دانشگاه تهران

پردیس دانشکده‌های فنی

دانشکده مهندسی برق و کامپیوتر

گزارش تمرین سری دوم درس شبکه عصبی

یلدا فروتن

۸۱۰۱۹۶۲۶۵

استاد درس

جناب آقای دکتر کلهر

پاییز ۹۸

۱. طراحی شبکه Multi-Layer Perceptron

۱,۱ دانش اولیه

۱,۱,۱ دلایل استفاده از تعداد لایه‌های بیشتر از دو در MLP

لایه اول همانند مادلاین hyperplane می‌سازد البته در اینجا hyperplane ها نرم هستند. لایه دوم با استفاده از hyperplane ها نرم لایه اول، convex hyperpolygon می‌سازد. بنابراین در یک MLP دولایه می‌توان از متصل کردن hyperpolygon ها، multi-hyperploygon یا non-convex hyperpolygon ساخت. حال افزودن لایه‌های بیشتر در زمینه گفته‌شده، تفکیک کلاس‌های convex یا non-convex، هیچ مزیتی ندارد بلکه ممکن است redundancy بیشتری به شبکه بدهد و احتمال رسیدن به جواب را افزایش می‌دهد. به طور کلی اگر ورودی‌ها correlation، disturbance و distortion نداشته باشند، دو لایه برای مساله رگرسیون و کلسیفیکیشن کافی است.

۲,۱,۱ تابع هزینه Cross entropy

آنتروپی یک معیاری از اشتباهات تصادفی است که درحین انتقال یک سیگنال به وجود می‌آید؛ بنابراین می‌تواند معیاری از بازدهی سیستم نیز باشد. به طورکلی برای هر مجموعه می‌توان آنتروپی یک توزیع را به صورت زیر محاسبه کرد اما در اینجا پارامترها به صورتی که در روند یادگیری یک شبکه استفاده می‌شوند، تعریف شده‌اند:

$$H(q) = -\sum_{c=1}^{C} q(y_c) \cdot log(q(y_c))$$

در رابطه بالا، C تعداد کلاس‌های خروجی یک شبکه است و target شبکه $q(y_c)$ ها هستند. حال با استفاده از شبکه باید label هر داده از training set پیشبینی شود. بدیهی است که بازدهی سیستم برای مساپل مدنظر در شبکه، نمی‌تواند ۱۰۰ درصد باشد و همواره مقداری خطا وجود دارد. در این حالت است که آنتروپی غیرصفر

می‌شود. به عنوان مثال آگه گفته شود تمام ورودی‌های شبکه نقاط سبزرنگ هستند، در این حالت آنتروپی صفر است و سیستم دارای بازدهی ۱۰۰ است. اما برای تعداد دسته‌های بیشتر آنتروپی قطعاً غیرصفر است. حال فرض شود سیستم برای کلاس‌های c مقادیر $p(y_c)$ را پیشبینی کرده است در این حالت می‌توان cross entropy را طبق معادله زیر محاسبه کرد:

$$H_p(q) = -\sum_{c=1}^{C} q(y_c) \cdot log(p(y_c))$$

از آنجایی که خطا صفر غیرواقعی است، مقدار $p(y)$ و $q(y)$ یکی نیستند، بنابراین طبق روابط بالا cross entropy معمولاً بزرگتر از entropy است. حال شبکه باید بهترین $p(y)$ را پیشبینی کند. درواقع یافتن بهترین پیشبینی کار کلسیفایر است. بدیهی است که بهترین پیشبینی به target نزدیک است؛ درنتیجه آنتروپی predict و target باید نزدیک به صفر باشد و البته هرگز صفر نخواهد شد. حال کلسیفایر برای هر N تعداد نمونه‌ای که در training set قرار دارد، cross entropy loss را محاسبه می‌کند. از آنجایی احتمال هر نمونه 1/N است، cross entropy به صورت زیر می‌شود:

$$q(y_i) = \frac{1}{N} \Rightarrow H_p(q) = -\frac{1}{N} \sum_{i=1}^{N} log(p(y_i))$$

برای نمونه اگر خروجی‌های مساله به صورت دو کلاس باشند، از تابع Binary Cross Entropy استفاده می‌شود و رابطه آن به صورت زیر است:

$$H_p(q) = -\frac{1}{N} \sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

به عنوان مثال اگر دو کلاس وجود داشته باشد و برچسب‌های صفر و یک را به آنها اختصاص داده شده باشد، تابع loss به صورت زیر محاسبه می‌گردد:

$$y = 0 \rightarrow H_p(q) = {}^{-1}/_{N}\left(\log\left(1 - p(y)\right)\right) \rightarrow H_p(q) = \begin{cases} 0 \\ \infty \end{cases}$$

$$y = 1 \rightarrow H_p(q) = {}^{-1}/_{N}\log\left(1 - p(y)\right) \rightarrow H_p(q) = \begin{cases} 0 \\ \infty \end{cases}$$

بنابراین اگر target مربوط به یک کلاس ۰ باشد و سیستم به درستی ۰ پیشبینی کرده باشد، در این صورت مقدار خطا صفر خواهد بود. اما اگر شبکه به اشتباه ۱ تشخیص دهد، مقدار خطا بینهایت می‌شود. همین شیوه برای سایر label ها نیز درنظر گرفته شده است.

۳،۱،۱ تأثیر batch size و learning rate در آموزش شبکه

- **تأثیر batch size**

یکی از hyperparameter های مهم برای تیون کردن وزن‌ها در شبکه‌های عصبی، batch size است. در هنگام آموزش شبکه، برای داده‌های training set، سه نوع دسته‌بندی وجود دارد:

- Batch Based
- Stochastic Based (Point Based)
- Stochastic Mini-Batch Based

در batch based، مجموع گرادیان مربوط به همه sample ها محاسبه می‌گردد و وزن‌ها آپدیت می‌شوند. در point based، برای همه sample ها، هربار یک نقطه تصادفی انتخاب شده و وزن‌ها آپدیت می‌شوند. در mini-batch دسته بزرگ sample ها به دسته‌های کوچک‌تر تقسیم می‌شود. درواقع مزیت batch based آنست که در کوتاه مدت، تابع loss کاهش می‌یابد و سرعت همگرایی بالایی دارد اما به نبود امکان پرش تصادفی، ممکن است در یک نقطه local optimum گیر کند. حال mini-batch مزیت هر دو دسته بالا را دارد. بنابراین اگر mini-batch به تعداد sample ها باشد، همان batch based خواهد بود و یک دسته بزرگ وجود دارد اما اگر یک باشد، stochastic بوده هر sample یک mini-batch خواهد بود.

به طورکلی طراحان به دنبال استفاده از mini-batch هستند تا از امکان موازی‌سازی GPU ها استفاده کنند و سرعت پردازش داده‌ها را افزایش دهند. هرچند تقسیم نمونه‌ها به تعداد دسته‌های زیاد منجر به کاهش قدرت تعمیم شبکه شده و به اصطلاح overfitting را افزایش می‌دهد. همچنین همانطور که در پاراگراف قبل اشاره شد، batch size بزرگ، احتمال رفتن به سمت global optimum را از بین می‌برد.

راهکارهایی برای انتخاب batch size وجود دارد:

- هنگامی که تعداد نمونه‌ها کمتر از ۲۰۰۰ است، بهتر است از روش batch based استفاده کرد و همه داده‌ها را با هم آپدیت کرد.

