,) giving
280     predicted labels for each of
281     the elements of X. For all i, y_pred[i] = c
282     means that X[i] is predicted
283     to have class c, where 0 <= c < C.
284     """
285     y_pred = self.loss(X, y=None, reg=0.0).argmax(
286         axis=1)
287
288     return y_pred
289
290 ## Criação de Pequeno Exemplo (*Toy Model*) para
291 # Testes Preliminares
292
293 A seguir, você irá inicializar um pequeno conjunto
294 de dados e um modelo simples que será usado para
295 iniciar os teste dessa implementação.
296
297 #%% md
298 # Cria um pequeno conjunto de dados aleatórios e um
299 # modelo simples para verificar sua implementação.
300 # Veja que foi fixado a 'semente aleatória' para
301 # possibilhar a repetição de experimentos
302
303 def init_toy_model():
304     np.random.seed(0)
305     return TwoLayerNet(input_size, hidden_size,
306                         num_classes, std=1e-1)
307
308 def init_toy_data():
309     np.random.seed(1)
310     X = 10 * np.random.randn(num_inputs, input_size)
311     y = np.array([0, 1, 2, 2, 1])
312     return X, y
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
778
779
779
780
781
782
783
784
785
786
787
787
788
789
789
790
791
792
793
794
795
796
797
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
888
889
889
890
891
892
893
894
895
896
897
898
899
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
918
919
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
948
949
949
950
951
952
953
954
955
956
957
958
959
959
960
961
962
963
964
965
966
967
968
969
969
970
971
972
973
974
975
976
977
978
979
979
980
981
982
983
984
985
986
987
988
989
989
990
991
992
993
994
995
996
997
998
999
999
1000
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1087
1088
1089
1090
1091
1092
1093
1094
1095
1095
1096
1097
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1185
1186
1187
1188
1189
1189
1190
1191
1192
1193
1194
1195
1195
1196
1197
1198
1199
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1285
1286
1287
1288
1289
1289
1290
1291
1292
1293
1294
1295
1295
1296
1297
1298
1299
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1385
1386
1387
1388
1389
1389
1390
1391
1392
1393
1394
1395
1395
1396
1397
1398
1399
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1485
1486
1487
1488
1489
1489
1490
1491
1492
1493
1494
1495
1495
1496
1497
1498
1499
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1585
1586
1587
1588
1589
1589
1590
1591
1592
1593
1594
1595
1595
1596
1597
1598
1599
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1685
1686
1687
1688
1689
1689
1690
1691
1692
1693
1694
1695
1695
1696
1697
1698
1699
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1785
1786
1787
1788
1789
1789
1790
1791
1792
1793
1794
1795
1795
1796
1797
1798
1799
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1885
1886
1887
1888
1889
1889
1890
1891
1892
1893
1894
1895
1895
1896
1897
1898
1899
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1985
1986
1987
1988
1989
1989
1990
1991
1992
1993
1994
1995
1995
1996
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2085
2086
2087
2088
2089
2089
2090
2091
2092
2093
2094
2095
2095
2096
2097
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2185
2186
2187
2188
2189
2189
2190
2191
2192
2193
2194
2194
2195
2196
2197
2198
2199
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2276
2277
2278
2279
2280
2281
2282
2283
2284
2284
2285
2286
2287
2288
2289
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2309
2310
2311
2312
2313
2314
2315
2316
2317
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2376
2377
2378
2379
2380
2381
2382
2383
2384
2384
2385
238
```

```
309 net = init_toy_model()
310 X, y = init_toy_data()
311 #%% md
312 # Passo de Propagação (*Forward pass*): cálculo de *
313 # scores*
314 Retorne ao código da classe `TwoLayerNet` e analise
315 o método `TwoLayerNet.loss`.
316
317 Esta função é muito similar as funções de perda que
318 foram discutidas em sala de aula: ela utiliza os
319 dados e os pesos (ou parâmetros) para calcular os *
320 # scores* de cada classe, o valor de perda/custo e os
321 gradientes em relação aos parâmetros.
322
323 Você deve examinar com cuidado a primeira parte do
324 passo de propagação (*forward pass*) que utiliza os
325 pesos e vieses (*biases*) para calcular os *scores*
326 para todas as entradas.
327
328 #%% 
329 scores = net.loss(X)
330 print('Your scores:')
331 print(scores)
332 print()
333 print('correct scores:')
334 correct_scores = np.asarray([
335     [-0.81233741, -1.27654624, -0.70335995],
336     [-0.17129677, -1.18803311, -0.47310444],
337     [-0.51590475, -1.01354314, -0.8504215 ],
338     [-0.15419291, -0.48629638, -0.52901952],
339     [-0.00618733, -0.12435261, -0.15226949]])
340 print(correct_scores)
341 print()
342
343 # A diferença deve ser bem pequena, algo < 1e-7
344 print('Difference between your scores and correct
345 scores:')
346 print(np.sum(np.abs(scores - correct_scores)))
347 #%% md
348 # Passo de Propagação (*Forward pass*): cálculo da
349 # perda/custo
350 Na mesma função, você deve examinar (o código e
```

