Computational Intelligence
Dr. Mozayani
Fall 2022
Hoorieh Sabzevari - 98412004
HW1

۱. یکی از رایج‌ترین الگوریتم‌ها در شبکه‌های عصبی Gradient Descent (GD) است که الگوریتمی برای به حداقل رساندن تابع هزینه J(W,b) در هر مرحله است. این به طور مکرر وزن‌ها و بایاس‌ها را در تلاش برای رسیدن به global minimum در یک تابع هزینه آپدیت می‌کند. هنگامی که یک خروجی داریم، این خروجی را با خروجی مورد انتظار مقایسه می‌کنیم و محاسبه می‌کنیم که چقدر خطا دارد. با این خطا، اکنون می توانیم آن را به عقب منتشر کنیم، هر وزن و بایاس را آپدیت کنیم و سعی کنیم این خطا را به حداقل برسانیم.

گرادیان کاهشی تصادفی پارامترها را برای هر iteration آپدیت می‌کند که منجر به تعداد بیشتری آپدیت می‌شود. بنابراین این یک رویکرد سریع‌تر است که به تصمیم گیری سریع‌تر کمک می‌کند.

در روش گرادیان کاهشی minibatch می‌توانیم مجموعه‌ی داده خود را برداریم و آن را به چند تکه یا batch تقسیم کنیم. بنابراین به جای اینکه منتظر بمانیم تا الگوریتم در کل مجموعه داده اجرا شود و تنها پس از آپدیت وزن‌ها و بایاس اجرا شود، در پایان هر کدام به‌اصطلاح mini-batch آپدیت می‌شود. این به ما امکان می‌دهد تا به سرعت به سمت global minimum در تابع هزینه حرکت کنیم و وزن‌ها و بایاس‌ها را چندین بار در هر دوره آپدیت کنیم. رایج‌ترین اندازه‌ها برای mini-batch ۱۶، ۳۲، ۶۴، ۱۲۸، ۲۵۶ و ۵۱۲ هستند. برای تهیه mini-batch، از روش تصادفی کردن مجموعه داده برای تقسیم تصادفی مجموعه داده و سپس پارتیشن بندی آن به تعداد تکه‌های مناسب استفاده می‌کنند. در گرادیان کاهشی، هر batch برابر با کل مجموعه داده است.

می‌توانیم از کد Mini-batch Gradient Descent برای پیاده‌سازی همه نسخه‌های Gradient Descent استفاده کرد، فقط باید mini_batch_size را برابر با یک Stochastic

GD یا تعداد نمونه‌های آموزشی را روی Batch GD تنظیم کنیم. بنابراین، تفاوت اصلی بین Batch، Mini-Batch و Stochastic Gradient Descent تعداد نمونه های استفاده شده برای هر دوره و زمان و تلاش لازم برای رسیدن به مقدار global minimum تابع cost است.

از مشکلات SGD می‌توان به این مورد اشاره کرد که ما پارامترهای زیادی برای آپدیت کردن داریم اما تمام آن‌ها با یک step size یکسانی دارند و به همین علت در یک راستا حرکت نویزی یا زیگزاگی پیدا خواهند کرد و در راستای دیگر خیلی کند حرکت می‌کنند، یعنی به جواب می‌رسد اما بدلیل پرش‌های نابجا دیر به جواب می‌رسد.

مشکل دیگر این الگوریتم این است که در جاهایی که مشتق صفر است مثلا local minimum ها هم می‌ایستد و گیر می‌کند و دیگر مینیمم مطلق را پیدا نمی‌کند. همچنین مقدار پاسخ نهایی نیز تقریبی است.

مشکل سوم هم این است که به دلیل اینکه mini-batch داریم لذا حرکت نویزی پیدا خواهد کرد اما با وجود این گرادیان‌ها، به طور میانگین در مسیر درستی حرکت می‌کنند.