- هنگام استفاده از mini-batch، بهتر است batch size توانی از دو باشد زیرا حافظه‌ها (پردازنده‌های فیزیکی) توانی از دو هستند.

- استفاده از mini-batch ها مقدار loss (در واحد تکرار) را کاهش می‌دهد اما تغییرات loss را نویزی می‌کند.

- **تأثیر learning rate**

یکی از پارامترهای کنترل‌کننده وزن‌ها و بایاس، learning rate است. از دیگر کنترل‌کننده‌های ابتدایی و اساسی در طراحی شبکه می‌توان به تعداد نورون‌های هرلایه، تعداد لایه‌ها مخفی و activation function اشاره کرد. به طور کلی اگر learning rate کوچک باشد، سرعت یافتن وزن مناسب به روش گرادیان کم خواهد بود. درصورتی که برای مقادیر بزرگتر، سرعت رسیدن به هزینه کمتر افزایش می‌یابد اما ممکن از کاهش تابع هزینه دچار مشکل یا به اصطلاح واگرا گردد. بنابراین لازم است ابتدا نمودار تابع loss بر تعداد تکرار رسم گردد. اگر تابع هزینه افزایش یافت یا یک حالت پریودیک به خود گرفت، از Learning Rate کمتر استفاده کرد. زیرا احتمالاً مقدار آن بزرگ بوده و تابع هزینه واگرا شده است. بنابراین learning rate تأثیر به سزایی بر سرعت و میزان کاهش تابع هزینه می‌گذارد. درواقع گرادیان جهت حرکت در راستای کاهش تابع loss را می‌دهد اما این حرکت باید خیلی کوچک باشد تا شرط فاصله اقلیدسی نزدیک صفر را برآورده کند.

۴,۱,۱,۱ اهمیت استفاده از validation set

خوب بودن یک الگوریتم فقط به دلیل خوب fit شدن داده‌های training set نیست بلکه باید خطای ناشی از داده‌های validation نیز کم باشد. لازم به ذکر است خطای مربوط به داده‌های validation از هر مجموعه داده test دیگری کمتر است. یکی از موضوعات مهم درانتخاب داده‌ها مربوط به validation آن است که توزیع یکسانی با داده‌های test داشته باشند.

یکی از دلایل استفاده از مجموعه validation یا همان dev set آن است که حداقل داده‌ها در توقف دخالت داده می‌شوند. این درصورتی است داده‌های test نه مستقیم و نه غیرمستقیم در یادگیری شبکه تاثیری ندارند. گویا validation set به داده‌ها hint می‌دهد که بیشتر از این عمل آپدیت کردن وزن‌ها را ادامه نده زیرا تابع loss کمتر نمی‌شود و حتی در مواردی ممکن است افزایش یابد.

۲,۱ پیاده‌سازی یک شبکه MLP

هدف از انجام این تمرین نوشتن یک شبکه Multi-Layer Perceptron برای طبقه‌بندی اعداد ۰ تا ۹ به صورت دست‌نویس در دیتاست MNIST است. لازم به ذکر است این تمرین با استفاده از کتابخانه keras نوشته شده است. روند طراحی شبکه به صورت زیر است:

- **خواندن دیتاست MNIST**

در این قسمت ابتدا از کتابخانه keras، دیتاست MNIST خوانده شده است. همچنین از کتابخانه keras، np_utsils افزوده شده است که با آن می‌توان داده یک بردار کلاس را به ماتریس کلاس به صورت باینری تبدیل کرد تا بتوان از تابع هزینه cross entropy استفاده کرد. دیتاست MNIST دارای 70k عکس از اعداد ۰ تا ۹ به صورت دست‌نویس است. 60k از این تعداد مربوط به train_data و 10k باقی مربوط به داده‌های test_data هستند. بدیهی است که از داده‌های train به طور مستقیم برا آموزش شبکه استفاده می‌شود. اما داده‌های test تاثیری در بهبود شبکه ندارند. بنابراین از یک دسته داده دیگر تحت عنوان dev set یا همان validation set برای بهبود شبکه به کار گرفته می‌شود. از آنجایی که داده‌های validation به طور غیرمستقیم در آموزش شبکه تاثیرگذارند، باید داده‌های برچسب‌دار باشند. بنابراین در این تمرین ۲۰ درصد از داده‌های training جدا شده و به عنوان داده validation استفاده شده است. درنتیجه داده‌های مربوط به trianing، 48k و داده‌های validation، 12k خواهند بود.

داده‌های اختصاص داده شده به training و validation به صورت رندوم انتخاب شده‌اند.

Gradient descent به عنوان optimizer استفاده شده است که در ادامه به آن پرداخته می‌شود. یکی از راه‌های بهبود GD، feature scaling است. در دیتاست MNIST هر یک از تصاویر دارای 1*28*28 پیکسل است. عدد ۱ نشان دهنده grayscale بودن تصاویر است. همانطور که گفته شد، 28*28 یا 784 تعداد پیکسل‌های تصاویر MNIST هستند که به عنوان ویژگی‌ها به شبکه داده می‌شود. همچنین هر پیکسل دارای مقداری بین ۰ تا ۲۵۵ است بنابراین با تقسیم مقادیر ماتریس ویژگی، مقدار آنها بین ۰ تا ۱ شده است.

- **طراحی معماری شبکه**

گفته شد که ماتریس ویژگی به عنوان ورودی به شبکه داده می‌شود و به دلیل وجود اعداد ۰ تا ۹، خروجی به صورت ۱۰ کلاس خواهد بود. در این قسمت یه شبکه سه‌لایه (دو لایه پنهان و یک لایه خروجی) طراحی شده است. شبکه با استفاده از مدل sequential نوشته شده است.

با استفاده از سه لایه fully connected، که در کراس با Dense مشخص می‌شود، لایه‌های مدنظر ساخته شده‌اند. منظور از fully connected آن است که تمام نورون‌های یک لایه به لایه بعدی متصل است. این نوع اتصال ممکن است منجر به overfitting شود و قدرت تعمیم شبکه را با مشکل روبرو کند. به همین دلیل بعضاً از dropout استفاده می‌شود. در واقع با استفاده از dropout درصدی از نورون‌های هر لایه به صورت رندوم حذف می‌گردد. بدین صورت که دیگر ویژگی‌ها به همه نورون‌های لایه بعد نمی‌روند. البته در اینجا از dropout استفاده نشده است.