```

338 testar) a segunda parte do passo de propagação (*
    forward pass*) responsável pelo cálculo da perda
    envolvendo os dados e a regularização.
339 #%%
340 loss, _ = net.loss(X, y, reg=0.1)
341 correct_loss = 1.30378789133
342
343 # Novamente, a diferença deve ser pequena, algo < 1e
-12
344 print('Difference between your loss and correct loss
    :')
345 print(np.sum(np.abs(loss - correct_loss)))
346 #%% md
347 # Passo de Retropropagação (*Backward pass*)
348 Agora, você deve examinar o restante da função `TwoLayerNet.loss`, de modo que a compreender como a função calcula o gradiente da perda em relação aos parâmetros `W1`, `b1`, `W2` e `b2`.
349
350 Agora, uma vez que você pode examinar em detalhes (e depurar) o passo de retropropagação: por meio do uso de estimativas numéricas do gradiente.
351 #%%
352 from random import randrange
353
354 def eval_numerical_gradient(f, x, verbose=True, h=0.
    00001):
355     """
356     a naive implementation of numerical gradient of f
    at x
357     - f should be a function that takes a single
    argument
358     - x is the point (numpy array) to evaluate the
    gradient at
359     """
360
361     fx = f(x) # evaluate function value at original
    point
362     grad = np.zeros_like(x)
363     # iterate over all indexes in x
364     it = np.nditer(x, flags=['multi_index'], op_flags

```

```

364 =['readwrite'])
365     while not it.finished:
366
367         # evaluate function at x+h
368         ix = it.multi_index
369         oldval = x[ix]
370         x[ix] = oldval + h # increment by h
371         fxph = f(x) # evalute f(x + h)
372         x[ix] = oldval - h
373         fxmh = f(x) # evaluate f(x - h)
374         x[ix] = oldval # restore
375
376         # compute the partial derivative with centered
377         # formula
377         grad[ix] = (fxph - fxmh) / (2 * h) # the slope
378         if verbose:
379             print(ix, grad[ix])
380         it.iternext() # step to next dimension
381
382     return grad
383
384 # Usa verificação numérica do gradiente para checar
384 # sua implementação do passo de retropropagação.
385 # Se sua implementação estiver correta, a diferença
385 # entre a estimativa numérica do gradiente
386 # e o valor obtido analiticamente deve ser menor que
386 # 1e-8 para cada um dos parâmetros.
387
388 loss, grads = net.loss(X, y, reg=0.05)
389
390 # Os erros deve ser menores que 1e-8
391 for param_name in grads:
392     f = lambda W: net.loss(X, y, reg=0.05)[0]
393     param_grad_num = eval_numerical_gradient(f, net.
393     params[param_name], verbose=False)
394     print('%s max relative error: %e' % (param_name,
394     , rel_error(param_grad_num, grads[param_name])))
395 %% md
396 # Treinamento de uma Rede Simples
397 Para treinar uma rede, você deve usar o método SGD (
397 método de descida mais íngreme estocástico).

```

