به نام خدا

گزارش کار تمرین عملی شبکه ی عصبی

درس یادگیری ماشین

استاد: دکتر سیدین

دانشجو: ریحانه آهنی ۹۸۲۳۰۰۹

سوال اول

هدف این تمرین پیش بینی هزینه ی اقامت در هتل به ازای ویژگی های مختلف می باشد.

بخش الف

در بخش الف از ما خواسته است که بر اساس تمامی ویژگی ها به عنوان ورودی استفاده کنیم و یک شبکه ی عصبی چند لایه طراحی کنیم که بتواند هزینه اقامت را پیش بینی کند. هزینه ی اقامت ستون ADR می باشد. بنابراین ADR چیزی است که ما میخواهیم پیش بینی کنیم.

ویژگی های categorical عبارت اند از :

```
categorical = ['IsCanceled', 'ArrivalDateYear', 'ArrivalDateMonth', 'Meal', 'Country', 'MarketSegment', 'DistributionChannel', 'IsRepeatedGuest', 'ReservedRoomType', 'AssignedRoomType', 'DepositType', 'CustomerType', 'ReservationStatus']
```

با بررسی این ویژگی ها بر اساس اینکه آیا string هستند یا یک مقداد عددی هستند که مرتبا تکرار میشوند (به عنوان مثال iscanceled فقط دو مقدار ۰ با ۱ را دارد که اگرچه که یک عدد است و رشته نیست اما از نوع داده های کتگوریکال است و یا arrivalyear اگرچه عدادی مثل ۲۰۱۸ و ... هستند اما باید یا ان ها همانند داده هایی از نوع کتگوریکال رفتار کرد.) پس باید ستون هایی که این نوع داده ها را دارند هم در تست و هم در آموزش به کتگوریکال تبدیل کنیم.
برای این کار با استفاده از تابع pd.read_csv داده های یادگیری و تست را لود کرد. سپس با استفاده از لیست ستون های categorical که جمع آوری کرده بودیم، این داده ها را به نوع category تبدیل می کنیم. سپس با استفاده از تابع apply برای همه این ستون ها از عدد به جای اسم و مقدار آن استفاده می کنیم.

```
test[categorical] = test[categorical].astype('category')
test[categorical] = test[categorical].apply(lambda x: x.cat.codes)
```

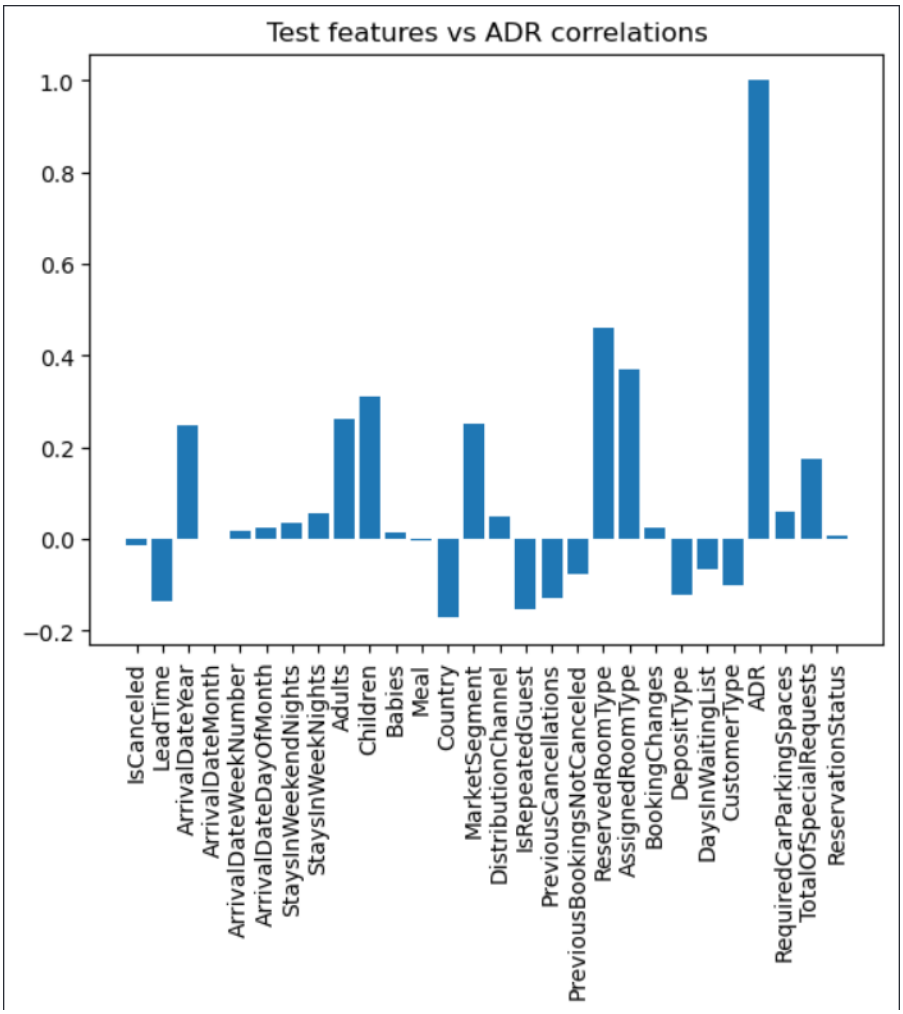
چون اکنون در مراحل آماده سازی داده ها قبل از اعمال مدل هستیم بهتر است تغییراتی که در بخش «د» مدنظر میباشد هم اعمال کنیم به این منظور باید ارتباط بین داده ها را پیدا کینم .
برای این کار از تابع corrwith استفاده میکنیم.

```
features = test.corrwith(test['ADR'])
plt.figure()
plt.title('Test features vs ADR correlations')
plt.bar(range(len(features)), features)
plt.xticks(range(len(features)), tuple(features.index), rotation=90)
plt.plot()
```

این تابع از توابع اماده و موجود در pandas هستند که بر اساس الگوریتم های زیر ارتباط داده ها را محاسبه میکند:

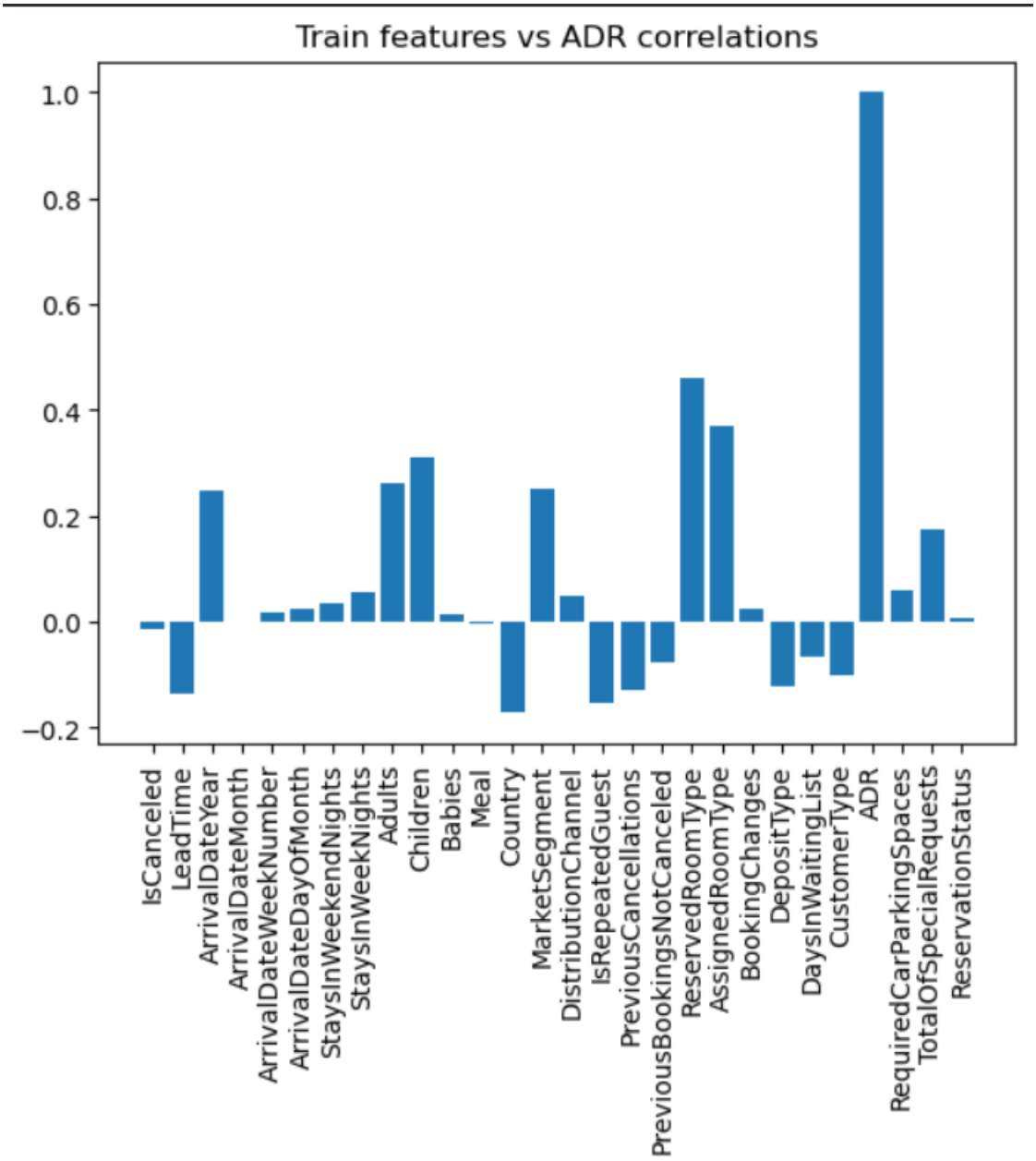
```
* pearson : standard correlation coefficient
* kendall : Kendall Tau correlation coefficient
* spearman : Spearman rank correlation
* callable: callable with input two 1d ndarrays and returning a float.
```

درمرحله ی بعد باید میزان ارتباط داده ها با ستون مد نظر یعنی ADR را رسم کنیم.



همانطور که از نمودار مشهود است میبینیم که برخی از داده ها همبستگی منفی دارند که این یعنی با کاهش این ویژگی ها ADR زیادتر خواهد شد. و کاملا واضح است که زیادترین همبستگی مربوط به خود ADR میباشد.

نمودار مربوط به داده ها آموزشی هم به صورت زیر است :



بعد از بدست آوردن این نمودار ها به تفسیر آن ها میپردازیم . ویژگی iscanceled در آموزش و تست اثری کاملا متفاوت و عکس دارد (که این مورد مهمی است که باید در پرووسس کردا داد ها مد نظر قرار دهیم)

در مرحله ی بعدی ما باید داده ها را اسکیل کنیم به روش زیر داده ها از ۰ تا ۱ اسکیل میکنیم .
برای انجام این کار از تابع اسکیلر sklearn استفاده میکنیم که یک کلاس داره به نام MinMaxScaler که در این کلاس متدی به نام fit_transform وجود دارد . که بخش fit آن پارامتر های میانگین و انحراف معیار را محاسبه میکند و بعد با استفاده از transform داده ها را به رنج ۰ تا ۱ اسکیل میکند. یکی از مشکلات کلاس MinMaxScaler نداشتن قابلیت دریافت ورودی به صورت ماتریس تک بعدی است. برای حل این مشکل با استفاده از تابع reshape شکل ماتریس را از یک ماتریس تک بعدی با اندازه n، به یک ماتریس دو بعدی با اندازه ۱ در n تبدیل می کنیم.

```
x_scaler = MinMaxScaler()
y_scaler = MinMaxScaler()

train_x = x_scaler.fit_transform(train[train.columns.drop('ADR')])
train_y = y_scaler.fit_transform(train['ADR'].to_numpy().reshape(-
1, 1))

test_x = x_scaler.fit_transform(test[train.columns.drop('ADR')])
test_y = y_scaler.fit_transform(test['ADR'].to_numpy().reshape(-1, 1))
```

یک سوال مهم که ممکن است در اینجا مطرح شود دلیل استفاده از MinMaxScaler است .
پاسخ این سوال این است که MinMaxScaler مقادیر مینیمم ماکزیمم را ذخیره میکند و در مراحل بعدی که نیاز است داده ها از رنج اسکیل شده به رنج اصلی خود برگردند (inverse scale) این کار به راحتی امکان پذیر می باشد.

پس از انجام این مراحل به سراغ تعریف مدل میرویم .

برای لایه ی اول ۳۰ نورون در نظر میگیریم و از تابع فعال ساز ReLU استفاده میکنیم.

برای لایه ی بعدی ۱۰۲۴ نوررون در نظر میگیریم و از تابع فعال ساز ReLU استفاده میکنیم و از یک drop out با مقدار ۰.۱ درصد استفاده میکنیم. استفاده از drop out به این دلیل است که ما در validation امکان یادگیری داشته باشیم در واقع این لایه به عنوان یک regularization کار خواهد کرد و از اورفیت شدن جلوگیری خواهد کرد و اگر از drop out استفاده نکنیم مدل تقریبا به سمت حفظ داده ها خواهد رفت و یادگیری صورت نمیگیرد.

لایه بعدی هم دارای ۱۰۲۴ نورون میباشد و مانند لایه قبل از ReLU و drop out ۰.۱ درصد استفاده میکند.

```
model = Sequential()
model.add(Dense(30, input_shape=(train_x.shape[1], ), activation='relu
'))
model.add(Dense(1024, activation='relu'))
model.add(Dropout(0.1))
model.add(Dense(1024, activation='relu'))
model.add(Dropout(0.1))
model.add(Dense(1, activation='tanh'))
```

از طرفی چون پایه ی اصلی این کار رگرسیون میباشد در لایه آخر از یک عدد نورون با تابع فعال ساز tanh استفاده میکنیم. یکی ار دلایل استفاده از tanh در اینجا به جای ReLU خاصیت clampingمیباشد.

بعد از تعریف مدل از دستوری به نام model_checkpoint_callback استفاده میکنیم به این منظور که بعد از هر ایپک بهترین مدل را ذخیره کند . این دستور حتی در زمانی که انجام فرایند به مشکلاتی مثل خاموش شدن و یا هنگ کردن برخورد کرد کارایی دارد به این صورت که وزن ها را ذخیره میکند با این کار در هنگام توقف در حین انجام کار نیازی به دوباره از اول اجرا کردن نمیباشد.

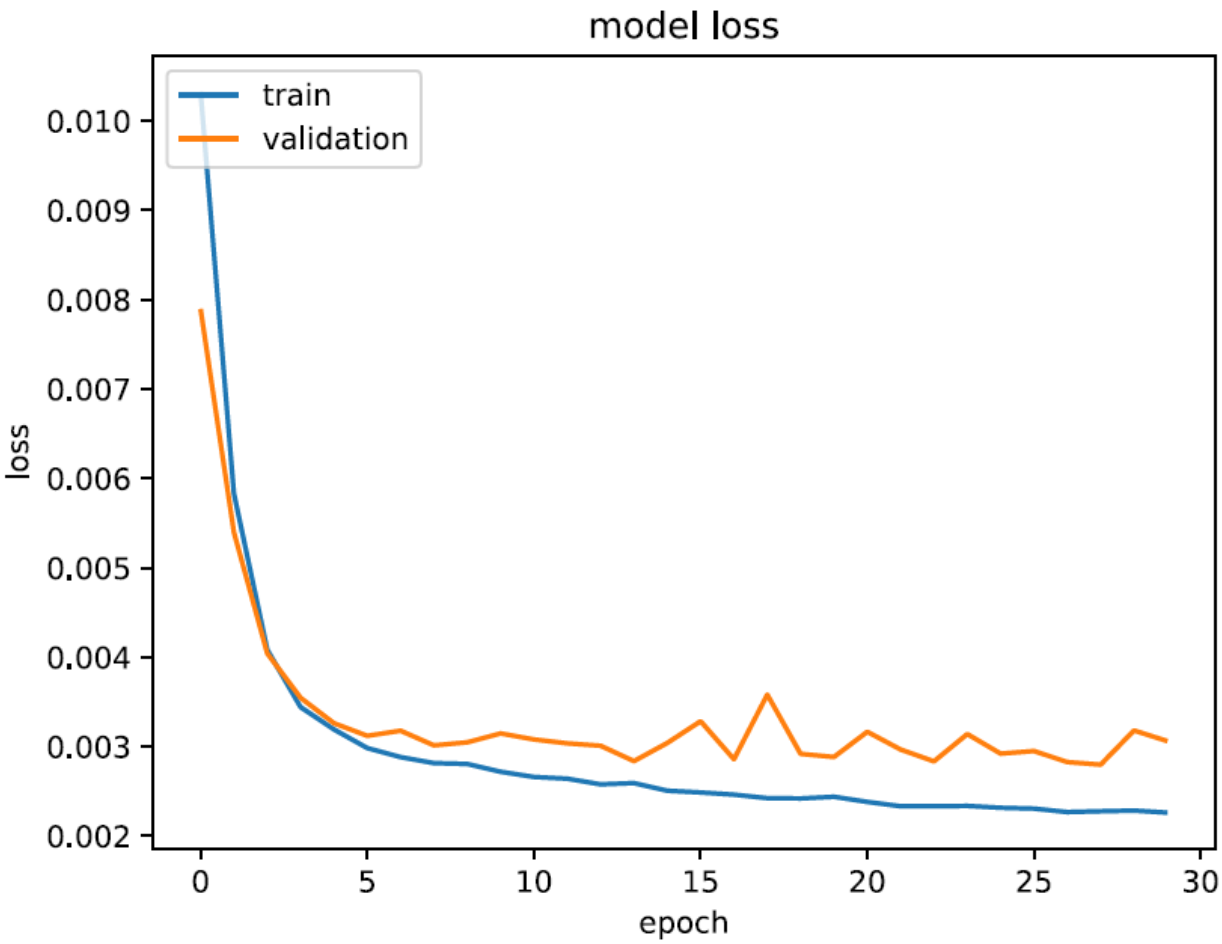
```
model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath='./backups/checkpoint',
    save_weights_only=True,
    monitor='val_loss',
    mode='min',
    save_best_only=True)
```

پس از این مراحل مدل را compile میکنیم .

چون در صورت مساله اشاره به این موضوع نشده بود که از چه optimizer ای میتوان استفاده کرد از adam optimizer استفاده میکنیم. و برای میزان loss از MSE استفاده میکنیم. استفاده از MSE به جای MAE به این دلیل است که اولی به دلبل به توان دو رساندن میزان خطا شدید تر برخورد میکند.

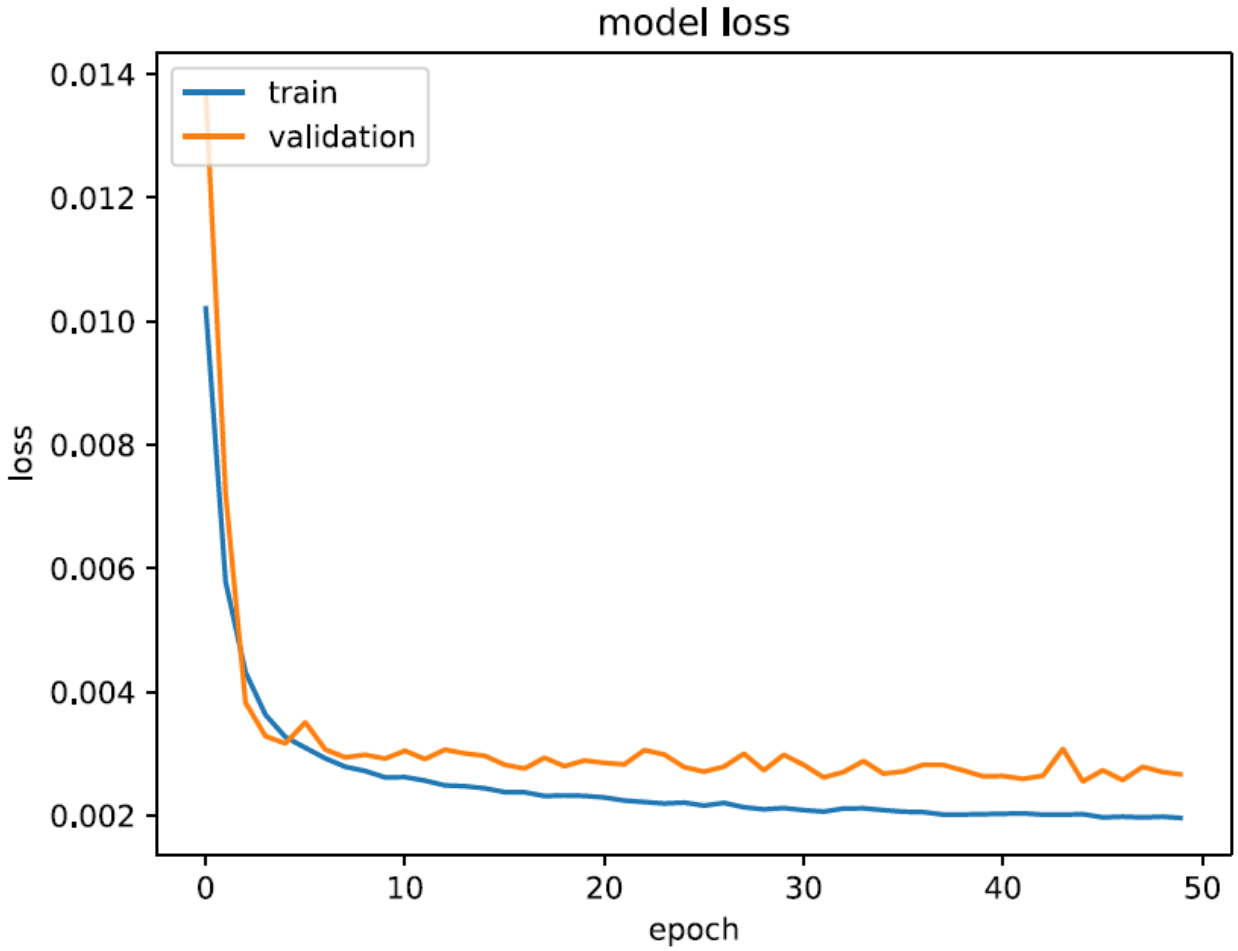
```
model.compile(loss='mse', optimizer='adam')
history = model.fit(train_x, train_y, epochs=30, batch_size=128, valid
ation_split=0.2, callbacks=[model_checkpoint_callback])
```

بعد از آموزش شبکه پس از ۳۰ ایپک و تست آن منحنی زیر نتیجه میشود :

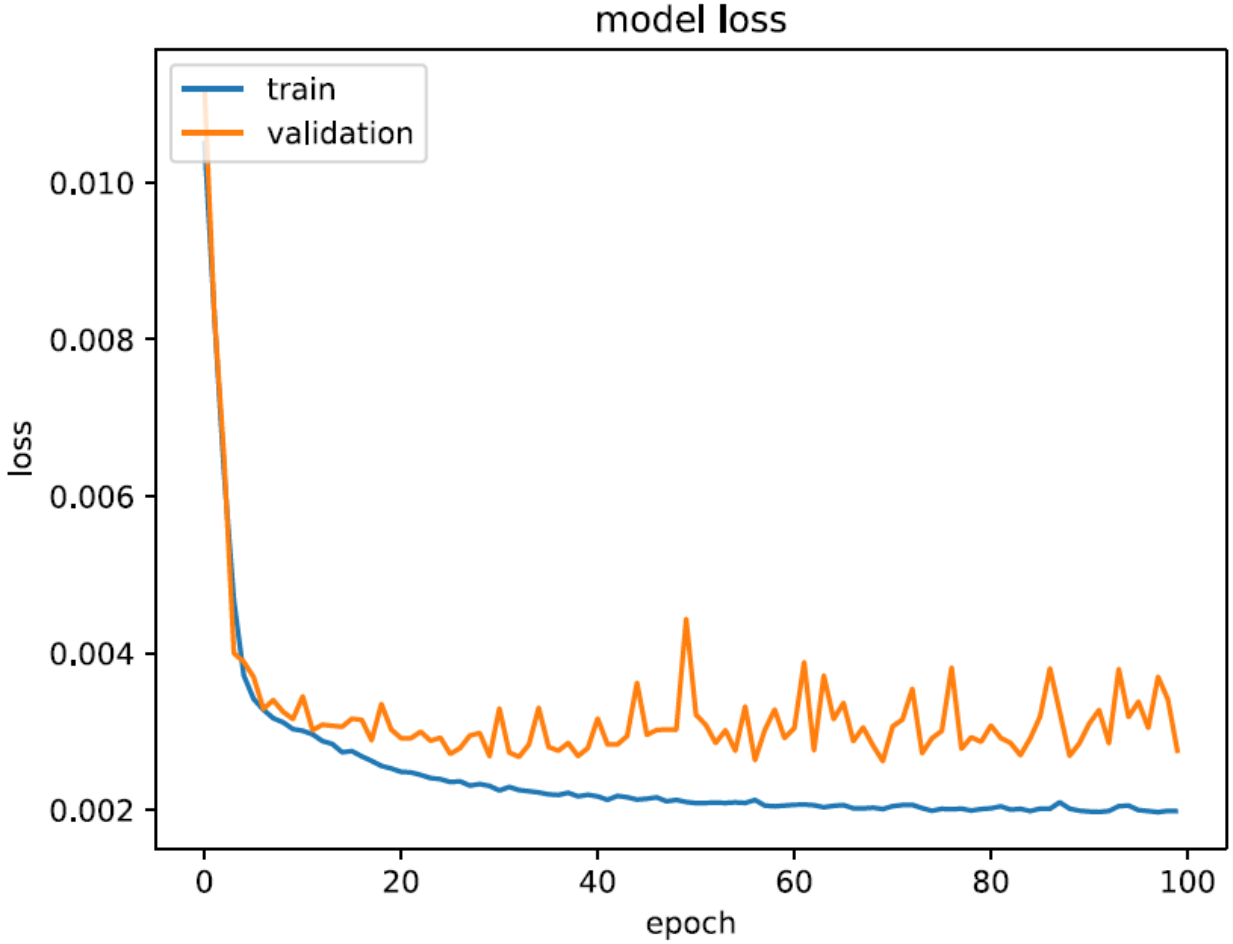


مشاهده میشود که در ایپک های اولیه ولیدشین از آموزشی ها بهتر است و ما به دلیل استفاده از drop out چنین چیزی را انتظار داشتیم. اما همانطور که مشهود است با گذشت زمان و انجام ایپک های بیشتر مقدار خطا روی آموزشی ها کم و روی ولیدشین زیاد تر خواهد شد و این نقطه محل رخداد اورفیت میباشد.

با تست کزدن مدل پس از ۵۰ ایپک شکل زیر حاصل خواهد شد:



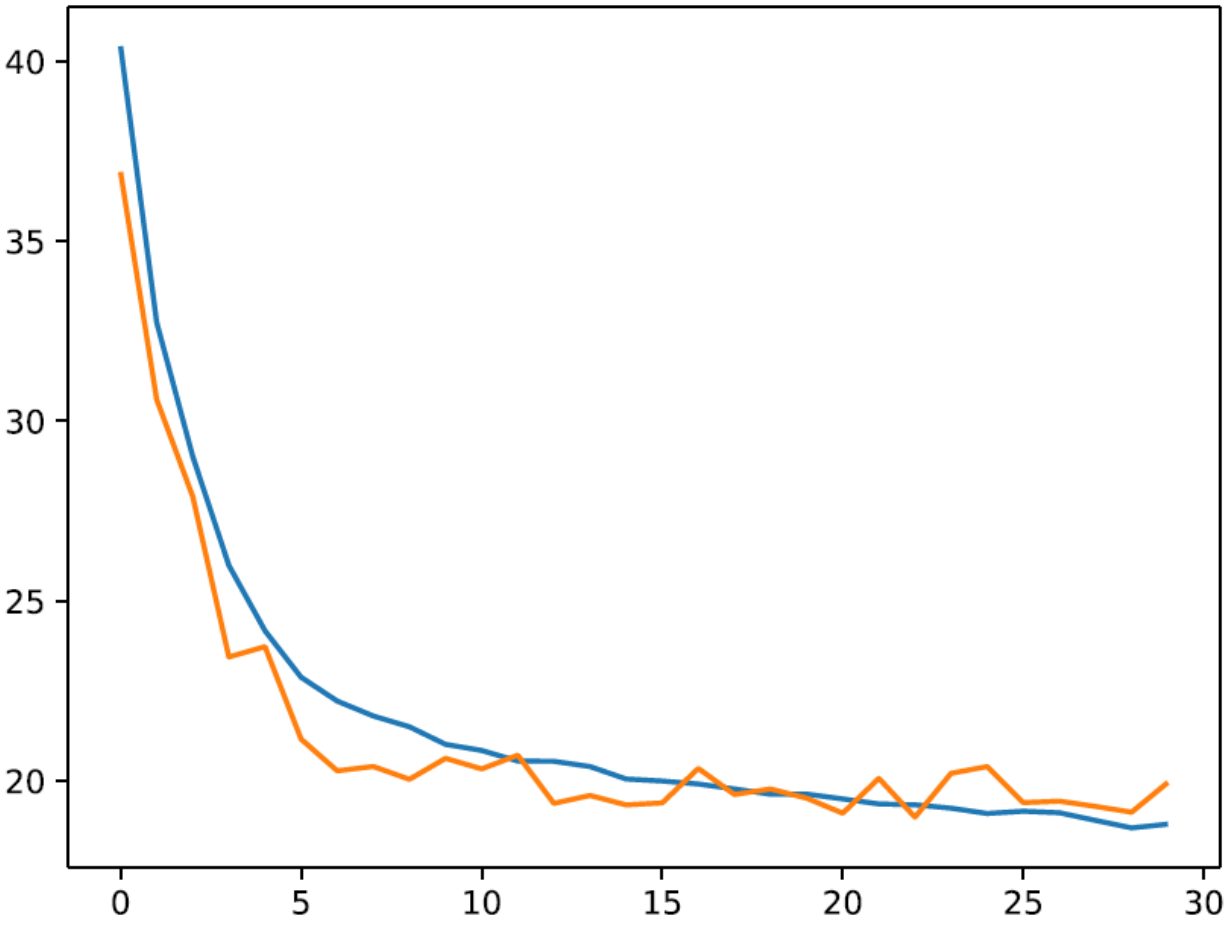
با تست کردن مدل با ۱۰۰ ایپک مشاهده میشود که نوسانات منحنی ولیدشن زیادتر میشود.



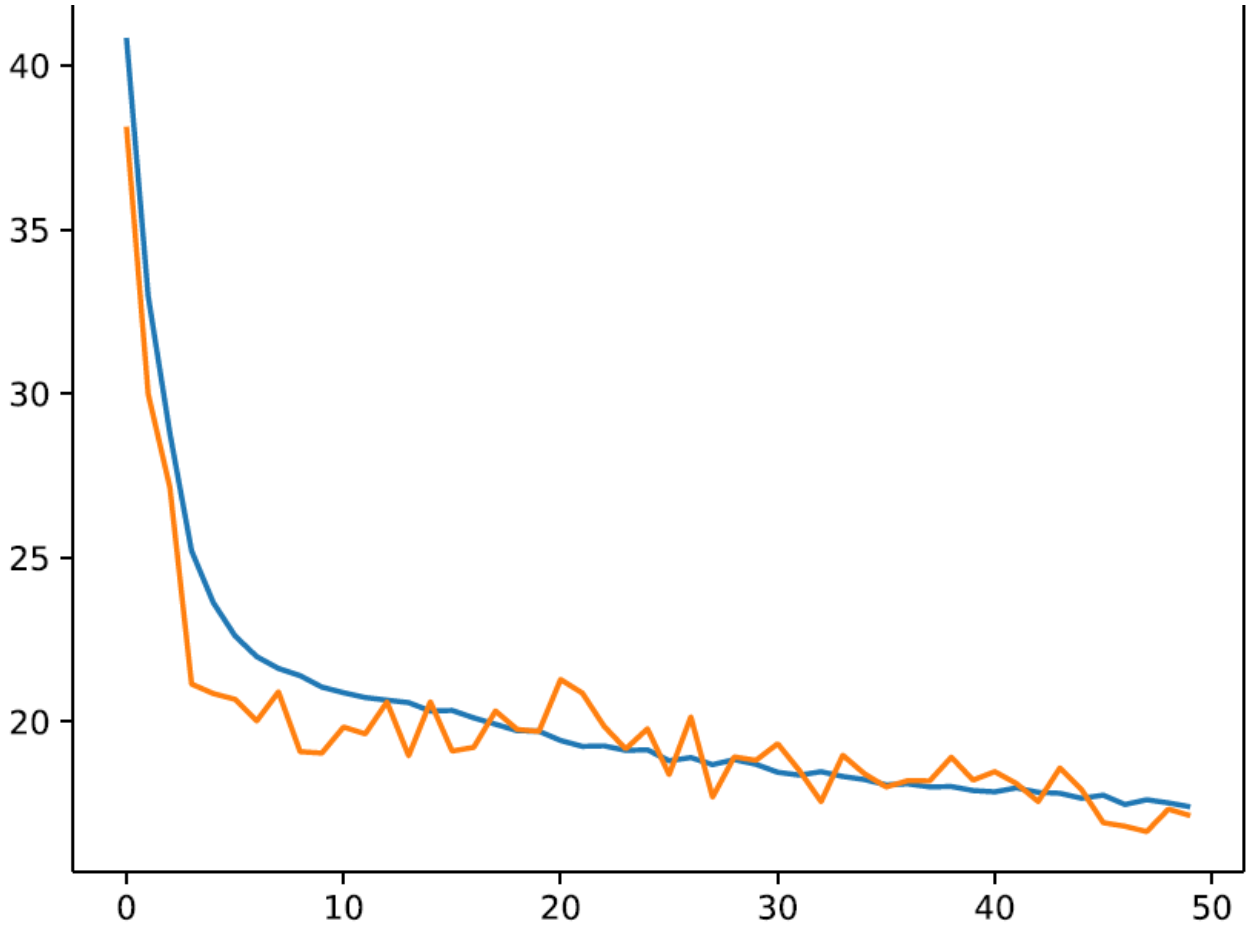
بخش ب

در این بخش ار ما خواسته است که به جای استفاده از MSE از MAE استفاده کنیم و منحتی ها را رسم کنیم.

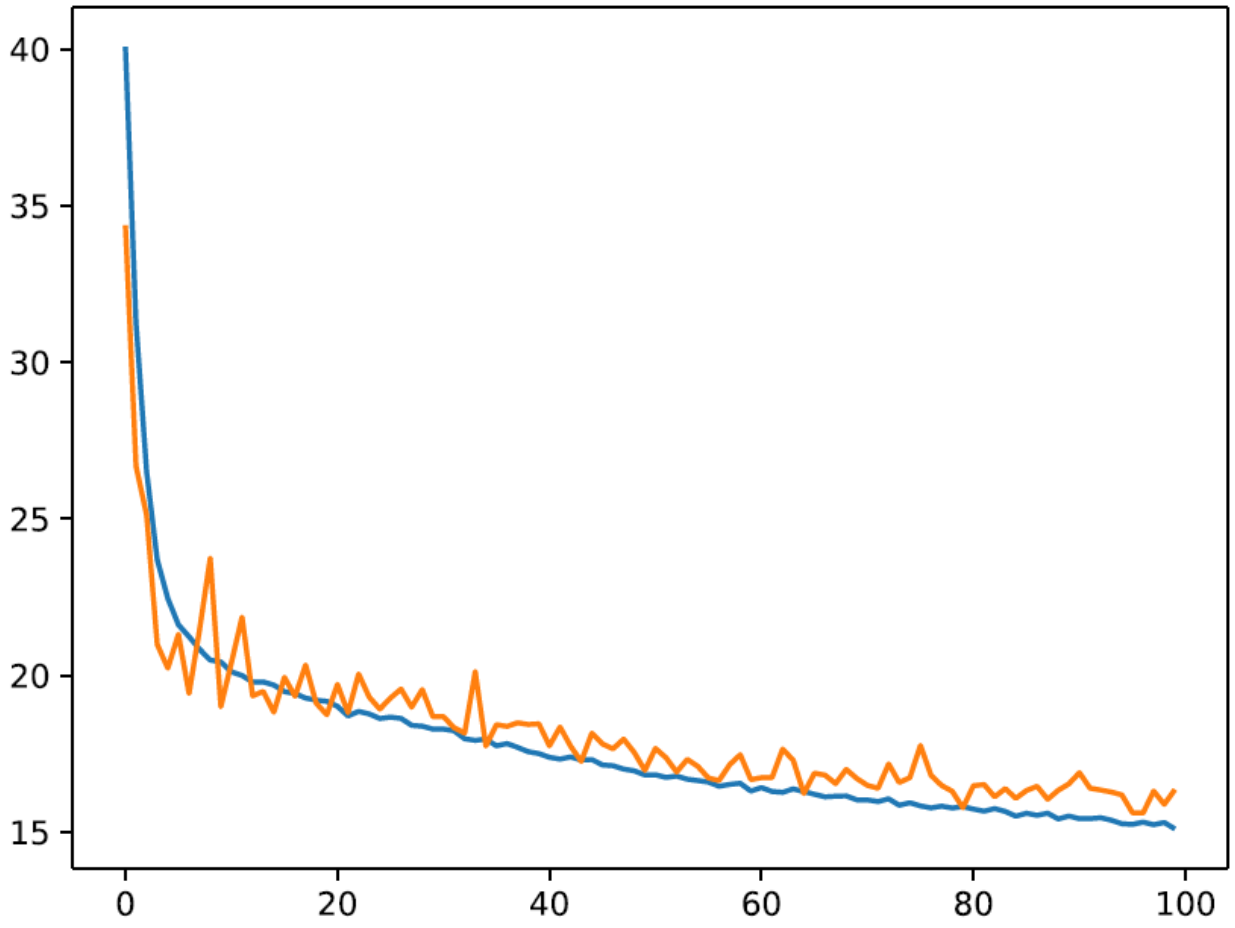
دراولین مرحله که در ۳۰ ایپک تست انجام میشود مشاهده میکنیم که تغییرات فاحشی بین دو نمودار در حالت ۳۰ ایپک در MSE و MAE وجود ندارد.



در حالت ۵۰ ایپک مشاهده میشود که Loss مربوط به ولیدشن در ابتدا کم است. اما همانند مورد قبلی با گذشت زمان میزان شبکه با مشکل overfit مواجه شده و Loss افزایش می یابد.



و بعد از ۱۰۰ ایپک شاهد نوسانی بودن واورفیت شدن می باشیم.

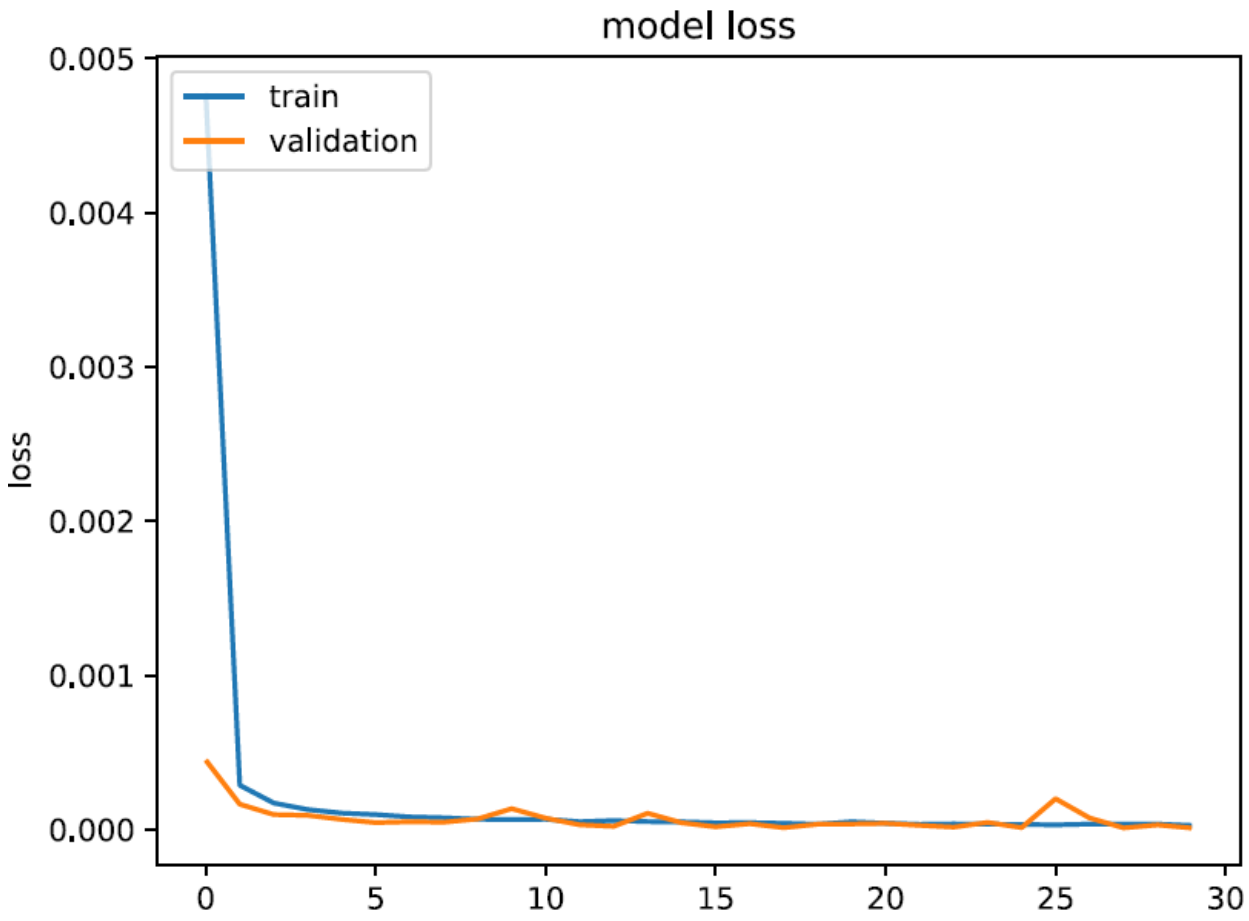


بخش د

در مراحل بعدی برای حل مشکلات (اورفیت شدن و عدم کاهش validation loss) از feature selection استفاده میشود و نتایج پس از ۳۰ ایپک را بررسی میکنیم.

مشاهده می شود که با انجام feature selection خطای ولیدشن بسیار کاهش یافته (به عددی در حدود 8e-6 میرسیم) که این نشان میدهد که انتخاب ویژگی های مرتبط بسیار مهم است و هر چه که تعداد فیچر ها کمتر باشند نتایج معمولا بهتر خواهد بود. (این لفظ کم بودن در هر دیتاست و با توجه به کاربرد میتواند معانی و تعداد مختلفی داشته باشد)

نمودار پس از feature selection



همچین باید به این نکته اشاره کنیم که بهترین مدل در فایل model.h5 ذخیره میشود .

```
model.save('model.h5')
```

بخش ج

حال در این مرحله روی داده های تست پیش بینی انجام میدهیم .

همچین همان طور که در بخش های قبل هم اشاره شد پس از پیشبینی کردن باید از inverse transformation استفاده کنیم و داده ها به مقادیر واقعی خودشان map کنیم.

```
y_pred = y_scaler.inverse_transform(y_pred)
test_y = y_scaler.inverse_transform(test_y)

result = pd.DataFrame({'pred': y_pred.flatten(), 'test': test_y.flatte
n()})
result['diff'] = result['pred'] - result['test']
result.to_csv('results_no_feature_selection.csv', index=False)
```

ستون اول که pred می باشد، مقادیری است که شبکه به عنوان خروجی داده است. ستون دوم که test می باشد مقدار اصلی آن sample می باشد و diff میزان اختلاف میان pred و test می باشد. این دو فایل به نام های results.csv و results_no_feature_selection.csv در فایل پروژه قابل دسترس هستند.

در نهایت باید به این اشاره کنیم تفاوت loss در حالت بدون feature selection و با feature selection عددی در رنج میلیون میباشد که این اهمیت feature selection را نشان میدهد.

برای استخراج ویژگی ها راه های متعددی وجود دارد اما راه حلی که ما از ان استفاده کردیم corrolation matrix می باشد . یکی دیگر از راه های متداول استفاده از سایر نرم افزار ها مثل Gluviz و Orange 3 میباشد که میتوان دیتاست درآن ها لود کرد و ارتباط بین داده ها را یا هم مقایسه و پیدا کرد.

*در انجام این تمرین از batch size = 128 استفاده شده است.

سوال دو

در این بخش باید شبکه ای برای طبقه بندی مجموعه داده CIFAR-10 طراحی کرد.

مجموعه داده CIFAR-10 مجموعه ای از تصاویر است که معمولاً برای آموزش الگوریتم های یادگیری ماشین و بینایی رایانه استفاده می شود.

مجموعه داده CIFAR-10 مجموعه تصاویری است که معمولاً برای آموزش الگوریتم های یادگیری ماشین و بینایی رایانه استفاده می شود. این یکی از پرکاربردترین مجموعه های داده برای تحقیقات یادگیری ماشین است. مجموعه داده CIFAR-10 شامل ۶۰،۰۰۰ تصویر رنگی ۳۲×۳۲ در ۱۰ کلاس مختلف است.

10 کلاس مختلف هواپیما ، ماشین ، پرنده ، گربه ، آهو ، سگ ، قورباغه ، اسب ، کشتی و کامیون را نشان می دهد. از هر کلاس ۶۰۰۰ تصویر وجود دارد.

از آنجا که تصاویر موجود در CIFAR-10 با وضوح کم (۳۲×۳۲) است ، این مجموعه داده می تواند به محققان اجازه دهد تا به سرعت خیلی بالا الگوریتم های مختلف را امتحان کنند تا ببینند چه عواملی برای یادگیری ماشین مفید است. انواع مختلفی از شبکه های عصبی کانولوشن یا همان NN بهترین روش تشخیص تصاویر در CIFAR-10 هستند.

این دیتاست مانند دیتاست های معروف دیگر در خود keras موجود است و می توان با استفاده از دستور زیر آن را لود کرد:

```
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

فرمت داده های x به شکل ماتریس هایی ۳۲ در ۳۲ در ۳ هستند. فرمت های داده y نیز عدد در نظر گرفته شده آن کلاس است.

پس از لود کردن آن، می توانیم ۱۰ تصویر اولیه آن را با استفاده از کتابخانه matplotlib نمایش دهیم:

```
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
 'dog', 'frog', 'horse', 'ship', 'truck']

plt.figure(figsize=(10,10))
for i in range(10):
    plt.subplot(5, 5, i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(x_train[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[y_train[i][0]])
```

پس از نمایش تصویر باید داده ها را برای استفاده توسط شبکه آماده کنیم.

```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

x_train /= 255
x_test /= 255
```

```
y_train = to_categorical(y_train, len(class_names))
y_test = to_categorical(y_test, len(class_names))
```

ابتدا داده های x را به نوع float تبدیل می کنیم و سپس بر ۲۵۵ تقسیم می کنیم. زیرا حداکثر مقداری که هر پیکسل می تواند داشته باشد ۲۵۵ می باشد که با تقسیم آن بر ۲۵۵ محدوده آن به ۰ تا ۱ تبدیل می شود. در نهایت با استفاده از تابع to_categorical خروجی ها را از Class Vector به Binary Class Matrix تبدیل می کنیم. اینکار موجب بهبود عملکرد شبکه میشود و در همین حین مقادیر را از شکل یک ماتریس با یک درایه و محدوده ۰ تا ۹ به یک ماتریس با ۱۰ درایه و محدوده ۰ تا ۱ باز می گرداند.

برای تعریف مدل ما از مدل زیر استفاده کردیم:

```
model = Sequential([
    Flatten(input_shape=(32,32,3)),
    Dense(1024, activation='relu'),
    Dense(512, activation='relu'),
    Dense(512, activation='relu'),
    Dense(10, activation='softmax')
])
```

لایه اول، لایه Flatten است که برای تبدیل ماتریس ۳۲ در ۳۲ در ۳ به یک ماتریس تک بعدی است.

لایه دوم یک لایه Dense با تابع فعال سازی ReLU و ۱۰۲۴ نورون است.

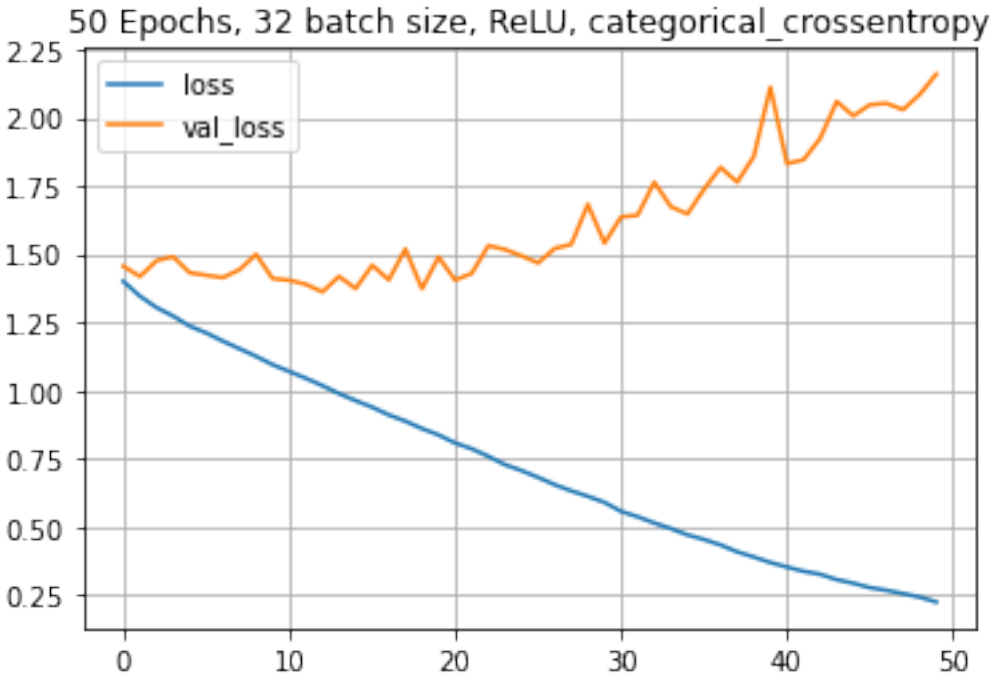
لایه سوم یک لایه Dense با تابع فعال سازی ReLU و ۵۱۲ نورون است.

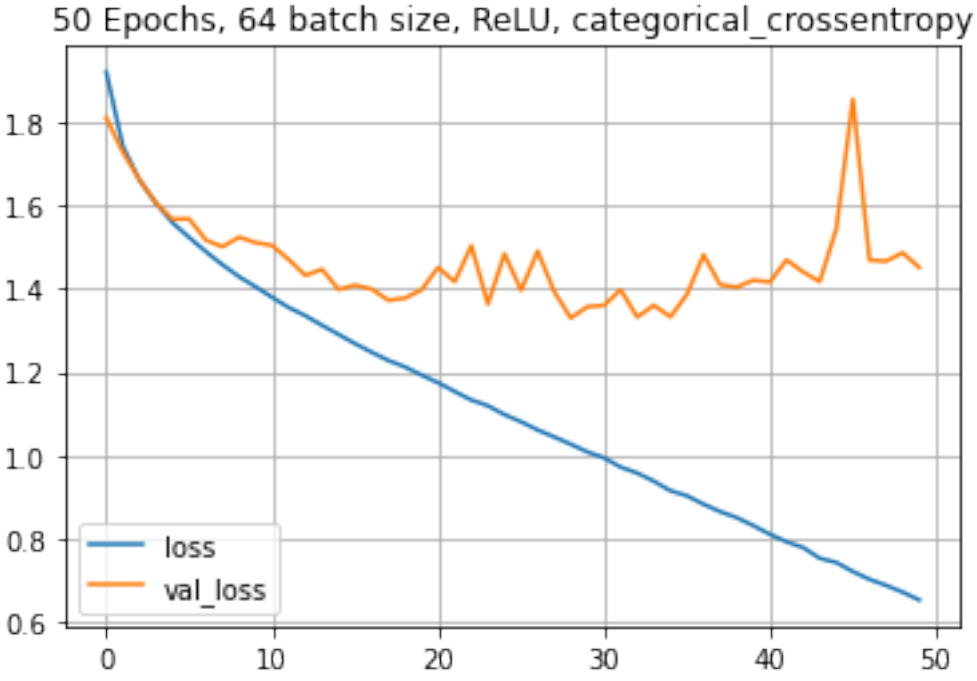
لایه چهارم یک لایه Dense با تابع فعال سازی ReLU و ۵۱۲ نورون است.

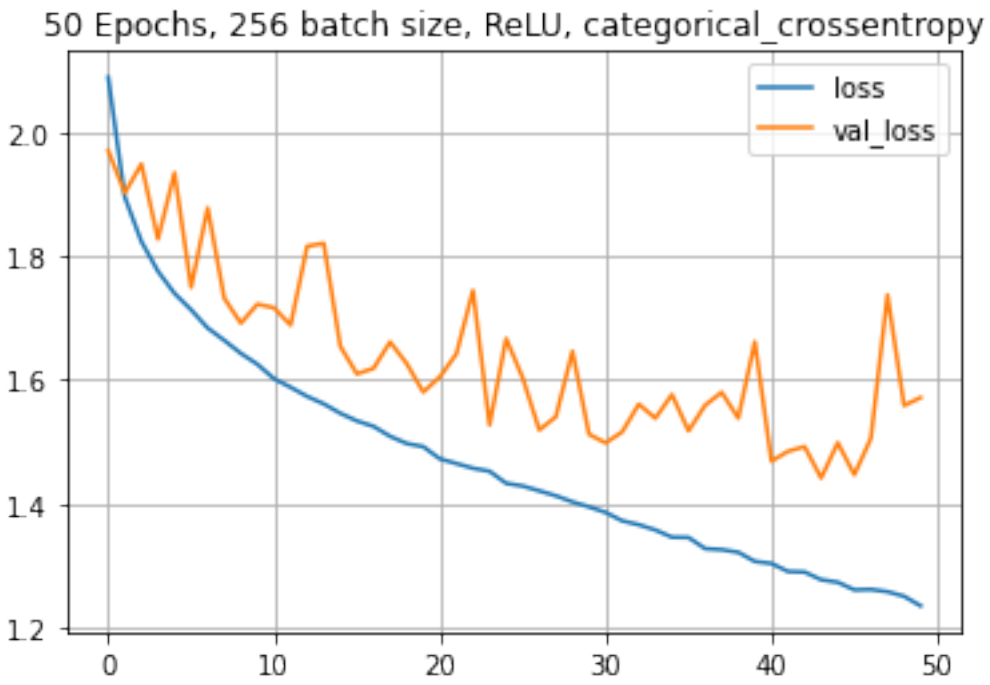
لایه پنجم و آخر نیز یک لایه Dense با تابع فعال سازی Softmax و ۱۰ نورون است.

علت استفاده از تابع Softmax در آخر برای بهبود عملکرد شبکه است. زیرا نوع مساله طبقه بندی است.

بخش الف



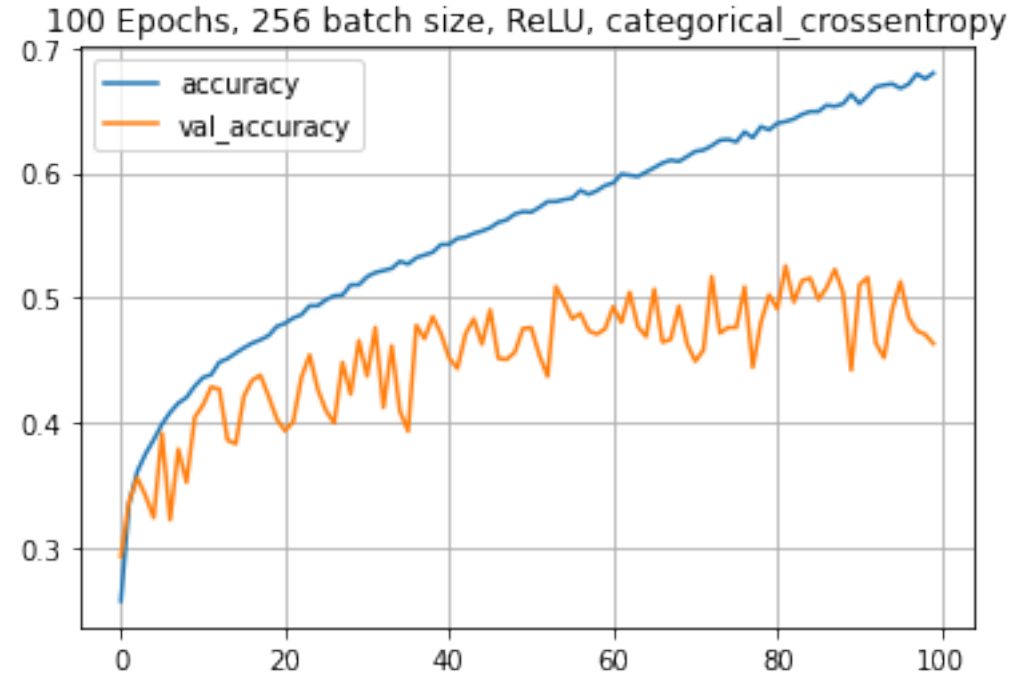


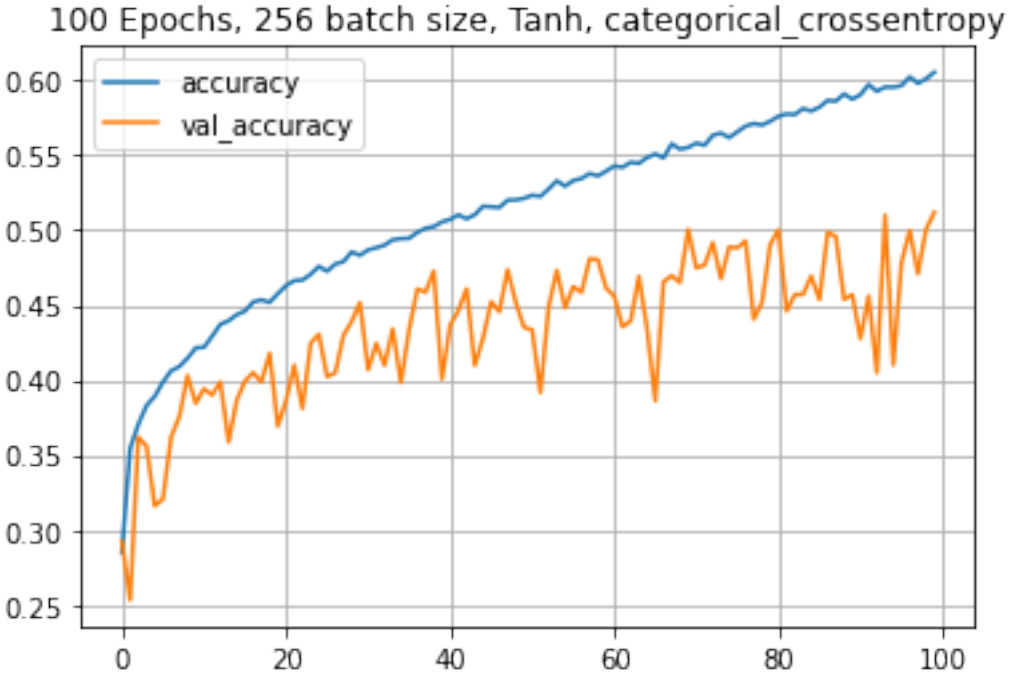


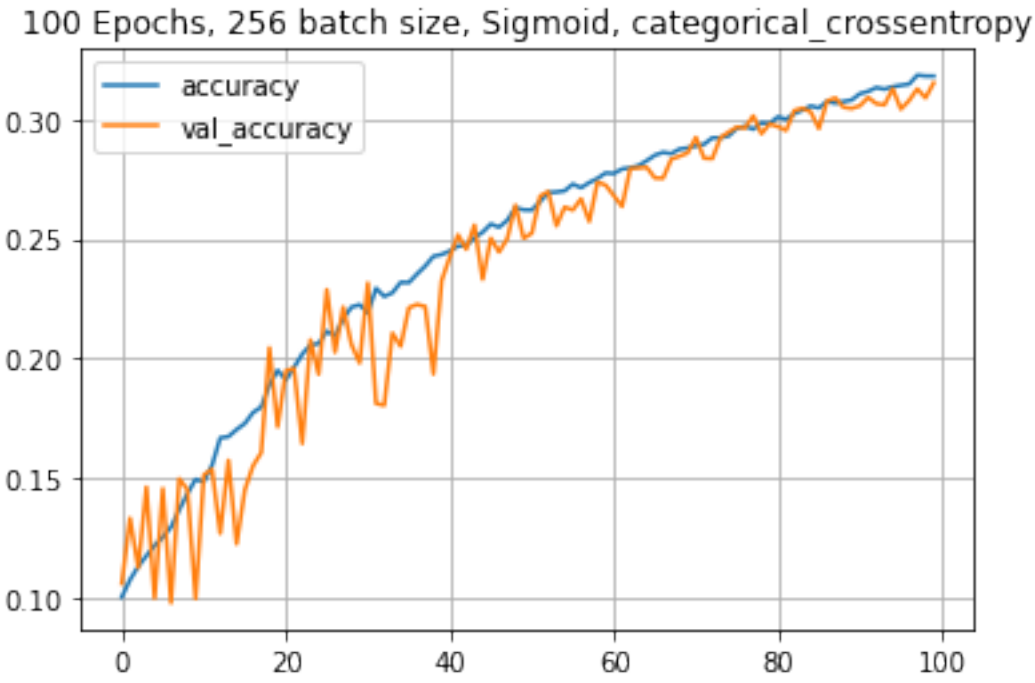
همانطور که مشاهده میشود، با افزایش تعداد batch ها، یادگیری شبکه در داده های validation بهبود پیدا می کند، از overfit جلوگیری می شود و همچنین با افزایش تعداد batch ها سرعت یادگیری و اجرای مدل نیز بیشتر می شود.

در همه آزمایش های این بخش از بهینه ساز SGD و تعداد Epoch برابر با ۵۰ و تابع Loss از نوع categorical_crossentropy استفاده شده است.

بخش ب


100 Epochs, 256 batch size, ReLU, categorical_crossentropy
accuracy
val_accuracy
0.7
0.6
0.5
0.4
0.3
0
20
40
60
80
100


100 Epochs, 256 batch size, Tanh, categorical_crossentropy
accuracy
val_accuracy
0.60
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0
20
40
60
80
100



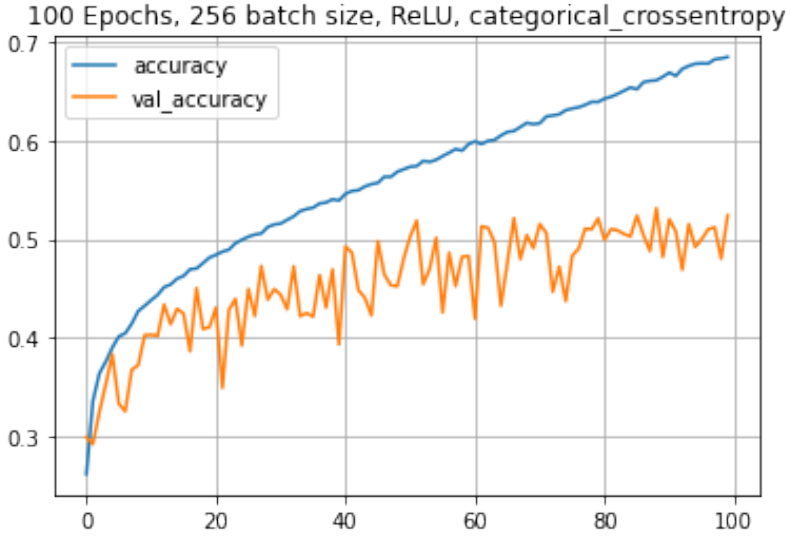
همانطور که مشاهده میشود، در یادگیری با استفاده از تابع ReLU سرعت یادگیری در داده های آموزش بالاست اما در یادگیری داده های validation پس از گذشت ۵۰ Epoch بهبود چشمگیری رخ نمی دهد و overfit رخ میدهد.

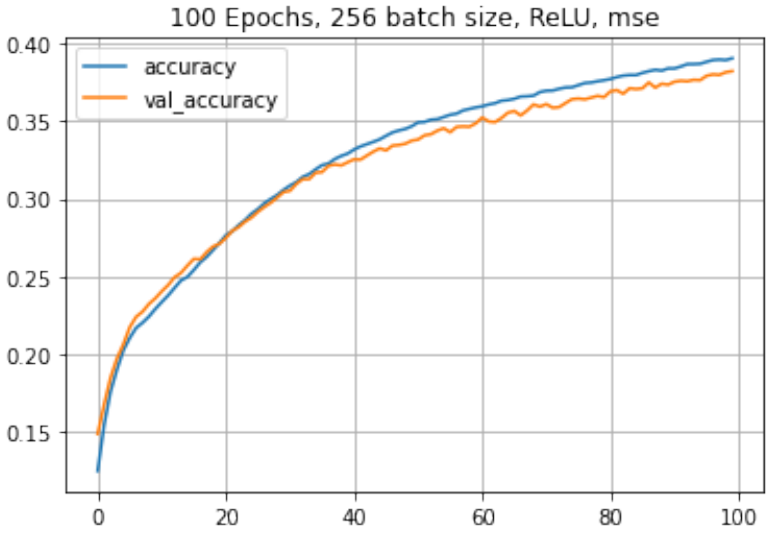
در یادگیری با تابع Tanh یادگیری نسبت به ReLU کندتر است و میزان نوسانات در یادگیری داده validation بالاتر است.

در یادگیری با تابع Sigmoid سرعت یادگیری بسیار کمتر از حالت های قبلی است اما میزان دقت شبکه در داده یادگیری و validation تقریبا یکسان است و overfit رخ نمی دهد.

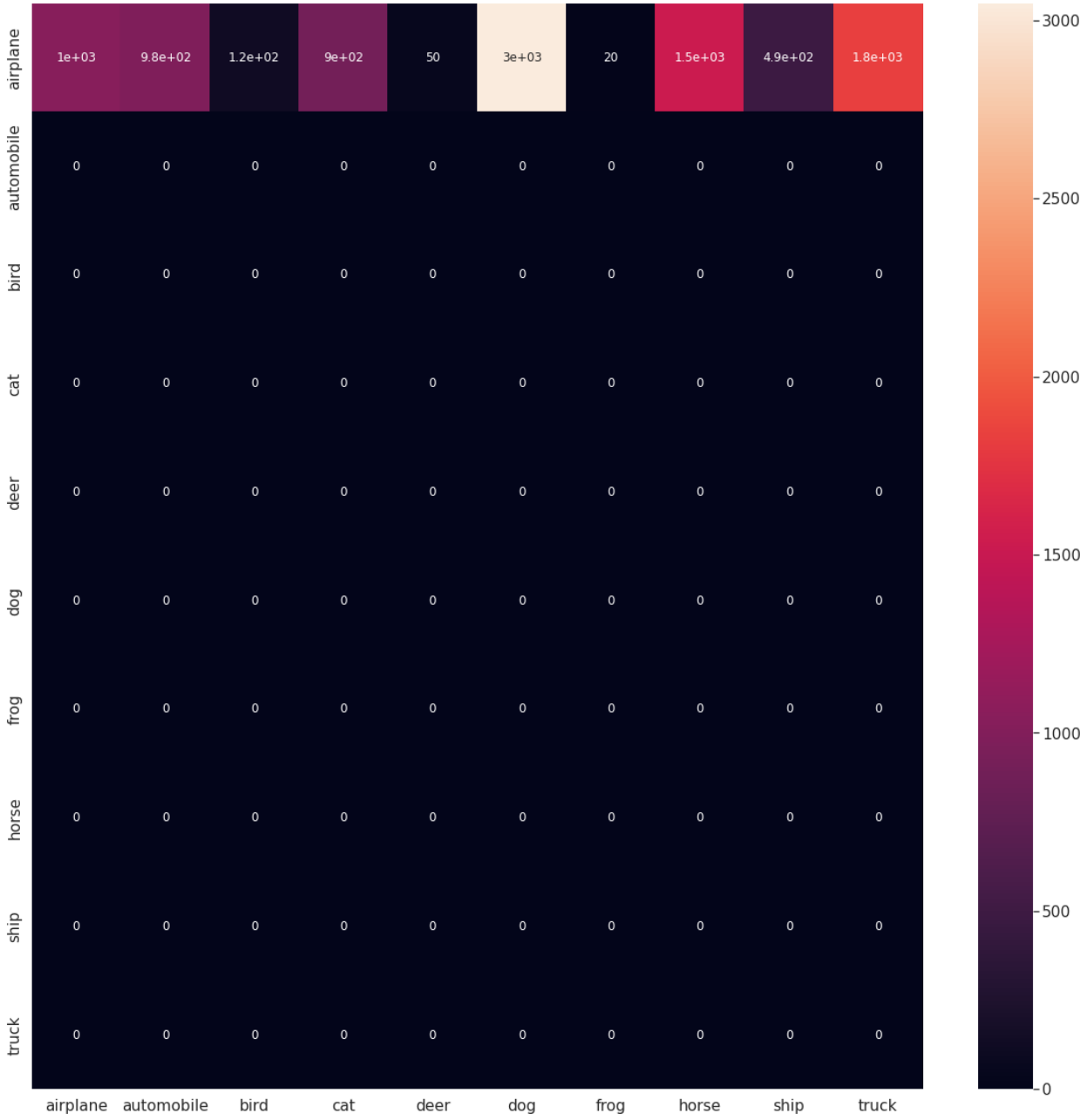
در همه این آزمایش ها از بهینه ساز SGD و Epoch برابر ۱۰۰ و batch size برابر ۲۵۶ استفاده شده است.

بخش ج





همانطور که مشاهده میشود شبکه در یادگیری با استفاده از تابع categorical_crossentrop بهتر عمل می کند که نتیجه ای قابل انتظار است، زیرا این تابع loss برای مساله های طبقه بندی بهتر می کند. ماتریس Confusion بهترین مدل از تمام این آزمایش ها به شکل زیر است:


airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck
1e+03 9.8e+02 1.2e+02 9e+02 50 3e+03 20 1.5e+03 4.9e+02 1.8e+03
3000
2500
2000
1500
1000
500
0

بخش امتیازی، استفاده از CNN

شبکه عصبی کانولوشن نوع خاصی از شبکه عصبی با چندین لایه است که داده‌هایی را که آرایش شبکه‌ای دارند، پردازش کرده و سپس ویژگی‌های مهم آن‌ها را استخراج می‌کند. یک مزیت بزرگ استفاده ازCNN ها این است که نیازی به انجام پیش‌پردازش زیادی روی تصاویر نیست.

در بیشتر الگوریتم‌هایی که پردازش تصویر را انجام می‌دهند، فیلترها معمولاً توسط یک مهندس بر اساس روش‌های اکتشافی ایجاد می‌شوند.CNN ها می‌توانند مهم‌ترین ویژگی فیلترها را بیاموزند و چون به پارامترهای زیادی احتیاج نیست، صرفه‌جویی زیادی در وقت و عملیات آزمون و خطا صورت می‌گیرد.

تا زمانی که با تصاویر با ابعاد بالا که هزاران پیکسل دارند کار نکنید، این صرفه‌جویی چندان به چشم نمی‌آید. هدف اصلی الگوریتم CNN این است که با حفظ ویژگی‌هایی که برای فهم آنچه داده‌ها نشان می‌دهند مهم هستند، داده‌ها را به فرم‌هایی که پردازش آن‌ها آسان‌تر است، درآورد. آن‌ها همچنین گزینه خوبی برای کار با مجموعه داده‌های عظیم هستند.

یک تفاوت بزرگ بین CNN و شبکه عصبی معمولی این است کهCNN ها برای مدیریت ریاضیات پشت‌صحنه، از کانولوشن استفاده می‌کنند. حداقل در یک لایه ازCNN ، به‌جای ضرب ماتریس از کانولوشن استفاده می‌شود. کانولوشن‌ها تا دو تابع را می‌گیرند و یک تابع را برمی‌گردانند.

CNNها با اعمال فیلتر روی داده‌های ورودی شما کار می‌کنند. چیزی که آن‌ها را بسیار خاص می‌کند، این است کهCNN ها می‌توانند فیلترها را هم‌زمان با فرایند آموزش، تنظیم کنند. به این ترتیب، حتی وقتی مجموعه داده‌های عظیمی مانند تصاویر داشته باشید، نتایج به‌خوبی و درلحظه دقیق‌تر می‌شوند.

از آنجا که می‌توان فیلترها را برای آموزش بهتر CNN تازه‌سازی کرد، نیاز به فیلترهای دستی از بین می‌رود و این انعطاف‌پذیری بیشتری در تعداد و ارتباط فیلترهایی که بر روی مجموعه داده‌ها اعمال می‌شوند، به ما می‌دهد. با استفاده از این الگوریتم، می‌توانیم روی مسائل پیچیده‌تری مانند تشخیص چهره کار کنیم.

شبکه طراحی شده به شرح زیر است:

```
model = Sequential()

model.add(Conv2D(8, kernel_size=(3, 3), input_shape=(32, 32, 3)
, activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(16, kernel_size=(3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(32, kernel_size=(3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())

model.add(Dense(128, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
```

این شبکه را می توان به دو بخش CNN و NN تقسیم کرد.

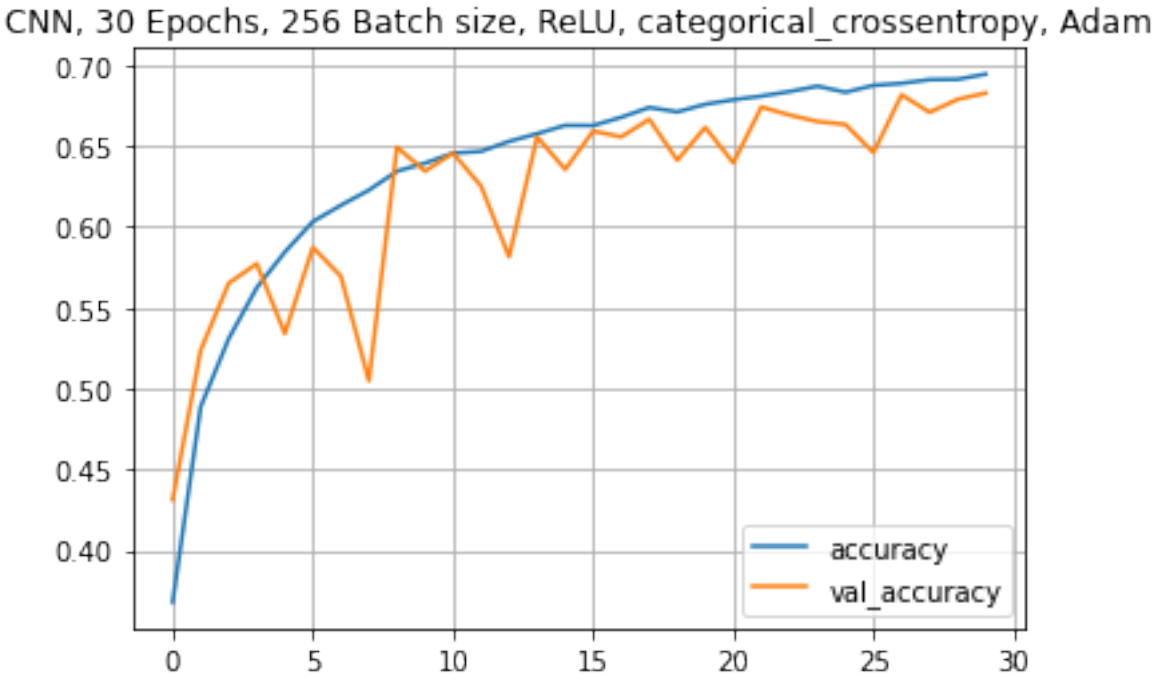
در بخش CNN ، ابتدا تصویر از یک کانولوشن با تعداد فیلتر ۱۶ عبور می کند. سپس با استفاده از لایه BatchNorm خروجی ها اسکیل میشوند (اینکار موجب یادگیری بهتر و سریع تر شبکه ها میشوند و از مشکلاتی از جمله Dying ReLU جلوگیری می کنند) و سپس از لایه MaxPooling برای کوچکتر شدن داده عبور می کنند.

در بخش های بعدی همینکار تکرار میشود با این تفاوت که تعداد فیلتر ها بیشتر میشوند و این موجب استخراج ویژگی های متمایزکننده هر تصویر میشود.

سپس این ویژگی ها از بخش CNN به بخش NN منتقل می شوند. در این بخش تعداد پارامتر های یادگیری از بخش بدون استفاده از CNN بسیار کمتر است و شبکه سریع تر اجرا میشوند. برای جلوگیری از Overfit شدن نیز از لایه Dropout نیز استفاده شده است (از این لایه می توان در بخش CNN نیز به همراه BatchNorm استفاده کرد).

در آخر یک لایه با ۱۰ نورون و تابع فعال ساز softmax وجود دارد.

برای بهینه ساز ما از بهینه ساز Adam استفاده کردیم زیرا علاوه بر امکانات SGD دارای ویژگی های بیشتری است که سرعت یادگیری را افزایش می دهند.



همانطور که مشاهده میشود این شبکه در با ۳۰ مرحله یادگیری می تواند هم در داده یادگیری و هم داده Validation دقت ۷۰ درصد داشته باشد و همچنین با مشکل overfit مواجه نشود. و همچنین یادگیری این شکبه تنها ۱۸۰ ثانیه طول می کشد درحالی که شبکه اولیه بیش از ۵ دقیقه طول کشید.

Confusion matrix این شبکه به شکل زیر است: