به نام خداوند بخشنده‌ی مهربان

گزارش پروژه‌ی دوم هوش مصنوعی

شبکه‌ی عصبی پرسپترون چندلایه

نیم‌سال اول ۱۴۰۰-۱۴۰۱

سید پارسا نشایی - ۹۸۱۰۶۱۳۴

دانشگاه صنعتی شریف
دانشکده‌ی مهندسی کامپیوتر

مقدمه

در این پروژه که ۶ بخش دارد، برنامه‌هایی به زبان Python نوشته شده‌اند تا خواسته‌های پروژه به کمک طراحی شبکه‌ی عصبی و یادگیری و تنظیم پارامترهای آن، تحقق یابند. در این پروژه، ابتدا ورودی‌ها و خروجی‌های مد نظر آن بخش در هر بخش آماده شده و سپس کد به یادگیری شبکه عصبی پرداخته و در نهایت، نتایج تحلیل و بررسی شده و چالش‌ها بیان شده‌اند.

کتاب‌خانه‌های استفاده شده، numpy برای عملیات مقدماتی برداری ریاضی (به اسم np در کد وارد شده) و نیز matplotlib.pyplot برای رسم نمودار (به اسم pplot در کد وارد شده) هستند. افزون بر این دو کتاب‌خانه، نیاز به کتاب‌خانه‌ای برای تسهیل محاسبات شبکه‌های عصبی و فرآیند طراحی و بررسی آن‌ها نیز وجود دارد. دو کتاب‌خانه‌ی مشهور در این خصوص، کتاب‌خانه‌ی PyTorch و نیز کتاب‌خانه‌ی TensorFlow (معمولا استفاده شده به همراه Keras) است. سعی شده که در اکثر بخش‌ها دو کد متفاوت برای حل آن بخش ارائه شود که یکی از PyTorch و

دیگری از Keras و TensorFlow استفاده کرده باشد؛ در برخی بخش‌ها نیز تنها از یکی از این دو استفاده شده است (گزارش حول TensorFlow + Keras نوشته شده است). نکته‌ای که باید به آن توجه شود این است که در نهایت تفاوت معناداری در استفاده از این دو کتاب‌خانه – به جز خود سورس کد برنامه – وجود ندارد، زیرا هر دو کتاب‌خانه در ابعاد مورد بررسی در این پروژه، کار یکسانی را انجام می‌دهند و دلیل نوشته شدن برخی بخش‌های پروژه با هر دو کتاب‌خانه، صرفا تمرین آموزشی برای خودم بوده است.

بخش‌های اول و دوم

فایل این بخش‌ها، 1_TF_EstimatingFunction است.
در این بخش‌ها، تعدادی تابع تعریف شده و شبکه‌ای طراحی شده تا بتواند آن‌ها را یاد بگیرد. در بخش اول، هدف تخمین تابع دقیق اصلی است و در بخش دوم، هدف تخمین تابع با نویز است (طی ایمیلی که به استاد محترم ارسال کردم، مقرر شد که هر دو مجموعه‌ی train و test نویز داشته باشند).

انواع نویز طراحی شده به شرح زیرند:

- نویز کم: با مقدار از صفر تا $\frac{1}{6}$
- نویز متوسط: با مقدار از صفر تا ۱
- نویز بالا: با مقدار از صفر تا ۵
- نویز بسیار بالا: با مقدار از صفر تا ۵۰

این نویزها توسط تابع np_noise ساخته شده و به صورت یک آرایه‌ی numpy برگردانده می‌شوند که عضو i ام آن، مقدار نویز در نقطه‌ی i ام است.

بخش اصلی کد، تابع run است. ورودی‌های این تابع به شرح زیرند:

- پارامتر layers_generator: این پارامتر خود یک تابع است که برای یافتن ساختار شبکه عصبی مورد استفاده صدا زده می‌شود. خروجی این تابع، یک مدل شبکه عصبی است.

- پارامتر np_func: این پارامتر، تابعی است که قصد تخمین آن را داریم.
- پارامتر train_bound: این پارامتر، کران داده‌های train را مشخص می‌کند، به طوری که train روی نقاط بازه‌ی $[-train\_bound, train\_bound]$ انجام می‌شود.
- پارامتر number_of_train_points: این پارامتر، تعداد نقاطی که باید به عنوان داده‌های train از بازه‌ی $[-train\_bound, train\_bound]$ انتخاب شوند را مشخص می‌کند.
- پارامتر test_bound: این پارامتر، کران داده‌های test را مشخص می‌کند، به طوری که test روی نقاط بازه‌ی $[-test\_bound, test\_bound]$ انجام می‌شود.
- پارامتر number_of_test_points: این پارامتر، تعداد نقاطی که باید به عنوان داده‌های test از بازه‌ی $[-test\_bound, test\_bound]$ انتخاب شوند را مشخص می‌کند.
- پارامتر number_of_iterations: این پارامتر، تعداد دفعات اجرای حلقه‌ی یادگیری مدل را مشخص می‌کند.
- پارامتر batch_size: این پارامتر، مشخص می‌کند که داده‌ها در دسته‌های چندتایی هنگام یادگیری به مدل داده شوند.
- پارامتر noise: این پارامتر، مقدار نویز را از بین پنج مقدار none و low و medium و high و very_high مشخص می‌کند. وجود این پارامتر و مقادیر آن باعث می‌شود که بتوان یک کد برای هر دو بخش اول و دوم پروژه ارائه داد.

درون این تابع، بازه‌های یادگیری و تست ساخته شده، تابع داده شده روی دامنه‌های ساخته شده اعمال شده (تا مقدار واقعی تابع در آن نقاط را داشته باشیم که بتوانیم train و test انجام دهیم) و نویز نیز در صورت لزوم و به مقدار لازم به این مقادیر اضافه شده است. سپس، بر اساس خروجی layers_generator، یک مدل پشت سر هم (Sequential) ساخته شده و کد با تابع loss خطای میانگین مربعات (mse) اقدام به ساختن مدل می‌کند. در کد تعیین شده تا خطای میانگین قدرمطلق (mae) برگردانده شود که دید خوبی از میزان خطای الگوریتم به ما می‌دهد. optimizer نیز adam انتخاب شده که نتیجه‌ی قابل قبولی را در اختیار قرار می‌دهد. سپس کد اقدام به fit کردن مدل با پارامترهای داده شده در ورودی تابع می‌کند و سپس به ارزیابی آن روی داده‌های تست پرداخته و mae آن را چاپ می‌کند. در نهایت، در یک نمودار، تابع اصلی با رنگ سبز و تابع خروجی شبکه عصبی با رنگ قرمز رسم می‌شوند.

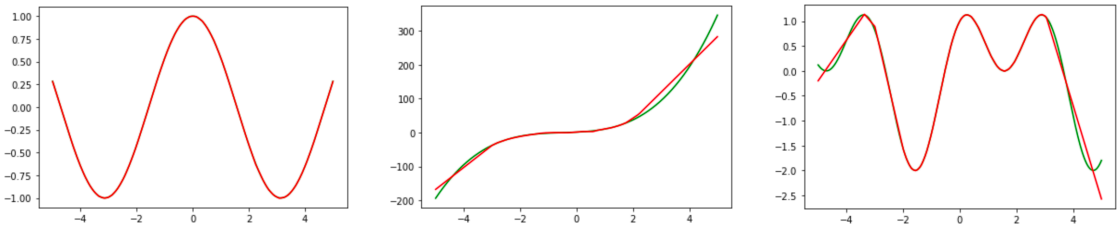
سه تابع ریاضی اصلی برای ارزیابی این بخش در نظر گرفته شده است:

- تابع $y(x) = \cos(x)$
- تابع $y(x) = 2x^3 + 3x^2 + 4x + 1$
- تابع $y(x) = \cos(2x) + \sin(x)$

مشاهده، تجزیه و تحلیل نتایج

در تمام مثال‌های زیر، سه تابع فوق به ترتیب از چپ به راست رسم شده‌اند.

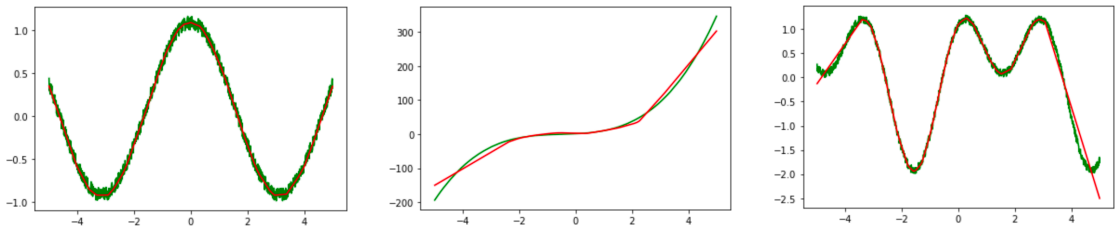
- رسم هر سه تابع با شبکه‌ای که لایه‌های میانی آن شامل دو لایه‌ی ۱۰۰ تایی و ۵۰ تایی با activiation برابر ReLU است (تابع ReLU برای اعداد منفی برابر صفر و برای سایر اعداد برابر خود عدد است) و نیز بازه‌ی بررسی هم train و هم test برابر $[-5,5]$ و شامل ۱۰۰۰ عدد، با batch_size و تعداد دفعات تکرار حلقه‌ی یادگیری هر دو برابر ۱۰۰۰:



MAE: 0.00104 7.931916 0.060087

به طور کلی تطابق مناسبی حاصل شده است، اما به خصوص در دو شکل سمت راست، هرچه از مرکز دورتر می‌شویم تطابق کم‌تر می‌شود.

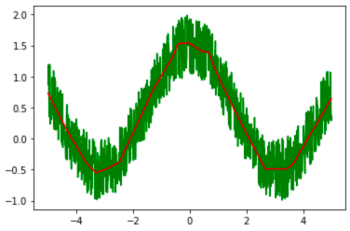
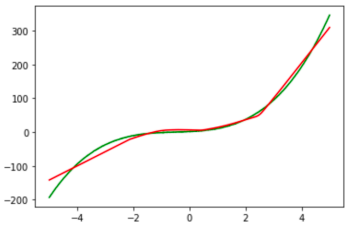
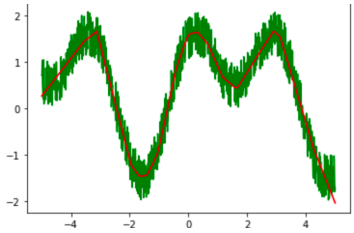
- رسم مورد فوق، اما به همراه نویز کم:



MAE: 0.042860 8.280737 0.082752

هم‌چنان به طور کلی تطابق مناسب است. در شکل وسط نویز به شکل قابل توجهی دیده نمی‌شود، زیرا نویز داده شده در مقایسه با برد تابع ناچیز است.

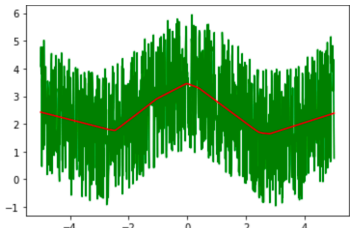
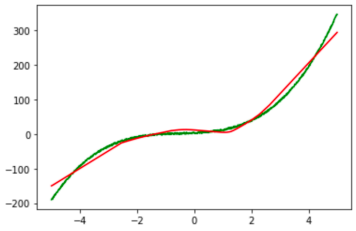
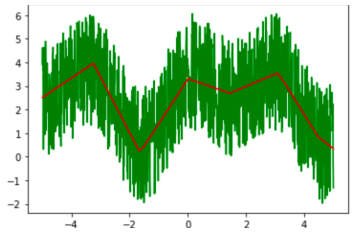
- رسم مورد فوق، اما به همراه نویز متوسط:

MAE: 0.246740 9.614349 0.256442

در این نمونه‌ها شاهد تکه تکه شدن تابع تخمین زده شده در شکل راست و چپ هستیم که ناشی از افزایش نویز است. در شکل وسط هم‌چنان نویز به شکل قابل توجهی دیده نمی‌شود، زیرا نویز داده شده در مقایسه با برد تابع ناچیز است.

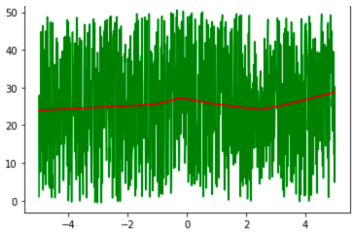
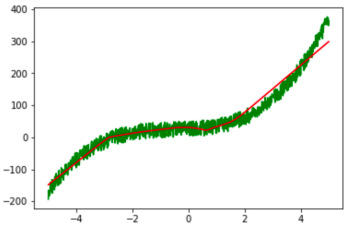
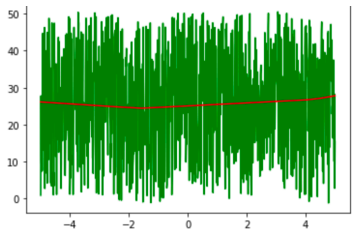
- رسم مورد فوق، اما به همراه نویز بالا:

MAE: 1.265571 10.693794 1.262824

در شکل راست و چپ عملا تابع اصلی از دست رفته، اما هم‌چنان نمودار داده شده با سیر کلی داده‌ها هماهنگ است و گونه‌ای از فرم سینوسی را دارد. در شکل وسط با زوم روی عکس مقداری نویز روی خط سبز قابل تشخیص بوده که البته هم‌چنان در مقایسه با برد تابع ناچیز است، اما شاهد افزایش تدریجی MAE هستیم.

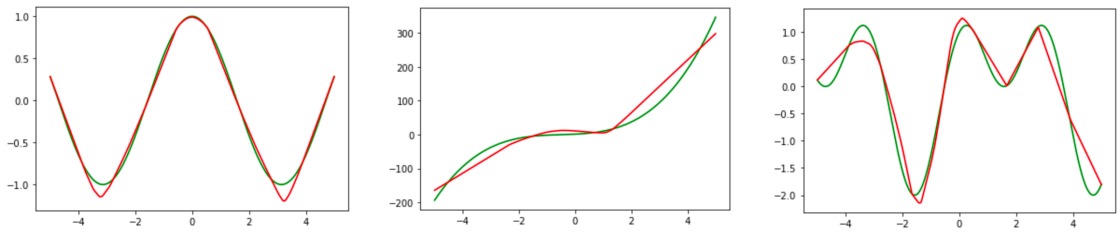
- رسم مورد فوق، اما به همراه نویز بسیار بالا (که مخصوص مشاهده نویز در تابع وسط طراحی شده):

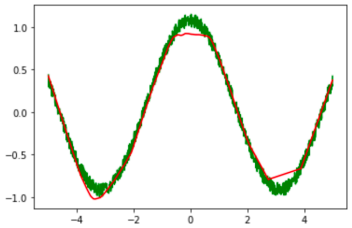
MAE: 12.188193 17.718154 12.219341

در شکل راست و چپ کاملا اشتباه تخمین صورت گرفته و تابع تخمین شده بیش‌تر به یک خط نزدیک است که البته با توجه به مقدار بسیار بالای نویز، منطقی است. در شکل وسط نیز نویز سبب شده که تابع کمی نادقیق‌تر تخمین زده شود.

- رسم مشابه مورد اول، اما با تعداد نقاط train بسیار کم‌تر:

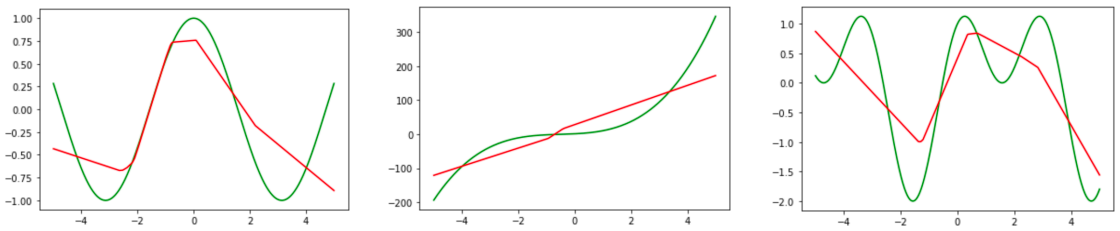


همان گونه که مشخص است، تعداد نقاط آموزشی کم‌تر سبب شده که دقت تخمین تابع به مقدار قابل توجهی کم‌تر شود و به عنوان مثال برای تابع سمت چپ که در مورد اول تطابق نزدیک به کامل داشت، در گوشه‌ها تطابق کم‌تری را شاهد هستیم. در نمودارهای دیگر نیز مشاهده می‌شود که نمودارهای تخمین زده شده بیش‌تر به شکل یک خط شکسته هستند تا یک منحنی که دلیل آن تعداد کم‌تر نقاط در دسترس برای به دست آوردن مقدار تابع هنگام فرآیند یادگیری بوده است.
طبیعتا اگر نویز به شرایط فوق اضافه شود، شرایط بدتر هم می‌شود؛ به عنوان مثال، در نویز کم شکل‌های زیر را شاهد هستیم:



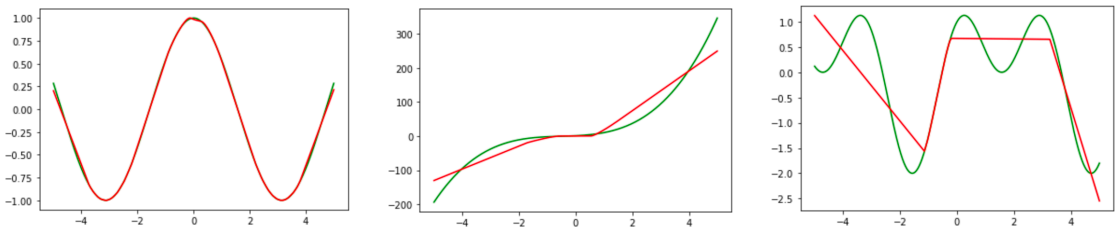
باقی شکل‌ها نیز در ژوپیتر نوت‌بوک موجود هستند.

- رسم مشابه مورد اول، اما با یک لایه کم‌تر و نیز تعداد نورون کم‌تر برای همان لایه (کل شبکه، یک لایه‌ی میانی به اندازه‌ی ۲۰ دارد):



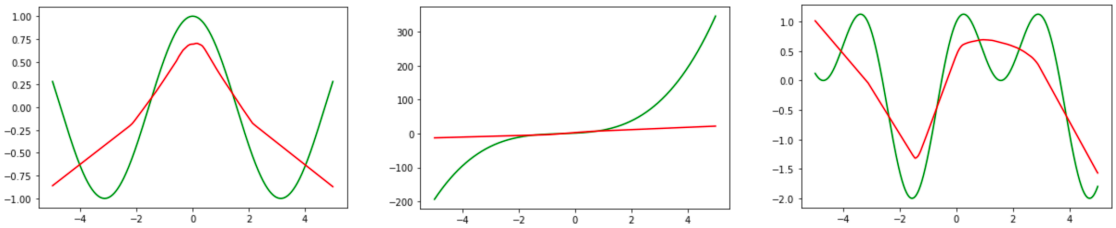
همان‌گونه که مشاهده می‌شود، کم کردن تعداد نورون‌ها و لایه‌ها، اثر منفی قابل توجهی بر میزان تطابق نمودار خروجی شبکه با نمودار واقعی می‌گذارد و در شرایط نویزی بودن نیز به شکل مشابهی بدتر می‌شود.

- رسم مشابه مورد قبل، اما با اندازه‌ی ۵۰۰ به جای ۲۰ برای لایه‌ی میانی:



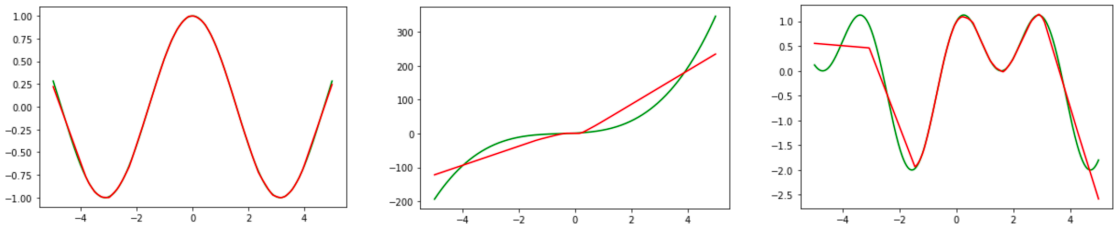
همان‌گونه که مشاهده می‌شود، زیاد کردن تعداد نورون‌ها سبب شده که برخی توابع مانند شکل چپ به خوبی تخمین زده شوند، اما هم‌چنان چون تنها یک لایه‌ی میانی وجود دارد، شبکه از تخمین زدن توابع پیچیده‌تر (مانند تابع سمت راست) ناتوان مانده است. از این شکل‌ها، نتیجه می‌گیریم که افزودن تعداد نورون‌های تک لایه‌ی میانی – حتی اگر ۲۵ برابرشان کنیم – هم‌چنان به شبکه توانایی تخمین توابع پیچیده را نمی‌دهند و بهتر است تعداد لایه‌ها را – ولو با تعداد کم‌تر نورون به ازای هر لایه به نسبت مثال ۲۵ برابر شدن – افزایش دهیم. با افزودن نویز نیز میزان عدم تطابق افزایش می‌یابد.

- رسم مشابه مورد اول، اما با ۱۰۰ بار اجرای حلقه‌ی یادگیری به جای ۱۰۰۰ بار:



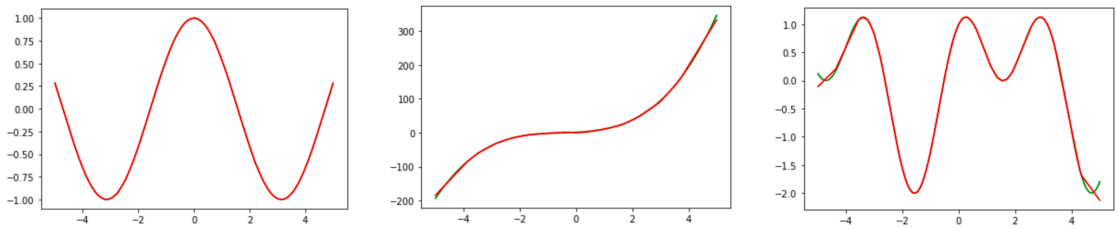
همان‌گونه که مشاهده می‌شود، کم کردن تعداد دفعات یادگیری مطابق انتظار باعث شده که لزوما الگوریتم به جواب بهینه نرسد و در نتیجه شامل خطا در تطابق باشد. با افزودن نویز نیز میزان عدم تطابق افزایش می‌یابد.

- رسم مشابه مورد قبل، اما با ۵۰۰ بار اجرای حلقه‌ی یادگیری به جای ۱۰۰ بار:



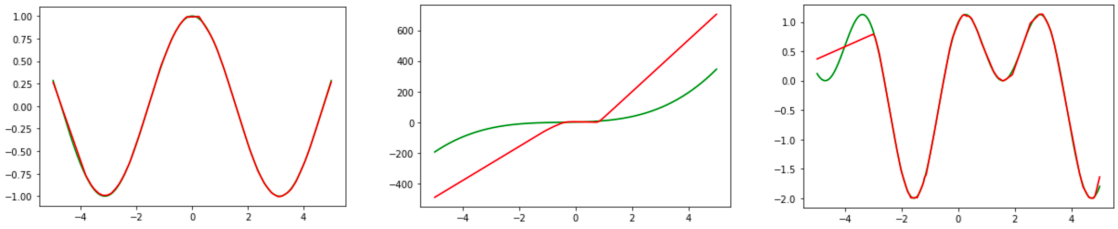
همان‌گونه که مشاهده می‌شود، این بار با افزایش تعداد دفعات یادگیری، تطابق بهتری نسبت به مورد قبل حاصل شده، اما هنوز به اندازه‌ی مورد اول که ۱۰۰۰ بار اجرا می‌شد نیست. با افزودن نویز نیز میزان عدم تطابق افزایش می‌یابد.

- رسم مشابه مورد قبل، اما با ۲۰۰۰ بار اجرای حلقه‌ی یادگیری به جای ۵۰۰ بار:



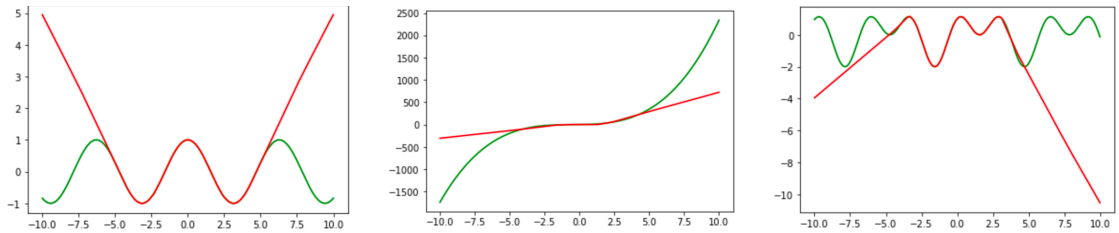
همان‌گونه که مشاهده می‌شود، این بار با افزایش تعداد دفعات یادگیری نسبت به حتی حالت اول، تطابق بهتری نسبت به آن حاصل شده که به خصوص در مقایسه‌ی نمودار وسط این مورد با مورد اول، واضح است؛ البته، با افزودن نویز، میزان عدم تطابق افزایش می‌یابد.

- رسم مشابه مورد اول، اما با train_bound برابر ۱۰ که عملا چگالی تعداد نقاط train در بازه‌ای که در نهایت test روی آن انجام می‌شود را نصف می‌کند:



همان‌گونه که مشاهده می‌شود، این کم شدن چگالی نقاط train، تاثیری منفی روی یادگیری – به خصوص در توابع پیچیده‌تر و چند جمله‌ای به نسبت توابع سینوسی – گذاشته است. با افزودن نویز نیز میزان عدم تطابق افزایش می‌یابد.

- رسم مشابه مورد اول، اما با test_bound برابر ۱۰:



همان‌گونه که مشاهده می‌شود، یادگیری فقط روی بازه‌ی منفی پنج تا پنج انجام می‌شود و روی این بازه دو تابع تخمینی و اصلی تطابق خوبی دارند، اما خارج از این بازه، تابع تخمینی به شکل خطی گسترش می‌یابد (زیرا شبکه RNN نیست و نمی‌تواند الگوهای تکرارشونده را یاد گرفته و به خاطر بسپارد) و در نتیجه در بیرون این بازه تطابق نخواهیم داشت. طبیعتا، مانند موارد قبل، با افزودن نویز نیز میزان عدم تطابق افزایش می‌یابد. به عبارت دیگر، در اصل، بی‌شمار تابع ممکن وجود دارد که در بازه‌ی train تطابق داشته باشند، اما در بازه‌ی test لزوما این طور نباشد.

در نهایت، از این آزمایش‌ها نتیجه می‌گیریم که با افزایش تعداد نقاط ورودی، کم بودن نویز، کم بودن پیچیدگی تابع، تعداد لایه و نورون بیش‌تر، چرخه‌های بیش‌تر شبکه برای تکمیل یادگیری و نیز تطابق بازه‌ی train و test، شبکه تخمین بهتری از تابع اصلی خواهد داشت.

بخش سوم

فایل این بخش، 3_TF_EstimatingFunction است.
در این بخش، هم‌چنان به دنبال تخمین زدن تابع هستیم، ولی این بار ورودی تابع دو بعدی است. دو تابع زیر تست شده‌اند:

- $f(x, y) = \sin(x) + \cos(y)$
- $f(x, y) = x^2 + y$

بخش اصلی کد، تابع run است که مشابه تابع run بخش اول و دوم است، با این تفاوت اصلی که آرایه‌های دامنه به شکل آرایه‌ای از آرایه‌ها طراحی شده‌اند تا ساختار دو بعدی داشته باشند. در نهایت نیز نمودار loss نسبت به تعداد iteration گذشته شده، رسم شده است که نزولی بودن آن، مطلوب است و مقدار mae نیز چاپ می‌شود. باقی کد، مشابه بخش‌های قبلی است.

در باقی کد، بعد از هر اجرای تابع run، مقدار predict شده‌ی شبکه‌ی به دست آمده برای چند نقطه‌ی نمونه و نیز مقدار mse چاپ می‌شود.

مشاهده، تجزیه و تحلیل نتایج

- تابع اول با شبکه‌ای که لایه‌های میانی آن شامل دو لایه‌ی ۲۵ تایی و ۲۵ تایی با activiation برابر ReLU است (تابع ReLU برای اعداد منفی برابر صفر و برای سایر اعداد برابر خود عدد است) و نیز بازه‌ی بررسی هم train و هم test برابر $[-1,1]$ و شامل ۵۰۰ عدد در هر بعد، با batch_size و تعداد دفعات تکرار حلقه‌ی یادگیری به ترتیب

برابر ۵۱۲ و ۱۰، به MSE برابر 2.94205201498698 44e-05 می‌رسد که با توجه به خروجی‌های داده شده در چند نقطه‌ی نمونه، مطلوب است.

- مانند مورد اول، اما با شبکه‌ای که لایه‌های میانی آن شامل یک لایه‌ی ۱۰ تایی با activiation برابر ReLU است که به MSE برابر 0.00023741282348055393 می‌رسد. به دلیل کم شدن تعداد نورون‌ها و لایه‌ها، MSE بیش‌تر از مورد اول شده است.
- مانند مورد قبل، اما تنها با ۲ تکرار حلقه‌ی یادگیری که به MSE برابر 0.0022514972370117903 می‌رسد. به دلیل کم شدن تعداد تکرار حلقه‌ی یادگیری، MSE بیش‌تر از مورد قبل شده است.
- مانند مورد قبل، اما با بازه‌ی train کوچک‌تر که به MSE برابر 0.12359391897916794 می‌رسد. به دلیل عدم تطابق بازه‌ی train و test، MSE بیش‌تر از مورد قبل شده است.
- مانند مورد دو تا قبلی، اما با تعداد نقاط train کم‌تر که به MSE برابر 1.0508798360824585 می‌رسد. به دلیل تعداد کم‌تر نقاط train که برای یادگیری وزن‌های مدل استفاده می‌شوند، MSE بیش‌تر از مورد دو تا قبلی شده است.

- تابع دوم با شبکه‌ای که لایه‌های میانی آن شامل دو لایه‌ی ۲۵ تایی و ۲۵ تایی با activiation برابر ReLU است (تابع ReLU برای اعداد منفی برابر صفر و برای سایر اعداد برابر خود عدد است) و نیز بازه‌ی بررسی هم train و هم test برابر $[-1,1]$ و شامل ۵۰۰ عدد در هر بعد، با batch_size و تعداد دفعات تکرار حلقه‌ی یادگیری به ترتیب برابر ۵۱۲ و ۱۰، به MSE برابر 3.3577878639334 81e-05 می‌رسد که با توجه به خروجی‌های داده شده در چند نقطه‌ی نمونه، مطلوب است. این مقدار کمی از MSE متناظرش در تابع اول بیش‌تر است که مشابه روندی بود که در دو بخش اول نیز دیدیم (توابع مثلثاتی، fit شدن نسبتا دقیق‌تری نسبت به توابع چندجمله‌ای داشتند)
- مانند مورد اول تابع دوم، اما با شبکه‌ای که لایه‌های میانی آن شامل یک لایه‌ی ۱۰ تایی با activiation برابر ReLU است که به MSE برابر 0.0005007627769373357 می‌رسد. به دلیل کم شدن تعداد نورون‌ها و لایه‌ها، MSE بیش‌تر از مورد اول تابع دوم شده است.
- مانند مورد قبل، اما تنها با ۲ تکرار حلقه‌ی یادگیری که به MSE برابر 0.005744677037000656 می‌رسد. به دلیل کم شدن تعداد تکرار حلقه‌ی یادگیری، MSE بیش‌تر از مورد قبل شده است.

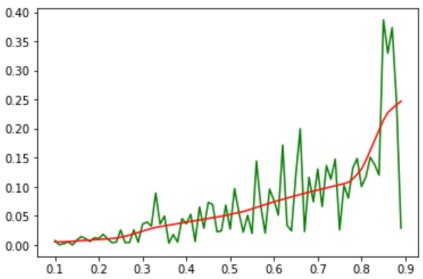
- مانند مورد قبل، اما با بازه‌ی train کوچک‌تر که به MSE برابر 0.0299963727593421 94 می‌رسد. به دلیل عدم تطابق بازه‌ی train و test، MSE بیش‌تر از مورد قبل شده است.
- مانند مورد دو تا قبلی، اما با تعداد نقاط train کم‌تر که به MSE برابر 1.087976098060608 می‌رسد. به دلیل تعداد کم‌تر نقاط train که برای یادگیری وزن‌های مدل استفاده می‌شوند، MSE بیش‌تر از مورد دو تا قبلی شده است.

بخش چهارم

فایل این بخش، 4_TF_EstimatingFunction است. روند کلی کار مشابه بخش‌های قبل است. تابع عجیبی که داده شده، همان تابع عجیبی است که در پروژه‌ی ژنتیک نیز برای تخمین زدن داده شده بود (البته این تابع واقعا خشم بسیار زیادی دارد (!) و غیر نرمال است، اما با این حال نهایت توان شبکه‌ی عصبی در تشخیص توابع بسیار عجیب و پر پرش را می‌سنجد). هرچه میزان پرش بیش‌تر باشد (خشم بیش‌تر)، امکان تطبیق به وضوح کم‌تر می‌شود و به شبکه‌ی بزرگ‌تری برای حدی از تخمین قابل قبول نیاز است. به وضوح این آزمایش‌ها نسبت به بخش اول، نتایج به شدت نامناسب‌تری دارند. در ادامه چند تابع با خشم کم‌تر نیز داده شده است.

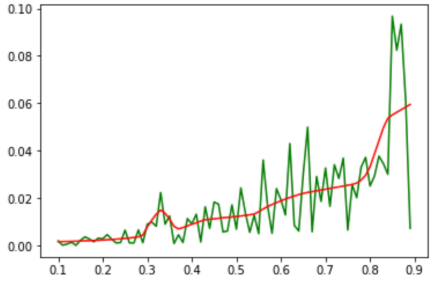
مشاهده، تجزیه و تحلیل نتایج

- مدل اول شامل لایه‌های میانی ۱۰۰ و ۵۰ با تابع فعال‌سازی ReLU:



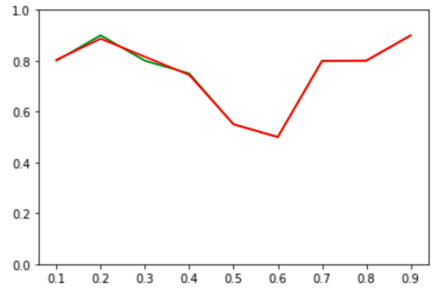
روند و trend کلی تابع که افزایشی است توسط شبکه عصبی پیدا شده و حتی در اعداد ۰.۸ به بالاتر که پرش قابل توجه داریم، شیب نمودار قرمز یافته شده افزایش یافته است، اما هم‌چنان شبکه موفق به کشف جزئیات دقیق‌تر افزایش‌ها نشده است (گویی این پرش‌های ناگهانی نویزهایی هستند که شبکه آن‌ها را ignore کرده است). مثلا بین ۰.۳ و ۰.۴ یک قله وجود دارد، اما شبکه آن را پیدا نکرده است.

- مدل اول شامل لایه‌های میانی ۱۰۰۰ و ۱۰۰۰ و ۱۰۰۰ و ۱۰۰۰ و ۵۰۰۰ و ۸۰۰۰ و ۳۰۰۰ با تابع فعال‌سازی ReLU:



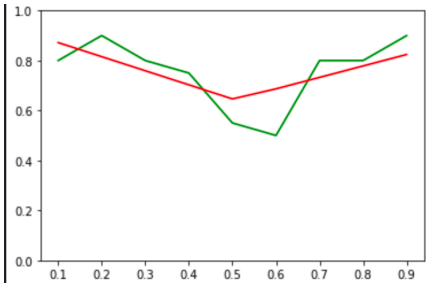
این مدل از مدل قبلی بزرگ‌تر است و در نتیجه نتیجه‌ی بهتری نسبت به قبلی داده است، اما هم‌چنان به دلیل ماهیت عجیب تابع، بهترین نتیجه‌ی ممکن نیست. هم‌چنان پرش‌های نهایی ignore شده‌اند، اما در سمت راست شیب نزدیک‌تری به شیب واقعی نمودار سبز به دست آمده و قله‌ی میان ۰.۳ و ۰.۴ که واقعا روند افزایشی نرمال بوده و صرفا یک پرش نبوده نیز کشف شده است.

- مدل سوم شامل لایه‌های میانی ۲۵۶ و ۵۱۲ و ۱۰۲۴ با تابع فعال‌سازی ReLU و برای تابع کم‌خشم‌تر:



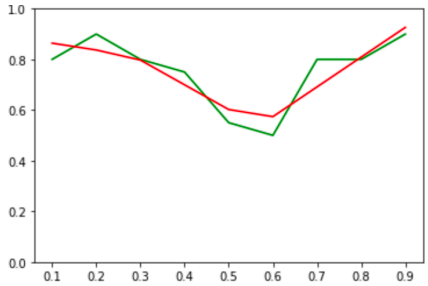
این تابع پرش‌های کم‌تری دارد و در نتیجه شبکه توانسته تقریبا به خوبی آن را تخمین بزند.

- مدل چهارم شامل لایه‌ی میانی ۱۰ با تابع فعال‌سازی ReLU و برای تابع کم‌خشم‌تر:



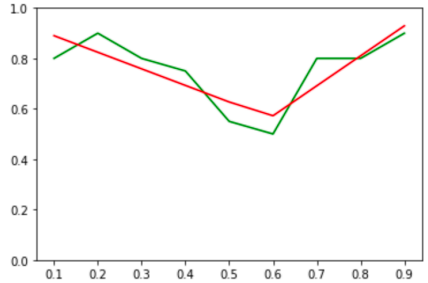
به دلیل کم شدن تعداد نورون‌ها و لایه‌ها، میزان دقت تخمین و تطابق نیز کم‌تر شده است.

- مدل پنجم شامل لایه‌ی میانی ۳۲ با تابع فعال‌سازی ReLU و برای تابع کم‌خشم‌تر:



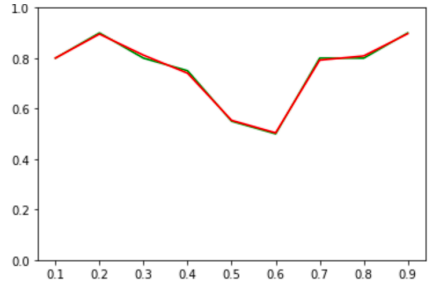
به دلیل افزایش تعداد نورون‌ها، دقت تطابق اندکی نسبت به قبلی بیش‌تر شده است.

- مدل ششم شامل لایه‌ی میانی ۱۰۰ با تابع فعال‌سازی ReLU و برای تابع کم‌خشم‌تر:



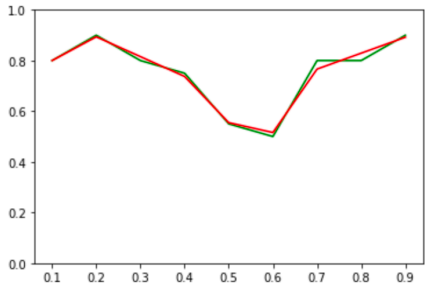
به دلیل افزایش تعداد نورون‌ها، دقت تطابق اندکی نسبت به قبلی بیش‌تر شده است ولی چندان محسوس نیست.

- مدل هفتم شامل لایه‌ی میانی ۱۰۰۰ با تابع فعال‌سازی ReLU و برای تابع کم‌خشم‌تر:



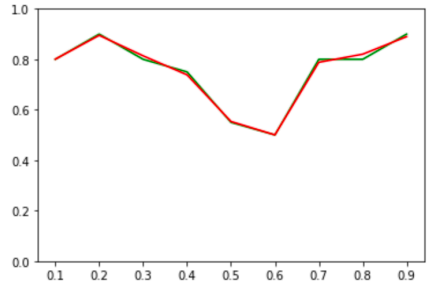
به دلیل افزایش قابل توجه تعداد نورون‌ها، تطابق تقریبا کامل است.

- مدل هشتم شامل لایه‌ی میانی ۵۰۰ با تابع فعال‌سازی ReLU و برای تابع کم‌خشم‌تر:



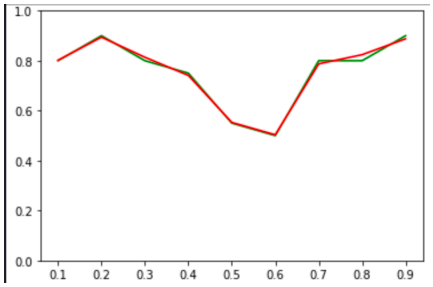
۵۰۰ کم‌تر از ۱۰۰۰ است برای همین کمی تطابق کم‌تر شده، ولی هم‌چنان تطابق نسبتا مناسب است.
همین روند جست‌وجوی باینری را برای یافتن کم‌ترین تعدادی که تطابق تقریبا کامل را ارائه دهد، ادامه می‌دهیم.

- مدل نهم شامل لایه‌ی میانی ۷۵۰ با تابع فعال‌سازی ReLU و برای تابع کم‌خشم‌تر:



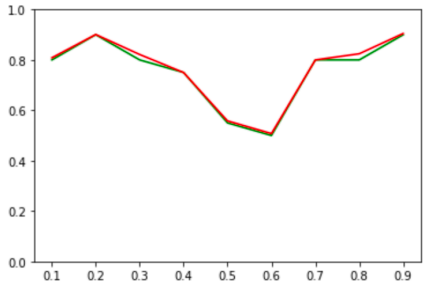
تطابق کمی از قبلی بیش‌تر شده است.

- مدل دهم شامل لایه‌ی میانی ۸۵۰ با تابع فعال‌سازی ReLU و برای تابع کم‌خشم‌تر:



تطابق کمی از قبلی بیش‌تر شده است اما چندان محسوس نیست.

- مدل یازدهم شامل لایه‌ی میانی ۹۵۰ با تابع فعال‌سازی ReLU و برای تابع کم‌خشم‌تر:



تطابق کمی از قبلی بیش‌تر شده است اما چندان محسوس نیست.

بخش پنجم

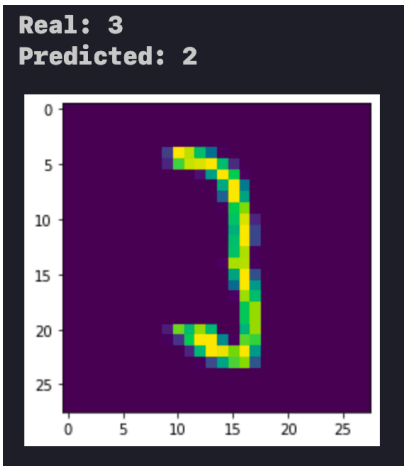
فایل این بخش، 5-TF است.

برای این بخش و بخش بعدی، از دیتاست MNIST استفاده شده که شامل عکس‌های سیاه–سفید ۲۸ در ۲۸ پیکسل از ارقام انگلیسی است. در کد این بخش، ابتدا به کمک ابزار داخلی keras این دیتاست بارگذاری شده و سپس reshape های لازم و نیز تبدیل خروجی به نوع categorical انجام پذیرفته است. در تابع run که ورودی batch_size نیز دریافت می‌کند و نیز یک ورودی validation_split دارد که مشخص می‌کند چند درصد داده‌ها طی فرایند یادگیری برای validation کنار گذاشته شوند، مدل با تابع loss برابر categorical_crossentropy که برای دسته‌بندی مناسب است کامپایل شده، سپس به داده fit شده و در نهایت روی داده‌های تست که در خود MNIST جدا شده‌اند، آزمایش شده و مقدار loss چاپ می‌شود. در ادامه، این تابع با پارامترهای مختلف صدا زده شده است.

مشاهده، تجزیه و تحلیل نتایج

- مدل اول شامل لایه‌های ۷۸۴ (۲۸*۲۸)، ۲۰۰، ۲۵۰ و ۱۰ لایه که لایه‌ی آخر با تابع فعال‌سازی softmax (برای مشخص کردن احتمال بودن در هر دسته‌بندی) و لایه‌های دیگر با تابع فعال‌سازی ReLU هستند، مقدار Loss برابر 0.08899032324552536 را می‌دهد.

این مدل، کارایی مناسبی دارد؛ به عنوان مثال، یکی از نمونه‌های mismatch شده (که به تصادف توسط کد انتخاب شده) در زیر قابل مشاهده است که نمونه‌ی واضحی نبوده و تشخیص آن برای انسان هم کمی سخت است:



- مدل دوم که مشابه مدل قبلی است، اما به جای دو لایه‌ی ۲۰۰ و ۲۵۰ یک لایه ۲۰ قرار گرفته و تنها ۲ بار حلقه در آن اجرا می‌شود، مقدار Loss برابر 0.12769579887390137 را می‌دهد که به دلیل کم شدن تعداد لایه‌ها و نورون‌ها و نیز تعداد دفعات اجرا، افزایش loss منطقی است.
- مدل سوم که مشابه مدل قبلی است، اما ۵ بار حلقه در آن اجرا می‌شود، مقدار Loss برابر 0.07428432255983353 را می‌دهد که به دلیل افزایش تعداد دفعات اجرای حلقه، loss کم‌تری از بخش قبل دارد. نکته‌ی قابل توجه، کم‌تر بودن loss این مدل نسبت به مدل اول که شبکه‌ی بزرگ‌تری بود است که می‌تواند نشان از مقداری overfit در مدل اول باشد، به این صورت که اندازه‌ی بزرگ‌تر از حد لازم مدل اول، اجازه‌ی حفظ کردن داده‌های train را به آن می‌داد و در نتیجه مدل در داده‌ی test لزوما به همان خوبی عمل نمی‌کرد.

بخش ششم

فایل این بخش، 6-TF است.

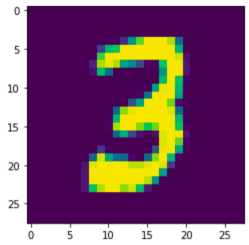
برای افزودن نویز به داده‌های MNIST، به پیکسل‌ها مقداری تصادفی از توزیع نرمال با میانگین صفر و انحراف معیار به ترتیب ۰.۲ برای نویز کم، ۰.۴ برای نویز متوسط و ۰.۶ برای نویز زیاد اضافه می‌شود (البته، به دلیل وجود np.clip تضمین می‌شود این مقدار بین صفر و یک باشد تا مقدار روشنایی هر پیکسل، معتبر بماند).

سپس تابع run مشابه بخش‌های قبلی نوشته شده که با توجه به پارامتر noise ورودی، نویز را به عکس‌ها اضافه کرده و در نهایت loss را چاپ می‌کند. در ادامه، در چند مرحله، ساختاری برای شبکه‌ی عصبی تعریف شده و تابع run صدا زده شده و سپس عکس اصلی، عکس نویزی و عکس بازیابی شده، نمایش داده می‌شوند.

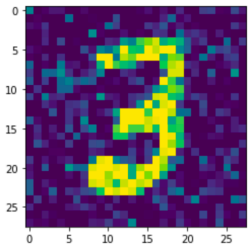
مشاهده، تجزیه و تحلیل نتایج

عددهای loss نوشته شده، در صورت عدم ذکر، مربوط به ارزیابی روی داده‌های test هستند. ضمنا عکس‌های آورده شده در گزارش، از داده‌های تست هستند، اما از داده‌های train نیز عکس در ژوپیتر نوت‌بوک آمده است.

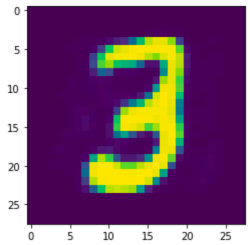
- مدل با لایه‌ی میانی ۱۰۰ با تابع فعال‌سازی ReLU و لایه‌ی نهایی با تابع فعال‌سازی sigmoid به اندازه‌ی ۷۸۴ (۲۸*۲۸) (زیرا این مدل یک عکس که ماتریسی ۲۸ در ۲۸ است را خروجی خواهد داد) و با نویز کم عکس‌ها که با ۱۵ بار اجرا به loss برابر 0.005665069445967674 می‌رسد.
  - یک نمونه عکس اصلی از داده، قبل از اعمال نویز با این مدل:



  - عکس پس از اعمال نویز:

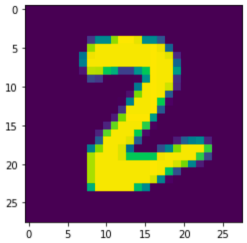


  - عکس denoise شده (بازیابی شده) توسط شبکه عصبی:

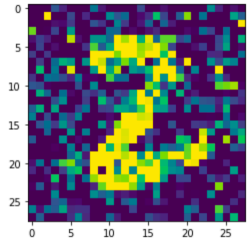


همان‌گونه که مشخص است، عکس به خوبی بازیابی شده است.

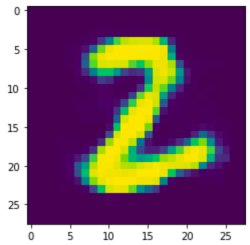
- مدل مشابه قبلی، ولی با نویز متوسط عکس‌ها که به loss برابر 0.012086220085620880 می‌رسد.
- یک نمونه عکس اصلی، قبل از اعمال نویز با این مدل:



- عکس پس از اعمال نویز:



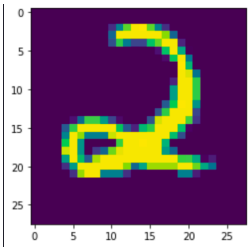
- عکس denoise شده (بازیابی شده) توسط شبکه عصبی:



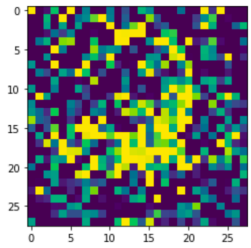
عکس باز هم به خوبی – البته با کیفیت کمی پایین‌تر – بازیابی شده است.

- مدل مشابه قبلی، ولی با نویز زیاد عکس‌ها که به loss برابر 0.020281182602047920 می‌رسد.

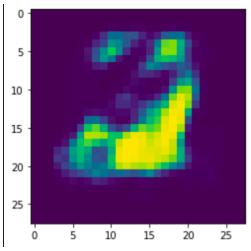
- یک نمونه عکس اصلی، قبل از اعمال نویز با این مدل:



- عکس پس از اعمال نویز:



- عکس denoise شده (بازیابی شده) توسط شبکه عصبی:



به دلیل نویز بسیار زیاد، این بار عکس به خوبی بازیابی نشده است.

در خصوص تفاوت loss در داده‌های train و test مشاهده می‌شود که در نویز کم، loss روی داده‌های train به میزان 0.0000553666614 بیش‌تر است (که البته این میزان بیش‌تر بودن، بسیار کوچک است و در عمل بی‌معناست و در نتیجه، یادگیری به خوبی روی داده‌های دیده نشده نیز generalize شده است)، در نویز متوسط، میزان loss داده‌های test به میزان 0.0001318529248 بیش‌تر است که هم‌چنان جزئی است، اما این بار loss در داده‌های آزمایشی بیش‌تر از آموزشی شده و مقدار اختلاف نیز نسبت به حالت کم‌نویز بیش‌تر بوده که منطقی است. در حالت نویز زیاد، میزان loss داده‌های تست به میزان 0.0004046801478 بیش‌تر است؛ در این حالت، اختلاف بیش‌تر از حالت نویز متوسط شده که نشان می‌دهد با افزایش نویز، امکان generalize شدن مدل کم‌تر می‌شود، اما هم‌چنان از سطح قابل قبولی برخوردار است.

چالش‌های اصلی

چالش مهمی که به آن برخوردم، نیاز به اجرای سریع‌تر کدها بود که بر روی لپ‌تاپ خودم که از GPU مجزا برخوردار نیست، دشوار بود، از این رو برای حل این چالش از Google Colab استفاده کردم که GPU در اختیارم قرار می‌داد. خود یادگیری کتاب‌خانه‌ها (مانند Keras) نیز چالش برانگیز بود که نیاز بود به آن‌ها مسلط باشم با بتوانم پروژه را به خوبی انجام دهم. یکی دیگر از چالش‌ها، در بخش توابع با ورودی بالاتر از یک‌بعد بود که در ابتدا هر نقطه‌ی ورودی را به شکل تاپل به شبکه ورودی می‌دادم و ارورهای عجیب می‌گرفتم تا بعد از آزمون و خطا و کمی جست‌وجو در اینترنت و Stack Overflow دریافتم که این ورودی‌ها باید به شکل آرایه‌ای از آرایه‌ها باشد.