```
398
399 Analise a função `TwoLayerNet.train` e examinando
   todas as partes que importantes para se implementar
   o procedimento de treinamento.
400
401 Você também deve realizar uma análise da função `TwoLayerNet.predict` pois ela será necessária
   durante o treinamento, uma vez que periodicamente o
   método realiza predições para acompanhar a acurácia
   ao longo do processo.
402
403 Uma vez que você tenha estudado tais métodos,
   execute o código abaixo para treinar um rede de duas
   camadas sobre o pequeno conjunto de dados aleatórios .
   Você deverá obter uma perda ao final do
   treinamento inferior a 0.2
404 #%%
405 net = init_toy_model()
406 stats = net.train(X, y, X, y,
407                     learning_rate=1e-1, reg=5e-6,
408                     num_iters=100, verbose=False)
409
410 print('Final training loss: ', stats['loss_history']
411       ][-1])
411
412 # Plota o histórico da função de perda
413 plt.plot(stats['loss_history'])
414 plt.xlabel('iteration')
415 plt.ylabel('training loss')
416 plt.title('Training Loss history')
417 plt.show()
418 #%% md
419 # Carregamento de dados
420 Agora que você testou uma rede de duas camadas
   passando pela verificação de gradientes e seu
   funcionamento sobre o pequeno conjunto de dados, é hora
   de carregar os dados do **CIFAR-10 dataset** de
   modo que você possa usá-los no treinamento de um
   classificador sobre dados reais.
421 #%%
422 from __future__ import print_function
```

```
423
424 from six.moves import cPickle as pickle
425 import numpy as np
426 import os
427 import platform
428
429 def load_pickle(f):
430     version = platform.python_version_tuple()
431     if version[0] == '2':
432         return pickle.load(f)
433     elif version[0] == '3':
434         return pickle.load(f, encoding='latin1')
435     raise ValueError("invalid python version: {}".format(version))
436
437 def load_CIFAR_batch(filename):
438     """ load single batch of cifar """
439     with open(filename, 'rb') as f:
440         datadict = load_pickle(f)
441         X = datadict['data']
442         Y = datadict['labels']
443         X = X.reshape(10000, 3, 32, 32).transpose(0,2,3,
444             1).astype("float")
445         Y = np.array(Y)
446         return X, Y
447
448 def load_CIFAR10(ROOT):
449     """ load all of cifar """
450     xs = []
451     ys = []
452     for b in range(1,6):
453         f = os.path.join(ROOT, 'data_batch_%d' % (b, ))
454         X, Y = load_CIFAR_batch(f)
455         xs.append(X)
456         ys.append(Y)
457     Xtr = np.concatenate(xs)
458     Ytr = np.concatenate(ys)
459     del X, Y
460     Xte, Yte = load_CIFAR_batch(os.path.join(ROOT,
461                                         'test_batch'))
462     return Xtr, Ytr, Xte, Yte
```

```
461
462 def get_CIFAR10_data(num_training=49000,
463     num_validation=1000, num_test=1000):
464     """
465         Carrega CIFAR-10 dataset a partit do disco e
466         realiza preprocessamento para preparar
467         os dados para a rede neural de duas camadas.
468         Estes são os mesmos passos usados para o modelo
469         SVM, porém condensado em uma única função.
470     """
471
472     # Carregga os dados CIFAR-10 brutos
473     cifar10_dir = 'cifar-10-batches-py'
474     X_train, y_train, X_test, y_test = load_CIFAR10(
475         cifar10_dir)
476
477     # Subdivide os dados em conjuntos
478     mask = list(range(num_training, num_training +
479                     num_validation))
480     X_val = X_train[mask]
481     y_val = y_train[mask]
482     mask = list(range(num_training))
483     X_train = X_train[mask]
484     y_train = y_train[mask]
485     mask = list(range(num_test))
486     X_test = X_test[mask]
487     y_test = y_test[mask]
488
489     return X_train, y_train, X_val, y_val, X_test,
490           y_test
491
492
493 # Usar a função definida acmina para obter os dados.
494 X_train, y_train, X_val, y_val, X_test, y_test =
495     get_CIFAR10_data()
496 print('Train data shape: ', X_train.shape)
497 print('Train labels shape: ', y_train.shape)
498 print('Validation data shape: ', X_val.shape)
499 print('Validation labels shape: ', y_val.shape)
500 print('Test data shape: ', X_test.shape)
501 print('Test labels shape: ', y_test.shape)
502 %% md
```

```

495 ## Visualizando amostras dos dados
496 Utilize a célula a seguir para visualizar algumas
amostras das 10 classes de imagens da base `CIFAR-10
` 

