دانشگاه صنعتی امیرکبیر

(پلی تکنیک تهران)

Neural Network Final take home solution

M.B Menhaj

Mohammad Reza Ramezani 400123120

T.A: M.H Amini

1401 Spring

فهرست

سوال 1 – فیلتر تطبیقی

حذف تطبیقی اکو را چگونه می توان با شبکه تک لایه انجام داد؟ کامل فرمول بندی و بحث نمایید . ابتدا روی SLPR کار کنید.

پاسخ:

یکی از کاربرد های مهم فیلتر های تطبیقی حذف کننده نویز، حذف تطبیقی اکو در خطوط تلفنی است؛ با توجه به ساختار این خطوط برای فواصل طولانی، پدیده اکو اجتناب نا پذیر خواهد بود. این پدیده به دلیل عدم انطباق امپدانس در دستگاه های های تقویت کننده[1] در ابتدا و انتهای خطوط تلفنی رخ می‌دهد. دیاگرامی از این شبکه ها در شکل زیر قابل مشاهده است:

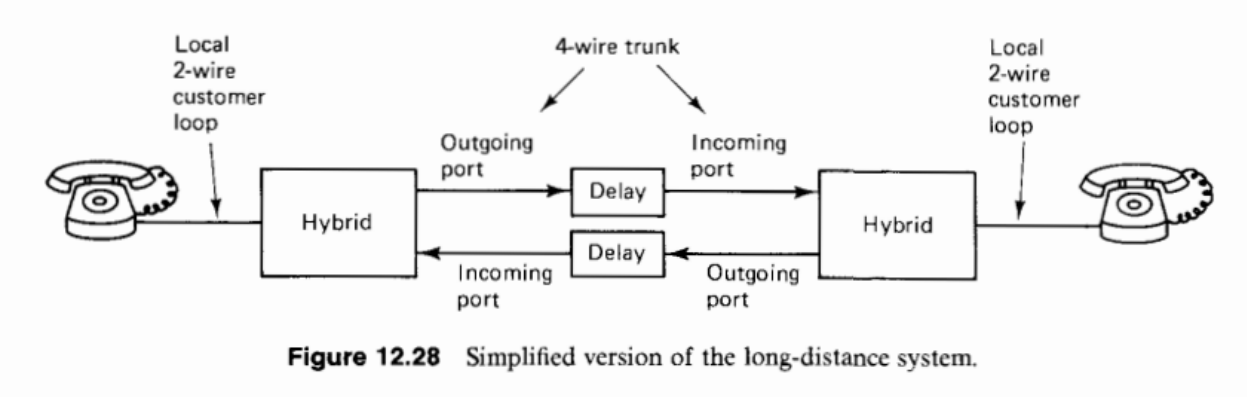


Figure 12.28 Simplified version of the long-distance system.

در واقع در این مسئله سعی داریم اثر اکو(نویز) روی سیگنال ورودی از تلفن را حذف نماییم، که این مهم با **تقلید** کردن اثر دستگاه هیبرید بر روی سیگنال تلفن و در ادامه حذف کردن آن از سیگنال تفلنی بدست آورد.

در شکل زیر دیاگرام موجود این مهم را توصیف می‌کند:

[1] Hybrid Device

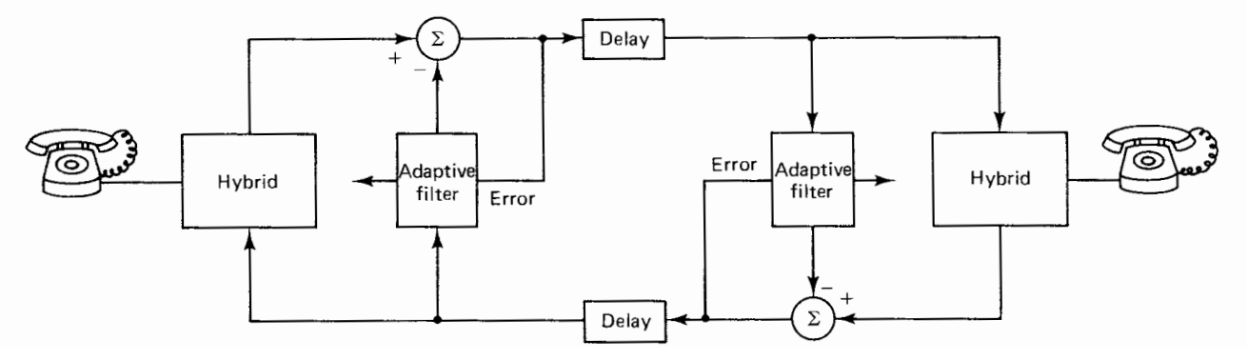


Figure 12.30 Long-distance system with adaptive echo cancellation.

باید توجه داشت که راه حل زیر با فرض خطی بودن سیستم هیبرید در نظر گرفته شده است.

باید توجه داشت که این فیلتر های تطبیقی در دو سوی خط تلفن باید قرار گیرد، اگرچه به طور شهودی به نظر می‌رسد که پارامتر های هر دو فیلتر که در دو سوی خط قرار دارند به طور یکسان است، اما با این حال در کارکرد تجربی نشان داده شده است که لزوما پارامتر های دو فیلتر و عملکرد آن ها یکسان نیست.

شبکه slpr

هدف در آموزش فیلتر تطبیقی، مینیممم کردن سیگنال خطا(خروجی جمع کننده در شکل بالا) است، سیگنال های زیر را برای فیلتر تعریف می‌کنیم:

$$e_k = h_k - a_k$$

$$x_k$$

که h_k معرف خروجی دستگاه هیبرید در لحظه k و a_k خروجی شبکه در لحظه k است. و نیز x_k هم ورودی دستگاه هیبرید و هم شبکه عصبی است.

چالش مهم در استفاده از ساختار slpr در اینجا این است که خروجی نرون های این شبکه به خاطر تابع فعال سازی آستانه دو مقداری، تنها خروجی صفر یا یک را می پذیرد.[۲] حال برای این مسئله که سیگنال خروجی هر مقدار خروجی (با فرض مثبت) می‌تواند داشته باشد، استفاده از یک نرون کار ساز نخواهد بود، در نتیجه برای رفع این محدودیت می توان از چند نرون استفاده کرد، که خروجی را بتوان به صورت عددی باینری از خروجی های هر نرون در شبکه slpr استخراج نمود به طوری که خروجی هر نرون معرف یک بیت باینری در عدد خروجی باشد. در واقع در اینجا نیازمندیم که یک مبدل آنالوگ به دیجیتال استفاده نماییم، رزولوشن سیگنال خروجی از شبکه وابسته به تعداد نرون های(N) شبکه عصبی است.

$$\text{دقت} = \frac{1}{2^{\text{تعداد نرون ها}}}$$

باید توجه داشت که در مرحله محاسبه خطا(e) سیگنال آنالوگ h_k برای مقایسه با خروجی شبکه باید ابتدا به مقدار باینری آن تبدیل شود:

$$\text{رزولوشن} = \frac{h_{max}}{2^N - 1}$$

$$\rightarrow \frac{2^N}{h_{max}} = \frac{h_{converted}}{h_k} \rightarrow h_{converted} = \frac{2^N}{h_{max}} \times h_k$$

[۲] با شرط استفاده از تابع فعال سازی sign

حال با بدست آوردن مقدار باینری $h_{converted}$ می‌توان عدد حاصل را با خروجی شبکه که خود یک عدد باینری است مقایسه کرد و سیگنال خطا را بدست آورد.

حال برای شبکه عصبی فوق داریم:

$$\bar{P} = \begin{bmatrix} x_k \\ x_{k-1} \\ \vdots \\ x_{k-m} \\ 1 \end{bmatrix} : input\ vector$$

در بردار ورودی بالا، به تعداد m نمونه تاخیر یافته ازسیگنال x_k داریم، برای تولید این سیگنال های تاخیر یافته نیاز به m دستگاه تاخیر دهنده داریم به شکل زیر توجه شود: (در شکل زیر به جای y_k باید x_k قرار داد)

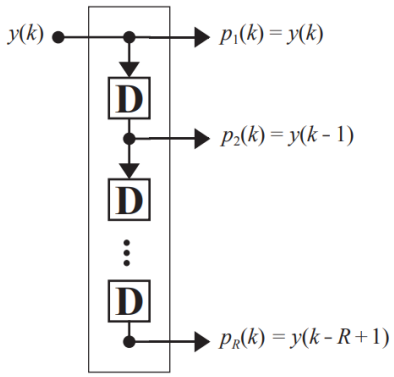


Figure 10.4 Tapped Delay Line

در ادامه:

$$\bar{a} = \begin{bmatrix} a_k^{(1)} \\ a_k^{(2)} \\ \vdots \\ a_k^{(N)} \end{bmatrix} : output\ neurons\ (N)$$

$$\bar{e} = \begin{bmatrix} h_k^{(1)} - a_k^{(1)} \\ h_k^{(2)} - a_k^{(2)} \\ \vdots \\ h_k^{(N)} - a_k^{(N)} \end{bmatrix}$$

و ماتریس وزن ها را می‌توان به صورت زیر تعریف کرد:

$$Q = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1m} & b_1 \\ w_{21} & w_{22} & \dots & w_{2m} & b_2 \\ \vdots & \vdots & \dots & \ddots & \vdots \\ w_{N1} & w_{N2} & \dots & w_{Nm} & b_N \end{bmatrix}$$

که در نهایت می توان رابطه زیر را برای نرون های داخل این شبکه نوشت:

$$sign\left(\begin{bmatrix} w_{11} & w_{12} & \dots & w_{1m} & b_1 \\ w_{21} & w_{22} & \dots & w_{2m} & b_2 \\ \vdots & \vdots & \dots & \ddots & \vdots \\ w_{N1} & w_{N2} & \dots & w_{Nm} & b_N \end{bmatrix} \times \begin{bmatrix} x_k \\ x_{k-1} \\ \vdots \\ x_{k-m} \\ 1 \end{bmatrix}\right) = \begin{bmatrix} a_k^{(1)} \\ a_k^{(2)} \\ \vdots \\ a_k^{(N)} \end{bmatrix}$$

حال به راحتی می‌توان قانون یادگیری را طبق آموخته های فصل 4 کتاب به صورت زیر در آورد:

$$w_{ij}^{new} = w_{ij}^{old} + e_i P_j \, , i = 1,2, \dots, N \, , j = 1,2, \dots, m$$

$$b_i^{new} = b_i^{old} + e_i \, , i = 1,2, \dots, N$$

و یا به صورت خلاصه تر در فرم ماتریسی به صورت زیر خواهد بود:

$$Q^{new} = Q^{old} + eP^T$$

استفاده از شبکه آدلاین

مزیت استفاده از شبکه آدلاین نسبت به شبکه پرسپترون با تابع فعال سازی sign در این است که برای تولید مقدار دلخواه خروجی شبکه نیازی به استفاده از چندین نرون نیست، در حالت قبل برای این که بتوان مقادیر متنوعی از سیگنال را در خروجی شاهر کرد نیازمند آن بودیم که به جای یک نرون چندین نرون را در خروجی قرار داده و سپس با باینری فرض کردن خروجی سیستم بتوان آن را با سیگنال مرجع h_k مقایسه کرد. در حالی که در شبکه آدلاین وجود تنها یک نرون کافی خواهد بود، این موضوع به طبع خود باعث کاهش تعداد پارامتر های سیستم به مقدار $\frac{1}{m}$ شده و در نتیجه هزینه محاسباتی بسیار کمتری را خواهیم داشت.

پارامتر های شبکه آدلاین پیشنهادی هم به صورت زیر خواهد بود:

$$e_k = h_k - a_k$$

$$, a_k = sign\left(\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \\ b \end{bmatrix}^T \times \begin{bmatrix} x_k \\ x_{k-1} \\ \vdots \\ x_{k-m} \\ 1 \end{bmatrix}\right) = sign(Q^T \times P)$$

$$\Rightarrow Q^{new} = Q^{old} + \alpha e P^T$$

که α همان نرخ یادگیری معرفی شده در فصل 7 کتاب شبکه های عصبی است.

نکته انتهایی: تعداد بلوک های نهایی و در نتیجه دسترسی به تعداد مشخصی از تاخیر های سیگنال ورودی می‌تواند تاثیر بزرگی روی همگرایی و پرفورمنس شبکه در حذف اکو داشته باشد، که این تعداد باید توسط طراح و با توجه به فرکانس و تاخیر موجود در سیستم (که باعث اکو شده) و همچنین نرخ نمونه برداری سینال ورودی انتخاب شود، که لازمه این امر داشتن دانش پیشین[۳] در مورد عملکرد و پارامتر های خط تلفن و دستگاه هیبرید است.

[3] Prior Knowledge

مراجع استفاده شده در حل سوال

[1] Hagan, M. T., Demuth, H. B., & Beale, M. (1997). *Neural network design*. PWS Publishing Co..

[2] Widrow, B., & Walach, E. (1983). Adaptive signal processing for adaptive control. *IFAC Proceedings Volumes*, *16*(9), 7-12.

سوال 2 – روند الگوریتم BP

سوال 3 – شناسایی سیستم با استفاده از شبکه عصبی

بخش اول – نمایش ورودی خروجی های سیستم

طبق صورت سوال، تابع تبدیل داده شده برای سیستم به صورت زیر است:

$$G(z) = \frac{y(z^{-1})}{x(z^{-1})} = \frac{0.5z^{-1} + 0.25}{z^{-2} + 0.75z^{-1} + 0.125}$$

با توجه به قطب های مخرج تابع تبدیل،تابع تبدیل فوق دارای سه ناحیه ROC به صورت زیر است:

1- $|z| < 2 \Rightarrow n \leq -1$
2- $2 < |z| < 4 \Rightarrow -\infty < n < +\infty$
3- $|z| > 4 \Rightarrow n \geq 0$

که بمسئله فوق را برای حالت سوم حل می کنیم،حال معکوس تابع تبدیل به صورت زیر در می‌آید:

$$G[n] = (-1)^n \times 2^{n+1}, n \geq 0$$

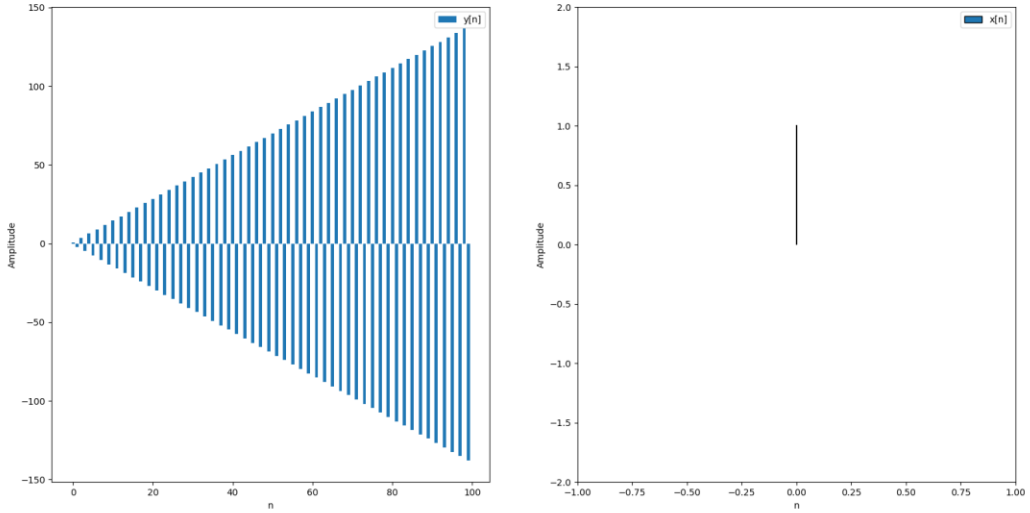
که می دانیم رابطه بدست آمده فوق، در واقع پاسخ ضربه سیستم را به نمایش می‌گذارد، یعنی معادل فوق داریم:

$$y[n] = (-1)^n \times 2^{n+1}, n \geq 0\,, x[n] = \sigma[n]$$

حال خروجی و ورودی سیستم را با کد زیر نمایش می دهیم:

```
1.  #importing stuff
2.  from matplotlib import pyplot as plt
3.  import math
4.
5.  #creating input and out put
6.  N = range(0,100)
7.  y = [(-1)**n*math.log(2**(2*n+1)+0.0001) for n in N]
8.  plt.subplot(1,2,1)
9.  plt.xlabel('n')
10. plt.ylabel('Amplitude')
11. plt.bar(N,y)
12. plt.legend(["y[n]"])
13. plt.subplot(1,2,2)
14. plt.xlabel('n')
15. plt.ylabel('Amplitude')
16. plt.xlim((-1,1))
17. plt.ylim((-2,2))
18. plt.arrow(0,0,0,1)
19. plt.legend(["x[n]"])
20. plt.show()
21.
```

که نتیجه کد فوق را در زیر مشاهده می‌کنیم:



محور عمودی مقدار خروجی $y[n]$ بر حسب لگاریتم [۴] بوده و محور افقی نیز بر حسب step های زمانی از یک تا عدد 100 است.

[۴] لگاریتمی کردن خروجی به دلیل ماهیت ناپایدار سیستم بوده و چون به ازای افزایش های کوچ در زمان، خروجی به اندازه غیر قابل کنترلی افزایش می یابد، باید محور را لگاریتمی کنیم که تغیرات خروجی قابل مشاهده باشد.

بخش دوم – شناسایی سیستم با استفاده از mlp

تابع تبدیل مطرح شده در قسمت قبل را در نظر بگیریم:

$$G(z) = \frac{y(z^{-1})}{x(z^{-1})} = \frac{0.5z^{-1} + 0.25}{z^{-2} + 0.75z^{-1} + 0.125}$$

حال با طرفین – وسطین خروجی را بر حسب ورودی سیستم می نویسیم:

$$z^{-2} * y(z^{-1}) + 0.75z^{-1} * y(z^{-1}) + 0.125 * y(z^{-1}) = 0.5z^{-1} * x(z^{-1}) + 0.25 * x(z^{-1})$$

حال با در نظر گرفتن اپراتور Z جملات بالا را به حوزه زمان منتقل می‌کنیم:

$$y[k] = 2x[k] + 4x[k-1] + -6y[k-1] - 8y[k-2]$$

همانطور که در بالا مشاهده می‌کنیم، مقدار خروجی در لحظه k وابسته به مقدار خروجی در لحظات قبلی و ورودی در لحظه k و لحظات قبلی است، و این موضوع برای هر سیستم خطی گسسته(دارای تبدیل z) صادق خواهد بود.

و این به آن معنی است که می توان خروجی هر سیستم گسسته را به صورت تابعی از خروجی های قبلی، و ورودی های همان لحظه و لحظات قبل در نظر گرفت.

در حالت کلی(چه سیستم خطی باشد و چه نه) می توان خروجی سیستم را به عنوان خروجی جعبه سیاهی که ورودی آن حالت های قبلی سیستم(چه خروجی های قبلی و چه ورودی های همان لحظه و قبلی) در نظر گرفت[1]. و این یعنی می توان رابطه زیر را نوشت:

$$y[k] = f(x[n-k], \dots, x[n-1], x[n], y[n-m], \dots, y[n-1])$$

مدل غیر خطی فوق، به نام مدل NARX[5] شناخته می‌شود.

در حالتی که سیستم غیر خطی فوق دارای یک ورودی و یک خروجی باشد می توان از معماری شبکه زیر برای تخمین خروجی استفاده نمود.

[5] Non Linear Auto Regressive with exogenous inputs

معماری شبکه mlp با دو لایه:

- لایه اول که کاملا متصل بوده و دارای تابع فعال سازی با شرط bounded, one-side saturated باشد. (مثلا تابع سیکموید یا هارد لیمیت)
- در لایه دوم که کاملا متصل است، باید از تابع فعال سازی خطی استفاده نمود.

دلیل انتخاب توابع BOSS در لایه اول، پیدا کردن روابط غیر خطی بین متغییر های بردار ورودی است، در حالیکه در لایه دوم به دلیل اختیار کردن هر مقدار دلخواه برای خروجی سیستم، باید از تابع خطی استفاده کنیم که توانایی داشتن هر مقدار دلخواهی را داشته باشد.

البته این معماری لزوما منحصر نیست و می توان تعداد لایه های غیر خطی مثل لایه اول و همچنین نرون های موجود در این لایه ها را برای دقیق تر شدن تخمین خروجی افزایش داد. در حالی که لایه انتهایی به دلیلی که ذکر شد، حتما باید دارای تابع فعال سازی خطی باشد.

ورودی ها:

همانظور که ذکر شد، ورودی های مدل بالا شامل لحظات قبل خروجی و لحظات قبل و حال ورودی سیستم اند، این که تا چه تعداد نمونه قبلی از هر کدام انتخاب کنیم بستگی به توان پردازشی سیستم ما و فضای ذخیره سازی در دست و در نهایت دقت مورد نظر برای تخمین نهایی است.

