گزارش پروژه اول هوش محاسباتی

محمدسپهر توکلی کرمانی – ۹۸۳۱۱۱۱

۱) دریافت دیتاست و پیش پردازش

در اولین قدم از پروژه باید دیتای مورد نیاز را دریافت و آنرا برای ورود به شبکه عصبی آماده کنیم. در گام اول دیتاست CIFAR۱۰ را دانلود کرده و با تابع زیر آنرا در یک آرایه جای میدهیم.

```
# read images and make an image array
def makeImageList(path):
    cv_img = []
    for img in path:
      n = cv.imread(img)
      n = n[:,:, ::-1]
      cv_img.append(n)
    return cv_img
```

این کار را یکبار برای داده های تست و یکبار برای داده های ترین انجام میدهیم و ۲ آرایه testData و trainData را بوجود می آوریم.

```
# file paths
path1 = glob.glob("/content/CIFAR10/test/airplane/*.jpg")
path2 = glob.glob("/content/CIFAR10/test/automobile/*.jpg")
path3 = glob.glob("/content/CIFAR10/test/bird/*.jpg")
path4 = glob.glob("/content/CIFAR10/test/cat/*.jpg")

path5 = glob.glob("/content/CIFAR10/train/airplane/*.jpg")
path6 = glob.glob("/content/CIFAR10/train/automobile/*.jpg")
path7 = glob.glob("/content/CIFAR10/train/bird/*.jpg")
path8 = glob.glob("/content/CIFAR10/train/cat/*.jpg")
```

```
# convert list to arrray
classList1 = makeImageList(path1)+makeImageList(path2)+makeImageList(path3)+makeImageList(path4)
testData = np.asarray(classList1)

classList2 = makeImageList(path5)+makeImageList(path6)+makeImageList(path7)+makeImageList(path8)
trainData = np.asarray(classList2)
```

مثلا برای داده های تست ماتریس به ابعاد ۴۰۰۰ در ۳۲ در ۳۲ در ۳ خواهیم داشت. در مرحله بعدی تصاویر را با تابع زیر خاکستری می کنیم.

```
# make all photos gray
testData = rgb2gray(testData)
trainData = rgb2gray(trainData)
```

سپس نرمالسازی میکنیم تا مقادیر بین ۰ و۱ باشند و reshape می کنیم تا مطابق ورودی شبکه ۱۰۲۴ نورون باشند.

```
# divide by 255 to normalize
testData = np.divide(testData,255)
trainData = np.divide(trainData,255)

# reshape to 1024 nuron
testData = testData.reshape(-1,1024)
trainData = trainData.reshape(-1,1024)
```

برای شافل کردن دیتا و لیبل ها از تابع زیر استفاده می کنیم:

```
# shuffle matrix
def unison_shuffled_copies(a, b):
    assert len(a) == len(b)
    p = np.random.permutation(len(a))
    return a[p], b[p]
```

و برای ساخت لیبل ها از ماتریس one-hot به شکل زیر استفاده می کنیم:

```
# make the labels for Test data
def makeLabelTest():
  label=np.zeros(16000, dtype=np.uint8)
  label=label.reshape(4000,4)
  for i in range(4000):
      if(i<1000):
        label[i,0] = 1
      elif(i<2000):
        label[i,1] = 1
      elif(i<3000):
        label[i,2] = 1
      elif(i<4000):
        label[i,3] = 1
  return label

# make the labels for Train data
def makeLabelTrain():
  label=np.zeros(80000, dtype=np.uint8)
  label=label.reshape(20000,4)
  for i in range(20000):
      if(i<5000):
        label[i,0] = 1
      elif(i<10000):
        label[i,1] = 1
      elif(i<15000):
        label[i,2] = 1
      elif(i<20000):
        label[i,3] = 1
  return label
```

```
#make label matrix
testLabel = makeLabelTest()
trainLabel = makeLabelTrain()
```

در آخر یک مجموعه برای تست و یک مجموعه برای ترین شامل دیتا و لبیل برای هر قسمت در نظر می گیریم :

```
# make a set which contains data and labels
train_set = []
test_set = []
for i in range(trainLabel.T.shape[1]):
  train_set.append((trainData.T[:, i].reshape(1024, 1), trainLabel.T[:, i].reshape(4, 1)))
for i in range(testLabel.T.shape[1]):
  test_set.append((testData[i, :].T.reshape(1024, 1), testLabel[i, :].T.reshape(4, 1)))
```

هایپرپارامتر ها را نیز به شکل زیر در ابتدا تعریف میکنیم :

```
# Hyperparametrs
number_of_train = 200
number_of_epochs = 10
batch_size = 16
batch_num = 20
learning_rate = 0.3
```

حالا دیتاست ما آماده ورود به شبکه عصبی است.

۲) feed forward

در این گام ابتدا ۲۰۰ داده اول را جدا میکنیم و سپس خروجی را برای آن ها حساب میکنیم .

وزن ها را اعدادی رندوم و بایاس ها را صفر initialize میکنیم .

```
# random the weight and bias of layers
W1 = np.random.normal(size=(16,1024))
W2 = np.random.normal(size=(16, 16))
W3 = np.random.normal(size=(4, 16))
b1 = np.zeros((16, 1))
b2 = np.zeros((16, 1))
b3 = np.zeros((4, 1))
```

تابع سیگموید نیز به شکل زیر تعریف می شود :

```
# sigmoid function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

دقت را نیز در آخر حساب میکنیم و میبینم که حدود ۲۰ درصد خواهد بود :

```
counter = 0
# find the sigmoid of each node with formula 1/(1+e^(-W+sig last node + bias)
for i in range (number_of_train):
    a0 = trainData[i].reshape(-1,1)
    a1 = sigmoid(W1 @ a0 + b1)
    a2 = sigmoid(W2 @ a1 + b2)
    a3 = sigmoid(W3 @ a2 + b3)

    output = np.argmax(a3)
    label = np.argmax(trainLabel[i],axis=0)
    if(output == label):counter+=1

print('Accuracy: ', counter/number_of_train * 100,"%")
```

```
Accuracy:  20.5 %
```

back propagation (۳

در این قسمت الگوریتم gradient descend را پیاده می کنیم و مشتقات جزیی را به شکل زیر به دست می آوریم و هدف کمینه کردن تابع هزینه است :

Last layer:

$$\frac{\partial Cost}{\partial w_{jk}^{(3)}} = \frac{\partial Cost}{\partial a_j^{(3)}} \times \frac{\partial a_j^{(3)}}{\partial z_j^{(3)}} \times \frac{\partial z_j^{(3)}}{\partial w_{jk}^{(3)}}$$

$$\frac{\partial Cost}{\partial w_{jk}^{(3)}} = 2(a_j^{(3)} - y_j) \times \sigma'(z_j^{(3)}) \times a_k^{(2)}$$

$$\frac{\partial Cost}{\partial b_j^{(3)}} = \frac{\partial Cost}{\partial a_j^{(3)}} \times \frac{\partial a_j^{(3)}}{\partial z_j^{(3)}} \times \frac{\partial z_j^{(3)}}{\partial b_j^{(3)}}$$

$$\frac{\partial Cost}{\partial b_j^{(3)}} = 2(a_j^{(3)} - y_j) \times \sigma'(z_j^{(3)}) \times 1$$

همانند بخش قبلی ابتدا مقادیر وزن و بایاس ها را مقدار دهی اولیه می کنیم :

```
# random the weight and bias of layers
W1 = np.random.normal(size=(16,1024))
W2 = np.random.normal(size=(16, 16))
W3 = np.random.normal(size=(4, 16))
b1 = np.zeros((16, 1))
b2 = np.zeros((16, 1))
b3 = np.zeros((4, 1))

total_costs = []
```

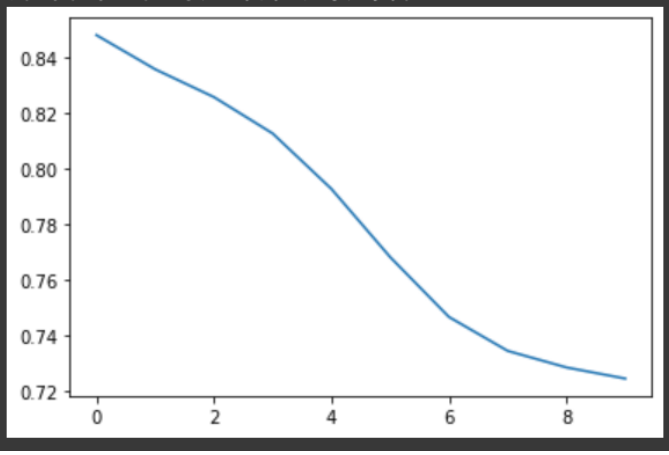
در آخر دقت را حساب میکنیم که حدود ۳۴ درصد است ، نمودار میانگین cost نمونه ها در هر epoch را رسم می کنیم که به شکل نزولی است. واضح است که به دلیل حجم محاسبات زمان اجرا طولانی است.

```
# get the start time
start = datetime.now()

for epoch in range(number_of_epochs):
    batches = [train_set[x:x+batch_size] for x in range(0, number_of_train, batch_size)]
    for batch in batches:
        # allocate grad_W matrix for each layer
        grad_W1 = np.zeros((16, 1024))
        grad_W2 = np.zeros((16, 16))
        grad_W3 = np.zeros((4, 16))
        # allocate grad_b for each layer
        grad_b1 = np.zeros((16, 1))
        grad_b2 = np.zeros((16, 1))
        grad_b3 = np.zeros((4, 1))

        for image, label in batch:
            # compute the output (image is equal to a0)
            a1 = sigmoid(W1 @ image + b1)
            a2 = sigmoid(W2 @ a1 + b2)
            a3 = sigmoid(W3 @ a2 + b3)

            # ---- Last layer
            # weight
            for j in range(grad_W3.shape[0]):
                for k in range(grad_W3.shape[1]):
                    grad_W3[j, k] += 2 * (a3[j, 0] - label[j, 0]) * a3[j, 0] * (1 - a3[j, 0]) * a2[k, 0]

            # bias
            for j in range(grad_b3.shape[0]):
                    grad_b3[j, 0] += 2 * (a3[j, 0] - label[j, 0]) * a3[j, 0] * (1 - a3[j, 0])

            # ---- 3rd layer
            # activation
            delta_3 = np.zeros((16, 1))
            for k in range(16):
                for j in range(4):
                    delta_3[k, 0] += 2 * (a3[j, 0] - label[j, 0]) * a3[j, 0] * (1 - a3[j, 0]) * W3[j, k]

            # weight
            for k in range(grad_W2.shape[0]):
                for m in range(grad_W2.shape[1]):
                    grad_W2[k, m] += delta_3[k, 0] * a2[k,0] * (1 - a2[k, 0]) * a1[m, 0]

            # bias
            for k in range(grad_b2.shape[0]):
                    grad_b2[k, 0] += delta_3[k, 0] * a2[k, 0] * (1 - a2[k, 0])