497 #%%
498 # Visualizar alguns exemplos do dataset.
499 # São exibidos apenas 7 exemplos de imagens de
treinamento de cada classe.
500 classes = ['plane', 'car', 'bird', 'cat', 'deer', '
dog', 'frog', 'horse', 'ship', 'truck']
501 num_classes = len(classes)
502 samples_per_class = 7
503 for y, cls in enumerate(classes):
504     idxs = np.flatnonzero(y_train == y)
505     idxs = np.random.choice(idxs, samples_per_class
, replace=False)
506     for i, idx in enumerate(idxs):
507         plt_idx = i * num_classes + y + 1
508         plt.subplot(samples_per_class, num_classes,
plt_idx)
509         plt.imshow(X_train[idx].astype('uint8'))
510         plt.axis('off')
511         if i == 0:
512             plt.title(cls)
513 plt.show()
514 #%% md
515 ## Normalização de dados
516 Conforme discutido em sala de aula é importante para
facilitar o treinamento que seja feito algum tipo
de tratamento dos dados. Neste caso, uma simples
subtração de uma imagem média está sendo utilizada
na célula a seguir, de modo a centralizar os dados
em torno da origem do sistemas de coordenadas.
517 #%%
518 # Normaliza os dados: subtrai a imagem média
519 mean_image = np.mean(X_train, axis=0)
520 X_train -= mean_image
521 X_val -= mean_image
522 X_test -= mean_image
523
524 # Redimensiona as imagens de matrizes para vetores

```

```
525 X_train = X_train.reshape(49000, -1)
526 X_val = X_val.reshape(1000, -1)
527 X_test = X_test.reshape(1000, -1)
528
529 print('Train data shape: ', X_train.shape)
530 print('Train labels shape: ', y_train.shape)
531 print('Validation data shape: ', X_val.shape)
532 print('Validation labels shape: ', y_val.shape)
533 print('Test data shape: ', X_test.shape)
534 print('Test labels shape: ', y_test.shape)
535 %% md
536 # Treinamento de uma Rede com Dados Reais
537 Para treinar sua rede, você deve usar **SGD**. Além disso, nesse processo a taxa de aprendizado será ajustada com um decaimento exponencial ao longo do processo de otimização, isto é, após cada época, a taxa de aprendizado é multiplicada pela taxa de decaimento (como esta última é menor que um, consequentemente a taxa de aprendizado é reduzida).
538 %%
539 input_size = 32 * 32 * 3
540 hidden_size = 50
541 num_classes = 10
542 net = TwoLayerNet(input_size, hidden_size,
      num_classes)
543
544 # Treinamento da rede
545 stats = net.train(X_train, y_train, X_val, y_val,
      num_iters=1000, batch_size=200,
      learning_rate=1e-4, learning_rate_decay=
      0.95,
      reg=0.25, verbose=True)
546
547 # Predição sobre o conjunto de validação
548 val_acc = (net.predict(X_val) == y_val).mean()
549 print('Validation accuracy: ', val_acc)
550 %% md
551 # Depuração do treinamento
552 Com os valores de parâmetros fornecidos acima, você deve ter obtido uma acurácia no conjunto de validação em torno de 0.29. O que não representa um
```

```

555 resultado muito bom...
556
557 Uma estratégia para melhorar o entendimento ( fornecer *insigths*) sobre o que pode estar errado é traçar os gráficos de evolução da função de perda e das acurácia de treinamento e validação ao longo do processo de otimização.
558 #%%
559 # Plota a função de pedar e as acurácia de treinamento e validação
560 plt.subplot(2, 1, 1)
561 plt.plot(stats['loss_history'])
562 plt.title('Loss history')
563 plt.xlabel('Iteration')
564 plt.ylabel('Loss')
565
566 plt.subplot(2, 1, 2)
567 plt.plot(stats['train_acc_history'], label='train')
568 plt.plot(stats['val_acc_history'], label='val')
569 plt.title('Classification accuracy history')
570 plt.xlabel('Epoch')
571 plt.ylabel('Classification accuracy')
572 plt.tight_layout()
573 plt.show()
574 #%% md
575 Uma outra estratégia é construir uma visualização dos pesos que foram obtidos na primeira camada da rede. Por trás disto, está o fato de que na maioria das redes neurais treinadas sobre dados visuais, os pesos da primeira camada geralmente exibem algum tipo de estrutura visível.
576 #%%
577 from past.builtins import xrange
578
579 from math import sqrt, ceil
580 import numpy as np
581
582 def visualize_grid(Xs, ubound=255.0, padding=1):
583     """
584     Reshape a 4D tensor of image data to a grid for easy visualization.

```