به طور مثال برای سیستم فوق که به عنوان یک جعبه سیاه آن را در نظر گرفته ایم، در هر متغییر تا دو لحظه قبل را به عنوان ورودی در نظی می‌گیریم. یعنی با این فرض پیش می رویم:

$$y[k] = f([x[k], x[k-1], x[k-2], y[k-1, y[k-2])$$

حال هدف بدست آوردن تخمینی از تابع f است یعنی:

$$\hat{y}[k] = \phi([x[k], x[k-1], x[k-2], y[k-1, y[k-2])$$

ابتدا یک دسته از ورودی و خروجی های سیستم را به عنوان نمونه داده آموزشی بدست می‌آوریم، با این شرط که حالت اولیه خروجی در لحظات 0 و 1 برابر با صفر است:

```
1.  #getting input and output of model
2.  N = range(0,256) #number of steps or samples in other word
3.  x = [math.sin(n*np.pi/8) for n in N]
4.  y = [0,0]
5.  for n in N[2:]:
6.      nth_output = -6*y[n-1] - 8*y[n-2]+2*x[n]+4*x[n-1]
7.      y.append(nth_output)
```

حال بردار های ورودی و خروجی را بر اساس این که 5 ورودی مجزا داریم (سه ورودی شبکه بر اساس نمونه های بدست آمده برای x که شامل لحظاتk,k-2 , k-1 اند، و دو ورودی شبکه از روی خروجی در لحظه k-1 و لحظه k-2 و در نهایت بردار خروجی شبکه که در لحظه k بدست می آید) :

```
x_k = x[2:] # x at step k
x_kk = x[1:-1] # x at step k-1
x_kkk = x[:-2]   #x at step k-2
y_k = y[2:] #y at step k
y_kk = y[1:-1] #y ate step k-1
y_kkk = y[:-2]   #y at step k-2

```

حال بردار های ورودی و خروجی را می‌سازیم:

```
data = np.array([x_k,x_kk,x_kkk,y_kk,y_kkk]).reshape(256,5)
target = np.array(y_k)[np.newaxis].reshape(256,1)
```

بعد از آماده شدن دیتاست و تارگت، حال باید مدل شبکه عصبی خود را توسعه دهیم:

```
input_layer = Input((5,))
nonlinear_layer = Dense(100,activation='sigmoid',name= 'BOSS_layer')(input_layer)
output_layer  = Dense(1,activation = 'linear',name='linear_layer')(nonlinear_layer)
model = Model(input_layer,output_layer)
model.summary()
```

و حال مدل خود را آموزش می دهیم:

```
model.compile(optimizer ='rmsprop',loss = 'mse')
history = model.fit(data,target,epochs =10,verbose= True)
```

اما نتیجه این شبکه دلخواه ما نیست، خروجی آموزش شبکه در زیر قابل مشاهده است:

```
Epoch 1/10 1/1 [==============================] - 0s 6ms/step - loss: 3074515.2500
Epoch 2/10 1/1 [==============================] - 0s 3ms/step - loss: 3074237.7500
Epoch 3/10 1/1 [==============================] - 0s 5ms/step - loss: 3074057.2500
Epoch 4/10 1/1 [==============================] - 0s 4ms/step - loss: 3073860.7500
Epoch 5/10 1/1 [==============================] - 0s 5ms/step - loss: 3073737.2500
Epoch 6/10 1/1 [==============================] - 0s 8ms/step - loss: 3073629.7500
Epoch 7/10 1/1 [==============================] - 0s 5ms/step - loss: 3073513.2500
Epoch 8/10 1/1 [==============================] - 0s 5ms/step - loss: 3073410.0000
Epoch 9/10 1/1 [==============================] - 0s 7ms/step - loss: 3073321.0000
Epoch 10/10 1/1 [==============================] - 0s 6ms/step - loss: 3073236.0000
```

دلیل این امر به سادگی دادن ورودی سری زمانی(داده های ورودی و تارگت با اهمیت مرحله زمانی) و به خاطر ذات ناپایدار شبکه در دست است، مقادیر خروجی در سیستم واقعی با گذشته زمان به سمت بی نهایت میل می کند و از این رو شبکه قادر به یافتن وزن های مناسب برای این افزایش نیست. یک راه حل خلاقانه برای حل مسئله فوق در زیر برایتخمین سیستم های ناپایدار ارائه می شود:

به جای استفاده از سری های زمانی برای آموزش شبکه، هر کدام از عناصر بردار ورودی را به عنوان متغییر مستقلی از بقیه عناصر در نظر بگیریم و سپس مسئله را حل نماییم.

در واقع همانطور که پیشتر مطرح شد، سعی داریم که تابع f زیر توسط شبکه تقریب بزنیم:

$$f([x[k], x[k-1], x[k-2], y[k-1, y[k-2])$$

در صورت مسئله فوق، هیچ الزامی به وابسته بودن زمانی بین ورودی های فوق وجود ندارد، در واقع می توان مسئله را به صورت زیر در نظر گرفت:

$$f(\alpha[k], \beta[k], \gamma[k], \lambda[k], \rho[k])$$

که متغییر های فوق هیچ وابیستگی به یکدیگر ندارند. [۶]

این بدان معنی است که خلاف حالت پیش که ورودی ها و خروجی آموزشی شبکه یک سری زمانی متصل به یک دیگر بود، در اینجا در وردی ها 5 عدد رندوم تولید کرده و خروجی yk را به ازای این اعداد رندوم کوچک بر اساس دینامیک داده شده در صورت سوال محاسبه می کنیم، به تعداد N بار این پروسه را تکرار می کنیم تا N سمپل برای آموزش شبکه بدست آید. از طرفی چون ورودی های شبکه اعداد رندومی بین 1, -1 اند، خروجی yk نیز ناپایدار نخواهد بود. به کد زیر دقت کنیم:

```
1.  N = range(0,1000)
2.  xk = [uniform(-1,1) for n in N] # x at step k
3.  xkk = [uniform(-1,1) for n in N] # x at step k-1
4.  xkkk = [uniform(-1,1) for n in N] #x at step k-2
5.  ykk = [uniform(-1,1) for n in N] # y at step k-1
6.  ykkk = [uniform(-1,1) for n in N] # y at step k-2
7.  yk = [(-6*ykk[n]-8*ykkk[n]+2*xk[n]+4*xkk[n]) for n in N]
8.  data = np.transpose(np.array([xk,xkk,xkkk,ykk,ykkk]))
9.  target = np.transpose(np.array(yk)[np.newaxis])
```

[۶] البته این روش تنها برای شبکه های تخمینگر narx خاصی است که معماری آن ها series parallel باشد، یعنی ورودی شبکه به خروجی لحظات قبل شبکه (مدل های ری کارنت) وابسته نیست.

نکته مهم: y[k] بدست آمده در این دیتاست بر اساس دینامیک داده شده در صورت سوال محاسبه می شود(رابطه ریکرسیو خروجی y[k] بر حسب ورودی ها)

معماری مدل شبکه عصبی تفاوتی با حالت قبل نداشته و به این صورت پیاده سازی می‌شود:

```
#building model
input_layer = Input((5,))
nonlinear_layer = Dense(20,activation='sigmoid',name= 'BOSS_layer')(input_layer)
output_layer  = Dense(1,activation = 'linear',name='linear_layer')(nonlinear_layer)
model = Model(input_layer,output_layer)
#model.summary()

model.compile(optimizer ='adam',loss = 'mse')
history = model.fit(data,target,epochs =10000,verbose= False)
print(history.history["loss"][-10:])
```

که خروجی مقدار خطای ده epoch آخر به صوررت زیر است:

```
[0.011505394242703915,  0.009561322629451752,  0.005232410039752722,  0.0040434375405311584,
0.002591110300272703,  0.002464599208906293,  0.0030026589520275593,  0.0035797557793557644,
0.006397805642336607, 0.007674115709960461]
```

همانطور که مشاهده می شود، شبکه به خوبی توانسته است خروجی سیستم را مدل کند. حال برای اطمینان از عملکرد شبکه داده تست طراحی کرده و خروجی شبکه را تست می کنیم:

```
test_data = np.random.rand(10,5)
test_target  =  np.transpose(np.array([(2*row[0]+4*row[1]-6*row[3]-8*row[4])  for  row  in
test_data])[np.newaxis])
target_hat  = model(test_data)
print(test_target-target_hat)
```

و خروجی در زیر قابل مشاهده است:

```
tf.Tensor( [[  1.11331940e-02] [-1.04904175e-05] [  1.79958344e-03] [  7.78675079e-04] [-
1.63054466e-03] [  1.17015839e-03] [-6.12020493e-03] [  8.36324692e-03] [  9.03701782e-03] [-
1.43527985e-03]], shape=(10, 1), dtype=float32)

```

همانطور که می‌بینیم، خروجی شبکه برای دیتای تست نیز بسیار عالی عمل کرده است.

بحثی در مورد انتخاب تابع هزینه و تعداد نرون های لایه مخفی:

همانطور که پیش تر مطرح شد، در معماری شبکه فوق تابع فعال سازی لایه آخر از نوع خطی است و دلیل انتخاب آن داشتن خروجی در هر مقدار دلخواه برای خروجی شبکه است. به همین دلیل، انتخاب تابع هزینه متناسب با خروجی شبکه باید باشد، طبق مباحث تدریس شده ما در اینجا چندین انتخاب داشتیم:

1- MSE
2- MAE
3- Binary Cross Entropy
4- Categorial Cross Entropy
5- Sparse Categorial Cross Entropy

که بر اساس تسک در دست باید یکی از این توابع را انتخاب نماییم، همانطور که از صورت مسئله معلوم است، تسک مورد نظر مسئله ما Regression خواهد بود و برای این تسک به طور معمول از دو تابع هزینه اولی (یعنی میانگین مجذور مربعات خطا و میانگین قدر مطلق خطا) استفاده می شود.

در مورد تعداد نرون های لایه نهفته نیز می توان بدین صورت بحث کرد که هر چقدر پیچیدگی سیستم ممورد نظر برای شناسایی بیشتر باشد، باید به تعداد بیشتری نرون در لایه مخفی شبکه اضافه نماییم تا عمل شناسایی بهتر صورت پذیرد، در نهایت انتخاب تعداد نرون های لایه مخفی یک امر مهندسی بوده و بر اساس tradeoff ها باید در نظر گرفته و محاسبه شود.

بخش سوم – بدست آوردن ضرایب تابع تبدیل با استفاده از mlp

همانطور که در طراحی ارائه شده در بخش قبل دیدیم، از یک شبکه mlp برای تقریب خروجی سیستم خطی استفاده نمودیم، البته باید در نظر داشت که در طراحی مورد نظر هیچگونه دانش قبلی در مورد خطی یا غیر خطی بودن سیستم وجود نداشت و سیستم مورد نظر به صورت جعبه سیاه در نظر گرفته شده بود، حال با این فرض که سیستم مورد نظر ما خطی است بخواهیم با پارامتر های بدست آمده از روی شبکه آموزش داده شده mlp ضرایب هر یک از جملات $y[k-1], y[k-2], x[k], x[k-1], x[k-2]$ را بدست آوریم به نظر ممکن پذیر نخواهد بود. به طور مثال به ماتریس وزن های شبکه آموزش داده شده در سوال قبل نگاهی بیاندازیم:

```
1.  from tensorflow.keras.models import load_model
2.  import numpy as np
3.
4.  model = load_model('narx_model')
5.  w = model.get_weights()
6.  print(w[0][0])
```

کد فوق ضرایب 20 نرون لایه مخفی برای عنصر اول بردار ورودی (یعنی x[k]) را می دهد:

```
1.  [-0.09504923  0.11832634  0.07350921  0.08798342  0.09289412  0.07531675
2.    0.06030732 -0.09308887 -0.08538324  0.08704334 -0.08144949 -0.06413831
3.   -0.08046224 -0.10713774  0.10551042 -0.07352626  0.06871989  0.07359492
4.   -0.06465038 -0.06604186]
5.
```

همانطور که مشاهده می کنیم، هیچ اطلاعاتی از ماتریس وزن های ورودی در مورد ضریب x[k] در تابع تبدیل اصلی نمی شود استخراج کرد. و این موضوع به پیچیده بودن شبکه mlp طراحی شده بر می‌گردد، در حالی که با فرض خطی بودن سیستم به راحتی میتوان ضرایب سیستم را با یک شبکه آدلاین بدست آورد که در قسمت بعد بحث میکنیم.

بخش چهارم – بدست آوردن ضرایب تابع تبدیل با شبکه آدلاین

فرض کنیم سیستم خطی گسسته زیر را در اختیار داریم:

$$G(z) = \frac{y(z)}{x(z)} = \frac{a_1 + a_2 z^{-1} + a_3 z^{-2} + \cdots + a_m z^{-m}}{b_1 + b_2 z^{-1} + b_3 z^{-2} + \cdots + b_n z^{-n}}, n > m$$

با طرفین وسطین و بدست آوردن y[k] داریم:

$$y[k] = \frac{1}{b_1}[a_1 x[k] + a_2 x[k-2] + \cdots + a_m x[k-m] - (b_2 y[k-1] + b_3 y[k-2] + \cdots + b_n y[k-n])]$$

همانطور که می‌بینیم خروجی y[k] عبارتی خطی بر حسب ورودی در لحظه k و لحظات قبل + خروجی در لحظات قبل است. در نتیجه تابع f در نظر گرفته شده در قسمت دوم سوال که ذکر شد یک تابع خطی خواهد بود، یعنی:

$$f: linear\ function\ of\ tapped\ delay\ of\ y, x$$

در نتیجه با داشتن این دانش قبلی در مورد سیستم، به جای استفاده از معماری پیشنهاد شده در قسمت دوم سوال که یک معماری با لایه مخفی غیر خطی بود، به سادگی می توان یک شبکه تک لایه خطی تک نرونه در نظر گرفت که وظیفه بدست آوردن خروجی در لحظه k را دارد.

حال به مانند بخش دوم سوال به تولید دیتای مورد نیاز شبکه برای آموزش می پردازیم:

```
1.  #bulding input vector and target
2.  N = range(0,1000)
3.  xk = [uniform(-10,10) for n in N] # x at step k
4.  xkk = [uniform(-10,10) for n in N] # x at step k-1
5.  xkkk = [uniform(-10,10) for n in N] #x at step k-2
6.  ykk = [uniform(-10,10) for n in N] # y at step k-1
7.  ykkk = [uniform(-10,10) for n in N] # y at step k-2
8.  yk = [(-6*ykk[n]-8*ykkk[n]+2*xk[n]+4*xkk[n]) for n in N]
9.  data = np.transpose(np.array([xk,xkk,xkkk,ykk,ykkk]))
10. target = np.transpose(np.array(yk)[np.newaxis])
```

معماری شبکه هم همانطور که ذکر شد یک شبکه تک لایه با یک تک نرون است، تابع loss انتخاب شده نیز به دلیلی که توضیح داده شد همانند بخش دوم سوال است:

```
1.  #building model
```

```
input_layer = Input((5,))
linear_layer = Dense(1,activation='linear',name= 'linear_layer',use_bias= False)(input_layer)
model = Model(input_layer,linear_layer)
model.summary()
model.compile(optimizer ='sgd',loss = 'mse')
history = model.fit(data,target,epochs =100,verbose= True)
print(history.history["loss"][-10:])
```

نکته: در کد فوق مقدار ترم بایاس را به خاطر داشتن دانش قبلی در مورد عدم وجود مقدار ثابت در تابع f به صورت پیش فرض صفر کرده ایم، اگر چه بدون انجام این عمل هم در روند آموزش خود شبکه به مقدار صفر برای ترم بایاس می‌رسید.

خروجی کد فوق را با هم می‌بینیم:

```
Epoch 1/100 32/32 [==============================] - 0s 1ms/step - loss: 180.1732
Epoch 2/100 32/32 [==============================] - 0s 1ms/step - loss: 1.2934e-11
Epoch 3/100 32/32 [==============================] - 0s 1ms/step - loss: 1.2812e-11
Epoch 4/100 32/32 [==============================] - 0s 1ms/step - loss: 1.3143e-11
Epoch 5/100 32/32 [==============================] - 0s 2ms/step - loss: 1.3093e-11
Epoch 6/100 32/32 [==============================] - 0s 2ms/step - loss: 1.5570e-11
Epoch 7/100
...
Epoch 99/100 32/32 [==============================] - 0s 2ms/step - loss: 1.3413e-11
Epoch 100/100 32/32 [==============================] - 0s 2ms/step - loss: 1.3256e-11
```

همانطور که می‌بینیم شبکه به سادگی توانسته است تابع هزینه مورد نظر را صفر کند.

به یاد بیاوریم که طبق معماری داده شده، بردار ورودی به شبکه از قرار زیر بوده است:

$$input\ vector = \begin{bmatrix} x[k] \\ x[k-1] \\ x[k-2] \\ y[k-1] \\ y[k-2] \end{bmatrix}^T$$

در نتیجه المان های ماتری وزن های بدست آمده با ضریب هر کدام از عناصر بردار ورودی در تابع f یک رابطه یک به یک دارند:

$$w = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \end{bmatrix} = \begin{bmatrix} \frac{a_1}{b_1} & \frac{a_2}{b_1} & \frac{a_3}{b_1} & -\frac{b_2}{b_1} & -\frac{b_3}{b_1} \end{bmatrix}^T$$

حال ماتریس وزن ها را از شبکه آموزش داده شده استخراج می‌کنیم:

```
1.  model.get_weights()
2.  [array([[ 2.000000e+00], [ 4.000000e+00], [-9.939633e-08], [-6.000000e+00], [-8.000000e+00]],
    dtype=float32)]
```

حال کافیست برای بدست آوردن ضرایب معادله شرح داده شده در فوق را حل کنیم:

$$\frac{a_1}{b_1} = 2, \frac{a_2}{b_1} = 4, \frac{a_3}{b_1} \cong 0, -\frac{b_2}{b_1} = 6, -\frac{b_3}{b_1} = 8$$

$$\Rightarrow a_1 = 0.25, a_2 0.5, a_3 \cong 0\ , b_1 = 0.125, b_2 = 0.75, b_3 = 1$$

که نتایج بدست آمده دقیقا ضرایب داده شده در تابع تبدیل صورت سوال است.

مراجع استفاده شده در حل این سوال

[1] Billings, S. A., & Fadzil, M. B. (1985). The practical identification of systems with nonlinearities. *IFAC Proceedings Volumes*, *18*(5), 155-160.

سوال 4 – پیاده سازی و تشریح PCA

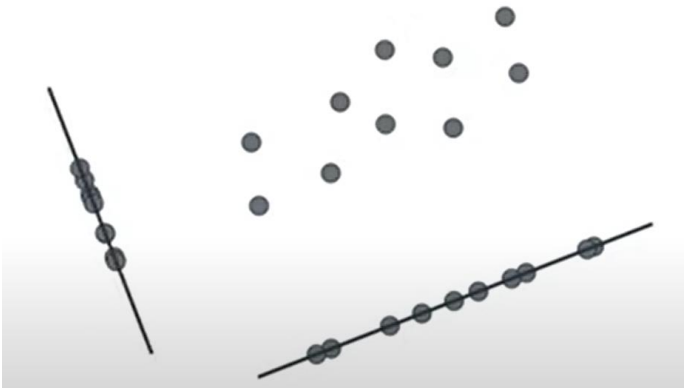
تشریح PCA و فرموله کردن آن

PCA و یا آنلیز مولفه های اصلی، یکی از رایج ترین راه ها برای کاهش ابعاد داده ها است و به طور گسترده و در مسئله پردازش تصویر استفاده می‌شود. کلید اصلی این ابزار در پیدا کردن رابطه ای بین ابعاد دیتای در دست، و سپس فشرده سازی اطلاعات دیتا در ابعاد پایین تر است.

به طور مثال به شکل زیر توجه کنیم:

در شکل بالا، نقاط دیتا با دایره های خاکستری رنگ به نمایش در آمده است، مختصات هر کدام از نقاط بالا را می‌توان به صورت جفت مولفه x, y

نشان داد که نمایشی دو بعدی است. حال در PCA هدف ما این است که با پروجکت کردن این نقاط روی یک خط، از تعداد ابعاد مورد نیاز برای نشان دادن دیتای فوق (که دو بعد بود) بکاهیم. به شکل زیر که دو نمونه خط برای دیتای فوق است توجه کنیم:



از بین دو خط فوق تمایل داریم خطی را انتخاب کنیم که پراکندگی داده های پروجکت شده روی آن بیشترین باشد، در این صورت هر نقطه داده روی خط متمایز شده از دیگری متمایز تر خواهد بود، معیار ما برای انتخاب این خط این است که واریانس داده پروجکت شده روی این خط بیشترین باشد.

حال برای پیدا کردن خط یا خطوطی که پروجکشن دیتا روی آن بیشترین واریانس ممکن را داشته باشد وظیفه ما خواهد بود.

برای محاسبه کردن این خطوط می شود به صورت زیر عمل نمود.

در ابتدا ماتریسی به نام ماتریس کواریانس را تشکیل می‌دهیم:

$$Q = \begin{bmatrix} var(x) & cov(x,y) \\ cov(x,y) & var(y) \end{bmatrix}$$

که x, y مختصات نقاط داده های ما خواهد بود.

مقادیر $cov(x,y)$ و $var(x)$ و $var(y)$ به صورت زیر محاسبه می‌شود:

$$cov(y,x) = \frac{\sum_{n=1}^{N}(\mathrm{x_n} . y_n)}{N}$$

$$var(x) = \frac{\sum_{n=1}^{N}(\mathrm{x_n})}{N}$$

$$var(y) = \frac{\sum_{n=1}^{N}(\mathrm{y_n})}{N}$$

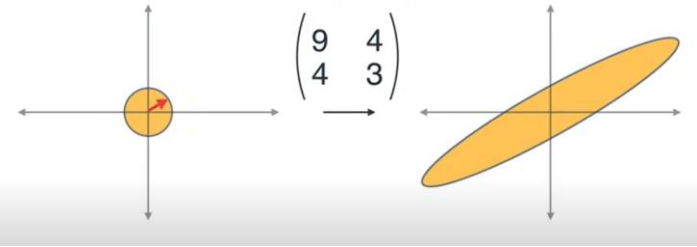
حال اگر Q را به عنوان نگاشتی خطی از مختصات اولیه به یک مختصات جدید در نظر بگیریم، می‌شود آموخته های خود در درس جبر خطی را در اینجا نیز تکرار کنیم:

$$X' = X \times Q$$

که ماتریس X ماتریس داده های ما به صورت زیر است:

$$X = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_N & y_N \end{bmatrix}$$

و ماتریس X' بدست آمده دیتا های نگاشت شده در فضای جدید است. به طور مثال به دایره واحد زیر که تحت نگاشت $\begin{bmatrix} 9 & 4 \\ 4 & 3 \end{bmatrix}$ به یک بیضی تبدیل شده است دقت کنیم:



برای ماتریس فوق می توان مقادیر ویژه و بردار های ویژه را به صورت زیر تعریف کرد:

$$eigenvalues = \det(\lambda I - Q) = 0$$

$$eigenvectors = P = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_i \end{bmatrix}^T , Qv_i = \lambda_i$$

همانطور که مشاهده کردیم، تحت تبدیل فوق می توان دیتای اولیه را به مختصات جدیدی منتقل کرد که در اینجا بردار های ویژه نقشی اساسی در تعیین کردن شکل این انتقال دارند. به سادگی و با شهود از روی ماتریس Q می توان نشان داد که پراکندگی (واریانس داده ها) در راستای بردار ویژه ای که متناظر با بیشترین مقدار ویژه باشد ، ماکزیمم است. در نتیجه اگر بردار ویژه ای را که متناظر با بیتشرین مقدار ویژه است را به عنوان خطی که پروجکش دیتا روی آن انجام می‌شود انتخاب کنیم، خطی را که پروجکشن دیتا بیشترین واریانس را دارد پیدا کرده ایم.

حال پس از پروجکت کردن دیتا روی برداری که بیشترین مقدار ویژه را داریم نقاط جدیدی بدست آوریم، ما ابعاد دیتای در دست را کاهش داده ایم.

