تمرین سری سوم

محاسبات نرم

استاد درس: دکتر هادی ویسی

حل تمرین: علی رحیمی

کیارش گرایلی

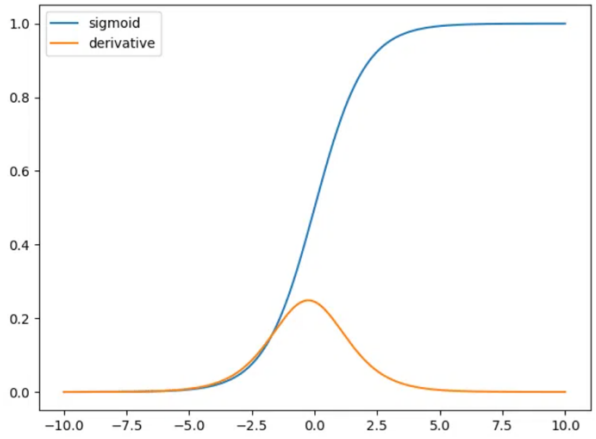
۸۳۰۴۰۰۰۴۸

پاسخ سوال اول:

قسمت اول:

در کار با شبکه های عصبی وقتی مدل خود را برای مدتی آموزش می‌دهید و به نظر می‌رسد عملکرد مدل بهبود نمی‌یابد، به احتمال زیاد مدل شما از مشکل ناپدید یا انفجاری شدن گرادیان رنج می‌برد. برای مدت طولانی، این مانع مانع بزرگی برای آموزش شبکه های عمیق بود.هنگام آموزش یک شبکه عصبی عمیق با یادگیری مبتنی بر گرادیان و پس انتشار خطا، مشتقات جزئی را با عبور از شبکه از لایه نهایی به لایه اولیه پیدا می کنیم. با استفاده از قانون زنجیره‌ای مشتق، لایه‌هایی که عمیق‌تر در شبکه هستند، ضرب‌های ماتریس پیوسته را طی می‌کنند تا مشتقات خود را محاسبه کنند.این بدان معناست که در شبکه ای از n لایه پنهان، n مشتق با هم ضرب خواهند شد. در این حالت اگر مشتقات حاصل شده مقادیر بزرگی باشند، شیب به صورت تصاعدی افزایش می‌یابد (که درواقع ما آن مقدار را در مدل منتشر می‌کنیم) تا در نهایت منفجر شوند، و این همان چیزی است که ما آن را مشکل گرادیان انفجاری می‌نامیم.

با استدلالی مشابه، اگر مشتقات کوچک باشند، شیب به صورت تصاعدی کاهش می‌یابد و مدام در مدل منتشر می‌شود تا زمانی که در نهایت ناپدید شود، و این مشکل ناپدید شدن گرادیان است. مشکل ناپدید شدن گرادیان معمولا زمانی رخ می دهد که از توابع فعال سازی Sigmoid یا Tanh در لایه پنهان استفاده می کنیم.



تابع سیگموید و گرادیان آن

چرا که این توابع مقادیری بین صفر و یک را خروجی می‌دهند که طبق استدلال فوق در هر لایه حداقل از Order صدم در صدم (ده هزارم) موجب کوچک شدن گرادیان (و وابسته به آن تغییرات وزن) می‌شود. در چنین حالتی ReLU یک تابع فعال‌سازی است که برای مشکل ناپدید شدن گرادیان توصیه شده است، اما همین تابع نیز تولید گرادیان‌های انفجاری را در صورتی که وزن‌ها به اندازه کافی بزرگ باشند، سهل می‌کند، به همین دلیل است که در صورت استفاده از ReLu وزن‌ها باید به مقادیر بسیار کوچک مقداردهی اولیه شوند تا در عین حل کردن محو شدن گرادیان، از انفجار نیز جلوگیری شود.

اگرچه استفاده از مقداردهی اولیه در ترکیب با هر نوع تابع فعال سازی Relu می تواند به طور قابل توجهی مشکل ناپدید شدن/انفجار گرادیان ها را در شروع آموزش کاهش دهد، اما تضمین نمی کند که این مشکل بعداً دوباره ظاهر نشود. برای حل چنین مشکلی راه حل های متفاوتی میتوان در نظر گرفت. یک راه ساده کاهش تعداد لایه ها است. این راه حلی است که می تواند در هر دو سناریو (شیب انفجار و ناپدید شدن) استفاده شود. با این حال، با کاهش تعداد لایه‌ها در شبکه، برخی از پیچیدگی‌های مدل‌های خود را کنار می‌گذاریم، زیرا داشتن لایه‌های بیشتر باعث می‌شود شبکه‌ها توانایی بیشتری برای یادگیری نگاشت‌های پیچیده داشته باشند.

راه حل دیگر برش گرادیان است. این راه حل بیشتر بابت مواجهه با گرادیان انفجاری به کار می‌رود. بررسی و محدود کردن اندازه شیب ها در حالی که مدل در حال یادگیری است. برش گرادیان روشی است که در آن مشتق خطا در طول انتشار به عقب در شبکه تغییر یا با یک آستانه‌ای بریده (محدود) می شود و از گرادیان های بریده شده برای به روز رسانی وزن ها استفاده می شود. همچنین با تغییر مقیاس مشتق خطا، به‌روزرسانی‌های وزن‌ها نیز مجدداً مقیاس‌گذاری می‌شوند و احتمال سرریز (overflow) یا زیرریزی (underflow) به‌طور چشمگیری کاهش می‌یابد. این کار خود معمولا با دو روش انجام می‌شود. یک حالت ایده برش ارزش است. به طوری که حداقل مقدار و حداکثر مقدار برش را تعریف می کنیم. حالت دوم ایده برش بر اساس نرم است. این ایده مشابه ایده برش ارزش با این تفاوت که در این حالت با ضرب بردار واحد گرادیان ها در آستانه، شیب‌ها را برش می دهیم.

قسمت دوم:

منظم‌سازی L1 که به‌عنوان نرم L1 یا Lasso نیز شناخته می‌شود (در مثال رگرسیون)، با کوچک کردن پارامترها به سمت 0 مشکل بیش برازش را کاهش می‌دهد. این روش درواقع معادل بی ارزش ساختن برخی از ویژگی‌ها نیز هست. به نرم L1 می‌توان به یک شکل از انتخاب ویژگی نیز نگاه کرد، زیرا وقتی یک ویژگی را با وزن 0 اختصاص می دهیم، مقادیر ویژگی را در 0 ضرب می کنیم که 0 را برمی گرداند و اهمیت آن ویژگی را از بین می برد. اگر ویژگی‌های ورودی مدل ما وزن‌های نزدیک‌تر به 0 داشته باشند، نرم L1 ما کم خواهد بود. مجموعه ای از ویژگی های ورودی که دارای وزن برابر با صفر و برای بقیه وزن‌ها غیر صفر خواهند بود.

$$Loss_{L1} = \sum_i (y_i - \sum_j x_{ij} w_j)^2 + \lambda \sum_j |w_j|$$

منظم‌سازیL2 ، یا نرم L2، یا Ridge (در مثال رگرسیون،) با اضافه کردن وزن‌ها به تابع هزینه همچنان مشابه نرم L1 با کوچک کردم وزن‌ها، با بیش برازش مبارزه می‌کند، اما آنها را دقیقاً 0 نمی‌کند. بنابراین، اگر با این روش ویژگی‌های کمتر مهم برای پیش‌بینی بر پیش‌بینی نهایی تأثیر دارد، اما تنها تأثیر کمی خواهد داشت. اجرای منظم‌سازی L2 معادل اضافه کردن مجموع مجذورات همه وزن‌های ویژگی است به تابع هزینه، لذا اگرچه برخی ممکن است به صفر نزدیک باشند اما وزن‌ها غیرصفر خواهند بود.

$$Loss_{L2} = \sum_i (y_i - \sum_j x_{ij} w_j)^2 + \lambda \sum_j w_j^2$$

همانطور که از فرمول تنظیم L1 و L2 نیز برمی‌آید، منظم سازی L1 قدر مطلق وزن را در تابع هزینه اضافه می کند، در حالی که تنظیم L2 مقدار مجذور وزن ها را در تابع هزینه اضافه می کند، لذا تفاوت شهودی اصلی بین منظم‌سازی L1 و L2 این است که تنظیم L1 سعی می‌کند میانه داده‌ها را تخمین بزند در حالی که منظم‌سازی L2 در حین محاسبه تابع loss در مرحله محاسبه گرادیان تلاش می‌کند تا loss را با کم کردن آن از میانگین توزیع داده‌ها به حداقل برساند.

زمانی که باید بین تنظیم L1 و L2 یکی را انتخاب کنید، فاکتورهای مهمی وجود دارد که باید در نظر بگیرید. مشکل عمده ای که باید در هنگام استفاده از تنظیم L2 در نظر گرفت این است که نسبت به مقادیر پرت مقاوم نیست. عبارات مجذور، تفاوت های خطای نقاط پرت را بی اثر نمی کند. حال آنکه برای lasso چنین مشکلی وجود ندارد. لذا اگر نخستین فاکتور را قوی (Robust) بودن مدل در نظر بگیرید، یک مدل در صورتی قوی است که خروجی و پیش بینی آن به طور مداوم دقیق باشد، حتی اگر یک یا چند متغیر ورودی یا مفروضات به دلیل شرایط پیش بینی نشده به شدت تغییر کند. پس طیق استدلال فوق، منظم‌سازی L1 به دلیلی نسبتاً واضح، قوی‌تر از منظم‌سازی L2 است. منظم‌سازی L2 مجذور وزن‌ها را می‌گیرد، بنابراین هزینه‌های پرت موجود در داده‌ها به‌طور تصاعدی افزایش می‌یابد. تنظیم L1 مقادیر مطلق وزن ها را می گیرد، بنابراین هزینه فقط به صورت خطی افزایش می یابد.

از طرفی دیگر، تنظیم L1 از فواصل منهتن برای رسیدن به یک نقطه استفاده می کند، بنابراین مسیرهای زیادی وجود دارد که می توان برای رسیدن به یک نقطه طی کرد. منظم‌سازی L2 از فواصل اقلیدسی استفاده می‌کند که سریع‌ترین راه را برای رسیدن به یک نقطه می‌گوید. این بدان معناست که نرم L2 فقط یک راه حل ممکن دارد. همچنین در پاسخ به سوال: کدام راه حل از نظر محاسباتی هزینه کمتری دارد؟ باید گفت از آنجایی که تنظیم L2 مجذور وزن ها را می گیرد، به عنوان یک راه حل بسته طبقه بندی می شود. از سوی دیگر نرم L1 شامل گرفتن مقادیر مطلق وزن ها است، به این معنی که راه حل یک تابع تکه ای غیر قابل تمایز است یا به زبان ساده، راه حل شکل بسته ای ندارد. منظم‌سازی L1 از نظر محاسباتی گران‌تر است، زیرا از نظر ریاضیات ماتریسی قابل حل نیست.

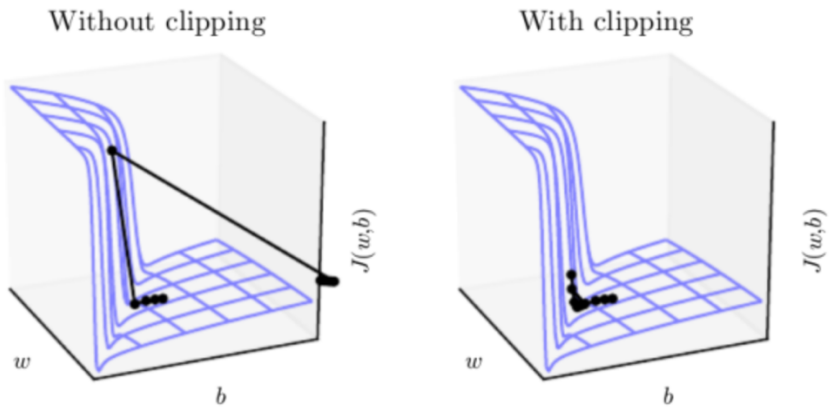
قسمت سوم:

شکست تقارن پدیده‌ای است که در واقع از علم فیزیک پدیدار شد. پدیده ای که در طی آن نوسانات (بی نهایت کوچک) بر روی سیستمی که از نقطه بحرانی عبور می‌کند، سبب می گردد تا تعیین کند که کدام شاخه از یک انشعاب در ادامه فرآیند برای آن انتخاب می‌شود. به بیان دیگر همین نوسانات بسیار کوچک سرنوشت سیستم را تعیین می‌کنند. برای یک ناظر بیرونی که از نوسانات یا نویز آگاه نیست، انتخاب مسیر سیستم، دلخواه به نظر می رسد. این فرآیند را «شکستن» تقارن می‌نامند، زیرا چنین انتقال‌هایی معمولاً سیستم را از حالت متقارن اما بی‌نظم به یک یا چند حالت معین می برند.

در علم یادگیری ماشین شکست تقارن عموما به مقدار دهی اولیه برای مدل سازی باز می‌گردد. شبکه های عصبی را در نظر بگیرید، عملیات جبری در هر نورون حاصل ضرب نقطه‌ای (ترکیب خطی) بین بردار ورودی یک لایه نرون و بردار وزن است (به انضمام بایاس) و سپس اعمال تابع فعال سازی (که برای سادگی میتوانید این حالت را عینا مشابه رگرسیون لجستیک در نظر بگیرید.) حال در چنین حالتی شبکه عصبی (و در واقع برخی از مدل‌های یادگیری ماشین) را در نظر بگیرید که دارای وزن‌هایی هستند که همگی با مقدار یکسانی مقداردهی اولیه شده‌اند. منطق شبکه و عملیات جبری که به آن اشاره شده، در این جا چند حالت را به وجود می آورد. یک حالت صفر در نظر گرفتن تمامی مقادیر هست که همانطور که قبلا اشاره شد به دلیل قانون بازنشر خطا (و وجود حاصل ضرب وزن‌ها و خروجی توابع فعال سازی) چنین مقدار دهی موجب ایجاد اختلال در به روز رسانی مقادیر وزن ها می شود و عمل یادگیری را مختل می‌سازد.

در حالت دیگر همانطور که می‌توانیم تصور کنیم اگر همه وزن‌ها یکسان باشند، همچنان اتفاقی نا خوش‌آیند برای یادگیری رخ می‌دهد. زیرا، با این شهود که هر نورون در یک لایه نشان‌دهنده یک ویژگی است، این شیوه مقدار دهی به این معنی است که افزودن نورون‌های بیشتر در یک لایه، بیانگری شبکه عصبی ما را برای هر مقدار ویژگی افزایش نمی‌دهد. گویی چنین لایه‌ای فقط یک نورون دارد. راه حل برای رفع این مشکل بسیار ساده است، فقط مقدار دهی وزن‌های اولیه را تصادفی کنید. برای مثال وزن را کاملا

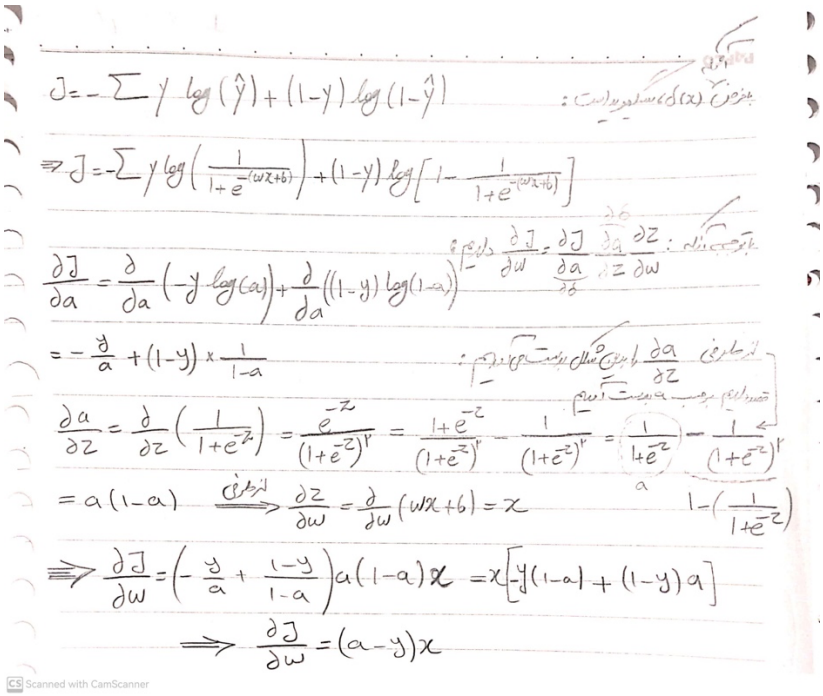
تصادفی کرده و بایاس را صفر می کنیم. به بیان دقیق‌تر مقدار دهی مدل با مقادیر تصادفی کوچک، تقارن را می شکند و به وزن های مختلف اجازه می دهد تا مستقل از یکدیگر یاد بگیرند.



اثر برش گرادیان در یک شبکه بازگشتی با دو پارامتر w و b.

همانطور که از شکل نیز بر می‌آید برش گرادیان می تواند باعث شود که نزول گرادیان در مجاورت شیب‌های بسیار تند عملکرد معقول تری داشته باشد. در سمت چپ نزول شیب بدون برش شیب از پایین این قعر (یا دره) کوچک فراتر می رود، سپس شیب بسیار بزرگی از روی صخره دریافت می کند. در حالت استفاده از تکنیک برش (که نمودار سمت راست است) فرود گرادیان با برش گرادیان واکنش متوسط تری نسبت به شیب بسیار تند (صخره مانند) دارد. در حالی که از صخره بالا می رود، اندازه گام محدود است به طوری که نمی توان آن را از منطقه شیب دار نزدیک دور کرد.

قسمت چهارم:



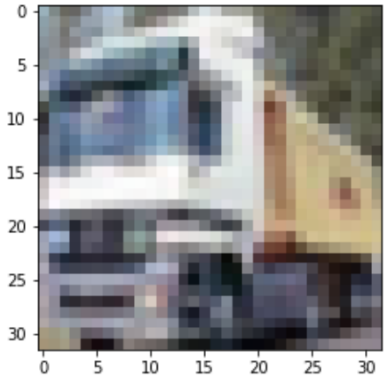
پاسخ سوال دوم:

توضیحات کلی در مورد الگوریتم های نوشته شده برای سوال دوم:

با توجه به آنکه اجرای الگوریتم مدت زمان زیادی را بر روی سیستم شخصی‌ام اخذ می‌کرد، تصمیم بر آن شد تا مدل مذکور را بر روی پلتفرم google colab پیاده‌سازی کنم. برای پیاده سازی نیز اجرای الگوریتم‌ها را برروی GPU اجرا کرده‌ام. برای افزایش سرعت همچنین کدهای پیاده سازی شده را با GPU اجرا کردم. لذا جهت اجرای برنامه کافی است که کد داده شده را در محیط colab بارگذاری شود. در همین راستا برای وارد کردن داده در سوال دوم به علت راحتی کار از داده cifar 10 موجود در keras.dataset که به‌طور درون ساختی در درون برنامه وجود دارد استفاده کرده‌ام. ناگفته نماند که با چک کردن اسناد هر دو سری داده مطمئن شدم که جزئیات داده ها عینا یکسان بوده است.

در بخش نخست الگوریتم، ابتدا کد را بر روی پردازنده گرافیکی تنظیم کرده و پس از خواندن داده‌ها جهت اطمینان دستگاه پردازنده گرافیکی و یک نمونه داده را خروجی میگیریم:

```
GPU device name is: /device:GPU:0

Sample image:
<matplotlib.image.AxesImage at 0x7f5f5a266250>
```



سپس طبق روال سایر تمرینات داده‌ها را خوانده و پس از نرمال کردن داده‌ها سایز داده‌های ورودی و خروجی را خروجی گرفته و مقدار y را نیز چاپ میکنیم:

```
Input size is: (50000, 32, 32, 3)
output size is: (50000, 1)

output would be like:
array([[6.],
       [9.],
       [9.],
       ...,
       [9.],
       [1.],
       [1.]], dtype=float16)
```

پیاده سازی الگوریتم: مطابق آنچه در صورت سوال گفته شده است در این تمرین قصد داریم تا با ساختن یک شبکه عصبی پیچشی (CNN)چند لایه، به هدف طبقه بندی عکس ها را روی داده CIFAR−10 برسیم. بخش اول تمرین تا قسمت "خ" شامل پیاده سازی مدل های مختلف برای هدف ذکر شده می باشد. برای این کار از روش برنامه نویسی شی‌گرا برای این کار استفاده کردیم. کلاسی

مرتبط، برای ساخت مدل های متفاوت را نوشتیم. این کلاس دارای سه بخش اصلی می باشد بخش اول که بخش مقدار دهی اولیه است، پیاده سازی ساختار کلی مدل شبکه عصبی بوده و بخش دوم و سوم هر کدام در واقع توابعی برای Compile مدل و یادگیری (یا fit کردن) مدل می باشند. این مدل را از آن رو بدین شکل پیاده سازی کرده ایم که بتوان برای پارامترهای مختلف، با حجم کدی کمتر، و تنها کار بر روی یک کلاس برای مدل مذکور کار بکنیم. لذا در هر بخش تنها با تغییر دادن پارامترهای هر مدل ذکر شده (در صورت تمرین) را خروجی می دهیم.درنتیجه، بخش چهارم (یا بخش نهایی کلاس) برای مدل کلی، مربوط به خروجی مدل خاص ساخته شده بر مبنای مقادیر خواسته شده است. این مقادیر شامل خطای داده یادگیری، خطایValidation ، دقت و همچنین مدت زمان اجرای الگوریتم و سرعت یادگیری مدل می باشد. در نهایت نیز با اجرای کلاس ساخته شده و مقداردهی پارامترهای مرتبط با هر مدل در هر بخش خواسته شده، خروجی را به دست آورده و خروجی مورد نظر را در همین گزارش برای تحلیل جنبه های مختلف آن ارائه کردیم.

بخش اول کد همان بخش ساخت مدل است:

```
class CNN_Model(Model):

  def __init__(self, x_train, y_train,
               activation_function = 'sigmoid',
               kernel1_size = (3, 3),
               kernel2_size = (3, 3),
               conv_layer_size = 2,
               one_layer = False,
               normalization = False):

    super(CNN_Model, self).__init__()
    #initializing values
    self.x_train = x_train
    self.y_train = y_train
    self.activation_function = activation_function
    self.kernel1_size = kernel1_size
    self.kernel2_size = kernel2_size
    self.conv_layer_size = conv_layer_size
    self.one_layer = one_layer
    self.normalization = normalization

    #building the network
    self.model = tf.keras.Sequential()
```

در این بخش مطابق پارامترهای ذکر شده در صورت تمرین مدل پیش فرض ساختیم. نخست لایهی پیچشی با اندازهی کرنل ۳×۳، تعداد کانال‌های خروجی: ۷ و تابع فعالسازی ReLU سپس لایه‌ای پیچشی با اندازهی کرنل: ۳×۳ ، تعداد کانال‌های خروجی: ۹ و تابع فعالسازی ReLU به همراه لایه‌ای ادغام بیشینه و اندازهی کرنل: ۲×۲ در بخش انتهایی نیز لایه drop_outبا احتمال ۳۰ درصد ۵که این مجموعه در واقع بخش استخراج ویژگی مدل می‌شود. در انتها نیز لایه دسته بندی را با لایه خطی باتعداد ده (تعداددسته‌ها) نرون خروجی می‌سازیم.

در نهایت نیز برای General بودن مدل ۳ پارامتر دیگر را تعریف میکنیم. نخست پارامتر Conv_layer_size که برای بخش الف به کار می‌رود. این پارامتر در صورتی که عدد ۳ یا ۴ به آن تعلق بگیرد، مطابق صورت سوال ۳ و ۴ لایه پیچشی می‌سازد. در حالت پیشی فرض نیز، این پارامتر مقدار ۲ را دارد که مدل پیش فرض ما ساخته خواهد شد. پارامتر بعدی One_layer است.این پارامتر نیز نیز حالت تک لایه (حالت پ در صورت سوال) بودن مدل را نشان میدهد که در صورت True بودن آن مدل تنها یک لایه پیچشی و در غیر این صورت

مدل پیش فرض ساخته خواهد شد. و در حالت نهایی نیز برای مدیریت Batch Normalization پارامتر normalization را تعریف میکنیم که مشابه قبل دو حالت دارد.

با استفاده از توضیحات بالا مدل را می سازیم.

```
#in case batch normalization is needed we should change first layer to a normalization layer
if self.normalization == True:
  self.model.add(keras.layers.SyncBatchNormalization())

#if batch normalization is not needed, creating the default model with dynamic first kernel size
else:
  self.model.add(keras.layers.Conv2D(7, self.kernel1_size, input_shape = (32,32,3), padding="same", activation = activation_function))

if self.one_layer == False:
  #after creating first layer there are three possibilities: 1, 2 or 3 conv layers remaining to build
  #keep in mind that we have already build the first con layer for first layer in default model
  if self.conv_layer_size > 2:
    for i in range(self.conv_layer_size - 2):
          self.model.add(keras.layers.Conv2D(7, (3, 3), activation = activation_function))

  #here is the default model with assumption of con_layer_size == 1, second
  if self.activation_function == 'leaky_relu':
    self.model.add(layers.LeakyReLU(alpha=0.3))
    self.model.add(keras.layers.Conv2D(9, self.kernel2_size))
  else:
    self.model.add(keras.layers.Conv2D(9, self.kernel2_size, activation = activation_function))
elif self.one_layer == True:
     self.model.add(keras.layers.MaxPooling2D(pool_size=(5, 5)))

self.model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))
self.model.add(keras.layers.Flatten())
self.model.add(keras.layers.Dropout(0.3))
self.model.add(keras.layers.Dense(10, activation="softmax"))
```

در این بخش مدیریت حالت های گوناگون را برای ساخت مدل با استفاده از حالتهای شرطی انجام می‌دهیم. در پایان این بخش، بخش اول مدل تمام می شود.

بخش دوم کد شامل تابع comp است . این تابع عینا مشابه تابع compile برای خود مدل تنسور می‌باشد. منتهی با این فرق که توان تغییر Optimizer و نوع Loss و نرخ یادگیری را داشته باشیم.

```
def comp(self, optimizer = 'adam', loss_s = 'sparse_categorical_crossentropy', learn_rate = 0.01):
  # training the model
  if optimizer == 'sgd':
    self.model.compile(loss = loss_s,
                       optimizer = keras.optimizers.SGD(learning_rate = learn_rate),
                       metrics =['accuracy'])
  else:
    self.model.compile(loss = loss_s,
    optimizer = keras.optimizers.Adam(lr = learn_rate),
    metrics =['accuracy'])
```

در این بخش مطابق مدل پیش فرض بهینه‌ساز Adam، تابع خطای categorical cross entropyو نرخ یادگیری یک صدم آموزش دهید با اندازه دسته ۳۲ را به عنوان مقادیر پیش فرض به تابع میدهیم. نکته مهم در این بخش تعیین مقدار Metrics برای بخش خاتمه یافتن الگوریتم (Early Stopping) می‌باشد.

بخش سوم کد شامل تابع fit است . این بخ هم مطابق بخش قبل می‌باشد. در این بخش نیز تابع Fit را مطابق تعریف با توانایی تغییر پارامترها برای مدلمان مینویسیم.

```
def fit(self, batch_s = 32, Epoch = 50, validation_perc = 0.15):

  callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience = 3)

  self.tic = time.time()

  self.history = self.model.fit(self.x_train, self.y_train,
  batch_size = batch_s,
  validation_split = validation_perc,
  epochs = Epoch,
  callbacks=[callback],
  shuffle = True)

  self.toc = time.time()
```

بخش آخر کد مربوط به خروجی دادن مدل است:

```
def output(self, model):
  plt.plot(self.history.history["loss"], label="Train_Loss for model:" + model)
  plt.plot(self.history.history["val_loss"], label="Valid_Loss for model:" + model)
  plt.legend()

  print()

  print('this model takes: '+ str(round(self.toc - self.tic, 4)) +' time')
```

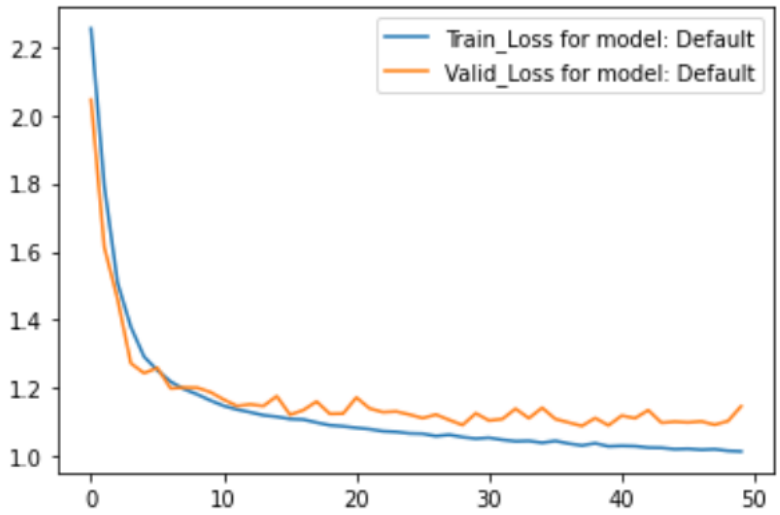
در این بخش با گزارش نموداری از روند تغییرات خطای شبکه در حین آموزش، بر روی دادههای train و validation این دو نمودار را روی یکدیگر قرار می دهیم. همچنین دقت مدل بر روی دادههای Validation و میانگین مدت زمان اجرای Epoch را برای تحلیل های آتی خروجی می دهیم.

مدل ها: تمامی مدل‌ها را به شکل زیر و تنها با تعیین پارامتر های مختص به هر مدل پیاده سازی میکنیم.

```
d_model = CNN_Model(x_train, y_train)
d_model.comp()
d_model.fit()
d_model.output(' Default')
```

نخست مدل پیش‌فرض را پیاده سازی میکنیم و خروجی‌های مربوطه را چاپ میکنیم:

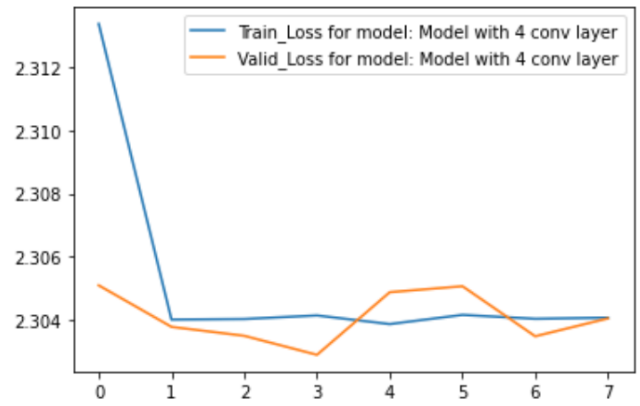
```
this model takes: 323.7041 time
```



مدل در طی ۵۰ دور اجرا (یا ۵۰ عدد Epoch) به نظر شروع به همگرایی کرده و با توجه به اختلاف دقت و مقدار loss به نظر می‌رسد در بلند مدت مدل در حال بیش برازش شدن است. گرچه چون تعداد تکرار ها همچنان متنهای است امکان نظر دهی قطعی وجود ندارد. مدت زمان الگوریتم نسبتا زیاد بوده که می‌تواند حاصل میزان نرخ یادگیر و سایز Batch باشد.

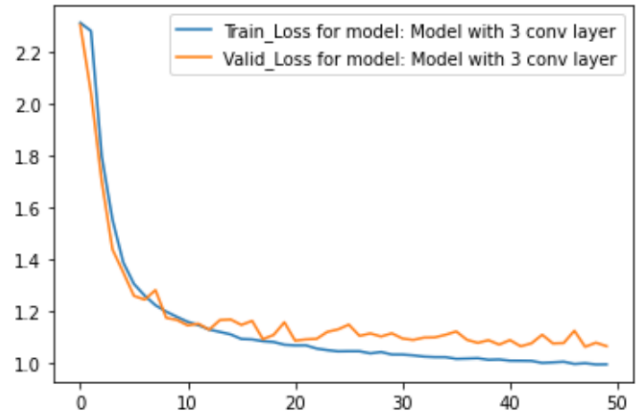
برای بخش الف، تعداد لایه‌های پیچشی را از دو لایه به سه و چهار لایه (با اندازه کرنل ۳×۳) تغییر می‌دهیم.

```
this model takes: 50.0662 time
```



برای مدل با چهار لایه پیچشی، مدل خطای خیلی زیادی داشته و هممچنین خیلی سریع واگرا می‌شود. مدل در تعداد ۷ تکرار به حالت اتمام خاتمه با patience برابر ۳ می‌رسد. لذا مدل بر مبنای میزان خطا و دقت اصلا عمل یادگیری ندارد.

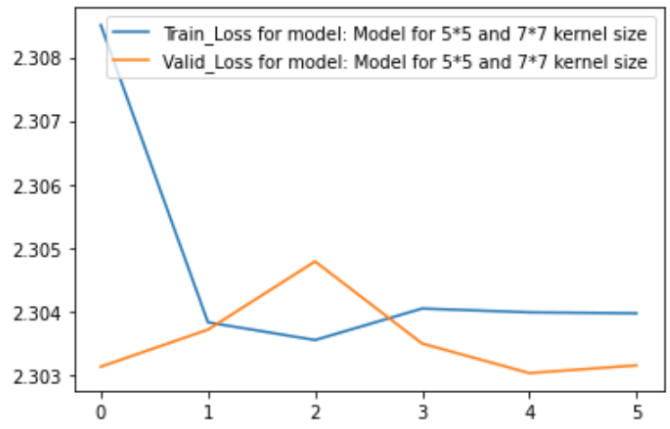
```
this model takes: 294.4361 time
```



مدل برای حالت ۳ لایه پیچشی بر خلاف مدل ۴ لایه عمل یادگیری را به طور قابل تاملی بهبود میدهد. همچنین به نسبت مدل پیش فرض در طی ۵۰ تکرار، با زمان کمتری به خطا و دقت بهتری نیز می‌رسد. در ضمن این مدل برای تعداد یکسان تکرار زمان کمتری را برای یادگیری اختصاص می‌دهد.

در مدل بعدی به تغییر اندازه‌ی کرنل لایه اول پیچشی به ۵×۵ و ۷×۷ می‌پردازیم.

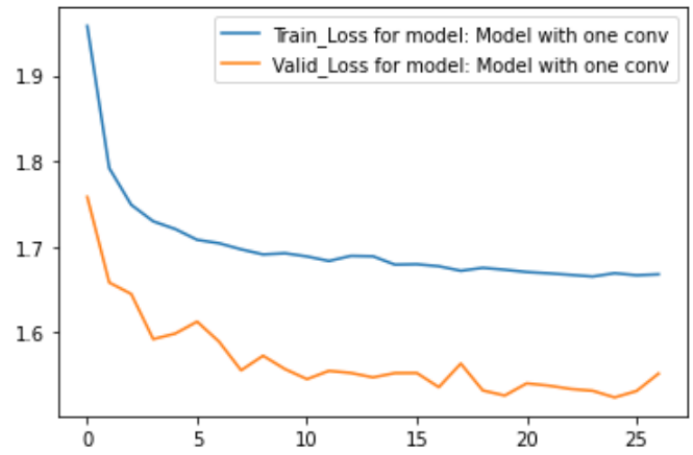
```
this model takes: 34.2205 time
```



این مدل نیز بسیار مشابه حالت ۴ لایه پیچشی می‌باشد. مدل خطای بالایی داشته ودقت کم که حاصل از یاد نگرفتن الگو و واگرایی مدل می باشد.

در مدل بعدی با استفاده از یک لایهی پیچشی با اندازهی کرنل ۵×۵ به جای استفاده از لایه‌های اول و دوم مدل بعدی را می‌سازیم.

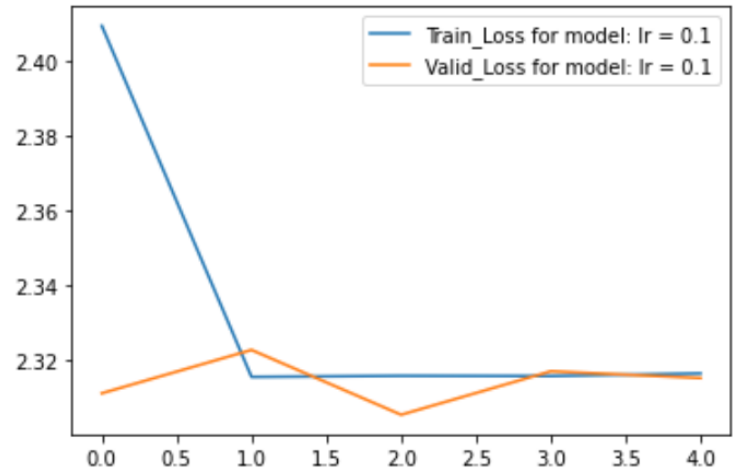
```
this model takes: 128.5305 time
```



در این حالت گرچه نسبت به مدل قبل بهبود وجود دارد اما همچنان مدل در عمل یادگیری (مخصوصا به تک نرخ بودن و عملا ثابت بودن تغییرات خطا) در طی چند تکرار آخر، عملا هیچ پیشرفتی ندارد و یادگیری الگو در نهایت واگرا می‌شود.

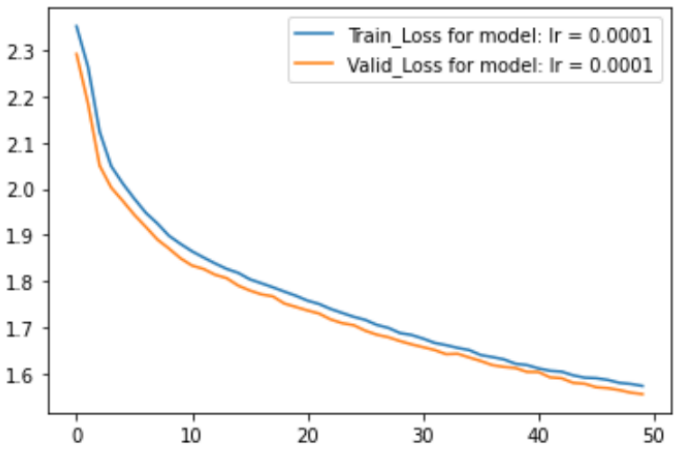
سه مدل بعدی مربوط به تغییر ضریب یادگیری بهینه‌سازی می‌باشد.

```
this model takes: 27.4661 time
```

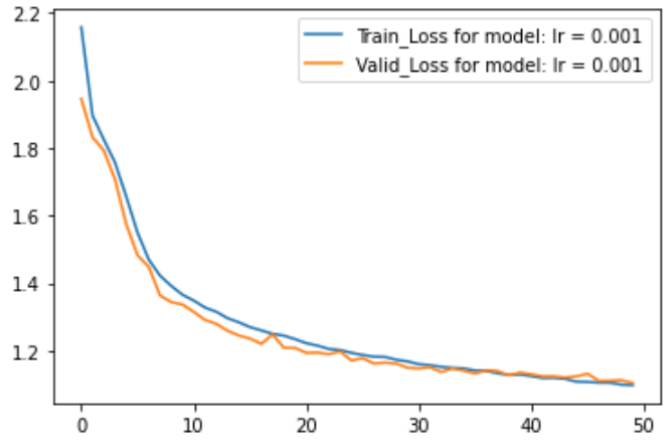


این مدل نیز مشابه مدل با چهار لایه در عمل یادگیری ناتوان بوده و خطا و دقت بسیار بدی دارد. این موضوع البته با توجه به نرخ بالای یادگیری تقریبا مطابق انتظار است. چرا که مقدار ۰.۱ برای این حجم از داده تصویری به نظر اصلا انتخاب مناسبی برای همگرا شدن به نقطه بهینه نمی‌باشد.

```
this model takes: 270.3291 time
```




```
this model takes: 323.3576 time
```

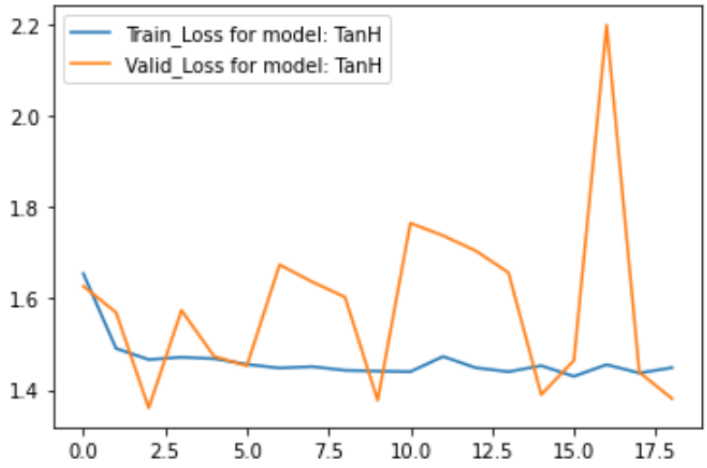


دو مدل بعدی مطابق انتظار همگرا شده و البته سرعت همگرایی بسیار پایینی نیز دارند. در این دو مدل بدیهی است ک هبا توجه به نرخ عمومی نزولی خطا و افزایش دقت، به احتمال بسیار قوی مدل در طی تکرار های آتی به مقادیر بهتر و بهتری برای خطا و دقت خواهند رسید (گه مشخصا این گزاره هم الراما برقرار نیست) چرا که نزخ کاهش شیب مناسبی داشته و همچنین مدل همگرا شده است. اما با دقت بیشتر مشاهده میشود که مقدار بیش از اندازه کوچک نرخ یادگیری نیز، با بسیار کم کردن سرعت یادگیری عملا در یادگیری الگو نسبت به زمان خیلی عملکرد ضعیفی دارد (به همین دلیل هم شیب کمتری دارد) و لذا خطای بالاتری در تشخیص الگو دارد و همچنین زمان کمتری هم میگیرد که نکته قابل تاملی است.

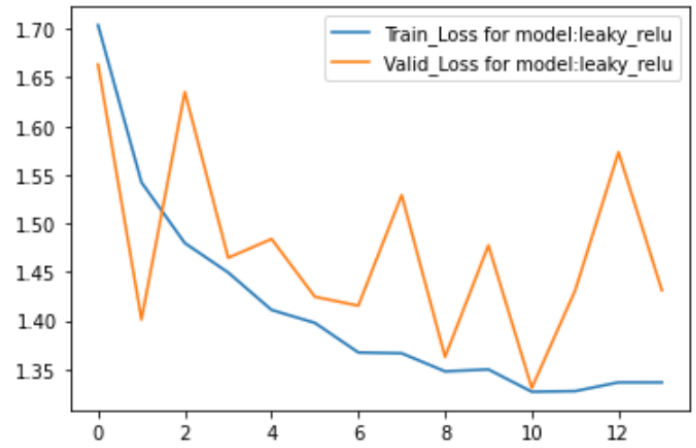
نکته مهم دیگر در این الگو کمتر بودن دایمی مقدار خطای Validation به طور کامل (در یک بازه تکرار الگوریتم) است. این موضوع البته (با توجه به پایین بودن نرخ یادگیری) می‌تواند در بلند مدت (با تکرار بیشتر) الزاما برقرار نباشد. منتهی (با فرض تحلیل در همین تعداد تکرار الگوریتم) این اتفاق می‌تواند به چند عامل بازگردد. نخست آنکه حجم داده تست (خوانده شود Validation) روی الگوریتم با توجه به آنکه خروجی گرفتن خطا پس از یادگیری اتفاق می‌افتد، تاثیر گذار است، چرا که حجم داده یادگیری بیشتر، منجر به یادگیری الگوی به نسبت خوبی شده و لذا دایما خطای روی تست را کم نشان می‌دهد. لذا تغییر پارامتر حجم داده تست می‌تواند تاثیرگذار باشد. از طرف دیگر ساده بودن الگوهای تست و یا پیچیده بودن الگوهای داده اموزشی نیز قطعا تاثیر یکسانی را در این راستا می‌گذارد.

در بخش تغییر تابع فعالساز به leaky ReLUو tanh مدل ها را برای هر کدام خروجی می‌دهیم.

```
this model takes: 102.4109 time
```



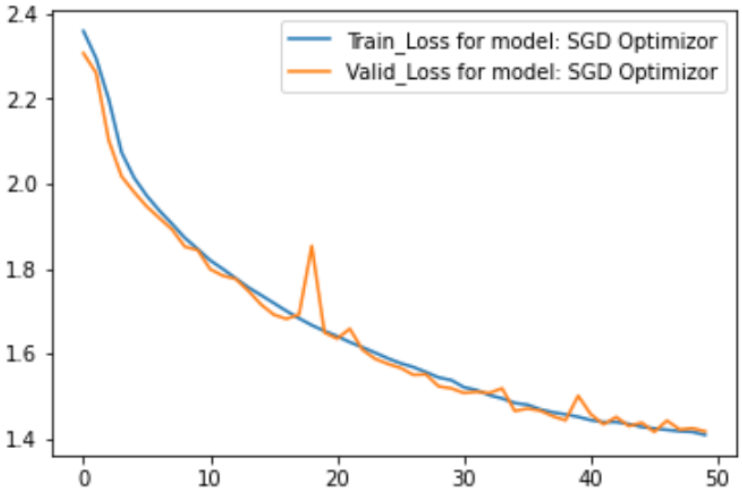

```
this model takes: 75.2734 time
```



مشابه استدلال های فوق هر دو مدل در یادگیری الگو عملکرد خیلی خوبی نخواهند داشت و در تعداد تکرار کمی با توجه به تغییرات خطا و افزایش نسبی آن‌ها متوقف می‌شوند. در این بین عملکرد Leaky Relu به مراتب بهتر بوده، هم ازین جهت که روند کاهش خطا کاهشی بوده و هم مقدار خطا و دقت بهتری دارد. نکته قابل توجه در این الگوها تغییر مداوم خطا می‌باشد. ای موضوع یعنی وابستگی مدل به داده‌ها افزایش یافته و با کارهایی مثل افزایش مقدار Batch و یا بهینه ساز SGD می‌تواند خیلی کمک کننده باشد. هر دو مدل حجم زمانی کمی میگیرند که البته وابسته به تعداد کم تکرار های الگوریتم است و نکته مثبتی الزاما به حساب نمی‌آید.

تغییر بهینه‌ساز به SGD کار بعدی برای مدل است.

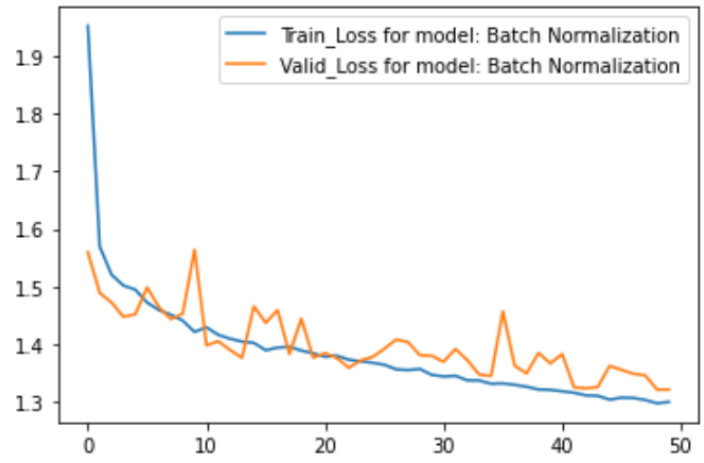
```
this model takes: 263.2935 time
```



مدل حاصل همگرا با زمان مناسب و البته مقدار خطای نسبتا قابل قبولی است. این مدل نیز در بین مدل‌های بدست آمده جزو مدل های مناسب به حساب نمی‌آید چرا که اگرچه در ابتدای کار خطا به حجم خوبی کاهش می‌یابد اما روند یادگیری با افزایش تکرار کند و کند تر شده و به مقدار مناسبی نمی‌رسد (چرا که دقت زیر ۵۰ درصد عملا از مقدار تصادفی دو حالته نیز بدتر بدست می آید.)

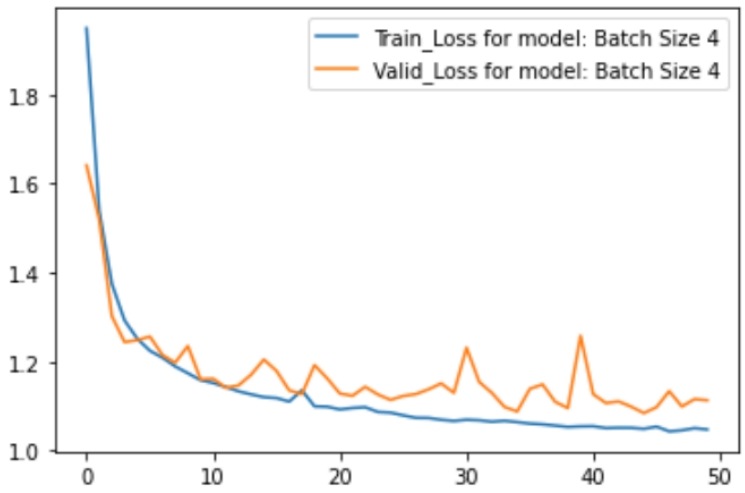
در مدلی با اضافه کردن لایه‌ی Batch Normalization به عنوان لایه اول شبکه، مشابه مدل قبلی کاهش ارور در ابتدا شیب خوبی دارد اما در امتداد یادگیری الگوریتم خطا همچنان مقدار بالایی دارد. گرچه خطای تست افت و خیز بالایی دارد.

```
this model takes: 263.4504 time
```



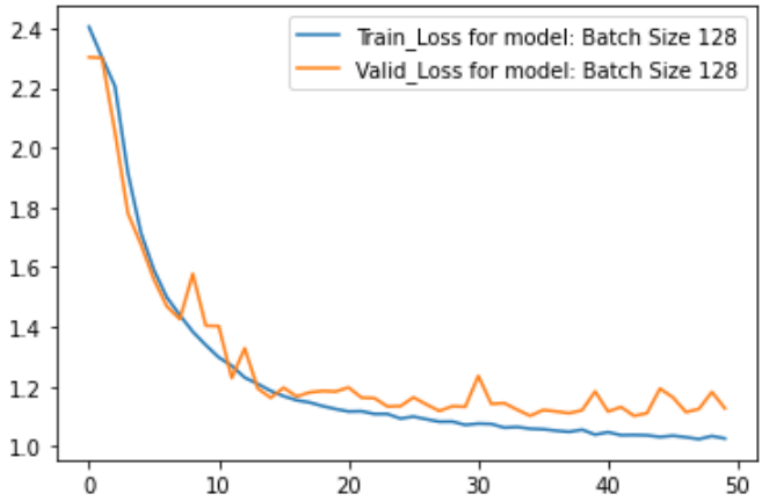
بخش بعدی تغییر اندازهی batch_sizeبه 4 و ۱۲۸ میباشد:

```
this model takes: 1601.7625 time
```



در سایز چهار، الگوریتم یادگیری کندی دارد چرا که با تعداد کمتری داده باید عمل به روز رسانی را برای حجم بالایی داده انجام دهد.
خطا برای این مدل کم و دقت به نسبت بالا بوده و با توجه به شیب کم شدن خطا انتظار آن میرود که مدل الگوهای بهتری را یاد بگیرد (تقریبا مشابه مقدار نرخ یادگیری کم.)

```
this model takes: 143.7253 time
```



در مدل با سایز ۱۲۸ نکته مهم در سرعت بالایی همگرایی شبکه است، اما از آنجایی که حجم بالایی داده را برای به روز رسانی وزن ها در نظر گرفته میشود، مدل برای یادگیری الگو جزییات کمتر (خوانده شود پیچیدگی کمتر) را فرا می‌گیرد و لذا در بلند مدت خطای بالاتری دارد.

مدل ها	کمینه خطای یادگیری	کمینه خطای Validation	تعداد Epoch	مدت زمان یادگیری	بیشینه دقت مدل
مدل پایه	۱.۰۱	۱.۰۹	۵۰	۳۲۳	۶۲٪
۳ و ۴ لایه	(۲/۴,۰.۹)	(۲.۳,۱.۰۶)	(۸,۵۰)	(۵۰,۲۹۴)	(۱٪,۶۲٪)
تغییر سایز کرنل	۲.۳	۲.۳	۶	۳۴	۱٪
تک لایه پیچشی	۱/۶	۱/۵	۲۷	۱۲۸	۴۵٪
نرخ یادگیری ۰.۱ و ۰.۰۰۱ و ۰.۰۰۰۱	(۱/۵,۱.۰۹,۲.۳)	(۱/۵,۱.۱,۲.۳)	(۵۰,۵۰,۵)	(۲۷۰,۳۲۳,۲۷)	(۴۴٪,۶۱٪,۱٪)
تابع فعال سازی TanH و Leaky ReLu	(۱.۳,۱/۴)	(۱.۳,۱.۳)	(۱۴,۱۹)	(۷۵,۱۰۲)	(۵۵٪,۵۲٪)
بهینه ساز SGD	۱/۴	۱/۴	۵۰	۲۶۳	۴۹٪
Batch Normalization	۱.۳	۱.۳	۵۰	۲۶۳	۵۷٪
اندازه Batch برابر ۴ و ۱۲۸	(۱.۰۲,۱.۰۴)	(۱.۱,۱.۰۹)	(۵۰,۵۰)	(۱۴۳,۱۶۰۱)	(۶۲٪,۶۱٪)
مدل MobileNetV2	۰.۹	۱/۵	۲۰*	۵۶۶	۵۵٪

بررسی پارامترهای هر مدل بر حسب مدل‌های خواسته شده: برای مدل‌هایی با چند پارامتر، اعداد از راست به چپ خوانده شوند (*در مدل آخر ماکسیمم تعداد تکرار مقدار ۲۰ بوده است)

در بخش خ، اشکال خواسته شده را با سایز کرنل مدل که شامل هفت فیلتر با ۷ سایز کرنل یکسان است پیاده سازی می کنیم. و خروجی را می بینیم. در نظر بگیرید که این بخش را با بهترین پارامترها (مدل b_model) پیاده سازی کردیم (در هر دو بخش کرنل و خروجی عکس از کرنل مرتبط، عکس اول مربوط به b_model با بهترین پارامترها و عکس دوم مربوط به بهترین مدل¯حدودی¯ در بین مدل‌های خواسته شده است. ناگفته نماند که در لحظه ارسال کد b_modelمربوط به بهترین مدل از بین مدل های خواسته شده است.)

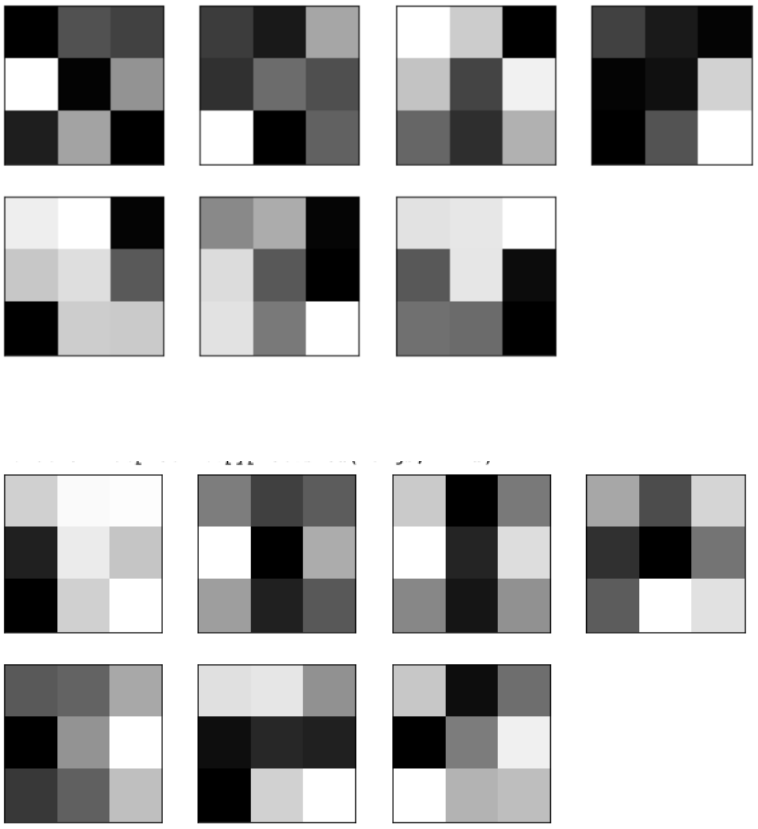
لذا برای خروجی کرنل ها کد فوق را پیاده سازی کرده که به سادگی فیلتر ها را نشان می‌دهد:

```
fig = plt.figure(figsize = (10,10))

col = 4
row = 4

n_filters = 7

for i in range(1, n_filters + 1):
  f = filters[:,:,:,i-1]
  fig = plt.subplot(row,col,i)
  fig.set_xticks([])
  fig.set_yticks([])
  plt.imshow(f[:,:,0], cmap = 'gray')
plt.show
```

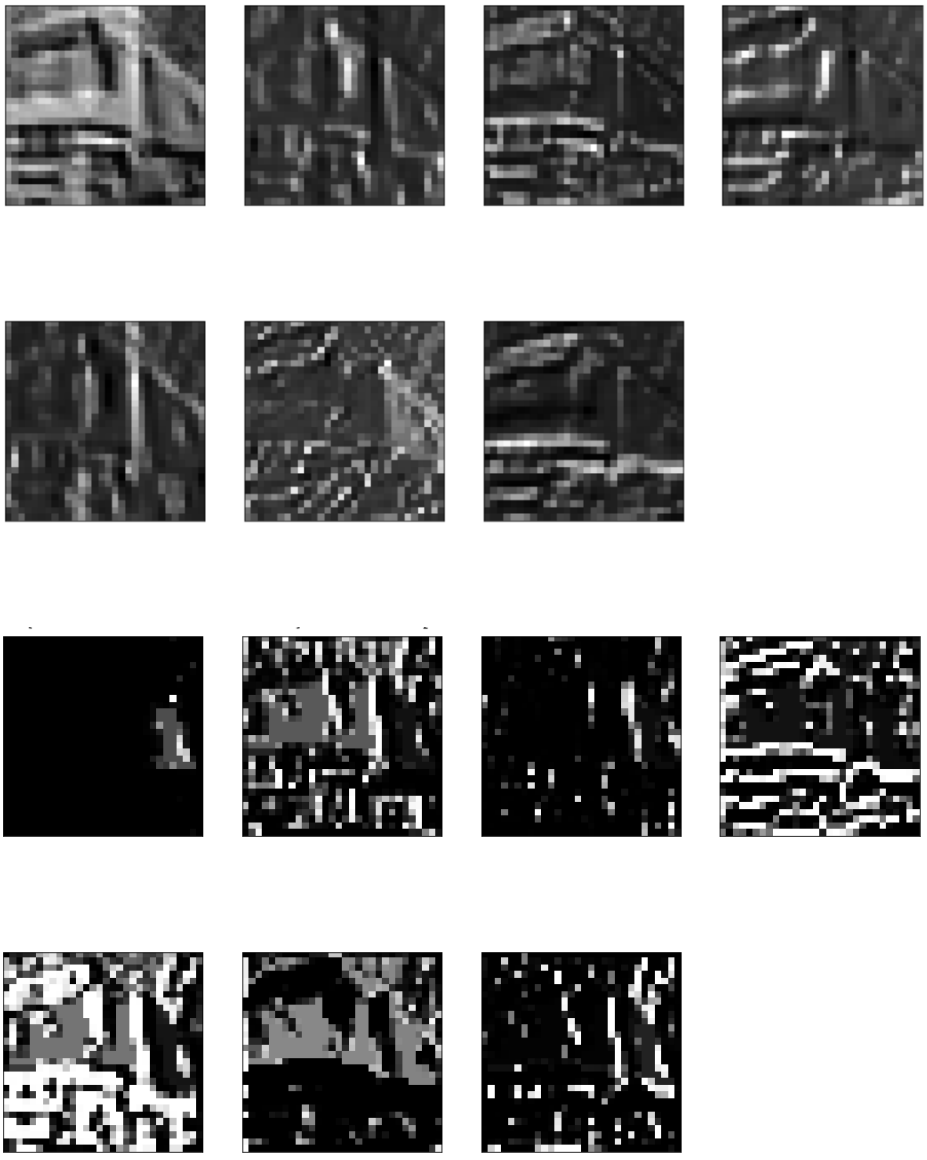
و برای گذراندن عکس از لایه اول نیز، ابتدا مدلی برای لایه اول به طور مجزا درست کرده و عکس را برای هر فیلتر با استفاده از مدل عبور می‌دهیم(به فرم زیر:)

```
out = [best_model.model.layers[1].output]

firt_layer_model = Model(inputs = best_model.model.inputs, outputs = out)

im = np.expand_dims(image, axis = 0)

predicted_image = firt_layer_model.predict(im)

col = 4
row = 3

for j in predicted_image:
  fig = plt.figure(figsize = (16,16))
  for i in range(1, n_filters + 1):

    fig = plt.subplot(row,col,i)
    fig.set_xticks([])
    fig.set_yticks([])
    plt.imshow(j[:,:,i-1], cmap = 'gray')
  plt.show
```

برای وزن کرنل‌ها اشکال زیر را میبینیم:



مطابق شکل که در صورت سوال خواسته شده (و البته خروجی الگوریتم ما نیز خوانش یکسانی با آنها دارد.) اما از آنجایی که تصاویر ابعداد زیادتری نسبت به ابعاد کرنل ها دارند، و ما سایز کرنل را کوچک در نظر گرفته‌ایم، لذا از روی سایز کرنل نمی توان الگوی دقیقی را برای یادگیری عکس های متفاوت در نظر داشت. اما با عبور دادن عکس از همین فیلتر ها به راحتی می توانیم ببینیم که به طور حدودی هر کدام از فیلتر ها عکس چه چیزهایی در عکس را مورد وقت قرار می دهند



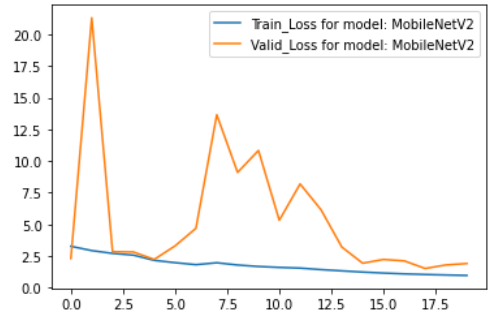
برای عکس سری اول که به شکل واضح‌تری عکس مورد نظر را نشان می‌دهد (چرا که مدل اساسا مدل بهتری نیز است،) در خروجی عکس از کرنل (اول) گویی رنگ های روشن و یا نور بازتاب مهمی دارند و در مثال عکس سوم گویی خطوط عمودی و خط یکی مانده به آخر خطوط اریب مد نظر کرنل قرار میگیرد.

در بخش پایانی برای مدل MobileNetV2 با استفاده از مدل ذکر شده، داده های ورودی خود را به این مدل می دهیم با توجه به اینکه مدل مذکور روی داده های دیگری یادگیری را انجام داده است و عملکرد بسیار خوبی روی آن داده‌ها دارد انتظار آن می‌رود که مدل بایاس بالایی نسبت به داده داشته باشد.

```python
from tensorflow.keras.applications import MobileNetV2

pre_trained_model = MobileNetV2(weights='imagenet', include_top = False)
flatten_layer = tf.keras.layers.Flatten()
dense_layer = tf.keras.layers.Dense(10, activation='softmax')

input_images = tf.keras.Input(shape=(32, 32, 3), name='input_image')
features = pre_trained_model(input_images)
flatten_features = flatten_layer(features)
final_outputs = dense_layer(flatten_features)

model = tf.keras.Model(inputs = input_images, outputs = final_outputs)

model.compile(loss = 'sparse_categorical_crossentropy',
      optimizer = keras.optimizers.Adam(lr = 0.01),
      metrics =['accuracy'])

callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=3)

tic = time.time()

history = model.fit(x_train, y_train,
                    batch_size = 32,
                    validation_split = 0.15,
                    epochs = 20,
                    callbacks=[callback],
                    shuffle = True)
toc = time.time()

plt.plot(history.history["loss"], label="Train_Loss for model:" + ' MobileNetV2')
plt.plot(history.history["val_loss"], label="Valid_Loss for model:" + ' MobileNetV2')
plt.legend()

print()

print('this model takes: '+ str(round(toc - tic, 4)) +' time')
```

چرا که داده ای که اکنون با آن می دهیم با داده که قبل عمل یادگیری را در تهران فرا گرفته متفاوت است لذا انتظار می رود که مدل در طی تکرار های کم درصد خطای نسبتاً بالایی با میزان دقت نسبتاً پایینی را داشته باشد حال آنکه متغیر مدرن و طولانی مدت به اجرا درآوردن آن احتمال بالای وجود دارد که مدل به شکلی بسیار پیشرفته عمل یادگیری را انجام دهد.

```
this model takes: 566.6014 time
```



خطا مطابق انتظار بالاست و نکته مهم دقیقا نشان دهنده استدلال ذکر شده است. چرا که مدل با فیت شدن بر روی داده دارای خطای یادگیری ثابتی است، منتهی چون با داده های دیگری فیت شده است دارای خطای تست بسیار زیاد بالایی است.