            # ---- 2nd layer
            # activation
            delta_2 = np.zeros((16, 1))
            for m in range(16):
                for k in range(16):
                    delta_2[m, 0] += delta_3[k, 0] * a2[k, 0] * (1 - a2[k, 0]) * W2[k, m]

            # weight
            for m in range(grad_W1.shape[0]):
                for v in range(grad_W1.shape[1]):
                    grad_W1[m, v] += delta_2[m, 0] * a1[m,0] * (1 - a1[m, 0]) * image[v, 0]
            # bias
            for m in range(grad_b1.shape[0]):
                    grad_b1[m, 0] += delta_2[m, 0] * a1[m, 0] * (1 - a1[m, 0])

        W3 = W3 - (learning_rate * (grad_W3 / batch_size))
        W2 = W2 - (learning_rate * (grad_W2 / batch_size))
        W1 = W1 - (learning_rate * (grad_W1 / batch_size))

        b3 = b3 - (learning_rate * (grad_b3 / batch_size))
        b2 = b2 - (learning_rate * (grad_b2 / batch_size))
        b1 = b1 - (learning_rate * (grad_b1 / batch_size))
```

```
    # calculate cost average per epoch
    cost = 0
    for train_data in train_set[:number_of_train]:
        a0 = train_data[0]
        a1 = sigmoid(W1 @ a0 + b1)
        a2 = sigmoid(W2 @ a1 + b2)
        a3 = sigmoid(W3 @ a2 + b3)

        for j in range(4):
            cost += np.power((a3[j, 0] - train_data[1][j,  0]), 2)

    cost /= number_of_train
    total_costs.append(cost)

# get the finish time
finish = datetime.now()
```

```
# show
epoch_size = [x for x in range(number_of_epochs)]
plt.plot(epoch_size, total_costs)

# find accurancy
correct = 0
for train_data in train_set[:number_of_train]:
    a0 = train_data[0]
    a1 = sigmoid(W1 @ a0 + b1)
    a2 = sigmoid(W2 @ a1 + b2)
    a3 = sigmoid(W3 @ a2 + b3)

    predicted = np.where(a3 == np.amax(a3))
    real = np.where(train_data[1] == np.amax(train_data[1]))

    if predicted == real:
        correct += 1

print('Accuracy: ', correct/number_of_train * 100,"%")
print("Duration time: ", finish-start)
```

```
Accuracy:  34.5 %
Duration time:  0:01:48.190711
```



۴) Vectorization

در این قسمت با استفاده از عملیات روی ماتریس ها سعی می کنیم که سرعت اجرا را کاهش دهیم تا بتوانیم از نمونه های بیشتری برای آموزش شبکه استفاده کنیم و زمان منطقی داشته باشد.
برای نتیجه گیری بهتر تعداد ایپاک ها را ۶۰ قرار می دهیم.

```
# random the weight and bias of layers
W1 = np.random.normal(size=(16,1024))
W2 = np.random.normal(size=(16, 16))
W3 = np.random.normal(size=(4, 16))
b1 = np.zeros((16, 1))
b2 = np.zeros((16, 1))
b3 = np.zeros((4, 1))

total_costs = []
number_of_epochs = 60
```

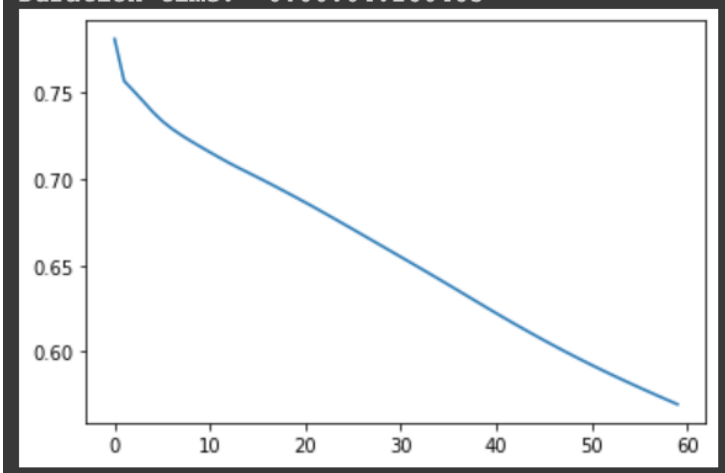
دقت را نیز مانند قبل محاسبه می کنیم که حدود ۵۶ است و زمان اجرا نیز از بخش قبلی خیلی کوتاه تر شده است.

```
# get the start time
start = datetime.now()

for epoch in range(number_of_epochs):
    batches = [train_set[x:x+batch_size] for x in range(0, number_of_train, batch_size)]
    for batch in batches:
        # allocate grad_W matrix for each layer
        grad_W1 = np.zeros((16, 1024))
        grad_W2 = np.zeros((16, 16))
        grad_W3 = np.zeros((4, 16))
        # allocate grad_b for each layer
        grad_b1 = np.zeros((16, 1))
        grad_b2 = np.zeros((16, 1))
        grad_b3 = np.zeros((4, 1))

        for image, label in batch:
            # compute the output (image is equal to a0)
            a1 = sigmoid(W1 @ image + b1)
            a2 = sigmoid(W2 @ a1 + b2)
            a3 = sigmoid(W3 @ a2 + b3)

            # ---- Last layer
            # weight
            grad_W3 += (2 * (a3 - label) * a3 * (1 - a3)) @ np.transpose(a2)

            # bias
            grad_b3 += 2 * (a3 - label) * a3 * (1 - a3)

            # ---- 3rd layer
            # activation
            delta_3 = np.zeros((16, 1))
            delta_3 += np.transpose(W3) @ (2 *(a3 - label) * (a3 * (1 - a3)))

            # weight
            grad_W2 += (a2 * (1 - a2) * delta_3) @ np.transpose(a1)

            # bias
            grad_b2 += delta_3 * a2 * (1 - a2)

            # ---- 2nd layer
            # activation
            delta_2 = np.zeros((16, 1))
            delta_2 += np.transpose(W2) @ delta_3 * a2 * (1 - a2)

            # weight
            grad_W1 += (delta_2 * a1 * (1 - a1)) @ np.transpose(image)

            # bias
            grad_b1 += delta_2 * a1 * (1 - a1)

        W3 = W3 - (learning_rate * (grad_W3 / batch_size))
        W2 = W2 - (learning_rate * (grad_W2 / batch_size))
        W1 = W1 - (learning_rate * (grad_W1 / batch_size))

        b3 = b3 - (learning_rate * (grad_b3 / batch_size))
        b2 = b2 - (learning_rate * (grad_b2 / batch_size))
        b1 = b1 - (learning_rate * (grad_b1 / batch_size))
```

```
        # calculate cost average per epoch
        cost = 0
        for train_data in train_set[:number_of_train]:
            a0 = train_data[0]
            a1 = sigmoid(W1 @ a0 + b1)
            a2 = sigmoid(W2 @ a1 + b2)
            a3 = sigmoid(W3 @ a2 + b3)

            for j in range(4):
                cost += np.power((a3[j, 0] - train_data[1][j,  0]), 2)

        cost /= number_of_train
        total_costs.append(cost)

    # get the finish time
    finish = datetime.now()
```

```
epoch_size = [x for x in range(number_of_epochs)]
plt.plot(epoch_size, total_costs)
correct = 0
for train_data in train_set[:number_of_train]:
    a0 = train_data[0]
    a1 = sigmoid(W1 @ a0 + b1)
    a2 = sigmoid(W2 @ a1 + b2)
    a3 = sigmoid(W3 @ a2 + b3)

    predicted = np.where(a3 == np.amax(a3))
    real = np.where(train_data[1] == np.amax(train_data[1]))

    if predicted == real:
        correct += 1

print('Accuracy: ', correct/number_of_train * 100,"%")
print("Duration time: ", finish-start)
```

```
Accuracy:  56.49999999999999 %
Duration time:  0:00:04.260463
```



۵) تست مدل

در این قسمت از همه ۸۰۰۰ داده ترین استفاده می کنیم و مدل را تست می کنیم :

```
# random the weight and bias of layers
W1 = np.random.normal(size=(16,1024))
W2 = np.random.normal(size=(16, 16))
W3 = np.random.normal(size=(4, 16))
b1 = np.zeros((16, 1))
b2 = np.zeros((16, 1))
b3 = np.zeros((4, 1))

total_costs = []
batch_size = 16
learning_rate = 0.3
number_of_epochs = 60
```

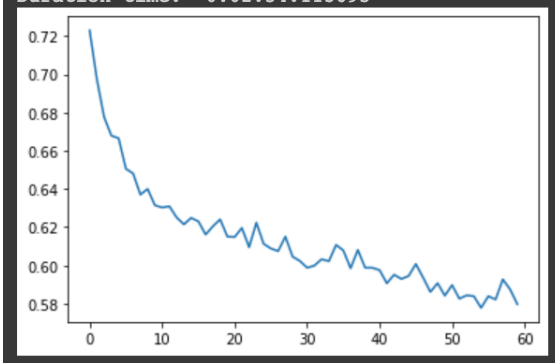
در آخر دقت برای داده های تست حدود ۵۱ و داده های ترین ۵۳ خواهد بود.

```
# get the start time
start = datetime.now()

for epoch in range(number_of_epochs):
    np.random.shuffle(train_set)
    batches = [train_set[x:x+batch_size] for x in range(0, 8000, batch_size)]
    for batch in batches:
        # allocate grad_W matrix for each layer
        grad_W1 = np.zeros((16, 1024))
        grad_W2 = np.zeros((16, 16))
        grad_W3 = np.zeros((4, 16))
        # allocate grad_b for each layer
        grad_b1 = np.zeros((16, 1))
        grad_b2 = np.zeros((16, 1))
        grad_b3 = np.zeros((4, 1))

        for image, label in batch:
            # compute the output (image is equal to a0)
            a1 = sigmoid(W1 @ image + b1)
            a2 = sigmoid(W2 @ a1 + b2)
            a3 = sigmoid(W3 @ a2 + b3)

            # ---- Last layer
            # weight
            grad_W3 += (2 * (a3 - label) * a3 * (1 - a3)) @ np.transpose(a2)

            # bias
            grad_b3 += 2 * (a3 - label) * a3 * (1 - a3)

            # ---- 3rd layer
            # activation
            delta_3 = np.zeros((16, 1))
            delta_3 += np.transpose(W3) @ (2 *(a3 - label) * (a3 * (1 - a3)))

            # weight
            grad_W2 += (a2 * (1 - a2) * delta_3) @ np.transpose(a1)

            # bias
            grad_b2 += delta_3 * a2 * (1 - a2)

            # ---- 2nd layer
            # activation
            delta_2 = np.zeros((16, 1))
            delta_2 += np.transpose(W2) @ delta_3 * a2 * (1 - a2)

            # weight
            grad_W1 += (delta_2 * a1 * (1 - a1)) @ np.transpose(image)

            # bias
            grad_b1 += delta_2 * a1 * (1 - a1)

        W3 = W3 - (learning_rate * (grad_W3 / batch_size))
        W2 = W2 - (learning_rate * (grad_W2 / batch_size))
        W1 = W1 - (learning_rate * (grad_W1 / batch_size))

        b3 = b3 - (learning_rate * (grad_b3 / batch_size))
        b2 = b2 - (learning_rate * (grad_b2 / batch_size))
        b1 = b1 - (learning_rate * (grad_b1 / batch_size))
```

```
    # calculate cost average per epoch
    cost = 0
    for train_data in train_set[0:8000]:
        a0 = train_data[0]
        a1 = sigmoid(W1 @ a0 + b1)
        a2 = sigmoid(W2 @ a1 + b2)
        a3 = sigmoid(W3 @ a2 + b3)

        for j in range(4):
            cost += np.power((a3[j, 0] - train_data[1][j,  0]), 2)

    cost /= 8000
    total_costs.append(cost)

# get the finish time
finish = datetime.now()
```

```
epoch_size = [x for x in range(number_of_epochs)]
plt.plot(epoch_size, total_costs)
correct = 0
for test_data in train_set[0:8000]:
    a0 = test_data[0]
    a1 = sigmoid(W1 @ a0 + b1)
    a2 = sigmoid(W2 @ a1 + b2)
    a3 = sigmoid(W3 @ a2 + b3)

    predicted = np.where(a3 == np.amax(a3))
    real = np.where(test_data[1] == np.amax(test_data[1]))

    if predicted == real:
        correct += 1

print(f"Accuracy For Train Data: {correct/8000 *100 }")
print("Duration time: ", finish-start)
```

```
Accuracy For Train Data: 53.6875
Duration time:  0:02:54.113695
```




```
correct = 0
for test_data in test_set:
    a0 = test_data[0]
    a1 = sigmoid(W1 @ a0 + b1)
    a2 = sigmoid(W2 @ a1 + b2)
    a3 = sigmoid(W3 @ a2 + b3)

    predicted = np.where(a3 == np.amax(a3))
    real = np.where(test_data[1] == np.amax(test_data[1]))

    if predicted == real:
        correct += 1
print(f"Accuracy For Test Data: {correct/4000 *100 }")
```

```
Accuracy For Test Data: 51.475
```

بخش امتیازی

بخش اول

۱)

برای پیاده سازی این شبکه CNN ابتدا مانند قبل باید دیتاست را آمادی کنیم که این کار را با استفاده از keras و tensorflow انجام می دهیم :

```
from tensorflow import keras
from keras import datasets, layers, losses
import numpy as np
import matplotlib.pyplot as plt

(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()
train_images, test_images = train_images / 255.0, test_images / 255.0
```

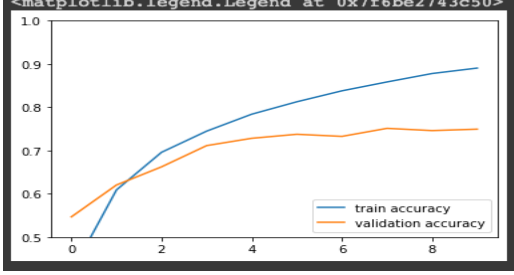
سپس معماری داده شده را پیاده می کنیم :

```
model = keras.Sequential([

    layers.Conv2D(32, (3, 3), padding='same', kernel_initializer='he_uniform', activation='relu', input_shape=(32, 32, 3)),
    layers.Conv2D(32, (3, 3), padding='same', kernel_initializer='he_uniform', activation='relu'),
    layers.MaxPool2D((2, 2)),
    # layers.Dropout(0.25),

    layers.Conv2D(64, (3, 3), padding='same', kernel_initializer='he_uniform', activation='relu'),
    layers.Conv2D(64, (3, 3), padding='same', kernel_initializer='he_uniform', activation='relu'),
    layers.MaxPool2D((2, 2)),
    # layers.Dropout(0.25),

    layers.Conv2D(128, (3, 3), padding='same', kernel_initializer='he_uniform', activation='relu'),
    layers.Conv2D(128, (3, 3), padding='same', kernel_initializer='he_uniform', activation='relu'),
    layers.MaxPool2D((2, 2)),
    # layers.Dropout(0.25),

    layers.Conv2D(128, (3, 3), padding='same', kernel_initializer='he_uniform', activation='relu'),
    layers.Conv2D(128, (3, 3), padding='same', kernel_initializer='he_uniform', activation='relu'),
    layers.MaxPool2D((2, 2)),
    # layers.Dropout(0.25),

    layers.Flatten(),
    layers.Dense(128, kernel_initializer='he_uniform', activation='relu'),
    layers.Dense(128, kernel_initializer='he_uniform', activation='relu'),
    layers.Dense(64, kernel_initializer='he_uniform', activation='relu'),
    # layers.Dropout(0.25),
    layers.Dense(10, kernel_initializer='he_uniform', activation='softmax')

])
```

و بعد مدل را آموزش می دهیم و با داده های تست آنرا ارزیابی می کنیم.

```
plt.plot(history.history['accuracy'], label='train accuracy')
plt.plot(history.history['val_accuracy'], label='validation accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')
```

```
<matplotlib.legend.Legend at 0x7f6be2743c50>
```



مشاهده می شود که مدل به خوبی عمل نمی کند و دچار overfitting شده است.

(۲

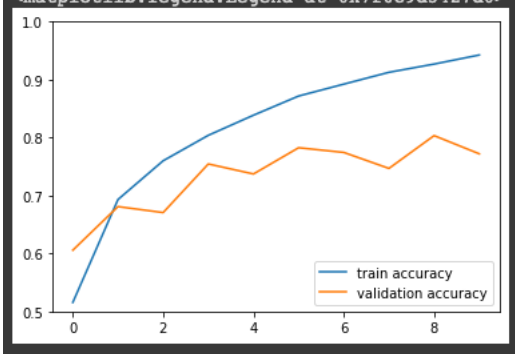
Batch normalization : نرمال‌سازی دسته‌ای تکنیکی برای آموزش شبکه‌های عصبی بسیار عمیق است که ورودی‌های یک لایه را برای هر mini batch اســتاندارد می‌کند به این معنی که میانگین صــفر و انحراف معیار یک خواهند داشــت. این امر باعث تثبیت فرآیند یادگیری و کاهش چشــمگیر تعداد دوره های آموزشــی(epoch) مورد نیاز برای آموزش شبکه های عمیق می شود. به این شکل آنرا به مدل اضافه می کنیم :

```
model = keras.Sequential([

    layers.Conv2D(32, (3, 3), padding='same', kernel_initializer='he_uniform', activation='relu', input_shape=(32, 32, 3)),
    layers.Conv2D(32, (3, 3), padding='same', kernel_initializer='he_uniform', activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPool2D((2, 2)),
    # layers.Dropout(0.25),

    layers.Conv2D(64, (3, 3), padding='same', kernel_initializer='he_uniform', activation='relu'),
    layers.Conv2D(64, (3, 3), padding='same', kernel_initializer='he_uniform', activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPool2D((2, 2)),
    # layers.Dropout(0.25),

    layers.Conv2D(128, (3, 3), padding='same', kernel_initializer='he_uniform', activation='relu'),
    layers.Conv2D(128, (3, 3), padding='same', kernel_initializer='he_uniform', activation='relu'),
        layers.BatchNormalization(),
    layers.MaxPool2D((2, 2)),
    # layers.Dropout(0.25),

    layers.Conv2D(128, (3, 3), padding='same', kernel_initializer='he_uniform', activation='relu'),
    layers.Conv2D(128, (3, 3), padding='same', kernel_initializer='he_uniform', activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPool2D((2, 2)),
    # layers.Dropout(0.25),

    layers.Flatten(),
    layers.Dense(128, kernel_initializer='he_uniform', activation='relu'),
    layers.Dense(128, kernel_initializer='he_uniform', activation='relu'),
    layers.Dense(64, kernel_initializer='he_uniform', activation='relu'),
    # layers.Dropout(0.25),
    layers.BatchNormalization(),
    layers.Dense(10, kernel_initializer='he_uniform', activation='softmax')

])
```

مشاهده میشود که سرعت آموزش کمی بهتر شده و هم چنین دقت داده های تست بهتر شده است :

```
plt.plot(history.history['accuracy'], label='train accuracy')
plt.plot(history.history['val_accuracy'], label='validation accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')
```

```
<matplotlib.legend.Legend at 0x7f6c9d3427d0>
```



(۳

dropout تکنیکی است که در آن نورون‌های انتخابی به شکل تصادفی در طول تمرین نادیده گرفته می‌شوند. این به این معنی است که سهم آنها در فعال سازی نورون های لایه های قبل به طور موقت در گذر رو به جلو حذف می شود و هرگونه به روز رسانی وزنی برای نورون در گذر به عقب اعمال نمی شود. در این مدل با استفاده از keras آن ها را در جای مناسب اضافه می کنیم :

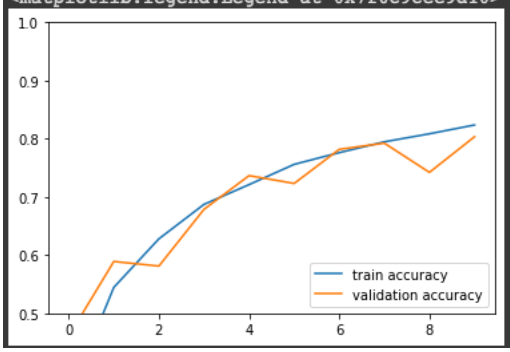
```
model = keras.Sequential([

    layers.Conv2D(32, (3, 3), padding='same', kernel_initializer='he_uniform', activation='relu', input_shape=(32, 32, 3)),
    layers.Conv2D(32, (3, 3), padding='same', kernel_initializer='he_uniform', activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPool2D((2, 2)),
    layers.Dropout(0.25),

    layers.Conv2D(64, (3, 3), padding='same', kernel_initializer='he_uniform', activation='relu'),
    layers.Conv2D(64, (3, 3), padding='same', kernel_initializer='he_uniform', activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPool2D((2, 2)),
    layers.Dropout(0.25),

    layers.Conv2D(128, (3, 3), padding='same', kernel_initializer='he_uniform', activation='relu'),
    layers.Conv2D(128, (3, 3), padding='same', kernel_initializer='he_uniform', activation='relu'),
        layers.BatchNormalization(),
    layers.MaxPool2D((2, 2)),
    layers.Dropout(0.25),

    layers.Conv2D(128, (3, 3), padding='same', kernel_initializer='he_uniform', activation='relu'),
    layers.Conv2D(128, (3, 3), padding='same', kernel_initializer='he_uniform', activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPool2D((2, 2)),
    layers.Dropout(0.25),

    layers.Flatten(),
    layers.Dense(128, kernel_initializer='he_uniform', activation='relu'),
    layers.Dense(128, kernel_initializer='he_uniform', activation='relu'),
    layers.Dense(64, kernel_initializer='he_uniform', activation='relu'),
    layers.Dropout(0.25),
    layers.BatchNormalization(),
    layers.Dense(10, kernel_initializer='he_uniform', activation='softmax')

])
```

مشـاهده می شـود که خطا تا کمتر از ۱ درصـد در ۱۰ ایپاک رسـیده و دقت که در مرحله قبلی حدود ۷۴ بود ه بالای ۸۰ درصد رسیده است و مشکل overfitting بر طرف شده است :

```
Epoch 10/10
1000/1000 [==============================] - 8s 8ms/step - loss: 0.5314 - accuracy: 0.8236 - val_loss: 0.6012 - val_accuracy: 0.8034
```

```
plt.plot(history.history['accuracy'], label='train accuracy')
plt.plot(history.history['val_accuracy'], label='validation accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')
```

```
<matplotlib.legend.Legend at 0x7f6c9cee9d10>
```



۴)
بهینه‌سـازها کلاس‌ها یا متد هایی هسـتند که برای تغییر ویژگی‌های مدل یادگیری عمیق ماشـین شـما مانند وزن‌ها و نرخ یادگیری به منظور کاهش خطاها استفاده می‌شوند. بهینه سازها به دریافت سریعتر نتایج کمک می کنند.

انواع مختلفی از optimizer ها داریم که چند مورد از آن ها را در اینا توضیح می دهیم :

- Adadelta: Optimizer that implements the Adadelta algorithm.
- Adagrad: Optimizer that implements the Adagrad algorithm.
- Adam: Optimizer that implements the Adam algorithm.
- Adamax: Optimizer that implements the Adamax algorithm.
- Ftrl: Optimizer that implements the FTRL algorithm.
- Nadam: Optimizer that implements the NAdam algorithm.
- Optimizer class: Base class for Keras optimizers.
- RMSprop: Optimizer that implements the RMSprop algorithm.
- SGD: Gradient descent (with momentum) optimizer.

در اینجا ما بهیه ساز های adam و sgd و adadelta را با هم بررسی و مقایسه می کنیم:

1) Adam

```
model.compile(optimizer='adam', loss=losses.SparseCategoricalCrossentropy(), metrics=['accuracy'])
history = model.fit(train_images, train_labels, batch_size=50, epochs=10, validation_data=(test_images, test_labels))
```

```
Epoch 1/10
1000/1000 [==============================] - 10s 9ms/step - loss: 1.7719 - accuracy: 0.3550 - val_loss: 1.5199 - val_accuracy: 0.4606
Epoch 2/10
1000/1000 [==============================] - 8s 8ms/step - loss: 1.2762 - accuracy: 0.5445 - val_loss: 1.1306 - val_accuracy: 0.5890
Epoch 3/10
1000/1000 [==============================] - 8s 8ms/step - loss: 1.0692 - accuracy: 0.6279 - val_loss: 1.2194 - val_accuracy: 0.5811
Epoch 4/10
1000/1000 [==============================] - 8s 8ms/step - loss: 0.9165 - accuracy: 0.6872 - val_loss: 0.9393 - val_accuracy: 0.6786
Epoch 5/10
1000/1000 [==============================] - 8s 8ms/step - loss: 0.8204 - accuracy: 0.7212 - val_loss: 0.7685 - val_accuracy: 0.7366
Epoch 6/10
1000/1000 [==============================] - 8s 8ms/step - loss: 0.7285 - accuracy: 0.7559 - val_loss: 0.8398 - val_accuracy: 0.7231
Epoch 7/10
1000/1000 [==============================] - 8s 8ms/step - loss: 0.6672 - accuracy: 0.7761 - val_loss: 0.6444 - val_accuracy: 0.7818
Epoch 8/10
1000/1000 [==============================] - 8s 8ms/step - loss: 0.6147 - accuracy: 0.7947 - val_loss: 0.6061 - val_accuracy: 0.7924
Epoch 9/10
1000/1000 [==============================] - 8s 8ms/step - loss: 0.5730 - accuracy: 0.8085 - val_loss: 0.7772 - val_accuracy: 0.7422
Epoch 10/10
1000/1000 [==============================] - 8s 8ms/step - loss: 0.5314 - accuracy: 0.8236 - val_loss: 0.6012 - val_accuracy: 0.8034
```

2) SGD

```
model.compile(optimizer='sgd', loss=losses.SparseCategoricalCrossentropy(), metrics=['accuracy'])
history = model.fit(train_images, train_labels, batch_size=50, epochs=10, validation_data=(test_images, test_labels))
```

```
Epoch 1/10
1000/1000 [==============================] - 9s 8ms/step - loss: 0.4343 - accuracy: 0.8542 - val_loss: 0.4902 - val_accuracy: 0.8379
Epoch 2/10
1000/1000 [==============================] - 8s 8ms/step - loss: 0.4094 - accuracy: 0.8632 - val_loss: 0.5061 - val_accuracy: 0.8344
Epoch 3/10
1000/1000 [==============================] - 8s 8ms/step - loss: 0.3992 - accuracy: 0.8659 - val_loss: 0.4771 - val_accuracy: 0.8451
Epoch 4/10
1000/1000 [==============================] - 8s 8ms/step - loss: 0.3927 - accuracy: 0.8695 - val_loss: 0.4720 - val_accuracy: 0.8437
Epoch 5/10
1000/1000 [==============================] - 8s 8ms/step - loss: 0.3833 - accuracy: 0.8715 - val_loss: 0.4711 - val_accuracy: 0.8489
Epoch 6/10
1000/1000 [==============================] - 8s 8ms/step - loss: 0.3806 - accuracy: 0.8725 - val_loss: 0.4778 - val_accuracy: 0.8470
Epoch 7/10
1000/1000 [==============================] - 8s 8ms/step - loss: 0.3742 - accuracy: 0.8741 - val_loss: 0.4924 - val_accuracy: 0.8414
Epoch 8/10
1000/1000 [==============================] - 8s 8ms/step - loss: 0.3728 - accuracy: 0.8758 - val_loss: 0.4904 - val_accuracy: 0.8419
Epoch 9/10
1000/1000 [==============================] - 9s 9ms/step - loss: 0.3652 - accuracy: 0.8769 - val_loss: 0.4706 - val_accuracy: 0.8486
Epoch 10/10
1000/1000 [==============================] - 8s 8ms/step - loss: 0.3627 - accuracy: 0.8799 - val_loss: 0.4910 - val_accuracy: 0.8421
```

3) Adadelta

```
model.compile(optimizer='adadelta', loss=losses.SparseCategoricalCrossentropy(), metrics=['accuracy'])
history = model.fit(train_images, train_labels, batch_size=50, epochs=10, validation_data=(test_images, test_labels))
```

```
Epoch 1/10
1000/1000 [==============================] - 10s 9ms/step - loss: 0.3535 - accuracy: 0.8833 - val_loss: 0.4663 - val_accuracy: 0.8514
Epoch 2/10
1000/1000 [==============================] - 8s 8ms/step - loss: 0.3545 - accuracy: 0.8812 - val_loss: 0.4657 - val_accuracy: 0.8519
Epoch 3/10
1000/1000 [==============================] - 9s 9ms/step - loss: 0.3536 - accuracy: 0.8815 - val_loss: 0.4652 - val_accuracy: 0.8523
Epoch 4/10
1000/1000 [==============================] - 9s 9ms/step - loss: 0.3517 - accuracy: 0.8815 - val_loss: 0.4639 - val_accuracy: 0.8517
Epoch 5/10
1000/1000 [==============================] - 8s 8ms/step - loss: 0.3532 - accuracy: 0.8813 - val_loss: 0.4642 - val_accuracy: 0.8511
Epoch 6/10
1000/1000 [==============================] - 9s 9ms/step - loss: 0.3564 - accuracy: 0.8804 - val_loss: 0.4632 - val_accuracy: 0.8519
Epoch 7/10
1000/1000 [==============================] - 9s 9ms/step - loss: 0.3496 - accuracy: 0.8834 - val_loss: 0.4628 - val_accuracy: 0.8513
Epoch 8/10
1000/1000 [==============================] - 9s 9ms/step - loss: 0.3526 - accuracy: 0.8813 - val_loss: 0.4642 - val_accuracy: 0.8515
Epoch 9/10
1000/1000 [==============================] - 9s 9ms/step - loss: 0.3513 - accuracy: 0.8841 - val_loss: 0.4621 - val_accuracy: 0.8516
Epoch 10/10
1000/1000 [==============================] - 9s 9ms/step - loss: 0.3529 - accuracy: 0.8823 - val_loss: 0.4631 - val_accuracy: 0.8516
```

به نظر می رسد که adadelta بهتر از بقیه عمل کرده و دقت به حدود ۸۵ رسیده است هم چنین میزان خطا کاهش بیشتری داشته است .

۵)

مهمترین معیار های ارزیابی در مسائل کلاس بندی ، accuracy, precision, و recall هستند.

accuracy: به عنوان درصد پیش بینی های صحیح برای داده های تست تعریف می شود. می توان آن را به راحتی با تقسیم تعداد پیش بینی های صحیح بر تعداد کل پیش بینی ها محاسبه کرد.

precision: به عنوان کسری از مثال‌های مرتبط (مثبت‌های واقعی) در بین همه نمونه‌هایی که پیش‌بینی شده بود به یک کلاس خاص تعلق دارند، تعریف می‌شود.

recall: یادآوری به عنوان کسری از مثال‌ها تعریف می‌شود که پیش‌بینی می‌شد با توجه به همه نمونه‌هایی که واقعاً به کلاس تعلق دارند، متعلق به یک کلاس باشند.

F1 : که میانگین هارمونیک از recall و precision است.

```
model.compile(optimizer="sgd",
              loss="binary_crossentropy",
              metrics=[keras.metrics.Precision(), keras.metrics.Recall()])
```

به این صورت و در آرایه metrics می توانیم مقادیر آن ها را به دست آوریم.

بخش دوم

۱)

افزایش داده ها (data augmentation) فرآیند اصلاح یا "افزایش" یک مجموعه داده با داده های اضافی است. این داده های اضافی می تواند هر چیزی از تصویر گرفته تا متن باشد و استفاده از آن در الگوریتم های یادگیری ماشینی به بهبود عملکرد آنها کمک می کند.

مثلاً می‌خواهیم مدلی برای طبقه‌بندی نژادهای سگ بسازیم و از اکثر نژادها به جز پاگ، تصاویر زیادی داریم. در نتیجه، مدل نمی تواند پاگ ها را به خوبی طبقه بندی کند. ما می‌توانیم داده‌ها را با افزودن برخی تصاویر (واقعی یا جعلی) از پاگ‌ها، یا با ضرب کردن تصاویر پاگ موجود خود (مثلاً با تکثیر و تحریف آنها برای منحصربه‌فرد ساختن آنها) افزایش دهیم.