البته در مثال فوق چون تنها دو بعد داشتیم، در نتیجه دو بردار ویژه نیست بدست می‌آمد، حال اگر تعداد ابعاد(یا همان فیچر های دیتا) زیاد باشد، برای مسئله کاهش ابعاد می توان به تعداد m مقدار ویژه بزرگ را انتخاب کرد، سپس با ضرب دیتای اولیه در بردار های ویژه متناظر با این m مقدار ویژه، تعداد ابعاد داده ورودی را کاهش داد، یعنی:

$$T_{N\times m} = X_{N\times f} \times P_{f\times m}$$

که در رابطه فوق، ماتریس T نمایش جدید داده ها ، n تعداد نمونه های داده، f تعداد فیچر های داده اصلی(تعداد ستون های ماتریس X) و m تعداد بردار های ویژه انتخابی (m برداری که متناظر با m مقدار ویژه بزرگ هستند) و در نهایت P نیز از کنار هم گذاشتن آن m بردار ویژه انتخاب شده بدست می‌آید.

نکته بسیار مهم در ساختار بردار های ویژه بدست آمده، عمود بودن دو به دوی این بردار ها بر هم است، که این خاصیت از متقارن بودن ماتریس کواریانس پیشنهاد شده است.

پیشنهاد شبکه عصبی برای محاسبه PC ها

روش اول

در قسمت قبل شاهد بودیم که با ضرب ماتریس داده های ورودی در بردار های ویژه بدست آمده، یک نگاشت خطی از داده های ورودی به نمایشی جدید است، یعنی داشتیم:

$$T_{N\times m} = X_{N\times f} \times P_{f\times m}$$

که در واقع این نگاشت جدید سعی داشت با حذف کردن تعدادی از ابعاد، از تعداد f فیچر به تعدادm فیچر برسد با این شرط که $m < f$.

معادله فوق را می‌توان به صورت زیر باز نویسی کرد:

$$y_{N\times m} = X_{N\times f} \times P_{f\times m}$$

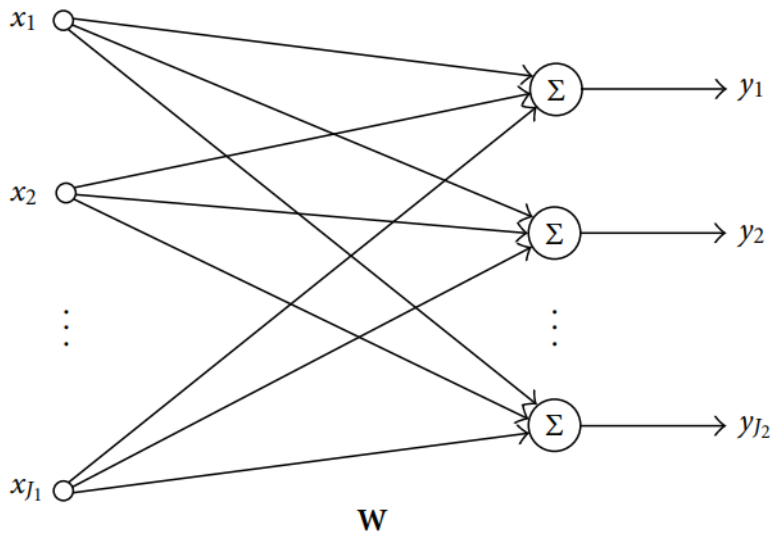
که به نمایش جدید بدست آمده y به اصطلاح **مولفه های اصلی**(PCs) می‌گوییم.

یک شبکه پیشنهادی برای بدست آوردن ماتریس P می‌تواند انتخاب زیر باشد:

شبکه ای تک لایه با توابع فعال سازی خطی و با مشخصات زیر:

- تعداد m تعداد نرون که به تمامی مولفه های بردار ورودی متصل است(fully connected)

شکل زیر نمایش این شبکه است:



در واقع شبکه پیشنهادی نوع خاصی ا برای لایه اول این شبکه داریم:

$$y_{N\times m} = X_{N\times f} \times W_{f\times m}$$

که معرف یک شبکه خطی ساده است. هر سطر از ماتریس W معرف وزن های نرون i ام در لایه اول است.

و در انتها خطای بازسازی[۷] به صورت زیر تعریف می‌شود:

$$e = X - \hat{X}$$

که ماتریس $\hat{X}$ از رابطه زیر بدست می‌آید[۸]:

$$\hat{X} = y \times W^T$$

و در نهایت می توان قانون آپدیت وزن ها را بر اساس آموخته های فصل هفت کتاب شبکه عصبی و بر اساس قانون یادگیری هب[۹] به صورت زیر نوشت[۱۰]:

$$W_{new} = W_{old} + \alpha e^T \times y$$

---

[7] Reconstruction error

[8] دلیل این که ماتریس W در محاسبه y ها و در محاسبه $\hat{X}$ ها یکسان است به orthogonal بودن ماتریس W بر می‌گردد.

[9] Hebbian rule

[10] اثبات این موضوع از حوصله این پاسخنامه خارج بوده و در مقاله مرجعی که قانون یادگیری بر اساس آن نوشته شده قابل پیگیری است.

که روال فوق همانطور که اشاره شد حالت خاصی از همان شبکه آدلاین فصل هفت است. (مرجع پیاده سازی این مقاله بوده است [3])

در کد زیر پیاده سازی ساده ای از این شبکه را مشاهده می‌کنیم:

```
1.  import numpy as np
2.  from matplotlib import pyplot as plt
3.
4.  class PCA():
5.      def __init__(self,input_matrix,input_feature_size,PC_size):
6.          self.w = np.ones((input_feature_size,PC_size))
7.          self.x = input_matrix
8.          self.y = np.matmul(self.x,self.w)
9.          self.xbar = np.matmul(self.y,np.transpose(self.w))
10.     def update(self,alpha):
11.         self.e = self.x - self.xbar
12.         #learning rule based on paper
13.         self.w = self.w + alpha*np.matmul(np.transpose(self.e),self.y)
14.         self.y = np.matmul(self.x,self.w)
15.         self.xbar = np.matmul(self.y,np.transpose(self.w))
16.     def get_pcs(self):
17.         return self.y
18.
```

روش دوم

به طور کلی الگوریتم های موجود برای محاسبه PCA مبتنی بر شبکه های عصبی به دو دسته کلی مسئله بهینه سازی تقسیم می‌شود:

1- مینمم سازی امید ریاضی خطای بازسازی که به صورت زیر نوشته می‌شود:

$$J = E((x - \hat{x})^2) = E(e^2) = E(x - yW^T)$$

که ثابت می‌شود که مسئله مینمم سازی فوق به W متعامد با بیشترین واریانس منجر می‌شود که در واقع همان بردار های ویژه ماتریس کواریانس هستند.[4]

راه دیگر ماکزیمم کردن واریانس مولفه های اصلی(PCs) های بدست آمده از ضرب ماتریس ورودی در ماتریس وزن ها است، در حالی که ماتریس وزن ها را بتوان متعامد نگه داشت، یعنی:

$$J = E(y^T y), subject\ to\ W^T W = I$$

حال می توان شبکه هایی با ساختار های متفاوت برای حصول این نتیجه طراحی کرد، اما نشان داده شده است که شبکه های با توابع فعالسازی خطی روند همگرایی طولانی تری از نمونه های غیر خطی دارند.[5]

یک شبکه معرفی شده شبه PCA می تواند شبکه های auto encoder با توابع فعال سازی خطی یا غیر خطی و با دو لایه که در لایه نخست ابتدا مولفه ها استخراج شده و سپس در لایه دوم تخمینی از ورودی (همانند مثال حل شده بالا) محاسبه می‌گردد. ایده خلاقانه این روش این است که ابتدا تمام ورودی ها را به تنها یک پارامتر کد کرده(مثلا y_1) و سپس با فیکس کردن ضرایب بدست آمده برای این پارامتر، یک y جدید در نظر گرفته و دوباره به آموزش شبکه می‌پردازد. در مقاله بررسی شده برای این نوع شبکه گفته شده که این نوع آموزش دادن باعث شده که پارامتر های y به ترتیب اهمیت به کد کردن دیتای ورودی بپردازند و در نتیجه از با فیکس کردن هر پارمتر، پارمتر بعدی نسبت به پارامتر قبلی مستقل خطی خواهد بود.[6]

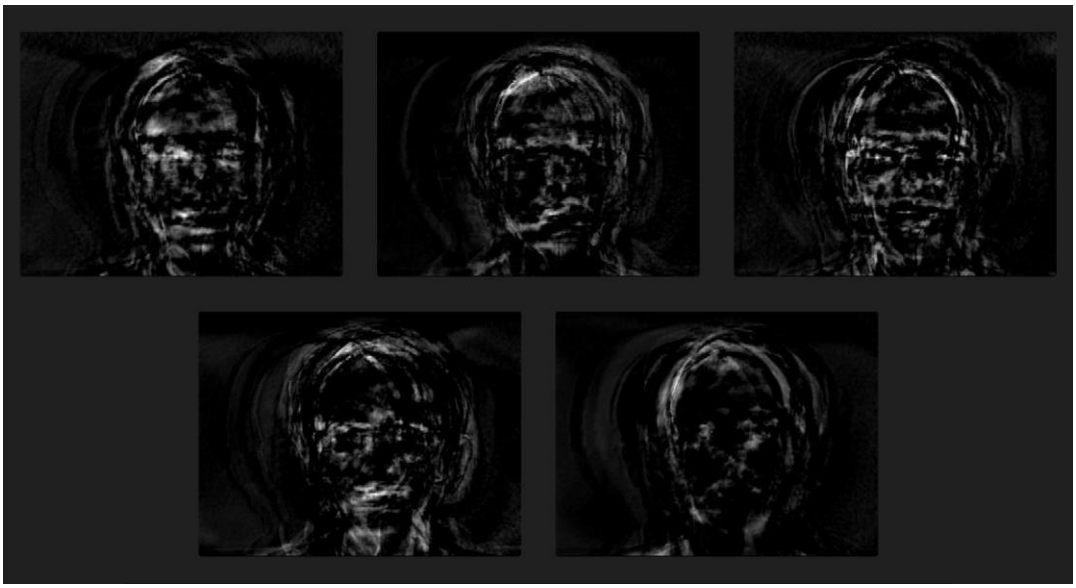
نکته مهم: در شبکه معرفی شده فوق، نیازی به خطی بودن توابع فعال سازی نرون ها وجود ندارد و همین موضوع باعث قدرتمند تر شدن شبکه در تخمین بهتر PC ها می‌گردد.

حال به پیاده سازی این شبکه می پردازیم:

```
1.  #importing stuff
2.  import os
3.  from tensorflow import keras
4.  from tensorflow.keras.layers import Dense,Conv2D,Input
```

```
import numpy as np
from matplotlib import pyplot as plt
from PIL import Image
from tensorflow.keras.models import Model

# preparing dataset
dataset_folder = 'yalefaces/'
def face(folder):
    faces = []
    for file_name in os.listdir(folder):
        face = Image.open(folder+file_name)
        face = np.asarray(face)
        faces.append(face)
    return np.array(faces)
faces = face(dataset_folder)
vetorized_faces = faces.reshape(166,77760)/255
f = vetorized_faces[0].reshape(243,320)*255
f = Image.fromarray(f)
f.show()

#building model
N = vetorized_faces.shape[0] #Number of data Sample point
F = vetorized_faces.shape[1] #arbitrary input size feature
M = 400 #arbitrary PC feature size
reconstructed_data = vetorized_faces
data = vetorized_faces
layer = Input(shape =(F,),name='Input_layer')
PC_layer = Dense(1,activation= 'linear',use_bias=False,name='PC_layer')(layer)
output_layer = Dense(F,activation= 'linear',use_bias=False,name='output_layer')(PC_layer)
model = Model(layer,output_layer)
model.summary()
model.compile(optimizer = keras.optimizers.Adam(learning_rate =0.1),loss = "mae")
#extracting PCs
PCs =[]
vectors = []
for i in range(0,M):
    model.fit(data,reconstructed_data,batch_size =int(N/20),epochs=30,verbose =False)
    PC_Model = Model(model.input,model.get_layer('PC_layer').output)
    vectors.append(PC_Model.get_weights())
    #get Pc out of the model an store it in PCs array for visualization purpose
    PCs.append(np.transpose(PC_Model(data)))
    ## to cancel out the effect of ith Pc in reconstructed data for i+1th step ,
    #  we simply subtract each step reconstructed data from original one
    reconstructed_data = reconstructed_data - np.array(model(data))
PCs = np.array(PCs)
#saving PCs
vectors = np.array(vectors)


```

نمونه ای از principal components های بدست آمده از دیتاست چهره دانشگاه yale [7] را در زیر مشاهده می کنیم:



حال می‌توان با PCs بدست آمده و ضرب آن در هر عکس ورودی، عمل کاهش ابعاد (در این مثال از 77760 به 400) را انجام داد.

لازم به ذکر است که در شبیه سازی فوق، تنها از توابع فعال سازی خطی برای محاسبه مولفه های اصلی استفاده شده، در حالی که مقید به استفاده از توابع خطی نیستیم و می توان از توابع غیر خطی(که دقت بهتری در تخمین خروجی دارند) سود جست.

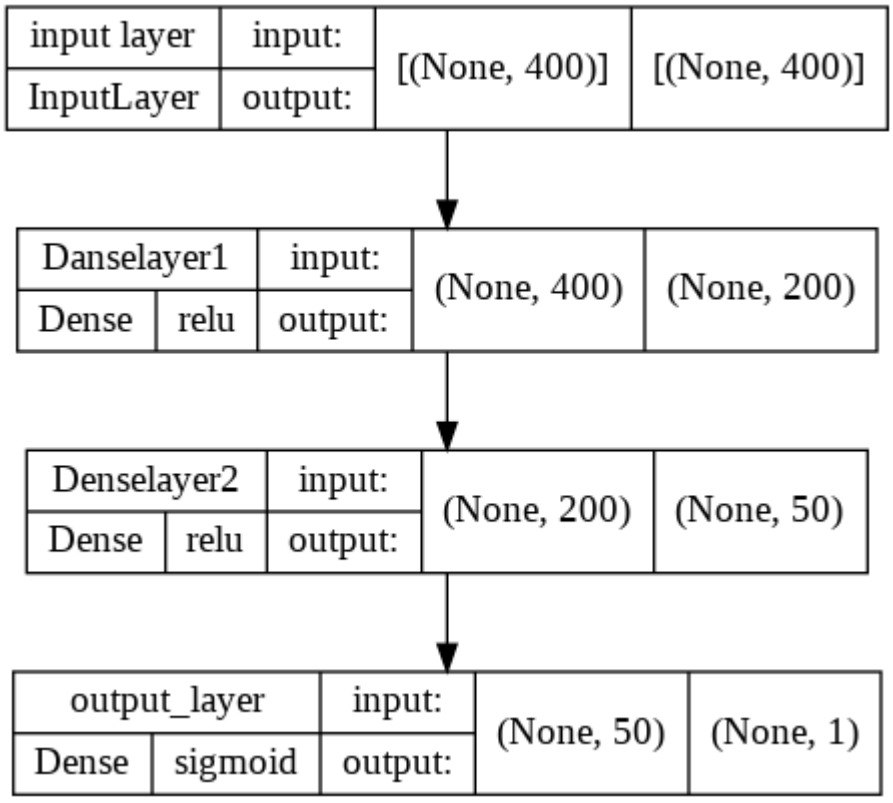
تشخصیص چهره به کمک شبکه عصبی معرفی شده در قسمت قبل

همانطور که در قسمت قبل مشاهده کردیم، با ضرب ماتریس PC ها در در هر تصویر ورودی می توان عمل کاهش ابعاد و محاسبه کردن 400 مولفه اصلی چهره را انجام داد، در این قسمت سعی داریم که به کمک ماتریس ضرایب بدست آمده، دو چهره را از یک دیگر تفکیک کنیم:

$$y = X \times W$$

و سپس از این نمایش کاهش داده شده، مسئله تشخیص چهره دو نفر را حل میکنیم(یعنی ورودی های شبکه ما همین ماتریس با ابعاد کاهش یافته است)

برای مسئله دسته بندی و تشخیص چهره دو نفر از یک دیگر معماری شبکه عصبی زیر را انتخاب کرده ایم:



که در لایه آخر تمام اطلاعات در یک نرون که دو مقدار 0 و 1 قرار می گیرد ، کد می‌شود. که خروجی 0 به معنای شخص A بوده و خروجی 1 به معنای شخص B است.

در شبکه فوق از سه لایه استفاده شده ، همانطور که در شکل بالا می بینیم توابع فعال سازی لایه ها و خروجی آن ها به شرح زیر اند:

- لایه اول: 200 نرون، با تابع فعال سازی relu
- لایه دوم: 50 نرون، با تابع فعال سازی relu
- لایه سوم و خروجی: 1 نرون، با تابع فعال سازی sigmoid

و با توجه به تابع فعال سازی انتخاب شده برای لایه خروجی که sigmoid است، از loss باینری کراس آنتروپی نیز استفاده شده.

حال به عنوان نمونه دو عکس از دیتاست را نشان می دهیم:

شخص اول (با خروجی 0) در سمت چپ و شخص دوم (با خروجی مرتبط 1) در سمت راست تصویر قابل مشاهده است.

کد زده شده برای این قسمت نیز به صورت زیر در خواهد آمد:

```
1.  #importing stuff
2.  import os
3.  from tensorflow import keras
4.  from tensorflow.keras.layers import Dense,Conv2D,Input
5.  import numpy as np
6.  from matplotlib import pyplot as plt
7.  from PIL import Image
8.  from tensorflow.keras.models import Model
9.
10. # preparing dataset
11. dataset_folder = 'yalefaces/'
12. def face(folder):
13.     faces = []
14.     target = []
15.     for file_name in os.listdir(folder):
```

```
        if file_name.split('.')[0] == 'subject01':
            face = Image.open(folder+file_name)
            face = np.asarray(face)
            faces.append(face)
            target.append(0)
        elif file_name.split('.')[0] == 'subject02':
            face = Image.open(folder+file_name)
            face = np.asarray(face)
            faces.append(face)
            target.append(1)
    return np.array(faces),np.array(target)
faces,target = face(dataset_folder)

#loading vectors
vectors = np.load('face_PCs.npy')

#reshaping PCA vectors and Dataset to vectors(also rnormalizing)
vectors = vectors.reshape(77760,400)
faces = faces.reshape(23,77760)/255

#reducing dataset features with Principal component analysis
Faces_reduces_dimension = np.matmul(faces,vectors)

#building model
in_layer = Input(shape=(400))
layer = Dense(200,activation = 'relu')(in_layer)
layer = Dense(50,activation = 'relu')(layer)
output_layer = Dense(1,activation = 'sigmoid')(layer)
model = Model(in_layer,output_layer)
model.compile(optimizer=keras.optimizers.Adam(learning_rate = 0.1e-3),loss = 'bce')
keras.utils.plot_model(model,show_shapes = True,show_layer_names = True)

#fitting model to the dataset
model.fit(Faces_reduces_dimension,target,epochs = 100,validation_split = 0.2,shuffle = True)

#testing model classifier
out = ['subject0','subject1']
x = out[int(np.heaviside(model(np.array([Faces_reduces_dimension[0]]))-0.5,1))]
print(x)
```

مراجع استفاده شده در حل سوال

[1] https://www.youtube.com/watch?v=g-Hb26agBFg

[2] https://www.youtube.com/watch?v=5v4CozbY1_0

[3] Qiu, J., Wang, H., Lu, J., Zhang, B., & Du, K. L. (2012). Neural network implementations for PCA and its extensions. *International Scholarly Research Notices*, *2012*.

[4] Xu, L. (1993). Least mean square error reconstruction principle for self-organizing neural-nets. *Neural networks*, *6*(5), 627-648.

[5] Miao, Y., & Hua, Y. (1998). Fast subspace tracking and neural network learning by a novel information criterion. *IEEE Transactions on Signal Processing*, *46*(7), 1967-1979.

[6] Ladjal, S., Newson, A., & Pham, C. H. (2019). A PCA-like autoencoder. *arXiv preprint arXiv:1904.01277*.

[7] *Yale Face Database B*

سوال 5 – خوشه بندی و فشرده سازی با کمک شبکه های عصبی

بخش اول – معرفی شبکه های مناسب برای عمل خوشه بندی

عمل خوشه بندی یا Clustering یکی از وظایف اصلی سیستم های یادگیر است، عمل خوشه بندی بدان معناست که در فضای پارامتر های ورودی، با کاهش ابعاد دیتای ورودی و عمل فشرده سازی، سمپل هایی از دیتا که در فضای پارامتر های ورودی به یک دیگر نزدیک اند را کشف کرده و سپس هر داده جدید را به یکی از این دسته های کشف شده نسبت داد، این عمل به صورت بدون ناظر صورت گرفته و معیار ما نزدیک بودن داده های ورودی است.

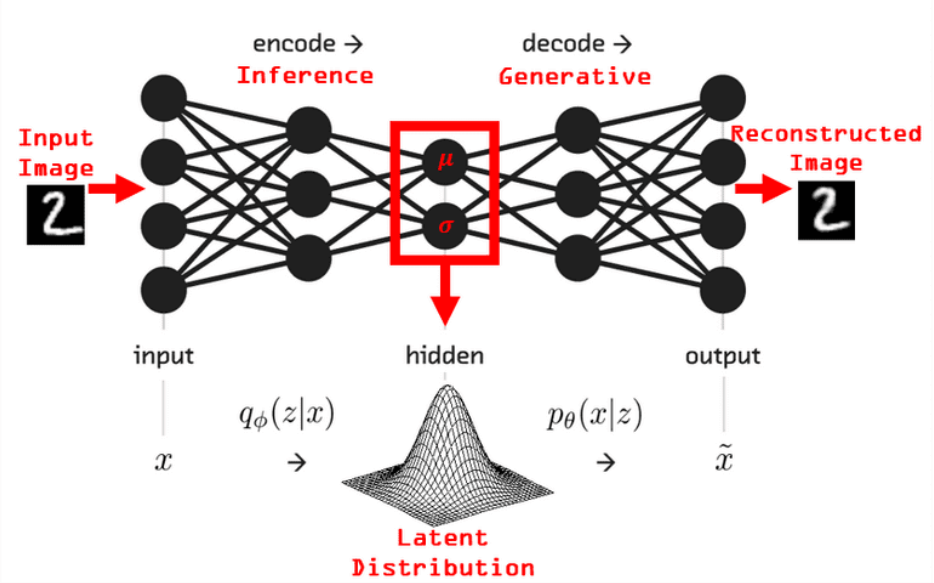
به طور کلی چندین شبکه برای عمل خوشه بندی مورد استفاده قرار می گیرد:

- شبکه های Autoencoder
- شبکه های Self-Organizing Map
- شبکه های LVQ

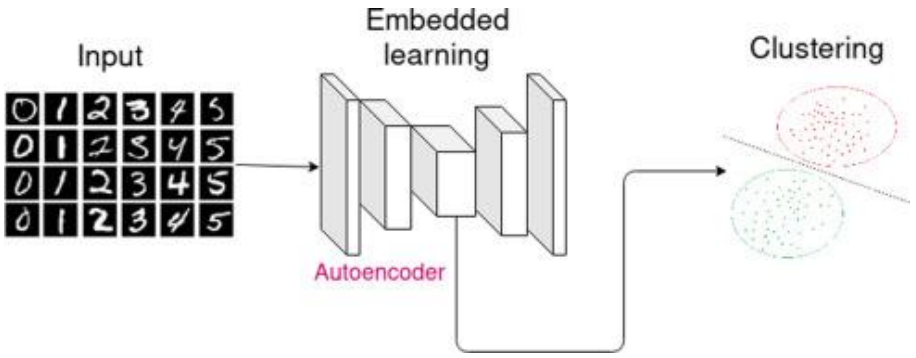
که در اینجا تصمیم داریم از شبکه اولی برای عمل خوشه بندی استفاده نماییم.

