به نام خدا

دانشگاه تهران

دانشکده مهندسی

برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

تمرین extra

نام و نام خانوادگی	فاطمه نائینیان – محمد عبائیانی
شماره دانشجویی	۸۱۰۱۹۸۴۷۹ - ۸۱۰۱۹۸۴۳۲
تاریخ ارسال گزارش	1401-09-28

فهرست

پاسخ ۱ – تشخیص تقلب (fraud detection) با استفاده از شبکه عمیق

(1

در توسعه مدل های تشخیص تقلب چالش های زیادی وجود دارد اما بزرگترین چالش این است که تقلب به ندرت اتفاق می افتد و طبیعتا داده های موجود از تقلب در مقایسه با داده های اصلی تعداد خیلی کمتری دارند.

اگر داده ها را همانطور که در دسترس داریم به مدل بدهیم و مدل را اجرا کنیم، مدل دقت خوبی میدهد. اما این دقت ناشی از این است که داده های تقلبی نیز به عنوان داده های اصلی برچسب زده شده اند. مثلا اگر ۰.۱٪ داده ها تقلب باشند، مدل ۹۹.۹٪ دقت میگیرد اما این درصد قابل استناد نیست چون recall بسیار کمی دارد. بنابراین مدل کردن دیتاهای imbalanced سبب می شود تا دیتاهای کلاس کوچکتر برچسب کلاس بزرگتر را بگیرند.

یکی از راه هایی که برای حل این چالش وجود دارد این است که داده های غیر بالانس را بالانس کنیم. به این ترتیب که با کمک داده هایی که از کلاس کوچک تر داریم، داده های جدید تولید کنیم تا تعداد این کلاس به کلاس بزرگتر برسد. این روش oversampling نامیده می شود.

(2

شبکه معماری بسیار ساده ای دارد. ابتدا داده ها را oversample میکنیم تا تعداد داده های هر کلاس برابر شود. سپس داده ها را با نویز جمع میکنیم و به autoencoder میدهیم. autoencoder شامل ۷ لایه است. لایه ها fully connected هستند. autoencoder داده های نویزدار را میگیرد و در نهایت داده های denoise شده را خروجی میدهد. مقادیر ورودی های هر لایه ان به شکل زیر است.

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

شکل ۱: لایه های **autoencoder**

سپس خروجی autoencoder را به classifier میدهیم. classifier از ۶ لایه fully connected تشکیل شده است که در اخر خروجی ان را از یک SoftMax عبور میدهیم تا دو کلاس از هم تفکیک شوند. اندازه ورودی هر لایه به شکل زیر است.

Denoised Dataset (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

شکل ۲: لایه های **classifier**

(3

هنگامی که دیتاست بالانس نباشد باید ان را بالانس کنیم. برای اینکار روش های گوناگونی وجود دارد که چند مورد را در ادامه مشاهده میکنیم.

SMOTE : این روش در مقاله استفاده شده است. الگوریتم knn را برای داده های کلاس کوچکتر اجرا میکنید و سپس داده ای رندوم که مابین داده اصلی و همسایه اش است را پیدا میکند. این روش پیچیده و زمان بر است. ولی به دلیل اینکه داده های جدید از روی داده های قبلی حساب نمی شوند، داده تکراری نخواهیم داشت.

Cluster based Oversampling : این روش برعکس بقیه روش ها نمیخواهد تا تعداد داده های کلاس ها برابر شود بلکه با الگوریتم k means داده ها را cluster میکند و سعی میکند تا داده های هر cluster را یکسان کند. در این روش اگر تعداد داده های دو کلاس خیلی متفاوت باشد نمیتواند به خوبی عمل کند و در نهایت باز هم مدل کلاس کوچکتر را در نظر نمیگیرد. در کل اگر کلاس ها خیلی تفاوت تعداد نداشته باشند و داده ها دارای cluster های جدا باشند میتواند کمک کننده باشد.

Random Oversampling : در این روش تعداد داده های کلاس کمتر را با duplicate کردن داده ها به صورت رندوم افزایش میدهد. اما این روش اگر کلاس کوچکتر خیلی کوچک باشد باعث می شود تا تعداد زیادی داده تکراری داشته باشیم و عملکرد مدل خراب می شود. ساده و سریع است و هنگامی خوب است که دو کلاس تفاوت داده کمی داشته باشند.

Random Undersampling : در این روش تعداد داده های کلاس بزرگتر را با حذف کردن داده ها به صورت رندوم کاهش میدهد تا تعداد ان به کلاس کوچکتر برسد. اما این روش اگر کلاس کوچکتر خیلی

کوچک باشد باعث می شود تا تعداد زیادی داده از دست برود و عملکرد مدل خراب می شود. ساده و سریع است و هنگامی خوب است که دو کلاس تفاوت داده کمی داشته باشند. در غیر این صورت خطای بزرگی در مدل رخ میدهد.

(4

برای پیاده سازی مدل ابتدا لازم است تا دیتاست را لود کنیم. سپس طبق مقاله ستون time را حذف میکنیم. ستون class به عنوان برچسب کلاس ها و باقی ستون ها را به عنوان ویژگی های داده ها برای یادگیری به مدل میدهیم. سپس ٪۲۰ داده ها را به عنوان داده های تست جدا میکنیم.

```
data_frame = pd.read_csv("creditcard.csv")
```

```
data_frame = data_frame.drop(['Time'], axis=1)
Y = data_frame['Class']
X = data_frame.drop(['Class'], axis=1)

X['Amount'] = (X['Amount']-X['Amount'].mean())/ X['Amount'].std()

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.2, random_state=42)
```

شکل ۳: انجام **preprocessing** بر روی داده ها

در ادامه با کمک تابع SMOTE که در مقاله نام برده شده، داده ها را resample میکنیم. به این ترتیب تعداد داده های کلاس کوچک و بزرگ برابر می شود. سپس یک نویز گوسی به داده ها اضافه میکنیم.

```
smote = SMOTE(random_state=50, k_neighbors=4, sampling_strategy='minority')
X_resample, y_resample = smote.fit_resample(X_train, y_train)
noise_train = 0.4 * np.random.normal(loc=0.0, scale=1.0, size=X_resample.shape)
noise_test = 0.4 * np.random.normal(loc=0.0, scale=1.0, size=X_test.shape)
x_train_noise = X_resample + noise_train
x_test_noise = X_test + noise_test
```

شکل ۴: انجام تابع **SMOTE** و اعمال نویز

حال همانطور که در مقاله ذکر شد باید داده ها را از یک autoencoder عبور دهیم. حال طبق مقاله به شکل زیر ان را تعریف میکنیم.

```
encoder = Sequential()
encoder.add(Dense(22, activation="tanh", input_shape=(X_resample.shape[1],)))
encoder.add(Dense(15, activation="relu"))
encoder.add(Dense(10, activation="relu"))
encoder.add(Dense(15, activation='relu'))
encoder.add(Dense(22, activation='tanh'))
encoder.add(Dense(29))
encoder.compile(optimizer='adam', loss='mean_squared_error')
encoder.fit(x_train_noise,X_resample ,epochs=10,batch_size=512,shuffle=True,validation_data=(x_test_noise, X_test))
```

شکل ۵: پیاده سازی **autoencoder**

حال باید خروجی autoencoder را به fully connected ها بدهیم تا ان ها را طبقه بندی کند. طبقه مقاله classifier به شکل زیر تعریف می شود.

```
model = Sequential()
model.add(Dense(22, activation="relu",input_shape=(29,)))
model.add(Dense(15, activation="relu"))
model.add(Dense(10, activation="relu"))
model.add(Dense(5, activation="relu"))
model.add(Dense(2, activation="softmax"))
model.compile(optimizer=Adam(learning_rate=0.001), loss='categorical_crossentropy', metrics=['accuracy',[metrics.Recall(thresholds=x,class_id=1) for x in np.arange(0,1,0.01)],[metrics
history = model.fit(x_train_encoder,Y_train,epochs=2,batch_size=256,shuffle=True,validation_data=(x_valid_encoder,Y_test))
```

شکل ۶: پیاده سازی **autoencoder**

در بخش metrics به تعداد زیادی ترشلد مدل را ارزیابی میکنیم که نتایج ان در بخش های بعد قابل مشاهده است.

(5

هنگامی که برچسب ها نامتوازن باشند، معیار دقت به تنهایی کافی نیست. این معیار برابر است با تعداد برچسب های درست تقسیم بر تعداد کل داده ها. هنگامی که یکی از کلاس ها تعداد داده های بسیار کمتری نسبت به کلاس بزرگتر داشته باشد، در این صورت مدل برچسب های کلاس کوچک را اشتباه تشخیص میدهد و برچسب کلاس بزرگتر را به آن میزند. با وجود این برچسب های اشتباه، چون کلاس کوچکتر داده های کمتری دارد، تاثیر زیادی روی دقت نمیگذارد و چون داده های کلاس بزرگتر درست طبقه بندی شده اند، دقت بالایی خواهیم داشت.

معیار مکمل میتواند recall باشد. این معیار برابر است با تعداد برچسب های درست کلاس کوچک تقسیم بر کل داده های کلاس کوچک. مسلما این معیار میتواند کمک زیادی در سنجش مدل داشته باشد.

Classification	Actual Positive Sample	Actual Negative Sample
predict as positive	TP	FP
predict as negative	FN	TN

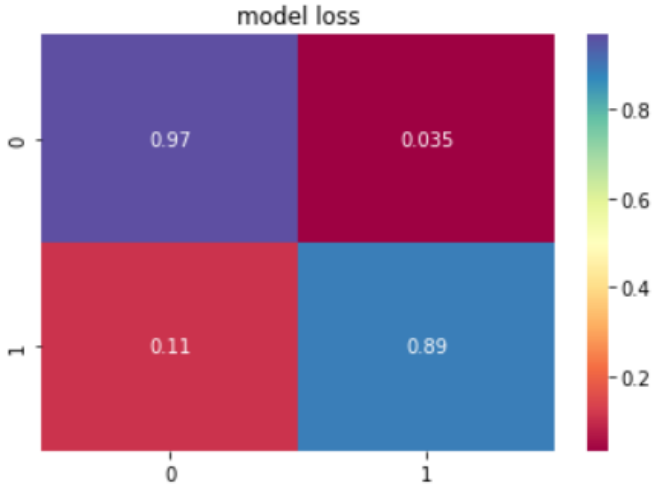
شکل ۷: جدول جهت معرفی **Recall**

$$recall = \frac{TP}{TP + FN}$$

برای مدلی که oversample شده و از autoencoder عبور داده شده، نتایج زیر را خواهیم داشت.

با توجه به نتایج زیر به خوبی مدل را بررسی میکنیم. مدل قابلیت تفکیک دو سری داده را دارد. در این حالت که داده های تست oversample نشده اند، مدل توانسته ۸۹ درصد داده های تقلب را تشخیص دهد که پیشرفت بسیار خوبی نسبت به زمانیست که این اتفاق نمی افتد. دقت برابر ٪۹۶.۵ درصد است که در این حالت همه ی دقت از دست رفته مربوط به کلاس کوچکتر نیست. Recall برابر ٪۹۲.۶ شده است که نشان میدهد مدل کنونی قابلیت تفکیک دو کلاس را دارد. در ادامه به طور کامل دو مدل را مقایسه میکنیم.

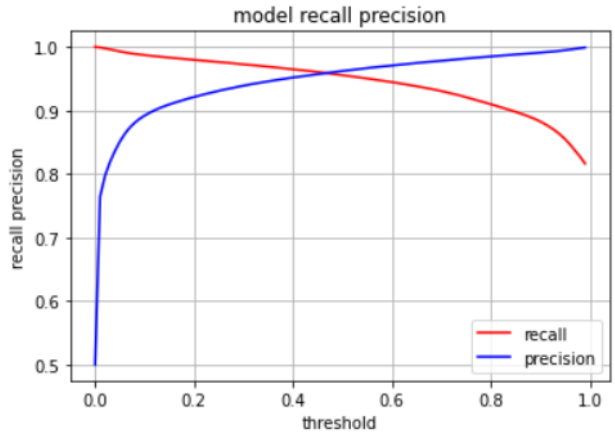
```
Accuracy: 0.965205
Precision: 0.521037
Recall: 0.926547
F1 score: 0.531486
Text(0.5, 1.0, 'model loss')
```



شکل ۸: **confusion matrix** برای حالت **oversampling**

(6

همانند شکل ۷ مقاله مدل را با ترشلد های مختلف می سنجیم. این ترشلد ها را برای recall و precision در نظر میگیریم. ترشلد شامل اعداد ۰ تا ۱ با فاصله ۰.۰۱ می شود. نتایج را در شکل زیر میبینیم.



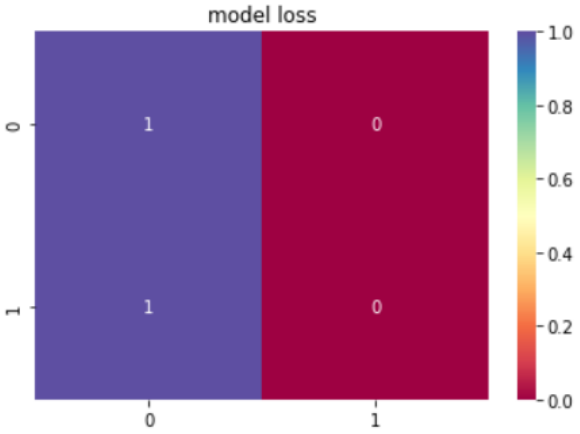
شکل ۹: نمودار مقدار **reall, precision** برحسب ترشلد برای حالت **oversampling**

به ازای ترشلد های مد نظر مقاله جدول زیر را میتوانیم داشته باشیم. میبینیم با افزایش ترشلد، recall کاهش می یابد و precision افزایش می یابد.

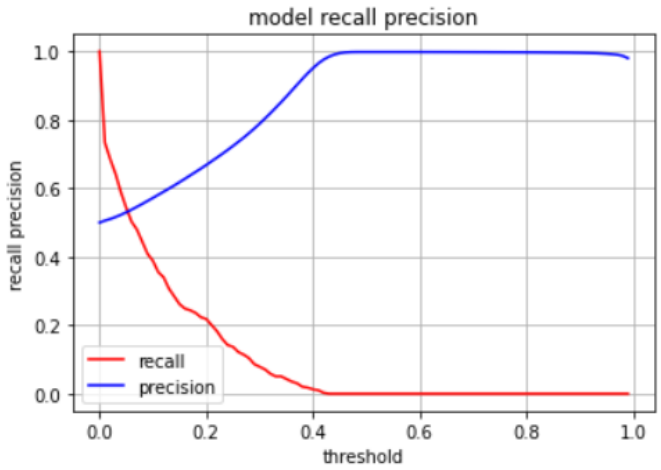
(7

حال مدل را بدون اینکه resample کنیم و از autoencoder عبور دهیم، اموزش میدهیم. به ازای ترشلد های مختلف نتایج زیر حاصل می شود.

```
Accuracy: 0.998280
Precision: 0.499140
Recall: 0.500000
F1 score: 0.499570
/usr/local/lib/python3.8/dist-packages/sklearn/
  _warn_prf(average, modifier, msg_start, len(r
Text(0.5, 1.0, 'model loss')
```



شکل ۱۰: **confusion matrix** برای حالت بدون **oversampling**



شکل ۱۱:نمودار مقدار **reall, precision** برحسب ترشلد برای حالت بدون **oversampling**

حال از مقایسه دو روش به این نتیجه میرسیم که هنگامی که مدل را بدون اینکه resample کنیم و از autoencoder عبور دهیم، اموزش میدهیم، مدل recall در حدود صفر میگیرد. دلیل این موضوع این است که تمامی داده های کلاس کوچکتر برچسب داده های کلاس بزرگتر را گرفته اند و با وجود precision زیاد، این مقدار قابلیت ارجاع ندارد و همه چیز را در نظر نگرفته است.

در جدول زیر مقایسه دقیق تری از دو روش بدون oversample و با oversample انجام میدهیم.

	Accuracy	Precision	Recall	F1 score
With oversample and autoencoder	0.965205	0.521037	0.926547	0.531489
Without oversample and autoencoder	0.99828	0.499140	0.5	0.49957

جدول ۱: مقایسه دو حالت با **oversampling** و بدون **oversampling**

با بررسی جدول بالا به این نتیجه میرسیم فرایند oversample کردن و autoencoder تاثیر به سزایی روی طبقه بندی داده ها داشته است. پس میتوان نتیجه گرفت زمانی که دیتا ها بالانس نیستند میتوان با کمک روش های مشابه از کلاس کوچکتر داده تولید کرد تا بتوان کلاس کوچکتر را به درستی تشخیص داد. هنگامی که داده های خام را طبقه بندی میکنیم، همه کلاس کوچکتر برچسب کلاس بزرگتر را میگیرند. به همین دلیل Recall = 0.5 بوده است اما بعد از فرایند oversample کردن و autoencoder

این مقدار به ۰.۹۲۶۵ رسیده است. در بخش های قبلی توضحی دادیم که هنگامی که کلاس ها بالانس نیستند استفاده از معیار دقت کافی نیست برای همین با Recall میفهمیم مدل پیشرفت خوبی داشته است. اگر به accuracy دقت کنیم میبینیم برای داده های خام عملکرد بهتری داشته ولی میدانیم زیاد بودن ان به دلیل این است که کلاس کوچک برچسب کلاس بزرگتر را گرفته است.

پاسخ ۳ - تشخیص کاراکتر نوری(Optical character recognition)

الف)

تفاوت دو مدل CNN و DCNN به صورت کلی در تعداد لایه ها می باشد.

شبکه ی DCNN برای اجرای الگوریتم deep learning نیاز به لایه های بیشتری داشته، برای همین عموما شبکه های عمیق دارای حدود ۳۰ تا ۱۰۰ لایه میباشند در صورتی که شبکه های CNN تعداد ۵ تا ۱۰ لایه دارند. پس به همین دلیل که تعداد لایه های آن بیشتر است، به آن Deep CNN می گویند.

ب)

الگوریتم Adam:

این بهنیه ساز جایگزینی برای stochastic gradient descent است که از ترکیب دو الگوریتم SGD و RMSP به دست آمده است.

این الگوریتم درواقع ممان اول و دوم را همزمان در نظر گرفته و وزن ها را با توجه به هر دوی آنها تغییر میدهد که موجب همگرایی سریع تر شده.

در این الگوریتم دو پارامتر برای تعیین نسبت هرکدام از ممان ها استفاده میشود که روابط آنها به شکل زیر است:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\left[\frac{\delta L}{\delta w_t}\right] \; v_t = \beta_2 v_{t-1} + (1-\beta_2)\left[\frac{\delta L}{\delta w_t}\right]^2$$

$$\widehat{m_t} = \frac{m_t}{1-\beta_1^t} \widehat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$w_{t+1} = w_t - \widehat{m_t}\left(\frac{\alpha}{\sqrt{\widehat{v_t}}+\varepsilon}\right)$$

شکل ۱۲: الگوریتم **Adam**

الگوریتم Adadelta:

الگوریتم Adadelta به صورت کلی مانند SGD الگوریتمی مبنی بر گرادیان است، با این تفاوت که با پنجره بندی روی گرادیان های به دست آمده میتواند learning rate را متناسب با آنها تغییر درهد.

ورژن اصلی این الگوریتم Adagradنام دارد که در آن قابلیت تعیین LR اولیه وجود نداشت ولی در Adadelta اضافه شده است.

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

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

شکل ۱۳: الگوریتم **Adadelta**

الگوریتم Momentum:

این الگوریتم بخشی را به الگوریتم SGD اضافه میکند که به عنوان momentum یا ممان شناخته میشود. ممان درواقع مجموع گرادیان های به دست آمده در مراحل قبل میباشد که با ضریبی موسوم به ضریب ممان به فرمولاسیون SGD اضاف میشود و وزن های مرحله ی بعد را می سازد.

$$\nu_j \leftarrow \eta * \nu_j - \alpha * \nabla_w \sum_1^m L_m(w)$$
$$\omega_j \leftarrow \nu_j + \omega_j$$

شکل ۱۴: الگوریتم **Momentum**

ج)

در بخش پیش پردازش به دلیل اینکه ابعاد تصاویر HODA متفاوت می باشد نیاز به reshape کردن تصاویر به ابعاد مورد نظر(در اینجا 40*40) میباشد. همچنین برای از بین بردن shade های مختلف، تصاویر با یک فیلتر median به صورت باینری در می آیند.

معماری ۴ بلوک به صورت سری را شامل میشود.

در بلوک اول ۳۲ feature map به وجود می آید که سایز هر کرنل آنها ۳*۳ می باید ، همچنین بعد از هر لایه کانولوشن یک لایه فعال ساز relu وجود دارد که فیچر ها را استخراج کند. بعد از این لایه نیز batch normalization انجام میشود.

۳ بلوک دیگر نیز از اتصال این لایه ها به یکدیگر ساخته میشوند و معماری نهایی را ایجاد میکنند.

```
#block 1
model.add(SeparableConv2D(64, (3, 3), padding="same", input_shape=(40,40,1)))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=1))
model.add(MaxPooling2D(pool_size=(3, 3),strides = (2,2)))
model.add(Dropout(0.1))

#block 2
model.add(SeparableConv2D(128, (3, 3), padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=1))
model.add(SeparableConv2D(128, (3, 3), padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=1))
model.add(MaxPooling2D(pool_size=(3, 3),strides = (2,2)))
model.add(Dropout(0.2))

#block 3
model.add(SeparableConv2D(256, (3, 3), padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=1))
model.add(SeparableConv2D(256, (3, 3), padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=1))
model.add(MaxPooling2D(pool_size=(3, 3),strides = (2,2)))
model.add(Dropout(0.3))

#block 4
model.add(SeparableConv2D(512, (3, 3), padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=1))
model.add(SeparableConv2D(512, (3, 3), padding="same"))
model.add(BatchNormalization(axis=1))
model.add(MaxPooling2D(pool_size=(3, 3),strides = (2,2)))
model.add(Dropout(0.4))

#classifier
model.add(Flatten())
model.add(Dense(1024, activation="relu"))
model.add(BatchNormalization(axis=1))
model.add(Dropout(0.5))
model.add(Dense(10,activation="softmax"))
```

شکل ۱۵: پیاده سازی لایه های **DCNN**

در این مقاله برای کاهش ابعاد از لایه های pooling بعد از هر بلوک کانولوشن استفاده شده و برای جلوگیری از overfitting بعد از هر لایه pooling یک لایه dropout قرار گرفته است. Dropout سبب می شود تا مدل وابسته به یک مسیر خاص نشود و هر بار به صورت رندوم یک مسیر را در اموزش قطع میکند.

د)

روش Adam:

خروجی های مدل

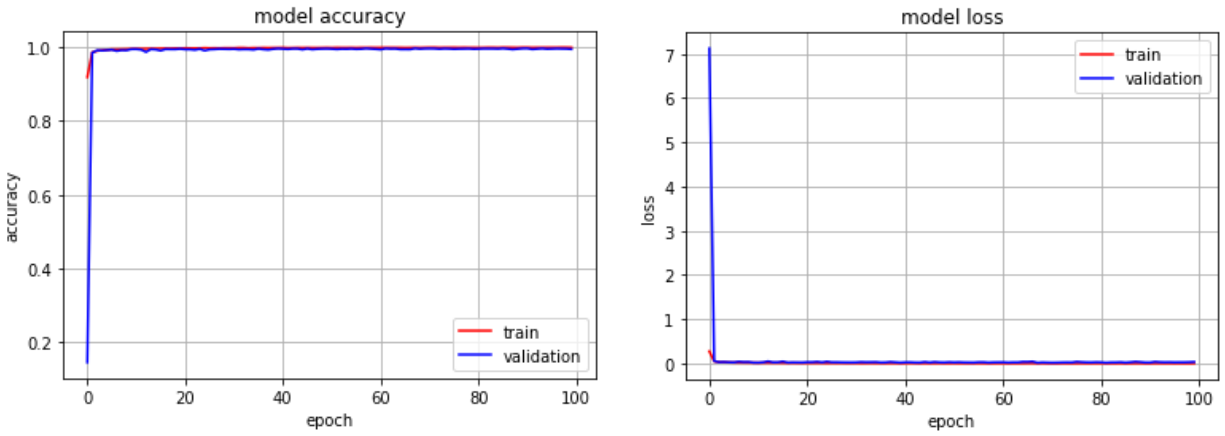
```
test loss = 0.035012
test accuracy = 0.995100
Accuracy: 0.995100
Precision: 0.995170
Recall: 0.995100
F1 score: 0.995107
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      2000
           1       1.00      1.00      1.00      2000
           2       0.99      0.98      0.99      2000
           3       0.97      1.00      0.98      2000
           4       1.00      0.98      0.99      2000
           5       1.00      1.00      1.00      2000
           6       1.00      1.00      1.00      2000
           7       1.00      1.00      1.00      2000
           8       1.00      1.00      1.00      2000
           9       1.00      1.00      1.00      2000

    accuracy                           1.00     20000
   macro avg       1.00      1.00      1.00     20000
weighted avg       1.00      1.00      1.00     20000
```

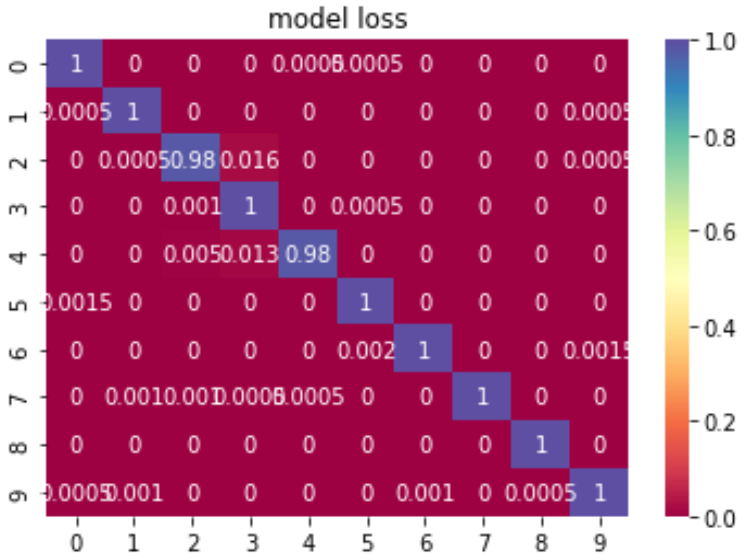
شکل ۱۶: ارزیابی **Adam**

نمودار accuracy و loss



شکل**17** و ۱۷: نمودار **loss** و **accuracy** برای **Adam**

ماتریس confusion:



شکل ۱۸: **confusion matrix** برای **Adam**

روش Adadelta:

خروجی های مدل:

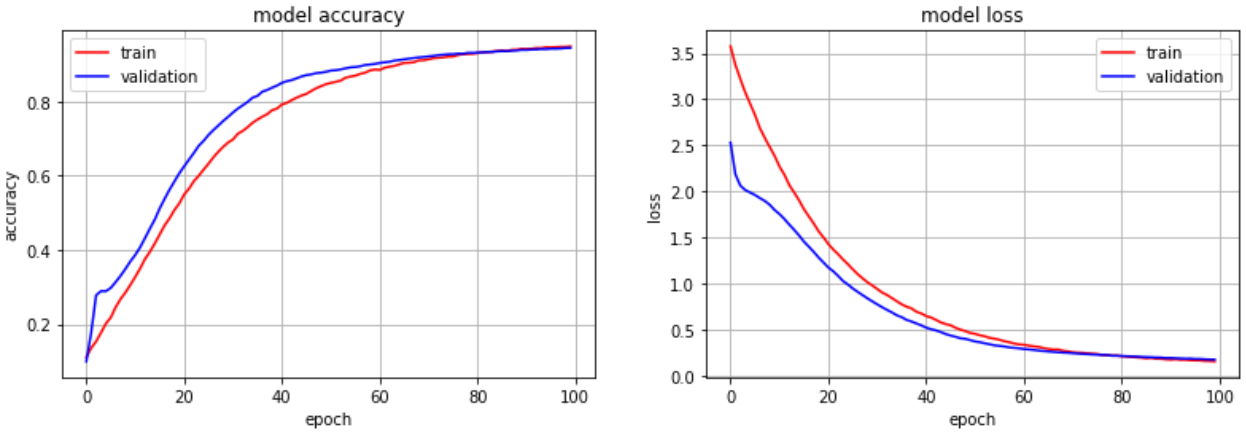
```
test loss = 0.182888
test accuracy = 0.943950
Accuracy: 0.943950
Precision: 0.947804
Recall: 0.943950
F1 score: 0.944315
              precision    recall  f1-score   support

           0       0.94      0.99      0.97      2000
           1       0.95      0.99      0.97      2000
           2       0.82      0.95      0.88      2000
           3       0.95      0.92      0.93      2000
           4       0.90      0.97      0.93      2000
           5       0.99      0.93      0.96      2000
           6       0.98      0.87      0.92      2000
           7       1.00      0.90      0.94      2000
           8       0.99      0.98      0.99      2000
           9       0.97      0.93      0.95      2000

    accuracy                           0.94     20000
   macro avg       0.95      0.94      0.94     20000
weighted avg       0.95      0.94      0.94     20000
```

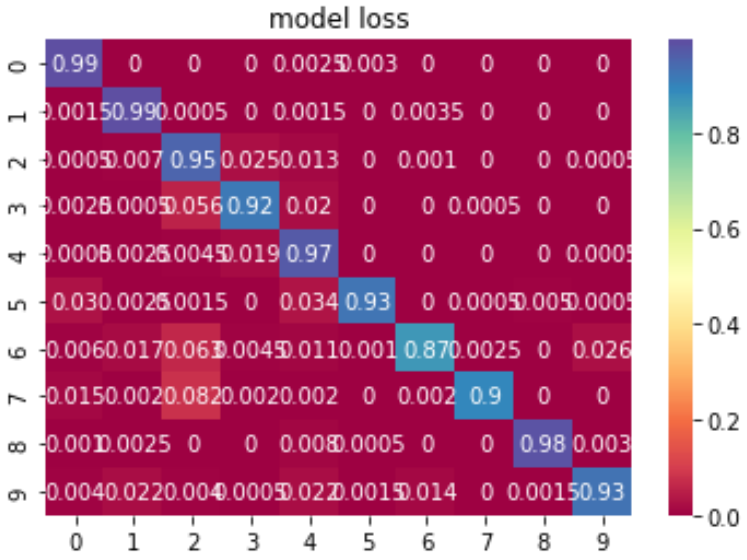
شکل ۱۹: ارزیابی **Adadelta**

نمودار accuracy و loss



شکل **21** و ۲۰: نمودار **loss** و **accuracy** برای **Adadelta**

ماتریس confusion:



شکل ۲۱: **confusion matrix** برای **Adadelta**

روش Momentum:

خروجی های مدل:

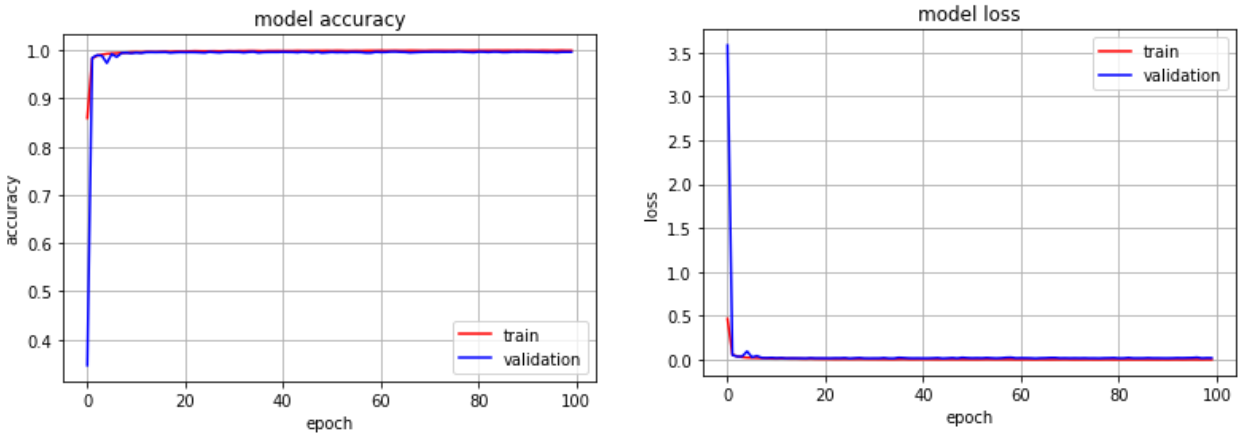
```
test loss = 0.018995
test accuracy = 0.996400
Accuracy: 0.996400
Precision: 0.996405
Recall: 0.996400
F1 score: 0.996399
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      2000
           1       1.00      1.00      1.00      2000
           2       0.99      0.99      0.99      2000
           3       0.99      0.99      0.99      2000
           4       0.99      1.00      0.99      2000
           5       1.00      1.00      1.00      2000
           6       1.00      1.00      1.00      2000
           7       1.00      1.00      1.00      2000
           8       1.00      1.00      1.00      2000
           9       1.00      1.00      1.00      2000

    accuracy                           1.00     20000
   macro avg       1.00      1.00      1.00     20000
weighted avg       1.00      1.00      1.00     20000
```

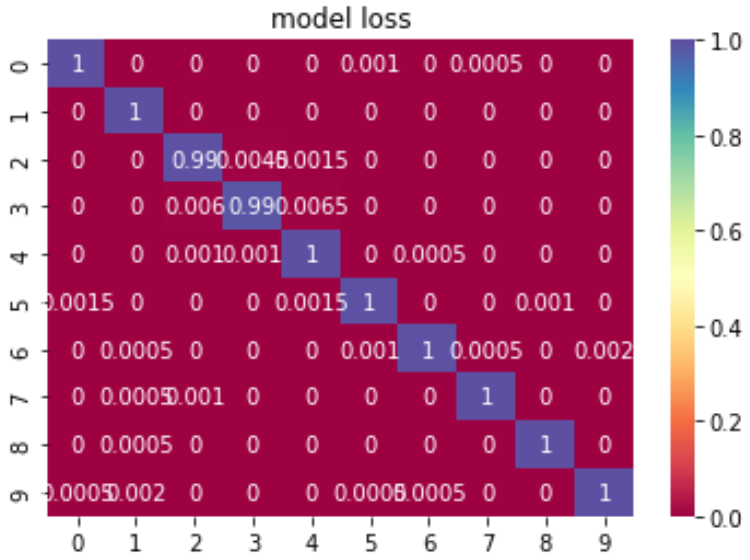
شکل ۲۲: ارزیابی **Momentum**

نمودار accuracy و loss



شکل **25** و ۲۳: نمودار **loss** و **accuracy** برای **Momentum**

ماتریس confusion:



شکل ۲۴: **confusion matrix** برای **Momentum**

تحلیل خروجی ها:

	Accuracy	Precision	Recall	F1 score	loss
Adam	0.9951	0.995170	0.9951	0.995107	0.035012
Adadelta	0.943950	0.947804	0.943950	0.944315	0.18288
Momentum	0.9964	0.996405	0.9964	0.996399	0.018995

جدول ۲: تحلیل خروجی هر سه بهینه ساز

طبق نمودار ها و همچنین مقدار accuracy, loss, precision, fl score میتوان نتیجه گرفت که دو روش Adam و Momentum عملکرد بهتری از adadelta دارن. همچنین از نمودار ها مشخص است که روش adadelta تغییرات کند تری دارد و دیر تر همگرا میشود.

از confusion matix نیز میتوان این نتیجه را گرفت که هرکدام از الگوریتم های adam و Momentum در قسمت های متفاوتی خطا دارند. همچنین خطایadadelta به میزان قابل توجهی در کلاس های مختلف از آنها بیشتر میباشد. در حالت کلی در هر سه روش بیشترین خطا مربوط به عدد ۰ و ۵ به دلیل شباهت زیاد آنها بوده است.

۵)

با توجه به اینکه مقاله در چند بلوک مجزا معماری را طراحی کرده و در هر بلوک از لایه های batch normalization و dropout برای جلوگیری از overfitting استفاده شده است بهتر از معماری های تک بلوکه عمل کرده و بنابراین بهترین معماری همان معماری مقاله است.

```
#block 1
model.add(SeparableConv2D(64, (3, 3), padding="same", input_shape=(40,40,1)))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=1))
model.add(MaxPooling2D(pool_size=(3, 3),strides = (2,2)))
model.add(Dropout(0.1))

#block 2
model.add(SeparableConv2D(128, (3, 3), padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=1))
model.add(SeparableConv2D(128, (3, 3), padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=1))
model.add(MaxPooling2D(pool_size=(3, 3),strides = (2,2)))
model.add(Dropout(0.2))

#block 3
model.add(SeparableConv2D(256, (3, 3), padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=1))
model.add(SeparableConv2D(256, (3, 3), padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=1))
model.add(MaxPooling2D(pool_size=(3, 3),strides = (2,2)))
model.add(Dropout(0.3))

#block 4
model.add(SeparableConv2D(512, (3, 3), padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=1))
model.add(SeparableConv2D(512, (3, 3), padding="same"))
model.add(BatchNormalization(axis=1))
model.add(MaxPooling2D(pool_size=(3, 3),strides = (2,2)))
model.add(Dropout(0.4))

#classifier
model.add(Flatten())
model.add(Dense(1024, activation="relu"))
model.add(BatchNormalization(axis=1))
model.add(Dropout(0.5))
model.add(Dense(10,activation="softmax"))
```

جدول ۳: معماری بهترین شبکه

بهترین پارامتر نیز Momentum با ضریب ۰.۹ خواهد بود که با epoch=100 و batch_size=128 ران شده است.