به نام خدا

پروژه عملی دوم هوش مصنوعی

استاد: دکتر آرش عبدی هجراندوست

پیاده‌سازی شبکه عصبی پرسپترون چندلایه

محمدصادق مجیدی یزدی ۹۸۱۰۶۰۰۴

فهرست عناوین

نکات کلی و ملزومات اجرای پروژه

در پیاده‌سازی پیش رو از کتابخانه‌های numpy و pandas برای کار با مجموعه‌داده‌ها و انجام محاسبات و عملیات‌های لازم استفاده شده است. همچنین از کتابخانه‌های sklearn و tensorflow و keras برای ساخت مدل‌ها، یادگیری آن‌ها، بهینه‌سازی‌ها، محاسبه loss و گزارش و تحلیل نتایج روی داده‌های آزمایشی استفاده شده است. از کتابخانه matplotlib برای نمایش گرافیکی نتایج مدل‌ها استفاده کرده‌ایم. برای نصب این کتابخانه‌ها کافیست پس از ایجاد یک python virtual event و اتصال به آن محیط، دستور زیر را در خط فرمان وارد کنید تا عملیات نصب به‌طور خودکار انجام شود.

pip install -r requirements.txt [python3 pip install -r requirements.txt]

البته استفاده از یک محیط توسعه پایتون مانند pycharm باعث تسهیل در اجرای این مراحل خواهد شد. در صورت وجود مشکل اجرای دستا این دستورات می‌تواند کمک کننده باشد.

pip install numpy

pip install pandas

pip install scipy

pip install scikit-learn

pip install matplotlib

pip install tensorflow

pip install keras

پس از آماده شدن محیط اجرای برنامه در صورت استفاده از ترمینال برای اجرا کافیست دستور زیر را اجرا کنید.

python3 filename.py

با اجرای برنامه، خروجی برنامه به صورت گرافیکی در قالب فایل "name.png" تولید و به شما نمایش داده می‌شود و همچنین خروجی متنی گزارشات یادگیری به فرمتی خوانا در کنسول (یا ترمینال) چاپ می‌شود.

هر بخش از پروژه حاوی یک فایل اصلی کد پایتون به فرمت .py است. فایل function_pt4.csv مربوط به تابع دلخواه دست‌نویس در قسمت چهارم است. همچنین این امکان در نظر گرفته شده است که در صورت بروز مشکل هنگام دانلود دیتاست mnist از api های آماده keras، از فایل محلی عملیات خواندن دیتاست انجام شود.

بخش اول: تابع را بیابیم

این مسئله در واقع معادل همان Regression است. ابتدا به توضیح کدهای نوشته‌شده می‌پردازیم و سپس به سراغ عملکرد مدل و آزمایش آن می‌رویم.

کلاس DatasetCreator:

```
class DatasetCreator:

    def __init__(self, function, train_n_samples, train_min, train_max, test_n_samples, test_min, test_max):
        self.function = function
        self.train_n_samples = train_n_samples
        self.train_min = train_min
        self.train_max = train_max
        self.test_n_samples = test_n_samples
        self.test_min = test_min
        self.test_max = test_max
```

این کلاس در واقع مسئولیت تامین داده‌ها برای مدل را دارد. در کنستراکتور این کلاس موارد زیر را ورودی می گیریم:

پارامتر function: در واقع همان تابعی است که قصد داریم آن را به وسیله شبکه بیابیم. با استفاده از آن می‌خواهیم تعدادی نقطه ورودی و خروجی تولید کنیم تا به عنوان دادگان آموزشی و آزمایشی استفاده شوند.

پارامتر train_n_sample: تعداد داده‌هایی (بخوانید نقطه‌ها) که باید برای دادگان آموزشی تولید کنیم.

پارامتر train_min: کمترین مقدار مجاز برای فیچر ورودی در دادگان آموزشی تولیدشده.

پارامتر tarin_max: بیشترین مقدار مجاز برای فیچرهای ورودی در دادگان آموزشی تولیدشده.

پارامتر test_n_sample: تعداد داده‌هایی که باید برای دادگان آزمایشی تولید کنیم.

پارامتر test_min: کمترین مقدار مجاز برای فیچر ورودی در دادگان آزمایشی تولیدشده.

پارامتر test_max: بیشترین مقدار مجاز برای فیچرهای ورودی در دادگان آزمایشی تولیدشده.

در ادامه دو تابع این کلاس برای تولید دادگان آزمایشی و آموزشی را می‌بینیم:

```python
def get_train_data(self):
    X = np.linspace(self.train_min, self.train_max, self.train_n_samples)
    labels = self.function(X)
    p = np.random.permutation(self.train_n_samples)
    return X[p], labels[p]

def get_test_data(self):
    X = np.linspace(self.test_min, self.test_max, self.test_n_samples)
    labels = self.function(X)
    return X, labels
```

در هر دو با استفاده از تابع linspace از کتابخانه نامپای تعداد مشخصی نقطه با فواصل برابر در بازه min و max ای که قبلا در ورودی گرفتیم تولید می‌کند. سپس این نقاط تولیدشده به عنوان ورودی به تابع function خودمان داده می‌شود و برچسب‌های خروجی تولید می‌شوند. همچنین برای پیشگیری از هر نوع تاثیر احتمالی ترتیب مشخصی که بین داده‌ها در دادگان آموزشی وجود دارد یک جایگشت دلخواه از آن را به عنوان مجموعه داده آموزشی مورد نظرمان خروجی می‌دهیم.

```python
class FunctionApprox:
    def __init__(self, dataset_creator: DatasetCreator, batch_size: int, num_epochs: int, k_folds: int = 5):
        self.batch_size = batch_size
        self.num_epochs = num_epochs
        self.k_folds = k_folds
        self.dataset_creator = dataset_creator
        self.loss_function = mean_squared_error
        self.optimizer = keras.optimizers.Adam(learning_rate=0.001)
        self.X, self.y = self.dataset_creator.get_train_data()
        self.model = None
        self.best_loss = None
        self.avg_loss = None
```

پارامتر dataset_creator آبجکت تولیدکننده دادگان آموزشی و آزمایشی برای مدل ما است که قبل از ورودی دادن به مدل، تنظیمات آن را با توجه به توضیحات بالا انجام داده‌ایم.

پارامتر batch_size تعداد داده‌های آموزشی که در هر مرحله از یادگیری و بهینه‌سازی در اختیار مدل قرار داده می‌شود تا بر اساس آن‌ها در جهت کمینه کردن loss خود پیش برود را تعیین می‌کند. در صورتی که داده‌ها را بخش‌بخش نکنیم و تماما در اختیار مدل خود قرار دهیم هر مرحله از یادگیری

و بهینه کردن loss مدت زمان و محاسبات زیادی صرف می‌شود و در مجموع بر روی کارایی مدل تاثیر منفی می گذارد. در صورتی هم که داده‌ها را به صورت تک به تک stochastic gradient descent در اختیار مدل قرار دهیم با این که هر مرحله از بهینه‌سازی به سرعت انجام می‌شود اما مقدار loss بسیار اندک تغییر می کند و به سختی به سمت مقدار بهینه سراسری می‌رود و گاها حتی به آن نقطه نمی‌رسد. پس استفاده از یک batch با تعداد معقول داده بهترین گزینه ما است. تعداد آن در اکثر بخش‌های پروژه برابر با ۳۲ که در بین مقادیر تست‌شده غالبا بهترین عملکرد را چه از لحاظ زمانی و چه از لحاظ بهینگی loss داشت، قرار داده شده است. البته قطعا به تاثیر تغییر این پارامتر در ادامه خواهیم پرداخت.

پارامتر num_epochs تعیین می کنید که مدل باید چند بار به صورت کامل یادگرفته شود. هر مرحله از نتایج و پارامترهای به دست آمده از مرحله قبل استفاده کرده و مجددا روی همان دادگان آموزشی یاد گرفته می‌شود تا این بار از نقطه‌ای بهینه‌تر به سمت مدل مطلوب پیش برود. انتخاب آن به در نظر گرفتن موارد خارجی زیادی نیاز دارد. غالبا استفاده از ۶ epoch در این پروژه پاسخگوی نیازهای ما بود.

پارامتر k_folds تعیین چگونگی اعمال Cross Validation روی داده‌های آموزشی را انجام می‌دهد. به طور خلاصه داده‌های در اختیار مدل هر بار $\frac{k-1}{k}$ شان نمونه‌گیری شده و در یادگیری شرکت می کنند و این عمل k بار تکرار می‌شود. این مقدار را به صورت پیش‌فرض برابر ۵ که مقداری معقول و پراستفاده است قرار می‌دهیم. لازم به ذکر است برای بهبود یادگیری خودمان در تمام بخش‌ها از تکنیک Cross Validation بهره برده‌ایم.

برای محاسبه loss مدل از معیار mean squared error که بهترین انتخاب در اکثر مسائل Regression است استفاده کرده‌ایم و استفاده از دیگر معیارها مانند cross entropy یا log loss در این مسئله خیلی جای بحث ندارند.

برای تکنیک بهینه‌سازی از تکنیک adam استفاده کرده‌ایم. تکنیک انتخابی ما در چگونگی مقداردهی اولیه وزن‌های شبکه و یادگیری آن‌ها در فرایند back propagation تعیین کننده است.

تکنیک‌های اولیه ساده‌ای مانند gradient descent وجود دارند که بر مبنای حرکت در خلاف جهت گرادیان سعی می‌کنند به نقطه بهینه سراسری نزدیک شوند، در مواجهه با تعداد وزن‌ها و داده‌های آموزشی بالا مانند ساختار یک شبکه عصبی، بسیار کند عمل می‌کنند و معمولا این مشکل را دارند که در یک مینیمم محلی گیر افتاده و متوقف می‌شوند. از طرفی نسخه stochastic این الگوریتم که به جای استفاده از تمامی داده‌ها در محاسبه گرادیان، از تک‌داده‌های ورودی استفاده می‌کند با این که سرعت نسبتا بهتری دارد ما با این مشکل رو به رو است که مقدار loss متناوبا تغییر می‌کند و روند کاملا کاهشی ندارد و ممکن است در نهایت در نقطه‌ای حتی غیر از مینیمم محلی سر در بیاوریم. الگوریتم‌های بهتری نیز برای این منظور ارائه شده‌اند که Adam یکی از معروف‌ترین و پرکاربردترین آن‌ها به خصوص در بحث یادگیری شبکه‌های عصبی است. این الگوریتم عملیات آپدیت وزن‌ها را بر اساس میانگین و واریانس قبلی وزن‌ها و نرخ یادگیری eta (یا همان learning rate) و تعدادی ضریب ثابت دیگر که از حوصله این گزارش خارج است انجام می‌دهد. این تکنیک از لحاظ زمانی و دقت یادگیری وزن‌ها کارایی خوبی دارد و انتخاب ما در سراسر این پروژه است. به صورت پیش‌فرض مقدار eta را برابر با ۰.۰۰۰۱ قرار می‌دهیم. این عدد و ۰.۰۰۱ و ۰.۰۰۰۰۱ سه مقدار پرتناوب برای نرخ یادگیری در مدل‌ها هستند.

در ادامه معماری شبکه عصبی استفاده‌شده را بررسی می‌کنیم:

```
def get_model(self):
    model = keras.Sequential()
    model.add(keras.layers.Dense(128, input_dim=1, activation='relu'))
    model.add(keras.layers.Dense(256, activation='relu'))
    model.add(keras.layers.Dense(512, activation='relu'))
    model.add(keras.layers.Dense(1))

    model.compile(loss=self.loss_function, optimizer=self.optimizer)
    return model
```

این تابع شبکه عصبی را ساخته و کامپایل می‌کند و خروجی می‌دهد.
شبکه عصبی ما شامل لایه اول با ۱۲۸ نورون با تابع فعال‌ساز ReLU که تک‌ورودی ما به تمام ۱۲۸ نورون آن وصل است، لایه دوم با ۲۵۶ نورون ReLU و لایه سوم با ۵۱۲ نورون ReLU و لایه خروجی تک‌نورونی که فاقد فعال‌ساز است یا به عبارت دیگر تابع فعال‌ساز آن linear است و ترکیب‌خطی ورودی‌ها را عینا به خروجی منتقل می‌کند. می‌دانیم در مسائل regression مناسب است در خروجی به تعداد ابعاد خروجی تابع مورد نظر نورون linear بگذاریم تا به راحتی بتوانند هر مقدار حقیقی را تقریب بزنند. وزن اولیه نورون‌های شبکه طبق پیش‌فرض خود کتابخانه با یک توزیع نرمال تعیین می‌شوند.
برای یادآوری خوب است ذکر کنیم که تابع فعال‌ساز ReLU در واقع به صورت $ReLU(x) = max(0, x)$ عمل کرده و یک تابع غیرخطی بسیار پرکاربرد و مناسب در شبکه‌های عصبی است. دیگر تابع غیرخطی که در مسائل شبکه عصبی و به خصوص به عنوان نورون خروجی لایه آخر در مسائل دسته‌بندی باینری به کار برده می‌شود Sigmoid یا همان تابع logistic است. این تابع ورودی x را با فرمول $\frac{1}{1+e^{-x}}$ به صورت پیوسته به بازه 0 تا 1 نگاشت می‌کند. همچنین فعال‌ساز softmax نیز که معمولا در نورون‌های لایه آخر شبکه‌های عصبی مربوط به دسته‌بندی چندکلاسه کاربرد دارن از دیگر توابع فعال‌ساز معروف هستند و به هر نورون مربط به یک کلاس مجزا یک احتمال به صورت $\frac{e^x}{\sum_i^k e^x}$ نسبت می‌دهد تا کلاس با بیشترین احتمال انتخاب شود. نورون بدون هیچ تابع فعال‌سازی هم اصطلاحا دارای فعال‌ساز linear است که هیچ تغییری در ورودی آکسون ایجاد نمی‌کند و همان را خروجی می‌دهد.
در ادامه نحوه یاد گرفتن مدل را مشاهده خواهیم کرد:

```python
def train(self):
    loss_history = []
    model_history = []
    k_fold = KFold(n_splits=self.k_folds, shuffle=True)

    for train, test in k_fold.split(self.X, self.y):
        model = self.get_model()
        model.fit(self.X[train], self.y[train], batch_size=self.batch_size, epochs=self.num_epochs, verbose=True)
        scores = model.evaluate(self.X[test], self.y[test], verbose=0)
        loss_history.append(scores)
        model_history.append(model)
```

همان‌طور که مشخص است از کلاس KFold از کتابخانه sklearn برای اعمال Cross Validation در یادگری استفاده کرده‌ایم که طبق پیش‌فرض Fold-5 است. در هر Fold مدل را دریافت و آن را روی داده‌های آموزشی این fold با تعداد epoch و batch_size که قبلا مشخص شد،آموزش می‌دهیم. گزینه verbose را هم فعال می کنیم تا جزییات یادگیری در کنسول با فرمتی مناسب به نمایش دربیایند. پس از انجام آموزش، مقدار loss نهایی و خود مدل را در یک تاریخچه نگه‌داری می کنیم.

```python
for i in range(0, len(loss_history)):
    print('******************************************************************************')
    print(f' fold {i + 1} - Loss: {loss_history[i]}')
print('******************************************************************************')
print('******************************************************************************')

best_model_index = np.argmin(loss_history)
best_model = model_history[best_model_index]
self.model = best_model
self.best_loss = loss_history[best_model_index]
self.avg_loss = np.mean(loss_history)
return best_model
```

سپس مقادیر loss تمام fold ها را نمایش داده و مدلی که در بین همه‌ی foldها دارای کمترین loss است را انتخاب و بهترین مدل و loss و میانگین loss را در جایی ذخیره می کنیم.

```python
def plot_result(self):
    fig = plt.figure()
    test_X, test_y = self.dataset_creator.get_test_data()
    pred_y = self.model.predict(test_X)
    plt.plot(test_X, pred_y, '-r')
    plt.plot(test_X, test_y, '--b', alpha=0.5)
    return fig
```

این تابع داده‌های آزمایشی را با استفاده از dataset_creator از قبل داده‌شده تولید می‌کند. این دادگان برای پیش‌بینی مقدار خروجی به مدل داده می‌شود. در نهایت با استفاده از ابزار کتابخانه matplotlib نمودار تابع را بر حسب مقادیر واقعی با رنگ آبی و بر حسب مقادیر پیش‌بینی‌شده توسط مدل با رنگ قرمز رسم می‌کنیم.

```
if __name__ == '__main__':
    ds_creator = DatasetCreator(test_func_2, train_n_samples=20000, train_min=1, train_max=20
                                , test_n_samples=200000, test_min=0.7, test_max=30)
    f_predictor = FunctionApprox(ds_creator, batch_size=32, num_epochs=6)
    f_predictor.train()
    fig = f_predictor.plot_result()
    plt.show()
    plt.savefig('part_1.png')
```

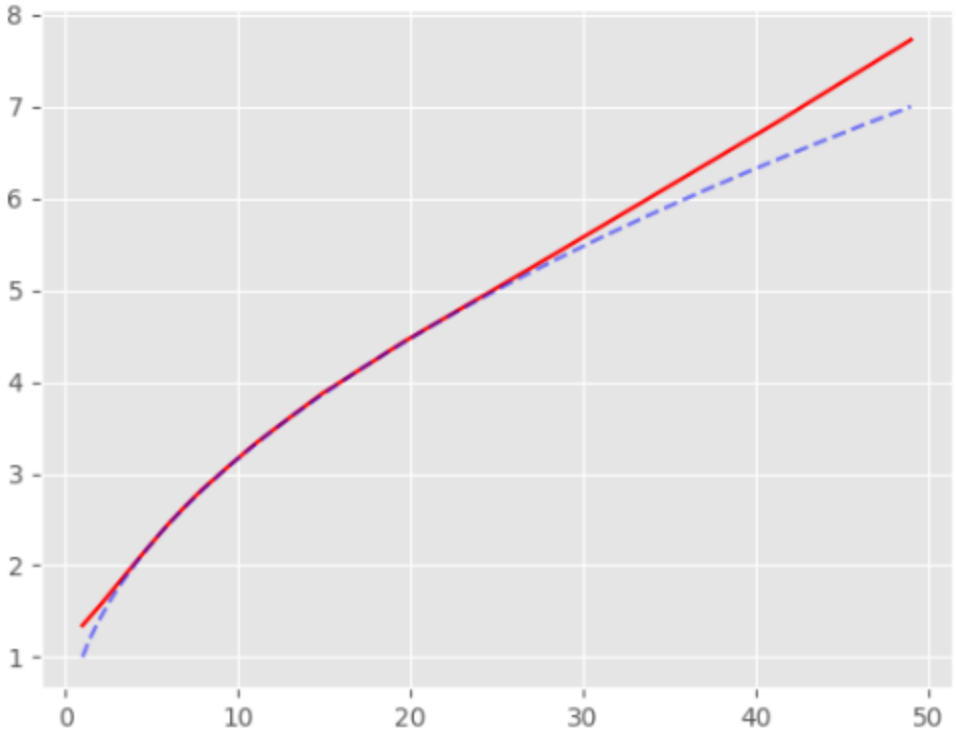
در اینجا نحوه استفاده و اتصال عناصری که تا اینجا معرفی کرده‌ایم برای ساخت و آموزش کامل یک مدل شبکه عصبی نشان داده شده است. اکثر پارامترهای قابل تغییر از این قسمت کد قابل مقداردهی هستند.

در ادامه به اجرا و بررسی مدل روی چند نمونه و تحلیل نتایج می‌پردازیم.

فرض کنید تابع مورد نظر اولیه ما تابع بسیار ساده جذر است. $f(x) = \sqrt{x}$

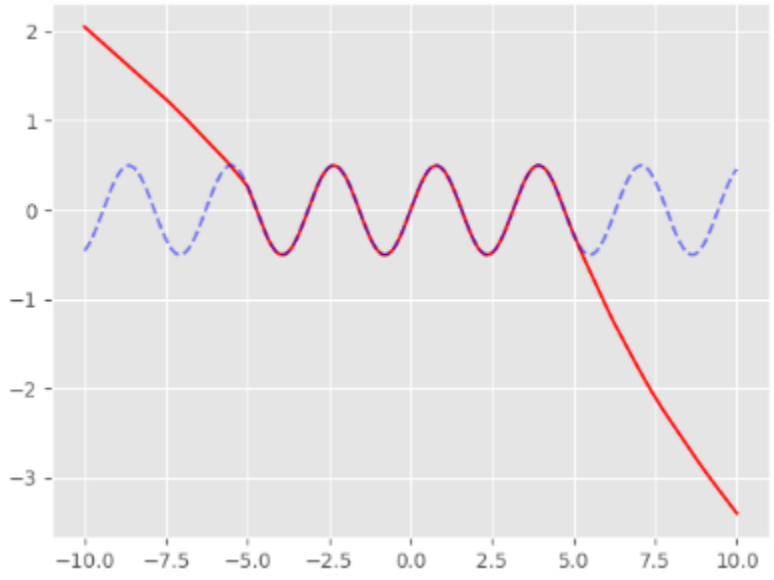
دادگان آموزشی در بازه ۴ تا ۲۵ و دادگان آزمایشی در بازه ۱ تا ۴۹ به مدل داده می‌شود. به منظور این که کارآمدی مدل در هر دو بخش برون‌یابی و درون‌یابی بررسی شود سعی شده در اکثر آزمایش‌ها بازه آموزش زیرمجموعه‌ای کوچک‌تر از بازه آزمایش باشد. تعداد ۲۴هزار نقطه برای آموزش و ۲۰۰۰۰۰ نقطه برای آزمایش تولید می‌شوند.

نتیجه به صورت زیر می‌شود:



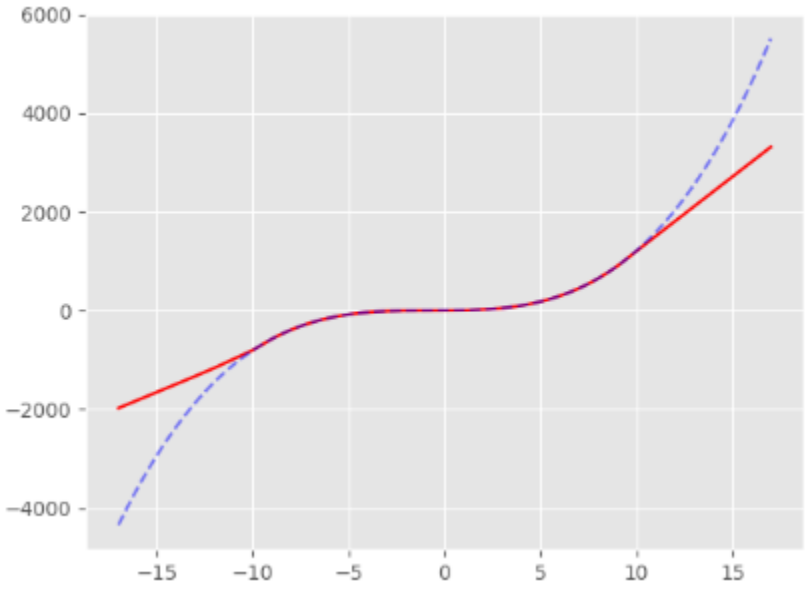
و مقدار میانگین loss در بین ۵ بار تکرار آموزش حدودا 0.0001 است.

تابع آزمایش‌شده دوم به صورت $f(x) = sin(x) * cos(x)$ است و داده‌های آموزشی در بازه ۵- و ۵ و داده‌های آزمایشی در بازه ۱۰- تا ۱۰ با همان تعداد قبلی تولید شده‌اند.



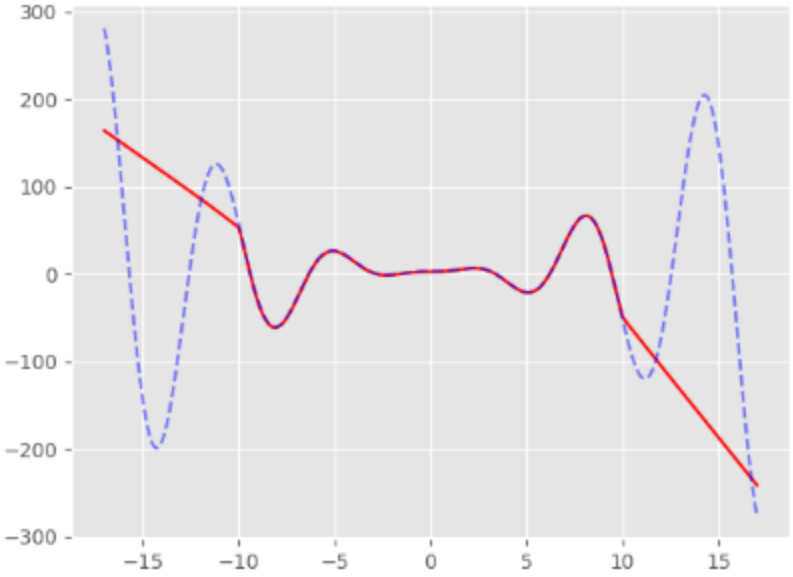
مقدار میانگین loss در بین foldها برابر ۰.۰۰۰۰۰۴ است.

تابع بعدی یک چندجمله‌ای به صورت $x^3 + 2x^2 + x + 5$ است. مقادیر ورودی در بازه ۱۰- تا ۱۰ برای داده‌های آموزشی و ۱۷- تا ۱۷ برای داده‌های آزمایشی با همان تعداد در نظر گرفته شده.



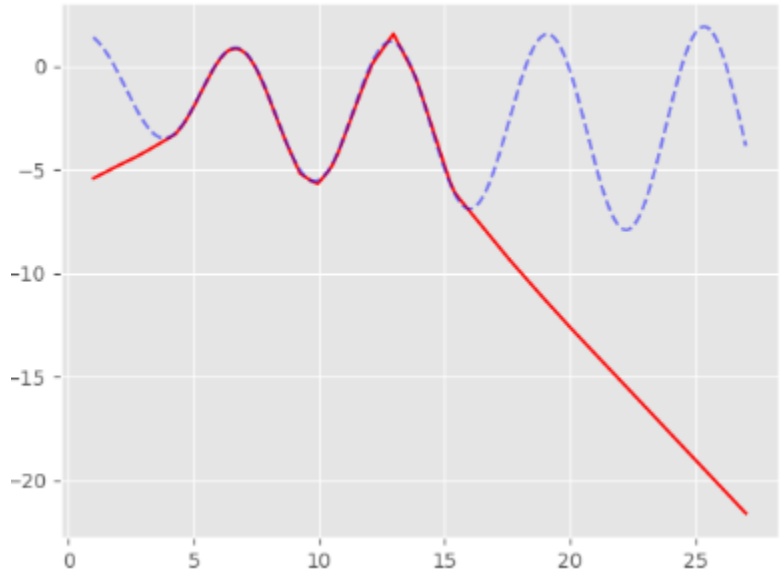
میانگین loss برابر ۲/۳۸۴ است.

به عنوان نمونه چهارم تابع ترکیبی $x^2 sin(x) + 3$ را تست می‌کنیم. وضعیت داده‌های آموزشی و آزمایشی را مانند همان مثال قبل در نظر می‌گیریم.



و مقدار loss میانگین آموزش‌ها برابر با ۱/۴۸۱ است.

به عنوان نمونه آخر یک تابع ترکیبی پیچیده مانند $f(x) = sin(x) - ln(x) + \sqrt{x}cos(x)$ را در اختیار شبکه قرار می‌دهیم. دادگان آموزشی در بازه ۴ تا ۱۶ و مقادیر آزمایشی از بازه ۱ تا ۲۷ برگزیده شده‌اند.



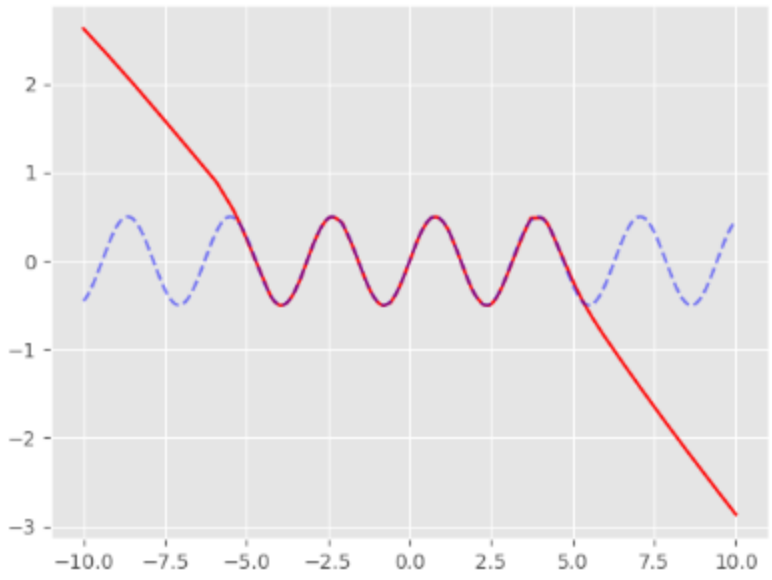
مقدار میانگین loss نیز برابر با ۰۰۱۳۸ است.

تمامی اجراها از تکنیک Cross Validation استفاده کردند. اجراها با توجه به حجم داده‌های ورودی که ۲۴ هزار نقطه ورودی بود در حدود ۴۰ ثانیه وقت می‌برد. نکته‌ای که به طور مشترک در تمامی این اجراها به چشم می‌خورد این است که عملکرد شبکه عصبی روی داده‌های آزمایشی تا جایی که در بازه آموزش خوش قرار داشته‌اند بسیار عالی و با دقت است ولی در خارج از این بازه فقط تا فاصله کمی تقریب خوبی از تابع است و در کل نمی‌تواند تخمینی مطمئن از تابع را در خارج از بازه آموزش در اختیار ما قرار دهد. این موضوع نشان می‌دهد که مدل شبکه عصبی در مسائل رگرسیون و در این مورد به خصوص که تقریب زدن تابع است، درون‌یابی بسیار عالی و قابل قبولی در حضور داده کافی و در زمان کم ارائه می‌دهد ولی در زمینه برون‌یابی نمی‌تواند کمک شایانی به ما کند. البته از پیش‌زمینه‌ای که در درس محاسبات عددی داشتم هیچ روش کلی برای برون یابی نداریم و تابع خارج از محدوده دید ما می‌تواند هر رفتار دلخواهی داشته باشد . همه‌ی روش‌های درون‌یابی در درس محاسبات عددی نیز فقط تا فاصله خیلی کوچکی می‌توانستند برونیابی را با تقریب خوبی انجام دهند. بنابراین قدرت شبکه عصبی در زمینه تقریب تابع در این بخش تا حد خوبی نمایش داده شد.

حال سعی می‌کنیم کمی با مقادیر و پارامترهایی که استفاده شده است بازی کنیم تا تاثیر تغییرات آن‌ها بر فرایند یادگیری و نتایج را مشاهده کنیم.

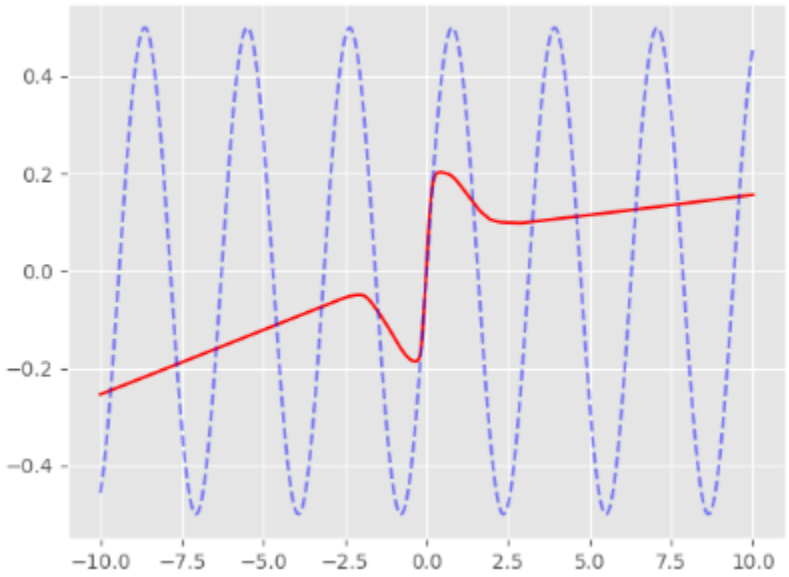
تابع دوم در مثال‌های بالا را در نظر می‌گیریم و تغییرات را روی مدل آن اعمال می‌کنیم. ابتدا روی تعداد داده‌های ورودی آموزشی بررسی انجام می‌دهیم. واضح است با افزایش مقدار داده‌ها نسبت به وضعیت فعلی، دقت و کمینگی loss بهبود خیلی کمی پیدا می‌کنند اما مدت زمان آموزش افزایش چشمگیری می‌یابد. حال تاثیر کم کردن آن را چک می‌کنیم.

با ۵۰۰۰ ورودی آموزشی:



مقدار loss به ۰.۰۰۰۰۷ رسیده است.

با ۲۵۶ داده ورودی:



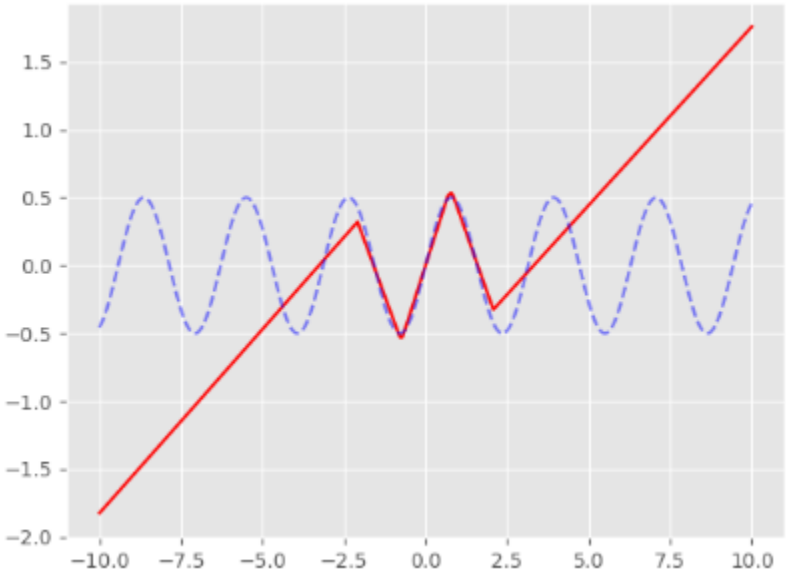
مقدار loss به ۰.۱۰۷ می‌رسد و زیاد شده است. همچنین ضعف خیلی واضحی در تابع آموزش‌داده‌شده مشاهده می‌شود زیرا تعداد داده‌های آموزشی حتی از مجموع نورون‌های شبکه کمتر

است. در کل کاهش ورودی منجر به کاهش دقت یادگیری خواهد شد و چون شبکه مدل خیلی قدرتمندی است بعضا منجر به بیش‌برازش روی آن مجموعه کوچک داده آموزش می‌شود.

روی میزان پیچیدگی در مثال‌های بالا مانور زیادی دادیم و دیدیم که قدرت شبکه عصبی تمامی آن‌ها را با دقت خیلی خوبی از ساده تا سخت جوابگو بود.

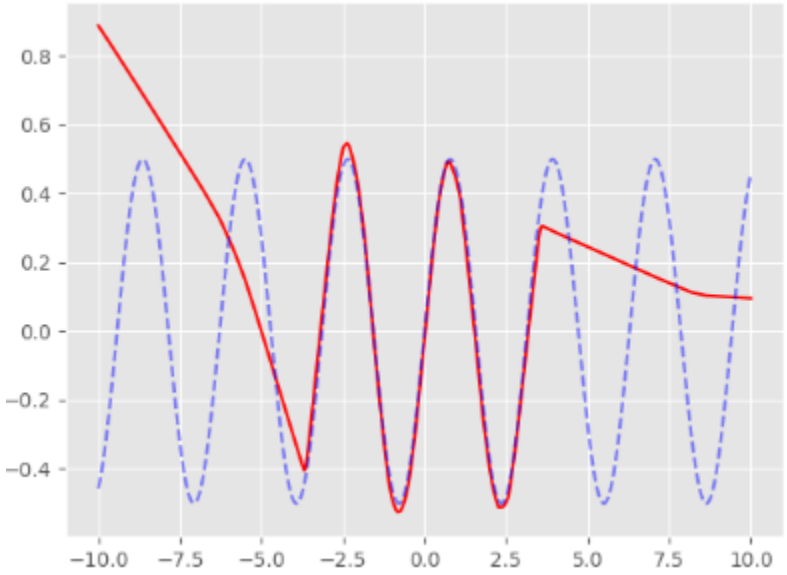
با تمرکز روی ساختار شبکه عصبی، می‌دانیم چه عرض شبکه و چه عمق آن را افزایش دهیم قدرت مدل بیشتر می‌شود و توابع خیلی سخت‌تری را می‌تواند یاد بگیرد. با توجه به این که همین شبکه فعلی هم تمام توابع ساده و پیچیده فعلی ما را به خوبی یاد گرفت افزایش حجم شبکه کار بیهوده‌ای خواهد بود و صرفا یادگیری آن طولانی‌تر خواهد شد. حال برای مثال دو لایه میانی را از شبکه فعلی حذف می‌کنیم و یک تک لایه ۱۲۸ نورونی ReLU اول را نگه می‌داریم و با همان شرایط اولیه یادگیری را انجام می‌دهیم.

نتیجه به صورت زیر می‌شود:



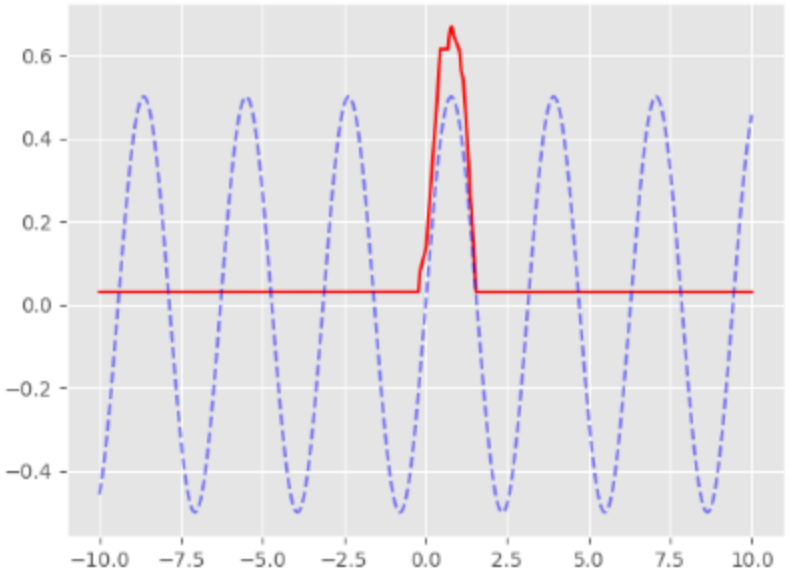
مشاهده می‌کنیم تقریب کلی از شکل تابع زده شده است ولی بسیار تیز و انعطاف‌ناپذیر است و دقت کمی دارد. بنابراین کاهش تعداد نورون‌ها و مخصوصا کاهش لایه‌ها به کاهش قدرت مدل می‌انجامد و نتیجه حاصل مطلوب نخواهد شد. پس شبکه هرچه عمیق‌تر و عریض‌تر بهتر. البته کاهش ابعاد شبکه به افزایش سرعت یادگیری می‌انجامد.

درباره تعداد چرخه‌های یادگیری شبکه در توضیحات num_epochs توضیحاتی داده شد. هرچه مقدار آن را زیاد کنیم مدت زمان یادگیری افزایش می‌یباد ولی دقت تا یک حدی افزایش می‌یابد و مدل تقریبا به بهترین حالت بهینه خود با این مجموعه داده آموزشی می‌رسد. کاهش آن ولی منجر به کاهش دقت خواهد شد. در زیر یادگیری همان تابع با epoch ۱ و 2folds نشان داده شده است که از دقت آن کاسته شده است.

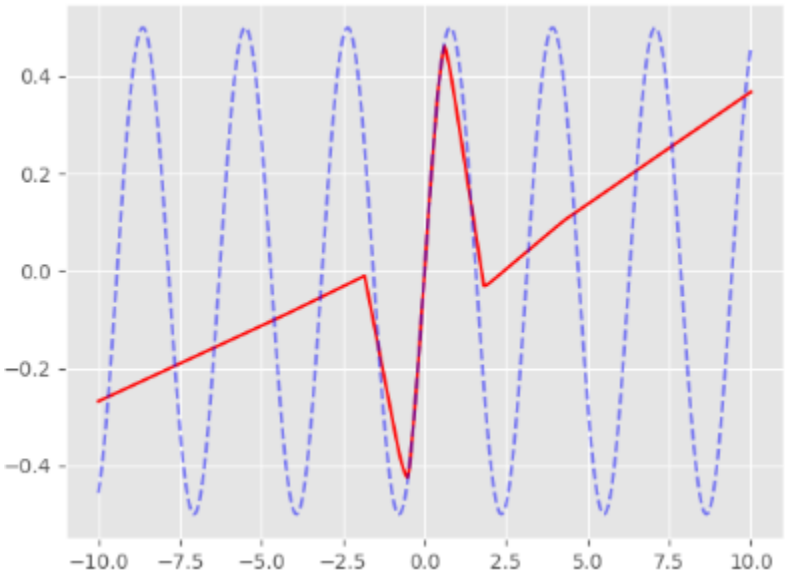


تاثیر learning rate در اپتیمایزر را بر یادگیری مدل بررسی می کنیم. هر چه این مقدار را نسبت به حالت قبلی کاهش دادیم مدت زمان یادگیری کمی افزایش یافت و از جایی به بعد نتیجه بدتری تولید کرد زیرا در واقع آن قدر همگرایی کند می‌شود که مدل تقریبا تغییر زیادی نمی کند وقتی که آموزش به پایان رسیده است. از آن طرف اگر آن را افزایش دهیم از یک جا به بعد مقدار loss شروع به تناوب و بعد از یک نقطه دیگر کاملا واگرا می‌شود.

نتیجه مدل ما با eta = 0.1:



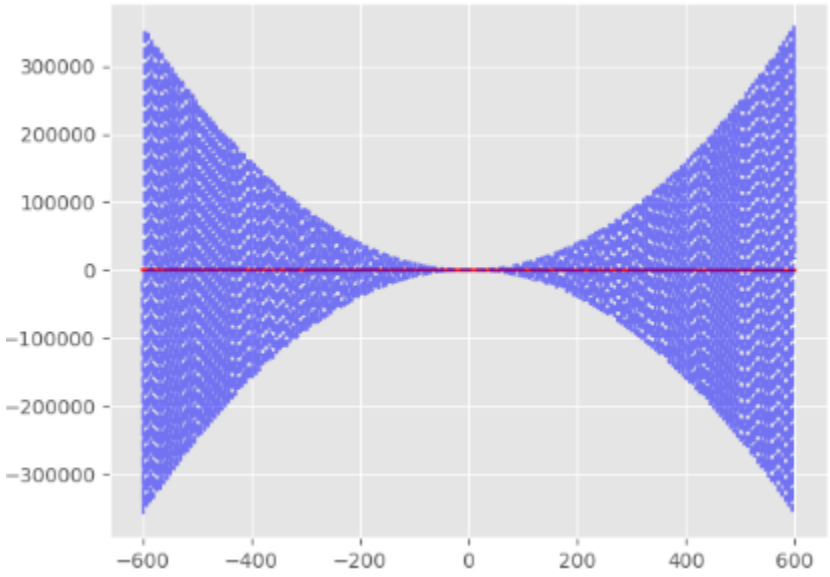
نتیجه مدل با eta = 0.00001:



که در هر دو مورد نتیجه ضعیفی حتی در همان بازه درون‌یابی آموزش را شاهد هستیم. این مورد بر اهمیت بسیار بالای تعیین نرخ یادگیری در اپتیمایزر در یادگیری مدل صحه می گذارد.

با افزایش وسعت دامنه تا زمانی که نسبت تعداد داده‌های آموزشی به بازی بزرگ باشد عملکرد مدل بسیار قوی و عالی است حتی در توابع بسیار پیچیده. از آن طرف اگر بازه آموزش به نسبت تعداد داده‌ها

خیلی بزرگ شود مدل دیگر دقت سابق را نخواهد داشت و تابع آموزش داده‌شده انعطاف لازم را مانند خود تابع پیچیده نخواهد داشت. برای مثال تابع پیچیده مثال ۴ام را در نظر می‌گیریم و ۱۰۰ داده آموزش در بازه ۵۰۰- تا ۵۰۰ به ورودی شبکه می‌دهیم. نتیجه زیر را می‌گیریم.



که عملا نتوانسته هیچ تخمینی از تابع در این بازه آموزشی بزرگ بزند. بنابراین نسبت این بازه به تعداد نقاط ورودی عامل مهمی است. اگر هم بازه را کوچک‌تر کنیم دقت بهتر می‌شود و مدل همچنان تقریب خوبی از تابع در همان بازه کوچک‌شده و نه در خارج آن ارائه می‌کند.

در این بخش اول توضیحات کد و همچنین بازی با تمام پارامترهای ممکن بسیار با تفصیل انجام شد. در بخش‌های بعدی توضیح کد منوط به تغییرات اعمال‌شده است و بیشتر به نتایج خواسته‌شده پرداخته شده است و همچنین به خاطر تشابه نتایج تغییرات در پارامترها، دیگر این تغییرات در بخش‌های بعدی بررسی نشده‌اند.

بخش دوم: تابع را با داده‌های نویزدار بیابیم

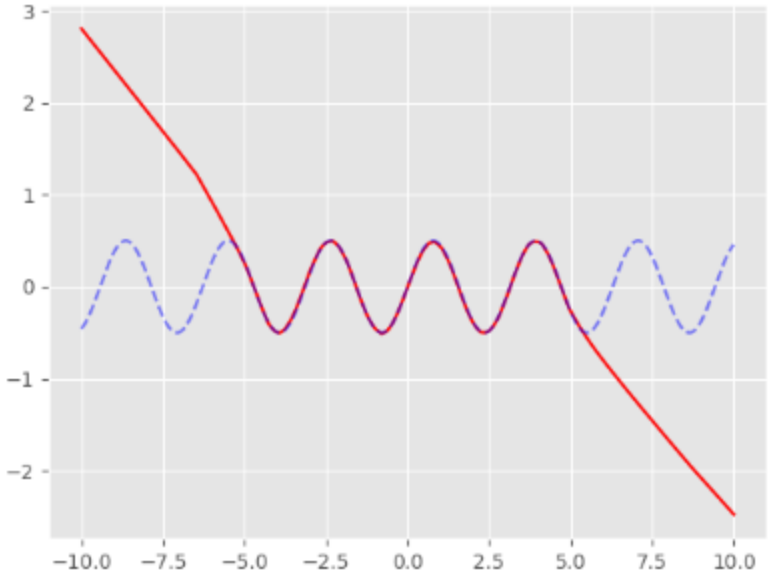
کد این بخش نیز کاملا مشابه بخش قبل است و ساختار شبکه همان قبلی است. صرفا بخش افزودن نویز به داده‌های آموزشی به DatasetCreator اضافه شده است که به صورت زیر عمل می‌کند:

```
def make_noisy(self, x):
    noise = (np.random.rand(*x.shape) - 0.5) * 2 * self.noise_co
    return np.multiply(1 + noise, x)
```

این تابع با دریافت عدد x و با داشتن یک ضریب ثابت c، مقداری در بازه x + cx تا x - cx به عنوان نویزدارشده آن خروجی می‌دهد. هرچه ضریب بزرگ‌تر باشد، مقدار نویز داده‌شده به دیتا بیشتر است.

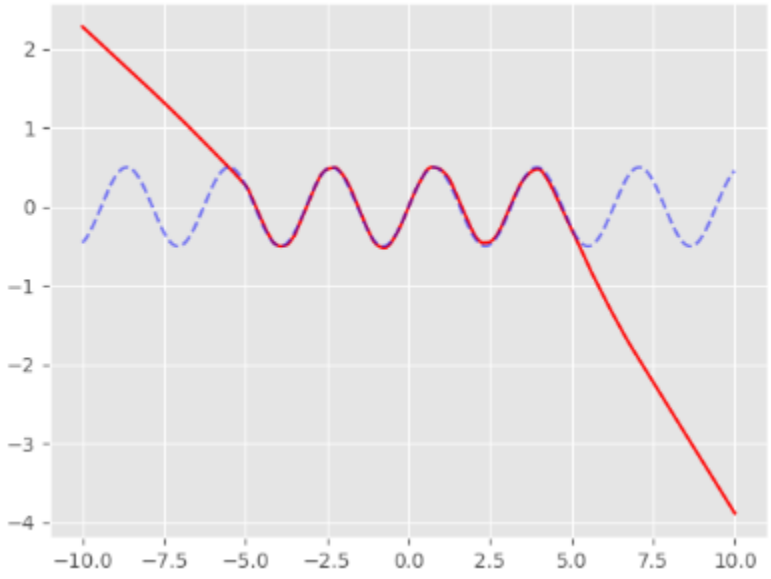
حال همان تابع مثال دوم بخش قبل یعنی $f(x) = sin(x) * cos(x)$ را به عنوان تابعی ثابت در نظر گرفته و در هر مرحله مقادیر مختلفی را به عنوان نویز به داده‌های آموزشی شبکه افزوده و قدرت شبکه در حضور داده نویزدار را می‌سنجیم.

حالت اول: ضریب نویز را 0.1 که مقدار کمی است در نظر می‌گیریم. باقی شرایط کاملا ثابت و مشابه بخش اول است.



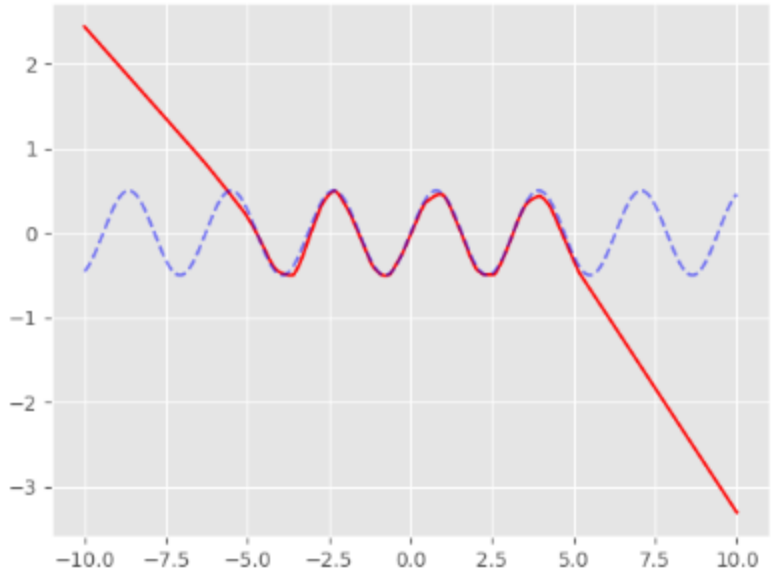
مقدار میانگین loss برابر با 0.0006 است. مشاهده می‌کنیم که تقریب زده شده خیلی نزدیک به همان بخش اول و داده‌های بی‌نویز است.

حالت دوم: ضریب نویز را 0.5 قرار می‌دهیم و باقی شرایط ثابت است.



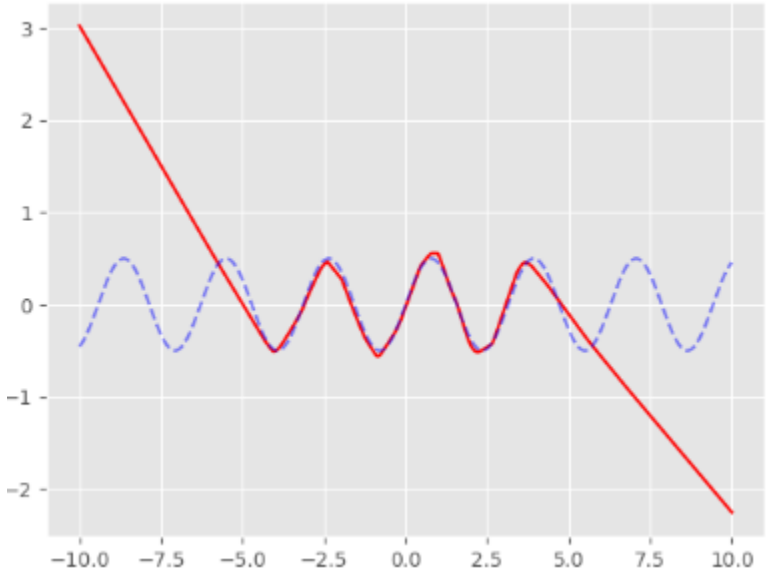
مقدار میانگین loss برابر با 0.011 می‌شود. افزایش زیادی نسبت به حالت اول به نظر می‌آید.

حالت سوم: این بار ضریب نویز را ۱ می‌گیریم(داده می‌تواند به اندازه خودش نویز داشته باشد)



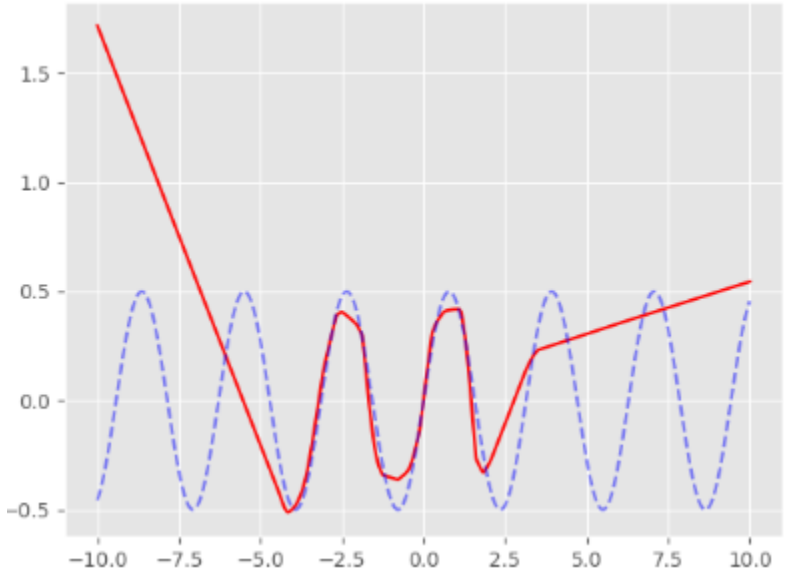
میانگین loss برابر با 0.047 است. کمی نسبت به بخش اول نادقیق‌تر است که البته خیلی تفاوت ناچیز است و غالبا در اطراف نقاط اکسترمم مشاهده می‌شوند. نویز کمی خودنمایی می‌کند.

حالت چهارم: ضریب نویز را ۳ می‌گیریم که مقدار خیلی زیادی است و ممکن است عملا خود داده‌ها را محو کند.



مقدار loss میانگین 0.367 است که برای این تابع زیاد است. نویز باعث شده تابع آموزش دیده شده با وجود فرم کلی مشابه تابع اصلی، نادقیقی‌ها و تیزی‌ها و عدم انعطاف‌های قابل توجهی داشته باشد و قدرت نویز کم کم بر قدرت شبکه غلبه می کند.

با ضریب ۸:



در کل از آزمایش‌های انجام‌شده می‌توانیم نتیجه بگیریم که تا وقتی نویز تا حد معقولی باشد، شبکه بسیار خوب می‌تواند تابع اصلی را از میان داده‌های دارای نویز درون‌یابی کند. وقتی مقدار نویز خیلی زیاد می‌شود و عملا باعث می‌شود خود داده‌ها کم‌رنگ شوند، شبکه عملکرد ضعیف‌تری از خود نشان می‌دهد و سخت‌تر می‌تواند تابع اصلی موردنظر را حدس بزند که البته طبیعی است و این موضوع باعث نمی‌شود قدرت شبکه زیر سوال برود، زیرا در اینجا انگشت اتهام به جای مدل باید به سمت داده‌های آموزش برود که چرا چنین نویزی در میان داده‌ها وجود دارد. بنابراین عملکرد شبکه مصنوعی در حضور نویز منطقی در این بخش هم بسیار خوب و معقول است.

بخش سوم: تابع را با چند ورودی بیابیم

کد این بخش بسیار مشابه کد بخش اول است. تفاوتی که وجود دارد این است که به دلیل سر و کار داشتن با ورودی‌های چندبعدی باید کارهایی که روی آرایه‌ها و... می‌کنیم با توجه به تعداد ورودی‌های تابع موردنظر انجام شود. برای همین تعداد ورودی‌های تابع را با عنوان num_of_features به کنستراکتور هر دو کلاس اصلی DatasetCreator و FunctionApprox اضافه می‌کنیم. نمونه‌ای از تغییر کارها با توجه به این ابعاد در زیر آمده است:

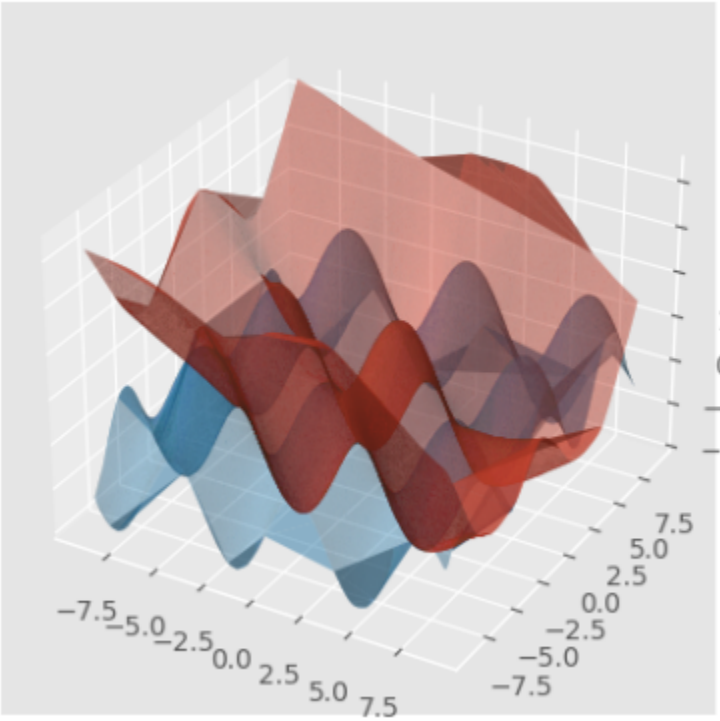
```
def get_train_data(self):
    X = np.random.uniform(self.train_min, self.train_max, (self.train_n_samples, self.num_of_features))
    labels = self.function(*X.T)
    p = np.random.permutation(self.train_n_samples)
    return X[p], labels[p]
```

همچنین دیگر تغییر اصلی ایجاد شده در کد مربوط به بخشی است که نتایج را می‌خواهیم روی نمودار نشان دهیم. در این بخش نموداری که رسم می‌کنیم به صورت ۳ بعدی و بر اساس دو ورودی اول تابع است. به این نکته هم توجه کنید که رسم این مدل نمودار در این تعداد زیاد نقطه کمی زمان‌بر تر از حالت دوبعدی است.

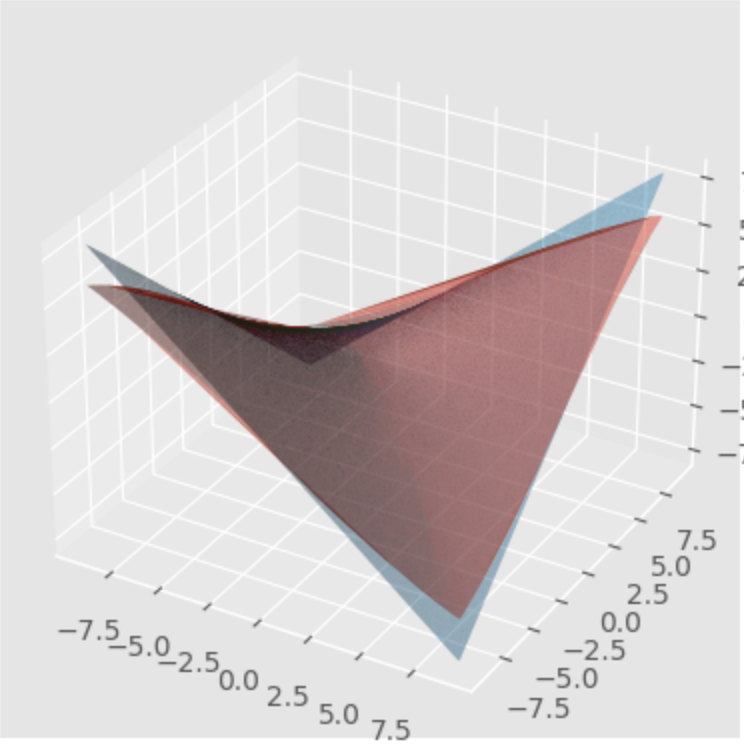
```
def plot_result(self):
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    test_X, test_y = self.dataset_creator.get_test_data()
    pred_y = self.model.predict(test_X)
    test_y = test_y.reshape(len(test_X[:, 0]))
    pred_y = pred_y.reshape(len(test_X[:, 0]))
    ax.plot_trisurf(test_X[:, 0], test_X[:, 1], pred_y, alpha=0.6)
    ax.plot_trisurf(test_X[:, 0], test_X[:, 1], test_y, alpha=0.5)
    return fig
```

باقی موارد مانند ساختار شبکه و مراحل یادگیری و ... ثابت و دقیقا مانند بخش اول است.

در آزمایش اول تابع با دو ورودی $f(x, y) = sin(x) + cos(y)$ را در نظر می گیریم.
داده‌های آزمایشی را در بازه (۵،-۵) و (۵،۵) و داده‌های آزمایشی را در بازه (۹،-۹) تا (۹،۹) تولید می کنیم. تعداد همان ۲۴۰۰۰ و ۲۰۰۰۰۰ است. نتیجه به صورت زیر است:

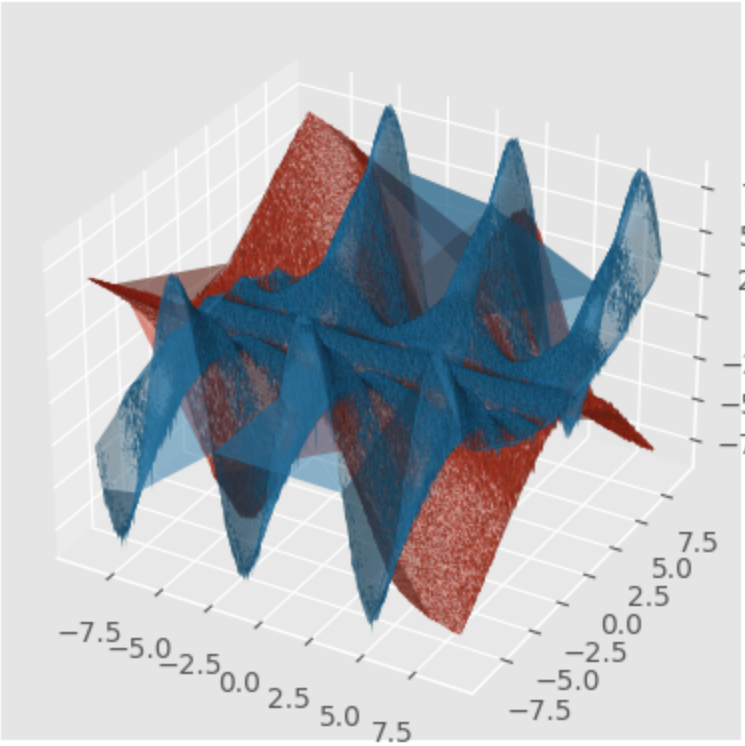


مقدار میانگین loss که همچنان MSE است برابر است با 0.0011 . عملکرد معقولی در بازه آموزش به نظر می‌آید.
در آزمایش دوم تابع دو ورودی $f(x, y) = xy$ را در نظر می گیریم. بازه‌های آموزشی و آزمایشی را دقیقا مانند تست اول در نظر می گیریم. نتیجه زیر حاصل می‌شود:

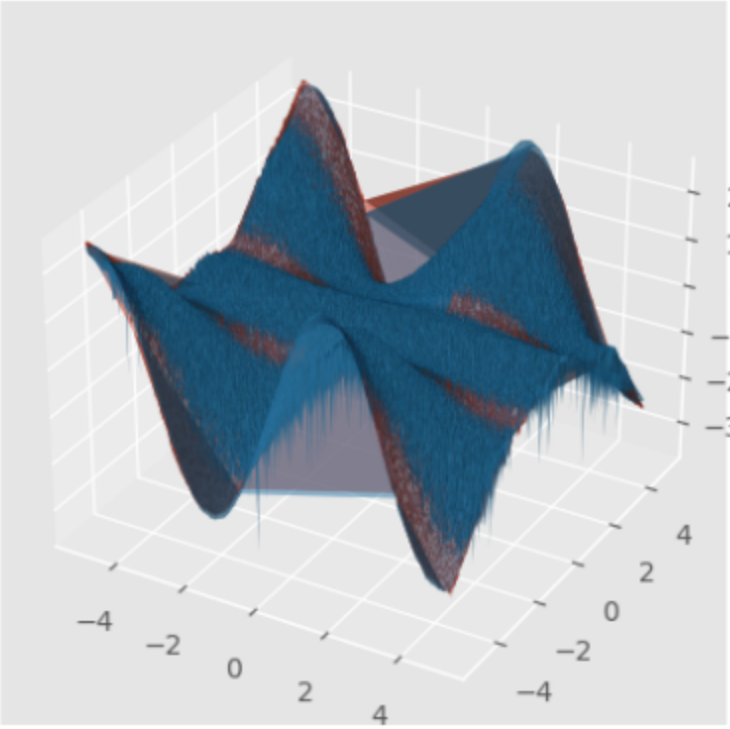


مقدار میانگین loss برابر با 0.0266 شده است. در این آزمایش هم تابع یادگرفته‌شده بسیار نزدیک به تابع اصلی است.

در آزمایش سوم و آخر یک تابع پیچیده سه‌بعدی به صورت $f(x, y, z) = sin(x) * y^2 + 3cos(y)^2 * ln(|z|)$ را در نظر می‌گیریم. ابتدا آن را با استفاده از ۲۴۰۰۰ نقطه از (۵-،۵-،۵-) تا (۵،۵،۵) برای داده‌های آموزشی و ۲۰۰۰۰۰ نقطه از (۹-،۹-،۹-) تا (۹،۹،۹) برای داده‌های آزمایشی آموزش داده و تست و رسم می‌کنیم. مجبوریم مقدار خروجی را صرفا بر اساس دو ورودی اول یعنی x و y انجام دهیم.



که نبود بعد z و همچنین پیچیدگی تابع و بزرگ‌تر بودن بازه آزمایش از آموزش باعث می‌شود میزان قدرت مدل در بازه درون‌یابی آموزشی به خوبی دیده نشود. مقدار میانگین loss در این آزمایش برابر با 0.762 بود. برای این که نشان دهیم مدل در بازه آزمایش به خوبی و با قدرت عالی تابع ۳ بعدی را تقریب زده این بار همین آزمایش را صرفا با بازه آموزشی و آزمایشی یکسان تکرار می‌کنیم. نتیجه را در زیر مشاهده می‌کنید:



که از این نما میزان انطباق مدل بر تابع واقعی در بازه آموزش کاملا آشکار است. در این آزمایش مقدار میانگین loss برابر با 1.021 بود.

در کل در این بخش نتیجه می‌گیریم که قدرت بالای شبکه عصبی محدود به یک ورودی تک‌بعدی برای تخمین تابع نمی‌شود و این قدرت در ابعاد ورودی بیشتر نیز تعمیم پید می‌کند. مشابه بخش اول در این ابعاد بالاتر نیز در ناحیه (فضا) آموزش مقدار انطباق و دقت مدل عالی و خارج از آن در نواحیکه نیاز به برون‌یابی دارند به دور از واقعیت و نادقیق است. در این بخش هم جنبه دیگری از قدرت شبکه‌های عصبی بر ما ثابت شد.

بخش چهارم: خط‌خطی را بیابیم

کلیت کد مشابه ۳ بخش قبل است. این بار کلاس DatasetCreator را به کل حذف می‌کنیم چون دیگر تابعی با ضابطه خاص نداریم که دیتاستی بر اساس آن تولید کنیم. در کنستراکتور کلاس اصلی FunctionApprox به جای dataset_creator مستقیما خود X و Y مربوط به فیچر و لیبل داده‌های آموزشی را پاس می‌دهیم.

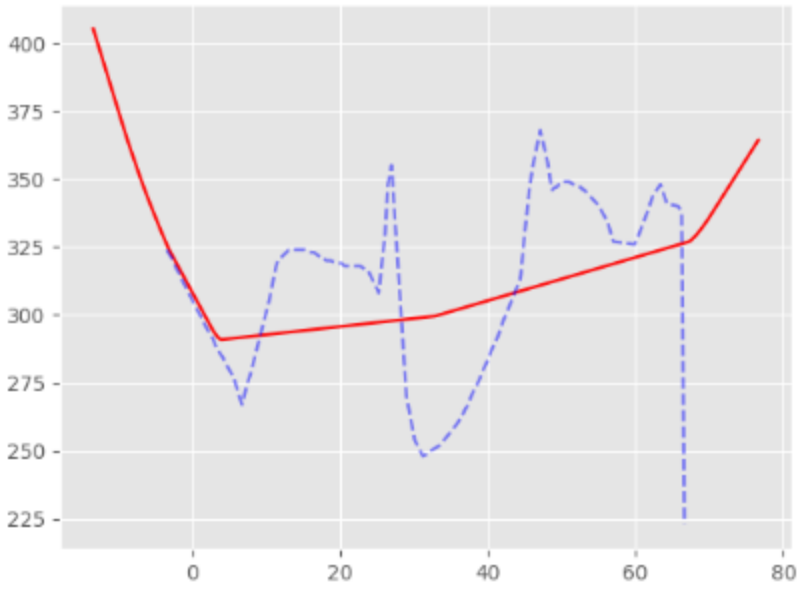
ضمنا در این بخش می‌خواهیم ساختار کوچک‌تری برای شبکه در نظر بگیریم.

```
def get_model(self):
    model = keras.Sequential()
    model.add(keras.layers.Dense(50, input_dim=1, activation='relu'))
    model.add(keras.layers.Dense(50, activation='relu'))
    model.add(keras.layers.Dense(1))

    model.compile(loss=self.loss_function, optimizer=self.optimizer)
    return model
```

شبکه جدید متشکل از دو لایه ۵۰ نورونی با تابع فعال‌سازی ReLU و یک تک‌نورون خطی خروجی برای ارائه خروجی رگرسیون است. در مجموع ۱۰۱ نورون داریم که برای یک شبکه عصبی ابعاد کم و کوچکی به حساب می‌آید.

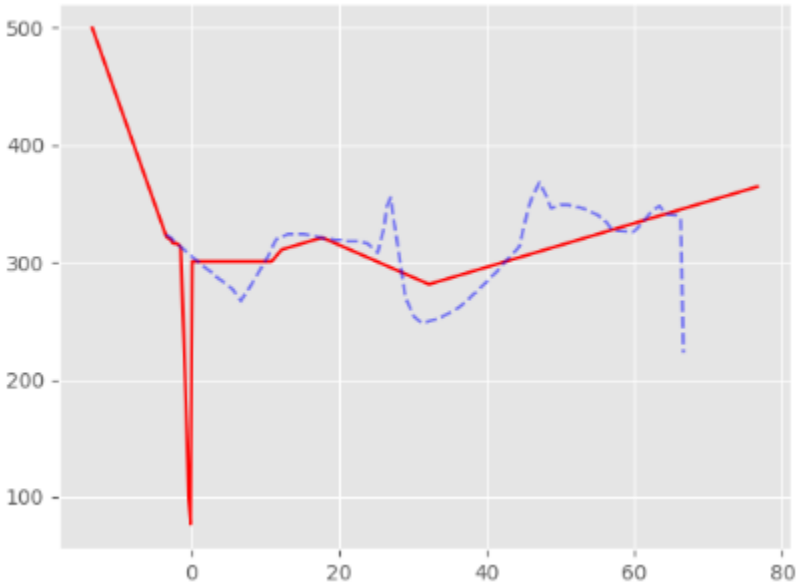
تغییر ایجاد شده دیگر در مقدار learning_rate است. الگوریتم بهینه‌سازی همچنان Adam است و loss همان MSE است ولی با توجه به اجراهایی که کردم مقادیر 0.001 و 0.1 هر دو نتیجه ضعیفی به دست دادند. بنابراین در این بخش برخلاف بخش‌های گذشته از نرخ 0.01 به جای 0.001 استفاده کرده‌ام که کاهش ابعاد داده‌ها و همچنین کوچک‌تر شدن شبکه می‌تواند از عواملی باشد که مسبب این تغییر شده‌اند.

در حالت نرخ یادگیری 0.001:



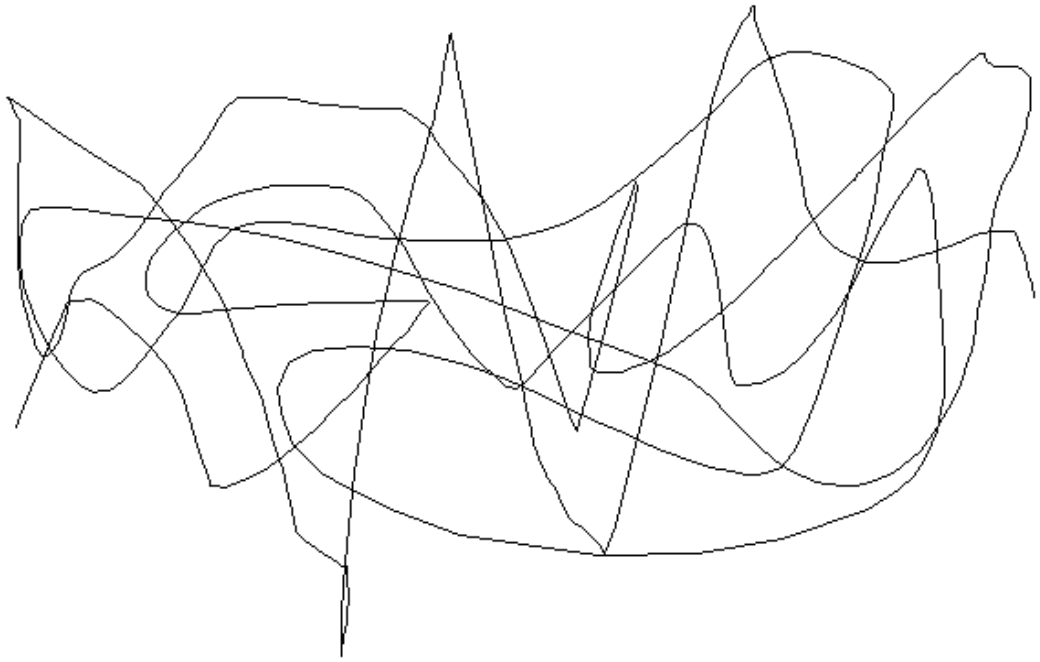
که ضعف بدی در بهینه‌سازی کردن را نشان می‌دهد.

در حالت نرخ یادگیری 0.1:

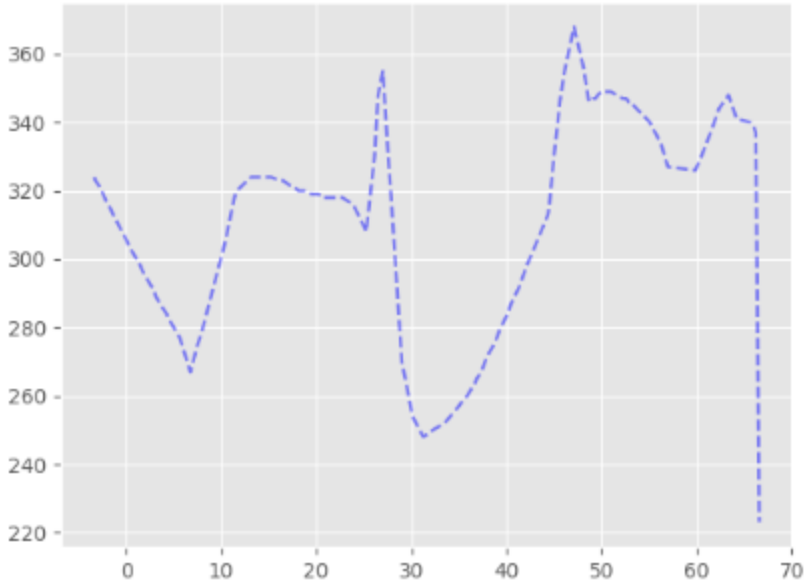


که در این نرخ بالا هم بهینه‌سازی و به تبع آن یادگیری به صورت ضعیف دنبال می‌شود.

یک تغییر اصلی دیگر در این بخش در نحوه بیان تابع و دادگان آموزشی است. چالش زیادی در تبدیل خط‌خطی دستی به یک تابع با یک‌سری مختصات مشخص داشتم و وقت زیادی صرف آن کردم. خط‌خطی اصلی من به صورت زیر است:



با تحمل کلی مشقت و استفاده از ابزارهای کار با عکس PILLOW و نامپای و ... موفق شدم یک فایل CSV متشکل از ۱۴۰ نقطه از بالای این خط‌خطی جدا کنم. با رسم آن شکل زیر حاصل می‌شود:

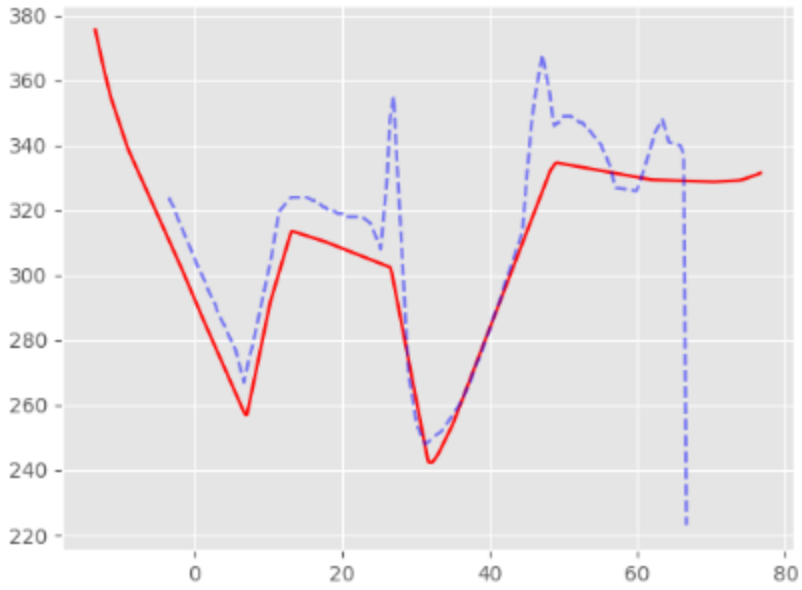


که به اندازه کافی پرت و بدفرم است! در ادامه فایل csv ساخته شده را می‌خوانیم و مقادیر آن را به مدل می‌دهیم تا آموزش انجام شود.

```
df = pd.read_csv('function_pt4.csv')
f_predictor = FunctionApprox(X=df['x'].to_numpy(), y=df['y'].to_numpy(), batch_size=32, num_epochs=450)
f_predictor.train()
fig = f_predictor.plot_result()
plt.show()
plt.savefig('part_4.png')
```

دقت می‌کنیم که تعداد epoch را افزایش زیادی داده‌ایم و ۴۵۰ قرار داده‌ایم. با توجه به این که هم حجم داده‌ها و هم سایز شبکه کاهش یافته است هر epoch تایم خیلی کمی می‌گیرد. بنابراین برای دستیابی به دقت بالاتر تعداد آن را افزایش می‌دهیم.

حال با اجرای یادگیری روی مدل نتیجه زیر به دست می‌آید:



می‌بینیم که شبکه توانسته با وجود تعداد کم نورون و داده ورودی، فرم کلی آن خط‌خطی را تا حد خوبی تشخیص دهد اما در نقاط زیادی ناهماهنگی‌هایی بین تابع اصلی و تابع یادگرفته‌شده دیده می‌شود. این تابع پر از پرش‌های ناگهانی است و به وضوح شاهد هستیم بیشتر عملکرد منفی مدل در این نواحی دارای پرش بوده است و بنابراین این پرش‌های ناگهانی در توابع ملاک مهمی هستند که مدل عصبی تا چه حد خوب بتواند آن‌ها را تشخیص دهد. اما در کل ماجرا، شبکه عصبی سبک ما در این بخش هم نسبتا قوی ظاهر شد و فرم کلی خط‌خطی را به خوبی تشخیص داد.

بخش پنجم: این دیگر کدام عدد است؟

ساختار کد همچنان به شکلی مشابه است. به تغییرات می‌پردازیم. کلاس DatasetCreator را حذف کرده‌ایم. برای استفاده از دیتاست ارقام دست‌نویس mnist آن را از api کتابخانه keras دریافت و در صورت برخورد مشکل دسترسی به api آن را از فایل لوکال کنار پروژه لود می کنیم. کد زیر مربوط به لود و پردازش دیتاست است:

```
if __name__ == '__main__':
    try:
        (X_train, y_train), (X_test, y_test) = keras.datasets.mnist.load_data()
    except:
        with np.load('mnist.npz', allow_pickle=True) as f:
            X_train, y_train = f['x_train'], f['y_train']
            X_test, y_test = f['x_test'], f['y_test']
    X_train = X_train.reshape(X_train.shape[0], -1)
    X_test = X_test.reshape(X_test.shape[0], -1)
    X_train = X_train.astype('float32')
    X_test = X_test.astype('float32')
    X_train /= 255
    X_test /= 255
    classifier = DigitImageClassifier(batch_size=32, num_epochs=6)
    classifier.train(X=X_train, y=y_train)
    classifier.evaluate_model(X=X_test, y=y_test)
```

بعد از لود مجموعه‌داده‌ها و لیبل‌های آموزشی و آزمایشی، عکس‌ها را از حالت دوبعدی ۲۸ در ۲۸ خارج کرده و به صورت مسطح ۷۶۴ بعدی در می‌آوریم تا راحت‌تر شبکه را تغذیه کنیم. دیتاتایپ‌ها را به اعشاری تبدیل و با تقسیم عدد هر پیکسل عکس به ۲۵۵ آن را نرمال و به بازه ۰ تا ۱ می‌بریم. پس از این مراحل دادگان به ترتیب در مراحل آموزش و پیش‌بینی به مدل داده می‌شوند.

کلاسی که مدل در آن قرار دارد به DigitImageClassifier تغییر نام داده است. همچنان از Cross Validation هنگام آموزش مدل استفاده می‌شود.

همچنین ساختار جدید شبکه برای عملیات دسته‌بندی به جای رگرسیون، به صورت زیر شده است:

```
def get_model(self):
    model = keras.Sequential()
    model.add(keras.layers.Dense(512, input_dim=28*28, activation='relu'))
    model.add(keras.layers.Dropout(0.2))
    model.add(keras.layers.Dense(512, activation='relu'))
    model.add(keras.layers.Dropout(0.2))
    model.add(keras.layers.Dense(10, activation='softmax'))

    model.compile(loss=self.loss_function, optimizer=self.optimizer, metrics=['accuracy'])
    return model
```

این بار در شبکه از دو لایه ۵۱۲ نورونی با تابع فعال‌ساز ReLU استفاده شده است. لایه اول هر نورون به ۷۸۴ ورودی مربوط به عکس متصل است. بعد از هر کدام از این لایه‌ها یک لایه Dropout با احتمال 0.2 قرار داده‌شده است. این لایه این طور عمل می‌کند که هر نورون را با احتمال 0.2 غیر فعال می‌کند و در یادگیری شرکت نمی‌دهد. این عمل نوعی منظم‌سازی یا اصطلاحا regularization کردن مناسب برای شبکه‌های عصبی است که در اینجا برای جلوگیری از ایجاد overfit در کلاس‌بندی‌های انجام‌شده قرار داده شده است. لایه آخر خروجی شامل ۱۰ نورون با تابع فعال‌ساز softmax است که دقیقا مناسب یک مسئله دسته‌بندی چندکلاسه است. هر نورون i در واقع خروجی این که احتمال تعلق عکس به کلاس i چقدر است را تولید می‌کند و در نهایت از روی بیشترین احتمال در بین ۱۰ نورون، کلاسی که حدس می‌زنیم داده مربوط به آن است را پیش‌بینی می‌کنیم. همچنین در هنگام کامپایل مدل، یک متریک جدید یعنی accuracy را به مدل معرفی می‌کنیم تا علاوه بر loss این مورد را هم در نظر داشته باشد و بهبود بخشد زیرا در این پروژه به دنبال دقت زیاد در تشخیص کلاس‌ها هستم.

با توجه به این که دیگر در یک مسئله رگرسیون نیستیم و با یک دسته‌بندی چندکلاسه سر و کار داریم، نیاز است ملاک خود برای محاسبه loss را تغییر دهیم. کتابخانه keras یک نمونه از cross entropy را که مخصوص دسته‌بندی‌هایی هست که هر داده فقط به یک دسته و نه بیشتر تعلق دارد را ارائه میکند که به نام sparse_categorical_crossentropy است. این معیار loss به نظر کاملا منطبق بر نیاز مسئله فعلی ما است.

```
def evaluate_model(self, X, y):
    self.model.evaluate(X, y)
    pred_y = self.model.predict(X)
    y_pred_bool = np.argmax(pred_y, axis=1)
    print(classification_report(y, y_pred_bool))
    self.plot_mis_classification_ex(X, y, y_pred_bool)

def plot_mis_classification_ex(self, X, y_true, y_pred):
    for i in range(len(y_pred)):
        if y_pred[i] != y_true[i]:
            fig = plt.figure()
            image = X[i].reshape(28, 28)
            plt.imshow(image)
            plt.show()
            plt.savefig('part_5.png')
            print('\nMisClassified example:')
            print(f'Actual: {y_true[i]}')
            print(f'Predicted: {y_pred[i]}')
            break
```

تابع evaluate_model به ازای عکس‌های دادگان آزمایشی، رقم مربوط به هر یک را با استفاده از مدل آموزش‌دیده پیش‌بینی می‌کند و نهایتا گزارش score های مختلف را با توجه به مقادیر ارقام واقعی و مقادیر پیش‌بینی‌شده چاپ می‌کند. یک نمونه از داده‌های اشتباه تشخیص داده‌شده را نیز چاپ می‌کند تا موارد ضعف مدل را بررسی کنیم.

حال مدل را روی حدود ۶۰۰۰۰ عکس ارقام آموزشی آموزش می‌دهیم. پس از آموزش در یکی از fold ها به دقت 98.15 رسیدیم و در کل به میانگین دقت روی داده آموزشی 97.79 دست پیدا کردیم.

```
Average Loss: 0.08518759161233902 , Average Accuracy: 97.78666615486145%
```

و در داده‌های آزمایشی دقت خوب 97.75 درصد به دست آمد.

```
loss: 0.0799 - accuracy: 0.9775
```

مقادیر loss نیز در تصاویر مشهود است.

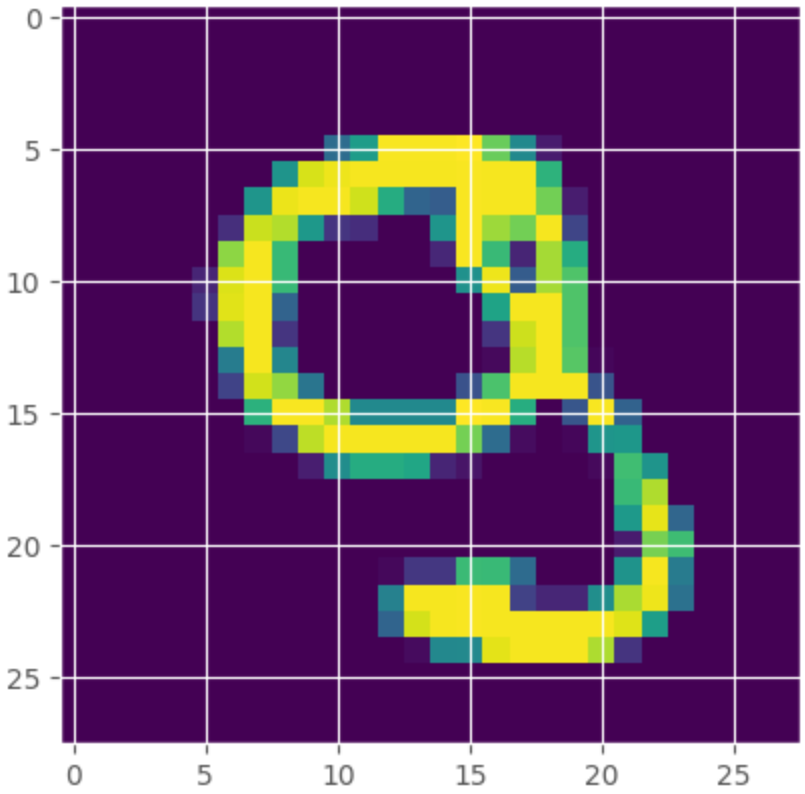
با توجه به این آمارها عمده داده‌ها به خوبی کلاس‌بندی شده‌اند و با وجود دقت خیلی بالا، overfit رخ نداده و داده‌های تست هم همپای داده‌های آموزش با دقت کلاس‌بندی می‌شوند. گزارش کامل سایر score ها مانند fscore و ... به صورت کلی و به ازای هر کلاس در زیر آمده است:

	precision	recall	f1-score	support
0	0.98	0.98	0.98	980
1	0.99	0.99	0.99	1135
2	0.98	0.98	0.98	1032
3	0.98	0.97	0.97	1010
4	0.98	0.98	0.98	982
5	0.98	0.97	0.98	892
6	0.98	0.98	0.98	958
7	0.96	0.98	0.97	1028
8	0.97	0.97	0.97	974
9	0.98	0.96	0.97	1009
accuracy			0.98	10000
macro avg	0.98	0.98	0.98	10000
weighted avg	0.98	0.98	0.98	10000

که نشان می‌دهد مدل شبکه عصبی در همه کلاس‌ها به صورت متعادل، تمامی scoreهای معروف مانند percision , recall و ... را با مقدار زیادی می گذراند. در نتیجه قدرت شبکه عصبی به رگرسیون محدود نشده و در این قسمت به خوبی نشان داد که دسته‌بند توانایی نیز هست.

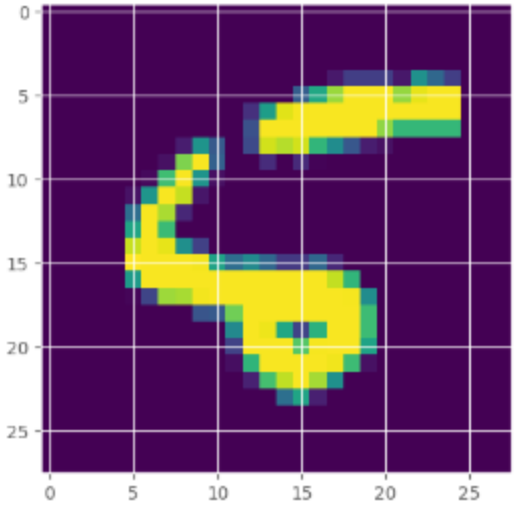
حال از بین آن درصد کمی که اشتباه کلاس‌بندی شده‌اند، می‌خواهیم دو نمونه را بررسی کنیم:

نمونه اول:



که این رقم در واقعیت ۹ است ولی مدل آن را ۸ پیش‌بینی کرده است. دلیل آن می‌تواند نزدیک بودن دنباله انتهایی عدد به مرکز آن باشد که آن را خیلی شبیه به فرم کلی ۸ می‌کند و مدل به اشتباه افتاده است.

نمونه دوم:



این عدد دارای برچسب واقعی ۵ است ولی مدل آن را ۶ پیش‌بینی کرده است. البته با توجه به دست‌خط خیلی بد این تصویر، حتی انسان هم ممکن است به اشتباه بیفتد و نتواند کلاس این عدد را به خوبی تشخیص دهد.

بنابراین حتی این نمونه‌هایی هم که به اشتباه دسته‌بندی شده‌اند، اکثرا اشتباهات خیلی نزدیک و انسان‌گونه‌ای است و نمی‌توان مدل را به ضعف متهم کرد.

با کاهش تعداد پارامتر epoch از ۶ به ۲، یادگیری سریع‌تر در زمان نزدیک به ۳ دقیقه رخ داده (در حالت عادی ۱۰ ۱۱ دقیقه فرایند آموزش طول کشید) ولی دقت به طور میانگین 0.6 درصد کاهش داشته و به ۹۷.۲ رسیده است. با افزودن تعداد این پارامتر صرفا زمان بیشتری نسبت به حالت عادی صرف یادگیری شد و دقت مدل تغییر خاصی نکرد.

با کاهش batch_size به ۱ با این که به دقتی نزدیک به دقت فعلی رسیدیم ولی این موضوع با هزینه نزدیک به ۱ ساعت زمان یادگیری انجام شد که نشان می‌دهد استفاده از batch بسیار سودمندانه‌تر از stochastic در این مسئله است. افزایش مقدار batch_size به ۱۲۸ دقت را به ۹۸ افزایش و یادگیری را سریع‌تر نزدیک به 4 دقیقه انجام داد.

با کاهش تعداد داده‌های آموزشی ورودی به ۱۰۰۰۰ آموزش در تقریبا ۱ دقیقه انجام و دقت 95.87 درصدی در داده‌های آموزشی به طور میانگین و دقت 95.7 درصدی در داده‌های آموزشی به

دست آمد. کاهش حجم داده‌های آموزشی منجر به سریع‌تر شدن فرایند آموزش و کمی کاهش دقت شد. با کاهش سایز به ۱۰۰۰ دقت به ۸۸ درصد می‌رسد که البته همچنان دقت خوبی‌است و مهر تاییدی دیگر بر قدرت شبکه عصبی است.

در آزمایشی دیگر این بار بر روی ساختار شبکه عصبی‌مان تغییری ایجاد کرده و لایه‌های Dropout را حذف و تعداد نورون‌هایه هر لایه را از ۵۱۲ به ۱۲۸ کاهش می‌دهیم. دقت کاهش ۰.۳ درصدی داشته و به 97.4 می‌رسد که البته هنوز خیلی خوب است و شبکه کوچک‌تر شده است. البته تعداد زیاد داده آموزش همچنان تاثیر زیادی بر خوب بودن دقت دارد.

یک لایه را هم کامل حذف می‌کنیم و فقط یک لایه ۱۲۸تایی و لایه سافت‌مکس باقی می‌ماند. دقت همچنان همان 97.4 می‌ماند!! در اینجا شبکه عصبی با وجود کوچک بودن به دلیل تغذیه مناسب به خوبی آموزش دیده است و همچنان دقت بالای خود را حتی با این ساختار کوچک حفظ می‌کند. زمان کلی یادگیری هم نسبت به حالت عادی کاهش می‌یابد.

باقی پارامترها آنچنان محلی برای صحبت ندارند و در بخش‌های قبل صحبت راجع به آن‌ها انجام شده است و در این بخش هم به همان صورت تاثیرگذار هستند.

بخش ششم: بیرون کشیدن عدد از میان نویزها

همچنان کد این بخش از ساختار بخش‌های قبلی پیروی می‌کند. یک کلاس به نام NoisyDataset برای تهیه یک دیتاست از تصاویر با نویز قابل تنظیم به کد اضافه کرده‌ایم.

```
class NoisyDataset:
    LOW, MEDIUM, HIGH, VERY_HIGH = 0, 1, 2, 3

    def __init__(self, noise_level, train_n_samples, test_n_samples) -> None:
        self.noise_level = noise_level
        self.train_n_samples = train_n_samples
        self.test_n_samples = test_n_samples
        try:
            X_train, _, _, _ = keras.datasets.mnist.load_data()
        except:
            with np.load('mnist.npz', allow_pickle=True) as f:
                X_train = f['x_train']

        X_train = X_train.astype('float32')
        X_train /= 255
```

عملیات لود کردن دیتاست mnist دقیقا به همان صورت بخش قبل در کنستراکتور صورت گرفته است. پس از لود عملیات‌های زیر روی دیتاست انجام می‌شود. (چون فقط به یک مجموعه از عکس‌ها و نه حتی برچسب ارقامشان احتیاج داریم فقط X_Train را استفاده می‌کنیم)

```
p = np.random.permutation((len(X_train)))
all_img = X_train[p]
all_img = all_img[0: self.train_n_samples + self.test_n_samples]
self.train_labels = all_img[0:self.train_n_samples]
self.test_labels = all_img[self.train_n_samples: self.train_n_samples+self.test_n_samples]
noisy_images = np.array(list(map(lambda x: self.make_noisy(x), all_img))).reshape(-1, 28, 28)

self.train_images = noisy_images[0:self.train_n_samples]
self.test_images = noisy_images[self.train_n_samples: self.train_n_samples + self.test_n_samples]

self.train_images = self.train_images.reshape(self.train_images.shape[0], -1)
self.test_images = self.test_images.reshape(self.test_images.shape[0], -1)
self.train_labels = self.train_labels.reshape(self.train_labels.shape[0], -1)
self.test_labels = self.test_labels.reshape(self.test_labels.shape[0], -1)
```

ابتدا آن‌ها را با جایگشتی رندوم درهم می‌زنیم تا پخش شوند. سپس به تعدادی معین که مثلا در این قسمت ۱۰۰۰۰ است از ابتدای این جایگشت عکس‌ها را برداشته و به عنوان لیبل داده‌های خود برمی‌گزینیم (جون این داده‌ها اصلی و بدون نویز هستند). تعداد کمی هم (مثلا ۱۰) برای تست چشمی عملیات رفع نویز انتخاب می‌کنیم و به عنوان برچسب داده‌های آزمایش قرار می‌دهیم. سپس به تمام عکس‌ها به همان ترتیبی که بودند مقداری نویز با شدت معلوم وارد کرده و دقیقا ۱۰۰۰۰ تای اول عکس‌های نویز دار را به عنوان داده‌های آموزش و ۱۰تای بعد را به عنوان داده‌های آزمایش انتخاب می‌کنیم نهایتا هم برای کار راحت تر با آن‌ها آن‌ها را به صورت مسطح به جای دو بعدی در می‌آوریم.

```
def make_noisy(self, image):
    res = image
    if self.noise_level >= NoisyDataset.LOW:
        res = skimage.util.random_noise(image)
    if self.noise_level >= NoisyDataset.MEDIUM:
        res = skimage.util.random_noise(res)
        res = skimage.util.random_noise(res)
    if self.noise_level >= NoisyDataset.HIGH:
        res = skimage.util.random_noise(res)
        res = skimage.util.random_noise(res, clip=False)
    if self.noise_level >= NoisyDataset.VERY_HIGH:
        res = skimage.util.random_noise(res, mode='s&p', clip=False)

    return res

def get_train_dataset(self):
    return self.train_images, self.train_labels

def get_test_dataset(self):
    return self.test_images, self.test_labels
```

تابع make_noisy یک عکس و یک حد نویز را ورودی گرفته و بسته به شدت خواسته شده نویز به عکس وارد کرده و سپس آن را خروجی می‌دهد. نویز اعمال شده به عکس‌ها در سه سطح اول از جنس گاوسی با شدت‌های مختلف و در سطح آخر ترکیبی از گاوسی و salt & pepper است که نویز سنگین‌تری است. از کتابخانه skimg آماده برای ایجاد نویز در عکس استفاده شده است.

کلاس اصلی که مدل در آن قرار دارد که نام آن به DigitImageDeNoiser تغییر کرده است غالبا مشابه قبل است. مراحل آموزش همچنان به همان صورت سابق انجام می‌شود. معیار loss دوباره در این مسئله به MSE برمی گردد و همچنان از الگوریتم بهینه‌ساز آدام با نرخ یادگیری ۰.۰۰۰۱ استفاده می‌شود.

نحوه ساخت دیتاست و استفاده آن در مدل نیز به این صورت است:

```
dataset = NoisyDataset(noise_level=NoisyDataset.LOW, train_n_samples=10000, test_n_samples=10)
train_set = dataset.get_train_dataset()
test_set = dataset.get_test_dataset()

model = DigitImageDeNoise(batch_size=32, num_epochs=10)
model.train(X=train_set[0], y=train_set[1])
fig = model.evaluate_model(X=test_set[0], y=test_set[1])
```

حال سوال بزرگ و اصلی که پیش روی ما هست این است که آیا شبکه عصبی توان یادگیری برای عملیات رفع نویز را دارد؟ از لحاظ تجربی با توجه به قدرتی که تا به حال در همین پروژه از این مدل دیده‌ایم می‌توان حدس زد که قطعا چنین چیزی برای شبکه امکان‌پذیر است. از طرف دیگر طبق تئوری هر شبکه عصبی به اندازه بزرگ می‌تواند هر تابع یا مسأله دلخواه را حل و یا شبیه کند. بنابراین پاسخ این سوال یک بله بزرگ است. سوال اصلی‌تر این است که چگونه؟ قطعا باید به دنبال یک معماری خوب برای شبکه عصبی خود باشیم تا بتواند جوابگوی این مساله باشد.

با نمونه مشابهی از این مسئله در جبر خطی مواجه شده بودم. در آنجا یک راهکار رفع نویز این بود که با استفاده از تجزیه SVD عکس اصلی را به یک سری مولفه مهم کاهش داده یا اصطلاحا سریالایز می کنیم به گونه‌ای که بخش باارزش اطلاعات آن حفظ شود و اطلاعات نامهم بیشتر از دست بروند. سپس با برگرداندن تجزیه SVD به شکل اولیه ماتریس عکس، نویزهای عکس تا حد خوبی از بین رفته بود. در اینجا نیز به ساختار مشابهی نیاز داریم تا اطلاعات اصلی و کلیدی و با ارزش عکس را از ۷۸۴ پیکسل اصلی آن استخراج و در تعداد کمتری فضا سریال کند . مجدد آن را به همان ۷۸۴ پیکسل دیسریلایز کند. پس یک معماری خوب این است که یک لایه در ابتدا با تعداد کمتری نورون مثلا ۶۴ و با تابع فعال‌ساز ReLU قرار دهیم که تمام ۷۸۴ پیکسل عکس نویزدار ورودی به آن‌ها

متصل هستند و این نورون‌ها نقش سریالایزر را برای ما بازی کنند. سپس یک لایه با ۷۸۴ نورون سیگموییدی به عنوان لایه خروجی بگذاریم تا دقیقا به یک عکس خروجس با پیکسل‌های نرمال‌شده مپ شود. این لایه نقش دیسریالایزر را ایفا می‌کند. مدل به صورت زیر می‌شود:

```
def get_model(self):
    model = keras.Sequential()
    model.add(keras.layers.Dense(64, input_dim=784, activation='relu'))
    model.add(keras.layers.Dense(784, activation='sigmoid'))

    model.compile(loss=self.loss_function, optimizer=self.optimizer)
    return model
```

که پیاده‌سازی همان توضیحات بالا است.

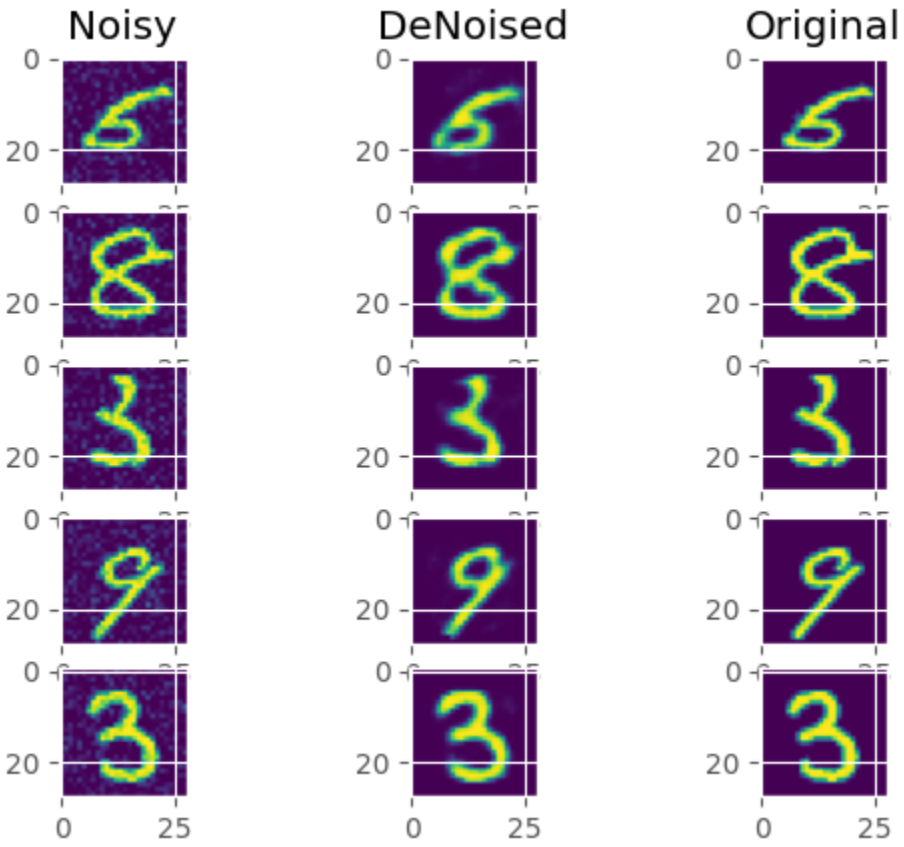
در نهایت تابع evaluate کردن مدل به صورت زیر نوشته شده است:

```
def evaluate_model(self, X, y):
    pred_y = self.model.predict(X)
    d = X.shape[0]
    fig = plt.figure(dpi=100)
    axs = fig.subplots(d, 3)

    for i in range(d):
        axs[i, 0].imshow(X[i].reshape(28, 28))
        axs[i, 1].imshow(pred_y[i].reshape(28, 28))
        axs[i, 2].imshow(y[i].reshape(28, 28))

    axs[0, 0].set_title("Noisy")
    axs[0, 1].set_title("DeNoised")
    axs[0, 2].set_title("Original")
    return fig
```

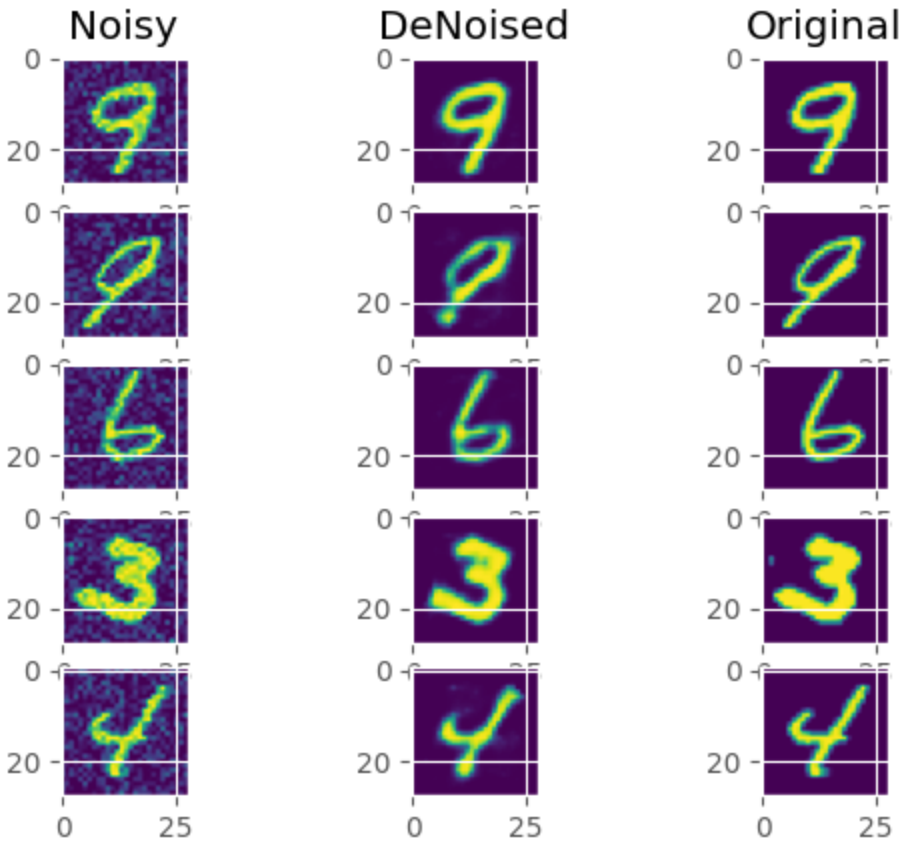
که پس از انجام پیش‌بینی و رفع نویز عکس‌های آزمایشی، نتایج آن‌ها را به ترتیب در هر سطر نمایش می‌دهد. در هر سطر به ازای یک داده، نسخه original (اصلی)، نسخه noisy (دارای نویز) و نسخه denoised (رفع نویز شده) به نمایش در می‌آیند.

حال نتایج یادگیری مدل را در زیر مشاهده می‌کنیم:

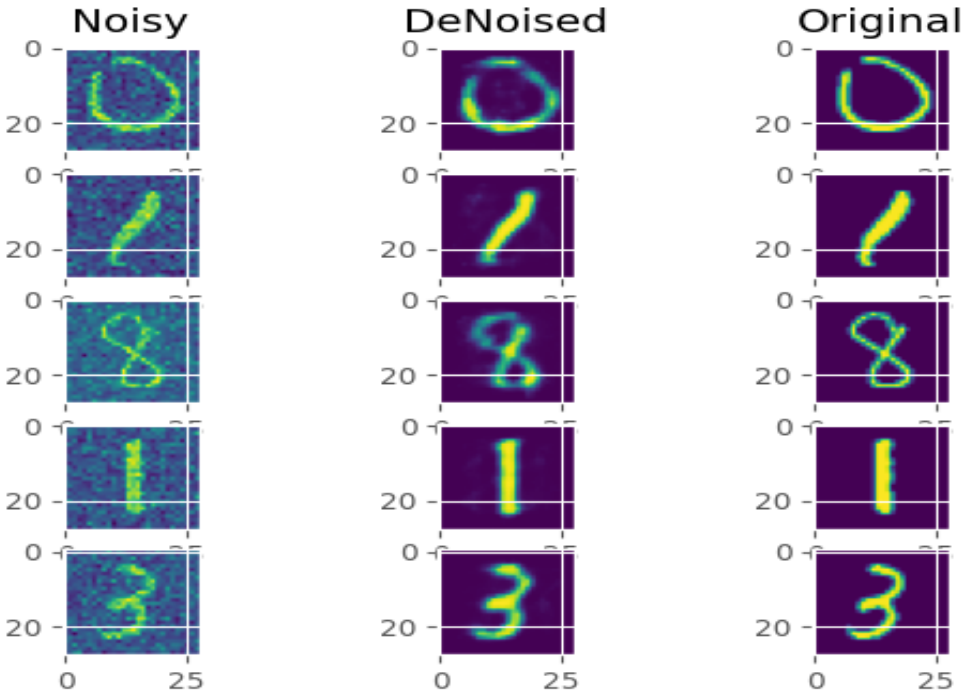
حالت اول: نویز کم loss = 0.012



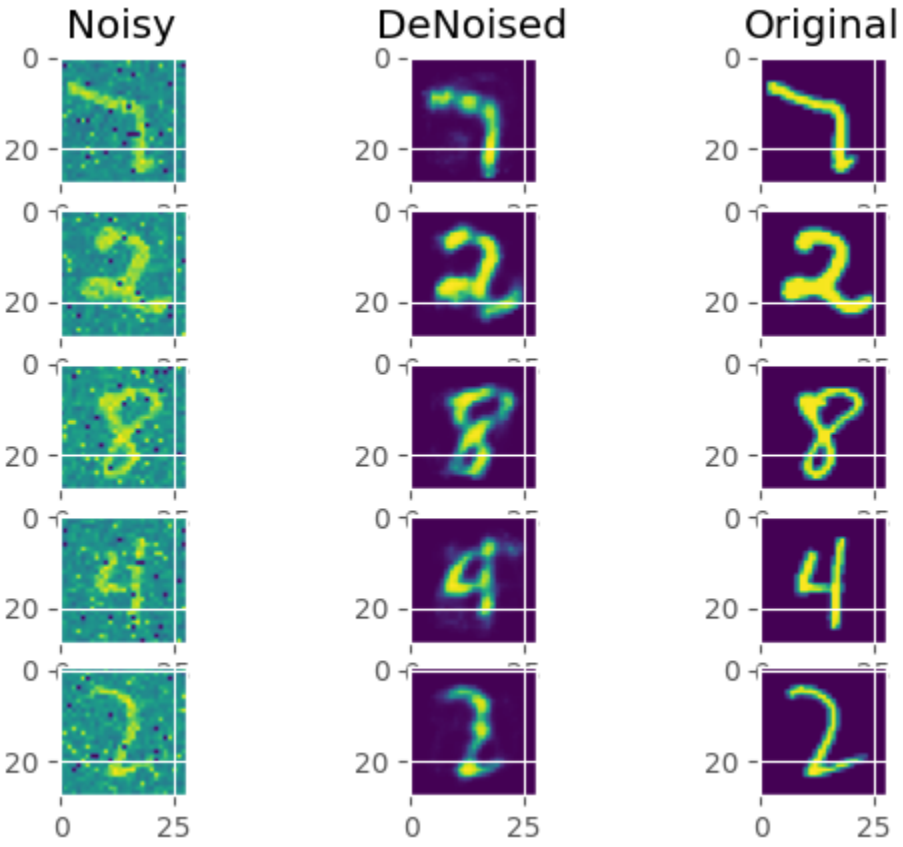
حالت دوم: نویز متوسط loss = 0.014



حالت سوم: نویز زیاد loss = 0.016



حالت چهارم: نویز خیلی زیاد loss = 0.021



مشاهده می‌کنیم که شبکه عصبی عملیات رفع نویز را با کیفیتی خیلی خوب و دقیق انجام می‌دهد. خیلی نمونه‌های کمی هستند که رفع نویز شده آن‌ها فرق بزرگی با نمونه اصلی آن داشته باشد. بنابراین شبکه عصبی در این بخش یک کاربرد قدرتمند دیگر خود در بازسازی تصاویر را نشان داد.

چالش‌ها

از لحاظ کار با کتابخانه‌ها و توابع کتابخانه‌ای با اکثر این کتابخانه‌ها آشنا و از قبل کار کرده بودم و چالشی نداشتم. در تبدیل خط‌خطی به یک تابع قابل قبول با چالش‌های زیادی مواجه شدم. این که چگونه این نقاط را به دست بیاورم؟ تخمین چشمی که خطا و محدودیت زیادی داشت. بنابراین یک نسخه png از آن تهیه و با استفاده از کتابخانه PILLOW سعی کردم نقاطی از عکس را که تیره هستند تشخیص و مکان‌یابی کنم. سپس از هر ستونی تقریبا بالاترین نقطه سیاهش رو برداشتم تا یک تابع خوب به دست بیاید. سپس نقاط به دست‌آمده را در یک فایل functio_pt4.csv ریختم تا در مراحل بعدی بازمصرف شود.

چالش دیگر تصمیم برای ساختار شبکه عصبی بود. این که لایه‌ها چند نورونه باشند و چند لایه داشته باشیم و چه توابع فعال‌سازی استفاده کنیم. بعضا best practice هایی از اینترنت پیدا می‌کردم یا با آزمایش یکی را انتخاب می‌کردم.

در ضمن در بخش افزودن نویز برای پیدا کردن یک نمونه خوب از چند نفر پرس‌وجو کردم و کتابخانه skimg و نحوه تولید نویز با آن را به من معرفی کردند. رسم ۳ بعدی هم اندکی وقت و سرچ لازم داشت تا متوجه نحوه استفاده از آن شوم.

با تشکر از حوصله و توجه شما