شبکه های Auto Encoder به همراه الگوریتم Kmean

معماری این شبکه ها بدین صورت است که ابتدا بردار ورودی را دریافت کرده، و سپس طی چند مرحله فشرده سازی و کاهش ابعاد، به یک نمایش کوچک تر دیتاست ورودی دست می‌یابند، سپس با افزایش ابعاد نرون های میانی، سعی می نمایند که بردار ورودی را دوباره بسازند. مزیت این شبکه ها کد کردن اطلاعات ورودی در لایه های میانی است که ابعاد بسیار کمتری از دیتای ورودی دارند، یکی از کاربرد های این شبکه ها این است که می توانند پس از آموزش، از قسمت انکدر خود برای فشرده سازی و در نتیجه خوشه بندی دیتا استفاده کنند. معماری شبکه فوق را در زیر می بینیم:



همانطور که اشاره شد، با آموزش شبکه فوق و سپس جدا کردن قسمت دیکدر می توان به عمل فشرده سازی و پیدا کردن نمایشی کوچکتر از دیتای اولیه پرداخت:



در این روش، ابتدا نمایش کوچک تر دیتا توسط شبکه auto encoder بدست می آید و سپس با الگوریتم های کلاسترینگ کلاسیک (مانند Kmean) دیتای بدست آمده به خوشه های متفاوت تقسیم می شود.

واضح است که از همان ابتدا نیز می توان از الگوریتم kmean برای خوشه بندی دیتا استفاده نمود، اما مزیت استفاده از ترکیب AE و kmean در این است که الگوریتم kmean برای داده هایی با تعداد فیچر های زیاد(مثل توصیر و صوت) بسیار کند عمل می‌کند، در اینجا استفاده از دانش پیشین کمک می کند که به جای استفاده از دیتای ورودی با ابعاد بالا، از نمایشی کوچکتر با ابعاد پایین استفاده کنیم، چون می دانیم که اطلاعات نهفته در دیتای اولیه قابلیت کاهش ابعاد را به وضوح خواهد داشت و نیازی نیست که برای عمل خوشه بندی مثلا سگ از گربه از

تمام پیکسل های تصویر استفاده کرد، بلکه فیچر هایی مثل شکل پوزه، شکل گوش و چشم برای جدا کردن این دو طبقه کافی اند، در واقع در اینجا با اتو انکدر این فیچر های با ابعاد پایین تر را کشف کرده و سپس به عمل خوشه بندی می‌پردازیم.

روش فوق در مقالات متعدد چه به صورت منفصل و چه متصل پیاده سازی شده است، منظور از منفصل بودن این است که مثل الگوریتم فوق، ابتدا یک شبکه AE فیچر های با ابعاد پایین تر را استخراج کرده و **سپس** در مرحله ای جدا به عمل خوشه بندی توسط kmean پرداخته شده است، در حالی که می توان عمل خوشه بندی و استخراج فیچر های با ابعاد پایین را تنها توسط یک معماری یک پارچه نیز انجام داد(که در این حل برای جلوگیزی از پیچدگی از روش های منفصل استفاده شده)

بخش دوم – پیاده سازی

همانطور که اشاره شد، معماری پیشنهاد شده برای خوشه بندی تصاویر یک معماری شبکه auto encoder به همراه یک بخش خوشه بندی که از الگوریتم k mean استفاده می‌کند، می‌باشد.

دیتاست انتخاب شده برای این بخش قسمتی از دیتاست Fruit 360 است که دو میوه بلوبری و موز را به عنوان دیتای اولیه انتخاب کرده ایم:

نمونه ای از دیتای ورودی:

حال در کد زیر، پیش پردازش و لود کردن دیتا را انجام می‌دهیم:

```
#dataset loading and pre processing data
def data_loader(dataset_path):
    dataset_color = []
    #shuffling pictures
    dir = listdir(dataset_path)
    for fruit_type in dir:
        file = listdir(dataset_path+'/'+fruit_type)
        #random.shuffle(file)
        for filename in file:
            #print(dataset_path+'/'+fruit_type+'/'+filename)
            fruit = Image.open(dataset_path+'/'+fruit_type+'/'+filename)
            fruit_color  = np.asarray(fruit)
            dataset_color.append(fruit_color)
    #return the dataset
    return np.array(dataset_color)/255
data = data_loader('dataset')
```

حال مدل اتو انکدر مورد نیاز را طراحی می‌کنیم:

```
#building model
input_layer = Input((100,100,3))
layer = Conv2D(50,(6,6),activation = 'tanh',padding ='same')(input_layer)
layer = MaxPooling2D(pool_size = (4,4),strides =(4,4))(layer)
layer = Conv2D(50,(5,5),activation = 'tanh',padding= 'same')(layer)
layer = MaxPooling2D(pool_size = (4,4),strides =(6,6))(layer)
layer = Conv2D(1,(3,3),activation = 'tanh')(layer)
encoder_layer = Conv2D(1,(1,2),activation = 'elu',name = 'encoder_end')(layer)
layer = UpSampling2D((5,5))(encoder_layer)
layer = Conv2D(50,(3,3),activation = 'tanh',padding = 'same')(layer)
layer = UpSampling2D((5,5))(layer)
layer = Conv2D(100,(5,5),activation = 'tanh',padding = 'same')(layer)
layer = UpSampling2D((2,4))(layer)
layer = Conv2D(3,(6,6),activation = 'sigmoid',padding = 'same')(layer)
model = Model(input_layer,layer)
#model.summary()
model.compile(optimizer = 'sgd',loss = 'mae')
model.fit(data,data,batch_size = 400,epochs = 50,verbose = False)
```

مدل شبکه عصبی فوق، ابتدا تصویر رنگی ورودی با ابعاد 100 در 100 را با لایه های کانولوشنی و Mazpooling به دو پارامتر رسانده، و سپس با افزایش ابعاد دوباره ابعاد تصویر اولیه را می‌سازد.

در زیر خلاصه ای از شبکه فوق را مشاهده می‌کنیم:

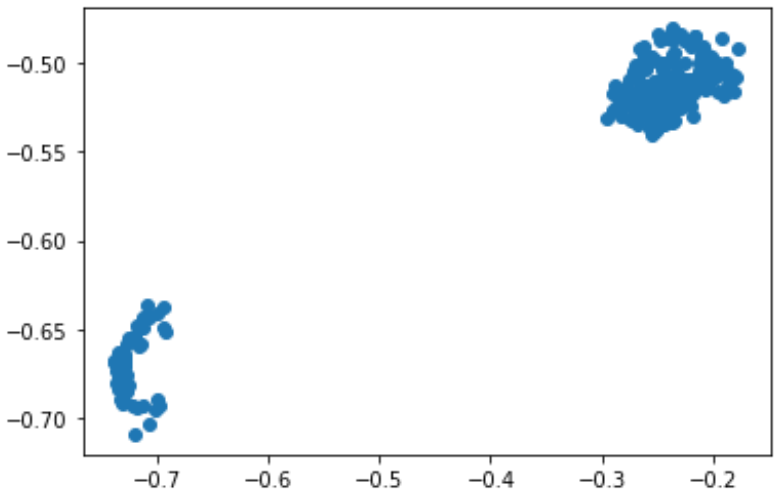
```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 100, 100, 3)]     0

 conv2d_6 (Conv2D)           (None, 100, 100, 50)      5450

 max_pooling2d_2 (MaxPooling  (None, 25, 25, 50)       0
 2D)
```

```
11.
12.  conv2d_7 (Conv2D)           (None, 25, 25, 50)        62550
13.
14.  max_pooling2d_3 (MaxPooling  (None, 4, 4, 50)         0
15.  2D)
16.
17.  conv2d_8 (Conv2D)           (None, 2, 2, 1)           451
18.
19.  encoder_end (Conv2D)        (None, 2, 1, 1)           3
20.
21.  up_sampling2d_3 (UpSampling  (None, 10, 5, 1)         0
22.  2D)
23.
24.  conv2d_9 (Conv2D)           (None, 10, 5, 50)         500
25.
26.  up_sampling2d_4 (UpSampling  (None, 50, 25, 50)       0
27.  2D)
28.
29.  conv2d_10 (Conv2D)          (None, 50, 25, 100)       125100
30.
31.  up_sampling2d_5 (UpSampling  (None, 100, 100, 100)    0
32.  2D)
33.
34.  conv2d_11 (Conv2D)          (None, 100, 100, 3)       10803
35.
36. =================================================================
37. Total params: 204,857
38. Trainable params: 204,857
39. Non-trainable params: 0
```

در ادامه، پس از آموزش شبکه به نمایش خروجی لایه پنهان bottle neck می پردازیم:

```
1.  encoder =Model(model.input,model.get_layer("encoder_end").output)
2.  encoded = encoder.predict(data)
3.  encoded = encoded[:,:,0,0]
4.  x = encoded[:,0]
5.  y = encoded[:,1]
6.  plt.scatter(x,y)
```

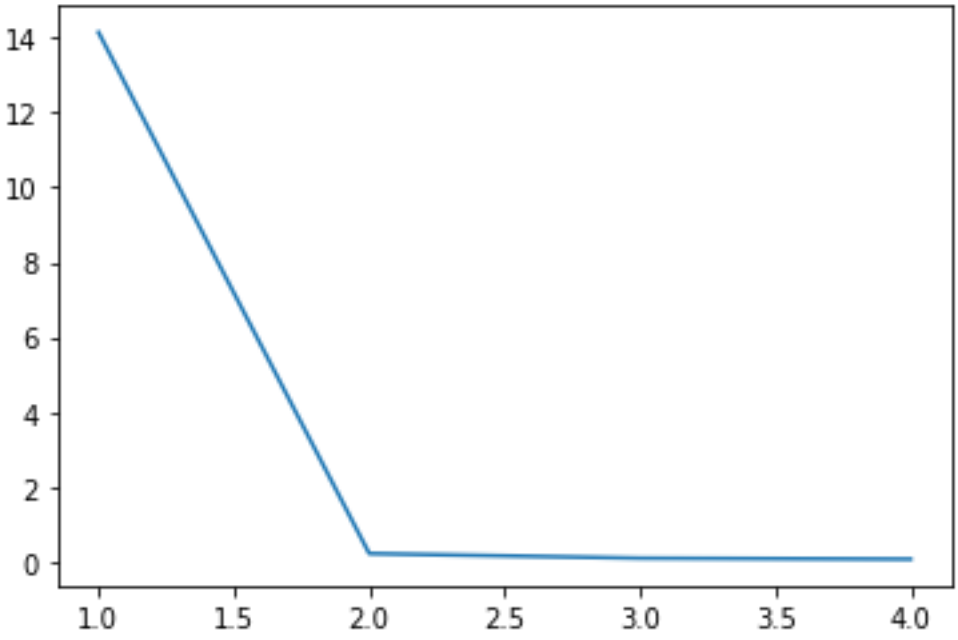
که خروجی کد فوق به صورت زیر خواهد بود:



همانطور که به وضوح قابل مشاهده است، شبکه اتوانکدر توانسته در لایه میانی خود که ابعاد کوچکتری دارد، به جدا سازی کامل دو دسته از دیتا(که با دانش پیشین ما در مورد دیتاست سازگار است) بپردازد.

حال با این نمایش کوچک شده از دیتای اولیه، می توانیم الگوریتم kmean را پیاده سازی کنیم و دیتای در دست را خوشه بندی نماییم.

جهت انتخاب تعداد کلاستر ها در الگوریتم kmean ، ابتدا نیاز است نموداری موسوم به elbow را رسم کنیم، این نمودار به ما کمک می‌کند که با تخیص دادن نقطه زانویی، در مورد تعداد کلاستر های بهینه تصمیم بگیریم:



همانطور که در تصویر فوق مشخص است، نقطه شکست نمودار elbow در مقدار 2 بوده که نشان دهنده این است که تعداد 2 خوشه برای مسئله فوق بهینه خواهد بود، که این موضوع با داشتن دانش پیشین ما در مورد دیتاست مطابقت می کند.

حال کافیست که دیتای در دست را با تعداد دو خوشه کلاستر کنیم و سپس می توان از قسمت انکدر شبکه اتو انکدر + الگوریتم kmean برای خوشه بندی دیتای جدید استفاده کرد:

```
kmean = KMeans(2,init = "random",n_init=10,random_state = 0)
label = kmean.fit_predict(encoded)
#test function
path = '/content/test_blue_berry.jpg'
image = Image.open(path)
image =np.array([np.asarray(image)])
encoded_test = np.array([encoder.predict(image)[0,:,0,0]])
cluster = kmean.predict(encoded_test)
cluster = cluster - 1
print("it belong to cluster{}".format(cluster))
```

در کد فوق، نتیجه خوشه بندی یک تصویر تست بلوبری انجام شده است، نتیجه کد را می توانید با اجرا کردن کد در پوشه سوال مشاهده کنید.

در ادامه، روند انجام کار خوشه بندی برای تصاویر سیاه و سفید به طور یکسانی مشابه با روند فوق است، تنها تفاوت آن سیاه و سفید کردن دیتاست در هنگام لود کردن تصویر، و تنظیم کردن ابعاد ورودی و خروجی اتوانکدر است، قسمت های متفاوت برای تصویر سیاه و سفید را در زیر مشاهده می کنیم:

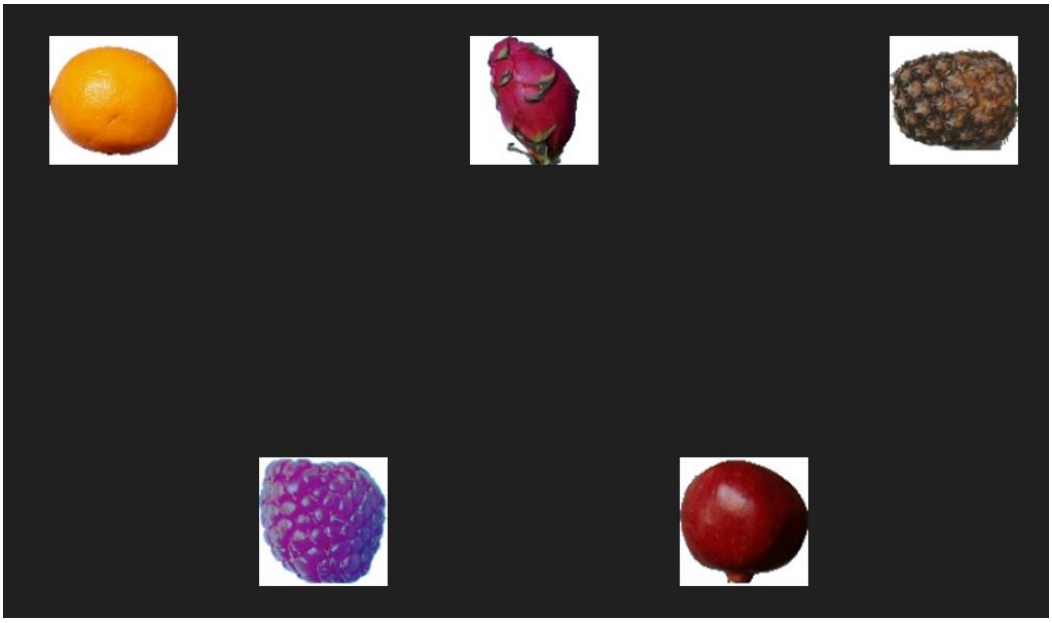
```
#dataset loading and pre processing data
def data_loader(dataset_path):
    dataset_gray = []
    #shuffling pictures
    dir = listdir(dataset_path)
    for fruit_type in dir:
        file = listdir(dataset_path+'/'+fruit_type)
        #random.shuffle(file)
        for filename in file:
            #print(dataset_path+'/'+fruit_type+'/'+filename)
            fruit = Image.open(dataset_path+'/'+fruit_type+'/'+filename)
            fruit = ImageOps.grayscale(fruit)
            fruit_gray  = np.asarray(fruit)
            dataset_gray.append(fruit_gray)
    #return the dataset
    return np.array(dataset_gray)/255
data = np.expand_dims(data_loader('dataset'),3)
#building model
input_layer = Input((100,100,1))
layer = Conv2D(50,(6,6),activation = 'tanh',padding ='same')(input_layer)
layer = MaxPooling2D(pool_size = (4,4),strides =(4,4))(layer)
layer = Conv2D(50,(5,5),activation = 'tanh',padding= 'same')(layer)
layer = MaxPooling2D(pool_size = (4,4),strides =(6,6))(layer)
layer = Conv2D(1,(3,3),activation = 'tanh')(layer)
encoder_layer = Conv2D(1,(1,2),activation = 'elu',name = 'encoder_end')(layer)
layer = UpSampling2D((5,5))(encoder_layer)
layer = Conv2D(50,(3,3),activation = 'tanh',padding = 'same')(layer)
layer = UpSampling2D((5,5))(layer)
layer = Conv2D(100,(5,5),activation = 'tanh',padding = 'same')(layer)
layer = UpSampling2D((2,4))(layer)
layer = Conv2D(1,(6,6),activation = 'sigmoid',padding = 'same')(layer)
model = Model(input_layer,layer)
#model.summary()
model.compile(optimizer = 'sgd',loss = 'mae')
model.fit(data,data,batch_size = 400,epochs = 50,verbose = False)

```

نتایج برای عکس های سیاه و سفید نیز کاملا مشابه عکس های رنگی است.

سوال 6 – رنگی کردن تصاویر

برای این سوال، قسمتی از دیتاست fruit 360 را انتخاب کرده ایم که شامل 1000 عکس از میوه های مختلف با سایز 100*100 پیکسل و با بکگراند سفید اند.

در زیر نمونه هایی از این دیتاست را مشاهده می‌کنیم:



دیتاست در دست، برای انجام عمل رنگ آمیزی باید ابتدا مورد پیش پردازش قرار گیرد، هدف این قسمت شافلینگ نمونه های دیتاست برای جلوگیری از بایاس، تقسیم دیتاست به دو نمونه عکس های خاکستری و رنگی و سپس نرمالیزه کردن آن ها بین مقدار 0 و 1 است. در زیر تابعی که این امور را انجام میدهد مشاهده می‌کنیم:

```
#dataset loading and pre processing data
def data_loader(path):
    dataset_color = []
    dataset_gray = []
    #shuffling pictures
    dir = listdir(path)
    random.shuffle(dir)
    for filename in dir:
        fruit = Image.open(path+filename)
        fruit_color  = np.asarray(fruit)
        dataset_color.append(fruit_color)
        #converting to grayscale
        fruit_gray = ImageOps.grayscale(fruit)
        fruit_gray = np.asarray(fruit_gray)
        dataset_gray.append(fruit_gray)
    #return the color and  gray datasets and normalizig them
    return np.array(dataset_color)/255,np.array(dataset_gray)/255
folder = "dataset/natural_images/fruit/"
```

```
19. color_dataset,gray_dataset = data_loader(folder)
```

حال اگر به *shape* دیتاست خاکستری(ورودی های شبکه عصبی) نگاه کنیم متوجه می شویم که ابعاد آن به صورت (1000,100,100) است، در حالی که باید به صورت (1000,100,100,1) باشد. (آن بعد اضافه در انتها معرف کتعداد کانال های تصویر است، و حال اینجا که تصویر ما تک کانال و سیاه و سفید بوده، باید آن را تصحیح کرد):

```
1.  gray_dataset = gray_dataset.reshape(1000,100,100,1)
2.  print(gray_dataset.shape)
3.  (1000, 100, 100, 1)
```

حال دیتاست را به دو قسمت آموزش و تست تقسیم می کنیم:

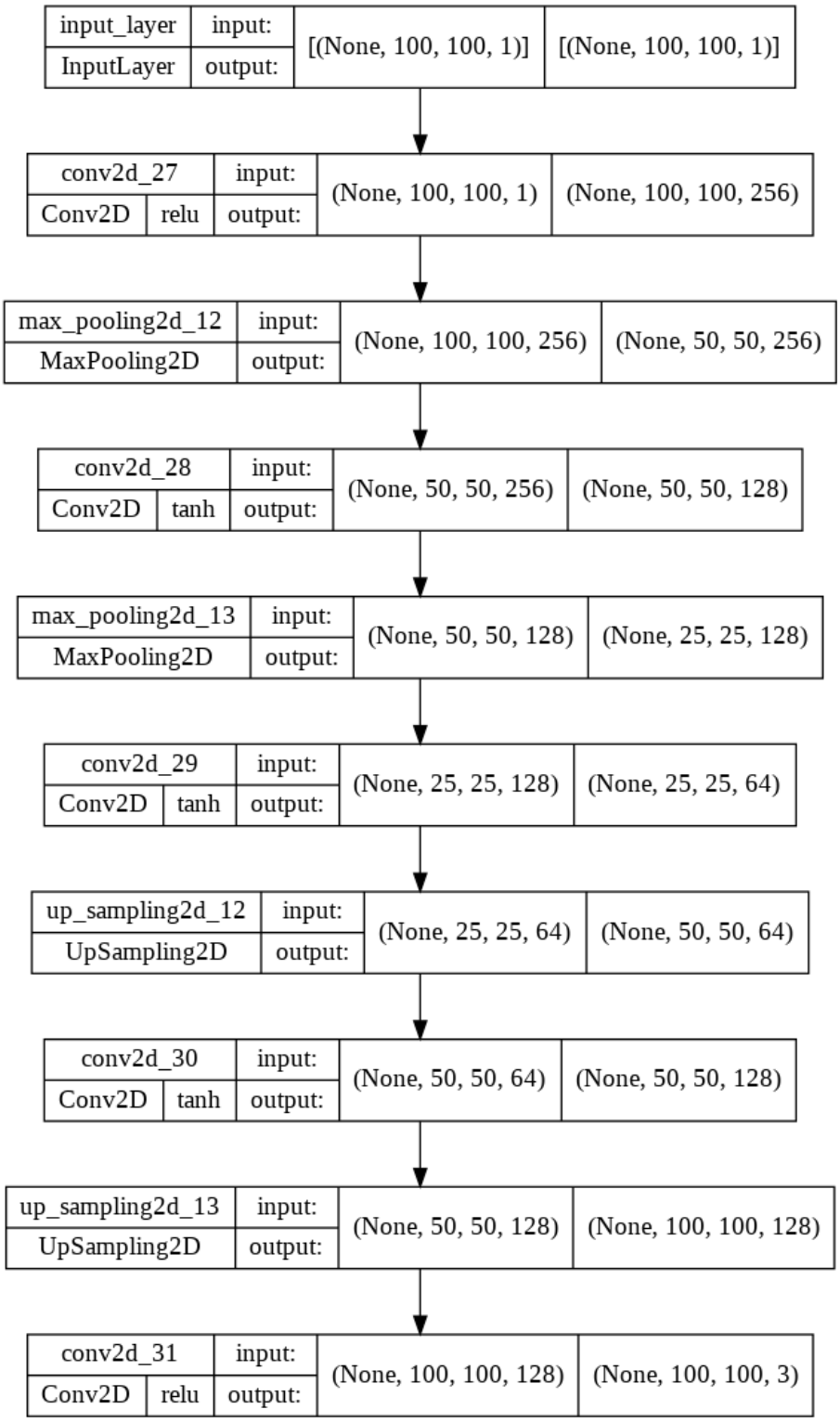
```
1.  train_gray = gray_dataset[:900]
2.  train_color = color_dataset[:900]
3.  test_gray = gray_dataset[900:]
4.  test_color = color_dataset[900:]
5.
```

حال باید مدل شبکه عصبی مورد نیاز برای تسک رنگی کردن را بسازیم. شبکه طراحی شده برای حل این سوال یک autoencoder است که در لایه ورودی عکس ها در قالب یک تنسور با یک کانال(سیاه و سفید) و در لایه خروجی همان طول و عرض تصویر و با سه کانال رنگی (RGB) تحویل می‌دهد.