باید ماتریس ویژگی که 28*28 است به یک بردار 784 درایه‌ای تبدیل گردد یا به اصطلاح flatten شود. لایه اول ماتریس ویژگی (28*28) را به 512 نورون متصل می‌کند. لایه دوم 512 نورون (خروجی لایه اول) را به 512 نورون دیگر متصل می‌کند و لایه سوم خروجی لایه دوم را به 10 کلاس که همان خروجی‌های شبکه هستند متصل می‌کند. همچنین activation function لایه اول و دوم تابع relu و لایه سوم تابع softmax است. درنهایت شبکه تحت عنوان net به صورت زیر خواهد بود:

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 512)               401920
_________________________________________________________________
dense_5 (Dense)              (None, 512)               262656
_________________________________________________________________
dense_6 (Dense)              (None, 10)                5130
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
```

شکل ۱ – معماری لایه‌های طراحی شده برای شبکه MLP

- **آموزش شبکه**

نحوه محاسبه Loss Function با استفاده از Cross Entropy است که پیشتر راجع به آن توضیح داده شد. همچنین Optimizer استفاده شده در این قسمت، Stochastic Gradient Descent است که در این قسمت است که Learning Rate تعیین می‌شود. از ۳۰ ایپاک برای یاددهی شبکه استفاده شده است. در مسیر forward، 48k ورودی برچسب‌دار مربوط به train_data به شبکه داده و خروجی ده کلاس پیشبینی می‌شود.

در مسیر backward، مقدار loss محاسبه می‌شود و optimizer که در اینجا Stochastic GD است مقادیر وزن و بایاس و درنهایت loss را آپدیت می‌کند.

در قسمت اعتبارسنجی شبکه نیز مقدار loss محاسبه می‌گردد. بدیهی است به تعداد ایپاک باید تابع هزینه برای داده‌های train و valid وجود داشته باشد. اما تابع loss رو هر داده ورودی پیاده شده است. بنابراین لازم از در هر ایپاک، مقدار loss به صورت میانگین نمایش داده شود. درنهایت برای هر ایپاک، مقدار train_loss و valid_loss مشاهده گردید که در ادامه آمده است. این مقادیر برای batch_size = 32 و learning rate = 0.1 هستند.

با استفاده از شبکه train شده جدول زیر پر شده است. لازم به ذکر است به عنوان validationscore از دو پارامتر دقت و خطا شبکه در تشخیص برچسب داده‌های validation استفاده شده است.

جدول ۱ – بررسی عملکرد شبکه با تغییر پارامترهای مدل همچون learning rate و batch size

Batch Size	Learning Rate	Runtime(sec)	Validationscore	
			Validation Accuracy	Validation Loss
4	0.1	1669	96.03	0.230
4	0.01	1675	98.18	0.092
32	0.1	217	98.15	0.086
32	**0.01**	**272**	**98.19**	**0.087**

همانطور که مشاهده می‌شود افزایش batch size منجر افزایش دقت و کاهش loss در تشخیص برچسب داده‌های validation می‌شود. همچنین افزایش learning rate، دقت شبکه را کاهش می‌دهد اما به قیمت افزایش زمان runtime خواهد بود. البته معیار مقایسه در اینجا، دقت شبکه در تشخیص داده‌های validation است که در تمرین دوم نیز از آن استفاده می‌شود. در حالت batch size=32 و learning rate =0.01 بهتر جوابگو بوده است.

هنگامی که از batch size کمتر استفاده می‌شود، استفاده از امکان موازی‌سازی پروسه کاهش یافته و runtime شبکه افزایش می‌یابد؛ به گونه‌ای که برای batch size=4 زمان runtime درحدود ۲۷ دقیقه و برای batch size =32 این زمان ۴ دقیقه طول کشید.

- **تست شبکه آموزش‌داده شده**

تا اینجا شبکه با استفاده از داده‌های train آموزش داده شده است. حال لازم است به مدل مدنظر با بهترین هایپرپارامترهای بدست آمده، داده‌های test اعمال گردند. قبلاً گفته شد که استفاده از داده‌های test در آموزش شبکه هیچی نقشی ندارند بلکه قدرت تعمیم شبکه برای داده‌هایی که

شبکه ندیده است را می‌سنجد. با استفاده از امکان evaluate می‌توان خطا و دقت شبکه برای داده‌های test را مشاهده کرد که در ادامه گزارش شده است.

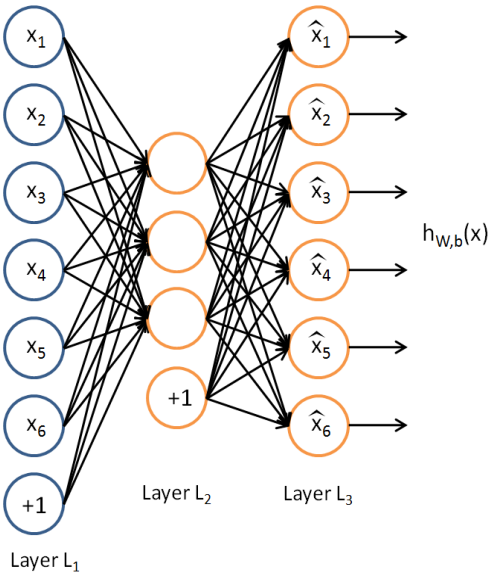
```
Test Loss is 0.07258734022355638 .
Test Accuracy is 0.9777 .
```

شکل ۲ – میزان خطا و دقت برای شبکه طراحی شده

۲. طراحی شبکه Autoencoders

۱,۲ طراحی یک شبکه Autoencoder و مقایسه با نتایج سؤال ۱

شبکه‌های Autoencoder، دارای دو بخش encoder و decoder هستند. در بخش encoder ابعاد داده‌های ورودی در جهت حذف پوچی، کاهش یافته و اطلاعات را فشرده می‌سازد. بخش decoder ورودی را از اطلاعات فشرده‌شده بازیابی می‌کند. بنابراین Autoencoder یک شبکه عصبی است که به یاد می‌گیرد که به صورت خودکار، داده‌های ورودی را encode و سپس decode کند. درواقع Autoencoder یک شبکه عصبی supervised است که با به کارگیری backpropagation، در راستای یکی کردن target با ورودی قدم برمی‌دارد. بنابراین هدف از Autoencoder یافتن تابع $h_{w,b}(x) = x$ است. البته این شبکه‌ها خاص بوده و دقت پایینی برای داده‌هایی که ندیده‌اند، دارد. در ادامه یک شبکه Autoencoder آمده است.

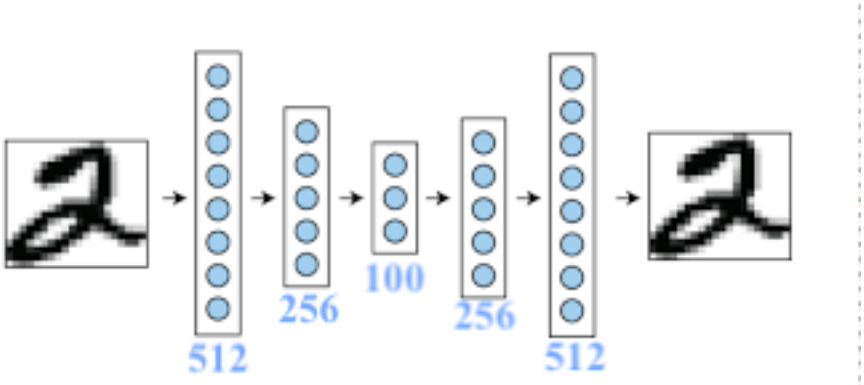


شکل ۳ – معماری یک شبکه Autoencoder

برای طراحی یک شبکه Autoencoder همانند سؤال یک عمل شده است با این تفاوت که خروجی همان ورودی است و تعداد لایه‌ها افزایش یافته است. در ابتدا داده‌های MNIST خوانده شده‌اند و بدیهی است که نیازی به داده‌های y_train و y_test نیست و خروجی شبکه باید معادل ورودی باشد. همانطور که پیشتر دیده شد، برای

بهبود GD لازم اس feature scaling صورت گیرد و ماتریس ورودی که شامل 28*28 پیکسل ۰ تا ۲۵۵ است به صورت اعداد ۰ تا ۱ درآید. همچنین ماتریس ورودی به یک بردار ۷۸۴ درایه‌ای تبدیل گردید

برای طراحی شبکه از ۶ لایه استفاده شده است؛ لایه اول، بردار ورودی که دارای ۷۸۴ درایه است را به ۵۱۲ نورون تبدیل می‌کند. لایه دوم خروجی لایه اول را ۲۵۶ نورون و لایه سوم، خروجی لایه دوم را به ۱۰۰ نورون تبدیل می‌کند. تا این قسمت بخش انکودر پیاده‌سازی شده است. در بخش انکودر، ۱۰۰ نورون ابتدا به ۲۵۶ نورون، سپس ۵۱۲ نورون و درنهایت به همان ۷۸۴ درایه ورودی تبدیل می‌شوند. در ۵ لایه پنهان، از تابع فعال‌ساز sigmoid و در لایه خروجی از softmax استفاده شده است. بنابراین بخش دیکودر نیز طراحی گردید. معماری Autoencoder طراحی‌شده در ادامه آمده است.



شکل ۴ – معماری Autoencoder طراحی شده

از Stochastic Gradient Descend به عنوان optimizer استفاده شده است. برای یکسان بودن شرایط سؤال یک و دو لازم است به غیر پارامترهایی که در مقایسه تأثیر دارند، بقیه یکسان باشند. پس از learning rate=0.01 و batch size=32 استفاده شده است. همچنین تابع هزینه cross entropy است. مدل Autoencoder در ادامه آمده است.

Layer (type)	Output Shape	Param #
input_3 (InputLayer)	(None, 784)	0
dense_13 (Dense)	(None, 512)	401920
dense_14 (Dense)	(None, 256)	131328
dense_15 (Dense)	(None, 100)	25700
dense_16 (Dense)	(None, 256)	25856
dense_17 (Dense)	(None, 512)	131584
dense_18 (Dense)	(None, 784)	402192

Total params: 1,118,580
Trainable params: 1,118,580
Non-trainable params: 0

شکل ۵ – معماری لایه‌ها برای Autoencoder طراحی‌شده

درنهایت با امکان fit، داده‌های مربوط به train و validation به مدل داده شده است. لازم به ذکر است همانند سؤال اول، ۲۰ درصد از داده‌های train به validation اختصاص داده شده است. دقت و هزینه شبکه برای داده‌های train و validation در ادامه آمده است. مشاهده می‌شود که دقت برای داده‌های train و validation ۸۰,۹۳ درصد است.

```
Epoch 30/30
48000/48000 [==============================] - 10s 213us/step - loss: 0.7399 - acc: 0.8087 - val_loss: 0.7328 - val_acc: 0.8093
```

شکل ۶ – دقت و خطای شبکه autoencoder طراحی‌شده با اعمال داده‌های train و validation

حال با اعمال داده‌های test به شبکه، که تاکنون ندیده است، دقتی در ۸۰,۷۲ درصد به دست می‌اید که در ادامه قابل‌مشاهده است.

```
Test Loss is 0.7469665225028992 .
Test Accuracy is 0.8072424722671508 .
```

شکل ۷ – دقت و خطای شبکه autoencoder طراحی‌شده با اعمال داده‌های test

حال شبکه encoder و وزن‌های مربوط به آن در دو فایل تحت عنوان encoder.json و encoder_weights ذخیره شده و به شبکه mlp سؤال اول داده می‌شود. به گونه‌ای که ورودی mlp دیگر، ماتریس ویژگی 28*28 نخواهد بود بلکه خروجی انکودر است.

در این قسمت، خروجی شبکه انکودر که همان ورودی‌های کاهش یافته است، به شبکه mlp طراحی‌شده در سؤال اول داده می‌شود و با سؤال یک که 784 پیکسل بدون کاهش بعد به mlp داده شد مقایسه می‌شود. ابتدا لازم است دو فایل مربوط به انکودر آپلود شوند. در دنهایت با اعمال x_train و y_train به شبکه و اختصاص دادن ۲۰ درصد از داده‌های train به validation، دقت شبکه برای ۳۰ ایپاک مشاهده شد که در ادامه آمده است. همانطور که مشاهده می‌شود دقت شبکه برای داده‌های validation در حدود ۹۷ درصد است. درصورتی که برای mlp درحدود ۹۸ درصد بود. احتمالاً به این دلیل است که در هنگام کاهش بعد، برخی از اطلاعات از بین می‌رود و دقت کاهش می‌یابد.

```
Epoch 30/30
48000/48000 [==============================] - 8s 176us/step - loss: 0.0013 - acc: 1.0000 - val_loss: 0.1157 - val_acc: 0.9766
```

شکل ۸ – دقت و خطای شبکه mlp کاهش بعد داده شده با استفاده از انکودر و اعمال داده‌های train و validation

۲,۲ طراحی یک Autoencoder به روش PCA

یکی از روش‌های کاهش بعد خطی داده‌ها، روش PCA یا Principal Component Analysis است که درنهایت به تعداد component ها، ویژگی باقی می‌ماند. به گونه‌ای که برای دیتاست MNIST، ویژگی‌ها از ۷۸۴ به ۱۰۰ کاهش می‌یابند. بنابراین با استفاده از pca، داده‌های MNIST (train و test) به صورت 60k بردار ۱۰۰ درایه‌ای می‌شوند. درنهایت داده‌های کاهش بعدیافته به شبکه MLP سؤال اول داده شده است. با استفاده از learning rate=0.01 برای GD و batch size=32 دقت شبکه برای داده‌های validation به صورت زیر شده است. همانطور که مشاهده می‌شود دقت شبکه MLP با استفاده از داده‌های کاهش بعدیافته، ۹۷,۸۵ درصد است.

```
Epoch 30/30
48000/48000 [==============================] - 7s 139us/step - loss: 0.0263 - acc: 0.9946 - val_loss: 0.0715 - val_acc: 0.9785
```

شکل ۹ – دقت و خطای شبکه mlp کاهش بعد داده شده با استفاده از PCA و اعمال داده‌های train و validation

درنهایت یک جدول برای مقایسه حالت‌های بررسی‌شده در ادامه آمده است. همانطور که پیشتر گفته شد، معیار مقایسه دقت شبکه در تشخیص داده‌های validation است. مشاهده می‌شود که دقت شبکه برای حالت MLP بدون کاهش ابعاد ورودی، بیش‌تر خواهد بود.

جدول ۲ – مقایسه دقت برای سه حالت مختلف شبکه MLP

	MLP	**MLP using encoder**	**MLP using PCA**
Validation Accuracy	**98.19**	97.66	97.85

۳. کاهش ویژگی با استفاده از RBM ها

هنگامی که دیتاست شامل داده‌های grayscale، همانند داده‌های MNIST، است، یعنی پیکسل‌ها نماینده طیف سیاه تا سفید هستند، RBM برنولی می‌تواند به صورت غیرخطی از داده‌های ویژگی استخراج کند. (BernouliRBM)

بنابراین یکی از کاربردهای RBM کاهش ابعاد است. در این مساله ابتدا همانند دو قسمت قبل از داده‌های MNIST فراخانی شده و در دو دسته Train و Test تقسیم شده‌اند. پس از تغییرات اولیه بر روی داده‌ها، همچون اسکیل کردن و تغییر فرمت، بخش RBM طراحی گردید. در ابتدا از کتابخانه sklearn، BernouliRBM فراخوانی شده است. پارامترهای مختلفی برای تابع برنولی وجود دارد؛ همچون میزان کاهش ابعاد ویژگی، نرخ Learning Rate، Batch size و تعداد تکرار برای آپدیت کردن پارامترها. مدل طراحی شده برای کاهش ویژگی تا ۱۰۰ بعد بر روی داده‌های train اعمال شده است.

