Xiangyu Zheng 21331025 CS7CS4 Machine Learning week8 assignment

(i) (a)

The function is named simpleCNN, and the parameters are input and kernel

```
def simpleCNN(input, kernel):
```

Get the dimension of the matrix and calculate it according to the four for loops

```
for i in range(0, i_l - 1):
        for j in range(0, i_w - 1):
                for ki in range(0, k_l - 1):
                        for kj in range(0, k_w - 1):
```

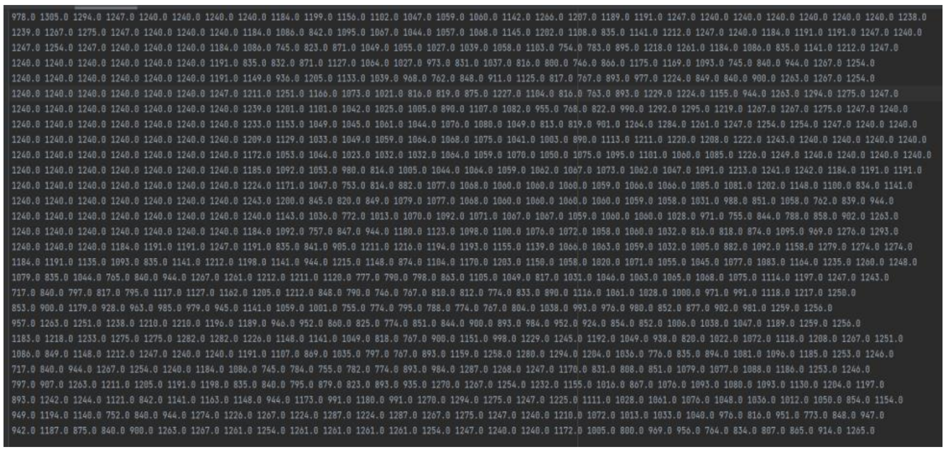
(b)

A 30*30 picture is selected.

Create two kernels:

```
kernel1 = [[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]]
kernel2 = [[0, -1, 0], [-1, 8, -1], [0, -1, 0]]
```

Call the function and output the two 28*28 metrics and picture results：



```
978.0 1305.0 1294.0 1247.0 1240.0 1240.0 1240.0 1240.0 1184.0 1199.0 1156.0 1102.0 1047.0 1059.0 1060.0 1142.0 1266.0 1207.0 1189.0 1191.0 1247.0 1240.0 1240.0 1240.0 1240.0 1240.0 1240.0 1238.0
1239.0 1267.0 1275.0 1247.0 1240.0 1240.0 1240.0 1184.0 1086.0 842.0 1095.0 1067.0 1044.0 1057.0 1068.0 1145.0 1202.0 1108.0 835.0 1141.0 1212.0 1247.0 1240.0 1184.0 1191.0 1191.0 1247.0 1240.0
1247.0 1254.0 1247.0 1240.0 1240.0 1240.0 1184.0 1086.0 745.0 823.0 871.0 1049.0 1055.0 1027.0 1039.0 1058.0 1103.0 754.0 783.0 895.0 1218.0 1261.0 1184.0 1086.0 835.0 1141.0 1212.0 1247.0
1240.0 1240.0 1240.0 1240.0 1240.0 1240.0 1191.0 835.0 832.0 871.0 1127.0 1064.0 1027.0 973.0 831.0 1037.0 816.0 800.0 746.0 866.0 1175.0 1169.0 1093.0 745.0 840.0 944.0 1267.0 1254.0
1240.0 1240.0 1240.0 1240.0 1240.0 1240.0 1191.0 1149.0 936.0 1205.0 1133.0 1039.0 968.0 762.0 848.0 911.0 1125.0 817.0 767.0 893.0 977.0 1224.0 849.0 840.0 900.0 1263.0 1267.0 1254.0
1240.0 1240.0 1240.0 1240.0 1240.0 1240.0 1247.0 1211.0 1251.0 1166.0 1073.0 1021.0 816.0 819.0 875.0 1227.0 1104.0 816.0 763.0 893.0 1229.0 1224.0 1155.0 944.0 1263.0 1294.0 1275.0 1247.0
1240.0 1240.0 1240.0 1240.0 1240.0 1240.0 1240.0 1239.0 1201.0 1101.0 1042.0 1025.0 1005.0 890.0 1107.0 1082.0 955.0 768.0 822.0 990.0 1292.0 1295.0 1219.0 1267.0 1267.0 1275.0 1247.0 1240.0
1240.0 1240.0 1240.0 1240.0 1240.0 1240.0 1240.0 1233.0 1153.0 1049.0 1045.0 1061.0 1044.0 1076.0 1080.0 1049.0 813.0 819.0 901.0 1264.0 1284.0 1261.0 1247.0 1254.0 1254.0 1247.0 1240.0 1240.0
1240.0 1240.0 1240.0 1240.0 1240.0 1240.0 1240.0 1209.0 1129.0 1033.0 1049.0 1059.0 1064.0 1068.0 1075.0 1041.0 1003.0 890.0 1113.0 1211.0 1220.0 1208.0 1222.0 1243.0 1240.0 1240.0 1240.0 1240.0
1240.0 1240.0 1240.0 1240.0 1240.0 1240.0 1240.0 1172.0 1053.0 1044.0 1023.0 1032.0 1032.0 1064.0 1059.0 1070.0 1050.0 1075.0 1095.0 1101.0 1060.0 1085.0 1226.0 1249.0 1240.0 1240.0 1240.0 1240.0
1240.0 1240.0 1240.0 1240.0 1240.0 1240.0 1240.0 1185.0 1092.0 1053.0 980.0 814.0 1005.0 1044.0 1064.0 1059.0 1062.0 1067.0 1073.0 1062.0 1047.0 1091.0 1213.0 1241.0 1242.0 1184.0 1191.0 1191.0
1240.0 1240.0 1240.0 1240.0 1240.0 1240.0 1240.0 1224.0 1171.0 1047.0 753.0 814.0 882.0 1077.0 1068.0 1060.0 1060.0 1060.0 1059.0 1066.0 1066.0 1085.0 1081.0 1202.0 1148.0 1100.0 834.0 1141.0
1240.0 1240.0 1240.0 1240.0 1240.0 1240.0 1240.0 1243.0 1200.0 845.0 820.0 849.0 1079.0 1077.0 1068.0 1060.0 1060.0 1060.0 1060.0 1059.0 1058.0 1031.0 988.0 851.0 1058.0 762.0 839.0 944.0
1240.0 1240.0 1240.0 1240.0 1240.0 1240.0 1240.0 1240.0 1143.0 1036.0 772.0 1013.0 1070.0 1092.0 1071.0 1067.0 1067.0 1059.0 1060.0 1060.0 1028.0 971.0 755.0 844.0 788.0 858.0 902.0 1263.0
1240.0 1240.0 1240.0 1240.0 1240.0 1240.0 1240.0 1184.0 1092.0 757.0 847.0 944.0 1180.0 1123.0 1098.0 1100.0 1076.0 1072.0 1058.0 1060.0 1032.0 816.0 818.0 874.0 1095.0 969.0 1276.0 1293.0
1240.0 1240.0 1240.0 1184.0 1191.0 1191.0 1247.0 1191.0 835.0 841.0 905.0 1211.0 1216.0 1194.0 1193.0 1155.0 1139.0 1066.0 1063.0 1059.0 1032.0 1005.0 882.0 1092.0 1158.0 1279.0 1274.0 1274.0
1184.0 1191.0 1135.0 1093.0 835.0 1141.0 1212.0 1198.0 1141.0 944.0 1215.0 1148.0 874.0 1104.0 1170.0 1203.0 1150.0 1058.0 1020.0 1071.0 1055.0 1045.0 1077.0 1083.0 1164.0 1235.0 1260.0 1248.0
1079.0 835.0 1044.0 765.0 840.0 944.0 1267.0 1261.0 1212.0 1211.0 1120.0 777.0 790.0 798.0 863.0 1105.0 1049.0 817.0 1031.0 1046.0 1063.0 1065.0 1068.0 1075.0 1114.0 1197.0 1247.0 1243.0
717.0 840.0 797.0 817.0 795.0 1117.0 1127.0 1162.0 1205.0 1212.0 848.0 790.0 746.0 767.0 810.0 812.0 774.0 833.0 890.0 1116.0 1061.0 1028.0 1000.0 971.0 991.0 1118.0 1217.0 1250.0
853.0 900.0 1179.0 928.0 963.0 985.0 979.0 945.0 1141.0 1059.0 1001.0 755.0 774.0 795.0 788.0 774.0 767.0 804.0 1038.0 993.0 976.0 980.0 852.0 877.0 902.0 981.0 1259.0 1256.0
957.0 1263.0 1251.0 1238.0 1210.0 1210.0 1196.0 1189.0 946.0 952.0 860.0 825.0 774.0 851.0 844.0 900.0 893.0 984.0 952.0 924.0 854.0 852.0 1006.0 1038.0 1047.0 1189.0 1259.0 1256.0
1183.0 1218.0 1233.0 1275.0 1275.0 1282.0 1282.0 1226.0 1148.0 1141.0 1049.0 818.0 767.0 900.0 1151.0 998.0 1229.0 1245.0 1192.0 1049.0 938.0 820.0 1022.0 1072.0 1118.0 1208.0 1267.0 1251.0
1086.0 849.0 1148.0 1212.0 1247.0 1240.0 1240.0 1191.0 1107.0 869.0 1035.0 797.0 767.0 893.0 1159.0 1258.0 1280.0 1294.0 1204.0 1036.0 776.0 835.0 894.0 1081.0 1096.0 1185.0 1253.0 1246.0
717.0 840.0 944.0 1267.0 1254.0 1240.0 1184.0 1086.0 745.0 784.0 755.0 782.0 774.0 893.0 984.0 1287.0 1268.0 1247.0 1170.0 831.0 808.0 851.0 1079.0 1077.0 1088.0 1186.0 1253.0 1246.0
797.0 907.0 1263.0 1211.0 1205.0 1191.0 1198.0 835.0 840.0 795.0 879.0 823.0 893.0 935.0 1270.0 1267.0 1254.0 1232.0 1155.0 1016.0 867.0 1076.0 1093.0 1080.0 1093.0 1130.0 1204.0 1197.0
893.0 1242.0 1244.0 1121.0 842.0 1141.0 1163.0 1148.0 944.0 1173.0 991.0 1180.0 991.0 1270.0 1294.0 1275.0 1247.0 1225.0 1111.0 1028.0 1061.0 1076.0 1048.0 1036.0 1012.0 1050.0 854.0 1154.0
949.0 1194.0 1140.0 752.0 840.0 944.0 1274.0 1226.0 1267.0 1224.0 1287.0 1224.0 1287.0 1267.0 1275.0 1247.0 1240.0 1210.0 1072.0 1013.0 1033.0 1040.0 976.0 816.0 951.0 773.0 848.0 947.0
942.0 1187.0 875.0 840.0 900.0 1263.0 1267.0 1261.0 1254.0 1261.0 1261.0 1261.0 1261.0 1254.0 1247.0 1240.0 1240.0 1240.0 1172.0 1005.0 800.0 969.0 956.0 764.0 834.0 807.0 865.0 914.0 1265.0
```

