دانشگاه صنعتی امیرکبیر

(پلی تکنیک تهران)

دانشکده مهندسی برق

گزارش‌کار پروژه نهایی

مقدمه ای بر هوش محاسباتی

پردازش سیگنال

نگارش

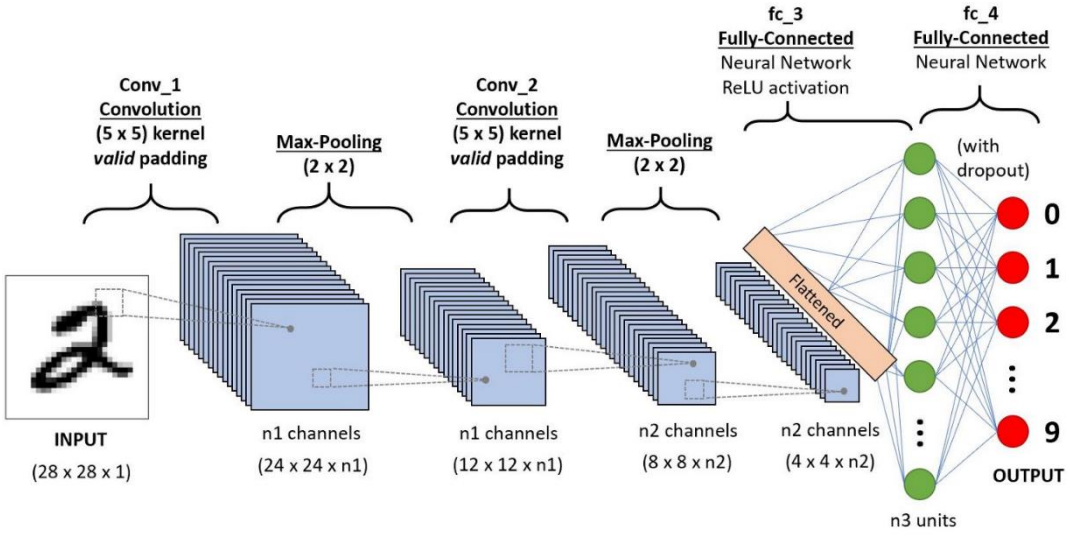
علی بابالو

پویا شریفی

پیش گزارش

2-1- شبکه های CNN

در یادگیری عمیق، شبکه عصبی کانولوشنال (CNN یا ConvNet) یک کلاس از شبکه عصبی مصنوعی (ANN) است که بیشتر برای تجزیه و تحلیل تصاویر بصری استفاده می شود. CNN ها همچنین به عنوان شبکه های عصبی مصنوعی تغییر ناپذیر(SIANN) شناخته می شوند که بر اساس معماری وزن مشترک هسته ها یا فیلترهایی است که در امتداد ویژگی های ورودی اسلاید می شوند و پاسخ های معادل ترجمه معروف به نقشه های ویژگی را ارائه می دهند. برخلاف شبکه‌های عصبی کانولوشن به دلیل عملیات نمونه‌برداری پایینی که در ورودی اعمال می‌کنند، نسبت به ترجمه ثابت نیستند. آنها در تشخیص تصویر و ویدئو، سیستم های توصیه کننده، طبقه بندی تصویر، تقسیم بندی تصویر، تجزیه و تحلیل تصویر پزشکی، پردازش زبان طبیعی، رابط های مغز و کامپیوتر و سری های زمانی مالی کاربرد دارند.



شکل 1) شماتیک کلی یک شبکه CNN

CNN ها نسخه های منظم پرسپترون های چندلایه هستند. پرسپترون های چندلایه معمولاً به معنای شبکه های کاملاً متصل هستند، یعنی هر نورون در یک لایه به تمام نورون های لایه بعدی متصل است. "اتصال کامل"

این شـــبکه ها آنها را مســـتعد به تناســـب بیش از حد داده ها (Over Fitting) می کند. روش‌های معمول منظم‌ســـازی یا جلوگیری از تطبیق بیش از حد عبارتند از: جریمه‌کردن پارامترها در طول تمرین (مانند کاهش وزن) یا کوتاه کردن اتصال (اتصالات نادیده گرفته شده، ترک تحصیل، و غیره). و الگوهای با پیچیدگی فزاینده را با استفاده از الگوهای کوچکتر و ساده تری که در فیلترهای آنها نقش بسته است، جمع آوری کنید. بنابراین، در مقیاس اتصال و پیچیدگی، CNN ها در سطح پایین تر قرار دارند.

شبکه‌های کانولوشن از فرآیندهای بیولوژیکی الهام گرفته شده‌اند که الگوی اتصال بین نورون‌ها شبیه سازماندهی قشر بینایی حیوانات است. تک تک نورون‌های قشری تنها در ناحیه‌ای محدود از میدان بینایی به محرک‌ها پاسخ می‌دهند که به عنوان میدان گیرنده شـــناخته می‌شـــود. میدان های گیرنده نورون های مختلف تا حدی با هم همپوشانی دارند به طوری که کل میدان بینایی را پوشش می دهند.

CNN ها از پیش پردازش نسبتا کمی در مقایسه با سایر الگوریتم های طبقه بندی تصویر استفاده می کنند. این بدان معنی اســت که شــبکه یاد می گیرد که فیلترها (یا هســته ها) را از طریق یادگیری خودکار بهینه کند، در حالی که در الگوریتم های سـنتی این فیلترها به صـورت دسـتی طراحی شـده اند. این اسـتقلال از دانش قبلی و مداخله انسان در استخراج ویژگی یک مزیت عمده است.

شبکه‌های عصبی کانولوشن، نوع تخصصی از شبکه‌های عصبی مصنوعی هستند که از یک عملیات ریاضی به نام کانولوشــن به جای ضــرب ماتریس عمومی حداقل در یکی از لایه‌های خود اســتفاده می‌کنند. آنها به طور خاص برای پردازش داده های پیکسلی طراحی شده اند و در تشخیص و پردازش تصویر استفاده می شوند.

در CNN، ورودی یک تانسور با شکل: (تعداد ورودی) × (ارتفاع ورودی) × (عرض ورودی) × (کانال های ورودی) است. پس از عبور از یک لایه کانولوشن، تصویر به یک نقشه ویژگی انتزاعی می شود که به آن نقشه فعال سازی نیز می گویند: (تعداد ورودی ها) × (ارتفاع نقشه ویژگی) × (عرض نقشه ویژگی) × (کانال های نقشه ویژگی).

لایه های کانولوشـن ورودی را در هم می پیچند و نتیجه آن را به لایه بعدی منتقل می کنند. این شـبیه به پاسخ یک نورون در قشـر بینایی به یک محرک خاص اسـت. هر نورون کانولوشـنال داده ها را فقط برای میدان گیرنده خود پردازش می کند. اگرچه شــبکه‌های عصــبی فید فوروارد کاملاً متصــل می‌توانند برای یادگیری ویژگی‌ها و طبقه‌بندی داده‌ها اسـتفاده شـوند، این معماری به طور کلی برای ورودی‌های بزرگ‌تر مانند تصـاویر با وضـوح بالا

غیر عملی است. این به تعداد بسیار زیادی نورون نیاز دارد، حتی در یک معماری کم عمق، به دلیل اندازه ورودی بزرگ تصاویر، جایی که هر پیکسل یک ویژگی ورودی مرتبط است. به عنوان مثال، یک لایه کاملاً متصل برای یک تصویر (کوچک) با اندازه 100×100 دارای 10000 وزن برای هر نورون در لایه دوم است. در عوض، پیچیدگی تعداد پارامترهای آزاد را کاهش می‌دهد و به شبکه اجازه می‌دهد عمیق‌تر شود. برای مثال، صرف نظر از اندازه تصویر، استفاده از یک ناحیه کاشی کاری 5×5، که هر کدام دارای وزن های مشترک یکسانی هستند، تنها به 25 پارامتر قابل یادگیری نیاز دارد. استفاده از وزن‌های منظم‌شده بر روی پارامترهای کمتر، از شیب‌های ناپدید شدن و مشکلات Exploding Gradients که در طول پس انتشار خطا در شبکه‌های عصبی سنتی دیده می‌شود، جلوگیری می‌کند. علاوه بر این، شبکه‌های عصبی کانولوشن برای داده‌هایی با توپولوژی شبکه مانند (مانند تصاویر) ایده‌آل هستند زیرا روابط فضایی بین ویژگی‌های جداگانه در طول کانولوشن و/یا ادغام در نظر گرفته می‌شود.

2-2- پیاده سازی شبکه CNN با TensorFlow

در TensorFlow، هر تصویر ورودی معمولاً به عنوان یک تنسور سه بعدی شکل [ارتفاع، عرض، کانال ها] نشان داده می شود. یک دسته کوچک به عنوان یک تنسور 4 بعدی شکل [اندازه mini-batch، ارتفاع، عرض، کانال ها] نشان داده می شود. وزن یک لایه کانولوشن به صورت یک تنسور 4 بعدی نشان داده می شود. شرایط بایاس یک لایه کانولوشن به سادگی به عنوان یک تنسور 1 بعدی شکل نشان داده می شود.

2-3- Transfer Learning

یادگیری انتقالی به استفاده از دانش به دست آمده از یک مسئله یادگیری ماشین در مشکل دیگر اشاره دارد. به عنوان مثال، استفاده از دانش به دست آمده از تشخیص گربه/سگ برای شناسایی ساختمان ها. مؤلفه اصلی یادگیری انتقالی، استفاده از مدل های از پیش آموزش دیده برای جمع آوری دانش از یک کار و اعمال آن در سایر وظایف است.

مهمتر از همه، Transfer Learning یک جهش بزرگ برای توسعه دهندگان هوش مصنوعی در نظر گرفته می شود، زیرا به ما اجازه می دهد تا برنامه ها را سریعتر و کارآمدتر توسعه دهیم.