در زیر خلاصه ای از معماری مدل aurtoencoder طراحی شده را می‌بینیم:

```
Model: "model_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_layer (InputLayer)    [(None, 100, 100, 1)]     0

 conv2d_27 (Conv2D)          (None, 100, 100, 256)     6656

 max_pooling2d_12 (MaxPoolin  (None, 50, 50, 256)      0
 g2D)

 conv2d_28 (Conv2D)          (None, 50, 50, 128)       295040

 max_pooling2d_13 (MaxPoolin  (None, 25, 25, 128)      0
 g2D)

 conv2d_29 (Conv2D)          (None, 25, 25, 64)        73792

 up_sampling2d_12 (UpSamplin  (None, 50, 50, 64)       0
 g2D)

 conv2d_30 (Conv2D)          (None, 50, 50, 128)       73856

 up_sampling2d_13 (UpSamplin  (None, 100, 100, 128)    0
 g2D)

 conv2d_31 (Conv2D)          (None, 100, 100, 3)       3459

=================================================================
Total params: 452,803
Trainable params: 452,803
Non-trainable params: 0
```

و تصویر زیر مسیر مستقیم عبور دیتا را در لایه ها به همراه توابع فعال سازی آن ها به نمایش می‌گذارد:



حال کدی که معماری فوق را می سازد مشاهده می کنیم:

```
input_layer = Input(shape=(100,100,1))
layer = Conv2D(256,(3,3),padding = 'same',activation = 'relu')(input_layer)
layer = MaxPooling2D((2,2))(layer)
layer = Conv2D(128,(3,3),padding = 'same',activation = 'tanh')(layer)
layer = MaxPooling2D((2,2))(layer)
layer = Conv2D(64,(3,3),padding = 'same',activation = 'tanh')(layer)
layer = UpSampling2D((2,2))(layer)
layer = Conv2D(128,(3,3),padding = 'same',activation = 'tanh')(layer)
layer = UpSampling2D((2,2))(layer)
output_layer = Conv2D(3,(3,3),padding = 'same',activation = 'relu')(layer)
model = Model(input_layer,output_layer)
model.compile(optimizer = 'adam',loss = 'mse',metrics = 'acc')
```

در واقع در شبکه autoencoder سعی داریم با فشرده و سپس استخراج ، داده اولیه را در خروجی باز سازی نماییم. با کوچک تر شدن ابعاد ورودی پس از گذر از هر لایه، فیچر های با اهمیت تر دیتاست استخراج شده و سپس با افزایش ابعاد سعی در باز سازی دیتاست اولیه با آن فیچر های مهم داریم.

حال در این قسمت مدل طراحی شده را با دیتاست در دست(ورودی تصویر خاکستری میوه و خروجی تصویر رنگی همان میوه) می پردازیم:

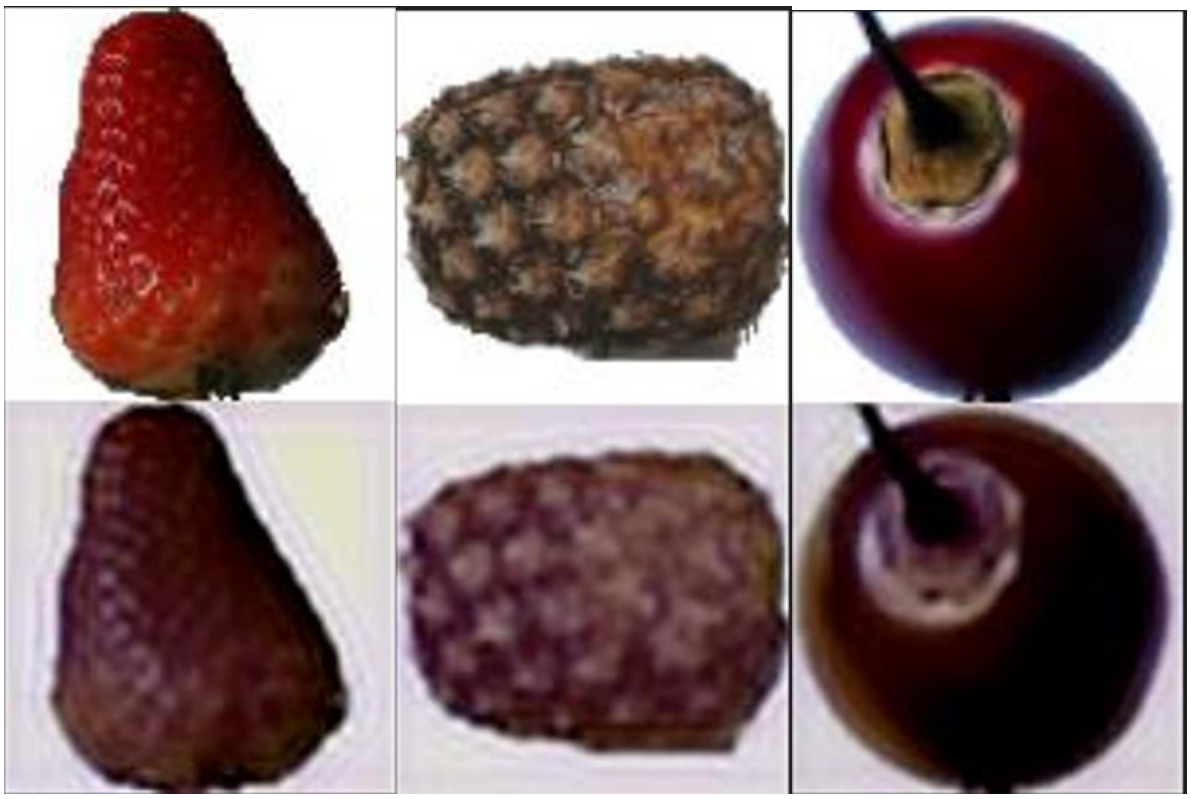
```
model.fit(train_gray,train_color,batch_size = 10,epochs = 6,validation_split = 0.2)
```

حال باید به رویارویی[11] شبکه آموزش دیده با دیتای تست بپردازیم:

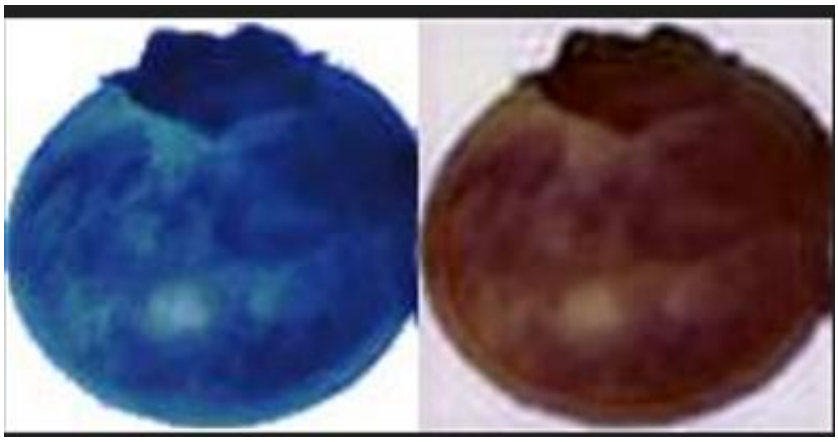
```
test = model(test_gray)
test = np.model(test_gray)
for j in len(0,test.shap[0]):
    rec_fruit = (255*(test[j]- test[j].min())/test[j].max()).astype(uint8)
    fruit = (255*test_color[j]).astype(uint8)
    comparison = np.concatenate(rec_fruit,fruit,axis = 0)
    image = Image.fromarray(comparison)
    image.save('test/test{}.jpg'.format(j))
```

حال در زیر به مقایسه عکس های رنگی شده توسط شبکه و خود عکس های رنگی اصلی می‌پردازیم:

[11] Inference



در بالا عکس های اصلی و در پایین نمونه های بازسازی شده رنگی از روی تصاویر سیاه و سفید همین میوه ها است، همانطور که مشاهده می شود، شبکه به مقدار قابل قبولی توانسته به رنگی کردن تصاویر میوه ها بپردازد. البته نمونه های جالبی نیز در دیتای تست وجود دارد:

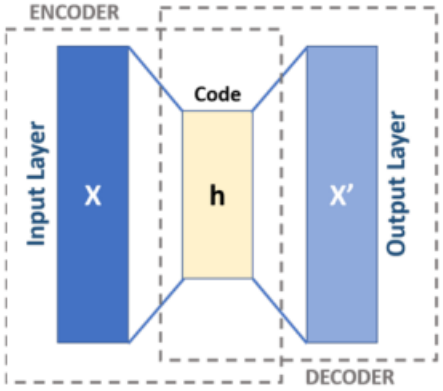


میوه فوق که بلو بری بوده ، توسط شبکه به رنگ سرخ رنگ آمیزی شده است! که علت این اتفاق می تواند به تعداد کمتر میوه بلوبری در دیتاست آموزش و اشتباه گرفتن این میوه با انار باشد. به صورت طبیعی هم میوه های کمی وجود دارند که رنگ آن ها در طیف آبی رنگ باشد(که دلیل این موضوع هم به خاطر سمی بودن میوه های آبی رنگ و در نتیجه وارد نشدن به چرخه کشاورزی توسط انسان است.) پس منطقی خواهد بود که پیش فرض شبکه عصبی در مورد میوه ها طیف رنگ های گرم شامل قرمز و زرد باشد.

سوال 7 – فرمول بندی و شبیه سازی VAE

VAE ها یا Variational Auto Encoder ها، حالت خاصی از شبکه های آتو انکدر اند که برای مقاصد تولید داده استفاده می گردند.

در آتو انکدر ها(که شکل آن در پایین رسم شده) سعی داشتیم که با طراحی خاصی از شبکه های عصبی، دیتای در دست را باز سازی کنیم، به طوری که خروجی شبکه به ورودی آن نزدیک باشد، و در این مسیر با کاهش ابعاد و سپس افزایش ابعاد، از اطلاعات نهفته در دیتاست خروجی را بازسازی کنیم:



یا به صورت ریاضی:

$$\hat{x} = D_\theta \left(E_\phi(x) \right) \cong x$$

که توابع D و E به ترتیب نشان دهنده likelihood دکدر و انکدر شبکه اتو انکدر هستند و پارامتر های آنان به ترتیب تتا و فی خواهد بود.

ببه طور طبیعی تابع هزینه ما به صورت خطای بازسازی به صورت زیر تعریف می‌شود:

$$\min \Sigma \left|\left| D_\theta \left(E_\phi(X) \right) - X \right|\right|$$

در اینجا X ماتریس نمونه هااز متغیر تصادفی x است. در اتو انکدر سعی داشتیم که پارامتر های تتا و فی را محاسبه کنیم:

$$\theta, \phi = \arg\left[\min\left[\Sigma \left|\left| D_\theta\left(E_\phi(X)\right) - X \right|\right|\right]\right]$$

حال بعد از آتو انکدر، فرض کنیم که هدف ما توسعه مدلی باشد که بتواند از فضایی با ابعاد پایین تر؛ داده ای با ابعاد بالاتر (که مطابق با توزیع احتمالاتی داده در دست است) بسازد. یعنی مدل ما در واقع نگاشتی از فضایی با ابعاد پایین تر به ابعاد بالاتر است:

$$G_\theta = \{\mathbb{R}^k \to \mathbb{R}^d, d > k\}_{z \to x}$$

$$z : latent\ vaiable\ , x: sample\ of\ data\ ditrubutaion$$

می دانیم که برای خروجی مدل بالا، برای نمونه x likelihood تقریبا در تمام فضای x برابر با صفر است، از این رو نمی توان به صورت مستقیم از الگوریتم های بهینه سازی برای یافتن پارامتر های تتا استفاده نمود، برای حل این مشکل، می توان فرض کرد که خروجی مدل فوق به صورت نویزی از ورودی فضای z انتخاب شده است، یعنی:

$$P(x|z) = N(x, G_\theta(z), \mu I)$$

که با ثابت در نظر گرفتن x می توان احتمال جوینت فوق را به صورت زیر ساده تر نوشت:

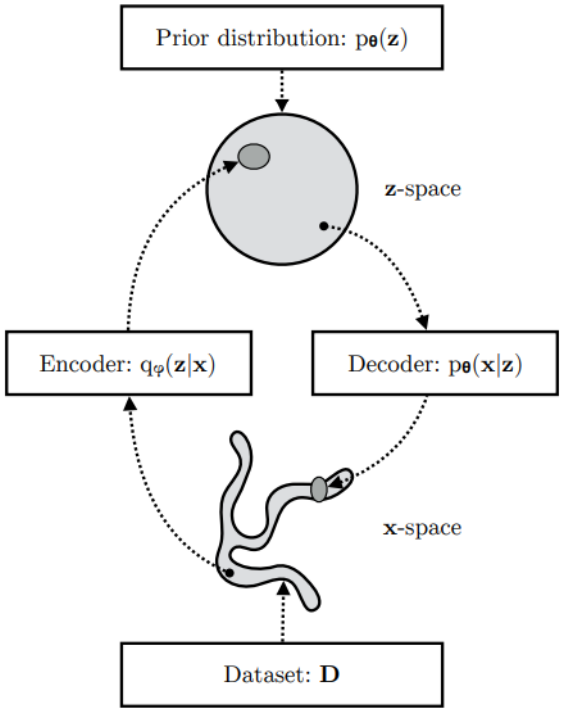
$$P_\theta(z)$$

حال با توضیحات فوق؛ می توان likelihood را برای x به صورت زیر نوشت:

$$P(x) = \int P(z)P(x|z)dz$$

حال مسئله VLB را مطرح می‌نماییم.

به صورت کلی، ستاپ یک شبکه VAE به صورت زیر است:



طبق شکل فوق، فضای z با یک تزویع نرمال و ثابت (مثلا گوسی) تعریف شده است، و می توانیم برای x نیز توزیعی به شکل زیر انتخاب کنیم:

$$x \sim P_\theta(x|z)$$

در نیمه سمت راست شکل صفحه قبل، مرحله ای که از فضای z به فضای x ما را می برد، image synthesis نام داشته و توسط قسمت دکدر شبکهVAE انجام می‌شود. در سمت چپ شکل فوق نیز دقیقا پروسه عکسی در حال رخ دادن است که قسمت انکدر انجام داده و فضای دیتاست را به فضای لایه نهفته نگاشت می کند(inference). در اینجا برای انکدر می توان نوشت:

$$P_\theta(z|x)$$

اما در واقع در بسیاری از موارد لزوما نمی خواهیم که توابع انکدر و دیکدر از یک دسته پارامتر استفاده کنند، پس یعنی می توان عبارت فوق را با توزیع دیگری که دارای پارامتر های فی است جایگزین کرد:

$$q_\phi\ (z|x)\ as\ proxy\ for\ P_\theta(z|x)$$

حال می‌خواهیم باند پایینی برای likelihood مشاهده x پیدا کنیم، در اینجا از فرم لگاریتم likelihood استفاده می‌کنیم:

$$\log P_\theta(x) = \mathbb{E}_{z\sim q_\phi(z|x)}[(\log P_\theta(x))] = \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log \frac{P_\theta(x,z)}{P_\theta(z|x)}\right]$$

$$= \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log\left(\frac{P_\theta(x,z)}{q_\phi(z|x)} \times \frac{q_\phi(z|x)}{P_\theta(z|x)}\right)\right]$$

$$= \underbrace{\mathbb{E}_{z\sim q_\phi(z|x)}\left[\log\left(\frac{P_\theta(x,z)}{q_\phi(z|x)}\right)\right]}_{VLB(Variational\ Lower\ Bound):\mathcal{L}_{\theta,\phi}} + \underbrace{\mathbb{E}_{z\sim q_\phi(z|x)}\left[\log\left(\frac{q_\phi(z|x)}{P_\theta(z|x)}\right)\right]}_{ELBO(Evidence\ Lower\ Bound):D_{KL}(q_\phi(z|x)||P_\theta(Z|X))}$$

که ترم های فوق را به صورت زیر جدا می کنیم:

$$D_{KL}: ELBO = \mathbb{E}_{z\sim q}\left(\frac{q(z)}{P(z)}\right)$$

که در مورد تعریف فوق می توان موارد زیر را نوشت:

- رابطه فوق خاصیت جا به جایی ندارد یعنی:

$$D_{KL}(q||P) \neq D_{KL}(P||q)$$

- تعریف فوق، فاصله دو توزیع p , q را اندازه می گیرد.
- همیشه مقداری بیشتر از صفر داشته و در صورت تساوی p,q برابر صفر می‌شود.

با توجه به بند سوم موراد بالا، برای رابطه likelihood متغییر x می‌توان نوشت:

$$\log P_\theta \geq VLB$$

حال به جای بهینه کردن خود لایک لی هود x، می توان عمل بهنیه سازی را برای VLB انجام داد.

با کمی تغییرات در VLB و بر اساس رابطه احتمال جوینت داریم:

$$\mathcal{L}_{\theta,\phi} = \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log\left(\frac{P_\theta(x,z)}{q_\phi(z|x)}\right)\right] = \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log\left(P_\theta(x|z)\frac{P_\theta(z)}{q_\phi(z|x)}\right)\right]$$

$$= \underbrace{\mathbb{E}_{z\sim q_\phi(z|x)}\left[\log\left(P_\theta(x|z)\right)\right]}_{reconstruction\ error} + \underbrace{\mathbb{E}_{z\sim q_\phi(z|x)}\left[\log\left(P_\theta(x|z)\frac{P_\theta(z)}{q_\phi(z|x)}\right)\right]}_{regularization}$$

همانطور که در رابطه بالا اشاره شد، ترم اول رابطه VLB نشان دهنده خطای بازسازی است، برای روشن تر شددن موضوع جمله اول را بسط می دهیم:

$$P_\theta(x|z) = N(x, G_\theta, \mu I)$$

$$\Rightarrow \log P_\theta(x|z) = -\frac{1}{2m}\left|\left|x - G_\theta\right|\right|^2 + constant$$

طبق روابط فوق، با ماکزیمم کردن VLB توقع داریم که توزیع q به سمت نقطه ای بودن میل کند.

و برای ترم دوم داریم :

$$\mathbb{E}_{z\sim q_\phi(z|x)}\left[\log\left(\frac{P_\theta(z)}{q_\phi(z|x)}\right)\right] = -D_{KL}\left(q_\phi(z|x)\left|\left|P(z)\right)\right)$$

که طبق رابطه فوق، با ماکزیمم کردنVLB انتاظر داریم توزیع q گسترده تر شود.

در نتیجه، ماکزیمم کردن VLB بده بستانی برای این که توزیع q چگونه باشد خواهد بود.

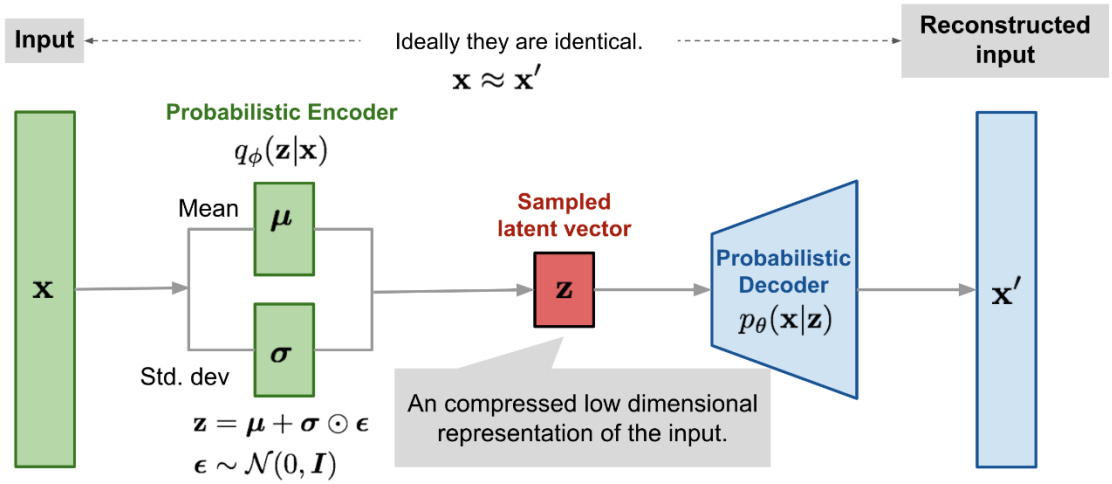
به صورت خلاصه:

- ماکزیمم کردن VLB منجر به ماکزیمم کردن P(x) می‌شود(چون VLBحد پایینی برای P(X) است)
- باعث مینمم شدن KL DIVERGENCE شده و توزیع بهتری از Q بدست می‌دهد.

در نتیجه به جای بهینه کردن likelihood(x) ما VLB را بهینه می نماییم:

$$objective = \max \sum_{i=1}^{n} \mathcal{L}_{\theta,\phi}(x_i)$$

حال ساختار VAE را در شکل زیر مشاهده می‌کنیم:



در قسمت سمت چپ، دو بلوکmean و std سعی در تعریف یک توزیع احتمالاتی نرمال در لایه پنهان دارند، همانطور که در قبل توضیح داده شد، برای این که مسئله بهینه سازی قابل انجام باشد، در بلوک قرمز رنگ z نویزی گوسی ضرب در واریانس بدست آمده لایه پنهان شده و به اضافه میانگین لایه پنهان می شود تا عمل نمونه برداری را انجام دهد. در واقع با چنین بلوکی برای q داریم:

$$q_\phi(z|x) = N(z, \mu(x), \sigma(x)I)$$

در ادامه به روش بهینه کردن VLB می پردازیم.

مسئله اصلی در شبکه VAE حال به صورت زیر تعریف می‌شود:

$$Data: D = \{x_i\}_{i=1,\dots,n}$$

$$\theta, \phi = \arg\left(\max\left(\sum\left(\mathcal{L}_{\theta,\phi}(x_i)\right)_{x_i \in D}\right)\right)$$

حال به گرفتن گرادیان برای روش SGD می پردازیم(برای قسمت دیکدر و پارامتر تتا):

$$\nabla_\theta \mathcal{L}_{\theta,\phi}(x) = \nabla \mathbb{E}_{z\sim q_\phi}[\log P_\theta(x,z) - \log q_\phi(z|x)] = \nabla \mathbb{E}_{z\sim q_\phi}[\log P_\theta(x,z)]$$

$$\cong \nabla_\theta \log P_\theta(x,z), z \sim q_\phi(z|x)$$

حال برای قسمت انکدر و پارامتر های تتا داریم :

$$\nabla_\phi \mathcal{L}_{\theta,\phi}(x) = \nabla_\phi \mathbb{E}_{z\sim q_\phi}[\log P_\theta(x,z) - \log q_\phi(z|x)]$$

در اینجا با استفاده از رابطه ای که برای q نوشتیم داریم:

$$q_\phi(z|x) = N(z, \mu(x), \sigma(x)I) = \mu(x) + \sigma(x).\epsilon\,, \epsilon \sim \mathrm{N}(0, \mathrm{I})$$

$$\mathcal{L}_{\theta,\phi}(x) = \mathbb{E}_{z\sim q_\phi}[\log P_\theta(x,z) - \log q_\phi(z|x)] = \mathbb{E}_{\epsilon\sim p(\epsilon)}[\log P_\theta(x,z) - \log q_\phi(z|x)]$$

$$\rightarrow \epsilon \sim P(\epsilon) \rightarrow z = \mu_\phi(x) + \sigma_\phi(x).\epsilon$$

$$\rightarrow \hat{\mathcal{L}}_{\theta,\phi}(x) = \log P_\theta(x,z) - \log q_\phi(z|x) \rightarrow \nabla_\phi \mathcal{L}_{\theta,\phi}(x) \cong \nabla_\phi \hat{\mathcal{L}}_{\theta,\phi}(x)$$

حال که ریاضیات VAE ها را شرح دادیم، می توانیم به شبیه سازی این شبکه عصبی خاص برای دیتاست fashion mnist بپردازیم، این کتابخانه شامل ده دسته بندی fashion به صورت زیر است:

Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

در مورد کد VAE ، معماری شبکه بسیار نزدیک به AE بوده و نیاز به توضیح مجدد تمام بخش های آن نیست، تنها دو تغییر کلیدی که این شبکه را با AE متفاوت می سازد بحث می‌کنیم، تمام روند لود کردن دیتاست و نمایش خروجی ها و .. به روال گذشته است.

1- اولین تفاوت – وجود لایه z

همانطور که در بالا اشاره شد، در لایه میانی VAE یک بلوکz وجود داشت که وظیفه انتخاب رندوم دیتا از توزیع نرمال بدست آمده در پارامترهای μ, σ است، حال در اینجا به استفاده از قابلیت ارث بری کلاس ها در پایتون، یک کلاس لایه از کتابخانه کراس ایجاد کرده تا عمل نمونه برداری بلوکz را انجام دهد:

```
class Z_block(layers.Layer):
    def call(self, inputs):
        z_mean, z_log_var = inputs
        batch = tf.shape(z_mean)[0]
        dim = tf.shape(z_mean)[1]
        epsilon = tf.keras.backend.random_normal(shape=(batch, dim))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon

```

در واقع در کلاس بالا و تابع call سعی داریم که رابطه ای که قبلا برای بلوک z نوشتیم را بدست آوریم. یعنی رابطه زیر:

$$z\_block = N(z, \mu(x), \sigma(x)I)$$

حال به تشکیل قسمت انکدر و دکدر VAE می پردازیم:

```
encoder_inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(32, 3, activation="relu", strides=2, padding="same")(encoder_inputs)
x = layers.Conv2D(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Flatten()(x)
x = layers.Dense(16, activation="relu")(x)
z_mean = layers.Dense(latent_dim, name="z_mean")(x)
z_log_var = layers.Dense(latent_dim, name="z_log_var")(x)
z = Z_block()([z_mean, z_log_var])
encoder = keras.Model(encoder_inputs, [z_mean, z_log_var, z], name="encoder")
encoder.summary()
latent_inputs = keras.Input(shape=(latent_dim,))
x = layers.Dense(7 * 7 * 64, activation="relu")(latent_inputs)
x = layers.Reshape((7, 7, 64))(x)
x = layers.Conv2DTranspose(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Conv2DTranspose(32, 3, activation="relu", strides=2, padding="same")(x)
decoder_outputs = layers.Conv2DTranspose(1, 3, activation="sigmoid", padding="same")(x)
decoder = keras.Model(latent_inputs, decoder_outputs, name="decoder")
decoder.summary()
```

و حال قسمت مهم vae ساخت مدلی است که شامل قسمت های دیکدر و انکدر باشد، و بتواند بر اساس تابع هزینه ای که در قبل اشاره شده بود وزن های شبکه را آموزش دهد:

```
class VAE(keras.Model):
    def __init__(self, encoder, decoder, **kwargs):
        super(VAE, self).__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder
        self.total_loss_tracker = keras.metrics.Mean(name="total_loss")
```

```
        self.reconstruction_loss_tracker = keras.metrics.Mean(
            name="reconstruction_loss"
        )
        self.kl_loss_tracker = keras.metrics.Mean(name="kl_loss")

    @property
    def metrics(self):
        return [
            self.total_loss_tracker,
            self.reconstruction_loss_tracker,
            self.kl_loss_tracker,
        ]

    def train_step(self, data):
        with tf.GradientTape() as tape:
            z_mean, z_log_var, z = self.encoder(data)
            reconstruction = self.decoder(z)
            reconstruction_loss = tf.reduce_mean(
                tf.reduce_sum(
                    keras.losses.binary_crossentropy(data, reconstruction), axis=(1, 2)
                )
            )
            kl_loss = -0.5 * (1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var))
            kl_loss = tf.reduce_mean(tf.reduce_sum(kl_loss, axis=1))
            total_loss = reconstruction_loss + kl_loss
        grads = tape.gradient(total_loss, self.trainable_weights)
        self.optimizer.apply_gradients(zip(grads, self.trainable_weights))
        self.total_loss_tracker.update_state(total_loss)
        self.reconstruction_loss_tracker.update_state(reconstruction_loss)
        self.kl_loss_tracker.update_state(kl_loss)
        return {
            "loss": self.total_loss_tracker.result(),
            "reconstruction_loss": self.reconstruction_loss_tracker.result(),
            "kl_loss": self.kl_loss_tracker.result(),
        }
```

همانطور که قبلا اشاره شد، با استفاده از خاصیت ارث بری کلاس ها در پایتون، یک کلاس جدید VAE که فرزند کلاس مادر MODEL از کتابخانه کراس است، ایجاد کرده ایم. حال به شرح قسمت های کلاس فوق می پردازیم:

- تابع init

در این تابع متغییر های داخلی کلاس دیکدر و انکدر را بر اساس ورودی های مدل کلی که شامل دو قسمت انکدر و دیکدر است تعریف می کنیم. در ادامه نیز توابع هزینه kl و خطای بازسازی را بر اساس متد های موجود در کتابخانه کراس تعریف کرده ایم.

- تابع metrics

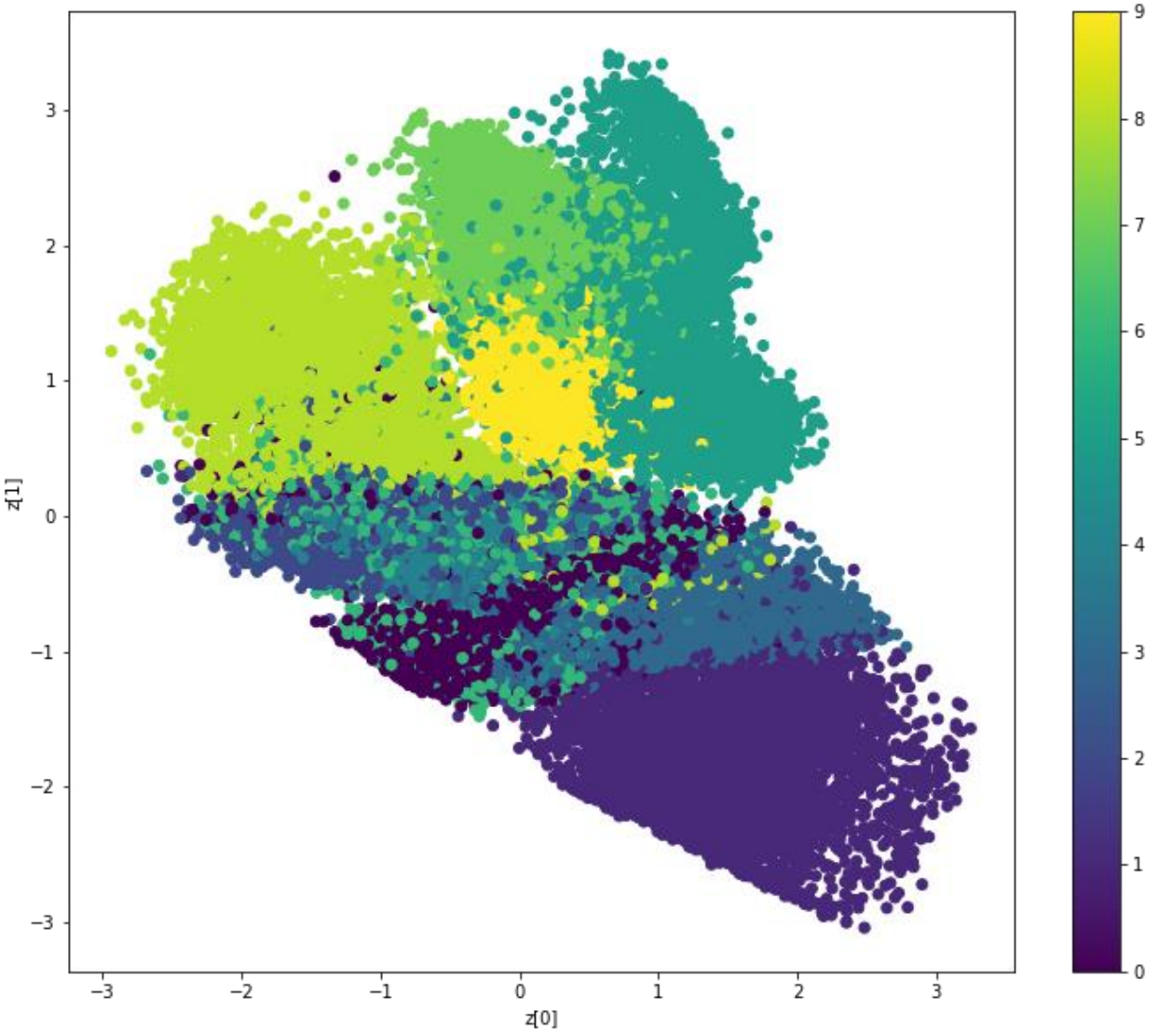
این تابع وظیفه بازگرداندن مقادیر خطای تعریف شده در قسمت قبل را دارد.

- تابع train_step

با استفاده از شی tape کتابخانه تنسور فلو، سعی داریم مسیر عبور دیتا از مدل vae را بدست آورده و سپس طبق این مسیر، پس انتشار خطا انجام داده و پارامتر ها را تنظیم نماییم.

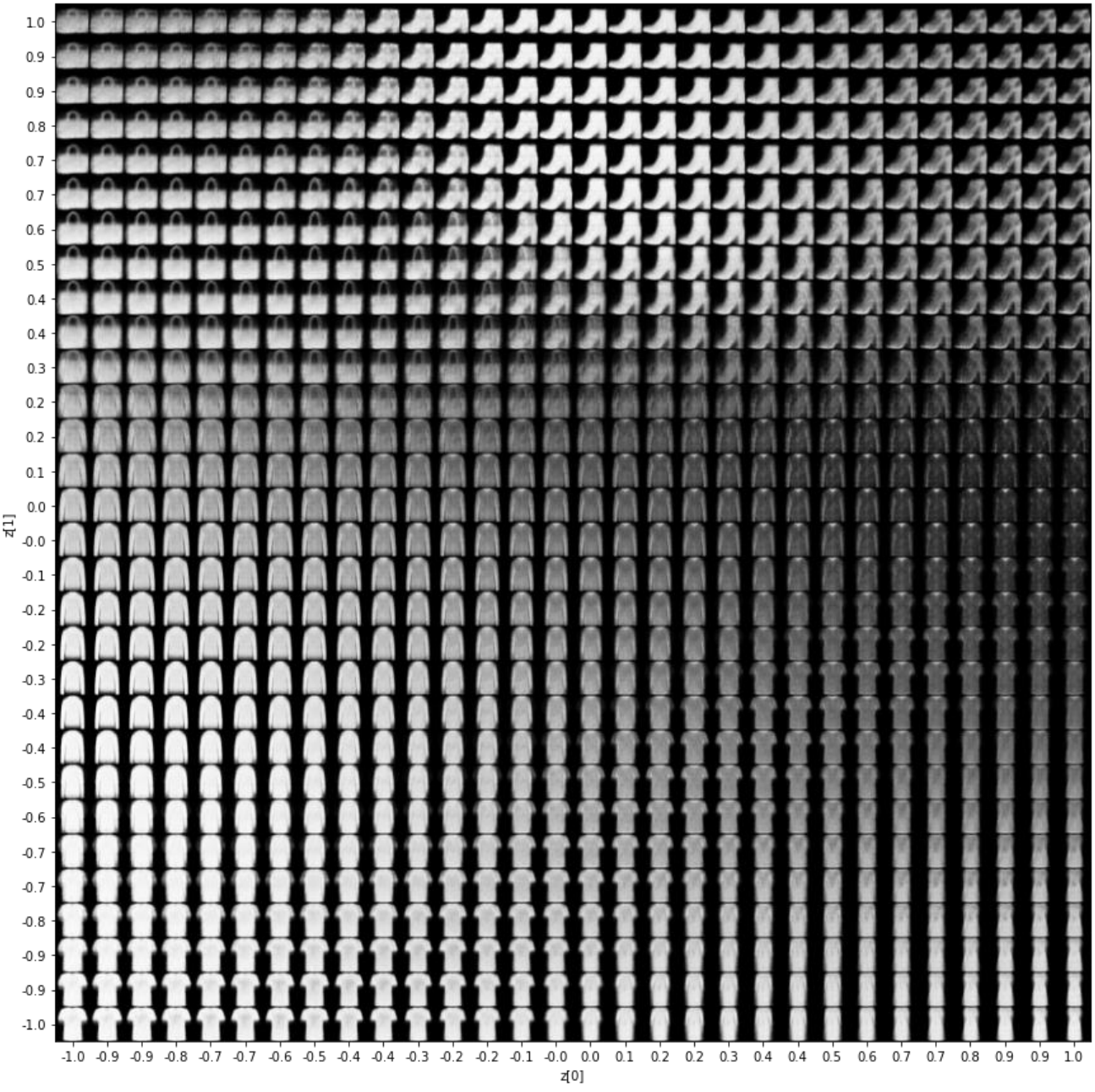
در این قسمت پارامتر های شبکه تنظیم شده و سپس متریک های تعریف شده در قسمت قبل آپدیت می گردند، و در نهایت متریک ها بازگردانده شده و برای نمایش پیرشفت شبکه در یادگیری به نمایش در می‌آیند.

بعد از آموزش شبکه می توانیم تصاویر ورودی نگاشت شده در لایه پنهان mean را مشاهده کنیم:



ستون رنگی سمت راست تصویر نشان دهنده کلاس دیتا است.

حال برای تولید دیتا می توان نمونه هایی از کلاستر های رسم شده در شکل بالا انتخاب کرد و سپس این مقادیر ره به ورودی قسمت دیکدر داد و عکس های نزدیک به واقعیت از توزیع احتمالاتی دیتاست تولید کرد:



در واقع شکل فوق نمایانگر خروجی دیتا به ازای نقاط مختلف دیتا پوینت در لایه پنهان mean است.

منابع استفاده شده برای حل این سوال

[1] Diederik P. Kingma and Max Welling (2019), “An Introduction to Variational Autoencoders”, Foundations and Trends R in Machine Learning: Vol. xx, No. xx, pp 1–18. DOI: 10.1561/XXXXXXXXX.

[2] Razavi, A., Van den Oord, A., & Vinyals, O. (2019). Generating diverse high-fidelity images with vq-vae-2. *Advances in neural information processing systems*, *32*.

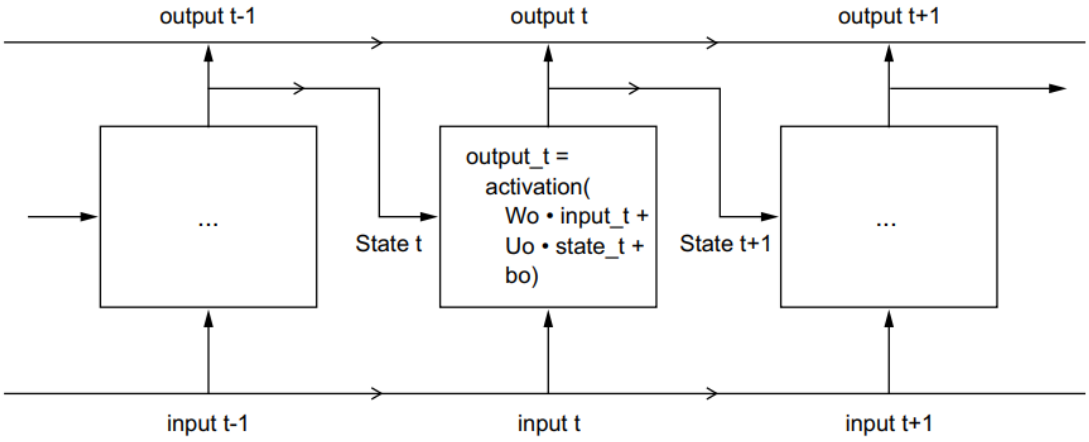
[3] https://www.youtube.com/watch?v=c27SHdQr4lw&t=2s

[4] https://www.youtube.com/watch?v=8wrLjnQ7EWQ&t=518s

سوال 8 – تشریح برتری شبکه های LSTM بر RNN ساده و پیاده سازی آن برای طبقه بندی دیتایECG

بخش اول – تشریح برتری LSTM بر RNN معمولی

شبکه های RNN یا Recurrent Neural Network ها، نوع خاصی از شبکه های عصبی اند که به منظور پردازش دیتای با اهمیت زمانی و مرحله ای استفاده می‌شود. در واقع قدرت این شبکه ها نسبت به شبکه های معمول مثل پرسپترون و کانولوشنی در این است که خروجی هر نرون در هر لحظه زمانی، در مرحله زمانی بعد تر وارد چرخه یادگیری شبکه می شود و اینطور نیست که خروجی هر لحظه از دیگر لحظات مستقل باشد. در زیر دیاگرامی از این شبکه ها مشاهده می‌کنیم:



همانطور که واضح است، خروجی در هر مرحله زمانی به صورت زیر قابل محاسبه است:

$$h_t = activation(W \times input_t + U \times h_{t-1})$$

حال فرض کنیم که می خواهیم پارامتر های شبکه فوق را توسط عمل پس انتشار، تنظیم نماییم، فرض کنیم که خروجی شبکه در هر لحظه از زمان برابر h_t باشد، از آن جایی که خروجی هر مرحله زمانی وارد مرحله زمانی بعد می‌شود، و این که می خواهیم از آخرین مرحله خروجی خطا را حساب کنیم، آن گاه با روش پس انتشار گرادیان شبکه فوق به صورت زیر در می‌آید:

$$\frac{\partial Loss}{\partial w_0} = \frac{\partial Loss}{\partial \hat{a}} \cdot \frac{\partial \hat{a}}{\partial h_n} \cdot \frac{\partial h_n}{\partial h_{n-1}} \cdot \dots \cdot \frac{\partial h_0}{\partial w_0}$$

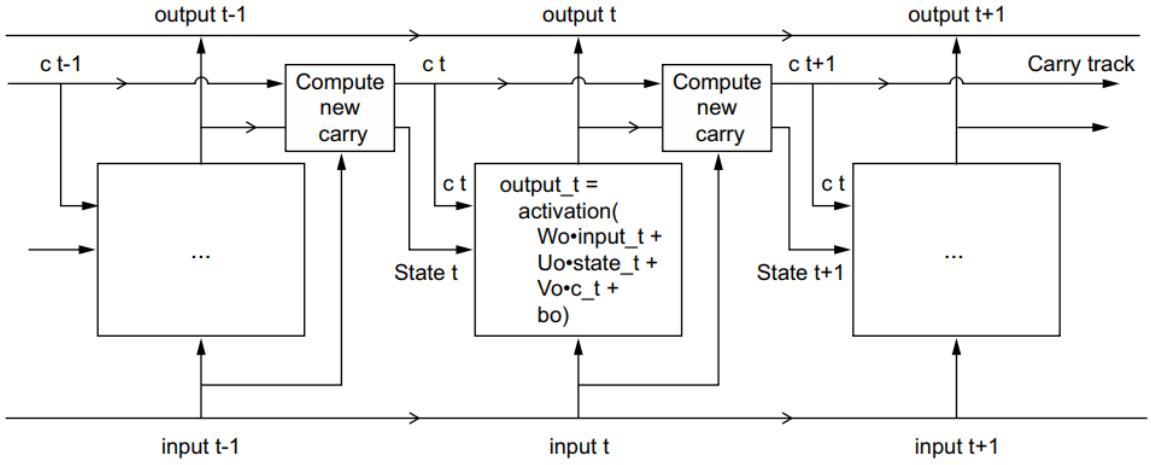
$$\rightarrow \frac{\partial Loss}{\partial w_0} = \frac{\partial Loss}{\partial \hat{a}} . w_h^T . w_{h-1}^T \dots . w_1^T . \frac{\partial h_0}{\partial w_0}$$

در نتیجه برای تنظیم پارامتر های شبکه در لحظه 0 نیاز مندیم که به اندازه n بار ضرب کردن مشتقات زنجیری از تابع loss به گرادیان نسبت به پارامتر w0 برسیم، در همین راستا اگر عددn به اندازه کافی بزرگ باشد، با مسئله انفجار و محو شدگی خطا مواجه خواهیم بود، فرض کنیم که تعداد مراحل زمان1000باشد و خطای خروجی 1.01 ، پس از هزار بار ضرب شدن این عدد در خودش به مقدار بسیار بزرگ 20959 می رسیم که برای تنظیم وزن کاملا نامناسب است، به این پدیده exploding gradian گفته می‌شود، در حالت دیگر فرض کنیم خطای نهایی 0.99 باشد، پس از هزار بار ضرب کردن این عدد در خودش به مقدار 0.00004317124 می رسیم که همچنان برای تنظیم پارامتر های وزن در لحظه 0 نامناسب خواهد بود، به این پدیده نیز vanishing gradient می گویند.

در حالت کلی این پدیده منحصر به شبکه های RNN با تعداد مراحل زمانی نیست، بلکه در شبکه های sequential که تعداد لایه ها زیاد باشد نیز اتفاق می افتد.

در سال 1997 ، Sepp Hochreiter و همکاران او، نوع جدیدی از شبکه ها به نامlong short term memory را عرضه کردند که مشکل گرادیان محو شونده را که در شبکه های معمولی RNN وجود داشت محو نماید.

معماری زیر یک شبکه LSTM را به تصویر می کشد:



همانطور که مشاهده می کنیم کلیت معماری این شبکه همانند شبکه های RNN است، با این تفاوت که یه خط حمل کننده دیتا (Carry track) اضافه شده است، وظیفه این خط انتقال دادن دیتا از یک مرحله به مرحله بعد بوده به گونه ای که تنها اطلاعات مورد نیاز در این مسیر حمل شود.

در هر مرحله عبات Ct که حاصل سه ترم متفاوت است به صورت زیر محاسبه می شود:

$$c_t = c_{t-1} f_t + i_t \times n_t$$

که به ترتیب هر کدام از عناصر رابطه بالا را با یکدیگر تحلیل می کنیم:

- f_t یا لایه فراموشی:
  - وظیفه این تابع این است که تصمیم بگیرد چه میزان از اطلاعاتی که از مرحله قبل بر روی خط حامل انتقال یافته، باید در خط انتقال باقی بماند، به طور معمول از تابع سیگموید برای آن استفاده می شود و رابطه آن به شکل زیر است:

$$f_t = \sigma\left(w_{f_t} \times h_{t-1}\right)$$

- i_t یا لایه به خاطر سپردن ورودی:
  - این تابع تصمیم می گیرد که چه مقدار از این اطلاعاتی که توسط لایه n_t انتخاب شده،به عنوان کاندیدی برای بهبود پاسخ آینده در خط حامل باقی بماند، به طور مرسوم تابع فعال ساز این لایه نیز سیگموید است:

$$i_t = \sigma\left(w_{i_t} \times h_{t-1}\right)$$

- n_t یا لایه کاندید های ورودی:
  - این لایه تصمیم می گیرد که چه مقدار از اطلاعات ورودی به عنوان کاندید جدید اطلاعات در خط حامل قرار بگیرد. و به طور مرسوم تابع فعال سازی tanh برای این قمست در نظر گرفته می‌شود.

$$n_t = \tanh\left(w_{n_t} \times h_{t-1}\right)$$

و در نهایت خروجی در هر لحظه زمانی بر حسب تابعی از لحظه قبل و اطلاعات خط حامل بدست می آید:

$$h_t = activation\left(w_{h_t} \times h_{t-1}\right) \times \tanh(c_t)$$

معماری معرفی شده در فوق، این قابلیت را دارد که در تعداد بسیار زیادی از مراحل زمانی بدون مشکل محو شدگی و انفجار گرادیان به یادگیری سری های زمانی بپردازد، در واقع خط حامل و توابع فعال سازی به کار رفته برای محاسبه خط حامل این توانایی را به شبکه می‌دهد که در مسیر پس انتشار، خطا به سمت بی نهایت و یا صفر میل نکند.

بخش دوم – پیاده سازی شبکه LSTM برای تشخیص ضربان قلب معمولی از ضربان قلب بیمار

دیتا ست در دست، شامل 3986 نمونه از دیتای ECG بوده که در دو کلاس نرمال و غیر نرمال به صورت دستی برچسب خورده است، هر نمونه(ردیف) از این دیتا ست شامل 141 پارامتر بوده که 140 پارامتر نخست دیتای مربوط به ضربان قلب و پارامتر نهایی نشان دهنده کلاس آن نمونه است، که مقدار صفر نشان دهنده ضربان قلب نرمال و مقدار یک نشان دهنده نمونه آنرمال است. در کد زیر پیش پردازشی برای جدا کردن این دیتا را مشاهده می‌کنیم، ابتدا کتابخانه های مورد نیاز را وارد محیط می‌کنیم:

```
1.  from matplotlib import pyplot as plt
2.  from tensorflow.keras.layers import LSTM, Input,Dense
3.  from tensorflow.keras.models import Model
4.  import pandas as pd
5.  import numpy as np
```

سپس به پردازش دیتا و جدا کردن دیتا از لیبل ها که پارامتر نهایی هر ردیف دیتا اند می پردازیم:

```
1.  dataset = pd.read_csv("ecg.csv")
2.  dataset = dataset.to_numpy()
3.  dataset = np.expand_dims(dataset, axis=2)
4.  labels = dataset[:,-1]
5.  dataset = dataset[:,:-1]
```

حال سایز دیتا و لیبل ها را چک می کنیم:

```
1.  print("target shape is :",labels.shape,"____ and dataset shape is :",dataset.shape)
2.  target shape is : (3986, 1) ____ and dataset shape is : (3986, 140, 1)
```

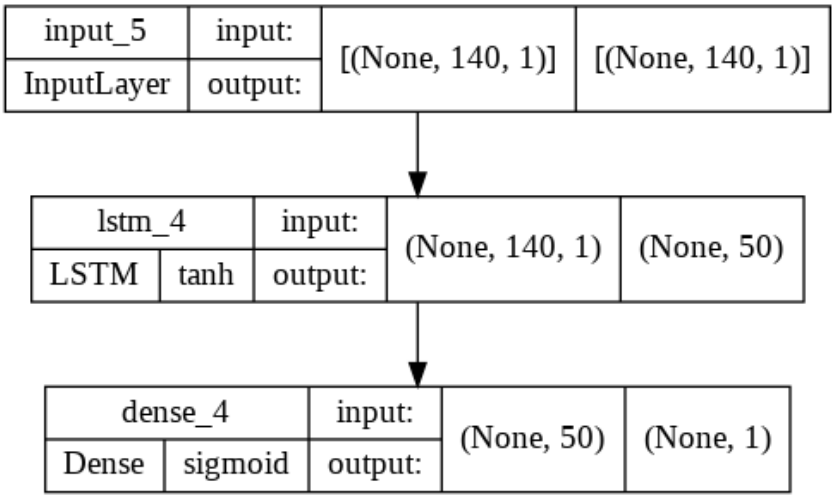
همانطور که دلخواه ما است، در دیتاست 3986 ردیف با 140 پارامتر داریم، و تارگت هم سازی برابر 3986 در 1 دارد که نشان دهنده 0 یا 1 بودن هر کدام از نمونه ها است.

حال به طراحی و ساخت مدل می پردازیم، معماری انتخاب شده برای این قسمت یک معماری ساده دو لایه بوده که شامل یک لایه LSTM و یک لایه پرسپترون تماما متصل است:

```
1.  #building model
2.  input_layer = Input((140,1))
3.  lstm = LSTM(50)(input_layer)
4.  output_layer = Dense(1,activation = 'sigmoid')(lstm)
```

```
5.  model = Model(input_layer,output_layer)
```

به صورت مختصر معماری شبکه فوق به صورت زیر است:



در لایه LSTM پنجاه نرون قرار داده ایم که دیتا را در پنجاه مرحله زمانی پردازش می‌کند، از طرافی در لایه انتهایی یک نرون با تابع فعال سازی سیگموید قرار دارد که قرار است خروجی ره به صورت احتمالی از این که نمونه ورودی جزو کدام کلاس است نمایش دهد، این شبکه شامل 10451 پارامتر و وزن است.

در ادامه به ساختن مدل می پردازیم:

```
1.  model.compile(optimizer = 'adam',loss = 'BCE',metrics='acc')
```

آپتیمایزر مدل فوق را adam انتخاب کرده ایم که به صورت پیش فرض یکی از آپتیمایزر های مناسب برای آموزش شبکه است، تابع loss نیز با توجه به این که خروجی شبکه(لایه پرسپترون خروجی) یک نرون با تابع فعال سازی سیگموید برای طبقه بندی بین دو کلاس است، Binary Cross Entropy انتخاب شده که انتخاب مناسبی برای مسئله درد دست(طبقه بندی بین دو کلاس) است. در نهایت متریک و معیار عملکرد شبکه را accuracy انتخاب کرده ایم.

قدم بعدی آموزش شبکه با دیتا ست در دست خواهد بود :

```
1.  model.fit(dataset,labels,shuffle=True,validation_split = 0.25,epochs = 10)
```

در کد فوق، آرگومان shuffle برابر True قرار گرفته که در نتیجه ردیف های دیتاست را به صورت رندوم برای آموزش شبکه استفاده می نماید، دلیل این موضوع هم به ساختار دیتاست بر میگردد که نیمه ابتدایی دیتاست

تنها نمونه هایی با لیبل 0 قرار دارد و نیمه دوم آن سمپل هایی با لیبل 1 ، برای جلوگیری از تصمیم گیری شبکه بر اساس اولویت دیتای ورودی، این آرگومان را فعال کرده ایم.

آرگومان بعدی دیتاست را به صورت اتوماتیک به دو قسمت آموزش و تست تقسیم می نماید، با قرار دادن 0.25، یک چهارم کل دیتا صرفا برای بررسی عملکرد شبکه استفاده شده و تنها 0.75 درصد دیتا برای آموزش استفاده می‌شود.

در نهایت نتیجه آموزش شبکه را مشاهده می‌کنیم:

```
1.  Epoch 1/10
2.  94/94 [==============================] - 10s 66ms/step - loss: 0.0118 - acc: 0.9970 - val_loss:
    nan - val_acc: 0.8987
3.  Epoch 2/10
4.  94/94 [==============================] - 6s 61ms/step - loss: 0.0097 - acc: 0.9977 - val_loss:
    nan - val_acc: 0.8997
5.  Epoch 3/10
6.  94/94 [==============================] - 6s 61ms/step - loss: 0.0094 - acc: 0.9983 - val_loss:
    nan - val_acc: 0.9007
7.  Epoch 4/10
8.  94/94 [==============================] - 6s 62ms/step - loss: 0.0075 - acc: 0.9983 - val_loss:
    nan - val_acc: 0.9097
9.  Epoch 5/10
10. 94/94 [==============================] - 6s 62ms/step - loss: 0.0122 - acc: 0.9973 - val_loss:
    nan - val_acc: 0.9017
11. Epoch 6/10
12. 94/94 [==============================] - 6s 62ms/step - loss: 0.0098 - acc: 0.9977 - val_loss:
    nan - val_acc: 0.8997
13. Epoch 7/10
14. 94/94 [==============================] - 6s 61ms/step - loss: 0.0135 - acc: 0.9970 - val_loss:
    nan - val_acc: 0.9067
15. Epoch 8/10
16. 94/94 [==============================] - 6s 62ms/step - loss: 0.0089 - acc: 0.9977 - val_loss:
    nan - val_acc: 0.9047
17. Epoch 9/10
18. 94/94 [==============================] - 6s 61ms/step - loss: 0.0071 - acc: 0.9970 - val_loss:
    nan - val_acc: 0.9057
19. Epoch 10/10
20. 94/94 [==============================] - 6s 61ms/step - loss: 0.0197 - acc: 0.9957 - val_loss:
    nan - val_acc: 0.9067
21.
```

همانطور که مشاهده می کنیم، شبکه با دقت بسیار عالی 90 درصد روی دیتای تست توانسته است که ضربان قلب نرمال را از ضربان قلب آنرمال تشخیص دهد.

البته باید توجه داشت که طبق نتایج فوق، واضح است که شبکه دچار پدیده overfitting شده است که یک پدیده رایج در شبکه های LSTM است ،برای جلوگیری از رخداد این پدیده می توان در هر لایه تعدادی عملگر dropout قرار داد که با حذف رندوم خروجی های هر لایه، از overfit شدن شبکه جلوگیری نماید، یک راه حل دیگر افزایش درصد دیتای تست به مقادیری بیشتر از 35 درصد خواهد بود .

مراجع استفاده شده برای حل این سوال

[1] https://towardsdatascience.com/how-the-lstm-improves-the-rnn-1ef156b75121

[2] https://towardsdatascience.com/rnns-from-theory-to-pytorch-f0af30b610e1

[3] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*(8), 1735-1780.

[4] Chollet, F. (2018). *Deep learning mit python und keras: das praxis-handbuch vom entwickler der keras-bibliothek*. MITP-Verlags GmbH & Co. KG.

[5] https://towardsdatascience.com/multi-class-text-classification-with-lstm-1590bee1bd17

سوال 9 – نمایش Heatmap های توجه شبکه کانولوشنی با استفاده از Global Average Pooling

بخش اول – تشریح چگونگی به نمایش در آوردن Heatmap های توجه، و اهمیت تفسیر پذیری

یکی از اساسی ترین مسائل هنگامی که یک مدل برای بینایی کامپیوتر توسعه می دهیم، مسئله **تفسیر** کردن عملیاتی است که مدل ما در حال انجام دادن آن است. به طور مثال هنگامی که یک شبکه کانولوشنی برای طبقه بندی عکس های حیوانات(مثل سگ و گربه) آموزش می‌دهیم، از کجا بفهمیم که هر قسمت از شبکه در حال انجام چه وظیفه ای است؟ به طور مثال چرا یه مدل طبقه بندی کننه یک عکس را یخچال تشخیص می دهد در حالی که در داخل تصویر چیزی جز یک کامیون نیست؟

مسئله **تفسیر پذیری** در کاربرد های عکس برداری پزشکی خود را بیشتر نشان می‌دهد، زیرا در اینجا سعی داریم مدلی توسعه دهیم که کار یک متخصص را تقلید کند، پس باید بتوانیم در سطحی قابل قبول عملکرد قسمت های مختلف را تفسیر کنیم، مثلا مدل توسعه داده شده به چه المان هایی در یک عکس MRI دقت کرده که توانسته تومور مغزی را تشخیص بدهد؟

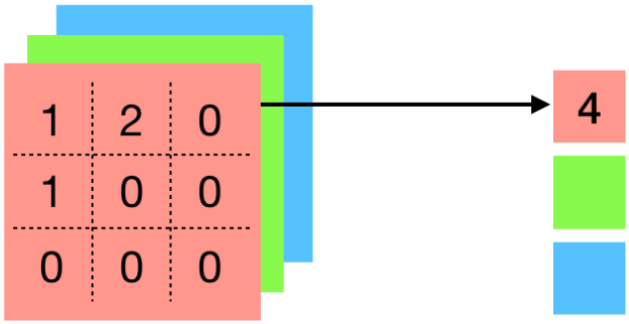
اگر چه در حالت کلی تفسیر عملکرد اجزای شبکه های عصبی به راحتی میسر نیست(مانند بخش دوم سوال سوم که سعی داشتیم وزن های تابع تبدیل را از مدل چند لایه استخراج کنیم) با این حال شبکه های کانولوشنی یکی از شبکه هایی هستند که به خاطر ماهیت عملکرد آن ها بر روی تصویر، تا حد زیادی قابل تفسیر برای انسان اند.

برای حل این مسئله اساسی، روش های متعددی توسعه داده شده است(برای شبکه های کانولوشنی) که به سه مورد آن اشاره می کنیم:

1- به نمایش در آوردن خروجی لایه های کانولوشنی میانی: کارآمد برای درک این که لایه کانولوشنی چگونه به صورت پی در پی دیتای ورودی را تغییر می‌دهند
2- به نمایش در آوردن خود فیلتر های کانولوشنی: برای اینکه بفهیمم هر لایه دقیقا چه الگو یا مفهومی را در از داخل ورودی استخراج می کند(مثلا فیلتر های تشخیص لبه و ..)
3- به نمایش در آوردن هیت مپ های کانولوشنی: می توانیم تخیص دهیم که فیلتر کانولوشنی به کدام قسمت از تصویر ورودی بیشترین تاثیر و اهمیت را داده است.

که در این مسئله، مورد سوم از موارد فوق را بررسی می‌کنیم.

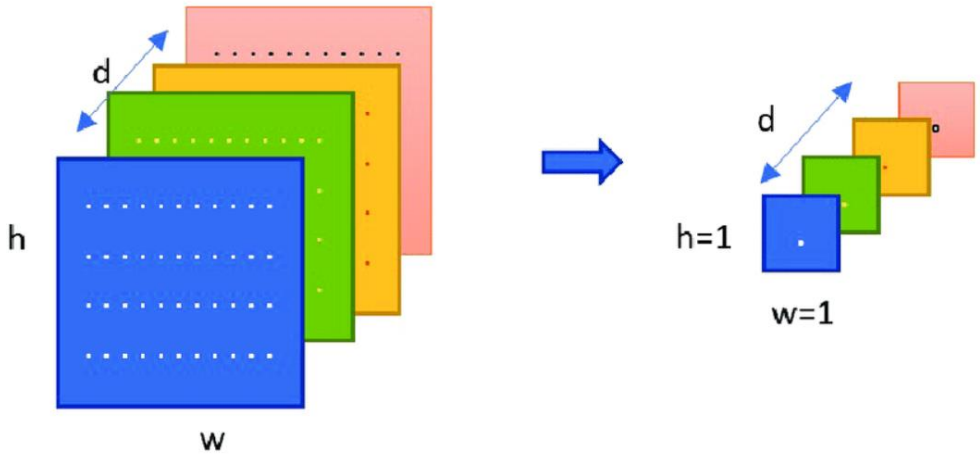
در ابتدا لازم است که فیلتر Global Average Pooling را شرح دهیم، همانطور که از اسم این فیلتر پیداست، وظیفه این فیلتر گرفتن میانگین تمام پیکسل های تصویر ورودی است، به شکل زیر دقت کنیم:



خروجی این فیلتر به سادگی توسط رابطه زیر قابل تعریف است(با فرض این که سایز ورودی m در n باشد) :

$$GAP = \frac{1}{m \times n} \sum_{i=1}^{m} \sum_{j=1}^{n} value_{ij}$$

به صورت کلی اگر به تعداد d عدد فیلتر کانولوشنی در ورودی GAP داشته باشیم، سایز خروجی این لایه برابر با (d,1) خواهد بود، یعنی شکل زیر:



حال معماری شبکه ای را که برای انجام عمل طبقه بندی (مثلا سگ و گربه) به صورت زیر فرض کنیم:

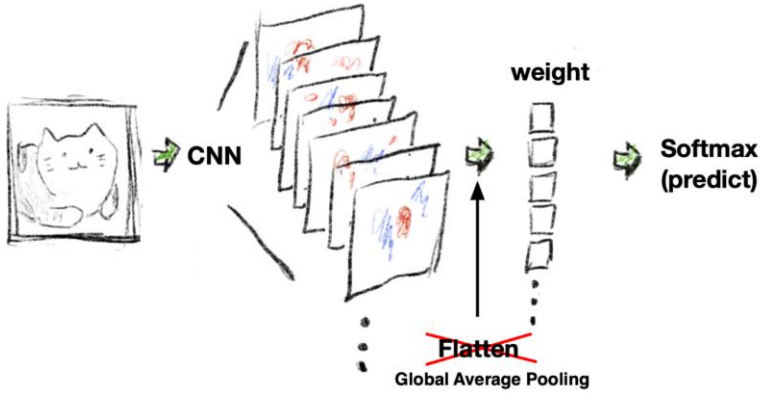
- در بخش اول، به تعداد دلخواه لایه کانولوشنی که لایه آخر به تعداد Dعدد فیلتر کانولوشنی با سایز دلخواه
- در لایه دوم یک GAP

- در لایه سوم یک پرسپترون با تعداد نرون وابسته به تعداد طبقات

حال تغییر ابعاد را در گذر از هر لایه با یکدیگر مرور می کنیم:

- ورودی یک تصویر رنگی (دارای سه کانال قرمز و سبز و آبی)با ابعاد دلخواه $m \times n$ وبه تعداد P نمونه، یعنی:$(P, m, n, 3)$
- پس از گذر از تعداد دلخواه فیلتر کانولوشنی، و در لایه آخر این بخش d فیلتر کانولوشنی با سایز دلخواه داریم:(P, M, N, d)
- پس از گذر از لایه GAP همانطور که قبلا اشاره کردیم داریم: (P, d)
- و در نهایت پس از عبور از لایه پرسپترون خروجی که عمل طبقه بندی را انجام می‌دهد، خروجی به صورت مقابل خواهد بود:(P, x) که البته با فرض این که مسئله ما طبقه بندی سگ و گربه باشد به یک نرون احتیاج خواهیم داشت.

به صورت خلاصه، معماری شبکه فوق به صورت زیر است:



حال طبق توضیحاتی که قبل تر دادیم، یک روش برای استخراج HeatMap های شبکه بالا، این است که ببنیم در هر نمونه، کدام یکی از d فیلتر کانولوشنی لایه اول، بیشترین سهم را در انتخاب آن نمونه به عنوان یک طبقه خاص داشته، و سپس خروجی آن فیلتر خاص را روی تصویر اولیه نشان دهیم.

