به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

# درس شبکه‌های عصبی و یادگیری عمیق

## تمرین Extra

| | |
|---|---|
| نام و نام خانوادگی | آرمان فروزش – مهسا ندافی قهنویه |
| شماره دانشجویی | 810100490 - 111946 |
| تاریخ ارسال گزارش | 1401.۰9.28 |

# فهرست

# شکل‌ها

# جدول‌ها

# پاسخ 1. تشخیص تقلب (fraud detection) با استفاده از شبکه عمیق

## ۱-۱. بزرگترین چالش ها در توسعه مدل‌های تشخیص تقلب چیست؟ این مقاله برای حل این چالش‌ها از چه متدهایی استفاده کرده‌است؟

با توجه به اینکه امروزه استفاده از کارت‌های اعتباری برای پرداخت بسیار پرکاربرد شده‌اند تشخیص تقلب در این فرایندهای پرداختی امری حیاتی‌است. استفاده از روش‌های کلاسیک برای تشخیص تقلب، با توجه به اینکه حجم داده‌ها بسیار زیاد است، ناکارآمد است پس امروزه برای تشخیص تقلب از شبکه‌های عصبی عمیق استفاده فراوانی می‌کنند. یکی از مهمترین چالش‌هایی که در استفاده از یادگیری عمیق برای تشخیص تقلب با آن روبرو هستیم، متوازن نبودن دیتاست های مربوط به این کار است چراکه هموتره بخش کوچکی از داده ها مربوط به داده های تقلب است (کمتر از یک درصد) و با استفاده از این داده ها شبکه در پایان با اینکه دقت بالای 99درصد دارد، توانایی تشخیص تقلب را ندارد.

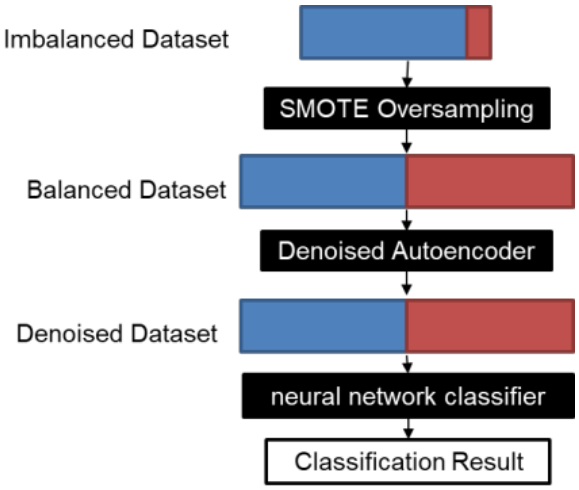
مقاله مربوطه برای حل این مشکل روش oversampling برای متوازن(balance) کردن دیتاست استفاده می‌کند. از طرف دیگر، از یک شبکه autoencoder برای نویزگیری داده ها نیز استفاده می‌کند.

## ۱-2. در مورد معماری شبکه ارائه شده در مقاله به شکل مختصر توضیح دهید.

همانگونه که در بخش قبل اشاره شد این مقاله ایتدا با استفاده از روش‌های oversampling دیتاست مورد استفاده را balance می‌کند و سپس از یک شبکه autoencoder برای نویزگیری داده ها استفاده میکند و در نهایت از یک شبکه عصبی برای طبقه‌بندی داده‌ها به دو دسته normal و fraud استفاده می‌کند.

دیتاست مورد استفاده در این مسئله، دیتاست Credit Card Fraud Detection-Kaggle می‌باشد. این دیتاست شامل داده مربوط به جزئیات تراکنش‌های مالی میباشد که در آن کمتر از 0.5درصد داده ها مربوط به دسته fraud هستند. در این داده ها ویژگی های اصلی V1,V2V…V28 هستند که نرمالیزه شده‌اند و در کنار آنها ویژگی های Time و Amount نیز قرار دارند که در این مسئله ویژگی Time برای ما اهمیتی ندارد و از آن صرف نظر میکنیم و داد های مربوط به Amount نبز به نرمالیزه شدن دارند. ویژگی مهم دیگری که در دستاست موجود است Class میباشد که برای fraud مقدارش 1 و در بقیه

حالات 1 می‌باشد که ما از این ویژگی به عنوان خروجی شبکه استفاده می‌کنیم. روند کلی که در این مقاله انجام می‌شود در شکل 1.1 قابل مشاهده می‌باشد.




**شکل 1.1** روند کلی صورت گرفته در مقاله

در این مقاله برای انجام oversampling از روش SMOTE استفاده شده اس که در آن برای کلاسی تعداد کمتری  داده دارد، داده ایجاد میکند تا دیتاست balance شود.

معماری مورد استفاده برای بخش نویزگیر شبکه در جدول1.1 قابل مشاهده است. با توجه یه اینکه این شبکه autoencoder است پس تعداد ورودی ها و خروجی های شبکه با هم برابر هستند و با توجه به دیتاست مورد استفاده، ویژگی‌های مورد استفاده ما به عنوان ورودی 29تا هستند پس سایز ورودی و خروجی شبکه نویزگیر نیز 29 میباشد. این شبکه شامل 6لایه fully connected میباشد که در نهایت خروجی نویزگیری شده را به ما میدهد. با توجه به اطلاعات مقاله، loss function مورد استفاده برای این شبکه نیز mse میباشد. با توجه به اینکه در مقاله راجع به توابع فعالسازی لایه ها صحبتی نشده، در شبیه‌سازی آن از تابع relu استفاده می‌کنیم.

معماری مورد استفاده برای شبکه Classifier نیز در جدول2.1 قابل مشاهده است. این شبکه نیز یک شبکه fully connected شامل 5 لایه مخفی می‌باشد که در لایه آخر با توجه به خروجی ما که باینریست، سایز لایه آخر 2 میباشد و در لایه آخر از تابع softmax و در بقیه لایه‌ها از تابع relu استفاده شده‌است. Loss fuction این شبکه نیز binary cross entropy می‌باشد.

**جدول 1.1** مدل شبکه استفاده شده برای **denoised autoencoder**

| Dataset with noise (29) |
|---|
| Fully-Connected-Layer (22) |
| Fully-Connected-Layer (15) |
| Fully-Connected-Layer (10) |
| Fully-Connected-Layer (15) |
| Fully-Connected-Layer (22) |
| Fully-Connected-Layer (29) |
| Square Loss Function |

**جدول 2.1** مدل شبکه استفاده شده برای **classifier**

| Dataset with noise (29) |
|---|
| Fully-Connected-Layer (22) |
| Fully-Connected-Layer (15) |
| Fully-Connected-Layer (10) |
| Fully-Connected-Layer (5) |
| Fully-Connected-Layer (2) |
| SoftMax Cross Entropy Loss Function |

## ۱-3. انواع روش‌های Resampling موجود برای balance کردن دیتاست را نامبرده و مزایا و معایب هر کدام را توضیح دهید.

روش‌های resampling برای balance کردن دیتاست های نامتوازن استفاده می‌شوند و در کل میتوان آنها را به سه دسته oversampling، undersampling و hybrid تقسیم بندی کرد. در روش undersampling داده های مربوط به کلاسی که اکثریت را تشکیل میدهد را بصورت تصادفی پاک میکنیم تا اینکه تعداد داده های کلاس ها باهم برابر شوند. اما در روش oversampling با استفاده از روش‌هایی همچون SMOTE و ADASYN داده های مربوط به کلاس اقلیت را با سنتز کردن به تعدا دکلاس دیگر میرسانیم. در نهایت روش hybrid ترکیبی از دو رش قبلی است.

روش undersampling اصولا زمانی که اختلاف دو کلاس اقلیت و اکثریت کم باشد مورد استفاده قرار میگیرد ام در مسائلی که اختلاف این دو کلاس زیاد باشد استفاده از این روش باعث از بین رفتن بخش زیادی از داد ها میشود و باعث میشود که دقت شبکه به اندازه مناسبی نرسد.

اما استفاده از oversampling در دیتاست هایی که اختلاف دو کلاس زیاد است کابرد و عملکرد بهتری دارد اما از طرفی نیز با توجه به اینکه داد های را زیاد میکند، باعث افزایش زمان یادگیری شبکه می‌شود. نکته مهمی که در استفاده از oversampling باید به آن دقت کرد این است که داده های test و evaluation را باید قبل از انجام oversampling از داده های train جدا کنیم و SMOTE را فقط روی داده های train انجام دهیم، چرا که هنگاه انجام oversampling داده های ایجاد شده بسیار شبیه داد های موجود است و باعث میشود که داده های Test و evaluation را شبکه هنگامtrain ببیند و ارزیابی ما دقیق نشود.

در این مسئله ما از روش SMOTE oversampling استفاده می‌کنیم.

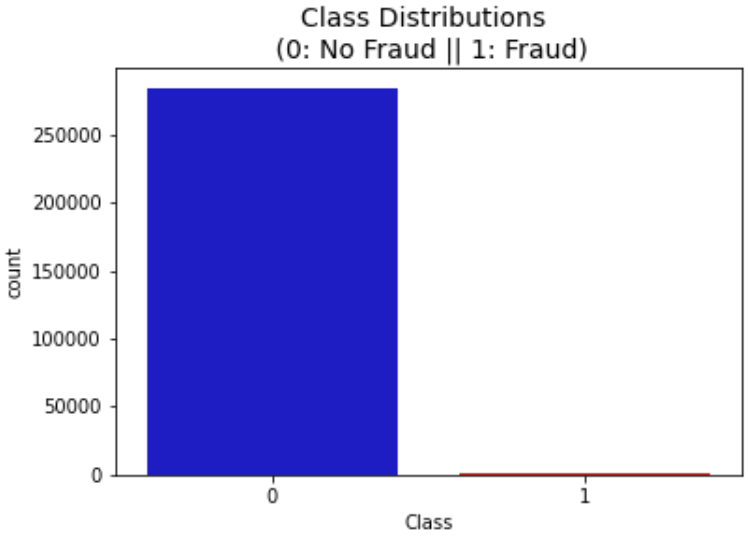
## ۱-4. مدل ارائه شده را پیاده‌سازی کرده و با استفاده از این داده آموزش دهید.

در ابتدا داده ذکر شده را با استفاده از دستور شکل زیر دانلود و در درایو ذخیره میکنیم.

```
!wget https://github.com/nsethi31/Kaggle-Data-Credit-Card-Fraud-Detection/raw/master/creditcard.csv

--2022-12-10 18:06:42--  https://github.com/nsethi31/Kaggle-Data-Credit-Card-Fraud-Detection/raw/master/creditcard.csv
Resolving github.com (github.com)... 140.82.113.4
Connecting to github.com (github.com)|140.82.113.4|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://raw.githubusercontent.com/nsethi31/Kaggle-Data-Credit-Card-Fraud-Detection/master/creditcard.csv [following]
--2022-12-10 18:06:42--  https://raw.githubusercontent.com/nsethi31/Kaggle-Data-Credit-Card-Fraud-Detection/master/creditcard.csv
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.110.133, 185.199.108.133, 185.199.111.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.110.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 102634230 (98M) [text/plain]
Saving to: ‘creditcard.csv’

creditcard.csv      100%[===================>]  97.88M  51.2MB/s    in 1.9s

2022-12-10 18:06:45 (51.2 MB/s) - ‘creditcard.csv’ saved [102634230/102634230]
```

**شکل 2.1** دانلود داده ها

سپس داده ها را با استفاده از کتابخانه pandas لود میکنیم و مشاهده میشود که داده ها در دو کلاس 0 و 1 هستند که صفر مربوط به داده های not fraud و 1 مربوط به داده های fraud میباشد. سپس برای بررسی بیشتر تعداد داده های موجود در هر کلاس را محاسبه میکنیم و میبینیم که این دیتاست کاملا imbalance می‌باشد.




**شکل 3.1** بررسی تعدا داده های مربوط به هر کلاس

با دقت بیشتر در دیتاست و همانگونه که در مقاله نیز ذکر شده بود داده های مربوط به ستون های v1,…v28 نرمالیزه شده هستند و تنها داده های مربوط به ویژگی های Time و Amount نیاز به نرمالیزه شدن دارند و با توجه به اینکه در مقاله ذکر شده بود در این مسئله ما از داده های Time استفاده نمیکنیم، پس ستون مربوط به Time را از دیتاست حذف و در ادامه داده های Amount را نیز با استفاده از MinMaxScaler از sklearn نرمالیزه میکنیم و مقادیر آن بین 0 و 1 قرار می‌گیرند.

| | scaled_amount | V1 | V2 | V3 | V4 | V5 | V6 | V7 | V8 | V9 | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.005824 | -1.359807 | -0.072781 | 2.536347 | 1.378155 | -0.338321 | 0.462388 | 0.239599 | 0.098698 | 0.363787 | ... |
| 1 | 0.000105 | 1.191857 | 0.266151 | 0.166480 | 0.448154 | 0.060018 | -0.082361 | -0.078803 | 0.085102 | -0.255425 | ... |
| 2 | 0.014739 | -1.358354 | -1.340163 | 1.773209 | 0.379780 | -0.503198 | 1.800499 | 0.791461 | 0.247676 | -1.514654 | ... |
| 3 | 0.004807 | -0.966272 | -0.185226 | 1.792993 | -0.863291 | -0.010309 | 1.247203 | 0.237609 | 0.377436 | -1.387024 | ... |
| 4 | 0.002724 | -1.158233 | 0.877737 | 1.548718 | 0.403034 | -0.407193 | 0.095921 | 0.592941 | -0.270533 | 0.817739 | ... |

5 rows × 30 columns

**شکل 4.1** بررسی کلی دیتاست بعد از نرمالیزه کردن

همانگونه که در شکل 4.1 نیز داده میشود، در نهایت 30 ویژگی داریم که ویژگی Class را به عنوان خروجی(y) و بقیه 29 ویژگی را به عنوان ورودی(x) در نظر میگیریم.

پیش از آن که بخواهیم با استفاده از روش SMOTE داده‌ها را balance کنیم به دلیلی که در بخش قبل نیز بیان شد باید داده های train،test و val را جدا کرده و SMOTE را تنها برروی داده‌های train اعمال کنیم. با توجه به اینکه در مقاله ضریبی برای انتخاب train test مشخص نکرده بود، ما 0.2 داده ها را به test و از 0.8 باقیمانده نیز 0.2را به val و بقیه داده ها را به train اختصاص می‌دهیم.

```
Shape of X: (284807, 29)
Shape of y: (284807,)
```

```
Number transactions X_train dataset:  (182276, 29)
Number transactions y_train dataset:  (182276,)
Number transactions X_test dataset:  (56962, 29)
Number transactions y_test dataset:  (56962,)
Number transactions X_val dataset:  (45569, 29)
Number transactions y_val dataset:  (45569,)
```

**شکل 5.1** ابعاد داده های **train, test, val**

برای oversampling داده ها در این مسئله از کتابخانه imblearn استفاده شده است که در آن میتوان با استفاده از دستور ()SMOTE و resample کردن داده ها، oversampling را انجام دادۀ، از طرفی نیز میتوان در تابع ذکر شده با تنظیم متغیر sampling_strategy و با قرار دادن عدید بین 0 و 1 در این متغیر، میزان سنتز کردن داده های اقلیت را تعیین کرد، به اینصورت که داده های اقلیت به تعداد ضریب وارد شده در تعدا داده های اکثریت افزایش خواهتئ یافت، برای مثال با قرار دادن 0.5 ، تعدا داده های اقلیت به اندازه نصف داده های اکثریت افزایش می‌یابند.

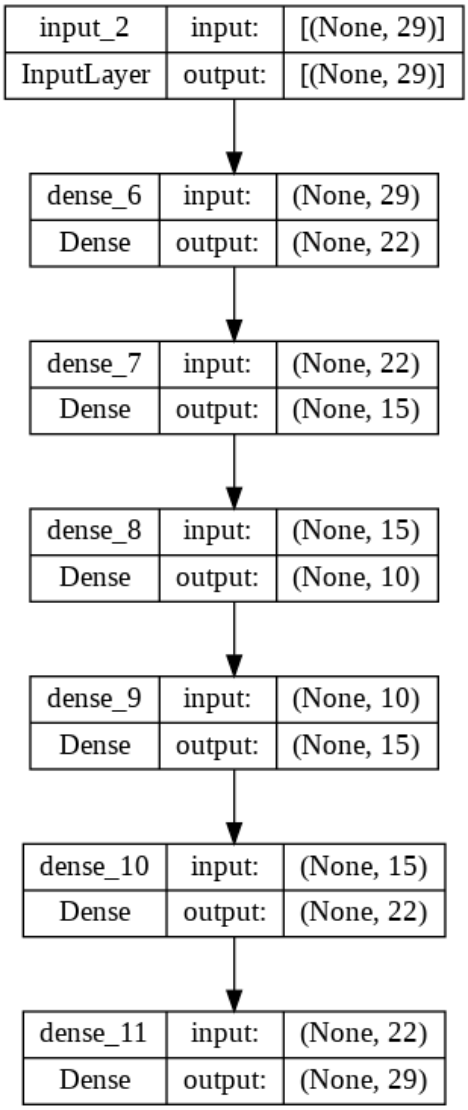
```
from imblearn.over_sampling import SMOTE

print("Before OverSampling, counts of label '1': {}".format(sum(y_train==1)))
print("Before OverSampling, counts of label '0': {} \n".format(sum(y_train==0)))

sm = SMOTE(sampling_strategy=0.5)
X_train_res, y_train_res = sm.fit_resample(X_train, y_train.ravel())

print('After OverSampling, the shape of train_X: {}'.format(X_train_res.shape))
print('After OverSampling, the shape of train_y: {} \n'.format(y_train_res.shape))

print("After OverSampling, counts of label '1': {}".format(sum(y_train_res==1)))
print("After OverSampling, counts of label '0': {}".format(sum(y_train_res==0)))
```

```
Before OverSampling, counts of label '1': 306
Before OverSampling, counts of label '0': 181970

After OverSampling, the shape of train_X: (272955, 29)
After OverSampling, the shape of train_y: (272955,)

After OverSampling, counts of label '1': 90985
After OverSampling, counts of label '0': 181970
```

**شکل 6.1 oversampling**

همانگونه که در شکل 6.1 مشاهده میکنید، تعداد داده های مربوط به کلاس fraud قبل از oversampling تنها 306 عدد بود که پس از oversampling به 90985 عدد افزایش پیدا کرد.

در ادامه این مقاله از یک شبکه autoencoder برای denoise کردن داده های ورودی استفاده میکند. معماری این شبکه را در شکل 7.1 مشاهده میکنید. برای آموزش این شبکه باید یکسری داده آغشته به نویز گوسی را به عنوان ورودی به شبکه بدهیم و در خروجی، داده های بدون نویز را دریافت کنیم. پس برای ایجا یک دیتاست نویزی از داده های تست استفاده میکنیم و با اضافه کردن نویز با مینگین صفر و واریانس 0.1 به آن یکی دیتاست نویزی جدید ایجاد میکنیم. از دیتاست نویزی ایجا شده به عنوان ورودی autoencoder و از داده های بدون نویز به عنوان خروجی سبکه استفاده می‌کنیم.




**شکل 7.1** مدل شبکه **denoising autoencoder**

در این شبکه برای همه لایه بجز لایه آخر از تابع فعالسازی ReLU و در لایه آخر هم نیز از تابع linear استفاده شده‌است. تابع loss function مورد استفاده در این شبکه نیز با توجه به انتخاب مقاله mse میباشد. در این شبکه از adam به عنوان optimizer نیز استفاده شده است. در نهایت پس از 1000ایپاک و batch_size=1024 ، acc_val = 81.36 و loss_val=0.1356 رسید. برای اینکه بهترین وزن های شبکه با توجه به loss_val ذخیره شئند در فرایند آموزش از callback استفاده میکنیم. برای ذخیره کردن بهترین مدل نیز از دستور ModelCheckpoint از کتابخانه keras بهره میبریم و فایل بهترین مدل را در best_model_AutoEncoder.h5 ذخیره میکنیم که در نهایت بعدا میتوانیم با لود کردن آن، از شبکه‌ای که بهترین وزن ها را دارد استفاده کنیم.

```
import h5py
from keras.callbacks import EarlyStopping, ModelCheckpoint

es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=50)
mc = ModelCheckpoint('best_model_AutoEncoder.h5', monitor='val_loss', mode='min', verbose=1, save_best_only=True)
```

```
results_Denoising = Denoising_autoencoder.fit(noise_signal_train, clean_signal_train,
                            batch_size=1024,
                            epochs=1000,
                            shuffle=True,
                            validation_data = (noise_signal_val, clean_signal_val),
                            callbacks=[es, mc])
```

**شکل 8.1** فرایند آموزش شبکه **denoised autoencoder**

در نهایت برای اینکه این شبکه را ببینیم تعداد از داده های val آغشته به نویز را به شبکه داده و خروجی آنها را با مقادیر بدون نویز مقایسه میکنیم ( خروجی شبکه را از مقادیر واقعی کم مینیم) و میبینیم که عملکرد شبکه نسبتا مناسب است.

| | scaled_amount | V1 | V2 | V3 | V4 | V5 | \ |
|---|---|---|---|---|---|---|---|
| 18456 | 0.001225 | -0.009171 | -0.077504 | -0.044878 | -0.070162 | -0.035370 | |
| 62262 | 0.000359 | 0.045271 | 0.318427 | 0.042105 | 0.135709 | 0.132202 | |
| 150576 | -0.000123 | 0.083578 | -0.062772 | -0.034202 | -0.029849 | 0.003076 | |
| 152002 | -0.000699 | -0.071464 | -0.124730 | 0.105225 | -0.262551 | -0.082694 | |
| 197934 | 0.001621 | -0.201798 | -0.152492 | -0.301745 | -0.123606 | 0.206779 | |
| 238493 | -0.000541 | 0.233272 | -0.041442 | 0.397366 | -0.208580 | 0.170981 | |
| 123170 | 0.001318 | 0.175625 | 0.026160 | -0.085856 | -0.086044 | -0.267816 | |
| 11910 | -0.000431 | -0.135725 | -0.056358 | -0.103302 | 0.027212 | 0.144614 | |
| 266164 | 0.001150 | 0.470891 | -0.461665 | -1.006749 | -0.073835 | 0.432027 | |
| 64125 | 0.001219 | -0.210002 | 0.105827 | 0.308770 | 0.017126 | 0.335073 | |

**شکل 9.1** ارزیابی شبکه **denoised autoencoder**

با توجه به روند موجود در شکل 1.1 ورودی پس از عبور از denoised autoencoder وارد شبکه Classifier میشود. معماری این شبکه در جدول 2.1 مشاهده میشود که در این شبکه در همه لایه ها به جز لایه آخر از تابع ReLU و در لایه آخر از تابع SoftMax استفاده شده است. Loss function این شبکه نیز با توجه به اینکه کلاس های خروجی دوتا هستند، Binary Cross Entropy میباشد. در نهایت شبکه نهایی بصورت شکل 10.1 می‌شود.