2-4- مزایای استفاده از مدل از قبل آموزش دیده شده

اگر ابتدا مزایا را در نظر بگیریم، قابل توجه ترین مزیت پیش تمرین، سهولت استفاده است. فرض کنید ما یک کار یادگیری ماشینی داریم که باید روی آن کار کنیم. تنها کاری که باید انجام دهیم این است که یک مدل از پیش آموزش دیده را پیدا کنیم که در کارهای مشابه آموزش دیده باشد و آن را در کاری که روی آن کار می کنیم اعمال کنیم. نیازی به ساخت مدل از ابتدا نیست.

قبل از آموزش به مدل ها اجازه می دهد تا به سرعت بهینه شوند. این بدان معناست که اگر یک مدل از قبل آموزش دیده استفاده شود، یک مدل می‌تواند سریع‌تر به عملکرد بهینه دست یابد. مدلی که در دانستن اینکه کدام پارامترها احتمالاً به نتایج خوبی دست می یابند می تواند سریعتر در مقایسه با شروع از صفر بهینه شود.

علاوه بر این، مدل‌های از قبل آموزش‌دیده شده این مزیت را دارند که به اندازه ساخت یک مدل از ابتدا به داده نیاز ندارند. این به این دلیل است که اکثر مدل های از پیش آموزش دیده موجود در اینترنت تا به امروز بر روی مجموعه داده های بسیار بزرگ آموزش داده شده اند. از این رو، استفاده از چنین مدلی برای یک کار متفاوت به داده های کمتری برای همگرایی نیاز دارد.

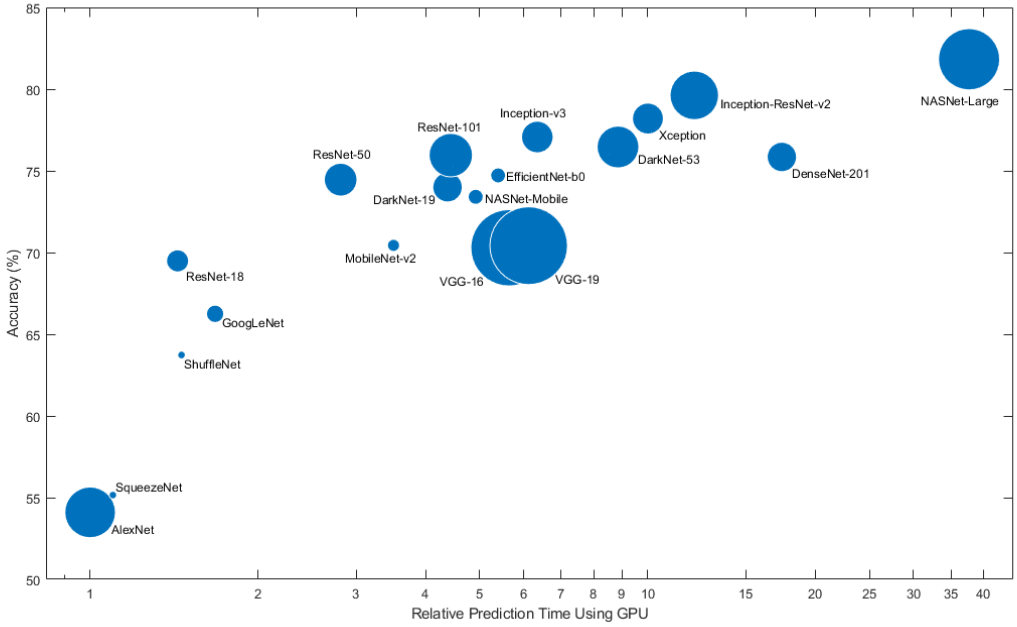
2-5- معایب استفاده از مدل از قبل آموزش دیده شده

اگرچه مفید است، اما قبل از آموزش باید با کمی احتیاط اعمال شود. اولا، ما همیشه به نتایج خوبی که توسط مدل از پیش آموزش دیده به دست می آید، نخواهیم رسید. عوامل متعددی ممکن است باعث این اتفاق شود. به عنوان مثال، استفاده از مجموعه داده از یک دامنه کاملاً متفاوت ممکن است نتایج یکسانی نداشته باشد. علاوه بر این، پارامترهای شبکه، نسبت تقسیم آزمون قطار و سخت افزار برای آموزش مورد استفاده عوامل تعیین کننده هستند.

علاوه بر این، تنظیم دقیق مدل های از پیش آموزش دیده می تواند کار دشواری باشد. برای تنظیم دقیق آنها به زمان و منابع CPU نیاز دارند.

2-6- مقایسه شبکه های از پیش آموزش داده شده استفاده شده

می‌توان یک شـبکه طبقه بندی تصـاویر از پیش آموزش دیده را انتخاب کرد که قبلاً آموخته اسـت ویژگی های قدرتمند و آموزنده را از تصاویر طبیعی استخراج کند و از آن به عنوان نقطه شروع برای یادگیری یک کار جدید



شکل 2)مقایسه شبکه های pretrained شده از منظر Accuracy و زمان پیش بینی

اسـتفاده کرد. اکثر شـبکه های از پیش آموزش دیده بر روی زیرمجموعه ای از پایگاه داده ImageNet آموزش داده می شوند که در چالش تشخیص تصویری در مقیاس بزرگ ImageNet (ILSVRC) استفاده می شود. این شـبکه ها بر روی بیش از یک میلیون تصـویر آموزش دیده اند و می توانند تصاویر را در 1000 دسـته شـی مانند صـفحه کلید، لیوان قهوه، مداد و بسـیاری از حیوانات طبقه بندی کنند. اسـتفاده از یک شـبکه از پیش آموزش دیده با یادگیری انتقال معمولاً بسیار سریعتر و ساده تر از آموزش شبکه از ابتدا است.

در شـکل فوق مقایسـه ای از شـبکه های مختلف از پیش آموزش دیده به تصـویر کشـیده شـده اسـت که در آن مساحت هر دایره با سایز شبکه بر روی دیسک نیز متناسب است.

همچنین می توان با آموزش شبکه بر روی مجموعه داده های جدید خود با شبکه از پیش آموزش دیده به عنوان نقطه شـروع، لایه های عمیق تر را در شـبکه تنظیم کرد. Fine-Tune شـبکه با یادگیری انتقال اغلب سـریعتر و آسانتر از ساخت و آموزش یک شبکه جدید است. شبکه قبلاً مجموعه ای غنی از ویژگی های تصویر را یاد گرفته

اســت، اما وقتی شــبکه را Fine-Tune شــود، می تواند ویژگی های خاص مجموعه داده جدید را یاد بگیرد. اگر مجموعه داده‌ای بسیار بزرگ باشد، ممکن است انتقال یادگیری سریع‌تر از آموزش از ابتدا نباشد.

Fine-Tune یک شبکه کندتر است و به تلاش بیشتری نسبت به استخراج ویژگی های ساده نیاز دارد، اما از آنجایی که شبکه می تواند استخراج مجموعه متفاوتی از ویژگی ها را بیاموزد، شبکه نهایی اغلب دقیق تر است. Fine-Tune معمولا بهتر از اســتخراج ویژگی عمل می کند تا زمانی که مجموعه داده های جدید خیلی کوچک نبــاشــــد، زیرا در ایـن صـــورت شـــبکـه داده هـایی بـرای یـادگـیـری ویـژگـی هـای جـدیـد دارد. ما در این قسمت، از سه شبکه EfficientNetB7، ResNet50 و همچنین Custom VGG استفاده کردیم که در ادامه به توضیحات آن ها می‌پردازیم.

شبکه ResNet50

ResNet مخفف Residual Network است و نوع خاصی از شبکه عصبی کانولوشنال (CNN) است که در مقاله سال 2015 "یادگیری عمیق باقیمانده برای تشخیص تصویر" توسط He Kaiming، Zhang Xiangyu، Ren Shaoqing و Sun Jian معرفی شده است. CNN ها معمولا برای تقویت برنامه های بینایی کامپیوتری استفاده می شوند.

ResNet-50 یک شبکه عصبی کانولوشن 50 لایه است (48 لایه کانولوشن، یک لایه MaxPool و یک لایه استخر متوسط). شبکه‌های عصبی باقی‌مانده نوعی شبکه عصبی مصنوعی (ANN) هستند که با چیدن بلوک‌های باقیمانده، شبکه‌ها را تشکیل می‌دهند.

معماری اولیه ResNet ResNet-34 بود که شامل 34 لایه وزنی بود. با استفاده از مفهوم اتصالات میانبر، روش جدیدی را برای افزودن لایه‌های کانولوشنال بیشتر به یک CNN، بدون مواجهه با مشکل گرادیان ناپدید، ارائه کرد. یک اتصال میانبر از روی برخی از لایه ها عبور می کند و یک شبکه معمولی را به یک شبکه باقی مانده تبدیل می کند.

شبکه معمولی مبتنی بر شبکه های عصبی VGG (VGG-16 و VGG-19) بود - هر شبکه کانولوشن دارای یک فیلتر 3×3 بود. با این حال، ResNet فیلترهای کمتری دارد و پیچیدگی کمتری نسبت به VGGNet دارد. یک ResNet 34 لایه می تواند به عملکرد 3.6 میلیارد FLOP دست یابد و یک ResNet 18 لایه کوچکتر می

تواند به 1.8 میلیارد FLOP دســـت یابد که به طور قابل توجهی ســـریعتر از یک شـــبکه VGG-19 با 19.6 میلیارد FLOP است (در مقاله ResNet بیشتر بخوانید. و همکاران، 2015).

معماری ResNet از دو قانون اساسی طراحی پیروی می کند. ابتدا تعداد فیلترها در هر لایه بسته به اندازه نقشه ویژگی خروجی یکسـان اسـت. دوم، اگر اندازه نقشـه ویژگی نصـف شـود، تعداد فیلترهای آن دو برابر برای حفظ پیچیدگی زمانی هر لایه است.