```
[BernoulliRBM] Iteration 1, pseudo-likelihood = -145.07, time = 7.95s
[BernoulliRBM] Iteration 2, pseudo-likelihood = -127.28, time = 9.60s
[BernoulliRBM] Iteration 3, pseudo-likelihood = -115.31, time = 9.51s
[BernoulliRBM] Iteration 4, pseudo-likelihood = -108.70, time = 9.51s
[BernoulliRBM] Iteration 5, pseudo-likelihood = -101.94, time = 9.53s
[BernoulliRBM] Iteration 6, pseudo-likelihood = -100.06, time = 9.52s
[BernoulliRBM] Iteration 7, pseudo-likelihood = -97.14, time = 9.50s
[BernoulliRBM] Iteration 8, pseudo-likelihood = -94.50, time = 9.53s
[BernoulliRBM] Iteration 9, pseudo-likelihood = -93.75, time = 9.52s
[BernoulliRBM] Iteration 10, pseudo-likelihood = -91.57, time = 9.54s
```

شکل ۱۰ – کاهش بعد ویژگی‌ها با استفاده از BernouliRBM

سپس شبکه MLP طراحی شده در سؤال یک اعمال می‌شود. البته باید توجه داشت که ورودی‌های لایه اول به صورت کاهش بعد یافته (۱۰۰تایی) به شبکه اعمال خواهد شد. شبکه همانند قبل دارای دو لایه مخفی با ۵۱۲ نورون و لایه آخر با ۱۰ نورون است. اپتیمایزر نیز SGD با learning rate=0.01 است. معیار اندازه‌گیری crossentropy است. در ادامه شبکه MLP و تعداد پارامترها آمده است.

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_16 (Dense)             (None, 512)               51712
_________________________________________________________________
dense_17 (Dense)             (None, 512)               262656
_________________________________________________________________
dense_18 (Dense)             (None, 10)                5130
=================================================================
Total params: 319,498
Trainable params: 319,498
Non-trainable params: 0
_________________________________________________________________
```

شکل ۱۱ – شبکه MLP طراحی شده در سؤال اول

درنهایت شبکه با داده‌های train جدید، batch_size=32 و تعداد ایپاک ۳۰ آموزش داده شد و دقت و خطا شبکه به صورت زیر مشاهده گردید. البته ۲۰ درصد از داده‌های train به داده‌های validation اختصاص داده شده است که همانند دو قسمت قبل می‌باشد. لازم به ذکر است runtime در حدود ۲۴۰ ثانیه طول کشید.

```
Epoch 30/30
48000/48000 [==============================] - 7s 155us/step - loss: 0.0853 - acc: 0.9745 - val_loss: 0.1126 - val_acc: 0.9664
```

شکل ۱۲ – میزان خطا و دقت شبکه برای داده‌های train و test با استفاده از RBM

درنهایت جدول انتهایی در سؤال دو، با افزودن مقادیر بدست آمده در سؤال سوم به صورت زیر درآمد. مشاهده می‌شود که شبکه MLP با استفاده از RBM دارای دقت درحدود ۹۶,۷ درصد است از سایر حالت کمتر است.

جدول ۳ – مقایسه حالت‌های مختلف کاهش ویژگی ورودی و اعمال MLP

	MLP	MLP using encoder	MLP using PCA	MLP using RBM
Validation Accuracy	**98.19**	97.66	97.85	96.64

۴. طراحی شبکه MLP جهت پیش‌بینی قیمت خانه

۱,۴ آموزش شبکه با تک‌لایه مخفی

در این بخش یک شبکه MLP در راستای پیش‌بینی قیمت خانه در شهر بوستون ارائه می‌شود. برخلاف تمرین ۱ تا ۳، این یک مساله رگرسیون است و هدف بدست آوردن یک تابع بین ورودی و خروجی است. دیتاست استفاده شده، ۱۳ نوع ویژگی مربوط به خانه، به عنوان مثال، متراژ، تعداد اتاق، دسترسی به مدرسه و ...، را بررسی می‌کند که مربوط به ۵۰۵ خانه است. بدیهی است که قیمت خانه، باید به عنوان خروجی تابع درنظر گرفته شود. پس لازم است با استفاده از یک بردار ۱۳ درایه‌ای، شبکه مدنظر طراحی شود و mapping صورت گیرد.

در ابتدا با استفاده از کتابخانه pandas، فایل مربوط به اطلاعات خانه در colab فراخوانی شده است. دیتاست به عنوان یک ماتریس 505*14 بعدی است. ۱۳ ستون هر سطر به عنوان ورودی به شبکه داده می‌شود و ستون آخر خروجی شبکه است. سپس با استفاده از کتابخانه sklearn داده‌های ورودی و خروجی نرمالایز شده‌اند تا شبکه مستقل از مقدار ویژگی‌ها تصمیم‌گیری کند. بدین صورت داده‌ها بین ۰ تا ۱ scale شده‌اند. همچنین با استفاده از دستور train_test_split، ۲۰ درصد از داده‌های دیتاست به عنوان داده‌های test استفاده شده است. بدین ترتیب داده‌های train اطلاعات مربوط به ۴۰۴ خانه و داده‌های test اطلاعات مربوط به ۱۰۱ خانه است. با استفاده از کتابخانه keras، از مدل sequential استفاده شده و شبکه مدنظر با تک لایه مخفی به صورت زیر ساخته شده است:

لایه اول دارای ۱۰ نورون بوده و ۱۳ ویژگی هر خانه را به عنوان ورودی دریافت می‌کند. لایه دوم همان لایه خروجی است و یک نورون به عنوان خروجی شبکه می‌سازد. لازم به ذکر است لایه‌ها dense و یا fully connected هستند. در لایه پنهان از تابع فعال‌ساز relu استفاده شده است. اپتیمایزر استفاده‌شده Adam است و معیار سنجش، Mean Square Error است. در ادامه خلاصه‌ای از پارامترهای شبکه آمده است.

```
Model: "sequential_17"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_33 (Dense)             (None, 10)                140
_________________________________________________________________
dense_34 (Dense)             (None, 1)                 11
=================================================================
Total params: 151
Trainable params: 151
Non-trainable params: 0
_________________________________________________________________
```

شکل ۱۳ – معماری لایه‌های شبکه تک‌لایه

برای آموزش شبکه از همان ۴۰۴ داده گفته شده، استفاده شده است به گونه‌ای که x و y به شبکه داده شده است. همچنین از batch size=16 و تعداد ۱۰۰ ایپاک برای تکرار آپدیت کردن وزن داده‌ها استفاده شده است. در نهایت ۲۰ درصد از داده‌ای train به عنوان داده‌های validation درنظرگرفته شده است. خطای mse شبکه، برای داده‌های train و validation بعد از گذراندن ۱۰۰ ایپاک آمده است. مشاهده می‌شود که خطای داده‌های validation از train بیشتر است.

```
Epoch 100/100
323/323 [==============================] - 0s 183us/step - loss: 0.0074 - val_loss: 0.0109
```

شکل ۱۴ – خطا MSE شبکه MLP تک‌لایه برای داده‌های valid و train

- **کاهش ابعاد ورودی با استفاده از PCA خطی**

در این قسمت با استفاده از کتابخانه sklearn، PCA خطی به محیط افزوده شده است و ابعاد ورودی با استفاده از آن کاهش داده شده است. همانطور که گفته شده، بردار ورودی شامل ۱۳ ویژگی، به ۱۱ ویژگی کاهش یافته است. ورودی‌های کاهش‌یافته به شبکه اعمال شده است و برای ایپاک ۱۰۰، خطا به صورت زیر مشاهده گردید. لازم به ذکر است به جز ابعاد ورودی، سایر پارامترها یکسان بوده است. مشاهده می‌شود که میزان خطا در حالت استفاده از PCA کاهش یافته است. درنتیجه با حذف اطلاعات، شبکه با خطای کمتری یاددهی می‌شود.

```
Epoch 100/100
323/323 [==============================] - 0s 221us/step - loss: 0.0062 - val_loss: 0.0097
```

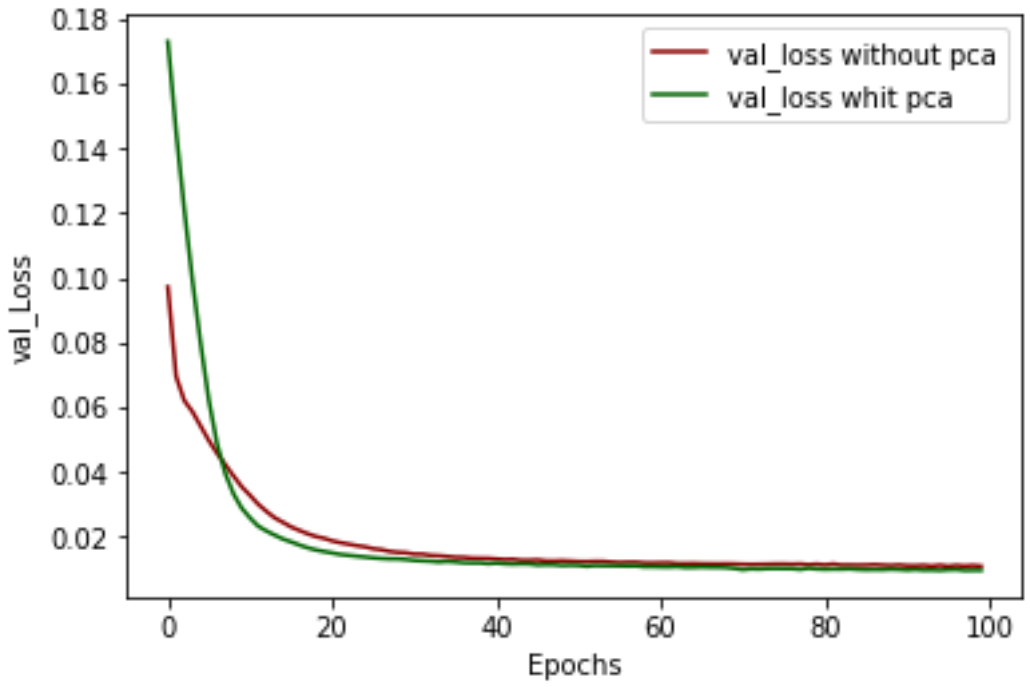
شکل ۱۵ – خطا MSE شبکه MLP تک‌لایه کاهش بعد داده شده با روش PCA برای داده‌های valid و train

همچنین دو معیار مینیمم و ماکزیمم MSE برای دو حالت ورودی و ورودی کاهش‌یافته بررسی شد که در ادامه آمده است. مشاهده می‌شود که برای حالت کاهش‌یافته، خطای MSE ماکزیمم، بیشتر شده است. بدین معنی که با حذف اطلاعات، درابتدا خطا افزایش می‌یابد اما بعد از گذراندن ۱۰۰ ایپاک مقدار خطا کمتر از حالت بدون کاهش بعد شده است.

```
Max_loss and Min_loss are 0.09721499629732636 , 0.010896338018937968 .
Max_loss_pca and Min_loss_pca are 0.17301385113854467 , 0.00963814733527945 .
```

شکل ۱۶ – خطا ماکزیمم و مینیمم دو شبکه MLP ساده و کاهش بعد داده شده

در ادامه خطای دو حالت بررسی شده، ورودی و ورودی کاهش‌یافته، برای شبکه MLP تک لایه در هر ایپاک به صورت زیر است.



شکل ۱۷ – بررسی خطا MSE برای دو شبکه MLP ساده و کاهش بعد داده شده

۲,۴ افزودن یک لایه مخفی به شبکه MLP طراحی‌شده

در این قسمت به شبکه طراحی شده در بخش قبلی یک لایه اضافه می‌شود. بنابراین شبکه با دو لایه مخفی خواهد بود. لایه اضافه‌شده، دارای تابع فعال‌ساز relu است و شامل نورون می‌باشد. سایر پارامترها، همچون درصد تخصیص داده‌ها، اپتیمایزر، معیار مقایسه، batch size، تعداد ایپاک و ... یکسان است. میزان خطا در ایپاک ۱۰۰ به صورت زیر مشاهده گردید. همانطور که انتظار می‌رفت، این خطا از خطای مربوط به شبکه تک لایه کمتر است.

```
Epoch 100/100
323/323 [==============================] - 0s 346us/step - loss: 0.0078 - val_loss: 0.0090
```

شکل ۱۸ – خطا MSE شبکه MLP دولایه برای داده‌های valid و train

سپس همانند حالت قبل، داده‌ها با استفاده از ابزار PCA خطی کاهش بعد داده شدند. به‌گونه‌ای‌که ورودی به جای ۱۳ ویژگی دارای ۱۱ ویژگی شد. پس از آموزش مجدد شبکه دولایه، با استفاده از ورودی کاهش بعد داده شده، خطای MSE بعد از ۱۰۰ ایپاک به صورت زیر مشاهده گردید.

```
Epoch 100/100
323/323 [==============================] - 0s 335us/step - loss: 0.0042 - val_loss: 0.0062
```

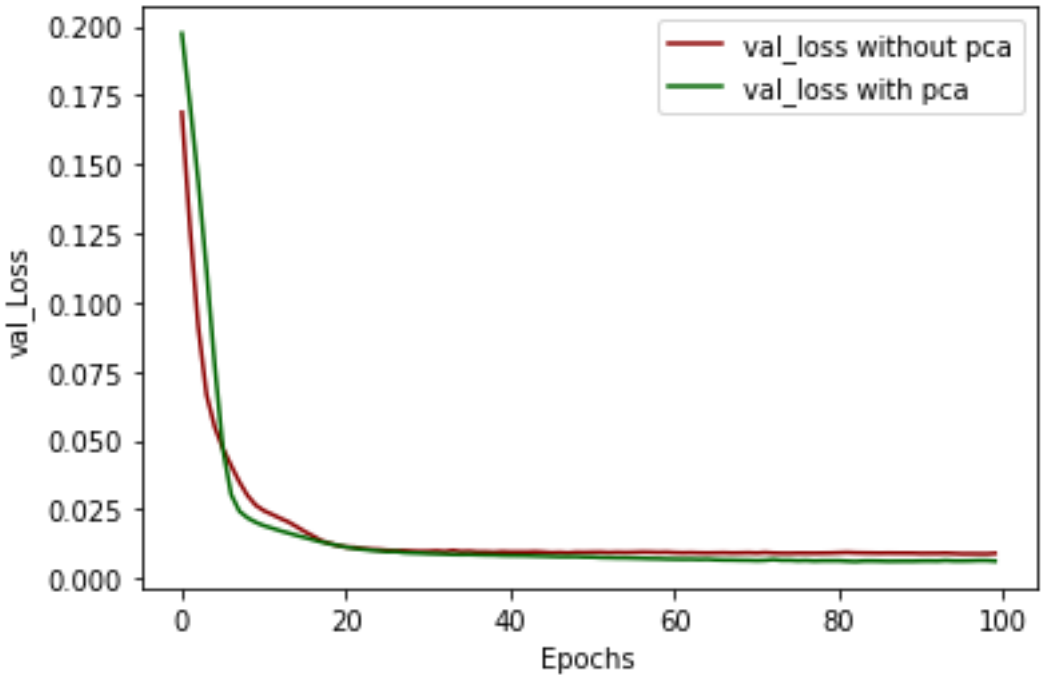
شکل ۱۹ – خطا MSE شبکه MLP تک‌لایه کاهش بعد داده شده با روش PCA برای داده‌های valid و train

همچنین خطای مینیمم و ماکزیمم برای دو حالت مذکور بررسی شد. ماکزیمم خطا برای حالت استفاده از کاهش بعد افزایش یافت.

```
Max_loss and Min_loss are 0.16885385965859448 , 0.008811200713094922 .
Max_loss_pca and Min_loss_pca are 0.19748268607589933 , 0.006025095210471018 .
```

شکل ۲۰ – خطا ماکزیمم و مینیمم دو شبکه MLP ساده و کاهش بعد داده شده

حال نمودار کاهش خطا داده‌های validation برای دو حالت بدون کاهش بعد و با کاهش بعد آمده است.



شکل ۲۱ – **بررسی خطا MSE برای دو شبکه MLP ساده و کاهش بعد داده شده**

درنهایت برای دو شبکه MLP طراحی شده و دو حالت استفاده از کاهش بعد و بدون آن، جدول زیر طراحی شده است. همانطور که مشاهده می‌شود بهترین حالت خطای MSE، استفاده از شبکه دو لایه و به همراه ۲ ویژگی کاهش بعد است. البته در این حالت خطای ماکزیمم بیشترین حالت خود را دارد که در ایپاک‌های اولیه اتفاق می‌افتد. همانطور که پیشتر گفته شد با کاهش بعد، در ابتدا شبکه با خطای زیادی کار را شروع می‌کند ولی پس از گذراندن ایپاک‌ها، خطا جبران می‌شود. به طور کلی کاهش بعد خطای ماکزیمم را افزایش می‌دهد. همچنین افزودن لایه باعث کاهش خطا گردیده است.

جدول ۴ – مقایسه خطا MSE شبکه MLP

Network	MSE Validation Loss	Maximum Validation Loss
One Hidden Layer	0.0109	**0.0972**
One Hidden Layer with PCA	0.0097	0.1730
Two Hidden Layer	0.0090	0.1688
Two Hidden Layer with PCA	**0.0062**	0.1974

Codes

Q1.

```
from keras.datasets import mnist
from keras.utils import np_utils

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train = x_train.reshape(60000, 784)
x_test  = x_test.reshape (10000, 784)

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

x_train /= 255
x_test  /= 255

y_train = np_utils.to_categorical(y_train)
y_test  = np_utils.to_categorical(y_test)
```

```
from keras.models import Sequential
from keras.layers import Dense

net = Sequential()
net.add(Dense(512, activation  = 'relu', input_shape = (784,)))
net.add(Dense(512, activation  = 'relu'))
net.add(Dense(10 , activation  = 'softmax'))
```

```
from keras import optimizers

sgd = optimizers.SGD(lr=0.1)
net.compile(optimizer=sgd, loss='categorical_crossentropy', metrics=['accuracy'])

net.summary()
```

```
import timeit

start = timeit.default_timer()
```

```
trained_model = net.fit(x_train, y_train, batch_size = 32, epochs = 30, validation_split = 0.2)
history = trained_model.history
```

```
stop = timeit.default_timer()
print('Time: ', stop - start)
```

```
score = net.evaluate(x_test, y_test, verbose=0)
print('Test Loss is', score[0], '.')
print('Test Accuracy is', score[1],'.')
```

Q2. With Decoder

```
from keras.datasets import mnist
from keras.utils import np_utils
import numpy as np

(x_train, _),(x_test, _) = mnist.load_data()
x_train = x_train.reshape((60000, 784))
x_test  = x_test.reshape ((10000, 784))

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

x_train /= 255
x_test  /= 255
```

```
from keras.layers import Input, Dense
from keras.models import Model

input_img = Input(shape=(784,))
encoded = Dense(512, activation='relu')(input_img)
encoded = Dense(256, activation='relu')(encoded)
encoded = Dense(100, activation='relu')(encoded)

decoded = Dense(256, activation='relu')(encoded)
decoded = Dense(512, activation='relu')(decoded)
decoded = Dense(784, activation='softmax')(decoded)

autoencoder = Model(input_img, decoded)
```

```
from keras import optimizers

sgd = optimizers.SGD(lr=0.01)
autoencoder.compile(optimizer=sgd, loss='binary_crossentropy', metrics=['accuracy'])

autoencoder.summary()
```

```
import timeit

start = timeit.default_timer()
```

```
trained_model = autoencoder.fit(x_train, x_train, batch_size = 32, epochs = 30, validation_split = 0.2)
history = trained_model.history
```

```
stop = timeit.default_timer()
print('Time: ', stop - start)
```

```
score = autoencoder.evaluate(x_test, x_test, verbose=0)
print('Test Loss is', score[0], '.')
print('Test Accuracy is', score[1],'.')
```

```
encoder = Model(input_img, encoded)
json_encoder = encoder.to_json()
with open("encoder.json", "w") as json_file:
    json_file.write(json_encoder)
#encoder.save_weights("encoder_weights.h5")
encoder.save_weights('encoder_weights')
```

Q2. MLP

```
from keras.datasets import mnist
from keras.utils import np_utils
import numpy as np

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train = x_train.reshape((60000, 784))
x_test  = x_test.reshape ((10000, 784))

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

x_train /= 255
x_test  /= 255

y_train = np_utils.to_categorical(y_train)
y_test  = np_utils.to_categorical(y_test)
```

```
from keras.models import load_model
from keras.models import model_from_json

json_file = open('encoder-2.json', 'r')
loaded_model = json_file.read()
json_file.close()
encoder = model_from_json(loaded_model)
encoder.load_weights("encoder_weights-2.dms")
print("Model loaded")
```

```
from keras.layers import Input, Dense
from keras.models import Model

mlp = Dense(512, activation='relu')(encoder.output)
mlp = Dense(512, activation='relu')(mlp)
mlp = Dense(10, activation='softmax')(mlp)

autoencoder = Model(encoder.inputs, mlp)
```

```
from keras import optimizers

sgd = optimizers.SGD(lr=0.01)
autoencoder.compile(optimizer=sgd, loss='categorical_crossentropy', metrics=['accuracy'])

autoencoder.summary()
```

```
import timeit

start = timeit.default_timer()
```

```
trained_model = autoencoder.fit(x_train, y_train, batch_size = 32, epochs = 30, validation_split = 0.2)
history = trained_model.history
```

```
stop = timeit.default_timer()
print('Time: ', stop - start)
```

Q2. PCA

```
from keras.datasets import mnist
from keras.utils import np_utils

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train = x_train.reshape(60000, 784)
x_test  = x_test.reshape (10000, 784)

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

x_train /= 255
x_test  /= 255

y_train = np_utils.to_categorical(y_train)
y_test  = np_utils.to_categorical(y_test)
```

```
from sklearn.decomposition import PCA
import numpy as np

pca = PCA(n_components=100)
x_train_pca = pca.fit_transform(x_train)
x_test_pca  = pca.transform(x_test)
explained_var = pca.explained_variance_ratio_
np.sum(explained_var[0:100])
pca_out = explained_var[0:100]
```

```
from keras.models import Sequential
from keras.layers import Dense

net = Sequential()
net.add(Dense(512, activation  = 'relu', input_shape = (100,)))
net.add(Dense(512, activation  = 'relu'))
net.add(Dense(10 , activation  = 'softmax'))
```

```
from keras import optimizers

sgd = optimizers.SGD(lr=0.01)
net.compile(optimizer=sgd, loss='categorical_crossentropy', metrics=['accuracy'])
net.summary()
```

```
import timeit

start = timeit.default_timer()
```

```
trained_model = net.fit(x_train_pca, y_train, batch_size = 32, epochs = 30, validation_split = 0.2)
history = trained_model.history
```

```
stop = timeit.default_timer()
print('Time: ', stop - start)
```

Q3.

```
[24] from keras.datasets import mnist
     from keras.utils import np_utils
     import numpy as np

     (x_train, y_train),(x_test, y_test) = mnist.load_data()
     x_train = x_train.reshape((60000, 784))
     x_test  = x_test.reshape ((10000, 784))

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

     x_train /= 255
     x_test  /= 255

     y_train = np_utils.to_categorical(y_train)
     y_test  = np_utils.to_categorical(y_test)
```

```
[25] from sklearn.neural_network import BernoulliRBM

     rbm = BernoulliRBM(n_components=100, learning_rate=0.01, batch_size=32, n_iter=10, verbose=True, random_state=None)
     rbm.fit(x_train)
     train=rbm.transform(x_train)
     test=rbm.transform(x_test)
```

```
[26] from keras.layers import Input, Dense
     from keras.models import Sequential
     from keras.models import Model

     net = Sequential()
     net.add(Dense(512, activation = 'relu', input_dim=100, kernel_initializer='normal'))
     net.add(Dense(512, activation  = 'relu'))
     net.add(Dense(10 , activation  = 'softmax'))
```

```
[27] from keras import optimizers

     sgd = optimizers.SGD(lr=0.01)
     net.compile(optimizer=sgd, loss='categorical_crossentropy', metrics=['accuracy'])
     net.summary()
```

```
[28] import timeit

     start = timeit.default_timer()
```

```
[29] trained_model = net.fit(train, y_train, batch_size = 32, epochs = 30, validation_split = 0.2)
     history = trained_model.history
```

```
     stop = timeit.default_timer()
     print('Time: ', stop - start)
```

Q4. One Layer Simple and PCA

[123]
```
import pandas as pd

df = pd.read_csv('house_data.csv')
dataset = df.values

# df
# dataset.shape, dataset
```

[124]
```
x = dataset[:, 0:13]
y = dataset[:, 13]

# x.shape, y.shape
```

[125]
```
from sklearn import preprocessing
import numpy as np

min_max_scaler = preprocessing.MinMaxScaler()
x_scale = min_max_scaler.fit_transform(x)

TransformY = preprocessing.MinMaxScaler()
y_scale = TransformY.fit_transform(y.reshape(y.shape[0],1))

#x_scale, y_scale
```

[126]
```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(x_scale, y_scale, test_size=0.2)

#x_train.shape,x_test.shape,x_val.shape
```

[127]
```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential([
    Dense(10, activation='relu', input_shape=(13,)),
    Dense(1, kernel_initializer='normal')])
```

[128]
```
from keras import optimizers

model.compile(loss='mean_squared_error', optimizer='adam')
model.summary()
```

[129]
```
trained_model = model.fit(x_train, y_train, batch_size=16, epochs=100, validation_split=0.2) #validation_data=(x_val, y_val))

history = trained_model.history
```

[130]
```
from sklearn.decomposition import PCA
import numpy as np

pca = PCA(n_components=11)
x_train_pca = pca.fit_transform(x_train)
x_test_pca  = pca.transform(x_test)
explained_var = pca.explained_variance_ratio_
np.sum(explained_var[0:11])
pca_out = explained_var[0:11]
```

[131]
```
from keras.models import Sequential
from keras.layers import Dense

model_pca = Sequential([
    Dense(10, activation='relu', input_shape=(11,)),
    Dense(1, kernel_initializer='normal')])
```

[132]
```
model_pca.compile(loss='mean_squared_error', optimizer='adam')
```

[133]
```
trained_model_pca = model_pca.fit(x_train_pca, y_train, batch_size=16, epochs=100, validation_split=0.2)#, validation_data=(x_val, y_val))
history_pca = trained_model_pca.history
```

[134]
```
max_loss = max(history['val_loss'])
min_loss = min(history['val_loss'])

max_loss_pca = max(history_pca['val_loss'])
min_loss_pca = min(history_pca['val_loss'])

print('Max_loss and Min_loss are', max_loss,',', min_loss,'.')
print('Max_loss_pca and Min_loss_pca are', max_loss_pca,',', min_loss_pca,'.')
```

Q4. Two Layer Simple and PCA

```
import pandas as pd

df = pd.read_csv('house_data.csv')
dataset = df.values

# df
# dataset.shape, dataset
```

```
x = dataset[:, 0:13]
y = dataset[:, 13]

# x.shape, y.shape
```

```
from sklearn import preprocessing
import numpy as np

min_max_scaler = preprocessing.MinMaxScaler()
x_scale = min_max_scaler.fit_transform(x)
TransformY = preprocessing.MinMaxScaler()
y_scale = TransformY.fit_transform(y.reshape(y.shape[0],1))

#x_scale
```

```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(x_scale, y_scale, test_size=0.2)

# x_train.shape
```

```
from keras.models import Sequential
from keras.layers import Dense
#8,6,1
model = Sequential([
    Dense(10, activation='relu', input_shape=(13,)),
    Dense(8, activation='relu'),
    Dense(1, kernel_initializer='normal')])
```

```
from keras import optimizers

model.compile(loss='mean_squared_error', optimizer='adam')
model.summary()
```

```
import timeit
start = timeit.default_timer()
```

```
trained_model = model.fit(x_train, y_train, batch_size=16, epochs=100, validation_split=0.2)
history = trained_model.history
```

```
from sklearn.decomposition import PCA
import numpy as np

pca = PCA(n_components=11)
x_train_pca = pca.fit_transform(x_train)
x_test_pca  = pca.transform(x_test)
explained_var = pca.explained_variance_ratio_
np.sum(explained_var[0:11])
pca_out = explained_var[0:11]
```

```
from keras.models import Sequential
from keras.layers import Dense
#128,16,1
model_pca = Sequential([
    Dense(10, activation='relu', input_shape=(11,)),
    Dense(8, activation='relu'),
    Dense(1, kernel_initializer='normal')])
```

```
model_pca.compile(loss='mean_squared_error', optimizer='adam')
```

```
trained_model_pca = model_pca.fit(x_train_pca, y_train, batch_size=16, epochs=100, validation_split=0.2)
history_pca = trained_model_pca.history
```

```
max_loss = max(history['val_loss'])
min_loss = min(history['val_loss'])

max_loss_pca = max(history_pca['val_loss'])
```