```

585
586     Inputs:
587     - Xs: Data of shape (N, H, W, C)
588     - ubound: Output grid will have values scaled to
      the range [0, ubound]
589     - padding: The number of blank pixels between
      elements of the grid
590     """
591     (N, H, W, C) = Xs.shape
592     grid_size = int(ceil(sqrt(N)))
593     grid_height = H * grid_size + padding * (grid_size
      - 1)
594     grid_width = W * grid_size + padding * (grid_size
      - 1)
595     grid = np.zeros((grid_height, grid_width, C))
596     next_idx = 0
597     y0, y1 = 0, H
598     for y in xrange(grid_size):
599         x0, x1 = 0, W
600         for x in xrange(grid_size):
601             if next_idx < N:
602                 img = Xs[next_idx]
603                 low, high = np.min(img), np.max(img)
604                 grid[y0:y1, x0:x1] = ubound * (img - low
      ) / (high - low)
605                 # grid[y0:y1, x0:x1] = Xs[next_idx]
606                 next_idx += 1
607                 x0 += W + padding
608                 x1 += W + padding
609                 y0 += H + padding
610                 y1 += H + padding
611                 # grid_max = np.max(grid)
612                 # grid_min = np.min(grid)
613                 # grid = ubound * (grid - grid_min) / (grid_max -
      grid_min)
614             return grid
615
616 def vis_grid(Xs):
617     """ visualize a grid of images """
618     (N, H, W, C) = Xs.shape
619     A = int(ceil(sqrt(N)))

```

```

620     G = np.ones((A*H+A, A*W+A, C), Xs.dtype)
621     G *= np.min(Xs)
622     n = 0
623     for y in range(A):
624         for x in range(A):
625             if n < N:
626                 G[y*H+y:(y+1)*H+y, x*W+x:(x+1)*W+x, :] = Xs[
627                     n,:,:,:, :]
628             n += 1
629     # normalize to [0,1]
630     maxg = G.max()
631     ming = G.min()
632     G = (G - ming)/(maxg-ming)
633     return G
634
635 def vis_nn(rows):
636     """ visualize array of arrays of images """
637     N = len(rows)
638     D = len(rows[0])
639     H,W,C = rows[0][0].shape
640     Xs = rows[0][0]
641     G = np.ones((N*H+N, D*W+D, C), Xs.dtype)
642     for y in range(N):
643         for x in range(D):
644             G[y*H+y:(y+1)*H+y, x*W+x:(x+1)*W+x, :] = rows[
645                 y][x]
646             # normalize to [0,1]
647             maxg = G.max()
648             ming = G.min()
649             G = (G - ming)/(maxg-ming)
650     # Visualiza os pesos da primeira camada da rede
651
652 def show_net_weights(net):
653     W1 = net.params['W1']
654     W1 = W1.reshape(32, 32, 3, -1).transpose(3, 0, 1
655     , 2)
656     plt.imshow(visualize_grid(W1, padding=3).astype(
657         'uint8'))
658     plt.gca().axis('off')

```

```
657     plt.show()
658
659 show_net_weights(net)
660 %% md
661 # Ajuste de hiperparâmetros
662
663 **0 que está errado?** Observando os gráficos e as visualizações acima, pode-se ver que a perda está reduzindo de forma *mais ou menos* linear, o que parece sugerir que a taxa de aprendizado pode estar muito baixa. Além disso, não há uma separação grande entre as acurárias de treinamento e validação, sugerindo que o modelo usado tem baixa capacidade e que talvez seu tamanho devesse ser aumentado. Por outro lado, com um modelo muito grande deve-se esperar observar mais *overfitting*, que se manifesta por meio de uma distância muita grande entre acurárias de treinamento e de validação.
664
665 **Ajuste**. Realizar o ajuste de hiperparâmetros e desenvolver uma noção intuitiva de como eles afetam o resultado final é uma parte importatnte do uso de Redes Neurais. Dessa forma, deseja-se que você realize várias práticas envolvendo o ajuste de hiperparâmetros. A seguir, você deve realizar experimentos com diferentes valores para os hiperparâmetros incluindo: tamanho da camada escondida, taxa de aprendizado, taxa de decaimento, número de épocas de treinamento e regularização.
666
667 **Resultados aproximados**. Você deve tentar alcançar uma acurácia de classificação (taxa de acerto) maior que 48% no conjunto de validação. (OBS : minha solução obteve uma acurácia acima de 52% no conjunto de validação!)
668
669 **Experimento**: O objetivo desse exercício é que você tente obter o melhor resultado sobre a base CIFAR-10, usando uma rede neural completamente conectada. Sinta-se livre para implementar quaisquer técnicas que desejar (p.ex., redução de
```

```
669 dimensionalidade via PCA, *dropout*, ou outras
estratégias ao otimizador, etc).
670
671 **OBS: Lembre-se de deixar documentado tudo que foi
feito! Caso necessário, acrescente mais células a
sua vontade.**
672 #%%
673 best_net = None # Armazenar o melhor modelo
encontrado nesse variável pois será
674 # usado no teste final (ver final
desse notebook)
675
676 #####
677 # TODO: Ajustar hiperparâmetros usando o conjunto de
validação. O melhor modelo #
678 # obtido ao longo do treinamento deve-se armazenado
em best_net. #
679 #
#
680 # Para auxiliar a depurar sua rede, pode ser
interessante se utilizar de ##
681 # visualizações similares as usadas acima. Essas
visualizações irão apresentar #
682 # diferenças qualitativas significativas
especialmente para redes com ajustes #
683 # ruins
.
#
684 #
#
685 # O ajuste fino de hiperparâmetros feito manualmente
pode ser divertido, mas #
686 # provavelmente você deverá considerar a
possibilidade de escrever código que #
687 # percorra todas as combinações possíveis de
hiperparâmetros de forma a tornar #
688 # automático o processo de busca (similar ao que foi
feito nas atividades #
689 # anteriores
).
```

```
689 #
690 #####
691 #%% md
692 Utilizei o Random Search para encontrar algumas
tendencias de parametros. Após isso, fazendo ajustes
manualmente selecionei alguns hiperparametros que
não tinham problemas de estouro de gradiente e que
pareciam convergir para uma resposta melhor e fui
reduzindo o número de opções de parametros até
chegar em uma quantidade suficientemente baixa para
aplicar o grid search.
693
694 com certeza existem buscas mais inteligentes que
podem ser feitas, mas para o propósito desse
exercício, acredito que o resultado foi satisfatório
.
695 #%%
696
697 # treinament original
698 # stats = net.train(X_train, y_train, X_val, y_val,
699 #                      num_iters=1000, batch_size=200,
700 #                      learning_rate=1e-4,
701 #                      learning_rate_decay=0.95,
702 #                      reg=0.25, verbose=True)
703 #
704 #
705 #
706 # def random_search_hyperparams(X_train, y_train,
707 #                                 X_val, y_val, num_experiments):
708 #     # Definindo os limites para os hiperparâmetros
709 #     hidden_size_options = [100, 200, 300]
710 #     learning_rate_options = [1e-3]
711 #     learning_rate_decay_options = [0.90]
712 #     reg_options = [ 0.05, 0.1]
713 #     num_iters_options = [3000]
714 #     batch_size_options = [200, 250]
715 #     input_size = 32 * 32 * 3
716 #     num_classes = 10
```

```

717 #
718 #     best_val_acc = 0
719 #     best_params = {}
720 #
721 #     for _ in range(num_experiments):
722 #         # Escolha aleatória dos hiperparâmetros
723 #         hidden_size = np.random.choice(
724 #             hidden_size_options)
725 #         learning_rate = np.random.choice(
726 #             learning_rate_options)
727 #         learning_rate_decay = np.random.choice(
728 #             learning_rate_decay_options)
729 #         reg = np.random.choice(reg_options)
730 #         num_iters = np.random.choice(
731 #             num_iters_options)
732 #         batch_size = np.random.choice(
733 #             batch_size_options)
734 #
735 #         # Cria uma nova instância da rede neural
736 #         # com os hiperparâmetros escolhidos
737 #         net = TwoLayerNet(input_size , hidden_size ,
738 #             num_classes)
739 #
740 #         # Treina a rede neural
741 #         stats = net.train(X_train, y_train, X_val
742 #             , y_val,
743 #                 num_iters=num_iters,
744 #                 batch_size=batch_size,
745 #                 learning_rate=
746 #                     learning_rate, learning_rate_decay=
747 #                         learning_rate_decay,
748 #                         reg=reg, verbose=False)
749 #
750 #         # Avalia o desempenho da rede na validação
751 #         val_acc = stats['val_acc_history'][-1]
752 #         print(f"Experiment with lr {learning_rate},

```

```

742 hidden_size {hidden_size}: Validation accuracy {
    val_acc")
743 #
744 #           # Atualiza os melhores parâmetros, se
    necessário
745 #           if val_acc > best_val_acc:
746 #               best_net = net
747 #               best_val_acc = val_acc
748 #               best_params = {
749 #                   'hidden_size': hidden_size,
750 #                   'learning_rate': learning_rate,
751 #                   'learning_rate_decay':
    learning_rate_decay,
752 #                       'reg': reg,
753 #                       'num_iters': num_iters,
754 #                       'batch_size': batch_size
755 #               }
756 #
757 #           print("Best validation accuracy achieved
    during random search: ", best_val_acc)
758 #       return best_net, best_params
759 #
760 # # Exemplo de uso
761 # best_net, best_params = random_search_hyperparams(
    X_train, y_train, X_val, y_val, num_experiments=45)
762 #%% md
763 os melhores parametros encontrados com o Random
Search foram:
764 Experiment with lr 0.001, lr_decay 0.9, reg 0.1,
    hidden_size 100, num_iters 1500, batch_size 200
765 Experiment with lr 0.001, lr_decay 0.9, reg 0.1,
    hidden_size 100: Validation accuracy 0.491
766 Experiment with lr 0.001, lr_decay 0.97, reg 0.1,
    hidden_size 100, num_iters 3000, batch_size 100
767 Experiment with lr 0.001, lr_decay 0.97, reg 0.1,
    hidden_size 100: Validation accuracy 0.505
768 Experiment with lr 0.0005, lr_decay 0.97, reg 0.25,
    hidden_size 200, num_iters 3000, batch_size 100
769 Experiment with lr 0.0005, lr_decay 0.97, reg 0.25,
    hidden_size 200: Validation accuracy 0.508
770 Experiment with lr 0.0005, lr_decay 0.935, reg 0.05

```

```
770 , hidden_size 100, num_iters 3000, batch_size 100
771 Experiment with lr 0.0005, lr_decay 0.935, reg 0.05
    , hidden_size 100: Validation accuracy 0.509
772 Experiment with lr 0.001, lr_decay 0.99, reg 0.1,
    hidden_size 300, num_iters 3000, batch_size 200
773 Experiment with lr 0.001, lr_decay 0.99, reg 0.1,
    hidden_size 300: Validation accuracy 0.512
774 Experiment with lr 0.001, lr_decay 0.9, reg 0.1,
    hidden_size 300, num_iters 3000, batch_size 100
775 Experiment with lr 0.001, lr_decay 0.9, reg 0.1,
    hidden_size 300: Validation accuracy 0.512
776 Experiment with lr 0.0005, lr_decay 0.99, reg 0.1,
    hidden_size 100, num_iters 3000, batch_size 200
777 Experiment with lr 0.0005, lr_decay 0.99, reg 0.1,
    hidden_size 100: Validation accuracy 0.514
778 Experiment with lr 0.001, lr_decay 0.97, reg 0.1,
    hidden_size 300, num_iters 3000, batch_size 100
779 Experiment with lr 0.001, lr_decay 0.97, reg 0.1,
    hidden_size 300: Validation accuracy 0.517
780 Experiment with lr 0.001, lr_decay 0.9, reg 0.05,
    hidden_size 200, num_iters 3000, batch_size 200
781 Experiment with lr 0.001, lr_decay 0.9, reg 0.05,
    hidden_size 200: Validation accuracy 0.53
782 Experiment with lr 0.001, lr_decay 0.9, reg 0.1,
    hidden_size 300, num_iters 3000, batch_size 250
783 Experiment with lr 0.001, lr_decay 0.9, reg 0.1,
    hidden_size 300: Validation accuracy 0.536
784 Experiment with lr 0.001, lr_decay 0.9, reg 0.1,
    hidden_size 200, num_iters 3000, batch_size 200
785 Experiment with lr 0.001, lr_decay 0.9, reg 0.1,
    hidden_size 200: Validation accuracy 0.537
786 Experiment with lr 0.001, lr_decay 0.9, reg 0.05,
    hidden_size 300, num_iters 3000, batch_size 200
787 Experiment with lr 0.001, lr_decay 0.9, reg 0.05,
    hidden_size 300: Validation accuracy 0.546
788 Experiment with lr 0.001, lr_decay 0.9, reg 0.1,
    hidden_size 300, num_iters 3000, batch_size 200
789 Experiment with lr 0.001, lr_decay 0.9, reg 0.1,
    hidden_size 300: Validation accuracy 0.549
790 #%%
791 # grid search
```

```

792 def grid_search_hyperparams(X_train, y_train, X_val
, y_val):
793     # Definindo os limites para os hiperparâmetros
794     hidden_size_options = [100, 200, 300]
795     learning_rate_options = 1e-3
796     learning_rate_decay_options = 0.90
797     reg_options = [ 0.05, 0.1]
798     num_iters_options = 3000
799     batch_size_options = [200, 250]
800     input_size = 32 * 32 * 3
801     num_classes = 10
802
803
804     best_val_acc = 0
805     best_params = {}
806
807     for i in hidden_size_options:
808         for j in reg_options:
809             for k in batch_size_options:
810
811
812                 # Cria uma nova instância da rede
813                 # neural com os hiperparâmetros escolhidos
814                 net = TwoLayerNet(input_size , i,
815                         num_classes)
816
817                 print(f"Experiment with lr {
818                     learning_rate_options}, lr_decay {
819                     learning_rate_decay_options}, reg {j}, hidden_size {
820                         i}, num_iters {num_iters_options}, batch_size {k}")
821                 # Treina a rede neural
822                 stats = net.train(X_train, y_train,
823                         X_val, y_val,
824                         num_iters=
825                         num_iters_options, batch_size= k,
826                         learning_rate=
827                         learning_rate_options , learning_rate_decay=
828                         learning_rate_decay_options,
829                         reg=j, verbose=
830                         False)
831

```

```

822          # Avalia o desempenho da rede na
823          # validação
824          val_acc = stats['val_acc_history'][-
825          1]
826          print(f"Experiment with lr {-
827          learning_rate_options}, lr_decay {-
828          learning_rate_decay_options}, reg {j}, hidden_size {-
829          i}, batch_size {k}, Validation accuracy {val_acc}")
830
831          # Atualiza os melhores parâmetros,
832          # se necessário
833          if val_acc > best_val_acc:
834              best_net = net
835              best_val_acc = val_acc
836              best_params = {
837                  'hidden_size': i,
838                  'learning_rate':
839                  learning_rate_options,
840                  'learning_rate_decay':
841                  learning_rate_decay_options,
842                  'reg': j,
843                  'num_iters':
844                  num_iters_options ,
845                  'batch_size': k
846              }
847
848
849 #####

```

```
849 #####  
850 # FIM DE SEU CÓDIGO  
#  
851 #####  
#####  
852 %% md  
853  
854 %% md  
855 ## Representando graficamente os resultados obtidos  
856 %%  
857 # Plota a função de pedar e as acurácia de  
# treinamento e validação  
858 plt.subplot(2, 1, 1)  
859 plt.plot(stats['loss_history'])  
860 plt.title('Loss history')  
861 plt.xlabel('Iteration')  
862 plt.ylabel('Loss')  
863  
864 plt.subplot(2, 1, 2)  
865 plt.plot(stats['train_acc_history'], label='train')  
866 plt.plot(stats['val_acc_history'], label='val')  
867 plt.title('Classification accuracy history')  
868 plt.xlabel('Epoch')  
869 plt.ylabel('Classification accuracy')  
870 plt.tight_layout()  
871 plt.show()  
872 %% md  
873 ## Visualizando pesos da primeira camada  
874 %%  
875 # visualizar os pesos da primeira camada da melhor  
# rede obtida  
876 show_net_weights(best_net)  
877 %% md  
878 # Executar predições sobre o conjunto de teste  
879 Quando você terminar com seus experimentos acima (**  
nunca antes!!!**), você deve avaliar sua rede final  
sobre o conjunto de teste e o resultado de acurácia  
(taxa de acerto) deve ser acima de 48%.  
880  
881 **Você deve tentar encontrar um resultado de  
acurácia igual ou acima de 52%.**
```

```
882 #%%
883 test_acc = (best_net.predict(X_test) == y_test).mean()
884 print('Test accuracy: ', test_acc)
885 #%%
886 # save to pdf
```