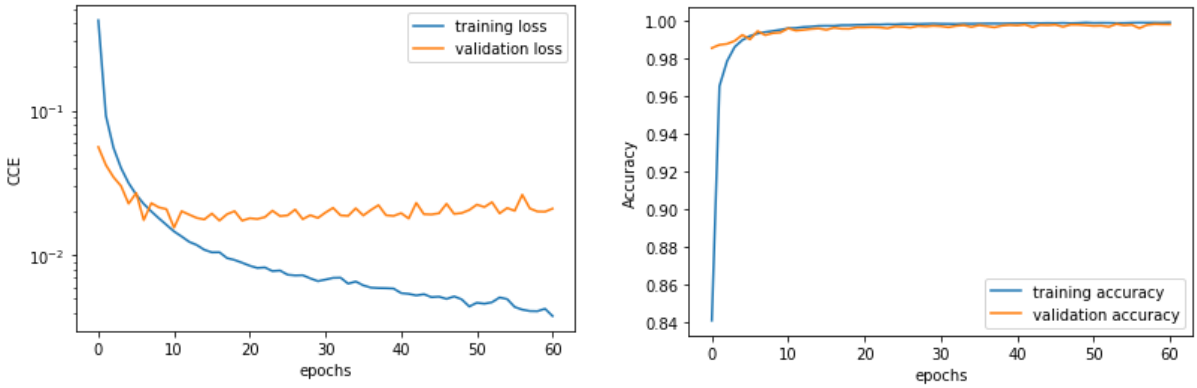
```
Model: "model_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 29)]              0

 model (Functional)          (None, 29)                2349

 dense_11 (Dense)            (None, 22)                660

 dense_12 (Dense)            (None, 15)                345

 dense_13 (Dense)            (None, 10)                160

 dense_14 (Dense)            (None, 5)                 55

 dense_15 (Dense)            (None, 2)                 12

=================================================================
Total params: 3,581
Trainable params: 1,232
Non-trainable params: 2,349
_________________________________________________________________
```

**شکل 10.1** مدل نهایی شبکه

همانگونه که مشاهده میشود در این شبکه لایه اول همان شبکه denoised autoencoder است که برای استفاده از آن ابتدا با استفاده از دستور load_model فایل best_model_AutoEncoder.h5 را که بهترین وزن های شبکه autoencoder هستند را لود کرده و به عنوان لایا اول شبکه در نظر میگیریم و ورودی شبکه به این لایه داده شده و سپس خروجی آن وارد شبکه Classsifier میشود. دقت شود که باید پس از لود کردن مدل autoencoder وزنهای تمام لایه های آن را با دستور layer.trainable = False، freeze کنیم تا در فرایند آموزش شبکه تغییر نکنند. در نهایت شبکه را با استفاده از adam optimizer آموزش میدهیم و برای ذخیره کردن بهترین وزن ها با توجه به loss_val از callback استفاده میکنیم و بهترین مدل را در best_model_withSMOTE.h5 ذخیره می‌کنیم.

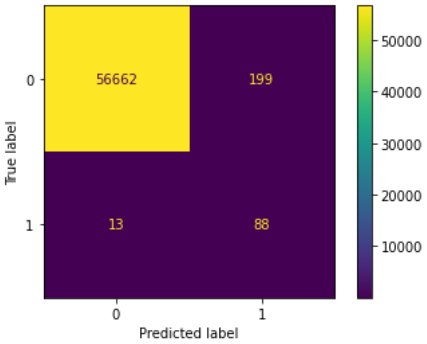
```
results_Classifier = Classifier.fit(X_train_res, y_train_res,
                                    batch_size=1024,
                                    epochs=1000,
                                    validation_data = (X_val, y_val),
                                    shuffle=True,
                                    callbacks=[es, mc])
```

```
Epoch 61/1000
258/267 [===========================>..] - ETA: 0s - loss: 0.0038 - acc: 0.9991
Epoch 61: val_loss did not improve from 0.01553
267/267 [==============================] - 1s 5ms/step - loss: 0.0038 - acc: 0.9991 - val_loss: 0.0210 - val_acc: 0.9981
Epoch 61: early stopping
```




**شکل 11.1** ارزیابی **training** شبکه با **oversampling**

**۱-5. نمودار Heatmap را برای Confusion matrix پیشبینی مدل بر روی داده‌های تست رسم کنید و مقادیر Recall، precision، accuracy و f1score را در گزارش ذکر کنید.**



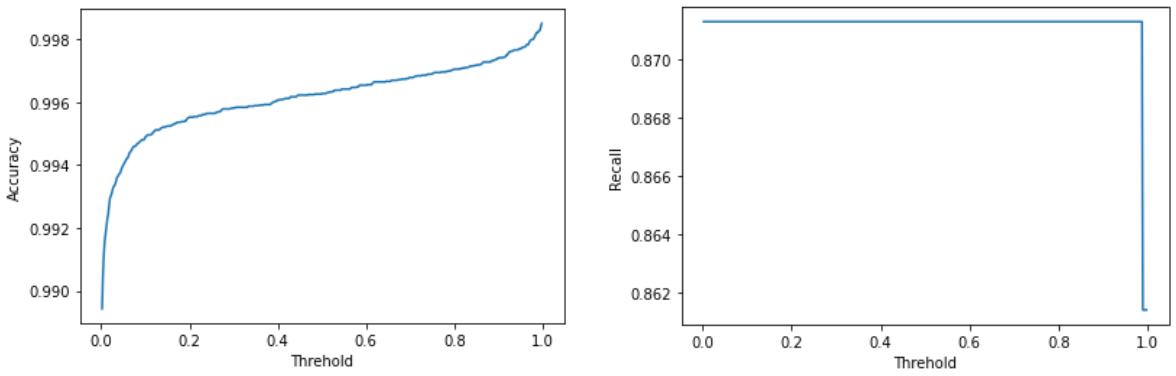

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 1.00 | 1.00 | 1.00 | 56861 |
| 1 | 0.31 | 0.87 | 0.45 | 101 |
| accuracy | | | 1.00 | 56962 |
| macro avg | 0.65 | 0.93 | 0.73 | 56962 |
| weighted avg | 1.00 | 1.00 | 1.00 | 56962 |

**شکل 12.1** ارزیابی مدل با **oversampling** برروی داده های تست

همانگونه که در شکل 12.1 مشاهده میشود دقت نهایی مدل 100 میباشد اما اگر به دقت کلاس ها توجه کنیم میبینیم که مدل در تشخیص کلاس fraud به انداز تشخیص کلاس no fraud بخوبی عمل نمیکند که دلیل اصلی آن imbalance بودن دیتاست که حتی با وجود oversampling هم دیتاهای ایجاد شده بسیار شبیه دیتاهای موجود در داده های train هستند و شبکه در مقابل داده های اقلیتی که ندیده است ضعف دارد. پس برای اینکه ارزیابی بهتری داشته باشیم باید به مقادیر precision و recall توجه کنیم. با توجه به تعریف ما در این مسئله هنگام ارزیابی، کلاس positive کلاس fraud (1) و کلاس negative کلاس no fraud (0) در نظر گرفته شده است. پس با توجه به رابطه $Recall = \frac{TP}{TP+FN}$ ، نشان دهنده نسبت تشخیص های درست تراکنش های *fraud* به مجموع تعداد درست تراکنس های *fraud* و تراکنش هایی که *fraud* بوده و به استباه *no fraud* تشخیص داده شده اند. با توجه به اینکه این مقدار 87درصد به دست آمده نشان میدهد که این شبکه تعداد بسیار اندکی(13) تا از تراکنش هایی که *fraud* بوده را به اشتباه *no fraud* تشخیص داده است که با توجه به هدف مسئله که تشخیص حداکثری *fraud* بوده است عملکرد مناسبی است.

**۱-6. با threshold های مختلف برای oversampling عملکرد مدل را بررسی و نمودار recall& accuracy را رسم کنید.**

در این مسئله برخلاف خواسته سوال مفهوم *threshold* را برای *SoftMax* در نظر گرفتیم. همانگونه که میدانیم تابع *SoftMax* با توجه به تعداد کلاس های خروجی که در این مسئله دوتاست، دو مقدار میدهد که این دو مقدار احتمال وجود ورودی در یکی از آن دو کلاس است و این دو مقدار عددی بین 0و1 هستند که مجموع آنها 1 خواهد بود. حال هرچه شبکه دقیق تر باشد، با افزایش *Threshold* نباید مقدار *recall* خیلی کاهش بیابد که این یعنی خروجی شبکه با احتمال بسیار نزدیک صفر به ما داده می‌شود.




(ب) (الف)

**شکل 13.1** الف) نمودار **recall-threshold** ب) نمودار **accuracy-threshold**

همانگونه که مشاهده میشود روند نزولی بسیار کمی در نمودار recall اتفاق افتاده است که این نشان دهنده عملکرد بسیار خوب شبکه در تشخیص درست تراکنش های fraud است (نتایج به دست آمده از نتایج مقاله بهتر شده‌است!). از طرفی نیز روند صعودی مناسب acc نیز در قسمت (ب) مشاهده میشود که دلیل تفاوت آن با نمودار مقاله به دلیل این است که ما ارزیابی ها را برروی مدل با بهترین وزن ها براساس loss_val انجام دادیم.

## ۱-7. مدل را با استفاده از داده‌های unbalanced و بدون حذف نویز، آموزش و موارد سؤال 6 را گزارش دهید، نتایج دو مدل را با هم مقایسه کنید.

برای آموزش شبکه با داده های unbalance، بعد از جداسازی داده های train,test و val همین داده ها را به عنوان ورودی به شبکه میدهیم. در این حالت دیگر نیاز به لود کردن مدل autoencoder نیز نمی‌باشد و مدل شبکه یصورت شکل 14.1 خواهد بود.

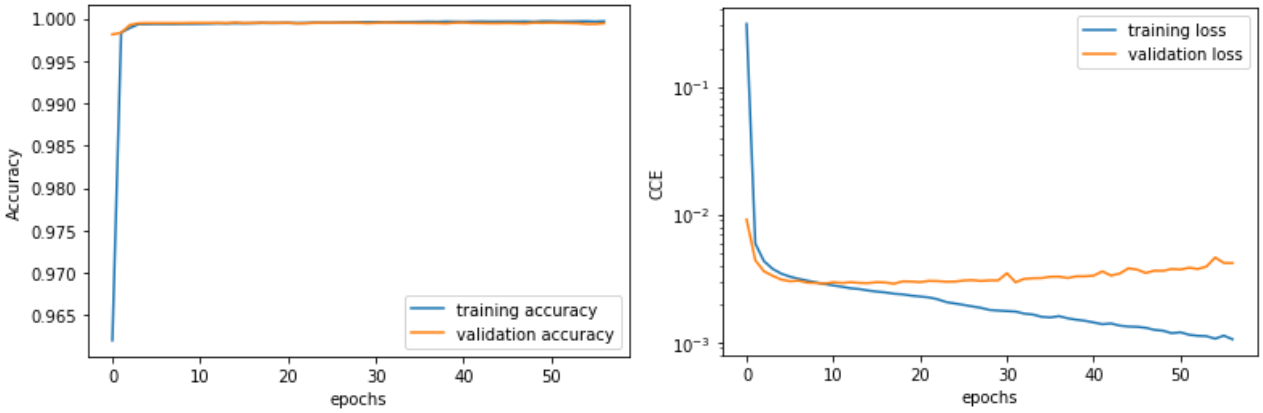
```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 29)]              0

 dense (Dense)               (None, 22)                660

 dense_1 (Dense)             (None, 15)                345

 dense_2 (Dense)             (None, 10)                160

 dense_3 (Dense)             (None, 5)                 55

 dense_4 (Dense)             (None, 2)                 12

=================================================================
Total params: 1,232
Trainable params: 1,232
Non-trainable params: 0
_________________________________________________________________
```

**شکل 14.1** مدل شبکه بدون **oversampling**

همانگونه که مشاهده میشود داده های ورودی مستقیما وارد لایه اول Classifier میشوند. در این شبکه نیز در همه لایه ها به جز لایه آخر از تابع ReLU و در لایه آخر از تابع SoftMax استفاده شده است. Loss function این شبکه نیز با توجه به اینکه کلاس های خروجی دوتا هستند، Binary Cross Entropy انتخاب شده است. برای اینکه شرایط برای هردو شبکه یکسان باشد، این شبکه را نیز به تعداد 57 ایپاک که برابر است با تعداد ایپاک های حالت قبل آموزش میدهیم و شبکه با بهترین وزن‌ها را در best_model_withoutSMOTE.h5 ذخیره می‌کنیم.

```
results_Classifier = Classifier.fit(X_train, y_train,
                                    batch_size=1024,
                                    epochs=57,
                                    validation_data = (X_val, y_val),
                                    #shuffle=True,
                                    callbacks=[mc])
```

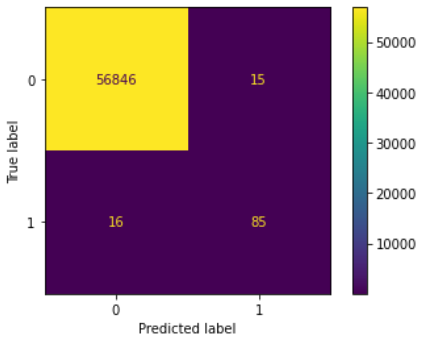
```
179/179 [==============================] - 1s 4ms/step - loss: 0.0011 - acc: 0.9996 - val_loss: 0.0042 - val_acc: 0.9994
Epoch 57/57
168/179 [===========================>..] - ETA: 0s - loss: 9.9845e-04 - acc: 0.9997
Epoch 57: val_loss did not improve from 0.00288
179/179 [==============================] - 1s 4ms/step - loss: 0.0011 - acc: 0.9997 - val_loss: 0.0042 - val_acc: 0.9995
```




**شکل 15.1** ارزیابی **training** شبکه بدون **oversampling**

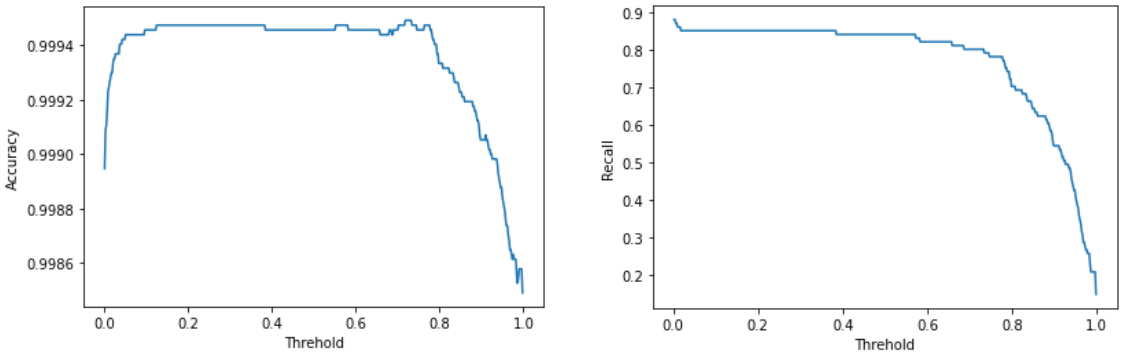
همانگونه که مشاهده میشود شبکه به خوبی آموزش دیده و حتی val_loss در آموزش شبکه نیز از شبکه با oversamplimg نیز کمتر شده‌است.

در نهایت نتایج ارزیابی این مدل روی داده های تست در شکل 16.1 قابل مشاهده است.




| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 1.00 | 1.00 | 1.00 | 56861 |
| 1 | 0.85 | 0.84 | 0.85 | 101 |
| accuracy | | | 1.00 | 56962 |
| macro avg | 0.92 | 0.92 | 0.92 | 56962 |
| weighted avg | 1.00 | 1.00 | 1.00 | 56962 |

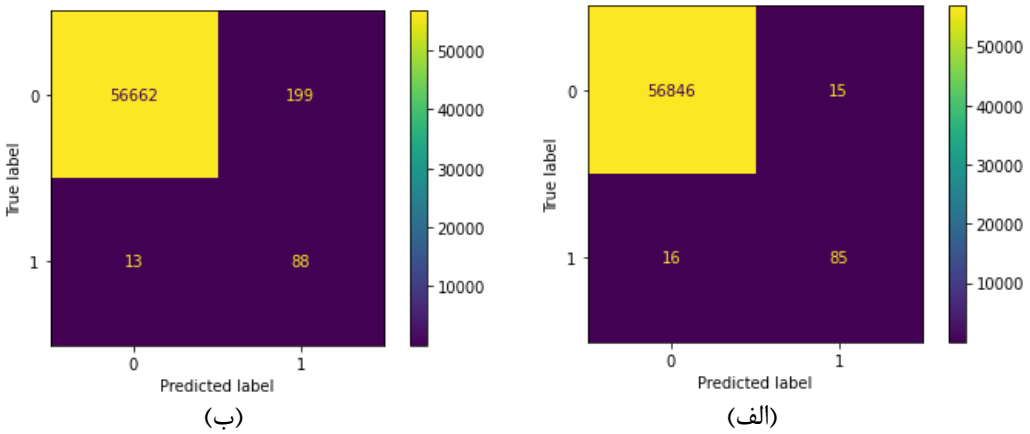
**شکل 16.1** ارزیابی مدل بدون **oversampling** برروی داده های تست




(ب) (الف)

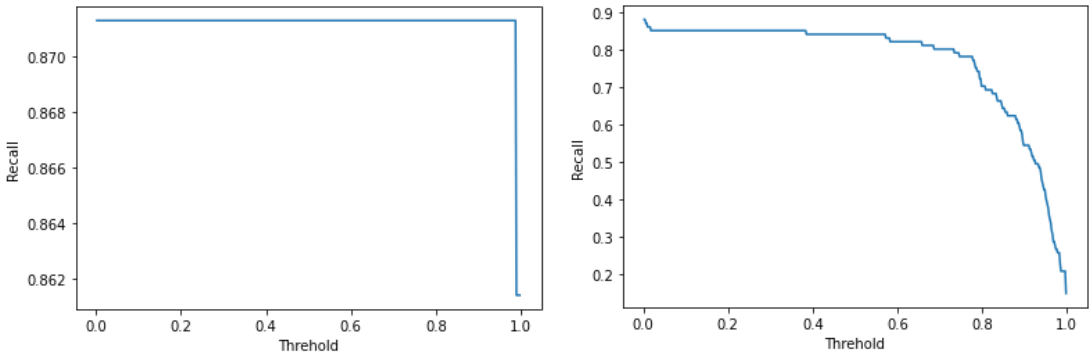
**شکل 17.1** الف) نمودار **recall-threshold** ب) نمودار **accuracy-threshold**

با دقت در نتایج نهایی میبینیم که این شبکه نیز دقت 100درصد دارد اما recall این شبکه نسبت به حالت با oversampling 3 درصد کمتر است که نشان دهنده این است که 3 داده مربوط به fraud را در دسته no fraud دسته بندی کرده است (با توجه به confusion matrix). اما این شبکه در تشخیص درست داده های no fraud عملکرد بسیار بهتری دارد و به همین دلیل نیز precision این شبکه با اختلاف از حالت قبل بهتر است اما در این مسئله برای ما این مورد که Fraud ها را بهتر تشخیص دهیم گزینه مورد اهمیت‌تر است. در حالت کلی همواره بین precision و recall حالت trade-off وجود دارد، اگر بخواهیم recall بالاتری داشته باشیم، precision کاهش خواهد یافت.




(ب) (الف)

**شکل confusion matrix 18.1** الف) بدون **oversampling** ب) با **oversampling**




(ب) (الف)

**شکل 19.1** نمودار **recall-threshold** الف) بدون **oversampling** ب) با **oversampling**

همانگونه که در تصویر 18.1 مشاهده میشود با مقایسه confusion matrix ها میبینیم که در مدل با oversampling 88 مورد از 101 مورد fraud به درستی تشخیص داده شده است اما در مدل بدون oversampling این مقدار 85 مورد است.

از طرفی نیز با توجه به شکل 19.1 در این مدل، recall با افزایش threshold رفته رفته کاهش میابد و به صفر میرسد که نشان دهنده این است که با افزایش threshold توانایی تشخیص fraud را از دست می‌دهد.

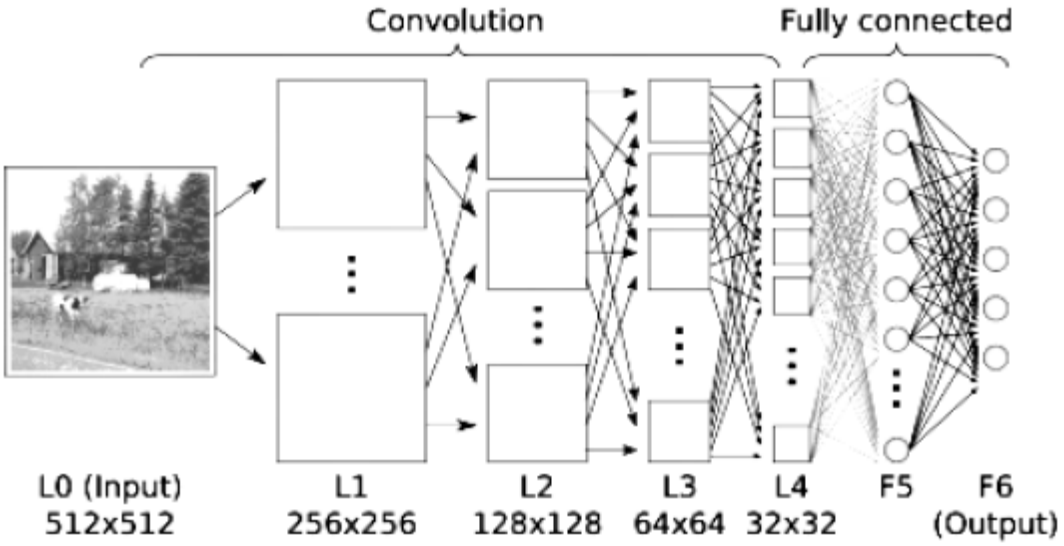
# پاسخ 3. تشخیص کاراکتر نوری (Optical character recognition)

در این سوال با استفاده از دیتاست HODA به بررسی تشخیص اعداد دست نویس فارسی با استفاده از optimizer های متفاوت میپردازیم.

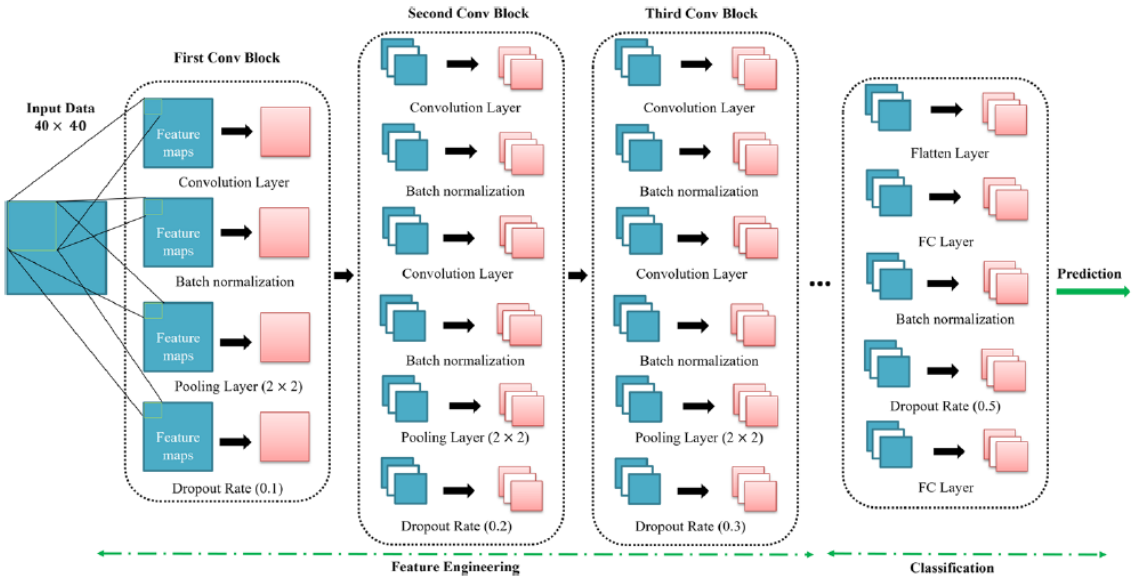
## 3-1. تفاوت بین شبکه های CNN و DCNN را توضیح دهید.

شبکه های کانولوشنی برای استخراج ویژگی از تصاویر استفاده میشود این دو شبکه تفاوت چندانی به جز در تعداد لایه ها با یکدیگر ندارند بیشتر معماری های مدرن شبکه های کانولوشنی بین 30 تا 100 لایه در خود دارند. DCNN برای شناسایی اجسام و الگو ها از **لایه های کانولوشنی مجزا** استفاده میکند که از الگوی عصبی سه بعدی موجود در قشر بینایی حیوانات الهام گرفته شده است.

نمونه ای از شبکه CNN و DCNN در شکل زیر دیده میشود.




شکل 3-1 معماری **CNN**




شکل 3-2 معماری **DCNN** استفاده شده در مقاله

## 3-2. سه روش بهینه سازی Adam و Adadelta و Momentum را توضیح دهید.

- Adam

الگوریتم آدام یک الگوریتم بهینه‌سازی است که می‌توان از آن به جای روش گرادیان کاهشی تصادفی کلاسیک برای به‌روزرسانی وزن‌های شبکه بر اساس تکرار در داده‌های آموزشی استفاده کرد. الگوریتم آدام را می‌توان به عنوان ترکیبی از RMSprop و گرادیان نزولی تصادفی با گشتاور (Momentum) در نظر گرفت.

Adam با نگه داشتنmoving averages of gradient مشابه RMSProp و مربع آن درست مانند Adadelta کار می کند. علاوه بر این، به روز رسانی آن متناسب است.

$$\text{aredgradient}$$

$$r_t = \nabla_\theta f_t(\theta_{t-1}),$$

$$s_t = w_1.s_{t-1}(1-w_1).r_t, \text{ (update biased 1st momentum estimate)}$$

$$d_t = w_2.d_{t-1}(1-w_2).r_t^2, \text{ (update biased 2nd momentum estimate)}$$

$$\widehat{s_t} = \frac{s_t}{(1-w_1^t)}, \text{ (bias 1st momentum)}$$

$$\widehat{d_t} = \frac{d_t}{(1-w_2^t)}, \text{ (bias 2nd momentum)}$$

$$\theta_t = \theta_{t-1} - \alpha \frac{\widehat{s_t}}{(\sqrt{\widehat{d_t}+\epsilon})}, \text{ (update parameter)}$$

در اینجا f(θ) تابع اسکالر تصادفی است . و معادلات زیر به روز رسانی Adam را انجام میدهند.

$$\alpha_t = \alpha . \frac{\sqrt{1-\omega_2^t}}{1-\omega_1^t},$$
$$\theta_t = \theta_{t-1} - \alpha_t . \frac{s_t}{\sqrt{d_t + \hat{\varepsilon}}}.$$

آسان بودن پیاده سازی ، نیاز کم به حافظه ، محاسبات بهینه ، عد وابستگی به مقیاس دهی مجدد قطری گرادیان ، مناسب بودن برای مسائل دارای داده یا پارامتر زیاد و مسائلی با گرادیان های نویزدار ، نیاز به تنظیم پارامتر کم از ویژگی های این الگوریتم می باشد.

- Adadelta

برای غلبه بر مشکل decaying learning rate که در که در adaGrad با آن مواجه می‌شویم، الگوریتم یادگیری جدیدی به نام Adadelta معرفی شد. با تنظیم اندازه پنجره، تجمع گرادیان مربع قبلی را مانند adagrad محدود می کند. نرخ یادگیری به طور کلی جذر نسبت بین میانگین متحرک مربع گرادیان و مربع به روز رسانی وزن است.

در اینجا، میانگین در حال اجرا به عنوان $E[g^2]t$ نشان داده می شود و به گرادیان فعلی و میانگین قبلی بستگی دارد.

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2;$$

اکنون معادله اصلاح شده SGD به روز رسانی با توجه به بردار به روز رسانی پارامتر را می توان به صورت زیر نوشت.

$$\Delta\theta_i = -\eta . g_{t,i},$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t,$$
$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]t + \varepsilon}} . g_t,$$

در نتیجه خواهیم داشت:

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} . g_t .$$

در این روش نرخ یادگیری کاهش نمی یابد و آموزش متوقف نمیشود اما نیازمند محاسبات زیاد است.

- Momentum

Momentum با هدف کاهش واریانس در گرادیان کاهشی تصادفی ابداع شد. Momentum، همگرایی به جهت مورد نظر را تسریع بخشیده و از گرایش به جهات نامربوط پیشگیری می‌کند. در این روش به جای اینکه تنها از مقدار گرادیان در مرحله ی فعلی به منظور هدایت جستجو استفاده شود، Momentum گرادیان مراحل گذشته را نیز محاسبه می کند و از آن برای تعیین جهت گرادیان استفاده می کند. هایپرپارامتر دیگری موسوم به Momentum در این روش استفاده می‌شود.

در درجه اول، SGD یکی از روش های بهینه سازی قابل دسترس است. با این حال، زمان صرف شده برای آموزش مدل نسبتاً بالاتر است. بنابراین، برای انجام همگرایی سریعتر، SGD با بهینه سازهای دیگر مانند adadelta، Adam و momentum بهینه می شود. وقتی در مورد تکانه صحبت می کنیم، توپ در حال حرکت به سمت پایین تپه به جمع آوری تکانه ادامه می دهد و در مسیر خود شتاب می گیرد تا به سرعت نهایی برسد. همین پدیده در مورد پارامتر به روز شده ما نیز اتفاق می افتد. ابعاد بیان تکانه در جهت مربوطه (در جهت نقطه گرادیان) افزایش می یابد و برای ابعادی که شیب آنها تغییر جهت می دهد، کاهش می یابد. این کار با افزودن کسر زمان گذشته بردار به روز رسانی به بردار به روز رسانی فعلی انجام می شود. به این ترتیب، کاهش به روز رسانی پارامتر اضافی صورت می گیرد. در نتیجه، همگرایی سریعتر با ضریب تکانه (γ) صورت می گیرد.

$$\upsilon_t = \gamma \upsilon_t - 1 + \eta \nabla_\theta J(\theta),$$
$$\theta = \theta - \upsilon_t,$$

برای به روز رسانی وزنها نیز اط معادلات زیر استفاده میشود:

$$v_t = \mu v_{t-1} - \alpha \nabla L_t(\omega_{t-1}),$$
$$\omega_t = \omega_{t-1} + v_t.$$

این روش نوسانات و واریانس بالای پارامترها را کاهش میدهد و همگرایی سریعی نسبت به گرادیان کاهشی دارد انا با اضافه شدن هایپرپارامتری دیگر که باید بصورت دستی و دقیق محاسبه شود متواند مشکل ساز باشد.

**3-3. معماری DCNN استفاده شده در مقاله راپیاده سازی کنید. پیش پردازش ونرمالایز سازی های مورد استفاده را بیان کنید. تعداد لایه های بکار رفته و نوع لایه و علت استفاده ازآن ها را توضیح دهید. به منظور جلوگیری از overfitting چه تکنیکی به کار رفته است.**

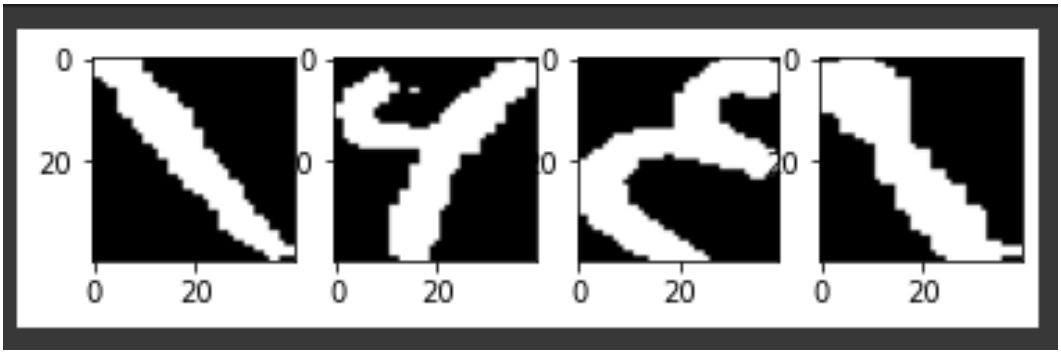
- **پیش پردازش**

دیتاست مورد نظر شامل 102352 نمونه از اعداد 0 تا 9 دست نویس می باشد که 60000 داده ی آموزشی و 20000 داده ی تست را در بر میگیرد این دیتا باید در 10 کلاس ، کلاس بندی شود.

طبیعت دیتاست دیجیتال ورودی میتواند در کارایی شبکه تاثیر گذار باشد پس بهت است قبل از هرگونه پردازش ریداندانس موجود در دیتا را از بین ببریم در این مقاله ابتدا تمام تصاویر موجود در دیتا به سایز 40*40 تغییر پیداکرده است سپس دیتا به باینری 0 برای پس زمینه و تا 255 برای اعداد تبدیل شده است پس از ان با تقسیم بر 255(نرمالایز کردن دیتا) و سپس با ترشلد گذاری به 0 و 1 تبدیل شده شده است. برای لیبل هر تصویر نیز به **one-hot** تبدیل شده است کد زیر پیش پردازش انجام شده را نشان میدهد.

```
def preprocessing(dataset_path):
    images, labels = read_cdb(dataset_path)
    assert len(images) == len(labels)

    IMAGES = np.zeros(shape=[len(images), 40, 40], dtype=np.float32)
    LABELS = np.zeros(shape=[len(labels)], dtype=int)

    for i in range(len(images)):
        image = images[i]
        image = cv2.resize(image,(40,40))#resize
        image = image / 255 #normalization
        image = np.where(image >= 0.5, 1, 0)#binarization
        IMAGES[i] = image
        LABELS[i] = labels[i]
    LABELS = to_categorical(LABELS.astype(dtype=np.float32))# One-
hot encoding label
    return IMAGES, LABELS
```

چند نمونه از تصاویر پس از پیش پردازش در زیر دیده میشود:




شکل 3-3 چند نمونه از دیتاست پیش پردازش شده

- معماری شبکه DCNN

بصورت بصری میتوان نوع لایه های موجود در معماری شبکه طراحی شده را مشاهده کرد جزئیات بیشتر در جدول زیر مشاهده میشود.

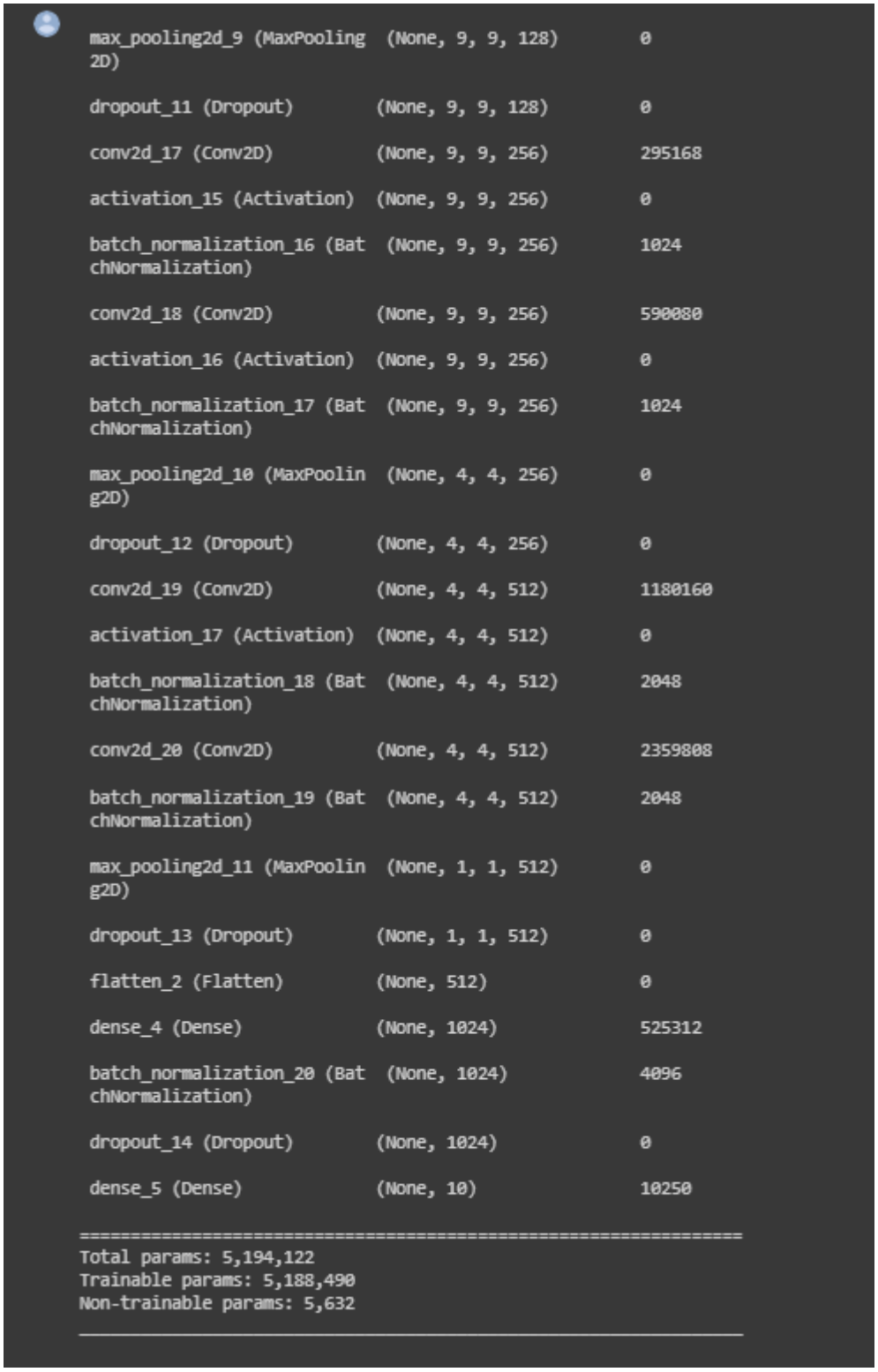
جدول 3-1 لایه های موجود در شبکه **DCNN**

| Block | No of Filter | Stride | Layers | Filter Size |
|---|---|---|---|---|
| Image Input | - | - | 40*40*1 | - |
| Feature Extraction | | | | |
| 1st Conv-Block | 64 | 1*1 | Conv Layer | 3*3 |
| | - | 1*1 | Activation Layer (Relu) | - |
| | - | - | Batch Normalization | - |
| | - | 2*2 | Pooling Layer | 3*3 |
| | - | - | Dropout (0.1) | - |
| 2nd Conv-Block | 128 | 1*1 | Conv Layer | 3*3 |
| | - | 1*1 | Activation Layer (Relu) | - |
| | - | - | Batch Normalization | - |
| | 128 | 1*1 | Conv Layer | 3*3 |
| | - | 1*1 | Activation Layer (Relu) | - |
| | - | - | Batch Normalization | - |
| | - | 2*2 | Pooling Layer | 3*3 |
| | - | - | Dropout (0.2) | - |
| 3rd Conv-Block | 256 | 1*1 | Conv Layer | 3*3 |
| | - | 1*1 | Activation Layer (Relu) | - |
| | - | - | Batch Normalization | - |
| | 256 | 1*1 | Conv Layer | 3*3 |
| | - | 1*1 | Activation Layer (Relu) | - |
| | - | - | Batch Normalization | - |
| | - | 2*2 | Pooling Layer | 3*3 |
| | - | - | Dropout (0.3) | - |
| 4th Conv-Block | 512 | 1*1 | Conv Layer | 3*3 |
| | - | 1*1 | Activation Layer (Relu) | - |
| | - | - | Batch Normalization | - |
| | 512 | 1*1 | Conv Layer | 3*3 |
| | - | - | Batch Normalization | - |
| | - | 2*2 | Pooling Layer | 3*3 |
| | - | - | Dropout (0.4) | - |
| Classification Block | | | | |
| | - | - | Flatten Layer | - |
| | - | - | FC Layer (1024) | - |
| | - | - | Batch Normalization | - |
| | - | - | Dropout (0.5) | - |
| | - | - | FC Layer (10 classes) | - |
| | - | - | Output | - |

کد این شبکه در زیر با اشتفاده از کراس دیده میشود:

```
model = Sequential()

# 1st Conv-Block
model.add(Conv2D(64, kernel_size=(3,3), strides=(1, 1), input_shape=(
40, 40,1), padding='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(3,3), strides=(2,2)))
model.add(Dropout(0.1))

# 2nd Conv-Block
model.add(Conv2D(128,kernel_size=(3,3), strides=(1, 1), padding='same
'))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(Conv2D(128,kernel_size=(3,3), strides=(1, 1), padding='same
'))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(3,3), strides=(2,2)))
model.add(Dropout(0.2))

# 3rd Conv-Block
model.add(Conv2D(256,kernel_size=(3,3), strides=(1, 1), padding='same
'))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(Conv2D(256,kernel_size=(3,3), strides=(1, 1), padding='same
'))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(3,3), strides=(2,2)))
model.add(Dropout(0.3))

# 4th Conv-Block
model.add(Conv2D(512,kernel_size=(3,3), strides=(1, 1), padding='same
'))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(Conv2D(512,kernel_size=(3,3), strides=(1, 1), padding='same
'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(3,3), strides=(2,2)))
model.add(Dropout(0.4))

# Classification Block
model.add(Flatten()) # 3D feature => 1D feature vector
```

```
model.add(Dense(1024, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
```



| | | | |
|---|---|---|---|
| max_pooling2d_9 (MaxPooling 2D) | (None, 9, 9, 128) | 0 |
| dropout_11 (Dropout) | (None, 9, 9, 128) | 0 |
| conv2d_17 (Conv2D) | (None, 9, 9, 256) | 295168 |
| activation_15 (Activation) | (None, 9, 9, 256) | 0 |
| batch_normalization_16 (Bat chNormalization) | (None, 9, 9, 256) | 1024 |
| conv2d_18 (Conv2D) | (None, 9, 9, 256) | 590080 |
| activation_16 (Activation) | (None, 9, 9, 256) | 0 |
| batch_normalization_17 (Bat chNormalization) | (None, 9, 9, 256) | 1024 |
| max_pooling2d_10 (MaxPoolin g2D) | (None, 4, 4, 256) | 0 |
| dropout_12 (Dropout) | (None, 4, 4, 256) | 0 |
| conv2d_19 (Conv2D) | (None, 4, 4, 512) | 1180160 |
| activation_17 (Activation) | (None, 4, 4, 512) | 0 |
| batch_normalization_18 (Bat chNormalization) | (None, 4, 4, 512) | 2048 |
| conv2d_20 (Conv2D) | (None, 4, 4, 512) | 2359808 |
| batch_normalization_19 (Bat chNormalization) | (None, 4, 4, 512) | 2048 |
| max_pooling2d_11 (MaxPoolin g2D) | (None, 1, 1, 512) | 0 |
| dropout_13 (Dropout) | (None, 1, 1, 512) | 0 |
| flatten_2 (Flatten) | (None, 512) | 0 |
| dense_4 (Dense) | (None, 1024) | 525312 |
| batch_normalization_20 (Bat chNormalization) | (None, 1024) | 4096 |
| dropout_14 (Dropout) | (None, 1024) | 0 |
| dense_5 (Dense) | (None, 10) | 10250 |

Total params: 5,194,122
Trainable params: 5,188,490
Non-trainable params: 5,632

شکل 3-4 خلاصه ای از تعداد پارامترهای موجود در شبکه

حال به بررسی لایه های موجود میپردازیم:

در لایه ی ورودی تصاویر با ابعاد (40,40,1) واردش شبکه میشوند در هر لایه شبکه CONV2D با ویژگی های مورد نظر قرار داده شده و سپس در هر مرحله لایه های زیر طبق جدول 3-1 قرار داده میشود بدین منظور از 4 لایه کانولوشن استفاده شده است.

- **Batch Normalizatio*n***:

عبارت است از نرمالیزه کردن batch های ورودی به یک لایه در آموزش یک شبکه عصبی عمیق. این کار به پایداری هر چه بیشتر فرآیند آموزش شبکه منجر می شود و همچنین تعداد epoch های لازم برای دستیابی به نتیجه مورد نظر را به طرز موثری کاهش می دهد.

$$x_{normalized} = \frac{x - \mu}{\sigma}$$

ممکن است یادگیری به شدت به مقادیر اولیه وابسته شود و دوما فرایند یادگیری نتواند خود را به مقادیر بهینه نهایی برساند بلکه در حواشی آن در حال جابجایی باشد. با استفاده از batch normalization می توان تا حد زیادی از این موارد جلوگیری کرد. در واقع در این روش با white کردن ماتریس covariance تا حدود زیادی مدل آماری ورودی ها را در طی فرایند یادگیری ثابت نگه می دارد.

- **Pooling**:

این عملگر سایز feature map را کاهش می دهد و در نتیجه تعداد پارامترهای لازم برای آموزش شبکه کاهش می یابد که این خود به کمتر شدن محاسبات می انجامد.همچنین این لایه نقاطی از feature map که دارای مقادیر چشمگیرتری هستند و در طبقه بندی موثر هستند را حفظ میکند و باقی نقاط را دور می ریزد.

لایه ادغام به عنوان لایه نمونه برداری فرعی نیز شناخته می شود. برای کاهش ابعاد تصویر بین دو لایه همزمان قرار می گیرد. علاوه بر این، عملیات ادغام پیچیدگی محاسباتی شبکه را به حداقل می رساند. که در اینجا از روش Max پنجره های 3*3 استفاده شده است.

- **Dropout:**

با استفاده از dropout به صورت تصادفی تعدادی از نورون ها رو خاموش می کنیم تا از overfitting جلوگیری شود این کار در انتهای هر لایه کانولوشنی انجام میپذیرد . همچنین با اینکار باعث می شویم که همه نورون ها در آموزش مشارکت داشته باشند و وزن یک سری از نورون ها بیهوده زیاد نشود.

این لایه بعد از هر لایه کانولوشنی با مقادیر 0.1 تا 0.4 قرار داده شده است.

در لایه آخر از 2 لایه fully connected با اندازه 1024 و 10 (10 کلاس) و بین آنها لایه dropout با اندازه 0.5 و batch normalization استفاده شده است تا عملیات لیبل دهی با استفادهاز فیچر مپ های ساخته شده در 4 لایه کانولوشنی قبلی را انجام دهد.

**4-3. نمودارهای accuracy و loss وهم چنین Confusion matrix و مقادیر precision ، Recall و f1score را برای هر کدام از سه روش بهینه سازی بیان کرده و مقایسه کنید.**

معیار های **precision** و **recall** و **F-Score** را از روی ماتریس **Confusion** میتوان به صورت زیر محاسبه کرد.

$$precision = \frac{TP}{TP + FP}, \quad Recall = \frac{TP}{TP + FN}, \quad Fscore = \frac{2 \times precision \times Recall}{(precision + Recall)}$$

**بدین منظور از** loss = MSE , epoch=20 , batch=256 **برای تمامی مدلها استفاده شده است.**

- **Adam**

```
model.compile(loss='MSE',optimizer=tf.optimizers.Adam(learning_rate=0
.0001),metrics=['accuracy'])
fitting=model.fit(X_train, y_train, batch_size=256, epochs=20, valida
tion_split=0.2)
```

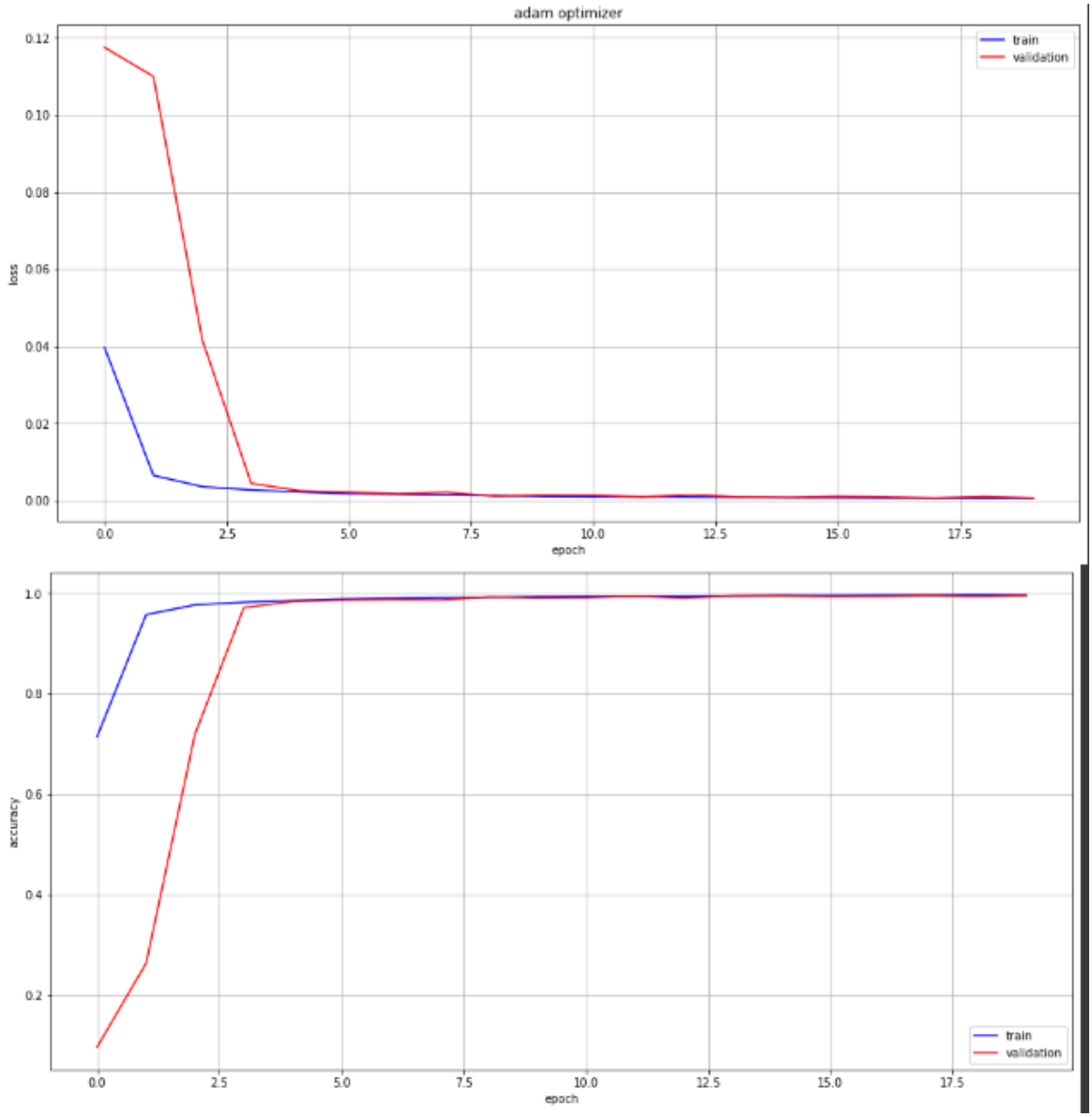
روند آموزش:

```
Epoch 13/20
188/188 [==============================] - 16s 83ms/step - loss: 9.2442e-04 - accuracy: 0.9943 - val_loss: 0.0015 - val_accuracy: 0.9904
Epoch 14/20
188/188 [==============================] - 16s 84ms/step - loss: 8.4294e-04 - accuracy: 0.9947 - val_loss: 8.8443e-04 - val_accuracy: 0.9949
Epoch 15/20
188/188 [==============================] - 16s 85ms/step - loss: 7.1576e-04 - accuracy: 0.9956 - val_loss: 8.0862e-04 - val_accuracy: 0.9951
Epoch 16/20
188/188 [==============================] - 16s 85ms/step - loss: 7.3230e-04 - accuracy: 0.9955 - val_loss: 0.0011 - val_accuracy: 0.9934
Epoch 17/20
188/188 [==============================] - 16s 84ms/step - loss: 6.4540e-04 - accuracy: 0.9960 - val_loss: 9.1866e-04 - val_accuracy: 0.9939
Epoch 18/20
188/188 [==============================] - 16s 84ms/step - loss: 6.1811e-04 - accuracy: 0.9961 - val_loss: 6.7688e-04 - val_accuracy: 0.9958
Epoch 19/20
188/188 [==============================] - 16s 85ms/step - loss: 5.5854e-04 - accuracy: 0.9967 - val_loss: 0.0010 - val_accuracy: 0.9937
Epoch 20/20
188/188 [==============================] - 16s 84ms/step - loss: 6.0836e-04 - accuracy: 0.9963 - val_loss: 6.6289e-04 - val_accuracy: 0.9958
```

نتایج:

```
# Plot the Loss finction for train & validation
fig = plt.figure(figsize=(16,8))
plt.plot(fitting.history['loss'],'b')
plt.plot(fitting.history['val_loss'],'r')
plt.title('adam optimizer')
plt.ylabel('loss')
plt.xlabel('epoch')
```

```
plt.legend(['train', 'validation'])
plt.grid()
# Plot the Accuracy
fig = plt.figure(figsize=(16,8))
plt.plot(fitting.history['accuracy'],'b')
plt.plot(fitting.history['val_accuracy'],'r')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'])
plt.grid()
```

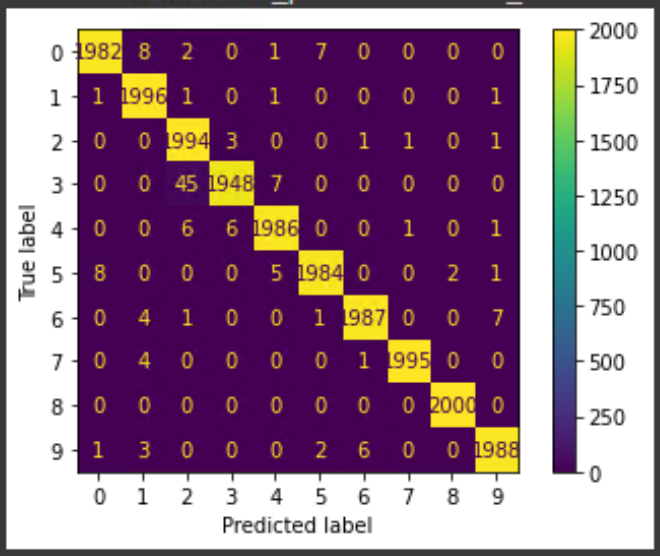



شکل 3-5 نمودار دقت و **LOSS** برای بهینه ساز **Adam**

```
y_pred= np.argmax(model.predict(X_test),axis=1)
```

```
y_t = np.argmax(y_test, axis=1)

print(classification_report(y_t, y_pred, digits=4))
print(ConfusionMatrixDisplay.from_predictions(y_t, y_pred))
```

```
625/625 [==============================] - 3s 5ms/step
              precision    recall  f1-score   support

           0     0.9950    0.9910    0.9930      2000
           1     0.9906    0.9980    0.9943      2000
           2     0.9732    0.9970    0.9849      2000
           3     0.9954    0.9740    0.9846      2000
           4     0.9930    0.9930    0.9930      2000
           5     0.9950    0.9920    0.9935      2000
           6     0.9960    0.9935    0.9947      2000
           7     0.9990    0.9975    0.9982      2000
           8     0.9990    1.0000    0.9995      2000
           9     0.9945    0.9940    0.9942      2000

    accuracy                         0.9930     20000
   macro avg     0.9931    0.9930    0.9930     20000
weighted avg     0.9931    0.9930    0.9930     20000

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay
```

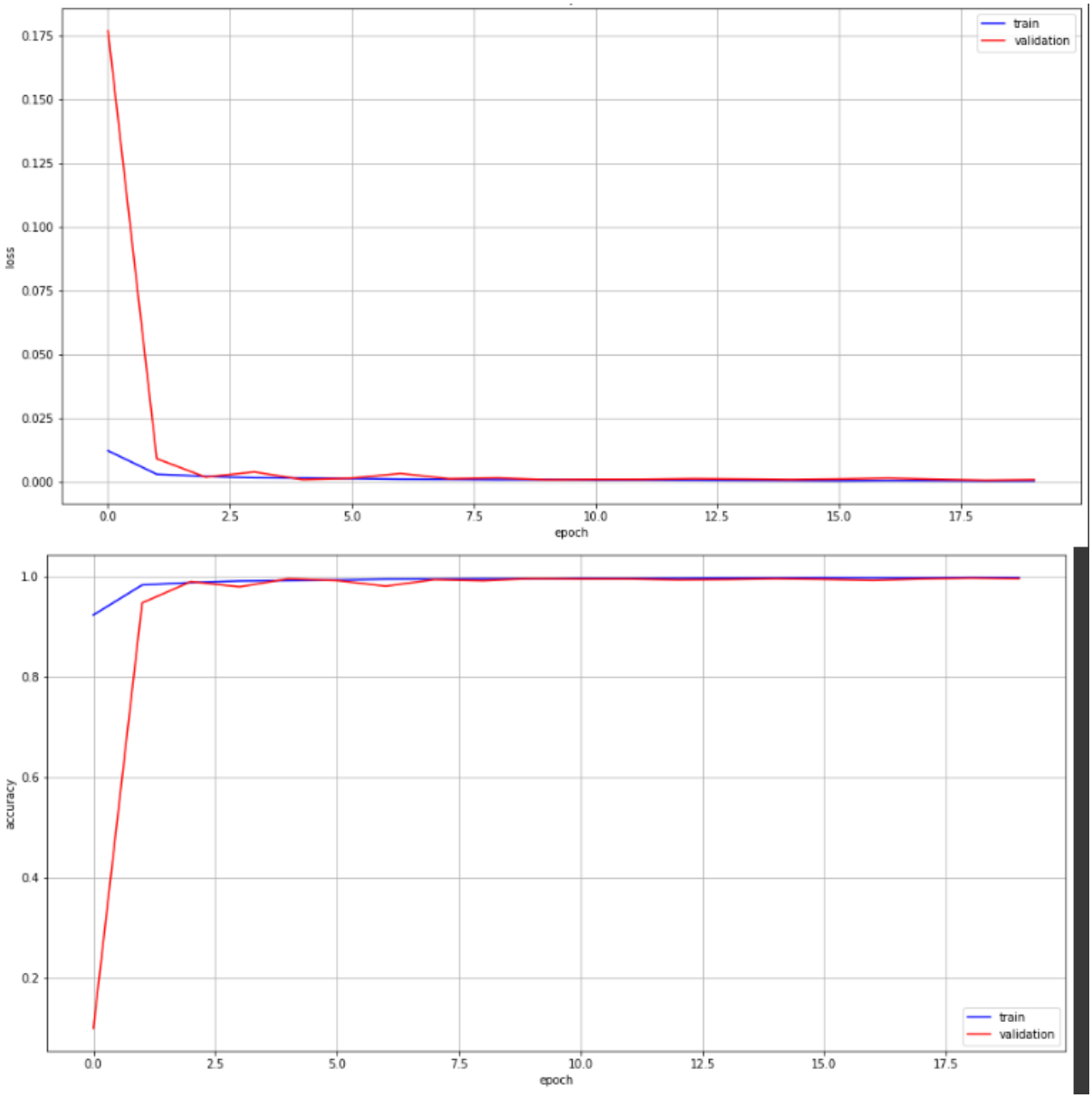



- **Adadelta**

```
model.compile(loss='MSE',optimizer=tf.optimizers.Adadelta(learning_rate=1),metrics=['accuracy'])
fitting=model.fit(X_train, y_train, batch_size=256, epochs=20, validation_split=0.2)
```

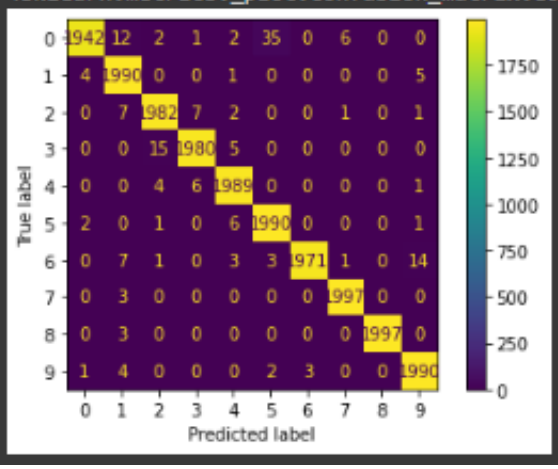
```
188/188 [==============================] - 16s 84ms/step - loss: 4.6633e-04 - accuracy: 0.9972 - val_loss: 8.2326e-04 - val_accuracy: 0.9952
Epoch 16/20
188/188 [==============================] - 16s 84ms/step - loss: 3.7434e-04 - accuracy: 0.9978 - val_loss: 0.0010 - val_accuracy: 0.9940
Epoch 17/20
188/188 [==============================] - 16s 84ms/step - loss: 4.9207e-04 - accuracy: 0.9972 - val_loss: 0.0014 - val_accuracy: 0.9921
Epoch 18/20
188/188 [==============================] - 16s 84ms/step - loss: 4.3784e-04 - accuracy: 0.9974 - val_loss: 9.2238e-04 - val_accuracy: 0.9948
Epoch 19/20
188/188 [==============================] - 16s 84ms/step - loss: 3.2269e-04 - accuracy: 0.9981 - val_loss: 5.7268e-04 - val_accuracy: 0.9966
Epoch 20/20
188/188 [==============================] - 16s 84ms/step - loss: 3.4425e-04 - accuracy: 0.9979 - val_loss: 7.6889e-04 - val_accuracy: 0.9954
```

نتایج:




شکل 3-6 نمودار دقت و **LOSS** برای بهینه ساز **Adelta**

```
625/625 [==============================] - 4s 5ms/step
              precision    recall  f1-score   support

           0     0.9964    0.9710    0.9835      2000
           1     0.9822    0.9950    0.9886      2000
           2     0.9885    0.9910    0.9898      2000
           3     0.9930    0.9900    0.9915      2000
           4     0.9905    0.9945    0.9925      2000
           5     0.9803    0.9950    0.9876      2000
           6     0.9985    0.9855    0.9919      2000
           7     0.9960    0.9985    0.9973      2000
           8     1.0000    0.9985    0.9992      2000
           9     0.9891    0.9950    0.9920      2000

    accuracy                         0.9914     20000
   macro avg     0.9915    0.9914    0.9914     20000
weighted avg     0.9915    0.9914    0.9914     20000

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay
```

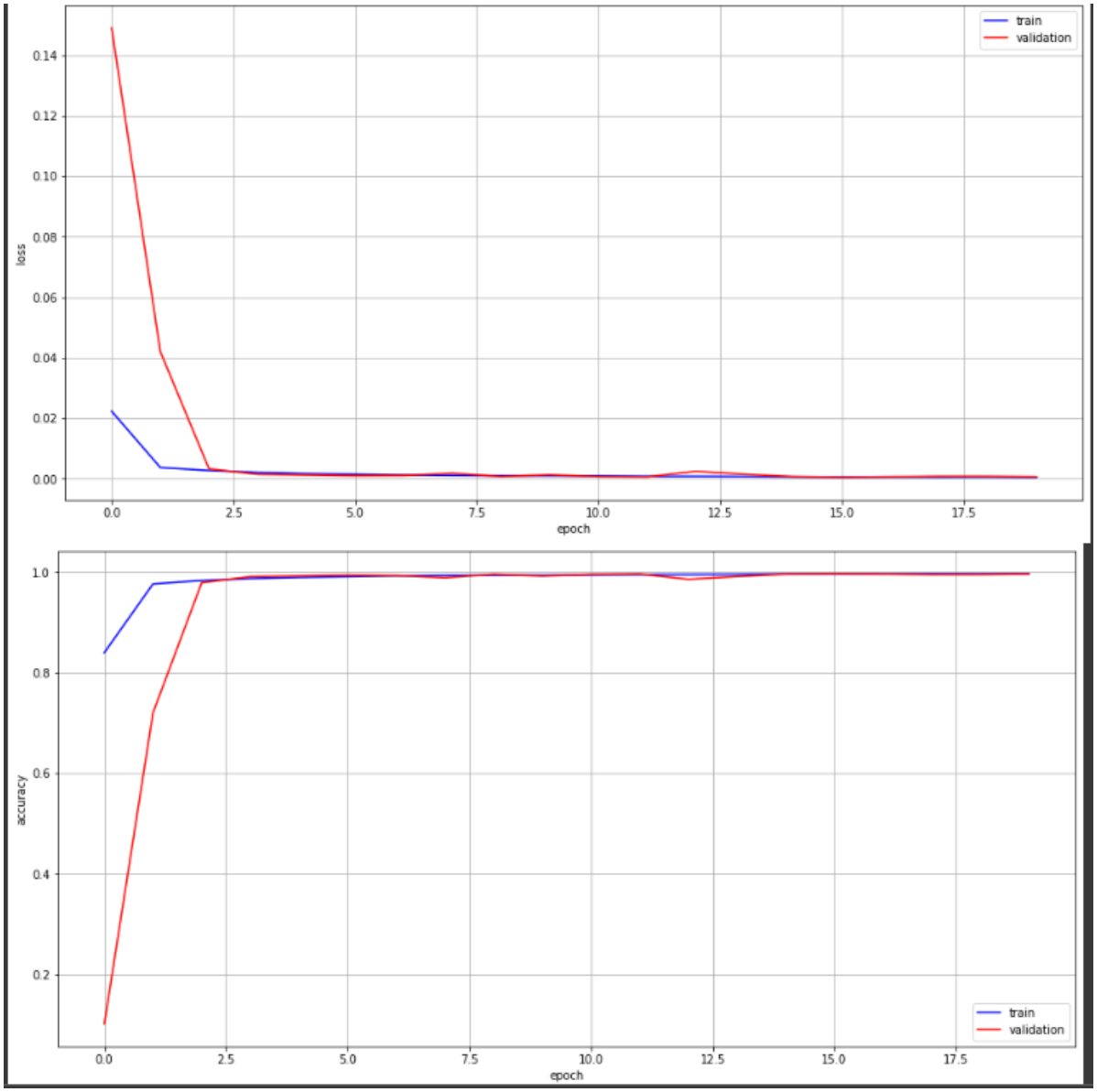



- **Momentum**

```
model.compile(loss='MSE',optimizer=tf.optimizers.SGD(learning_rate=0.
1, momentum=0.9),metrics=['accuracy'])
fitting=model.fit(X_train, y_train, batch_size=256, epochs=20, valida
tion_split=0.2)
```

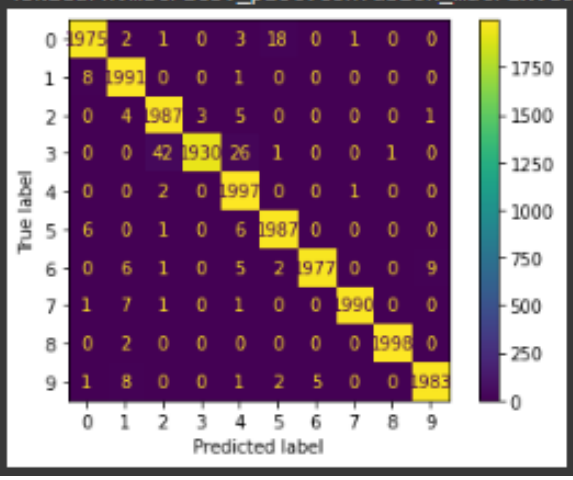
```
Epoch 15/20
188/188 [==============================] - 16s 85ms/step - loss: 6.3663e-04 - accuracy: 0.9959 - val_loss: 7.3964e-04 - val_accuracy: 0.9957
Epoch 16/20
188/188 [==============================] - 16s 85ms/step - loss: 6.5319e-04 - accuracy: 0.9963 - val_loss: 5.0768e-04 - val_accuracy: 0.9968
Epoch 17/20
188/188 [==============================] - 16s 85ms/step - loss: 6.4497e-04 - accuracy: 0.9959 - val_loss: 6.7491e-04 - val_accuracy: 0.9957
Epoch 18/20
188/188 [==============================] - 16s 85ms/step - loss: 5.6470e-04 - accuracy: 0.9964 - val_loss: 8.2113e-04 - val_accuracy: 0.9948
Epoch 19/20
188/188 [==============================] - 16s 85ms/step - loss: 6.0242e-04 - accuracy: 0.9962 - val_loss: 8.1135e-04 - val_accuracy: 0.9949
Epoch 20/20
188/188 [==============================] - 16s 85ms/step - loss: 5.2818e-04 - accuracy: 0.9967 - val_loss: 6.8913e-04 - val_accuracy: 0.9960
```

نتایج:




شکل 3-7 نمودار دقت و **LOSS** برای بهینه ساز **Momentum**

```
625/625 [==============================] - 4s 5ms/step
              precision    recall  f1-score   support

           0     0.9920    0.9875    0.9897      2000
           1     0.9856    0.9955    0.9905      2000
           2     0.9764    0.9935    0.9849      2000
           3     0.9984    0.9650    0.9814      2000
           4     0.9765    0.9985    0.9874      2000
           5     0.9886    0.9935    0.9910      2000
           6     0.9975    0.9885    0.9930      2000
           7     0.9990    0.9950    0.9970      2000
           8     0.9995    0.9990    0.9992      2000
           9     0.9950    0.9915    0.9932      2000

    accuracy                         0.9908     20000
   macro avg     0.9909    0.9908    0.9907     20000
weighted avg     0.9909    0.9908    0.9907     20000

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay o
```




میتوان مشاهده کرد که تمامی مدلها در تشخیص عدد 3 با 2 و 0 با 5 بد عمل کرده اند. و نمودار loss و دقت برای داده های ولیدیشن از ایپاک سوم بهبود یافته است و از ایپاک 5 به بعد تقریبا به مقدار ثابتی رسیده است.

## 3-5. معماری و پارامترهای بهترین شبکه ر ا بیان کنید.

پارامترهای کلی در نظر گرفته سده برای تمامی مدلها بصورت زیر می باشد:

```
fitting=model.fit(X_train, y_train, batch_size=256, epochs=20, valida
tion_split=0.2)
```

طبق نتایج قسمت قبل بهترین بهینه ساز ها به ترتیب و بهترین پارامترهای هر بخش بصورت زیر است:

1. **Adam optimizer with : learning_rate=0.0001 , loss='MSE'**
2. **Adadelta optimizer with : learning_rate=1 , loss='MSE'**

3. **Momentum optimizer with : SGD(learning_rate=0.1, momentum=0.9) , loss='MSE'**

باید دقت شود که تفاوت دقت بین مدلهای نذکور در حدود 0.05-0.1 می باشد.