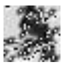
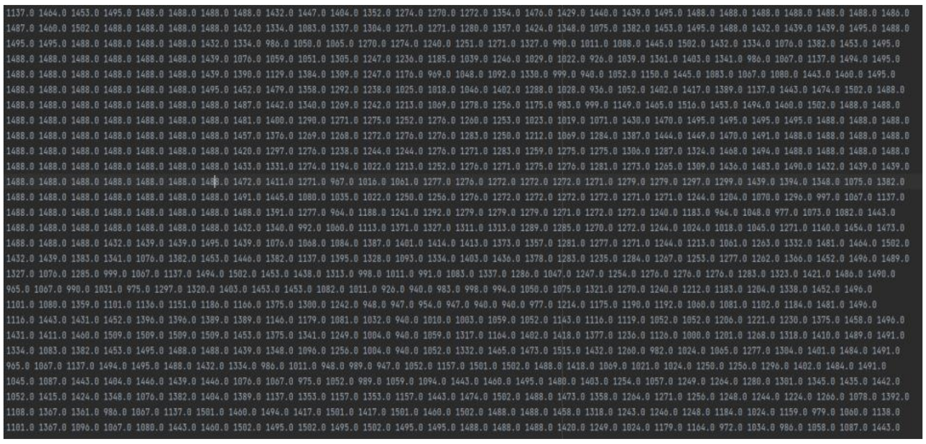


Figure 1.The output metric and picture uses kernel1



```
1137.0 1464.0 1453.0 1495.0 1488.0 1488.0 1488.0 1488.0 1432.0 1447.0 1404.0 1352.0 1274.0 1270.0 1272.0 1354.0 1476.0 1429.0 1440.0 1439.0 1495.0 1488.0 1488.0 1488.0 1488.0 1488.0 1488.0 1486.0
1487.0 1460.0 1502.0 1488.0 1488.0 1488.0 1488.0 1432.0 1334.0 1083.0 1337.0 1304.0 1271.0 1271.0 1280.0 1357.0 1424.0 1348.0 1075.0 1382.0 1453.0 1495.0 1488.0 1432.0 1439.0 1439.0 1495.0 1488.0
1495.0 1495.0 1488.0 1488.0 1488.0 1488.0 1432.0 1334.0 986.0 1050.0 1065.0 1270.0 1274.0 1240.0 1251.0 1271.0 1327.0 990.0 1011.0 1088.0 1445.0 1502.0 1432.0 1334.0 1076.0 1382.0 1453.0 1495.0
1488.0 1488.0 1488.0 1488.0 1488.0 1488.0 1439.0 1076.0 1059.0 1051.0 1305.0 1247.0 1236.0 1185.0 1039.0 1246.0 1029.0 1022.0 926.0 1039.0 1361.0 1403.0 1341.0 986.0 1067.0 1137.0 1494.0 1495.0
1488.0 1488.0 1488.0 1488.0 1488.0 1488.0 1439.0 1390.0 1129.0 1384.0 1309.0 1247.0 1176.0 969.0 1048.0 1092.0 1330.0 999.0 940.0 1052.0 1150.0 1445.0 1083.0 1067.0 1080.0 1443.0 1460.0 1495.0
1488.0 1488.0 1488.0 1488.0 1488.0 1488.0 1495.0 1452.0 1479.0 1358.0 1292.0 1238.0 1025.0 1018.0 1046.0 1402.0 1288.0 1028.0 936.0 1052.0 1402.0 1417.0 1389.0 1137.0 1443.0 1474.0 1502.0 1488.0
1488.0 1488.0 1488.0 1488.0 1488.0 1488.0 1488.0 1487.0 1442.0 1340.0 1269.0 1242.0 1213.0 1069.0 1278.0 1256.0 1175.0 983.0 999.0 1149.0 1465.0 1516.0 1453.0 1494.0 1460.0 1502.0 1488.0 1488.0
1488.0 1488.0 1488.0 1488.0 1488.0 1488.0 1488.0 1481.0 1400.0 1290.0 1271.0 1275.0 1252.0 1276.0 1260.0 1253.0 1023.0 1019.0 1071.0 1430.0 1470.0 1495.0 1495.0 1495.0 1495.0 1488.0 1488.0 1488.0
1488.0 1488.0 1488.0 1488.0 1488.0 1488.0 1488.0 1457.0 1376.0 1269.0 1268.0 1272.0 1276.0 1276.0 1283.0 1250.0 1212.0 1069.0 1284.0 1387.0 1444.0 1449.0 1470.0 1491.0 1488.0 1488.0 1488.0 1488.0
1488.0 1488.0 1488.0 1488.0 1488.0 1488.0 1488.0 1420.0 1297.0 1276.0 1238.0 1244.0 1244.0 1276.0 1271.0 1283.0 1259.0 1275.0 1275.0 1306.0 1287.0 1324.0 1468.0 1494.0 1488.0 1488.0 1488.0 1488.0
1488.0 1488.0 1488.0 1488.0 1488.0 1488.0 1488.0 1433.0 1331.0 1274.0 1194.0 1022.0 1213.0 1252.0 1276.0 1271.0 1275.0 1276.0 1281.0 1273.0 1265.0 1309.0 1436.0 1483.0 1490.0 1432.0 1439.0 1439.0
1488.0 1488.0 1488.0 1488.0 1488.0 1488.0 1488.0 1472.0 1411.0 1271.0 967.0 1016.0 1061.0 1277.0 1276.0 1272.0 1272.0 1272.0 1271.0 1279.0 1279.0 1297.0 1299.0 1439.0 1394.0 1348.0 1075.0 1382.0
1488.0 1488.0 1488.0 1488.0 1488.0 1488.0 1488.0 1491.0 1445.0 1080.0 1035.0 1022.0 1250.0 1256.0 1276.0 1272.0 1272.0 1272.0 1272.0 1271.0 1271.0 1244.0 1204.0 1070.0 1296.0 997.0 1067.0 1137.0
1488.0 1488.0 1488.0 1488.0 1488.0 1488.0 1488.0 1488.0 1391.0 1277.0 964.0 1188.0 1241.0 1292.0 1279.0 1279.0 1279.0 1271.0 1272.0 1272.0 1240.0 1183.0 964.0 1048.0 977.0 1073.0 1082.0 1443.0
1488.0 1488.0 1488.0 1488.0 1488.0 1488.0 1488.0 1432.0 1340.0 992.0 1060.0 1113.0 1371.0 1327.0 1311.0 1313.0 1289.0 1285.0 1270.0 1272.0 1244.0 1024.0 1018.0 1045.0 1271.0 1140.0 1454.0 1473.0
1488.0 1488.0 1488.0 1432.0 1439.0 1439.0 1495.0 1439.0 1076.0 1068.0 1084.0 1387.0 1401.0 1414.0 1413.0 1373.0 1357.0 1281.0 1277.0 1271.0 1244.0 1213.0 1061.0 1263.0 1332.0 1481.0 1464.0 1502.0
1432.0 1439.0 1383.0 1341.0 1076.0 1382.0 1453.0 1446.0 1382.0 1137.0 1395.0 1328.0 1093.0 1334.0 1403.0 1436.0 1378.0 1283.0 1235.0 1284.0 1267.0 1253.0 1277.0 1262.0 1366.0 1452.0 1496.0 1489.0
1327.0 1076.0 1285.0 999.0 1067.0 1137.0 1494.0 1502.0 1453.0 1438.0 1313.0 998.0 1011.0 991.0 1083.0 1337.0 1286.0 1047.0 1247.0 1254.0 1276.0 1276.0 1276.0 1283.0 1323.0 1421.0 1486.0 1490.0
965.0 1067.0 990.0 1031.0 975.0 1297.0 1320.0 1403.0 1453.0 1453.0 1082.0 1011.0 926.0 940.0 983.0 998.0 994.0 1050.0 1075.0 1321.0 1270.0 1240.0 1212.0 1183.0 1204.0 1338.0 1452.0 1496.0
1101.0 1080.0 1359.0 1101.0 1136.0 1151.0 1186.0 1166.0 1375.0 1300.0 1242.0 948.0 947.0 954.0 947.0 940.0 940.0 977.0 1214.0 1175.0 1190.0 1192.0 1060.0 1081.0 1102.0 1184.0 1481.0 1496.0
1116.0 1443.0 1431.0 1452.0 1396.0 1396.0 1389.0 1389.0 1146.0 1179.0 1081.0 1032.0 940.0 1010.0 1003.0 1059.0 1052.0 1143.0 1116.0 1119.0 1052.0 1052.0 1206.0 1221.0 1230.0 1375.0 1458.0 1496.0
1431.0 1411.0 1460.0 1509.0 1509.0 1509.0 1509.0 1453.0 1375.0 1341.0 1249.0 1004.0 940.0 1059.0 1317.0 1164.0 1402.0 1418.0 1377.0 1236.0 1126.0 1000.0 1201.0 1268.0 1318.0 1410.0 1489.0 1491.0
1334.0 1083.0 1382.0 1453.0 1495.0 1488.0 1488.0 1439.0 1348.0 1096.0 1256.0 1004.0 940.0 1052.0 1332.0 1465.0 1473.0 1515.0 1432.0 1260.0 982.0 1024.0 1065.0 1277.0 1304.0 1401.0 1484.0 1491.0
965.0 1067.0 1137.0 1494.0 1495.0 1488.0 1432.0 1334.0 986.0 1011.0 948.0 989.0 947.0 1052.0 1157.0 1501.0 1502.0 1488.0 1418.0 1069.0 1021.0 1024.0 1250.0 1256.0 1296.0 1402.0 1484.0 1491.0
1045.0 1087.0 1443.0 1404.0 1446.0 1439.0 1446.0 1076.0 1067.0 975.0 1052.0 989.0 1059.0 1094.0 1443.0 1460.0 1495.0 1480.0 1403.0 1254.0 1057.0 1249.0 1264.0 1280.0 1301.0 1345.0 1435.0 1442.0
1052.0 1415.0 1424.0 1348.0 1076.0 1382.0 1404.0 1389.0 1137.0 1353.0 1157.0 1353.0 1157.0 1443.0 1474.0 1502.0 1488.0 1473.0 1358.0 1264.0 1271.0 1256.0 1248.0 1244.0 1224.0 1266.0 1078.0 1392.0
1108.0 1367.0 1361.0 986.0 1067.0 1137.0 1501.0 1460.0 1494.0 1417.0 1501.0 1417.0 1501.0 1460.0 1502.0 1488.0 1488.0 1458.0 1318.0 1243.0 1246.0 1248.0 1184.0 1024.0 1159.0 979.0 1060.0 1138.0
1101.0 1367.0 1096.0 1067.0 1080.0 1443.0 1460.0 1502.0 1495.0 1502.0 1495.0 1502.0 1495.0 1495.0 1488.0 1488.0 1488.0 1420.0 1249.0 1024.0 1179.0 1164.0 972.0 1034.0 986.0 1058.0 1087.0 1443.0
```

Figure 2.The output metric and picture uses kernel2

(ii) (a)

The architecture of ConvNet is four Conv2D layers, the padding of each layer is the same to keep the image size, the activation function is relu,

the first and second layers are 16 filters, the output channel are also 16 each,

the third and fourth layers are 32 filters, the output channel are also 32 each,

all The size of filters is 3*3, the stride of the second and fourth layers is 2,

a dropout layer, the value is 0.5,

a flatten layer, an output layer, the channel is 10,

the activation function is softmax, and the l1 regularity is used. Parameter is 0.0001

(b) (i)

There are a total of 37146 parameters in this model, and there are a maximum of 20490 parameters in the dense output layer.

The first Conv2D layer 16 (filters)*3*3(kernel size)*3(picture shape)+16=448

The second Conv2D layer 16 (filters)*3*3(kernel size)*16 (input of the previous layer)+16=2320

The third Conv2D layer 32 (filters) *3*3 *16+32 = 4640

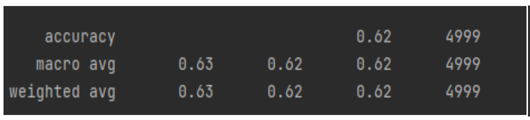
The fourth Conv2D layer 32*3*3*32+32 = 9248

The output using the flatten layer is 2048

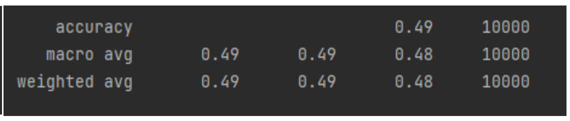
dense layer 10 (output) * 2048 (input) + 10 = 20490

So the dense layer has the most paramters.

According to the classification report, the data accuracy rate on the training set is 62%, and the test set is 49%. The baseline classifier uses DummyRegressor, the strategy is median, and the accuracy rate is only 10%.

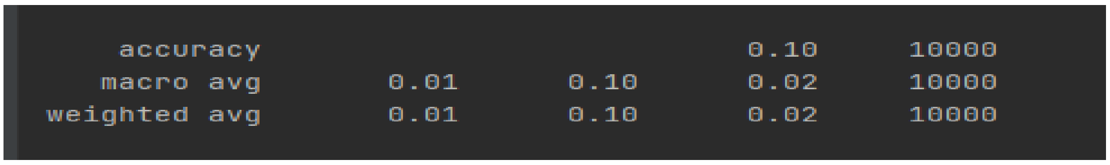


```
    accuracy                           0.62      4999
   macro avg       0.63      0.62      0.62      4999
weighted avg       0.63      0.62      0.62      4999
```



```
    accuracy                           0.49     10000
   macro avg       0.49      0.49      0.48     10000
weighted avg       0.49      0.49      0.48     10000
```

figure 3 training data and test data



```
    accuracy                           0.10     10000
   macro avg       0.01      0.10      0.02     10000
weighted avg       0.01      0.10      0.02     10000
```

figure 4 dummyRegression

(b) (ii)

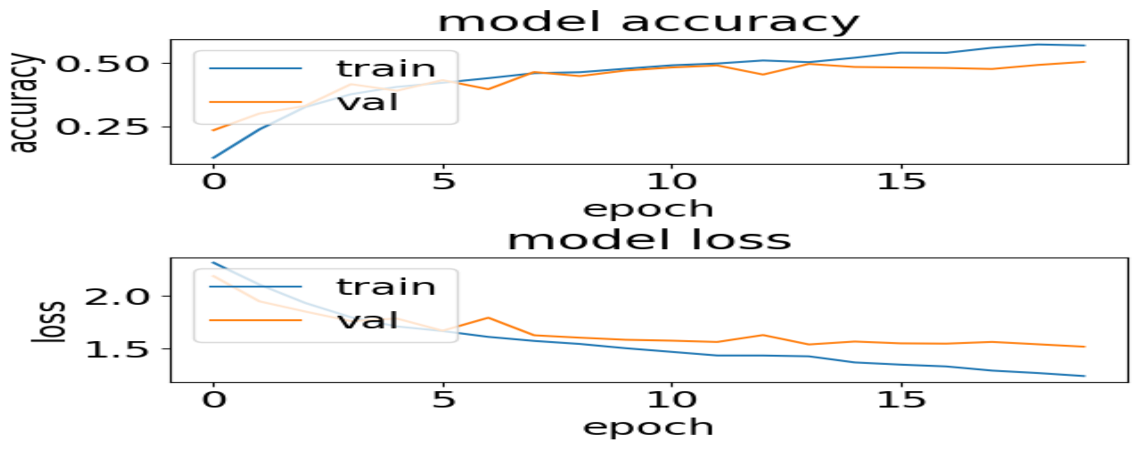


figure 5. n=5000

We can see from the figure that the accuracy of train exceeds the accuracy of validation from about 14 epochs, and the loss of validation is higher than the loss of train from the sixth epoch, which indicates that overfitting has occurred.

(b) (iii)

We can see that the prediction accuracy of figure6-9 training data and test data becomes more and more accurate as the data set increases, and the accuracy and loss curves of train and val are getting closer and closer, which shows that the overfitting situation is getting better. ,

We can see from figure10-11 that when training the network, it also becomes larger as the training set becomes larger.

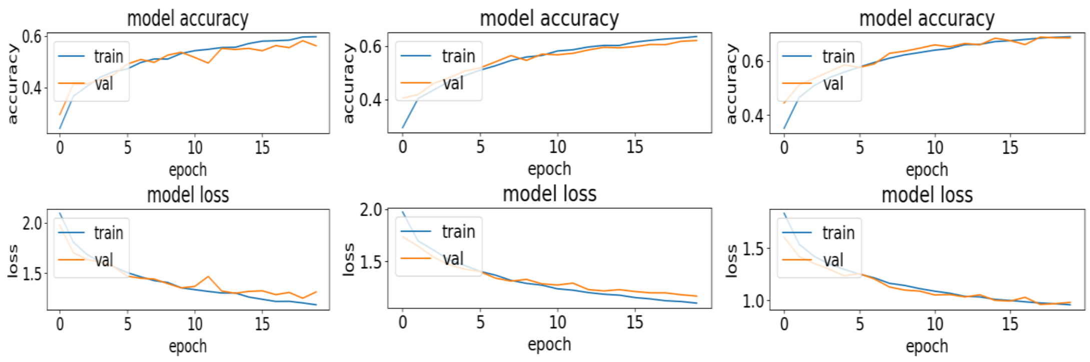
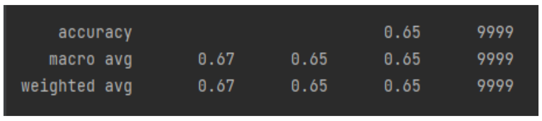
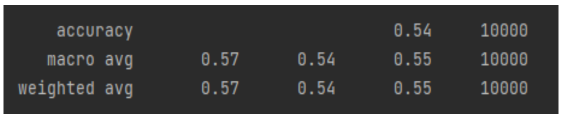


figure 6. n=10000 ,n=20000 and n=40000

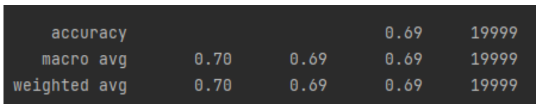


```
    accuracy                          0.65      9999
   macro avg       0.67      0.65      0.65      9999
weighted avg       0.67      0.65      0.65      9999
```

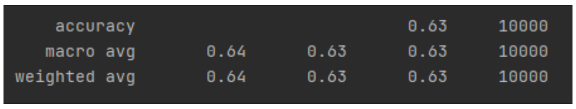


```
    accuracy                          0.54     10000
   macro avg       0.57      0.54      0.55     10000
weighted avg       0.57      0.54      0.55     10000
```

figure 7. train accuracy n=10000 and val accuracy n=10000

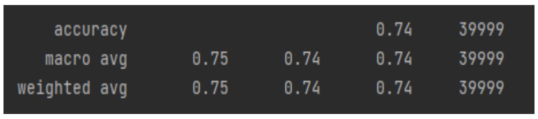


```
    accuracy                          0.69     19999
   macro avg       0.70      0.69      0.69     19999
weighted avg       0.70      0.69      0.69     19999
```

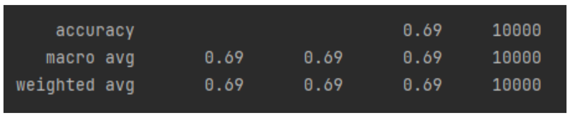


```
    accuracy                          0.63     10000
   macro avg       0.64      0.63      0.63     10000
weighted avg       0.64      0.63      0.63     10000
```

figure 8 train accuracy n=20000 and val accuracy n=20000

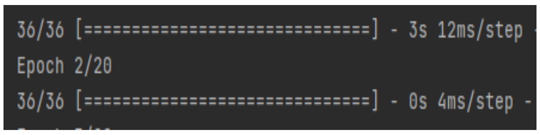


```
    accuracy                          0.74     39999
   macro avg       0.75      0.74      0.74     39999
weighted avg       0.75      0.74      0.74     39999
```

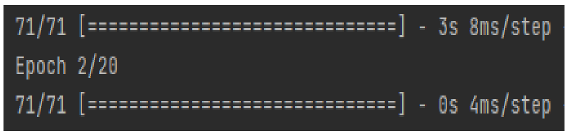


```
    accuracy                          0.69     10000
   macro avg       0.69      0.69      0.69     10000
weighted avg       0.69      0.69      0.69     10000
```

figure 9.train accuracy n=40000 and val accuracy n=40000

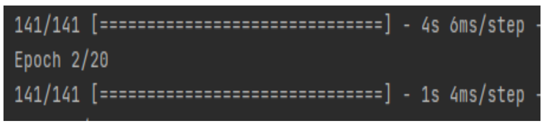


```
36/36 [==============================] - 3s 12ms/step
Epoch 2/20
36/36 [==============================] - 0s 4ms/step -
```

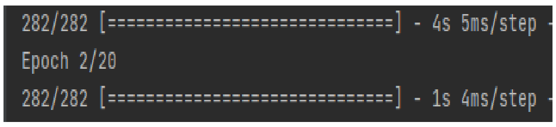


```
71/71 [==============================] - 3s 8ms/step
Epoch 2/20
71/71 [==============================] - 0s 4ms/step
```

figure 10.n=5000 and n=10000 time consuming each epoch



```
141/141 [==============================] - 4s 6ms/step
Epoch 2/20
141/141 [==============================] - 1s 4ms/step
```



```
282/282 [==============================] - 4s 5ms/step
Epoch 2/20
282/282 [==============================] - 1s 4ms/step
```

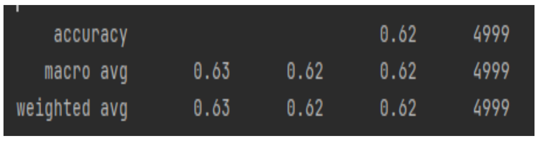
figure 11.n=20000 and n=40000 time consuming each epoch

(b) (iv)

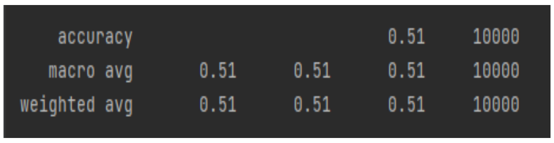
I chose the parameters 0, 0.1, 0.001, 0.0001 for comparison

We can see from figure12-15 that as the value of l1 becomes smaller and smaller, the accuracy rate becomes higher and higher, whether it is train or test

We can see from figure12-15 that no matter what l1 value is set, the effect of solving over-fitting/under-fitting is not as good as increasing the training data.

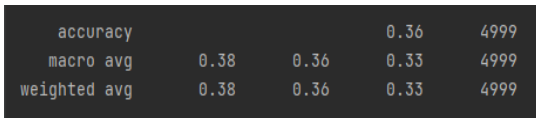


```
    accuracy                        0.62      4999
   macro avg       0.63      0.62   0.62      4999
weighted avg       0.63      0.62   0.62      4999
```

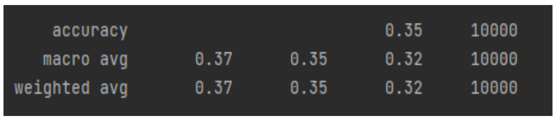


```
    accuracy                        0.51     10000
   macro avg       0.51      0.51   0.51     10000
weighted avg       0.51      0.51   0.51     10000
```

figure 12 train accuracy and val accuracy when l1 = 0

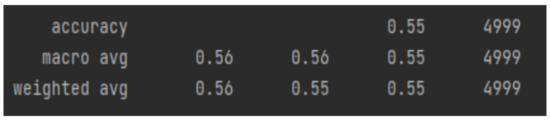


```
    accuracy                        0.36      4999
   macro avg       0.38      0.36   0.33      4999
weighted avg       0.38      0.36   0.33      4999
```

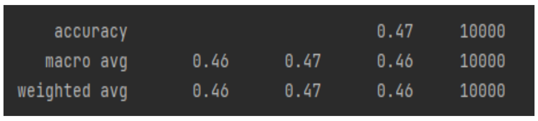


```
    accuracy                        0.35     10000
   macro avg       0.37      0.35   0.32     10000
weighted avg       0.37      0.35   0.32     10000
```

figure 13 train accuracy and val accuracy when l1 = 0.1

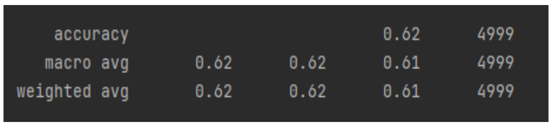


```
    accuracy                        0.55      4999
   macro avg       0.56      0.56   0.55      4999
weighted avg       0.56      0.55   0.55      4999
```

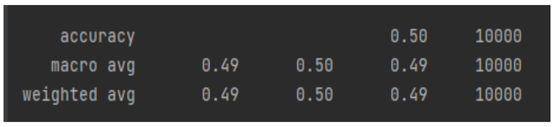


```
    accuracy                        0.47     10000
   macro avg       0.46      0.47   0.46     10000
weighted avg       0.46      0.47   0.46     10000
```

figure 14 train accuracy and val accuracy when l1 = 0.001



```
    accuracy                        0.62      4999
   macro avg       0.62      0.62   0.61      4999
weighted avg       0.62      0.62   0.61      4999
```



```
    accuracy                        0.50     10000
   macro avg       0.49      0.50   0.49     10000
weighted avg       0.49      0.50   0.49     10000
```

figure 15 train accuracy and val accuracy when l1 = 0.0001

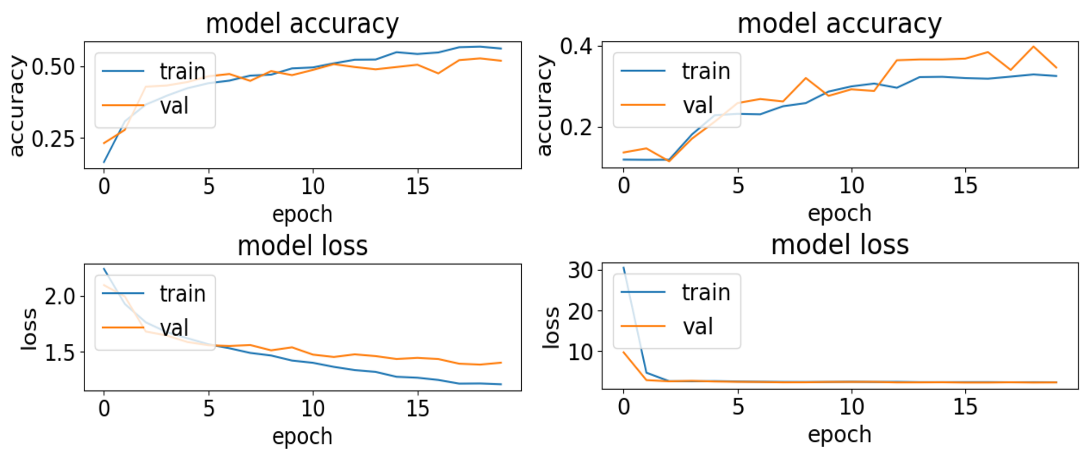


figure 16 l1=0 and l1=0.1

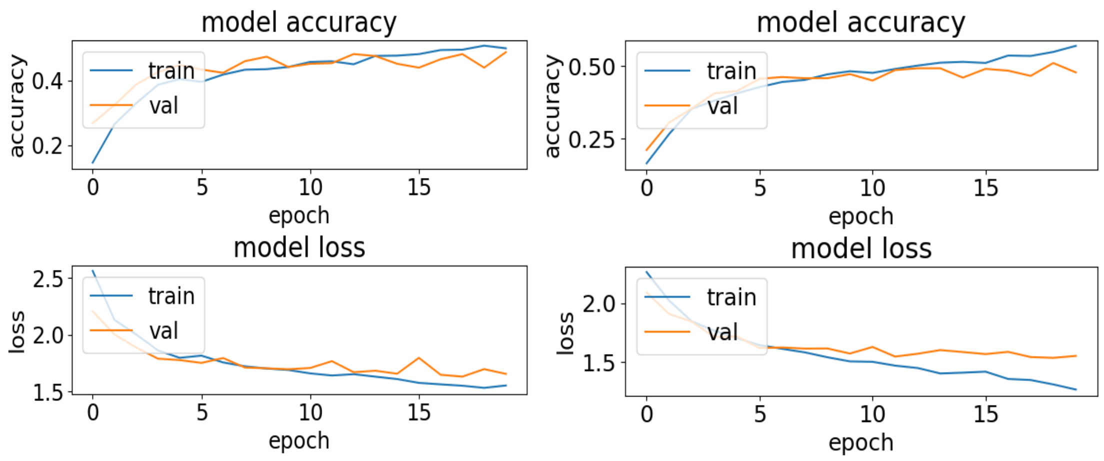


figure 17 l1=0.001 and l1=0.0001

(c) (i)

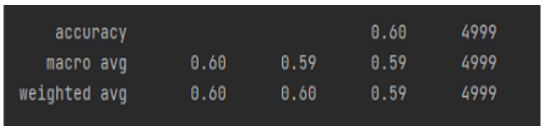
The modified structure:

```
model.add(Conv2D(16, (3, 3), padding='same',
input_shape=x_train.shape[1:], activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(32, (3, 3), padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
```

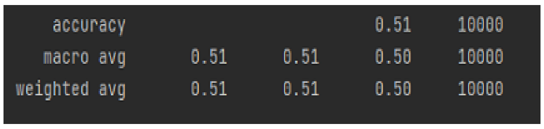
(ii)

This model has a total of 25578 parameters, including 448 in the first conv2d layer, 4,640 in the second conv2d layer, and 20,490 in the output layer.

We can compare the results in figure18 with the previous ones and find that there is no obvious difference in accuracy.



```
    accuracy                        0.60      4999
   macro avg       0.60      0.59      0.59      4999
weighted avg       0.60      0.60      0.59      4999
```



```
    accuracy                        0.51     10000
   macro avg       0.51      0.51      0.50     10000
weighted avg       0.51      0.51      0.50     10000
```

figure 18 max_pooling accuracy

I used the time() function to record the time used to train the two models. I found that the model using max pooling took 35.55 seconds, while the old model took 48.27 seconds. I think this should be the max pooling model with fewer parameters.

model training time:35.55241322517395

model training time:48.27109503746033

(d)

When trained for long enough on the full dataset this should achieve prediction accuracy above 70%. How does adding more layers layers affect the complex trade-off between prediction performance, over/under-fitting, the amount of training data needed and the time taken to train the network.

当在完整数据集上训练足够长的时间时，这应该能达到 70% 以上的预测准确率。 添加更多层如何影响预测性能、过拟合/欠拟合、所需的训练数据量和训练网络所需的时间之间的复杂权衡。

I use 3 models as tests, the first is the original model, and the second is to add two conv2D layers to the original model and use 8 filters:

```
model.add(Conv2D(8, (3, 3), padding='same',
input_shape=x_train.shape[1:], activation='relu'))
model.add(Conv2D(8, (3, 3), strides=(2, 2), padding='same',
activation='relu'))
model.add(Conv2D(16, (3, 3), padding='same', activation='relu'))
model.add(Conv2D(16, (3, 3), strides=(2, 2), padding='same',
activation='relu'))
model.add(Conv2D(32, (3, 3), padding='same', activation='relu'))
model.add(Conv2D(32, (3, 3), strides=(2, 2), padding='same',
activation='relu'))
```

The third one uses only two Conv2D layers and 16 filters:

```
model.add(Conv2D(16, (3, 3), padding='same',
input_shape=x_train.shape[1:], activation='relu'))
model.add(Conv2D(16, (3, 3), strides=(2, 2), padding='same',
activation='relu'))
```

So we have 2 conv2D layer models, 4 conv2D layer models and 6 conv2D layer models

First use 5000 data to test the performance, over-fitting/under-fitting, and the time required.

We can see through figure19-21 that the accuracy of both train and val is the highest in the original model, the second in the 2-layer model, and the lowest in the 6-layer model.

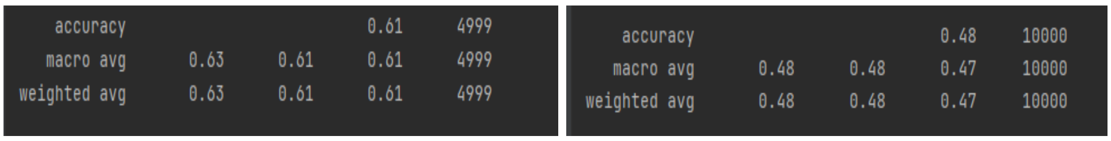
Through figure22, we can know that although the 6-layer model has the lowest accuracy, it does the best in terms of over-fitting/under-fitting

In terms of training model time, the 2-layer model is the fastest.

2layers model training time:5.196512460708618

4layers model training time:6.283828496932983

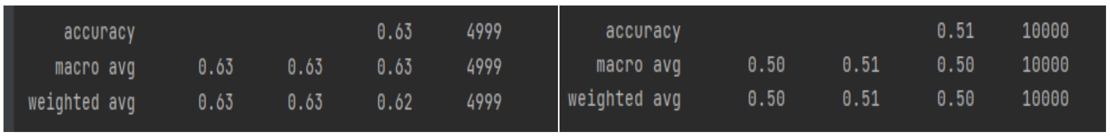
6layers model training time:6.789588212966919



```
    accuracy                        0.61      4999
   macro avg       0.63      0.61      0.61      4999
weighted avg       0.63      0.61      0.61      4999
```

```
    accuracy                        0.48     10000
   macro avg       0.48      0.48      0.47     10000
weighted avg       0.48      0.48      0.47     10000
```

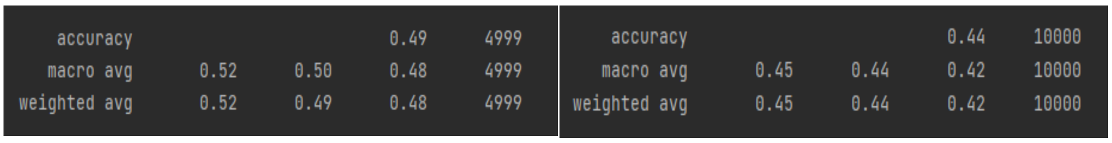
figure 19 2layers model accuracy



```
    accuracy                        0.63      4999
   macro avg       0.63      0.63      0.63      4999
weighted avg       0.63      0.63      0.62      4999
```

```
    accuracy                        0.51     10000
   macro avg       0.50      0.51      0.50     10000
weighted avg       0.50      0.51      0.50     10000
```

figure 20 4layers model accuracy



```
    accuracy                        0.49      4999
   macro avg       0.52      0.50      0.48      4999
weighted avg       0.52      0.49      0.48      4999
```

```
    accuracy                        0.44     10000
   macro avg       0.45      0.44      0.42     10000
weighted avg       0.45      0.44      0.42     10000
```

figure 21 6layers model accuracy

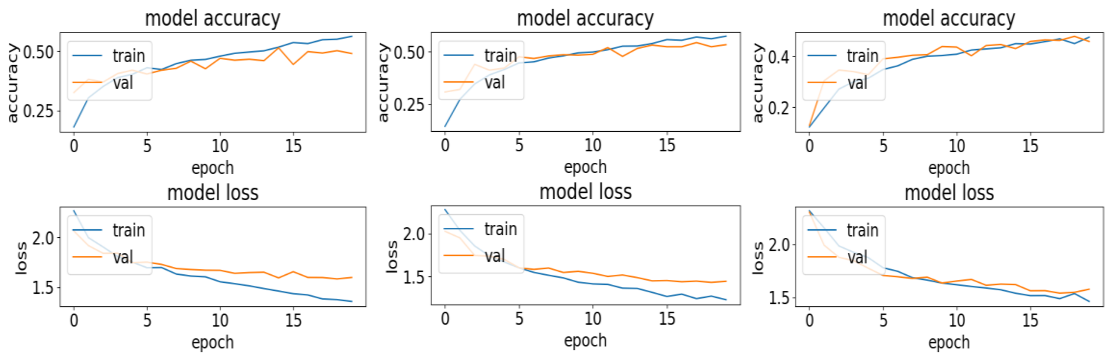
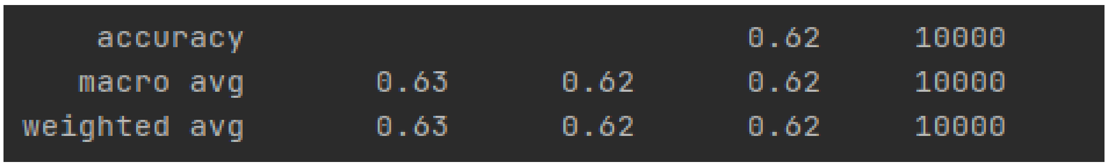


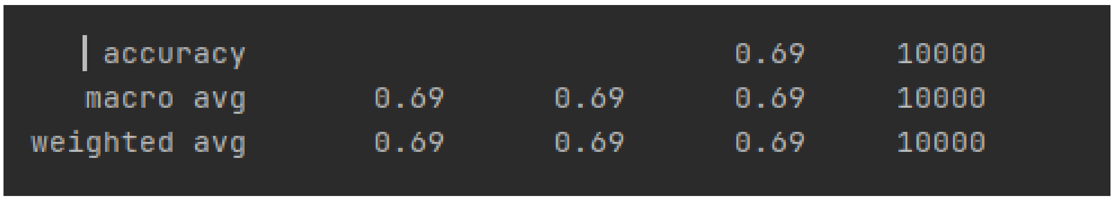
figure 22 2layers vs 4layers vs 6layers

Finally, we can see that with a data set of 50,000, the accuracy of the original model can reach 69%, which means that a relatively small data set can be required to achieve better results. This shows that the appropriate number of layers is the best.



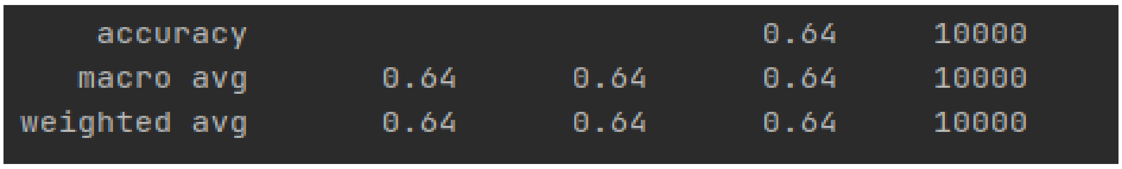
```
    accuracy                          0.62     10000
   macro avg       0.63      0.62     0.62     10000
weighted avg       0.63      0.62     0.62     10000
```

figure 23 2-layer model accuracy with n=50000



```
    accuracy                          0.69     10000
   macro avg       0.69      0.69     0.69     10000
weighted avg       0.69      0.69     0.69     10000
```

figure 24 4-layer model accuracy with n=50000



```
    accuracy                          0.64     10000
   macro avg       0.64      0.64     0.64     10000
weighted avg       0.64      0.64     0.64     10000
```

figure 25 6-layer model accuracy with n=50000

Appendix

code

i-a.py

```
import numpy as np
from PIL import Image

```

```python

def simpleCNN(input, kernel):
    output = []
    k_l = len(kernel)
    k_w = len(kernel[0])
    i_l = len(input)
    i_w = len(input[0])
    for i in range(0, i_l - 1):
        if i + k_l > i_l:
            break
        line = []
        for j in range(0, i_w - 1):
            if j + k_w > i_w:
                continue
            product = 0
            for ki in range(0, k_l - 1):
                for kj in range(0, k_w - 1):
                    product = product + input[i + ki][j + kj] * kernel[ki][kj]

            line.append(product)
        output.append(line)
    return output


im = Image.open('5.png')
rgb = np.array(im.convert('RGB'))
r = rgb[:, :, 0]
# Image.fromarray(np.uint8(r)).show()

kernel1 = [[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]]
kernel2 = [[0, -1, 0], [-1, 8, -1], [0, -1, 0]]
np.set_printoptions(threshold=np.inf)
np.set_printoptions(precision=1)
np.savetxt('kernel1.txt',np.array(simpleCNN(r, kernel1)),fmt='%.01f')

np.savetxt('kernel2.txt',np.array(simpleCNN(r, kernel2)),fmt='%.01f')

Image.fromarray(np.uint8(np.array(simpleCNN(r, kernel2)))).show()
# print(np.array(simpleCNN(r, kernel1)))
```

ii.py

```python
from time import time

import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, regularizers
from keras.layers import Dense, Dropout, Activation, Flatten, BatchNormalization
from keras.layers import Conv2D, MaxPooling2D, LeakyReLU
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.utils import shuffle
import matplotlib.pyplot as plt

plt.rc('font', size=18)
plt.rcParams['figure.constrained_layout.use'] = True
import sys

# Model / data parameters
num_classes = 10
input_shape = (32, 32, 3)

```

```python
# the data, split between train and test sets
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
n = 50000
x_train = x_train[1:n]
y_train = y_train[1:n]
# x_test=x_test[1:500]; y_test=y_test[1:500]

# Scale images to the [0, 1] range
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
print("orig x_train shape:", x_train.shape)

# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

use_saved_model = False
if use_saved_model:
    model = keras.models.load_model("cifar.model")
else:
    model = keras.Sequential()
    # model.add(Conv2D(8, (3, 3), padding='same', input_shape=x_train.shape[1:], activation='relu'))
    # model.add(Conv2D(8, (3, 3), strides=(2, 2), padding='same', activation='relu'))
    # model.add(Conv2D(16, (3, 3), padding='same', activation='relu'))
    # model.add(Conv2D(16, (3, 3), strides=(2, 2), padding='same', activation='relu'))


    model.add(Conv2D(16, (3, 3), padding='same', input_shape=x_train.shape[1:], activation='relu'))
    model.add(Conv2D(16, (3, 3), strides=(2, 2), padding='same', activation='relu'))
    # model.add(Conv2D(32, (3, 3), padding='same', activation='relu'))
    # model.add(Conv2D(32, (3, 3), strides=(2, 2), padding='same', activation='relu'))


    # model.add(Conv2D(16, (3, 3), padding='same', input_shape=x_train.shape[1:], activation='relu'))
    # model.add(MaxPooling2D(pool_size=(2, 2)))
    # model.add(Conv2D(32, (3, 3), padding='same', activation='relu'))
    # model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.5))
    model.add(Flatten())
    model.add(Dense(num_classes, activation='softmax', kernel_regularizer=regularizers.l1(0.0001)))

    model.compile(loss="categorical_crossentropy", optimizer='adam', metrics=["accuracy"])
    model.summary()

    batch_size = 128
    epochs = 20
    begin_time = time()
    history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1)
    end_time = time()
    print('model training time:'+ str(end_time - begin_time))
    model.save("cifar.model")
    plt.subplot(211)
    plt.plot(history.history['accuracy'])
    plt.plot(history.history['val_accuracy'])
    plt.title('model accuracy')
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.legend(['train', 'val'], loc='upper left')
    plt.subplot(212)
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title('model loss')
    plt.ylabel('loss');
    plt.xlabel('epoch')
    plt.legend(['train', 'val'], loc='upper left')
    plt.show()

preds = model.predict(x_train)
y_pred = np.argmax(preds, axis=1)
y_train1 = np.argmax(y_train, axis=1)
print(classification_report(y_train1, y_pred))
print(confusion_matrix(y_train1, y_pred))

preds = model.predict(x_test)
y_pred = np.argmax(preds, axis=1)
y_test1 = np.argmax(y_test, axis=1)
print(classification_report(y_test1, y_pred))
print(confusion_matrix(y_test1, y_pred))
```

ii-a-i.py

```python
from keras.losses import mean_squared_error
from sklearn.dummy import DummyRegressor
from sklearn.metrics import accuracy_score, classification_report
from tensorflow import keras

num_classes = 10
input_shape = (32, 32, 3)

(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
n = 5000
x_train = x_train[1:n]
y_train = y_train[1:n]

x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255

# y_train = keras.utils.to_categorical(y_train, num_classes)
# y_test = keras.utils.to_categorical(y_test, num_classes)

dummy = DummyRegressor(strategy='median').fit(x_train, y_train)
ydummy = dummy.predict(x_test)
print(classification_report(y_test, ydummy))
```