



دانشکده علوم ریاضی  
گروه علوم کامپیوتر

## گزارش تمرین ۳

نگارش

ریحانه داورزنی

استاد

دکتر سعیدرضا خردپیشه

اردیبهشت ۱۴۰۰

## فهرست مطالب

### ۱ مقدمه

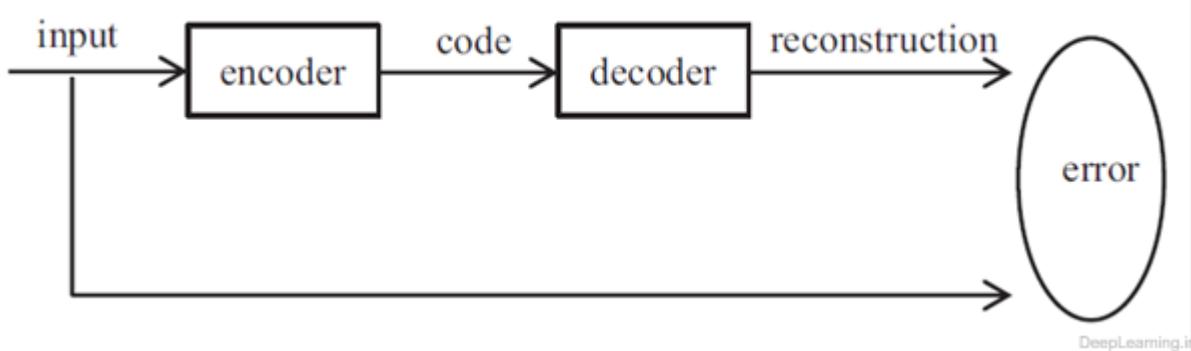
۶	سوال یک	۲
۶	مدل اول	۱.۲
۸	مدل دوم	۲.۲
۱۰	مدل سوم	۳.۲
۱۱	مدل چهارم	۴.۲
۱۲	مدل پنجم	۵.۲
۱۲	مدل ششم	۶.۲
۱۳	مدل هفتم	۷.۲
۱۴	مدل هشتم	۸.۲
۱۵	مدل نهم	۹.۲

### ۲ سوال دو

۱۷

## ۱ مقدمه

نوع خاصی از شبکه عصبی مصنوعی است که برای encode کردن بهینه یادگیری مورد استفاده قرار می‌گیرد. بجای آموزش شبکه و پیش‌بینی مقدار هدف  $Y$  در ازای ورودی  $X$ ، یک autoencoder آموزش می‌بینید تا ورودی  $X$  خود را بازسازی کند. بنابر این بردارهای خروجی همان ابعاد بردار ورودی را خواهند داشت. فرآیند کلی یک autoencoder در شکل زیر نشان داده شده است. در حین این فرآیند، با کمینه سازی خطای بازسازی<sup>۱</sup> بهینه می‌شود. کد متناظر همان ویژگی فراگرفته شده است.



autoencoder ها نقشی اساسی در یادگیری بدون نظارت و شبکه‌های عمیق ایفا می‌کنند. بازنمایی داده‌ها هدف استفاده از autoencoder هاست. استخراج ویژگی از مهم‌ترین بخش‌های حل یک مساله در یادگیری ماشین است. ویژگی‌های استخراج شده باید بتوانند برای دسته‌بندی استفاده شوند. بزرگترین مزیت autoencoder ها انتخاب خودکار این ویژگی‌های است. این دسته از شبکه‌های عصبی برای کاهش بُعد استفاده می‌شوند و هزینه‌های زمانی و حافظه‌ای پردازش را کاهش می‌دهند.

## استخراج ویژگی

برای مسائلی که در یادگیری ماشین مطرح می‌شوند همواره جهت دسته‌بندی و تشخیص داده‌های مختلف نیاز به ویژگی‌هایی داریم که موجب تمایز ورودی‌ها از یکدیگر

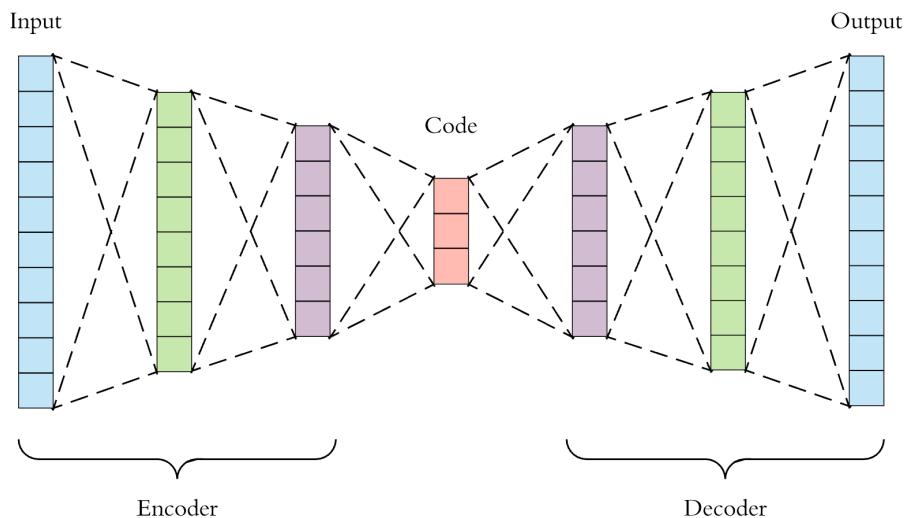
<sup>1</sup>reconstruction error

شوند. به مجموعه تکنیک‌هایی که منجر به یادگیری این ویژگی‌ها می‌شوند استخراج ویژگی می‌گوییم. چرا ما به استخراج ویژگی‌ها نیازمندیم؟ در یادگیری ماشین، بسته به نوع مساله، ورودی می‌تواند بسیار متنوع باشد. داده‌های ورودی اغلب افزونگی بسیاری دارند به این معنا که ما برای حل مساله نیاز به تمامی مقادیر داده‌ها نداریم و تنها بخشی از آن برای ما قابل استفاده است. از طرفی با توجه به محدودیت در قدرت پردازش از نظر حافظه و زمان مورد نیاز، ما باید با انجام تبدیلاتی بخشی از ورودی‌ها که برای ما قابل استفاده هستند را استخراج نماییم.

## ساختار دو بخشی autoencoder

ساختار اتوانکدر به دو بخش encoding و decoding تقسیم می‌شود. در بخش decoding داده‌های ورودی به فضای ویژگی‌ها نگاشت می‌شوند و در بخش encoding از فضای ویژگی مجدداً به حالت ابتدایی خود تبدیل می‌شوند. در واقع بخش اصلی یک autoencoder لایه‌ی پنهان میانی است که به عنوان ویژگی استخراج شده برای دسته‌بندی استفاده می‌شود.

تصویر زیر این دو بخشی بودن را به خوبی نشان می‌دهد.

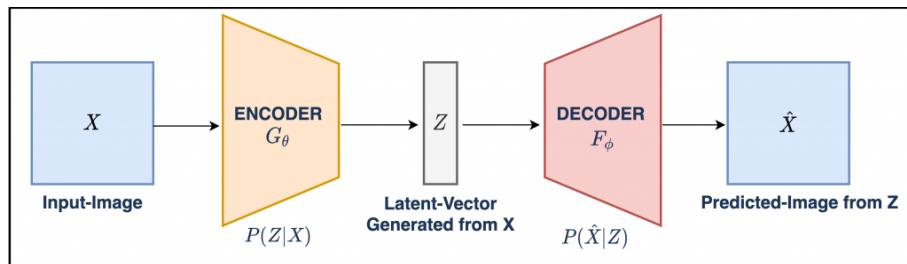


یکی از گونه‌های معروف *Variational AutoEncoder* ، autoencoder

است که در ادامه به آن می‌پردازیم.

## Variational autoencoder (VAE)

در VAE یک رابطه یک به یک نداریم بلکه عملاً روابط یک به چند است ، به این معنا که به جای توابع ، از توابع توزیع احتمال استفاده می‌شود. یعنی اگر یک  $x$  را به ما بدهد ،  $z$  هایی هستند که با آن  $z$  محتمل‌تر است ،  $x$  را ببینیم و در حالت عکس اگر یک  $z$  مشخص را در نظر بگیریم ، صرفاً به یک  $x$  ختم نخواهد شد ، بلکه مجموعه‌ای از  $x$  ها به واسطه‌ی یک  $z$  می‌توانند پدیدار شوند.



از فواید این روش می‌توان به این مورد اشاره کرد که از VAE به عنوان یک generator استفاده کرد. وقتی به شبکه decoder یک  $z$  انتخابی از توزیع ایجاد شده می‌دهیم ،  $x$  های مختلف تولید می‌کند که تا حدودی شبیه  $x$  ای که داشتیم است . به جای اینکه فقط یک آیتم خاص را learn کنیم ، مفهوم یا توزیع آیتم را یادمی‌گیرد.

## ۲ سوال یک

دیتاست این سوال ، شامل تصاویر خرابی است که در روند برنامه مشکل ایجاد می‌کند ، در ابتدا توسط کد زیر این تصاویر را حذف می‌کنیم:

```
for image in enumerate(images):
    try:
        Image.open('/content/files/dataset/dataset_updated/training_set/painting/' +image[1])
    except OSError:
        os.remove('/content/files/dataset/dataset_updated/training_set/painting/' +image[1])
```

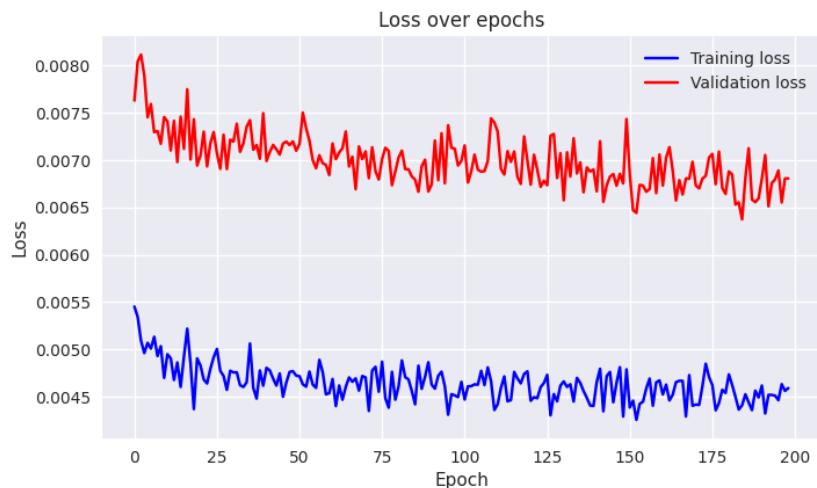
### ۱.۲ مدل اول

حال برای پیاده‌سازی شبکه اتوانکودر ، در ابتدا شبکه‌ای به صورت زیر را پیاده‌سازی می‌کنیم:

```
(color): Sequential(
    (0): Conv2d(1, 8, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(8, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): ReLU()
    (6): Conv2d(16, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (7): ReLU()
    (8): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU()
    (10): Conv2d(32, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (11): ReLU()
    (12): Upsample()
    (13): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (14): ReLU()
    (15): Upsample()
    (16): Conv2d(32, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU()
    (18): Upsample()
    (19): Conv2d(16, 2, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): Tanh()
)
```

در طول مدل‌هایی که برای این سوال پیاده‌سازی شده است ، از Mean Squared Error به عنوان تابع loss استفاده شده است دلیل آن این است که در مقالات اشاره شده است که MSE یک راه حل رایج در train کردن شبکه‌های عصبی که عمل رنگی کردن را انجام میدهد، است. [؟] [؟]  
تعداد epoch برای این شبکه 200 ، لرنینگ ریت برابر 0.001 و اپتیمایزر استفاده شده RMSprop است.

میزان loss در حالت validation و training به صورت زیر می‌باشد:



یک نمونه از تصویری که توسط این شبکه ، رنگی شده است را در زیر مشاهده می‌کنید:



## ۲.۲ مدل دوم

در مدل شبکه بعدی که به صورت زیر می‌باشد، عمیق‌تر از حالت قبل است و پارامترهای آن همانند مدل قبلی است:

```
(color): Sequential(  
    (0): Conv2d(1, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
    (1): ReLU()  
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))  
    (3): ReLU()  
    (4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
    (5): ReLU()  
    (6): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))  
    (7): ReLU()  
    (8): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
    (9): ReLU()  
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))  
    (11): ReLU()  
    (12): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
    (13): ReLU()  
    (14): Conv2d(512, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
    (15): ReLU()  
    (16): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
    (17): ReLU()  
    (18): Upsample()  
    (19): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
    (20): ReLU()  
    (21): Upsample()  
    (22): Conv2d(64, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
    (23): ReLU()  
    (24): Conv2d(32, 2, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
    (25): Tanh()  
    (26): Upsample()  
)
```

میزان loss در حالت validation و training با epoch 100 به صورت زیر می‌باشد:



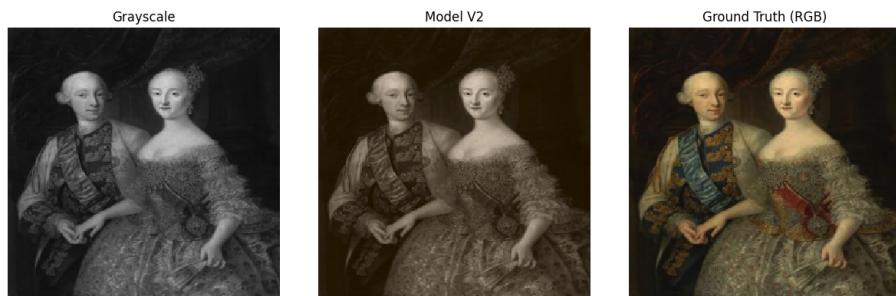
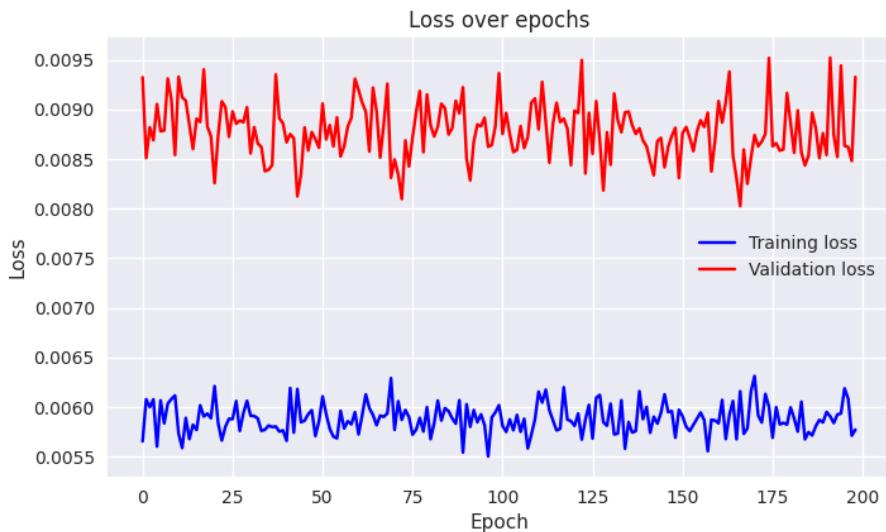
نمونه هایی از تصاویری که توسط این شبکه ، رنگی شده است را در زیر مشاهده می کنید:



همان طور که مشاهده می شود ، به دلیل عمیق بودن شبکه ، تصاویر به نحو بهتری نسبت به دیگر مدل ها رنگی شده اند . میزان loss تقریبا برابر مدل مرحله قبل است.

## ۳.۲ مدل سوم

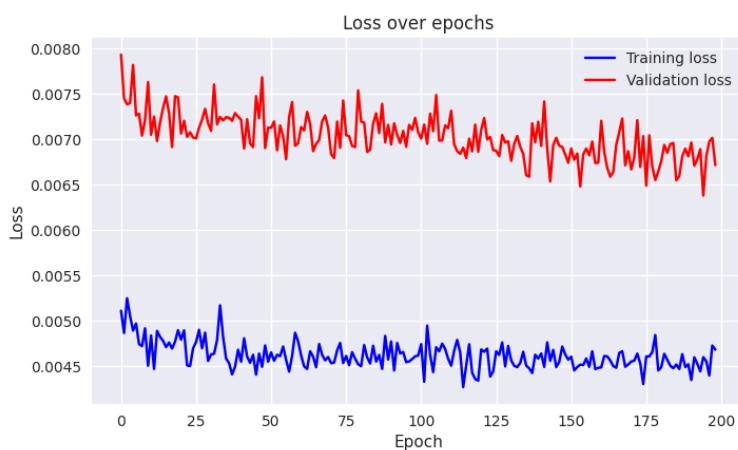
حال اگر همان مدل اول را دوباره train کنیم با این تفاوت که این بار لرنینگ را تغییر داده و برابر ۰.۰۰۴ قرار داده‌ایم، جواب‌های زیر حاصل می‌شود:



همان طورکه واضح است ، حاصل تصویر رنگی شده در مدل اول بهتر از این مدل است. و میزان loss نهایی هم در train و هم در validation در حالت قبل از مدل این مرحله کمتر است.

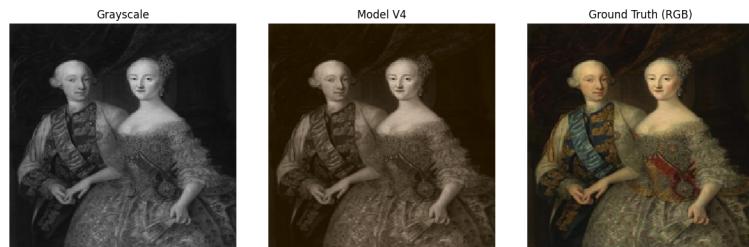
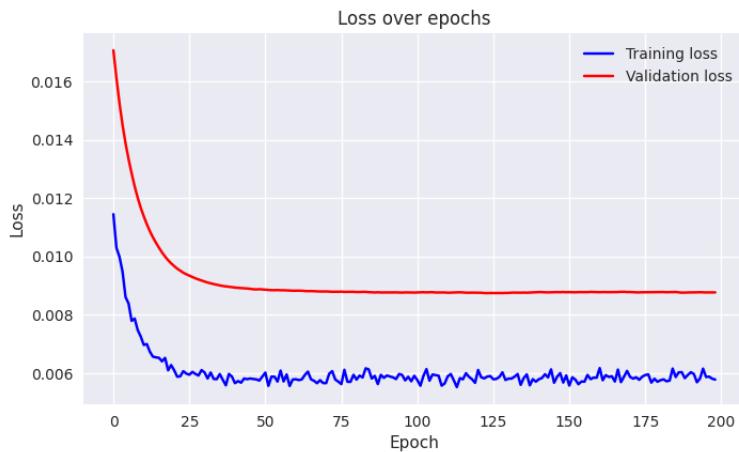
## ۴.۲ مدل چهارم

این بار مدل اول را با optimizer دیگری train می‌کنیم. از Adam برای این مرحله استفاده شده است. نتایج به شرح زیر است:  
تصاویر رنگی شده این مرحله، به نسبت مراحل قبل بهتر است و میزان loss تقریباً برابر اولین مدل می‌باشد.



## ۵.۲ مدل پنجم

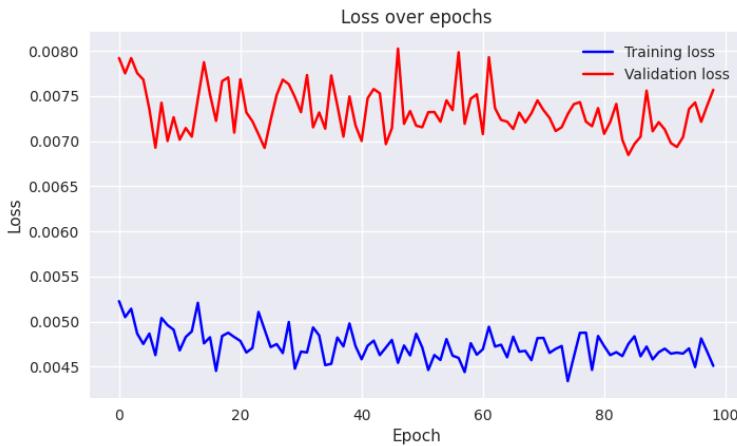
این بار مدل اول را با optimizer دیگری train می‌کنیم. از SGD برای این مرحله استفاده شده است. نتایج به شرح زیر است:



در این مدل ، تصویر رنگ شده نتیجه مطلوبی ندارد و میزان loss هم که روند نزولی دارد از مدل اول و چهارم بیشتر و از مدل دوم و سوم کمتر است.

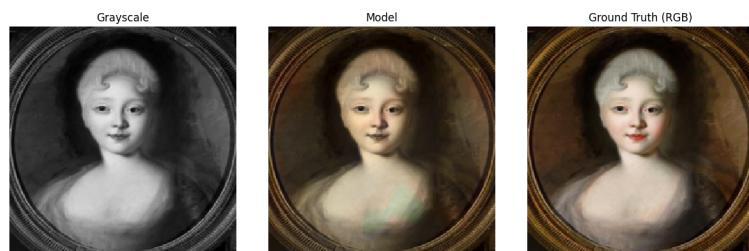
## ۶.۲ مدل ششم

مدل حالت دوم را که شبکه‌ای عمیق‌تر بود این بار با optimizer دیگری train می‌کنیم. از Adam برای این مرحله استفاده می‌کنیم . نتایج به صورت زیر می‌باشد:



## ۷.۲ مدل هفتم

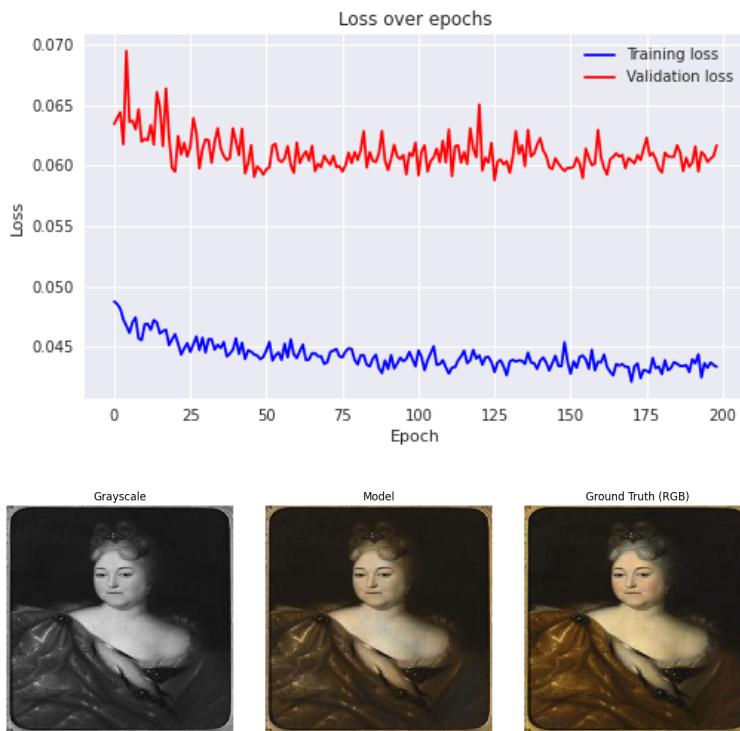
در این بخش ، از یک شبکه preTrain برای بخش encoder استفاده شده است. شبکه استفاده شده ، resnet18 است. با تعداد epoch برابر  $10^6$  و optimizer Adam ، حاصل به صورت زیر می باشد:



از لحاظ تصاویر رنگی شده ، نتایج به نسبت بهتری نسبت به مدل‌های قبل حاصل شده است ولی تفاوت اساسی ای با مدل‌های قبل ندارد . از لحاظ loss نیز تقریباً مدل‌های قبل که بهتر از بقیه عمل کرده بودند ، برابر است.

## ۸.۲ مدل هشتم

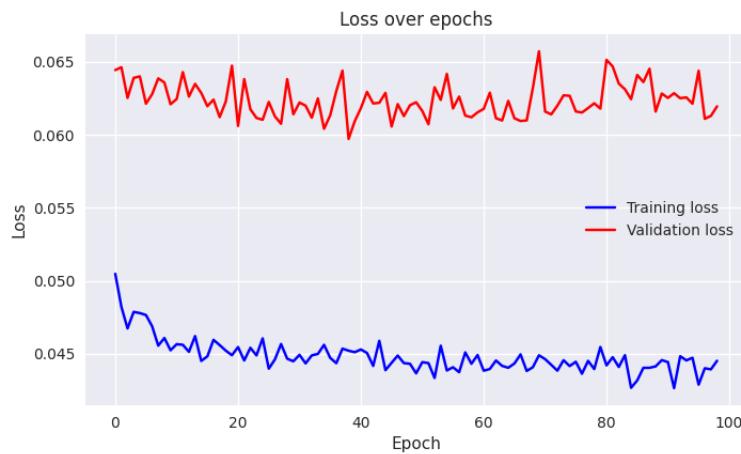
در این مدل ، به جای استفاده از تابع MSE ، از تابع MAE استفاده می‌شود و نیز از optimizer برای Adam است . نتایج به شرح زیر است :



نسبت به مدل چهارم که همانند این مدل است و فقط loss function این دو متفاوت است ، میزان loss این مدل به مقدار خیلی ناچیزی از مدل چهارم کمتر است و حاصل تصویر رنگ شده ، تفاوت آنچنانی با مدل چهارم ندارد.

## ۹.۲ مدل نهم

در این مدل ، به جای استفاده از تابع MSE در مدل ششم ، از تابع MAE استفاده میشود . نتایج به شرح زیر است :



نسبت به مدل ششم میزان loss به مقدار ناچیزی کمتر است و حاصل تصویر رنگ شده تفاوت چندانی با مدل ششم ندارد.

### ۳ سوال دو

برای این سوال یک شبکه VAE که لایه‌های آن کانولوشنی است را به صورت زیر پیاده‌سازی می‌کنیم.

```
VAE_CNN(  
    (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
    (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
    (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)  
    (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
    (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
    (bn3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
    (conv4): Conv2d(64, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)  
    (bn4): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
    (fc1): Linear(in_features=10000, out_features=2048, bias=True)  
    (fc_bn1): BatchNorm1d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
    (fc21): Linear(in_features=2048, out_features=2048, bias=True)  
    (fc22): Linear(in_features=2048, out_features=2048, bias=True)  
    (fc3): Linear(in_features=2048, out_features=2048, bias=True)  
    (fc_bn3): BatchNorm1d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
    (fc4): Linear(in_features=2048, out_features=10000, bias=True)  
    (fc_bn4): BatchNorm1d(10000, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
    (conv5): ConvTranspose2d(16, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), output_padding=(1, 1), bias=False)  
    (bn5): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
    (conv6): ConvTranspose2d(64, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
    (bn6): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
    (conv7): ConvTranspose2d(32, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), output_padding=(1, 1), bias=False)  
    (bn7): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
    (conv8): ConvTranspose2d(16, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
    (relu): ReLU()  
)
```

تابع loss در VAE از دو بخش تشکیل می‌شود ، قسمت اول تابع loss ای که به طور معمول استفاده می‌شود که در اینجا MSE است ، قسمت دوم به این منظور است که sample ای که می‌گیریم به یک تابع توزیع نرمال نزدیک باشد. به طور کلی به این منظور از این روش استفاده می‌کنیم که توزیع ای که تولید می‌شود شبیه یک توزیع استاندارد باشد. از مزایای آن می‌توان به این اشاره کرد که در مراحل بعد ، encoder را کنار گذاشته و از یک توزیع نرمال یک بردار  $z$  تولید کنیم و به decoder بدھیم تا خروجی موردنظر تولید شود. تابع loss به صورت زیر می‌باشد:

```

class customLoss(nn.Module):
    def __init__(self):
        super(customLoss, self).__init__()
        self.mse_loss = nn.MSELoss(reduction="sum")

    def forward(self, x_recon, x, mu, logvar):
        loss_MSE = self.mse_loss(x_recon, x)
        loss_KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())

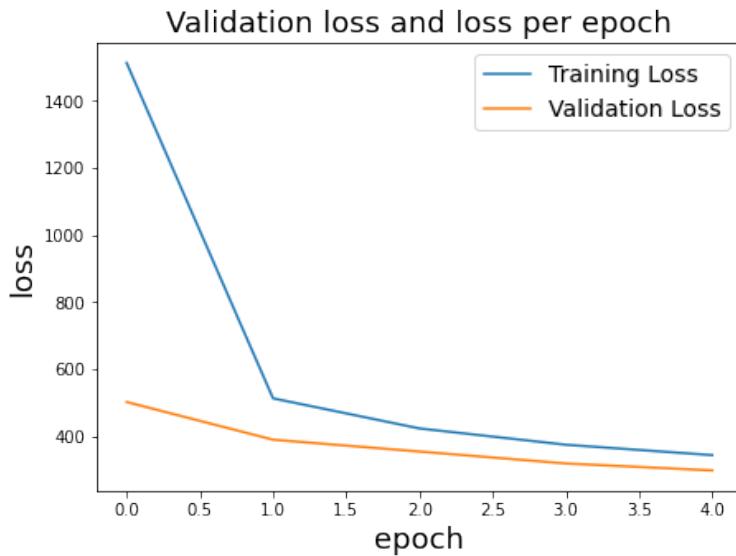
    return loss_MSE + loss_KLD

```

خروجی لایه‌های encoder و decoder ، دو بردار میانگین و واریانس است که به طور مثال در لایه encoder ، برای به دست آوردن  $z$  از توزیعی با این بردارهای میانگین و واریانس ، یک نمونه sample میکنیم . حال برای اینکه خطا را به عقب برگردانیم چون خروجی تابعی از آن میانگین و واریانس نیست ،  $z$  را به صورت زیر تعریف می‌کنیم:

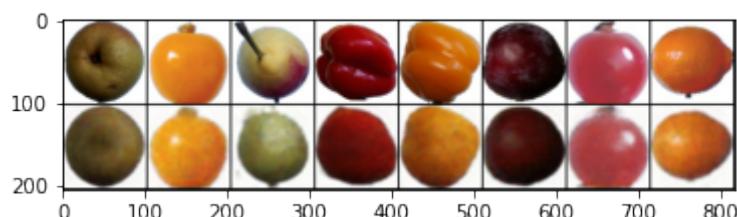
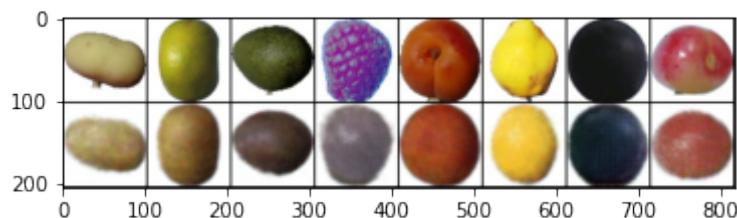
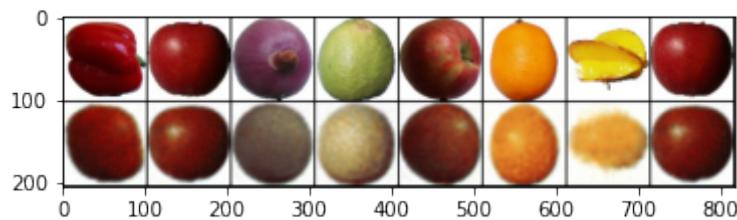
$$z_i = N(0, 1) \times \sigma + \mu$$

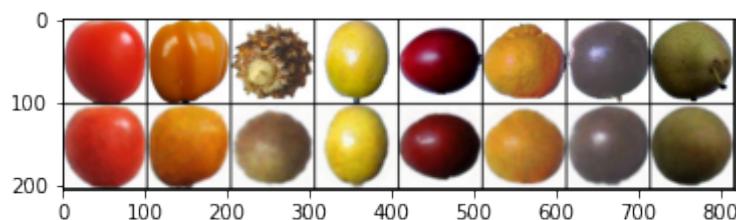
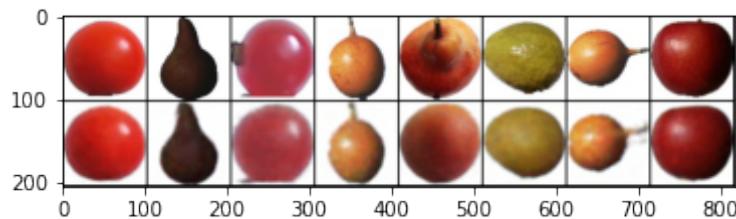
حال  $z$  از یک نرمال می‌آید و همچنین تابعی از  $\mu$  و  $\sigma$  نیز هست.  
نتایج این شبکه بعد از ۵ تا epoch به صورت زیر است (به دلیل حجم بالای دیتاهای و محدودیت زمانی که در colab وجود دارد ، نتوانستم با تعداد epoch بیشتری شبکه را train کنم).



همان طور که در تصویر قبل مشاهده می شود ، میزان loss روند نزولی ای دارد که قابل قبول است.

در زیر تصاویر بازسازی شده را از epoch اول به بعد ، به ترتیب مشاهده می کنید.  
همان طور که واضح است هرچه در تعداد epoch جلوتر برویم ، میزان کیفیت تصاویر بهتر می شود و در طی زمان پیشرفت کرده است. بدیهی است که با تعداد epoch بیشتر در نهایت نتایج بسیار مطلوب تری خواهیم گرفت.





برای تولید میوه جدید که پرسش این سوال است نیز همانند قسمت قبل ، هرچه تعداد epoch بیشتر باشد ، نمونه‌های جدید تولید شده از کیفیت بهتری برخوردار خواهد بود . و برای این کار یک sample به صورت رندوم انتخاب کرده و به بخش decoder می‌دهیم و خروجی را دریافت می‌کنیم. تصاویر میوه‌های تولیدشده به ترتیب epoch به صورت زیر می‌باشد.