ResNet-50 دارای معماری مبتنی بر مدلی اســـت که در بالا نشـــان داده شـــده اســـت، اما با یک تفاوت مهم. ResNet 50 لایه از یک طرح گلوگاه برای بلوک ســـاختمان اســـتفاده می کند. یک بلوک باقیملنده گلوگاه از پیچیدگی‌های 1×1 استفاده می‌کند که به نام «گلوگاه» شناخته می‌شود، که تعداد پارامترها و ضرب‌های ماتریس را کاهش می‌دهد. این امکان آموزش ســـریعتر هر لایه را فراهم می کند. از یک پشـــته ســـه لایه به جای دو لایه استفاده می کند.

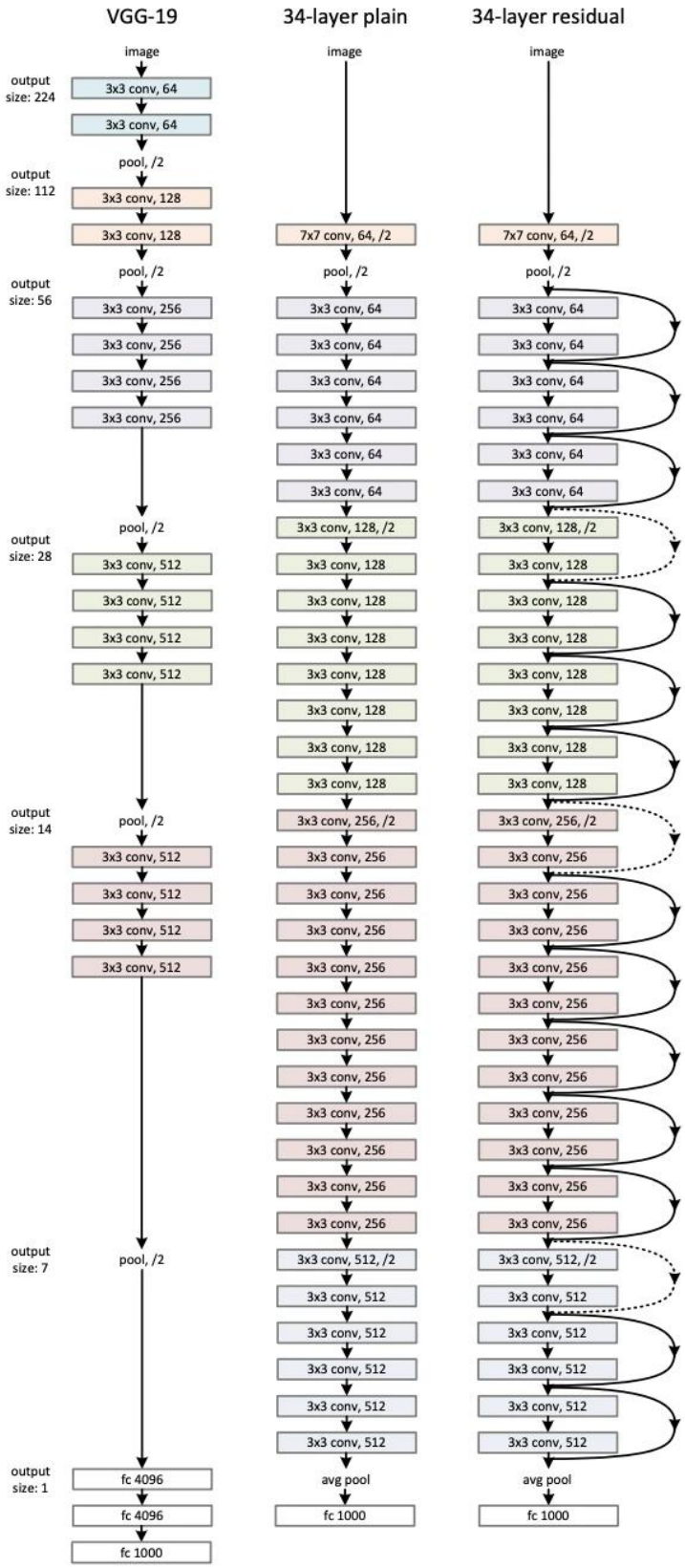
معماری ResNet 50 لایه شامل عناصر زیر است که در جدول زیر نشان داده شده است:

- **یک هسته کانولوشن 7×7** در کنار 64 هسته دیگر با یک stride در اندازه 2.
- **یک لایه Max pooling** حداکثر با اندازه stride 2.
- **9 لایه دیگر** 3 در 3 با 64 هسته کانولوشن، دیگری با اندازه 1 در 1 و 64 هسته، و لایه سوم با با اندازه 1 در 1 با 256 هسته. این 3 لایه 3 بار تکرار می شوند.
- **12 لایه دیگر** با 128 هسته با اندازه 1 در 1، 128 هسته با اندازه 1 در 1 و 512 هسته با اندازه 1 در 1، که 4 بار تکرار شده است.
- **18 لایه دیگر** با 256 هسته 1 در 1و 2 هسته 3 در 3، 256 و یک 1 در 1 با 1024 هسته که 6 بار تکرار شده است.
- **9 لایه دیگر** با 512 هسته 1 در 1، 512 هسته 3 در 3 و 2048 هسته 1 در 1 که 3 بار تکرار شده است.

  (تا این لحظه شبکه دارای 50 لایه است)
- **Average Pooling**، به دنبال آن یک لایه کاملا متصل با 1000 گره، با استفاده از تابع فعال ساز softmax.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

شکل 3) ساختار شبکه ResNet



شکل 4) مقایسه ساختار ResNet با شبکه معمولی CNN و VGG

به طور خلاصه ResNet-50 یک شبکه عصبی کانولوشن است که عمق 50 لایه دارد. یک نسخه از پیش آموزش دیده شبکه آموزش داده شده بر روی بیش از یک میلیون تصویر از پایگاه داده ImageNet قابل بارگیری است. شبکه از پیش آموزش دیده می تواند تصاویر را به 1000 دسته شی، مانند صفحه کلید، ماوس، مداد و بسیاری از حیوانات طبقه بندی کند.

شبکه EfficientNet B7

در معماری های طراحی شده در شبکه های کانولوشنی، سه روش برای افزایش دقت استفاده می‌شود.این سه روش شامل: افزایش عمق شبکه، ارتفاع شبکه و همچنین افزایش رزولوشن ورودی می‌باشد. که افزایش هر کدامم از این ویژگی‌ها می‌تواند باعث بهبود عملکرد شبکه شود.

در این مقاله به بررسی رابطه این سه ویژگی می‌پردازد و بدیهی می‌باشد که این سه ویژگی با یکدیگر ارتباط مستقیمی دارند، بدین صورت که با افزایش رزولوشن، ویژگی بیشتری برای بررسی وجود دارد بنابراین شبکه می‌تواند عمق بیشتری داشته باشد.

در واقع می توان به effecientNet به عنوان یک نوع جستجو برای کارآمد ترین شبکه ی عصبی با توجه به میزان توان محاسباتی نگاه کرد.

در این مقاله ایده ای در رابطه با طراحی شبکه جدید مطرح نشده است. بلکه با توجه به اینکه دستگاه های مختلف از توان پردازشی متفاوتی بهره مند هستند می‌خواهیم شیوه ای داشته باشیم که با توجه به دستگاه در دسترس و توانایی پردازش موجود چگونه یک شبکه را Scale کنیم. همچنین اگر از نظر زمانی، میزان زمان لازم برای آموزش شبکه را در نظر بگیریم با scale down شبکه زود تر و با scale up شبکه به مدت طولانی تری نیاز به آموزش دارد. به طور مثال اگر بخواهیم شبکه ی ما سریع‌تر آموزش ببیند و کمی کاهش دقت در نتایج شبکه مسئله ی خیلی مهمی نباشد میتوان از این روش استفاده نمود.

بنابراین Efficient-net یک راهی برای به دست آوردن بهینه ترین میزان برای scale up کردن با توجه به شرایط موجود می باشد.

سه روش برای scale up کردن یک شبکه ی کانوولوشنی به صورت زیر می‌باشد:

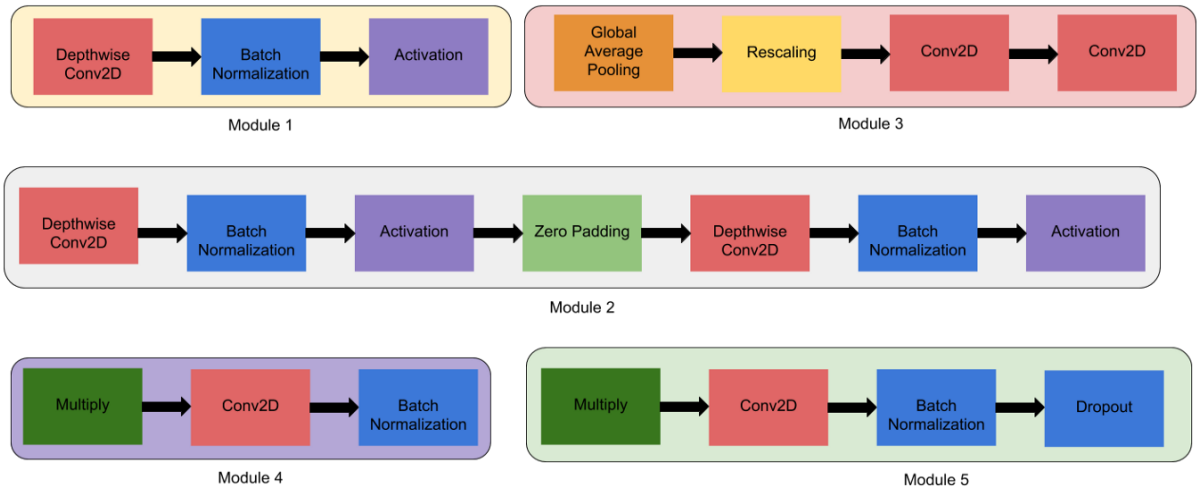
روش اول افزاش عمق: این روش بیشترین استفاده را در معماری های موجود تا کنون داشته است. منظور از افزایش عمق، افزایش تعداد لایه های یک شبکه می باشد.

روش دوم افزایش عرض: از این روش نسبت به عمق کمتر استفاده می‌شود. منظور از عرض نیز مقدار کانال های یک شبکه می باشد.

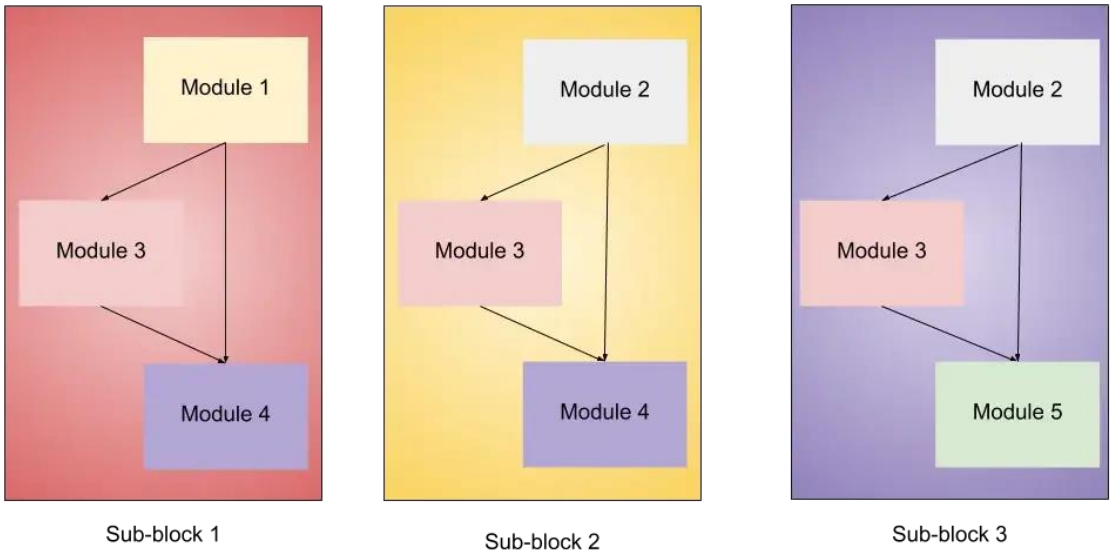
روش سوم افزایش resolution عکس ورودی: از این روش نیز گاها در مقالات مشاهده شده که استفاده شده است.

معماری که به عنوان baseline در نظر گرفته می شود نیز از اهمیت بسیاری برخوردار است، زیرا efficient netمعماری baseline را تغییری نمی دهد و صرفا آن را scale می‌کند. و علاوه بر آن به طور مثال اگر Alexnet به عنوان baseline استفاده شود، و بعد از تغییر scale با استفاده از عرض، طول و رزولوشن روی همان Alexnet مقایسه می‌شود و ب شبکه‌ی دیگری به طور مثال مانند REsnet مقایسه نمی شود. پس بنابراین داشتن یک baseline مناسب نیز از اهمیت ویژه ای برخوردار است.

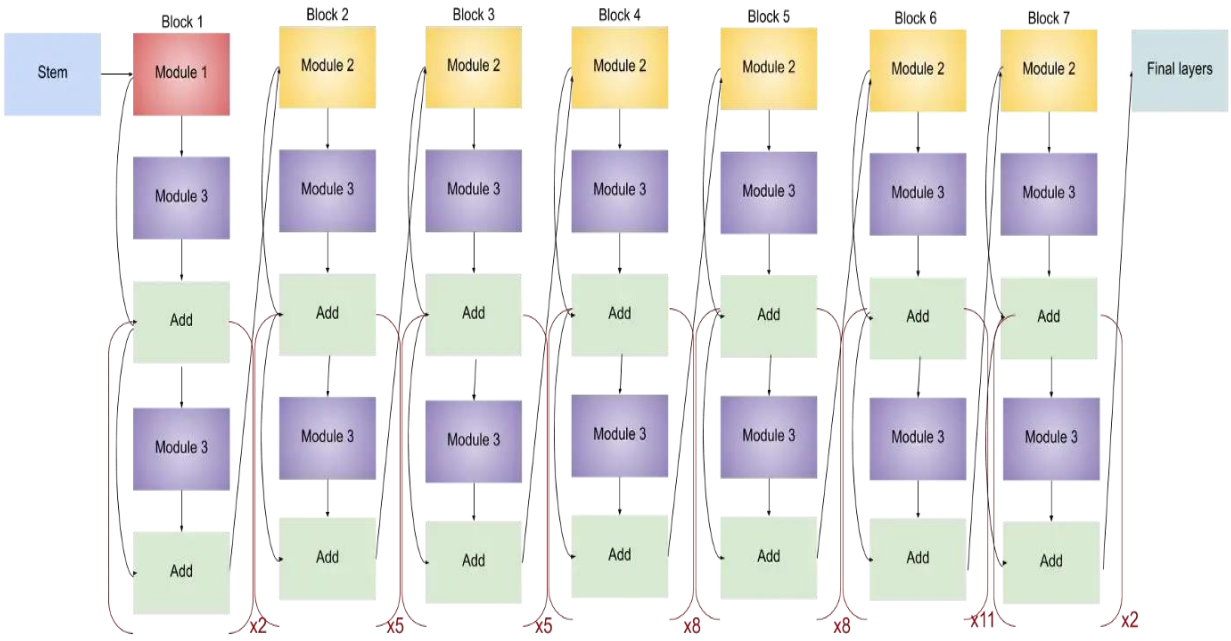
معماری شبکه Efficientnet B7 شامل چند ماژول است که ساب ماژول هایی را تشکیل می‌دهند که آن ساب‌ماژول های در طراحی شبکه استفاده می‌شوند که تصاویر آنرا در زیر مشاهده می‌کنید:



شکل۱۱) ماژول های طراحی EfficientNet



شکل۱۲)ساب ماژول های شبکه



شکل۱۳) شبکه نهایی EfficientNet B7

توضیح کد

- Importing Libraries

Importing Packages

```
In [1]: !pip install -q neurokit2
        !pip install -q wfdb
        !pip install -q ecg-plot
```

```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.2/1.2 MB 15.4 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 159.9/159.9 KB 3.8 MB/s eta 0:00:00
```

```
In [2]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        from scipy import signal
        import tensorflow as tf
        from tensorflow import keras
        import neurokit2 as nk
        import wfdb
        import ecg_plot
        import seaborn as sns
        from sklearn.decomposition import PCA
        from sklearn.model_selection import train_test_split
        from glob import glob
        from pathlib import Path
```

شکل۱۳) پکیج های مورد نیاز را اضافه می‌کنیم.

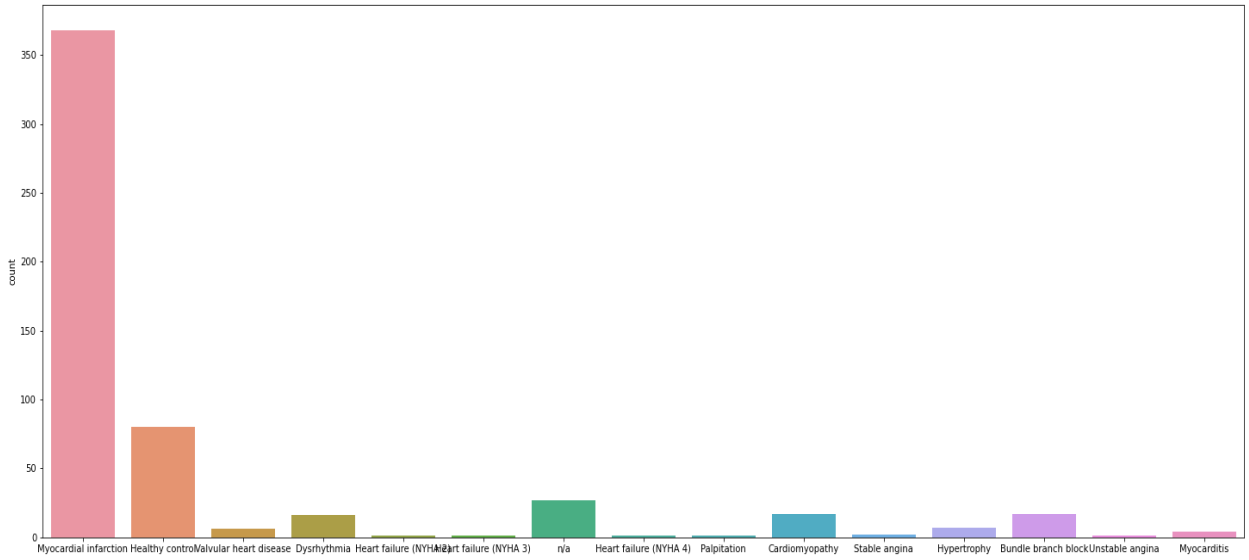
- Dataset Loading

در این قسمت با دستور win get دیتاست را دانلود و به درابو خودمان اضافه و از آنجا آن را لود می‌کنیم.

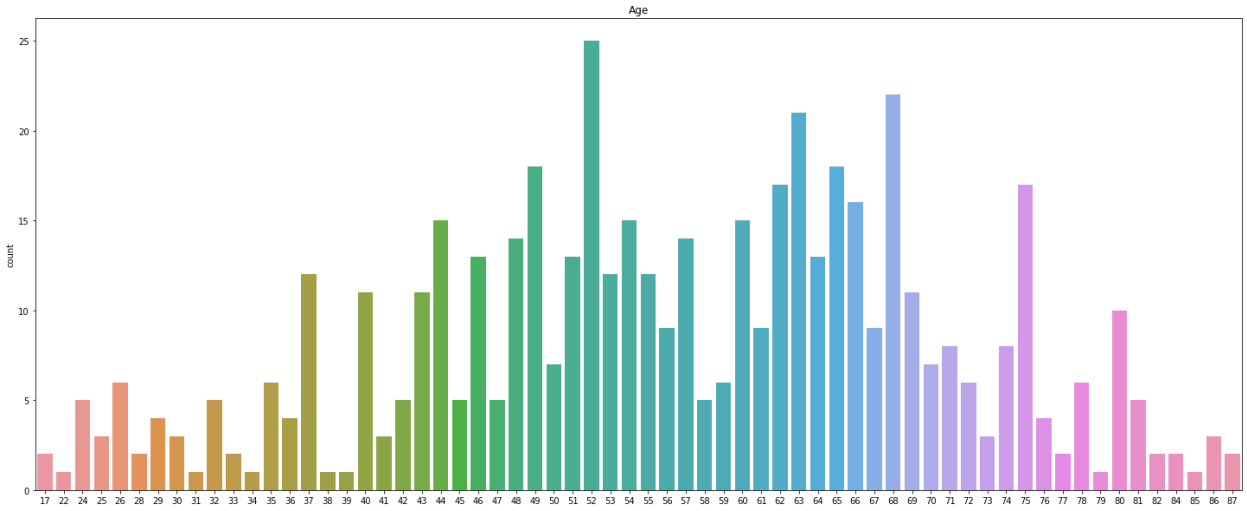
```
In [3]: # Mount GoogleDrive
        from google.colab import drive

        drive.mount("drive")

        Mounted at drive

In [ ]: # Run this section only the first time to download the dataset
        !wget -r -N -c -np https://physionet.org/files/ptbdb/1.0.0/
        !cp -r physionet.org/files/ptbdb/1.0.0/ drive/MyDrive/ml-dataset
        !rm -r physionet.org/

        --2023-01-17 18:12:09--  https://physionet.org/files/ptbdb/1.0.0/
        Resolving physionet.org (physionet.org)... 18.18.42.54
        Connecting to physionet.org (physionet.org)|18.18.42.54|:443... connected.
        HTTP request sent, awaiting response... 200 OK
        Length: unspecified [text/html]
        Saving to: 'physionet.org/files/ptbdb/1.0.0/index.html'

        physionet.org/files     [ <=>                ]  33.61K  --.-KB/s    in 0.009s

        Last-modified header missing -- time-stamps turned off.
        2023-01-17 18:12:09 (3.72 MB/s) - 'physionet.org/files/ptbdb/1.0.0/index.html' saved [34418]

        Loading robots.txt; please ignore errors.
        --2023-01-17 18:12:09--  https://physionet.org/robots.txt
        Reusing existing connection to physionet.org:443.
        HTTP request sent, awaiting response... 200 OK

            The file is already fully retrieved; nothing to do.
```

شکل۱۴) خواندن دیتاست

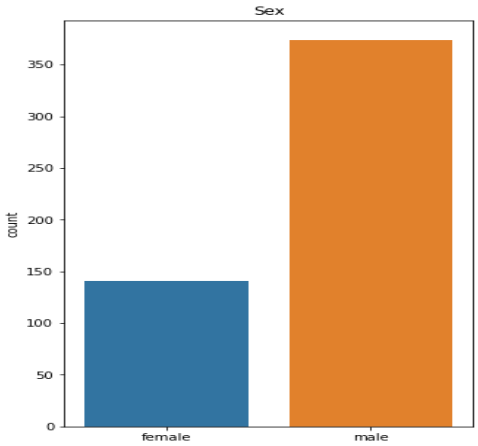
بعد از خواندن دیتاست برخی از توزیع های مربوط به بیماران و پراکندگی داده های مربوط به آنان را چاپ می-کنیم.



شکل ۱۵) توزیع بیماری های قلبی



شکل۱۶) توزیع سن بیماران



شکل۱۷) توزیع جنسیت بیماران

سپس دیتا هایی که مقدار nan دارند با کد زیر از دیتاست حذف می‌کنیم.

Data Cleaning

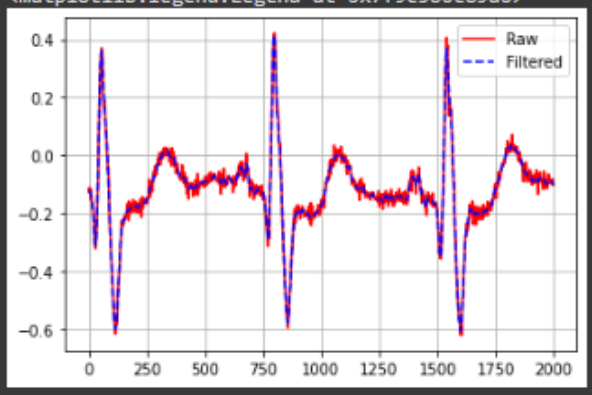
```
In [6]: # Removing data with n/a labels
for rec in records:
    data = wfdb.rdsamp(rec)
    label = " ".join(data[1]['comments'][4].split()[3:])
    if label == "n/a":
        records.remove(rec)
```

شکل۱۸) Data Cleaning

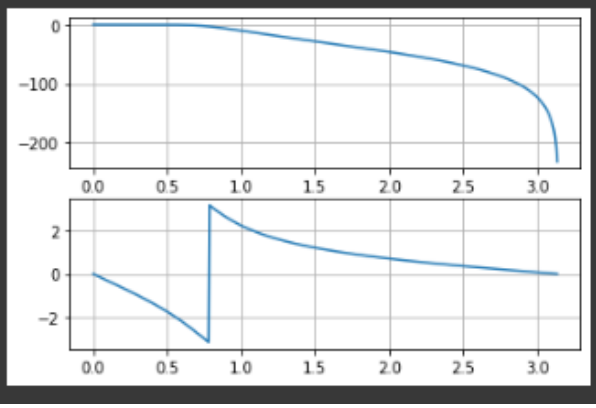
پس از حذف کردن دیتا های نامناسب دیتای سیگنال را بوسیله فیلتر های FIR, IIR نویزشان را می‌گیریم

```
# IIR Filtering
def iir_lp(x, cutoff, fs):
    sos = signal.butter(4, 2*cutoff/fs, output='sos')
    y = signal.sosfiltfilt(sos, x)
    return y

data = wfdb.rdsamp("drive/MyDrive/ml-dataset/patient001/s0010_re", channel_names=['i', 'ii'])[0]

plt.plot(data[:, 0][5000:7000], 'r')
plt.plot(iir_lp(data[:, 0][5000:7000], 50, 1000), 'b--')
plt.grid("on")
plt.legend(["Raw", "Filtered"])
```

```
<matplotlib.legend.Legend at 0x7f9c580c89a0>
```



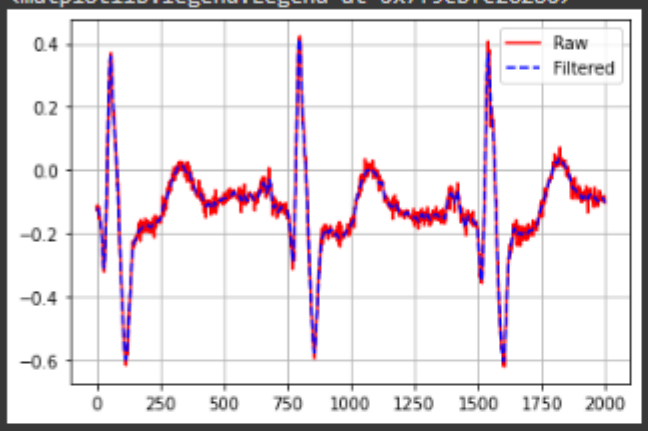

```
[ ] # Frequency response of IIR filter
    b, a = signal.butter(4, 0.25, output='ba')
    w, h = signal.freqz(b, a)
    fig, ax = plt.subplots(2, 1)
    ax[0].plot(w, 20*np.log10(np.abs(h)))
    ax[0].grid("on")
    ax[1].plot(w, np.angle(h))
    ax[1].grid("on")
```



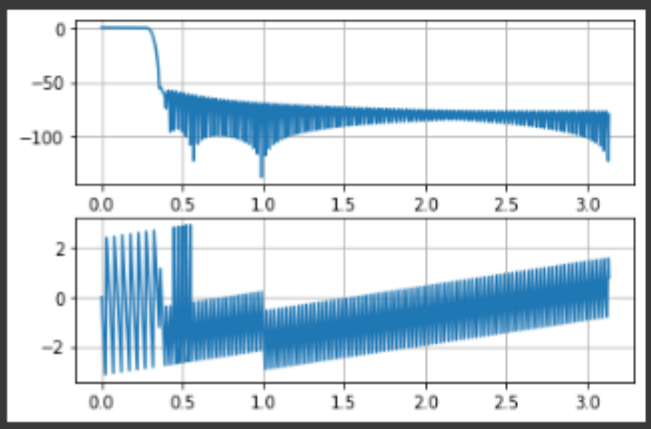
شکل۱۹) فیلتر IIR

```
[ ]  # FIR Filtering
     def fir_lp(x, cutoff, fs):
         h = signal.firwin(255, cutoff, fs=fs)
         y = signal.filtfilt(h, 1, x)
         return y

     plt.plot(data[:, 0][5000:7000], 'r')
     plt.plot(iir_lp(data[:, 0][5000:7000], 50, 1000), 'b--')
     plt.grid("on")
     plt.legend(["Raw", "Filtered"])
```

```
<matplotlib.legend.Legend at 0x7f9cbfc26280>
```




```
[ ]  # Frequency response of FIR filter
     h = signal.firwin(255, 50, fs=1000)
     w, h = signal.freqz(h , 1)
     fig, ax = plt.subplots(2, 1)
     ax[0].plot(w, 20*np.log10(np.abs(h)))
     ax[0].grid("on")
     ax[1].plot(w, np.angle(h))
     ax[1].grid("on")
```



شکل ۲۰) فیلتر FIR

- Generator

بعلت حجیم بودن دیتاست بهتر است از جنریتور ها استفاده کنیم این تایپ متغیر به اندازه batch size دیتا هارا گرفته و به حافظه موقت فرستاده و شبکه را با آن آموزش می‌دهد در واقع این توابع با yield کردن تابع به جای return کردن و بستن تابع و کپی کردن کامل دیتاست بر روی ram بصورت کلی باعث عدم استفاده از حجم زیاد سخت افزار می‌شوند و آموزش را ممکن می‌کنند.

Train

In [14]:
```
class DataGenerator(tf.keras.utils.Sequence):
    def __init__(self, data_dir, patients_list,
                 batch_size,
                 input_size=(10, 1000, 1),
                 shuffle=True):

        self.data_dir = data_dir
        self.patients_list = patients_list
        self.batch_size = batch_size
        self.input_size = input_size
        self.shuffle = shuffle

        self.n = len(self.patients_list)
        self.__get_recs()

    def __get_recs(self,):
        self.records = []
        for patient in self.patients_list:
            dir = Path(self.data_dir) / f"patient{str(patient).zfill(3)}"
            self.records.extend([dir / x[:-4] for x in glob(f"{str(dir)}/*.dat")])

    def on_epoch_end(self):
        if self.shuffle == True:
            np.random.shuffle(self.records)

    def __getitem__(self, index):
        rec = self.records[index]
        signals = preprocess(wfdb.rdsamp(rec)[0])
        if signals.shape[1] < self.batch_size*1000:
            np.pad(signals, ((0, 0),(0, self.batch_size*1000 - signals.shape[1])), 'constant')

        batched_signals = np.array([signals[:, i*1000:(i+1)*1000] for i in range(self.batch_size)])
        labels = " ".join(wfdb.rdsamp(rec)[1]['comments'][4].split()[3:])
        return (batched_signals.reshape(self.batch_size, *self.input_size),
                (np.array(labels) == 'Myocardial infarction').astype(int).repeat(self.batch_size))

    def __len__(self):
        return self.n
```

شکل ۲۱) دیتا جنریتور

In [15]:
```
# Splitting data into train, test, validation
NUM_SUBJECTS = 294
train_patients = np.random.choice(np.arange(1, NUM_SUBJECTS+1), int(0.8*NUM_SUBJECTS), replace=False)
test_patients = np.array([idx for idx in np.arange(1, NUM_SUBJECTS+1) if idx not in train_patients])

# Generators
train_gen = DataGenerator("/content/drive/MyDrive/ml-dataset", train_patients, 32)
test_gen = DataGenerator("/content/drive/MyDrive/ml-dataset", test_patients, 32)
```

In [16]:
```
# Splitting data into train, test, validation
test_patients = np.array([idx for idx in np.arange(1, NUM_SUBJECTS+1) if idx not in train_patients])
validation_patients = np.random.choice(test_patients, int(0.8*len(test_patients)), replace=False)
test_patients = np.array([i for i in test_patients if i not in validation_patients])

# Generators
train_gen = DataGenerator("/content/drive/MyDrive/ml-dataset", train_patients, 32)
test_gen = DataGenerator("/content/drive/MyDrive/ml-dataset", test_patients, 1)
validation_gen = DataGenerator("/content/drive/MyDrive/ml-dataset", validation_patients, 32)
```

In [17]:
```
# len_generator = 446
print(train_gen[0][0].shape)
print(train_gen[0][1].shape)
print(train_gen[2][0].shape)
```

```
(32, 10, 1000, 1)
(32,)
(32, 10, 1000, 1)
```

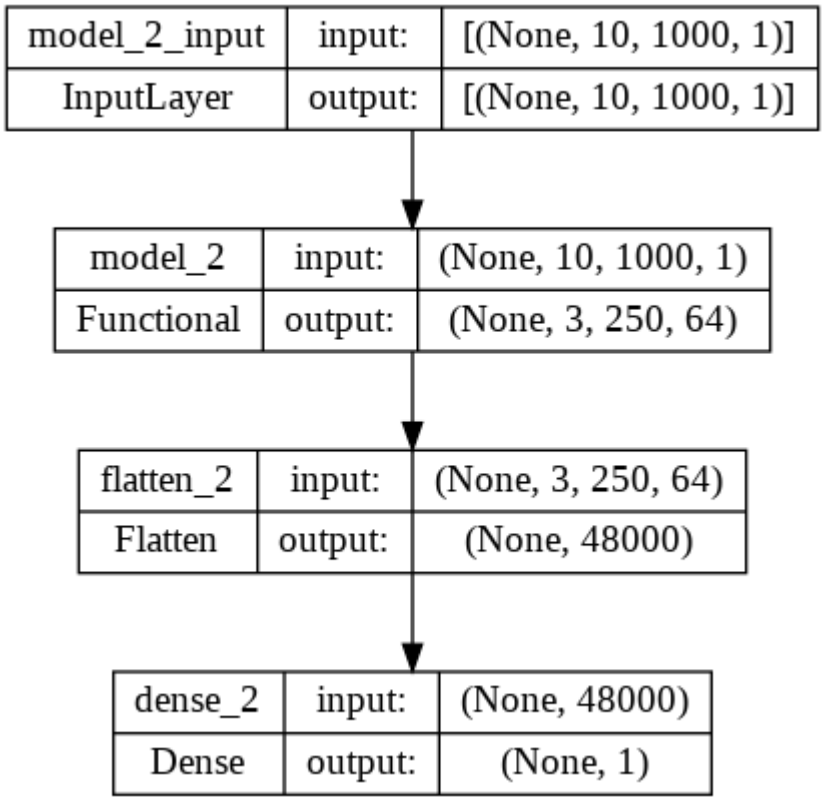
شکل۲۲) تقسیم دیتاست به داده أموزش و تست و ولیدیشن

- Training – ResNet

در این مرحله ما از رزنت استفاده کردیم اما برای حجیم نشدن شبکه از ۳- لایه اول این شبکه استفاده کردیم و نکته قابل ذکر دیگر این است که ما ابتدا از برای تابع فعال ساز لایه آخرمان از softmax استفاده کردیم سپس با توصیه مهندس کریمی از تابع فعالساز sigmoid نیز برای لایه أخر استفاده کردیم که نتایج را در اشکال زیر میبینید (سافت‌مکس عملکرد بهتری از سیگموید نشان داد):

مدلی که از آن برای پردازش صوت در این پروژه استفاده کردیم ,مدل زیر است:

```
from keras import layers
from keras.losses import BinaryCrossentropy

input_s = layers.Input((10,1000,1))

base_res = tf.keras.applications.resnet50.ResNet50(include_top=False,
                                        weights=None,
                                        input_tensor = input_s)
pretrained_model_res = tf.keras.Model(inputs=base_res.input, outputs=base_
res.layers[30].output)

model_res = keras.Sequential([
    pretrained_model_res,
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(1, activation="softmax")
])


model_res.compile(optimizer='adam', loss=BinaryCrossentropy(), metrics=['a
ccuracy'])

model_res.summary()
```

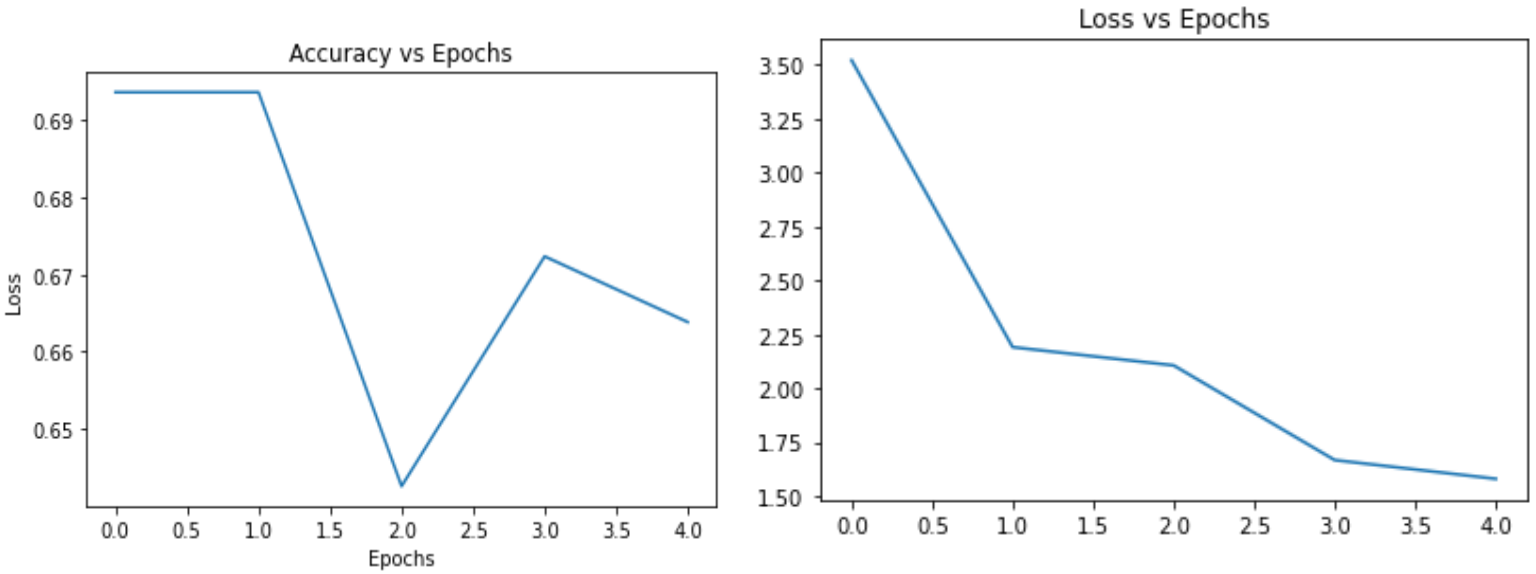


شکل۲۳) معماری شبکه رزنت

که نتیجه این شبکه با فعال ساز لایه أخر سافت‌مکس بصورت زیر می‌باشد:

```
history_res = model_res.fit(train_gen, epochs=5)

Epoch 1/5
235/235 [==============================] - 484s 2s/step - loss: 3.5206 - accuracy: 0.6936
Epoch 2/5
235/235 [==============================] - 478s 2s/step - loss: 2.1905 - accuracy: 0.6936
Epoch 3/5
235/235 [==============================] - 495s 2s/step - loss: 2.1059 - accuracy: 0.6426
Epoch 4/5
235/235 [==============================] - 486s 2s/step - loss: 1.6669 - accuracy: 0.6723
Epoch 5/5
235/235 [==============================] - 488s 2s/step - loss: 1.5800 - accuracy: 0.6638

[ ] model_res.evaluate(validation_gen)

47/47 [==============================] - 61s 1s/step - loss: 1.5374 - accuracy: 0.7021
[1.537388563156128, 0.7021276354789734]
```

شکل۲۴) فیت کردن مدل رزنت

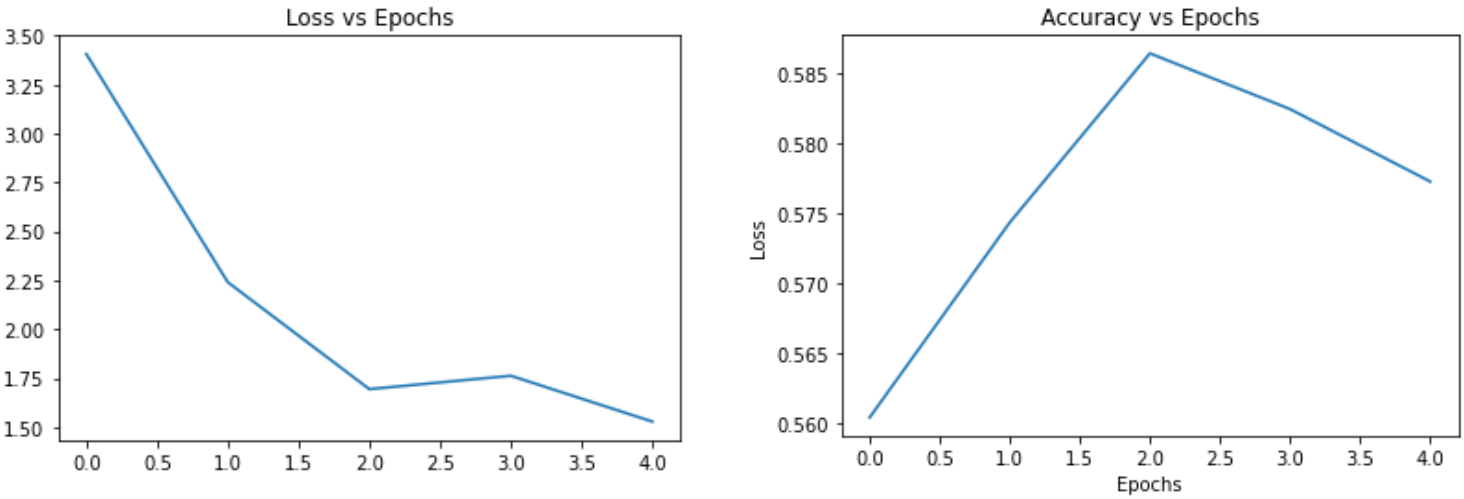


شکل۲۵) دقت و خطا مدل رزنت با سافت‌مکس

در قسمت پایین هم نتایج با تابع فعال ساز سیگموید را مشاهده می‌کنید:

```
[ ]  history_res = model_res.fit(train_gen, epochs=5)

     Epoch 1/5
     235/235 [==============================] - 453s 2s/step - loss: 3.4073 - accuracy: 0.5604
     Epoch 2/5
     235/235 [==============================] - 451s 2s/step - loss: 2.2402 - accuracy: 0.5743
     Epoch 3/5
     235/235 [==============================] - 455s 2s/step - loss: 1.6934 - accuracy: 0.5864
     Epoch 4/5
     235/235 [==============================] - 447s 2s/step - loss: 1.7618 - accuracy: 0.5824
     Epoch 5/5
     235/235 [==============================] - 444s 2s/step - loss: 1.5274 - accuracy: 0.5773

[ ]  model_res.evaluate(validation_gen)

     47/47 [==============================] - 49s 1s/step - loss: 2.3182 - accuracy: 0.2773
     [2.3181557655334473, 0.2772606313228607]
```

شکل۲۶) آموزش شبکه رزنت با سیگموید

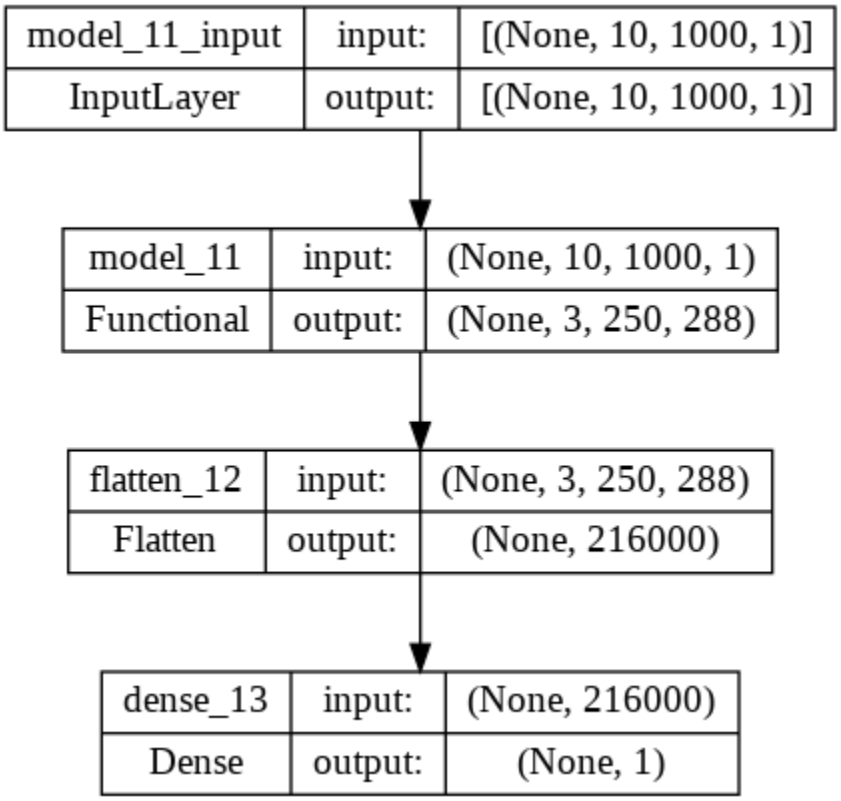


شکل۲۷) نتایج رزنت با تابع فعالساز سیگموید

- Training – EfficientNet B7

مانند مدل قبلی در اینجا از ۱۰۰ لایه اول EfficientNet B7 بهره بردیم که کد مدل همراه با نتایج در زیر مشخص هستند:

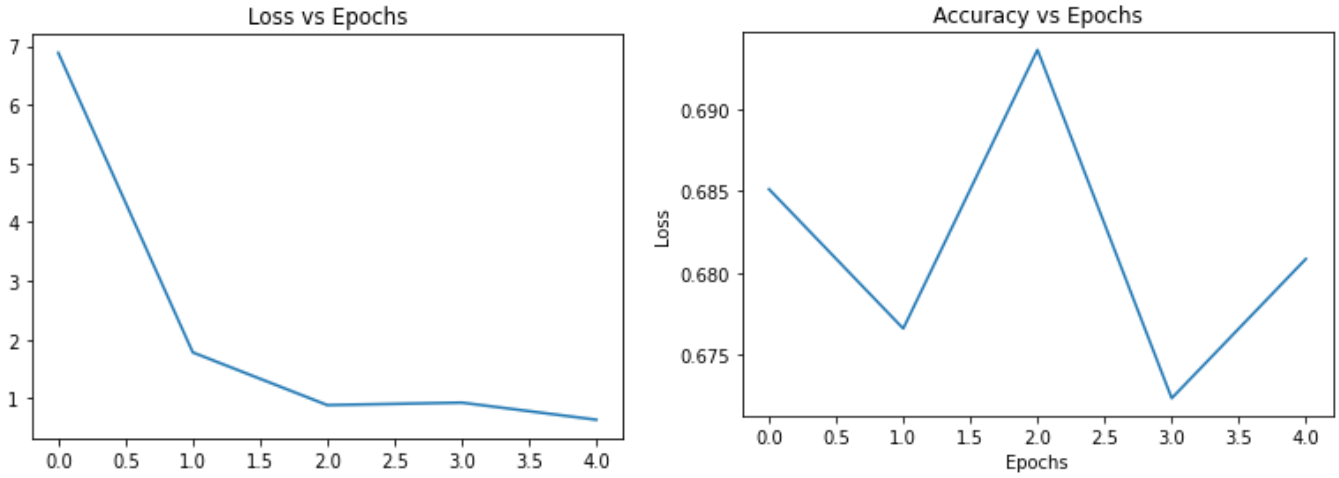
```
from keras import layers
input_s = layers.Input((10,1000,1))

base_effB7 = tf.keras.applications.efficientnet.EfficientNetB7(include_top
=False,
                                                weights=None,
                                                input_tensor = input_s)
pretrained_model_effB7 = tf.keras.Model(inputs=base_effB7.input, outputs=b
ase_effB7.layers[100].output)


model_effB7 = keras.Sequential([
    pretrained_model_effB7,
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(1, activation="softmax")
])


model_effB7.compile(optimizer='adam', loss=tf.keras.losses.BinaryCrossentr
opy(), metrics=['accuracy'])

model_effB7.summary()
```



شکل۲۸) معماری شبکه EfficientNet B7

```
history_Eff = model_effB7.fit_generator(train_gen, epochs=5)

<ipython-input-52-76a1cfb0f7e0>:1: UserWarning: `Model.fit_generator` is deprecated and will be removed in a future version. Please use `Model.fit`, which supports generators.
  history_Eff = model_effB7.fit_generator(train_gen, epochs=5)
Epoch 1/5
235/235 [==============================] - 978s 4s/step - loss: 6.8820 - accuracy: 0.6851
Epoch 2/5
235/235 [==============================] - 969s 4s/step - loss: 1.7832 - accuracy: 0.6766
Epoch 3/5
235/235 [==============================] - 987s 4s/step - loss: 0.8864 - accuracy: 0.6936
Epoch 4/5
235/235 [==============================] - 973s 4s/step - loss: 0.9288 - accuracy: 0.6723
Epoch 5/5
235/235 [==============================] - 978s 4s/step - loss: 0.6374 - accuracy: 0.6809

[ ] model_effB7.evaluate(validation_gen)

    47/47 [==============================] - 78s 2s/step - loss: 1.2220 - accuracy: 0.6596
    [1.2220107316970825, 0.6595744490623474]
```

شکل۲۹) آموزش شبکه EfficientNet B7



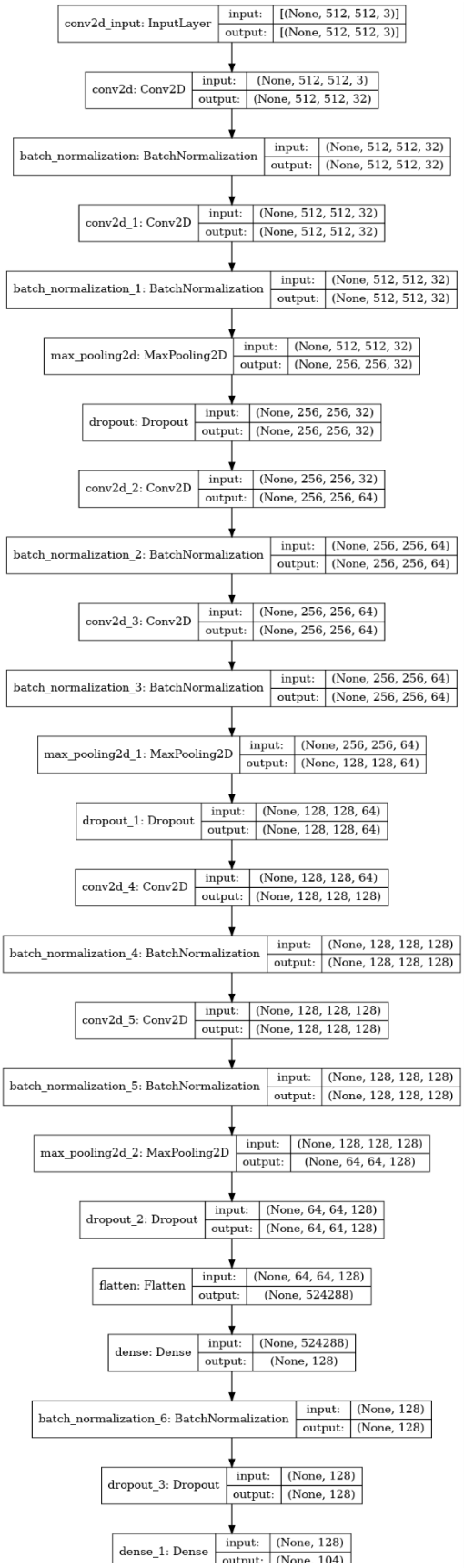
شکل ۳۰) نتایج معماری EfficientNet

- **Custom VGG**

در اینجا نیز یک مدل VGG کاستوم خودمان طراحی کردیم که نتایج و معماری را در پایین مشاهده می‌کنید.

```
model_VGG = tf.keras.Sequential()

# VGG_1
model_VGG.add(tf.keras.layers.Conv2D(32, 3, activation='relu', kernel_init
ializer='he_uniform', padding='same', input_shape=(10, 1000, 1)))
model_VGG.add(tf.keras.layers.BatchNormalization())
model_VGG.add(tf.keras.layers.Conv2D(32, 3, activation='relu', kernel_init
ializer='he_uniform', padding='same', input_shape=(10, 1000, 1)))
model_VGG.add(tf.keras.layers.BatchNormalization())
model_VGG.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
model_VGG.add(tf.keras.layers.Dropout(0.2))

# VGG_2
model_VGG.add(tf.keras.layers.Conv2D(64, 3, activation='relu', kernel_init
ializer='he_uniform', padding='same', input_shape=(10, 1000, 1)))
model_VGG.add(tf.keras.layers.BatchNormalization())
model_VGG.add(tf.keras.layers.Conv2D(64, 3, activation='relu', kernel_init
ializer='he_uniform', padding='same', input_shape=(10, 1000, 1)))
model_VGG.add(tf.keras.layers.BatchNormalization())
model_VGG.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
model_VGG.add(tf.keras.layers.Dropout(0.3))

# VGG_3
```

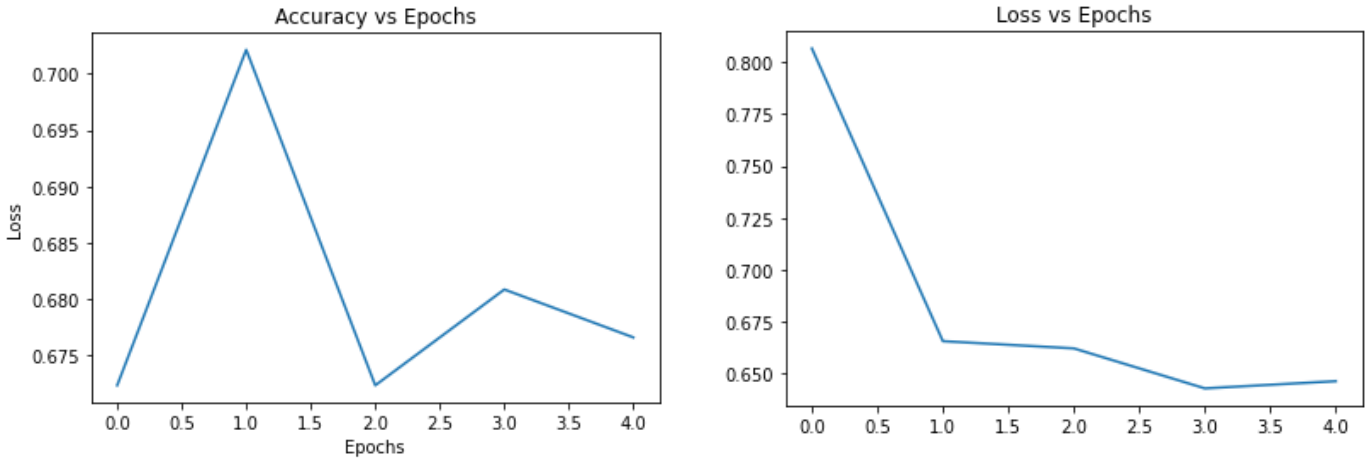
```
model_VGG.add(tf.keras.layers.Conv2D(128, 3, activation='relu', kernel_ini
tializer='he_uniform', padding='same', input_shape=(10, 1000, 1)))
model_VGG.add(tf.keras.layers.BatchNormalization())
model_VGG.add(tf.keras.layers.Conv2D(128, 3, activation='relu', kernel_ini
tializer='he_uniform', padding='same', input_shape=(10, 1000, 1)))
model_VGG.add(tf.keras.layers.BatchNormalization())
model_VGG.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
model_VGG.add(tf.keras.layers.Dropout(0.4))

# Output
model_VGG.add(tf.keras.layers.Flatten())
model_VGG.add(tf.keras.layers.Dense(128, activation='relu', kernel_initial
izer='he_uniform'))
model_VGG.add(tf.keras.layers.BatchNormalization())
model_VGG.add(tf.keras.layers.Dropout(0.5))
model_VGG.add(tf.keras.layers.Dense(1, activation='softmax'))
```



شکل30) معماری شبکه 3 Layer VGG custom

```
[ ] history_VGG = model_VGG.fit(train_gen, epochs=5)

    Epoch 1/5
    235/235 [==============================] - 791s 3s/step - loss: 0.8066 - accuracy: 0.6723
    Epoch 2/5
    235/235 [==============================] - 759s 3s/step - loss: 0.6657 - accuracy: 0.7021
    Epoch 3/5
    235/235 [==============================] - 767s 3s/step - loss: 0.6622 - accuracy: 0.6723
    Epoch 4/5
    235/235 [==============================] - 776s 3s/step - loss: 0.6429 - accuracy: 0.6809
    Epoch 5/5
    235/235 [==============================] - 770s 3s/step - loss: 0.6464 - accuracy: 0.6766

[ ]

[ ] model_VGG.evaluate(validation_gen)

    47/47 [==============================] - 74s 2s/step - loss: 0.7339 - accuracy: 0.5319
    [0.7339455485343933, 0.5319148898124695]
```

شکل۳۱) آموزش شبکه VGG



شکل۳۲) نتایج شبکه VGG

- **نتیجه گیری**

در مقاله رفرنس ما که در منابع ذکر شـــده محققان شـــبکه resnet شـــان را در ۱۰۰۰۰اپوک با کارت گرافیک GTX1080 Ti آموزش دادند که به دقت ۹۳ درصدی در ۲ ساعت رسیدند با توجه به سخت افزار محدود ما و بهینه نبودن مدلمان در طی ۵ اپوک به دقت تقریبا ۷۰ درصدی رسیدیم نشان از عملکرد خوب شبکمان است.

- Reference

ECG Heartbeat Classification: A Deep Transferable Representation – [Mohammad Kachuee, Shayan Fazeli, Majid Sarrafzadeh University of California, Los Angeles (UCLA) Los Angeles, USA]