با استفاده از momentum می‌توان مشکل گیر افتادن در local minimum ها را حل کرد و باعث می‌شود از روی قله‌های کوچک بپرد. فرآیند شبیه‌سازی آن مانند توپی است که از تپه‌ای در حال قل خوردن به پایین است. فرق آن با GD این است که learning rate فقط در گرادیان ضرب نمی‌شود بلکه در یک سرعت هم ضرب می‌شود. سرعت هم به نوعی میانگین گرادیان‌های قبلی است. برای رسیدن به مقصد مورد نظر، باید همان جهت را دنبال کنیم و هنگامی که مطمئن شدیم که مسیر درست را دنبال می‌کنیم، قدم های بزرگ‌تری برمی‌داریم و همچنان در همان جهت حرکت می‌کنیم. همین پارامتر سرعت باعث می‌شود که تعداد نوسان‌ها کمتر شود و خیلی سریع‌تر به نقطه‌ی بهینه همگرا شود.

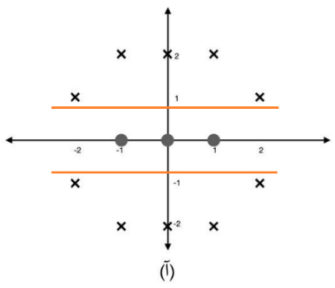
۲.

الف) تعداد نورون‌های لایه‌ی آخر برابر با دو برابر تعداد نقاط کلیدی است. برای مثال در این‌جا برابر با ۱۰ است. زیرا خروجی یک شبکه یک بردار شامل ۱۰ خروجی مرتب‌شده از مختصات ایکس و

ایگرگ هر نقطه‌ی کلیدی است. تابع فعالسازی نیز sigmoid است، زیرا تابع بین صفر و یک است که در واقع احتمال کلیدی بودن را می‌دهد. ویژگی تابع sigmoid این است که در نقاط نزدیک صفر و یک مشتق کمی دارد. لذا نقاطی که احتمال آن‌ها نزدیک صفر و یک است را به قطعیت نسبی شناسایی کرده است، پس تمرکز خود را روی بقیه‌ی نقاط می‌گذارد. از تابع MSE نیز برای تابع ضرر استفاده می‌کنیم زیرا MSE برای بررسی میزان نزدیکی تخمین‌ها به مقادیر واقعی استفاده می‌شود. با پایین آمدن MSE، پیش‌بینی می‌شود که به واقعی نزدیک‌تر شده‌ایم.

ب) از تابع فعالسازی sigmoid و تابع ضرر MSE_with_dontcare یا همان mean squered error که خودش نوشته است، استفاده شده است.

۳. Madaline ها از چند Adaline تشکیل شده اند، بنابراین به جای یک خروجی، ماتریس‌های وزنی و خروجی‌های متعدد داریم. در پایان، طبقه‌بندی‌های خطی با یکدیگر AND می‌شوند، بنابراین این فرصت را به ما می‌دهد که مناطق محدب را با خطوط ایجاد کنیم. از آنجایی که می توان از Madaline برای طبقه بندی داده‌ها در چندضلعی‌های محدب استفاده کرد، تصویر آ را می‌توان با استفاده از دو Adaline به کلاس‌های مناسب خود طبقه بندی کرد. از آنجایی که تنها یک ناحیه محدب وجود دارد، هیچ لایه پنهانی لازم نیست. اما در تصویر ب، به دلیل وضعیتی که در زیر نشان داده شده است، هیچ شکل محدبی وجود ندارد که بتواند از هم جدا شود.



۴.

الف) تفاوت اصلی بین این دو، این است که یک Perceptron آن پاسخ دودویی (مانند یک نتیجه طبقه بندی) را می‌گیرد و یک خطا را محاسبه می‌کند که برای آپدیت وزن ها استفاده می شود، در حالی که یک Adaline از یک مقدار پاسخ پیوسته برای آپدیت وزن‌ها استفاده می کند. پرسپترون

وزن ها را با محاسبه تفاوت بین مقادیر کلاس مورد انتظار و پیش بینی شده به روز می کند. به عبارت دیگر، پرسپترون همیشه ۱+ یا ۱- (مقادیر پیش بینی شده) را با ۱+ یا ۱- (مقادیر مورد انتظار) مقایسه می‌کند. پرسپترون تنها زمانی یاد می‌گیرد که خطا ایجاد شود. در مقابل، Adaline تفاوت بین مقدار کلاس مورد انتظار y (۱+ یا ۱-) و مقدار خروجی پیوسته y را از تابع خطی محاسبه می‌کند که می‌تواند هر عدد واقعی باشد. این بسیار مهم است زیرا به این معنی است که Adaline می‌تواند حتی زمانی که هیچ اشتباهی در طبقه بندی انجام نشده است یاد بگیرد. از آنجایی که Adaline همیشه و پرسپترون فقط پس از خطاها یاد می گیرد، Adaline راه حلی سریع‌تر از پرسپترون برای همان مشکل پیدا می‌کند. این واقعیت که Adaline این کار را انجام می‌دهد، اجازه می‌دهد آپدیت‌های آن، قبل از اینکه آستانه تعیین شود، بیشتر شبیه خطای واقعی باشند، که به نوبه‌ی خود به مدل اجازه می‌دهد تا سریع‌تر همگرا شود. (لینک)

پس قابلیت تعمیم‌پذیری Adaline از Perceptron بیشتر است. همچنین Madaline ترکیب AND چند Adaline است که بدیهی‌ست نسبت به دو مورد قبلی تعمیم‌پذیری بیشتری دارد.

مزیت MLP نیز نسبت به Perceptron و Adaline کلاسیک این است که با اتصال نورون‌های مصنوعی در این شبکه از طریق توابع فعال‌سازی غیرخطی، می‌توانیم مرزهای تصمیم‌گیری پیچیده و غیرخطی ایجاد کنیم که به ما امکان می‌دهد با مشکلاتی که در آن کلاس‌های مختلف قابل جداسازی خطی نیستند، مقابله کنیم. پس قابلیت تعمیم‌پذیری آن از تمام موارد دیگر ذکرشده بیشتر است. (لینک)

پس به ترتیب ابتدا MLP سپس Madaline، Adaline و در آخر Perceptron است.

ب) Overfitting یک خطای مدل‌سازی در آمار است و زمانی رخ می‌دهد که یک تابع خیلی نزدیک به مجموعه محدودی از نقاط داده fit شده باشد. در نتیجه، این مدل تنها در ارجاع به مجموعه داده های اولیه خود مفید است و نه به هیچ مجموعه داده دیگری. هنگامی که الگوریتم های یادگیری ماشین ساخته می‌شوند، از مجموعه داده‌های نمونه برای آموزش مدل استفاده می کنند. با این حال، زمانی که مدل برای مدت طولانی روی داده‌های نمونه آموزش می‌یابد یا زمانی که مدل بسیار پیچیده است، می‌تواند شروع به یادگیری «نویز» یا اطلاعات نامربوط در مجموعه

داده‌ها کند. وقتی مدل نویز را به خاطر می‌سپارد و خیلی نزدیک به مجموعه آموزشی منطبق می‌شود، مدل overfit می‌شود و نمی‌تواند به خوبی روی داده‌های جدید تعمیم یابد. اگر یک مدل نتواند به خوبی روی داده‌های جدید تعمیم یابد، آنگاه نمی‌تواند وظایف طبقه‌بندی یا پیش‌بینی را که برای آن در نظر گرفته شده است، انجام دهد. (لینک)

ج) از راه‌های جلوگیری از آن می‌توان به موارد زیر اشاره کرد:

۱) تقسیم و شکستن داده‌های آموزشی به تعدادی fold و اجرای مدل برای هرکدام از آن‌ها. خطای کل در این روش برابر است با میانگین خطاهای هر بخش.

۲) افزودن داده به مجموعه‌ی داده‌های آموزشی به همراه قوی‌تر کردن مدل

۳) داده‌افزایی یا data augmentation: به روش‌های گوناگون انجام می‌شود تا داده‌ها هم زیاد شود و هم متفاوت از هم دیده شوند.

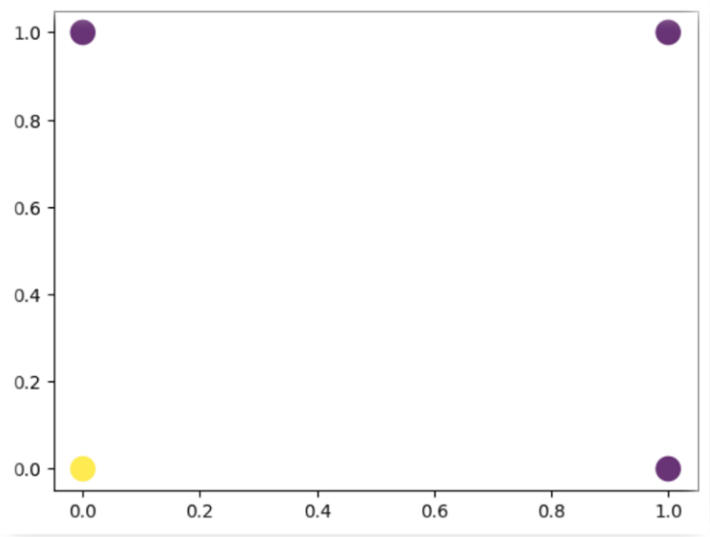
۴) ساده‌سازی داده‌ها

۵) افزودن نویز به دیتای ورودی

۵) لینک۱ ، لینک۲

ابتدا داده‌ها را پلات می‌کنیم:

```
x = np.array([[0,0], [1,1], [0,1], [1,0]])
y = np.array([1, 0, 0, 0])
plt.scatter(x[:,0], x[:,1], s=200, lw=0, alpha=.8, c=y);
```



سپس الگوریتم را پیاده‌سازی می‌کنیم تا وزن‌ها آپدیت شوند. در انتها با استفاده از تابع پلات دیتایی که تعریف کرده‌ایم خط نهایی را پلات می‌کنیم.

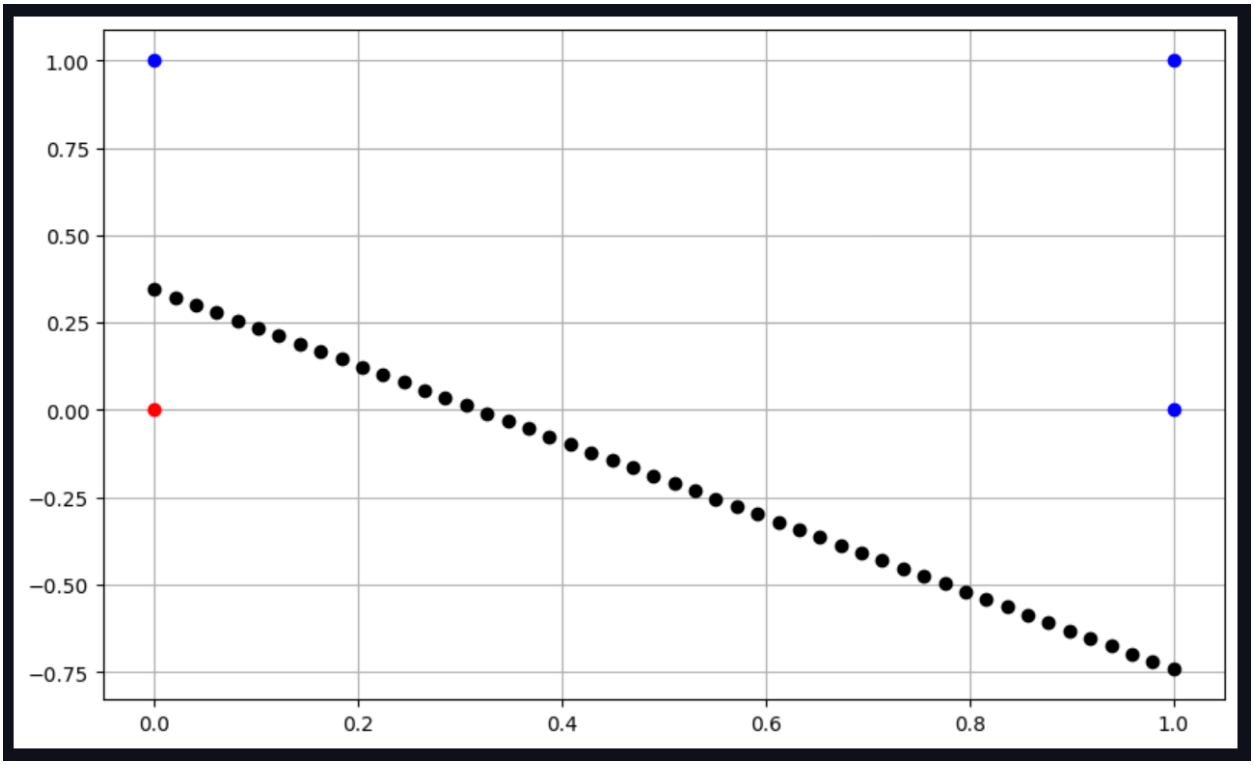
الگوریتم به این صورت است که ابتدا وزن‌ها را به صورت رندوم مقداردهی اولیه می‌کنیم. مشتق‌ها را نیز یک می‌گذاریم. سپس داده‌های ورودی و خروجی آموزش را تعریف می‌کنیم. برای فاز آموزش تعداد ایپاک ۱۰۰ و نرخ آموزش را برابر ۰.۲ قرار می‌دهیم. سپس به تعداد ایپاک‌ها الگوریتم گرادیان کاهشی را اعمال می‌کنیم و هربار وزن‌ها را آپدیت می‌کنیم. پس از آخرین ایپاک، وزن‌ها را چاپ کرده و خط جداکننده‌ی نهایی را پلات می‌کنیم.

```
alpha = 0.2
epochs = 100

# INITIALIZING WEIGHTS
w0 = np.random.randn()
w1 = np.random.randn()
```

```
w2 = np.random.randn()

print("Initial weights - ")
print("w0 = ",w0,", w1 = ", w1,", w2 = ", w2)

del_w0 = 1
del_w1 = 1
del_w2 = 1

#SPECIFYING TRAINING DATA
train_data_temp = [[1, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1]]
train_data = np.asarray(train_data_temp)

op_f = [1, 0, 0, 0] # NOR
op = np.asarray(op_f)

#TRAINING PROCESS
for i in range(epochs):
    j = 0
    for x in train_data:
        res = w0*x[0] + w1*x[1] + w2*x[2]

        if (res >= 0):
            act = 1
        else:
            act = 0

        # act = 1/(1+math.exp(-x))

        err = (op[j] - act)

        del_w0 = alpha*x[0]*err
        del_w1 = alpha*x[1]*err
        del_w2 = alpha*x[2]*err

        w0 = w0 + del_w0
        w1 = w1 + del_w1
        w2 = w2 + del_w2

        j = j + 1
print("\nFinal weights - ")
print("w0 = ",w0,", w1 = ",w1,", w2 = ",w2)


w = [w0, w1, w2]
```

```
x = np.array([[0,0], [1,1], [0,1], [1,0]])
y = np.array([1, 0, 0, 0])
plot_data(x, y, w)
```

```
Initial weights -
w0 =  0.874385168929969 , w1 =  1.165904313772931 , w2 =  0.9845235096699916

Final weights -
w0 =  0.07438516892996888 , w1 =  -0.2340956862270689 , w2 =  -0.21547649033000837
```



می‌بینیم که خط نهایی به خوبی دو کلاس را از هم تفکیک کرده است.

۶) در ابتدا تعداد کلاس‌ها و سایز عکس‌های ورودی را تعریف می‌کنیم. سپس دیتای آموزشی و تست را لود می‌کنیم و تقسیم بر ۲۵۵ می‌کنیم تا به رنج ۰ تا یک اسکیل شوند.

```
num_classes = 10
input_shape = (28, 28, 1)

# Load the data and split it between train and test sets
```

```
(x_train, y_train), (x_test, y_test) = datasets.mnist.load_data()

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

سپس لیبل داده‌های تست و ترین را to_categorical می‌کنیم تا به صورت یک بردار به تعداد کلاس‌های ما در بیایند و مقدار هر خانه از بردار احتمال مربوط بودن نمونه به آن کلاس می‌شود که بین صفر تا یک است.

```
y_train = to_categorical(y_train, num_classes)
y_test = to_categorical(y_test, num_classes)
```

در مرحله‌ی بعد مدل خود را به صورت sequential تعریف می‌کنیم. در لایه‌های اولیه و میانی از تابع فعالسازی relu و در لایه‌ی آخر از تابع فعالسازی softmax استفاده می‌کنیم. زیرا مسئله‌ی ما کلاس‌بندی چندکلاسه است. لایه‌ها نیز به صورت کاملا متصل هستند. برای بهبود مدل خود از dropout نیز استفاده کرده‌ایم.

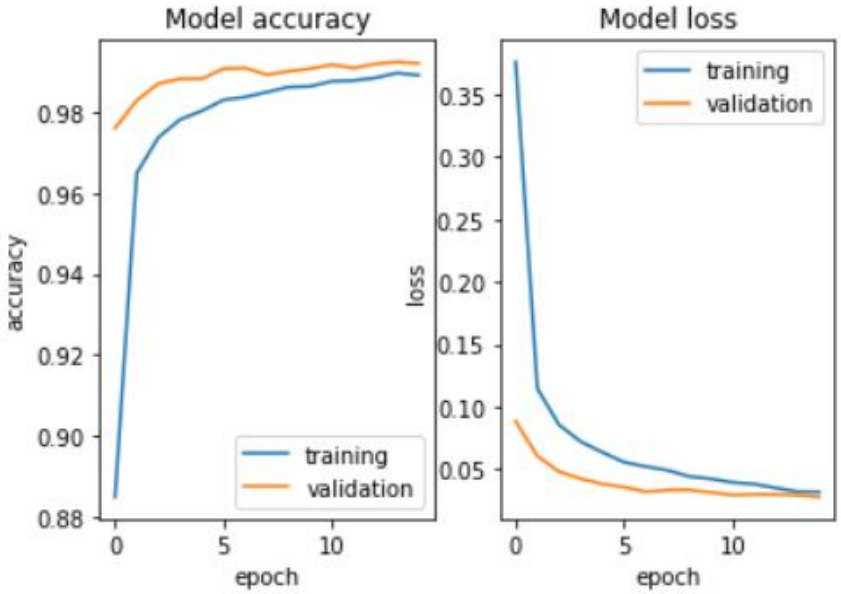
```
model = Sequential(
    [
        Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation="softmax"),
    ]
)

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2D  (None, 13, 13, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 5, 5, 64)         0
 2D)

 flatten (Flatten)           (None, 1600)              0

 dropout (Dropout)           (None, 1600)              0

 dense (Dense)               (None, 10)                16010

=================================================================
Total params: 34,826
Trainable params: 34,826
Non-trainable params: 0
_________________________________________________________________
```

مدل خود را کامپایل و روی داده‌های آموزشی اجرا می‌کنیم. بدلیل نوع مسئله از تابع ضرر categorical_crossentropy و از تابع بهینه‌ساز adam استفاده می‌کنیم. سایز batchها را ۱۲۸ و تعداد ایپاک‌ها را ۱۵ تعریف می‌کنیم. همچنین حدود ۱۰٪ از دیتاهای آموزشی را برای validation جدا می‌کنیم. سپس نمودار دقت و خطا را رسم می‌کنیم.

```
batch_size = 128
epochs = 15

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

history  =  model.fit(x_train,  y_train,  batch_size=batch_size,  epochs=epochs,
validation_split=0.1)
fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.plot(history.history['accuracy'])
ax1.plot(history.history['val_accuracy'])
ax1.set_title('Model accuracy')
```

```
ax1.set_ylabel('accuracy')
ax1.set_xlabel('epoch')
ax1.legend(['training', 'validation'], loc='lower right')

ax2.plot(history.history['loss'])
ax2.plot(history.history['val_loss'])
ax2.set_title('Model loss')
ax2.set_ylabel('loss')
ax2.set_xlabel('epoch')
ax2.legend(['training', 'validation'], loc='upper right')
```



همانطور که می‌بینیم به دقت بالای ۹۸٪ برای داده‌های آموزشی و validation رسیده و همچنین خطای کمتر از ۰.۳٪ را داریم.

برای حل این سوال از این لینک و همچنین تمرین درس یادگیری عمیق استفاده شد.