به زبان ساده تر، فرض کنیم که مدل فوق را روی دیتاستی شامل عکس های شگ و گربه آموزش داده ایم، سپس یک عکس نمونه گربه را انتخاب کرده و از شبکه عبور می‌دهیم، شبکه به طور مثال به ما می گوید که تصویر نشان داده شده یک گربه است، حال از اینجا (یعنی خروجی نرون آخر) باید برگردیم عقب و ببینیم که کدامین ورودی بیشترین تاثیر را روی این انختاب گذاشته، طبق معماری فوق رابطه نرون آخر با تابع فعال سازی سیگموید را در نظر بگیریم:

$$\hat{y} = \sigma\left(\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_d \\ b \end{bmatrix}^T \times \begin{bmatrix} GAP_1 \\ GAP_2 \\ \vdots \\ GAP_d \\ 1 \end{bmatrix}\right), GAP_n: Global\ Average\ Pool\ of\ nth\ conv\ filter$$

در نرون فوق،فرض کنیم بیشترین تاثیر در تصمیم گیری خروجی را حاصل ضرب $w_n \times GAP_n$ داشته است، حال خروجی فیلتر n امی که GAP آن بیشترین تاثیر در تصمیم نرون انتهایی را داشته پیدا می کنیم، طبق چیزی که قبلا نوشته بودیم، انتظار داریم که اندازه این خروجی دقیقا مطابق با اندازه تصویر ورودی ما باشد(چون پدینگ فیلتر کانولوشنی را برابر فرض کرده ایم)

حال با قرار دادن خروجی این فیلتر بر روی تصویر اصلی، می توانیم بفهمیم که دقیقا فیلتر کانولوشنی به چه قسمت هایی از تصویر اهمیت بیشتر(مطابق با اندازه بیشتر) داده است.

البته انجام این عمل صرفا منحصر به فیلتری که بیشترین تصمیم را بر روی خروجی گذاشته نیست، بلکه می توان خروجی تمام فیلتر های کانولوشنی را بر روی تصویر ورودی قرار داد تا بفهمیم دقیقا هر فیلتر به کدام قسمت های تصویر ورودی حساسیت بیشتری نشان داده.

بخش دوم – پیاده سازی

طبق معماری فوق که توضیح دادیم، برای نشان دادن هیت مپ ها توسط لایه GAP باید به ترتیب یک لایه کانولوشنی، لایه GAP و لایه پرسپترون تصمیم گیر داشته باشیم.

جهت بدست آوردناین معماری خاص شبکه طبقه بندی کننده دو راه وجود دارد:

1- ساختن مدلی از ابتدا برای حل مسئله
2- استفاده از مدل های از پیش آموزش داده شده

که ما در این جا به دلیل صرفه جویی در زمان و داشتن نتیجه بهتر، به جای طراحی و آموزش شبکه از ابتدا، از مدل از پیش آموزش داده شده RESNET50 استفاده کرده ایم، مزیت این شبکه در این است که دقیقا در سه لایه آخر این شبکه، معماری مورد نیاز ما گنجانده شده است، به سه لایه آخر مدل رز نت در زیر دقت کنیم:

```
conv5_block3_out (Activation)  (None, 7, 7, 2048)   0           ['conv5_block3_add[0][0]']

 avg_pool (GlobalAveragePooling  (None, 2048)        0           ['conv5_block3_out[0][0]']
 2D)

 predictions (Dense)            (None, 1000)         2049000     ['avg_pool[0][0]']

==================================================================================================
```

همانطور که واضح است، به ترتیب یک لایه کانولوشنی با 2048 فیلتر داریم، سپس یک لایه GAP و سپس یک لایه پرسپترون 1000 عددی. دلیل 1000 عدد بودن تعداد نرون های لایه آخر این است که مدلRESNET روی دیتا ست تصویر IMAGENET که شامل هزار دسته بندی مختلف است آموزش داده شده است(از جمله تصاویر حیوانات مثل سگ و گربه)

حال کافیست که تمامی گفته های فوق را در مورد خروجی این شبکه برای عکس نمونه از یک سگ اجرا کنیم و هیت مپ های توجه را بر روی تصویر ورودی استفاده نماییم.

ورودی مدل رز نت به صورت پیش فرض یک عکس رنگی به ابعاد 224 در 224 پیکسل است.

در اینجا لود کردن مدل و استخراج لایه های مورد نیاز از مدل رز نت را مشاهده می‌کنیم:

```
#from tensorflow.keras.layers import Input,GlobalAveragePooling2D,Conv2D,Dense
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Model
```

```
from matplotlib import pyplot as plt
import numpy as np
from os import listdir
from PIL import Image
from tensorflow.keras.applications.resnet50 import ResNet50,decode_predictions
from scipy.ndimage import zoom

#obtain resnet model
resnet_model = ResNet50()
resnet_model.summary()

# get useful output out of resnet model
convlayer_output = resnet_model.get_layer('conv5_block3_out').output
GAP_layer_output = resnet_model.get_layer('avg_pool').output
prediction_output = resnet_model.get_layer('predictions').output
model = Model(resnet_model.input,[convlayer_output,GAP_layer_output,prediction_output])
```

در چهار خط کد فوق، خروجی لایه های کانولوشنی و GAP و پرسپترون را دریافت می کنیم، سپس در خط آخر مدلی بر اساس مدل رز نت تولید کرده که خروجی هایش، همان سه پارامتر تعریف شده باشد.

سپس به کمک کتابخانه pillow که برای کار با تصاویر در پایتون است، تعداد عکس سگ و گربه را وارد محیط کار می‌نماییم:

```
data = []
for file in listdir('test'):
  image = Image.open('test/'+file)
  image = image.resize((224,224))
  data.append(np.asarray(image))
data = np.array(data)
```

همانطور که در کد فوق پیداست، سایز تمامی تصاویر به صورت 224 در 224 پیکسل تنظیم شده است.

در کد پایین، ابتدا وزن های نرونی که بیشترین خروجی را دارد(یعنی طبقه عکس ورودی را تخیص داده است) از لایه آخر استخراج می‌کنیم:

```
j = 12 #input picture
conv_out,GAP_out,predict = model.predict(np.expand_dims(data[j],axis=0))
perceptron_weight,_= resnet_model.get_layer('predictions').weights
perceptron_weight= perceptron_weight[:,np.argmax(predict)]
```

سپس با ضرب کردن elementwise این ضرائب در خروجی هر فیلتر کانولوشنی، می توانیم ببینیم که هر کدام از فیلتر های کانولوشنی به چه اندازه ای در فعال سازی نرون لایه آخر تاثیر داشته اند:

```
nine_filter = np.multiply(perceptron_weight,GAP_out[0])
```

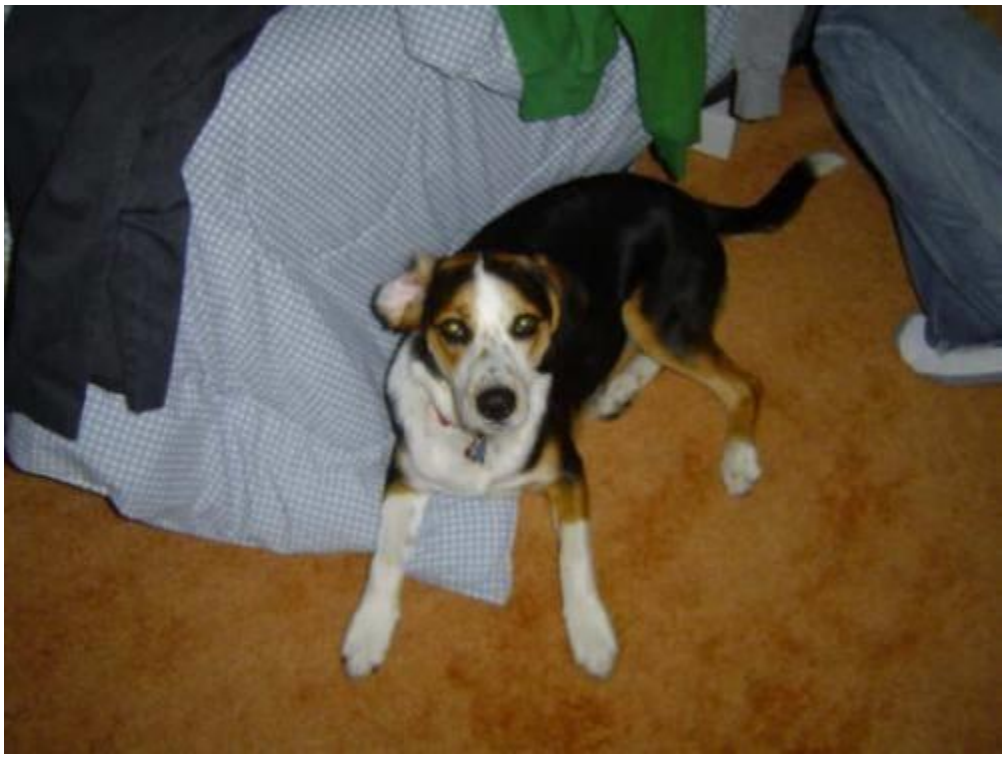
در نهایت، خروجی نه عدد فیلتر کانولوشنی که بیشترین تاثیر در تصمیم گیری نرون آخر را داشته اند، از لایه کانولوشنی استخراج می‌کنیم:

```
1.  nine_filter = np.argpartition(nine_filter,-9,axis = 0)[-9:]
2.  nine_heatmap = conv_out[0,:,:,[nine_filter]]
```

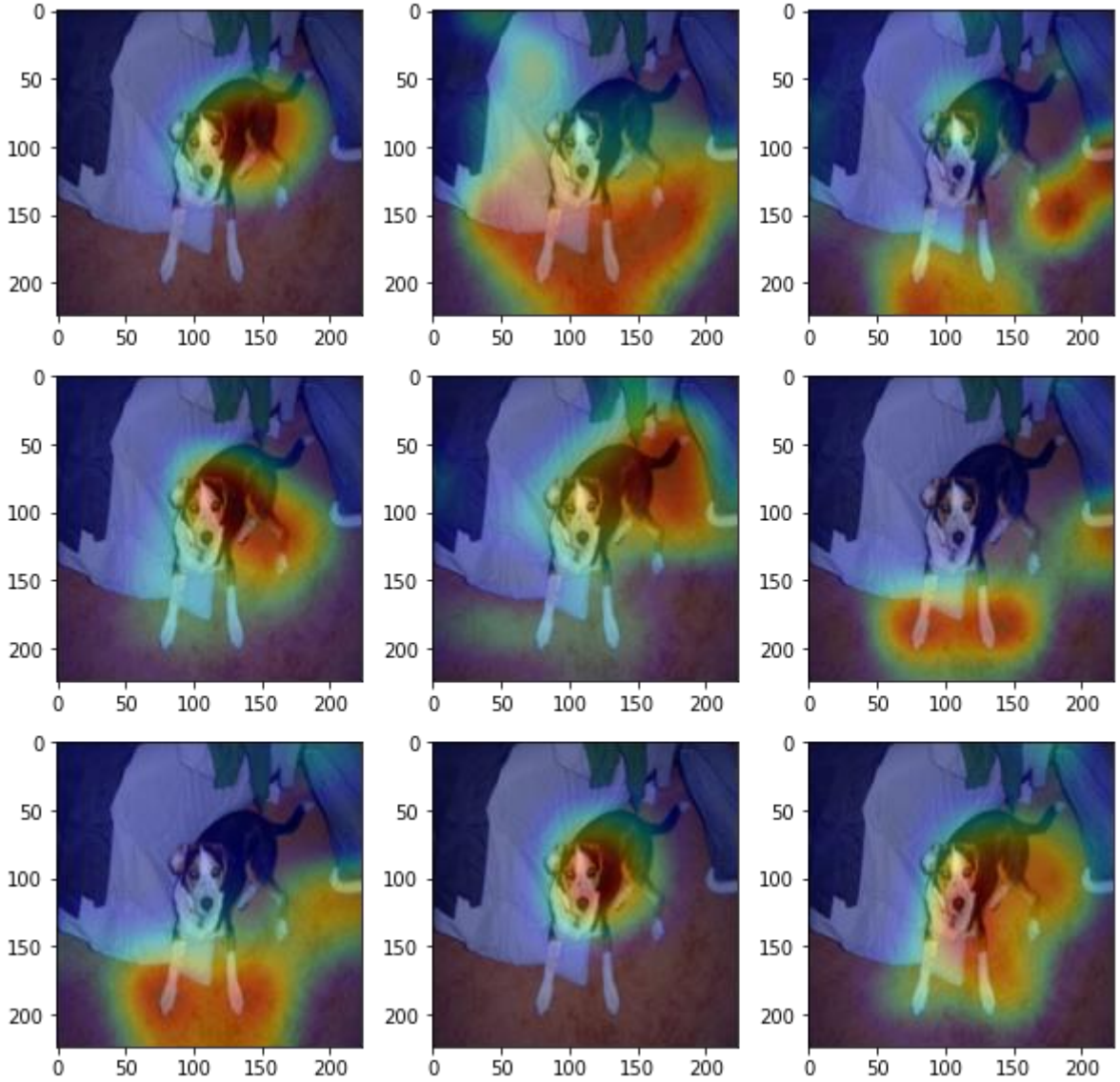
از آن جایی که خروجی فیلتر کانولوشنی به خاطر ساختار شبکه رز نت، به صورت 7 در 7 است، برای تصویر کردن این خروجی بر روی عکس اولیه، نیاز داریم که این مپ را بزرگ نمایی کنیم. در نهایت با محو کردن تقریبی خروجی این نه لایه بر روی تصویر اصلی می توانیم هیت مپ های توجه را به سادگی رسم کنیم:

```
1.  plt.figure(figsize= (10,10))
2.  for i in range(1,10):
3.    plt.subplot(3,3,i)
4.    plt.imshow(data[j])
5.    plt.imshow(zoom(nine_heatmap[0,i-1],zoom =(224/7)),cmap = 'jet',alpha=0.3)
```

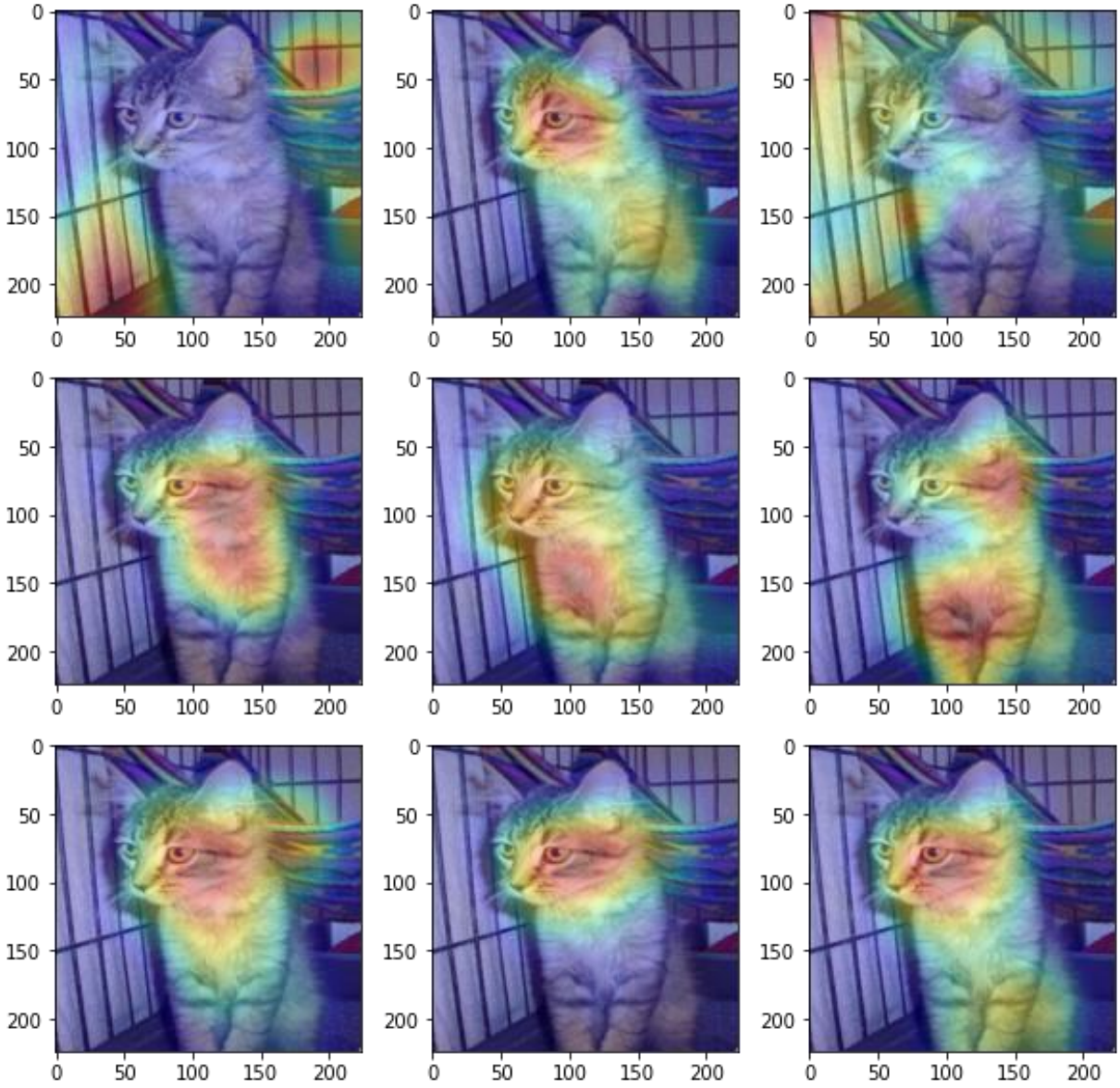
در ابتدا عکس ورودی شبکه را با یکدیگر مشاهده می کنیم:

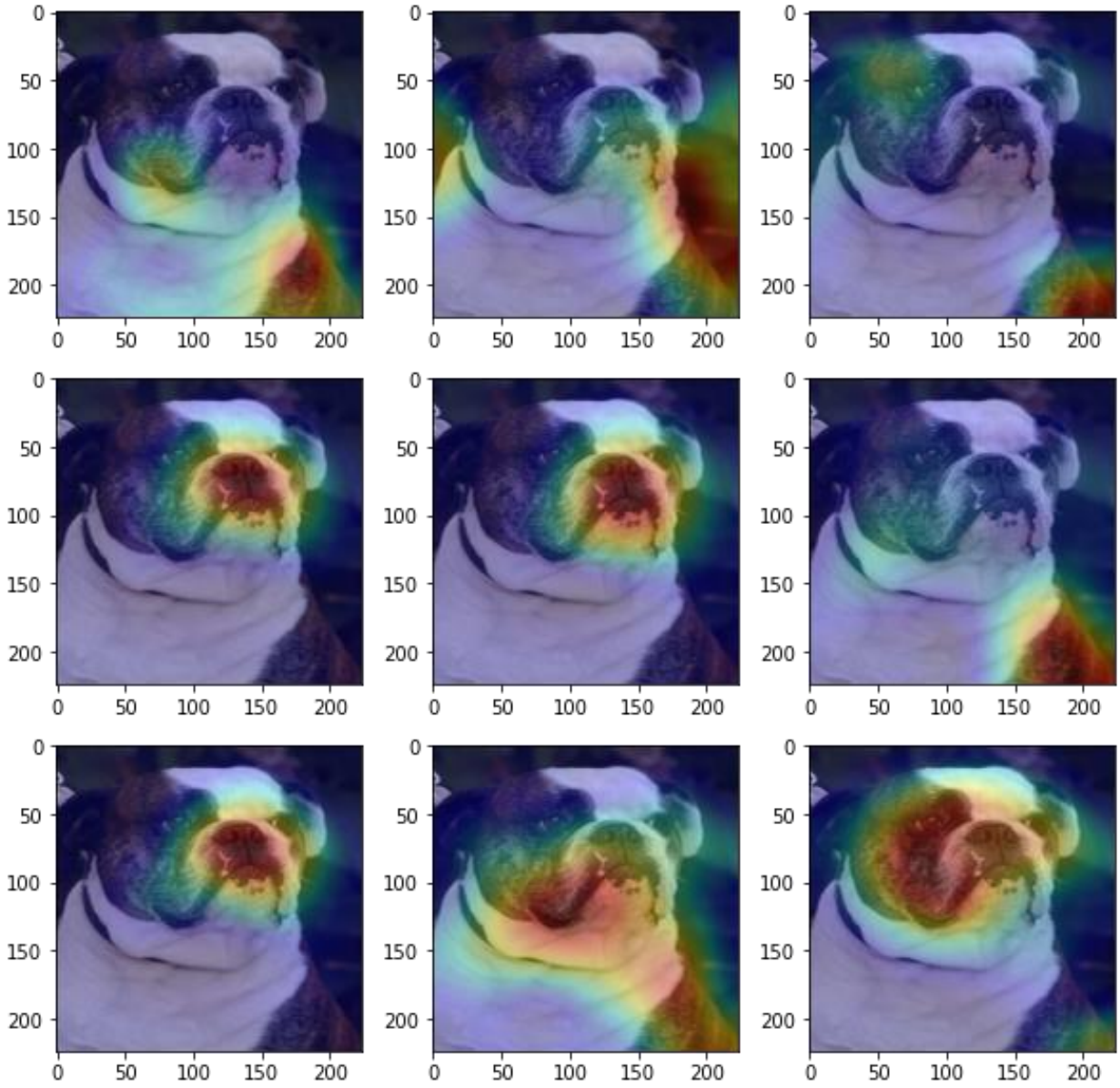


سپس هیت مپ های توجه نه فیلتر اثر گذار بر روی عکس:



در ادامه چند نمونه دیگر از این عکس ها و هیت مپ های توجه آنان را به نمایش می‌گذاریم:




0
50
100
150
200

مراجع استفاده برای حل این سوال

[1] Chollet, F. (2018). *Deep learning mit python und keras: das praxis-handbuch vom entwickler der keras-bibliothek*. MITP-Verlags GmbH & Co. KG.

[2] https://tree.rocks/get-heatmap-from-cnn-convolution-neural-network-aka-grad-cam-222e08f57a34

[3] https://colab.research.google.com/github/google/eng-edu/blob/master/ml/pc/exercises/image_classification_part1.ipynb#scrollTo=RXZT2UsyIVe_

[4] https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip