به نام خداوند جان و خرد

دانشگاه تهران

دانشکدگان فنی

دانشکده مهندسی برق و کامپیوتر

یادگیری ماشین

تمرین شماره ۴

نام و نام خانوادگی: علی خرم فر

شماره دانشجویی: ۸۱۰۱۰۲۱۲۹

خردادماه ۱۴۰۳

فهرست مطالب

۱_ پاسخ سوال ۱

۱-۱_ الف) هدف توابع فعال‌ساز در شبکه‌های MLP

توابع فعال‌ساز در شبکه‌های عصبی MLP نقش بسیار مهمی ایفا می‌کنند و هدف اصلی آنها اعمال نوعی تبدیل غیرخطی بر روی ورودی‌هاست. در ادامه برخی از این دلایل را بررسی می‌کنیم:

بدون توابع فعال‌ساز، هر لایه از شبکه تنها یک ترکیب خطی از ورودی‌های خود را تولید می‌کند. با اضافه کردن توابع فعال‌ساز غیرخطی، شبکه می‌تواند الگوهای پیچیده‌تری را یاد بگیرد که قادر به مدل‌سازی و پیش‌بینی داده‌های غیرخطی هستند. به طور کلی این توابع به شبکه کمک می‌کند تا الگوهای پیچیده‌تر و ویژگی‌های غیرخطی موجود در داده‌ها را شناسایی کرده و آنها را مدل‌سازی کرده تا تا بهتر بتواند داده‌ها را دسته‌بندی یا پیش‌بینی کند.همچنین توابع فعال‌ساز می‌توانند بر سرعت و دقت همگرایی مدل تأثیر مثبت بگذارند.

مشکلات عدم استفاده از توابع فعال‌ساز

اگر از توابع فعال‌ساز استفاده نشود، شبکه MLP به یک مدل خطی تبدیل می‌شود، به این معنا که هر لایه تنها یک ترکیب خطی از لایه قبلی خواهد بود. در نتیجه، شبکه نمی‌تواند الگوهای غیرخطی و پیچیده موجود در داده‌ها را یاد بگیرد که این مورد منجر به کاهش قدرت یادگیری شبکه می‌شود بنابراین توانایی پیش‌بینی آن به شدت کاهش می‌یابد. از طرفی شبکه قادر نخواهد بود داده‌ها را به درستی تفکیک کند، زیرا تمامی لایه‌ها تنها ترکیبات خطی از داده‌های ورودی خواهند بود و نمی‌توانند تفاوت‌های ظریف و پیچیده را شناسایی کنند.

در نتیجه، استفاده از توابع فعال‌ساز برای غیرخطی کردن خروجی‌ها و افزایش توانایی یادگیری و تفکیک‌پذیری شبکه MLP ضروری است.

۲-۱_ ب) مشکل رایج تابع فعال‌سازی Sigmoid

تابع فعال‌سازی Sigmoid که به صورت ریاضی به صورت زیر تعریف می‌شود، در گذشته به طور گسترده‌ای در شبکه‌های عصبی مورد استفاده قرار می‌گرفت. با این حال، این تابع دارای چندین مشکل است که در ادامه آن‌ها را بررسی می‌کنیم:

$$S(x) = \frac{1}{1 + e^{-x}}$$

$S(x)$ = sigmoid function
e = Euler's number

مشکل محوشدگی گرادیان یا Vanishing Gradient

تابع Sigmoid دارای خروجی بین ۰ و ۱ است و مشتق آن نیز بین ۰ و ۰.۲۵ قرار دارد. هنگامی که ورودی به این تابع بسیار بزرگ یا بسیار کوچک باشد، گرادیان به مقادیر بسیار کوچکی نزدیک می‌شود. این مسئله باعث می‌شود که در backpropagation، گرادیان‌ها به تدریج کوچک شده و به سمت صفر میل کرده و در نتیجه گرادیان‌های کوچک باعث می‌شوند که وزن‌های لایه‌های اولیه به کندی تغییر کند و به سختی بتوانند یاد بگیرند، که این مسئله باعث کندی روند آموزش و ناتوانی در یادگیری مدل می‌شود.

خروجی‌های نزدیک به صفر یا یک

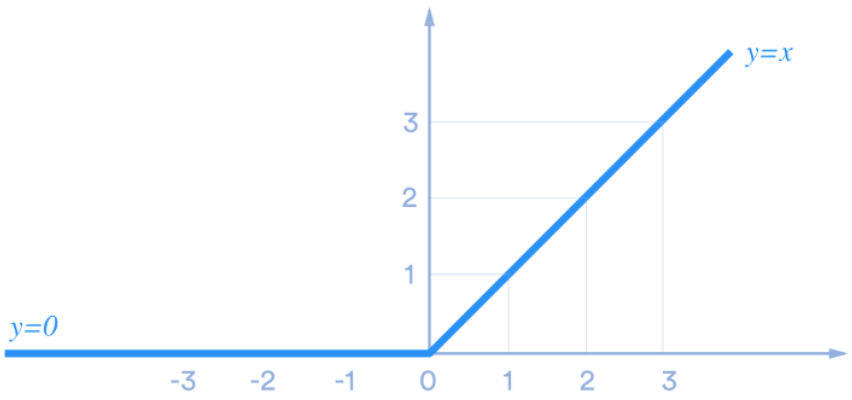
خروجی تابع Sigmoid در ورودی‌های بسیار بزرگ یا بسیار کوچک به ۰ یا ۱ نزدیک می‌شود. این حالت باعث می‌شود که نورون‌ها با تغییرات ورودی تاثیر کمی بر خروجی داشته باشد.

تابع فعال‌سازی جایگزین ReLU

کی از توابع فعال‌سازی که می‌تواند مشکلات تابع Sigmoid را برطرف کند، تابع فعال‌سازی ReLU (Rectified Linear Unit) است که به صورت زیر تعریف می‌شود:

$$f(x) = max(0,x)$$

شکل تابع آن نیز به صورت زیر است:

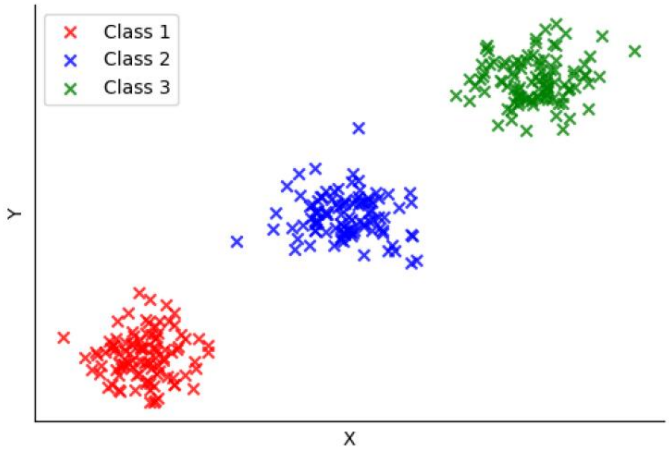


شکل ۱ تابع ReLU

تابع ReLU خروجی‌های منفی را صفر کرده و مقادیر مثبت را بدون تغییر عبور می‌دهد.در اینجا برای ورودی‌های مثبت، مشتق تابع ReLU برابر ۱ است که از محوشدگی گرادیان جلوگیری می‌کند و باعث می‌شود گرادیان‌ها به مقادیر بزرگ‌تری نسبت به Sigmoid برسند. از طرفی محاسبه تابع ReLU بسیار ساده و سریع است، که باعث بهبود کارایی محاسباتی می‌شود.

۳-۱_ ج) طراحی شبکه MLP برای دسته‌بندی

با توجه به تصویر زیر که شامل سه کلاس مختلف ۱، ۲ و ۳ است، هدف طراحی یک شبکه MLP است که بتواند این داده‌ها را با کمترین تعداد لایه و نرون دسته‌بندی کند.



شکل ۲ داده‌های هدف برای دسته‌بندی

باتوجه به شکل بالا ، داده‌ها به خوبی از هم جدا شده‌اند و نیازی به لایه‌های پیچیده برای تفکیک آن‌ها نیست. هرچند که نسبت به طبقه‌بند باینری پیچیدگی بیشتری داریم. اگر فرضا فقط داده‌های قرمز و آبی بودند، کافی بود با دو لایه و بدون نیاز به لایه مخفی طبقه‌بندی را انجام دهیم

ورودی‌ها (Input Layer):

تعداد ورودی‌ها برابر با تعداد ویژگی‌های هر داده است که در اینجا ۲ (X و Y) است.

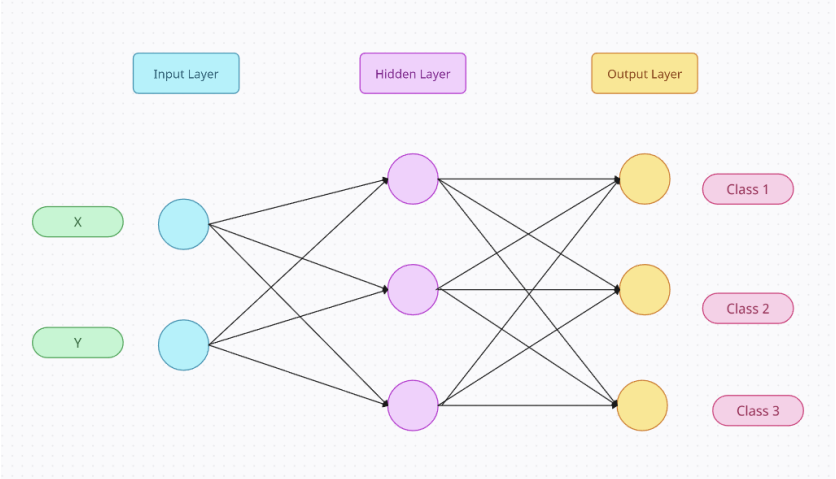
لایه‌های مخفی (Hidden Layers):

با توجه به سادگی تفکیک داده‌ها، یک لایه مخفی کافی است. از آنجایی که تفکیک داده‌ها ساده است، یک یا دو نرون کافی خواهد بود.باتوجه به شبیه‌سازی انجام شده برای این سوال ما از دو نرون استفاده می‌کنیم.

لایه خروجی (Output Layer):

تعداد خروجی‌ها برابر با تعداد کلاس‌ها است که در اینجا ۳ است. از تابع فعال‌سازی Softmax در لایه خروجی استفاده می‌کنیم تا احتمال تعلق هر داده به هر کلاس محاسبه شود.

برای ترسیم این شبکه نیز از ابزار creately.com استفاده شد:



شکل ۳ شبکه MLP برای دسته‌بندی داده‌ها

از تابع فعال‌سازی Softmax در لایه خروجی و ReLU در لایه مخفی استفاده می‌کنیم.

۴-۱_ د) طراحی شبکه MLP برای تابع

T(x) به صورت زیر تعریف شده است:

$$T(x) = \begin{cases} 3x & \text{if } 0 \leq x < \frac{1}{3} \\ \frac{3}{2}(1-x) & \text{if } \frac{1}{3} \leq x \leq 1 \end{cases} \quad x \in \mathbb{R}$$

هدف ما طراحی یک شبکه MLP با استفاده از تابع فعال‌سازی ReLU است به نحوی که معادل تابع بالا باشد.

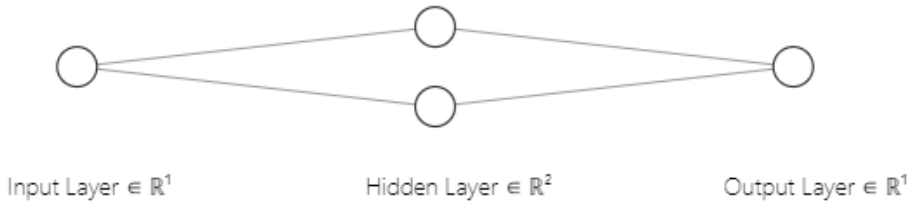
ورودی (Input Layer): دارای ۱ نرون که مقدار x را به عنوان ورودی می‌گیرد. این مقدار می‌تواند هر عددی در بازه [۰ ,۱] باشد.

لایه مخفی (Hidden Layer): شامل ۲ نرون با تابع فعال‌سازی ReLU است. دلیل استفاده از این تابع توانایی آن در مدل‌سازی روابط غیرخطی است.

لایه خروجی (Output Layer): شامل ۱ نرون با تابع فعال‌سازی خطی برای تولید خروجی نهایی تابع فعال‌سازی خطی انتخاب شده است زیرا خروجی نهایی تابع مورد نظر ما یک مقدار پیوسته است.

شکل نهایی شبکه به صورت زیر است اینبار از ابزار در لینک زیر استفاده شد:

https://alexlenail.me/NN-SVG/index.html



شکل ۴ شبکه MLP برای پیاده‌سازی تابع

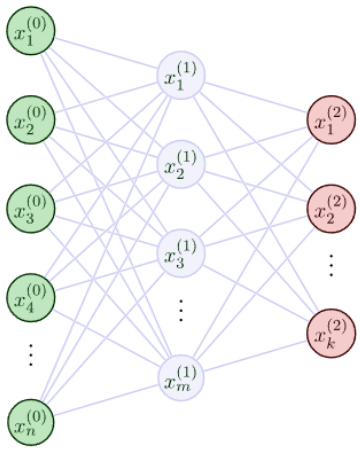
۵-۱_ ه) مدل MLP به عنوان یک Universal Approximator

تئوری Universal Approximation Theorem

این تئوری بیان می‌کند که یک شبکه عصبی چند لایه با حداقل یک لایه مخفی و تابع فعال‌سازی غیرخطی (مانند Sigmoid یا ReLU)، می‌تواند هر تابع پیوسته‌ای را بر روی یک بازه بسته و محدود با دقت دلخواه تقریب بزند، به شرطی که تعداد نرون‌های لایه مخفی کافی باشد.

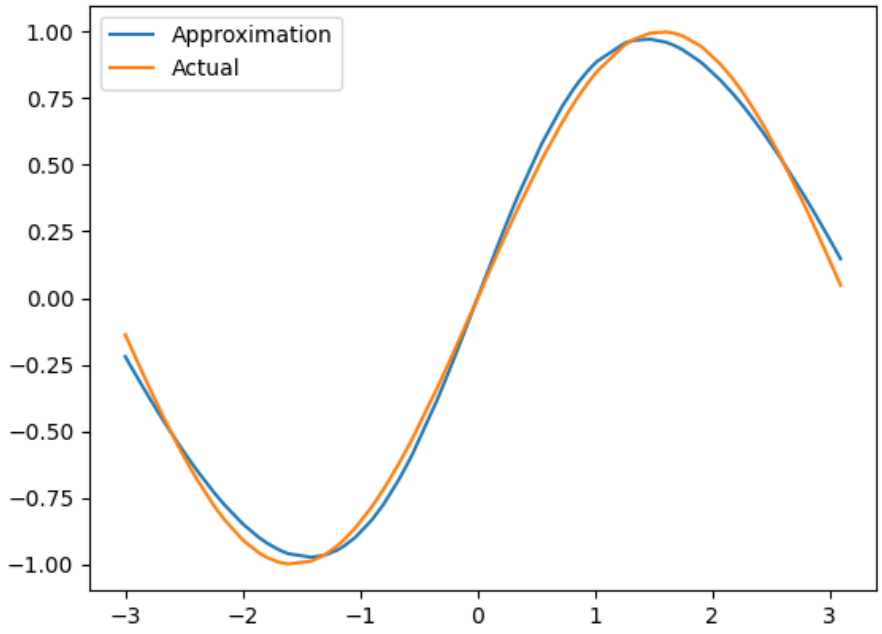
دلایل و مثال‌ها

شبکه‌های عصبی با ترکیب خطی ورودی‌ها و استفاده از توابع فعال‌سازی غیرخطی، می‌توانند روابط پیچیده و غیرخطی بین ورودی و خروجی را مدل‌سازی کنند. این خاصیت به این معناست که یک شبکه عصبی چند لایه با یک لایه مخفی با تابع فعال‌سازی غیرخطی (مانند Sigmoid یاReLU) و تعداد کافی نرون‌ها می‌تواند هر تابع پیوسته‌ای را با دقت دلخواه تقریب بزند.



شکل ۵ معماری شبکه به عنوان یک Universal Approximator

برای مثال در تابع سینوس در بازه $[-\pi, +\pi]$ که یک تابع پیچیده و غیرخطی است.MLP با یادگیری و تنظیم وزن‌ها و بایاس‌های خود می‌تواند شکل این تابع را با دقت بالا تقریب بزند. لایه‌های پنهان به عنوان فیلتری عمل می‌کنند که جزئیات و ویژگی‌های داده‌های ورودی را استخراج و مدل‌سازی می‌کنند. توابع فعال‌سازی غیرخطی به شبکه این امکان را می‌دهند که نه تنها روابط خطی بلکه روابط غیرخطی پیچیده را نیز قابل یادگیری باشد. به همین دلیل، یک *MLP* با ساختار مناسب می‌تواند هر تابع پیوسته را با دقت دلخواه تقریب بزند و به عنوان یک **Universal Approximator** عمل کند.



شکل ۶ تقریب تابع Sin با کمک MLP

۲_پاسخ سوال ۲

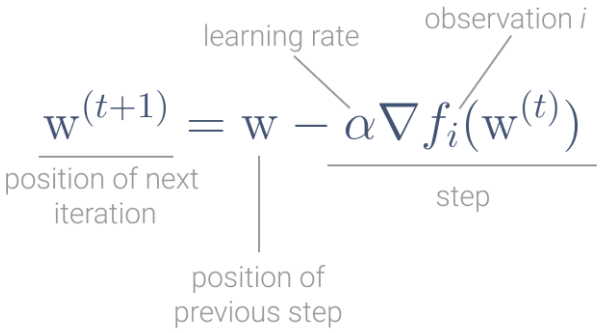
۲-۱_ پاسخ قسمت الف)

پاسخ به صورت فایل جداگانه پیوست شد.

۲-۲_ پاسخ قسمت ب)

الگوریتم Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) یکی از الگوریتم‌های مهم و پرکاربرد در بهینه‌سازی و یادگیری ماشین است که برای یافتن نقاط بهینه تابع هزینه استفاده می‌شود که به‌ویژه برای مجموعه داده‌های بزرگ بسیار پرکاربرد است. در روش Gradient Descent معمولی، گرادیان کل تابع هزینه نسبت به تمام داده‌ها محاسبه می‌شود، که در صورت بزرگی داده‌ها زمان‌بر و پرهزینه است. اما در SGD، به‌جای استفاده از کل داده‌ها، از یک نمونه تصادفی یا یک mini-batch کوچک از داده‌ها برای محاسبه گرادیان استفاده شده که این مورد باعث می‌شود که الگوریتم سریع‌تر و کارآمدتر باشد. همچنین این ویژگی باعث می‌شود که SGD به‌طور خاص در مسائل یادگیری عمیق که با مجموعه داده‌های بسیار بزرگ سروکار دارند، محبوب باشد.



مراحل اجرای SGD به این صورت است که ابتدا یک مقدار اولیه برای پارامترها انتخاب می‌شود. سپس برای هر نمونه یا mini-batch در داده‌ها، گرادیان تابع هزینه نسبت به پارامترها محاسبه شده و پارامترها با استفاده از نرخ یادگیری به‌روزرسانی می‌شوند. این فرآیند تا زمانی که معیار توقف مانند تعداد تکرارها یا تغییرات کوچک در تابع هزینه فعال شود، ادامه می‌یابد. یکی از بزرگترین مزایای SGD این است که به بهبود سرعت همگرایی به دلیل به‌روزرسانی‌های کمک کرده و باعث می‌شود تا از مینیمم‌های محلی فرار کند و به سمت مینیمم‌های بهتری حرکت کند.

با این حال، نوسانات بیشتر در مسیر همگرایی یکی از چالش‌های مهم این الگوریتم است. به دلیل طبیعت تصادفی آن، همگرایی بهینه ممکن است به آهستگی انجام شود و نیاز به تنظیم مناسب نرخ یادگیری وجود دارد. اگر نرخ یادگیری بسیار بزرگ باشد، الگوریتم ممکن است به نوسانات بیش از حد دچار شود و اگر

بسیار کوچک باشد، همگرایی بسیار کند خواهد بود. بنابراین، تعیین یک نرخ یادگیری مناسب برای دستیابی به عملکرد بهینه ضروری است.

روش نیوتون-رافسون- Newton–Raphson Method

روش Newton–Raphson نیز یکی دیگر از الگوریتم‌های قدرتمند برای بهینه‌سازی است که به‌ویژه در مسائل یافتن ریشه‌ها و نقاط مینیمم یا ماکزیمم توابع کاربرد دارد. این روش از اطلاعات گرادیان و همچنین هسین یا ماتریس مشتق دوم استفاده می‌کند تا نقاط بهینه را با سرعت بیشتری پیدا کند. بر خلاف روش‌های مبتنی بر گرادیان ساده که فقط از اطلاعات مشتق اول استفاده می‌کند، نیوتون-رافسون با استفاده از مشتقات دوم مسیر بهینه‌تری برای به‌روزرسانی پارامترها ارائه می‌دهد.

این روش شامل مقداردهی اولیه به پارامترها و سپس به‌روزرسانی آن‌ها با استفاده از اطلاعات گرادیان و هسین است. فرمول به‌روزرسانی پارامترها در این روش به صورت زیر است:

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$$

است که در آن f'' هسین تابع هزینه و f' گرادیان تابع هزینه در نقطه xk است. این فرآیند تا زمانی که معیار توقف مانند تغییرات کوچک در تابع هزینه یا پارامترها برآورده شود، ادامه می‌یابد.

یکی از مزایای برجسته این روش، همگرایی سریع‌تر نسبت به روش‌های گرادیانی ساده است، خصوصاً زمانی که نزدیک به نقطه بهینه قرار داریم. با این حال، محاسبه هسین می‌تواند بسیار پرهزینه باشد، به‌ویژه در مسائل با تعداد پارامترهای زیاد. این پیچیدگی محاسباتی باعث می‌شود که روش نیوتون-رافسون برای مسائل بزرگ‌تر و پیچیده‌تر کمتر مناسب باشد، مگر اینکه منابع محاسباتی کافی در دسترس باشد. علاوه بر این، در توابعی که هسین آن‌ها مثبت معین نیست، ممکن است مشکلاتی پیش بیاید که مانع از همگرایی شود.

در مجموع، انتخاب بین روش‌های مختلف بهینه‌سازی مانند SGD و نیوتون-رافسون بستگی به نوع مسئله، اندازه داده‌ها، منابع محاسباتی در دسترس و نیازهای خاص هر مسئله دارد. در حالی که SGD برای مسائل با داده‌های بزرگ و پویا مناسب است، نیوتون-رافسون در مسائل با داده‌های کمتر و نیاز به همگرایی سریع‌تر کاربرد دارد. فهم دقیق این الگوریتم‌ها و تنظیم پارامترهای مناسب برای هر کدام می‌تواند به بهینه‌سازی موثرتر و حل مسائل پیچیده‌تر کمک کند.

۳_ پاسخ سوال ۳

۳-۱_ ویژگی Transitional Invariance

یکی از ویژگی‌های مهم شبکه‌های عصبی کانولوشنی CNN که آن‌ها را از سایر انواع شبکه‌های عصبی متمایز می‌کند، ویژگی Transitional Invariance است. این ویژگی به شبکه‌های عصبی کانولوشنی اجازه می‌دهد تا الگوها و ویژگی‌های یکسان در تصاویر را صرف‌نظر از محل قرارگیری آن‌ها شناسایی کند. این بدیهی است که اشیای مختلف در تصاویر ممکن است در قسمت‌های مختلفی ظاهر شده و لزوما همیشه در پیکسل‌هایی خاص در تصویر نباشد. به عبارت دیگر، اگر یک الگو یا ویژگی در قسمت‌های مختلفی از تصویر ظاهر شود، شبکه با استفاده از ویژگی Transitional Invariance قادر به شناسایی آن الگو خواهد بود.

۳-۲_ بررسی اجزا

لایه‌های کانولوشنی (Convolutional Layers)

لایه‌های کانولوشنی اصلی‌ترین بخش CNNها هستند که نقش کلیدی در ایجاد ویژگی Transitional Invariance ایفا می‌کنند. این لایه‌ها شامل مجموعه‌ای از فیلترها یا کرنل‌ها هستند که بر روی ورودی تصویر اعمال شده و هر فیلتر به صورت محلی و با حرکت در سراسر تصویر (Sliding) عملیات کانولوشن را انجام می‌دهد. در ادامه هرکدام از اجزای این لایه را بررسی می‌کنیم:

فیلترها (Filters): فیلترها مجموعه‌ای از وزن‌ها هستند که در یک پنجره کوچک اعمال می‌شوند. هر فیلتر به دنبال ویژگی‌های خاصی در تصویر می‌گردد، مانند لبه‌ها یا گوشه ها. با حرکت در سراسر تصویر، فیلتر در نقاط مختلف تصویر اعمال شده که به شبکه کمک می‌کند تا الگوها را در موقعیت‌های مختلف شناسایی کند.

Sliding Window: حرکت فیلتر به صورت Sliding در تمام تصویر به معنای این است که هر بخش از تصویر توسط فیلتر بررسی شده که این حرکت منجر به تولید Feature Maps می‌شود که مکان ویژگی‌ها را در تصویر نشان می‌دهد.

لایه‌های Pooling:

لایه‌های Pooling نیز نقش مهمی در کاهش ابعاد نقشه‌های ویژگی و افزایش مقاومت شبکه به جابجایی‌های کوچک در تصویر ایفا می‌کنند که در ادامه انواع آن را بررسی می‌کنیم:

Max Pooling: یکی از رایج‌ترین انواع لایه‌های Pooling که به صورت محلی بر روی Feature maps اعمال و تنها بزرگ‌ترین مقدار در هر پنجره را انتخاب می‌کند. این کار باعث کاهش ابعاد Feature maps و حفظ اطلاعات مهم می‌شود. به دلیل انتخاب بزرگ‌ترین مقدار، مکان دقیق ویژگی در پنجره مورد نظر اهمیت کمتری پیدا می‌کند، که منجر به افزایش مقاومت شبکه به تغییرات مکانی کوچک می‌شود.

Average Pooling: این نوع Pooling میانگین مقادیر در هر پنجره را محاسبه می‌کند. اگرچه استفاده از آن کمتر رایج است، اما همچنان به کاهش حساسیت شبکه به تغییرات مکانی کمک می‌کند.

Strides و Padding:

تنظیم Strides و Padding در عملیات کانولوشن در افزایش تاثیر Transitional Invariance حائز اهمیت است. استفاده از Strides بزرگ‌تر منجر به کاهش ابعاد Feature maps و افزایش مقاومت به جابجایی‌های بزرگ‌تر می‌شود. انتخاب مناسب Strides می‌تواند به ایجاد تعادل بین دقت و مقاومت شبکه در برابر جابجایی کمک کند. همچنین Padding امکان حفظ ابعاد Feature maps و جلوگیری از از دست رفتن اطلاعات حاشیه ی تصویر را فراهم می‌کند. این کار به شبکه اجازه می‌دهد تا الگوها را در لبه‌های تصویر نیز شناسایی شوند.

در انتهای شبکه‌های CNN، معمولاً یک یا چند لایه Fully Connected قرار دارند که نقش طبقه‌بندی نهایی را بر عهده دارند. اگرچه این لایه‌ها به طور مستقیم در ایجاد ویژگی Transitional Invariance نقشی ندارند، اما اطلاعات استخراج شده از لایه‌های کانولوشنی و Pooling را برای تصمیم‌گیری نهایی ترکیب می‌کنند.

۳-۳_ پاسخ قسمت ج – پیاده‌سازی

MLP با معماری دلخواه

ابتدا کتابخانه‌های مورد نیاز را Import می‌کنیم. برای این تمرین از پکیج Keras استفاده شد. پس از آن مجموعه داده MNIST را بارگذاری و نرمال می‌کنیم تا مقادیر به بازه ۰ تا ۱ تبدیل شوند. پس از آن داده‌ها را به فرمت مناسب برای ورودی شبکه MLP تبدیل می‌کنیم. داده‌های تصویری به صورت تک‌بعدی (Flat) تبدیل می‌شوند.

در مرحله بعد یک شبکه MLP با معماری زیر طراحی می‌کنیم. این شبکه شامل سه لایه مخفی با تعداد نرون‌های متفاوت و یک لایه خروجی با ۱۰ نرون بوده که به تعداد دسته‌های MNIST است.

```
mlp_model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 512)	401,920
dense_1 (Dense)	(None, 256)	131,328
dense_2 (Dense)	(None, 128)	32,896
dense_3 (Dense)	(None, 10)	1,290

Total params: 567,434 (2.16 MB)
Trainable params: 567,434 (2.16 MB)
Non-trainable params: 0 (0.00 B)

پس از Compile مدل و آموزش آن بر روی داده‌های آموزشی در ۱۰ ایپاک و اندازه Bath ۱۲۸ تایی نتیجه زیر حاصل شد:

```
mlp_model.fit(train_images_mlp, train_labels, epochs=10, batch_size=128, validation_split=0.2)
```

```
Epoch 1/10
375/375 ━━━━━━━━━━━━━━━━━━━━ 5s 10ms/step - accuracy: 0.8552 - loss: 0.4886 - val_accuracy: 0.9582 - val_loss: 0.1354
Epoch 2/10
375/375 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - accuracy: 0.9700 - loss: 0.0980 - val_accuracy: 0.9733 - val_loss: 0.0905
Epoch 3/10
375/375 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - accuracy: 0.9820 - loss: 0.0592 - val_accuracy: 0.9729 - val_loss: 0.0919
Epoch 4/10
375/375 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - accuracy: 0.9866 - loss: 0.0411 - val_accuracy: 0.9701 - val_loss: 0.0994
Epoch 5/10
375/375 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - accuracy: 0.9899 - loss: 0.0311 - val_accuracy: 0.9710 - val_loss: 0.1105
Epoch 6/10
375/375 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - accuracy: 0.9913 - loss: 0.0260 - val_accuracy: 0.9744 - val_loss: 0.0942
Epoch 7/10
375/375 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9933 - loss: 0.0194 - val_accuracy: 0.9744 - val_loss: 0.0938
Epoch 8/10
375/375 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - accuracy: 0.9950 - loss: 0.0151 - val_accuracy: 0.9746 - val_loss: 0.1013
Epoch 9/10
375/375 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - accuracy: 0.9929 - loss: 0.0193 - val_accuracy: 0.9739 - val_loss: 0.1084
Epoch 10/10
375/375 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - accuracy: 0.9950 - loss: 0.0155 - val_accuracy: 0.9778 - val_loss: 0.0989
```

دقت این مدل به صورت زیر است:

MLP Test accuracy: 0.9779000282287598

پس شبکه MLP توانست دقت حدود ۹۷ درصدی را بدست آورد.

شبکه LeNet

در این بخش از مسئله شبکه LeNet را با استفاده از کتابخانه Keras و مطابق با معماری نشان داده شده در تصویر پیاده‌سازی می‌کنیم.

```
lenet_model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 6)	156
average_pooling2d (AveragePooling2D)	(None, 14, 14, 6)	0
conv2d_1 (Conv2D)	(None, 10, 10, 16)	2,416
average_pooling2d_1 (AveragePooling2D)	(None, 5, 5, 16)	0
flatten (Flatten)	(None, 400)	0
dense_4 (Dense)	(None, 120)	48,120
dense_5 (Dense)	(None, 84)	10,164
dense_6 (Dense)	(None, 10)	850

Total params: 61,706 (241.04 KB)
Trainable params: 61,706 (241.04 KB)
Non-trainable params: 0 (0.00 B)

لایه ورودی: با ابعاد ۲۸ در ۲۸ و یک کانال

لایه کانولوشنی اول: ۶ فیلتر با اندازه ۵در ۵، تابع Sigmoid و پدینگ از نوع 'same'

لایه Average Pooling اول: با اندازه ۲ در ۲ و strides=2

لایه کانولوشنی دوم: ۱۶ فیلتر با اندازه ۵در۵، تابع Sigmoid و بدون پدینگ

لایه Average Pooling دوم: با اندازه ۲ در ۲ و strides=2

لایه Flatten: برای تبدیل داده‌های ۲ بعدی به ۱ بعد

لایه Dense اول: با ۱۲۰ نرون و تابع Sigmoid

لایه Dense دوم: با ۸۴ نرون و تابع Sigmoid

لایه خروجی: با ۱۰ نرون و تابع Softmax برای طبقه‌بندی ۱۰ دسته

پس از Compile مدل و آموزش آن بر روی داده‌های آموزشی در ۱۰ ایپاک و اندازه Bath 128 تایی نتیجه زیر حاصل شد:

```
lenet_model.fit(train_images_lenet, train_labels, epochs=10, batch_size=128, validation_split=0.2)
```

```
Epoch 1/10
375/375 ━━━━━━━━━━━━━━━━━━━━ 15s 35ms/step - accuracy: 0.1886 - loss: 2.1836 - val_accuracy: 0.8562 - val_loss: 0.5714
Epoch 2/10
375/375 ━━━━━━━━━━━━━━━━━━━━ 13s 36ms/step - accuracy: 0.8768 - loss: 0.4719 - val_accuracy: 0.9207 - val_loss: 0.2693
Epoch 3/10
375/375 ━━━━━━━━━━━━━━━━━━━━ 13s 35ms/step - accuracy: 0.9245 - loss: 0.2597 - val_accuracy: 0.9411 - val_loss: 0.1988
Epoch 4/10
375/375 ━━━━━━━━━━━━━━━━━━━━ 13s 35ms/step - accuracy: 0.9395 - loss: 0.2040 - val_accuracy: 0.9461 - val_loss: 0.1766
Epoch 5/10
375/375 ━━━━━━━━━━━━━━━━━━━━ 14s 37ms/step - accuracy: 0.9498 - loss: 0.1657 - val_accuracy: 0.9578 - val_loss: 0.1375
Epoch 6/10
375/375 ━━━━━━━━━━━━━━━━━━━━ 13s 36ms/step - accuracy: 0.9581 - loss: 0.1408 - val_accuracy: 0.9636 - val_loss: 0.1214
Epoch 7/10
375/375 ━━━━━━━━━━━━━━━━━━━━ 14s 36ms/step - accuracy: 0.9628 - loss: 0.1234 - val_accuracy: 0.9672 - val_loss: 0.1087
Epoch 8/10
375/375 ━━━━━━━━━━━━━━━━━━━━ 20s 35ms/step - accuracy: 0.9676 - loss: 0.1101 - val_accuracy: 0.9728 - val_loss: 0.0925
Epoch 9/10
375/375 ━━━━━━━━━━━━━━━━━━━━ 14s 36ms/step - accuracy: 0.9729 - loss: 0.0880 - val_accuracy: 0.9728 - val_loss: 0.0883
Epoch 10/10
375/375 ━━━━━━━━━━━━━━━━━━━━ 13s 35ms/step - accuracy: 0.9747 - loss: 0.0824 - val_accuracy: 0.9738 - val_loss: 0.0836
<keras.src.callbacks.history.History at 0x7d3b671f9c00>
```

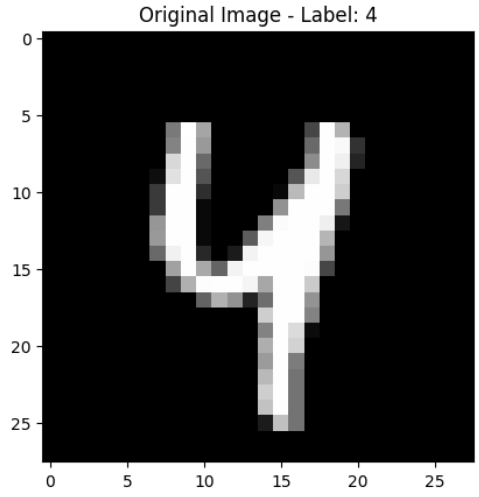
دقت این مدل به صورت زیر است:

LeNet Test accuracy: 0.9768999814987183

پس شبکه LeNet توانست دقت حدود ۹۷ درصدی را بدست آورد. تا اینجا نتیجه گرفتیم هردو مدل توانستند دقت مشابهی بدست آورند. حال برای مقایسه مقاومت آن‌ها دربرابر جابجایی آزمایش بخش بعدی را انجام می‌دهیم.

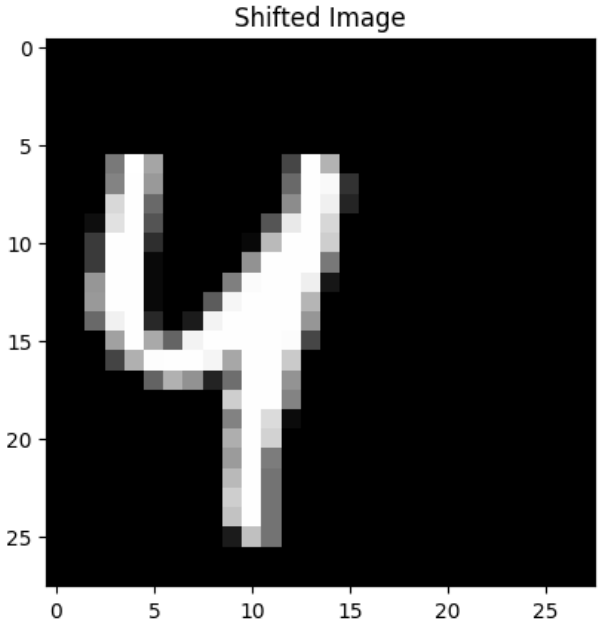
ارزیابی Translation

در این بخش، هدف ما مقایسه مقاومت شبکه‌های عصبی کانولوشنی و شبکه‌های عصبی MLP در برابر جابجایی تصاویر است. ابتدا یک تصویر از مجموعه داده MNIST انتخاب کرده و تابعی برای جابجایی تصویر پیاده‌سازی می‌کنیم:



```
translated_image =
  tf.keras.preprocessing.image.apply_affine_transform(image, tx=dx, ty=dy)
```

لیبل واقعی تصویر انتخاب شده **۴** است. پس از اینکه با کمک Transform ۵ واحد آن را به سمت چپ منتقل کردیم نتیجه زیر حاصل شد:



حال این تصویر را به ورودی هر دو مدل می‌دهیم. نتیجه زیر حاصل شد:

MLP Predicted Label: 6

LeNet Predicted Label: 4

شبکه LeNet توانست عدد جابجا شده را به درستی تشخیص دهد در حالی که شبکه MLP نتوانست. این تفاوت نشان‌دهنده اهمیت ویژگی Transitional Invariance در شبکه‌های CNN است. نیازی به تغییرات دیگر مثل zoom وجود نداشت و با همین جابجایی نتیجه متفاوت حاصل شد.

۴_پاسخ سوال ۴

۴-۱_ پاسخ قسمت الف)

ابتدا محاسبات را به صورت دستی انجام می‌دهیم:

محاسبه وزن‌ها

کلاس مثبت۱ : (2,3) و(1, 4)

کلاس منفی۱:(4,5) و (5,6)

برای کلاس مثبت :

$$w_1 + 4w_2 + b = 1$$
$$2w_1 + 3w_2 + b = 1$$

$$(2w_1 + 3w_2 + b) - (2w_1 + 3w_2 + b) = 1 - 1$$
$$w_1 - w_2 = 0 \rightarrow w_1 = w_2$$
$$w_1 + 4w_2 + b = 1 -> 5w_1 + b = 1 *$$

برای کلاس منفی :

$$4w_1 + 5w_2 + b = -1$$
$$5w_1 + 6w_2 + b = -1$$

$$w_1 = w_2 \rightarrow 9w_1 + b = -1 **$$

حال با توجه به * و **:

$$5w_1 + b = 1$$
$$9w_1 + b = -1$$
$$(5w_1 + b) - (9w_1 + b) = 1 - 1 \rightarrow 4w_1 = -2 \rightarrow w_1 = -\frac{1}{2}$$
$$w_1 = w_2 \rightarrow w_2 = -\frac{1}{2}$$

محاسبه b

$$5w_1 + b = 1 \rightarrow 5\left(-\frac{1}{2}\right) + b = 1 \rightarrow -\frac{5}{2} + b = 1 \rightarrow b = 1 + \frac{5}{2}$$

$$b = \frac{7}{2}$$

نتیجه نهایی محاسبات:

$$w = \left[-\frac{1}{2}, -\frac{1}{2}\right]^T$$

$$b = \frac{7}{2}$$

برای اطمینان از صحت محاسبات، می‌توانیم از توابع آماده مربوط به SVM در پایتون استفاده کنیم.

برای این منظور، از کتابخانه‌های scikit-learn و numpy استفاده می‌کنیم:

```
X = np.array([[2, 3], [1, 4], [4, 5], [5, 6]])
y = np.array([1, 1, -1, -1])
model = SVC(kernel='linear')
model.fit(X, y)
w = model.coef_
b = model.intercept_

print(f'w: {w}')
print(f'b: {b}')
```

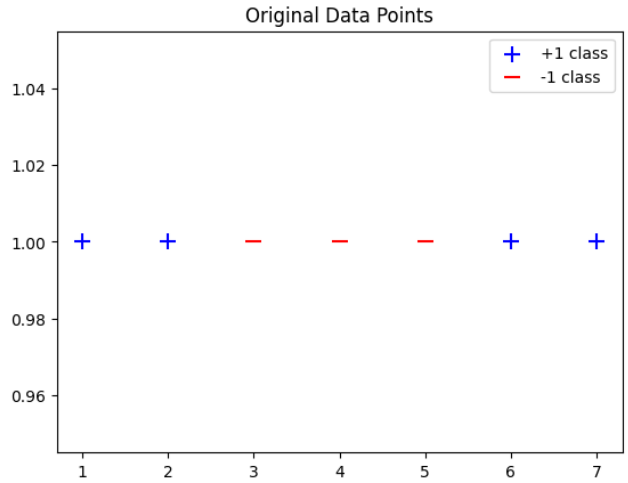
```
w: [[-0.5 -0.5]]
b: [3.5]
```

نتایج محاسبات دستی ما با نتایج حاصل از توابع آماده پایتون تطابق دارد.

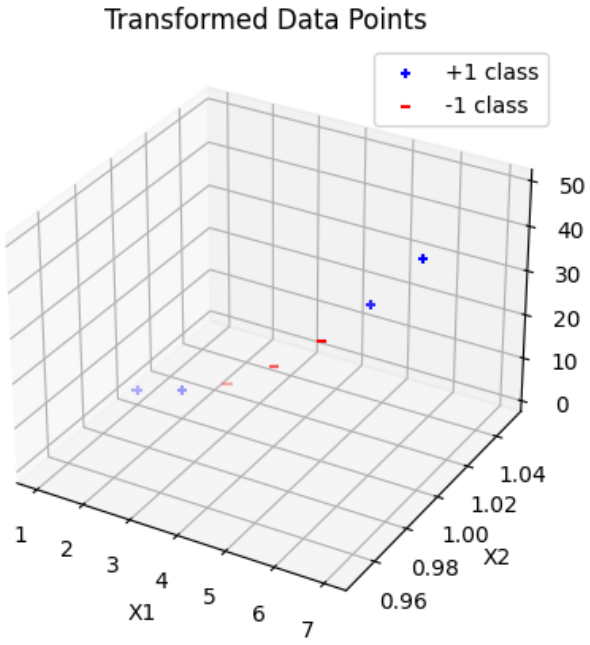
۲-۴_ پاسخ قسمت ب

پایگاه داده اول

ابتدا داده‌های اصلی را در پایتون پیاده‌سازی کرده و نمایش می‌دهیم

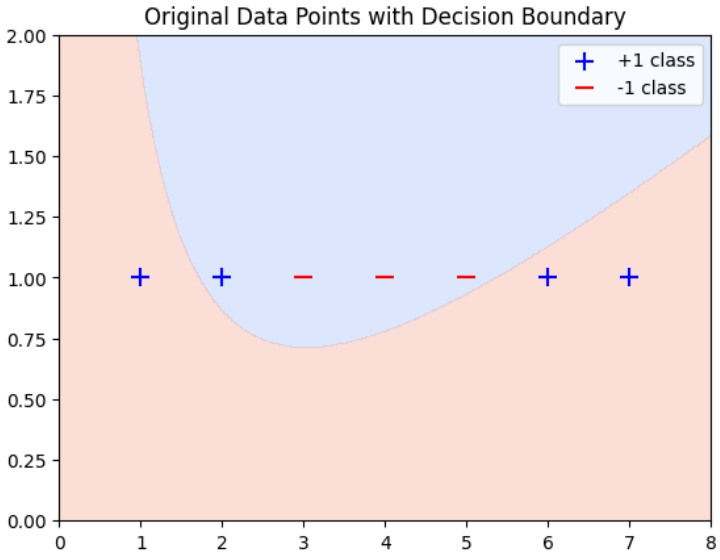


برای تبدیل به فضایی با ویژگی بیشتر، از تبدیل چندجمله‌ای یا Polynomial با درجه استفاده کردیم. این تبدیل ویژگی‌های جدیدی ایجاد کرده که شامل تمام ترکیبات خطی و غیرخطی درجه ۲ از ویژگی‌های اصلی است.پس از تبدیل داده‌ها، آنها در فضای سه‌بعدی نمایش می‌دهیم تا بتوانیم بصری بودن Linearly Seperableشدن آن ها را بررسی کنیم.

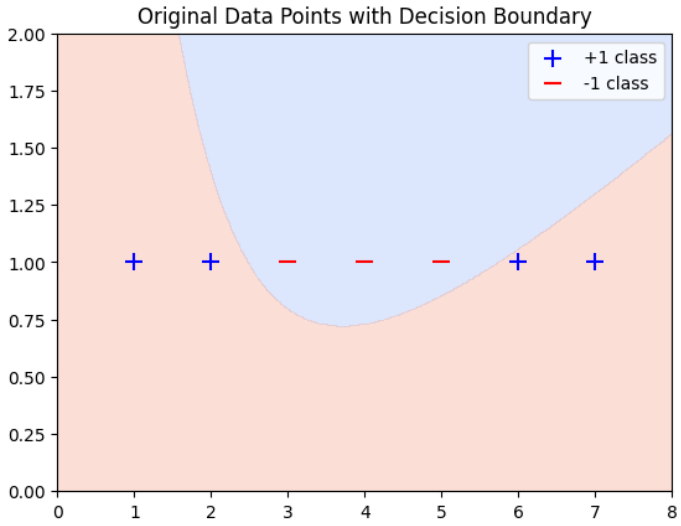


یک مدل SVM با کرنل خطی روی داده‌های تبدیل‌شده آموزش دادیم و دقت آن حدود ۸۶ درصد بود که نشان می‌دهد به صورت خطی داده‌ها قابل تفکیک هستند.

برای نمایش مرز تصمیم‌گیری مدل SVM در فضای اولیه، یک شبکه از نقاط تولید شد و این نقاط با استفاده از مدل SVM پیش‌بینی کردیم.



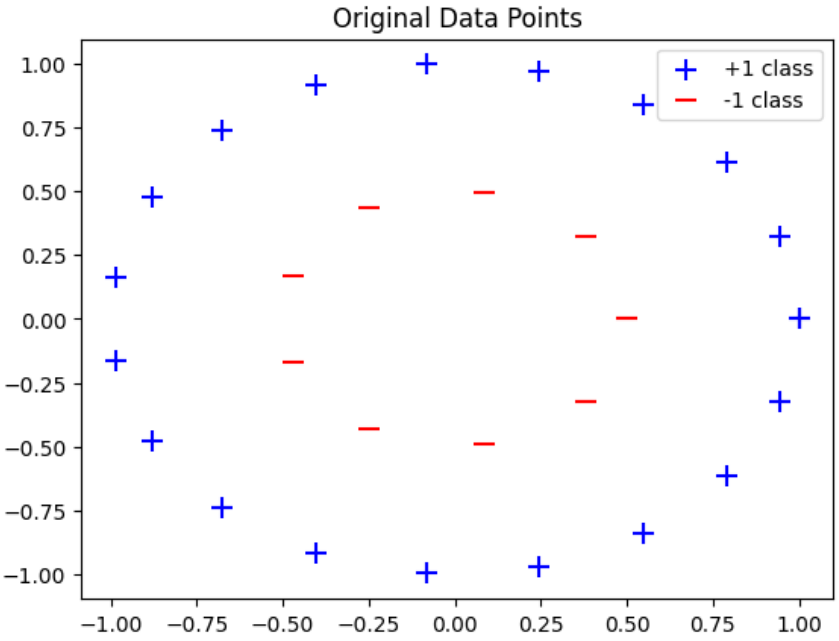
برای اینکه تبدیل به صورت کاملا خطی باشد و دقت برابر ۱ شود، درجه تبدیل چندجمله‌ای را برابر ۳ قرار دادیم که نتیجه زیر در فضای اولیه حاصل شد:



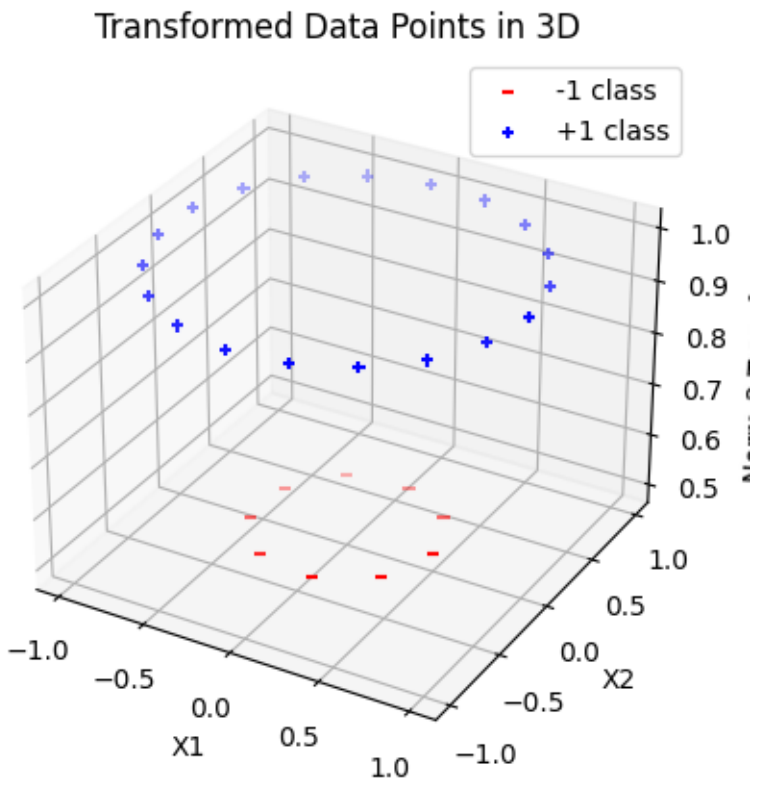
پایگاه داده دوم

تمام موارد قبلی گزارش شده برای این پایگاه داده نیز اجرا شد با این تفاوت که در اینجا از تبدیل **Norm-2** استفاده کردیم. حال نتایج را به صورت نمودار مشاهده می‌کنیم:

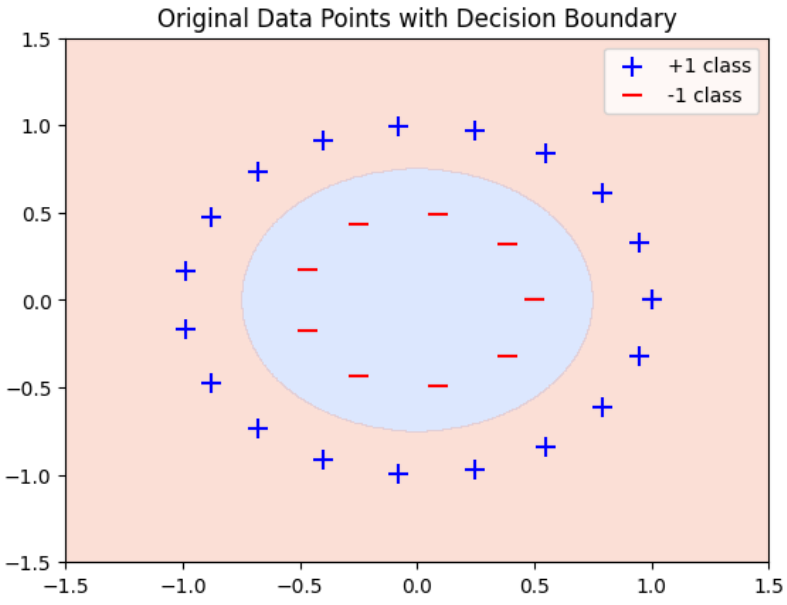
نمودار فضای اولیه:



نقاط در فضای جدید:



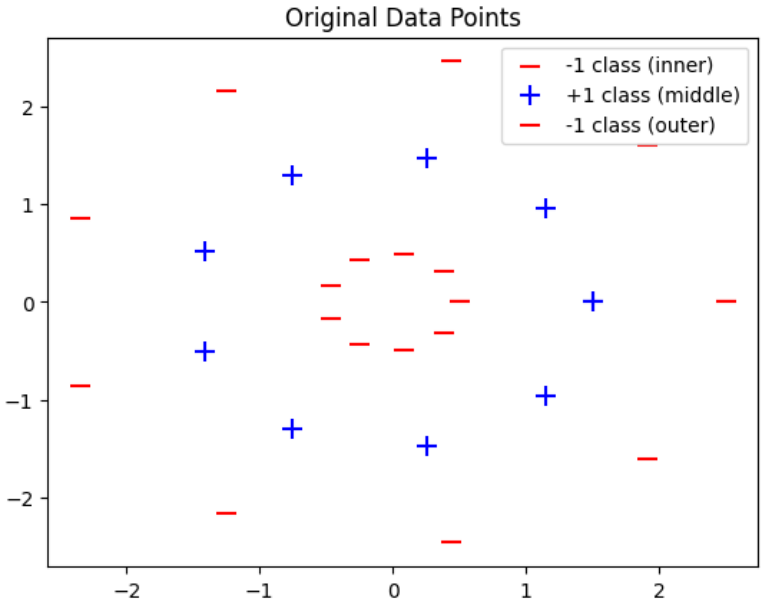
مرز تصمیم در فضای اولیه با کمک SVM بر روی داده‌ای فضای جدید که دقت برابر ۱ حاصل شد:



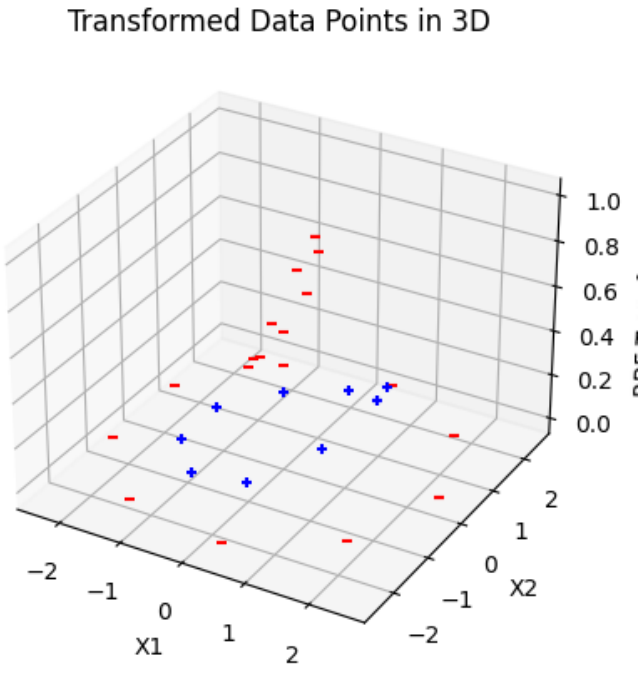
پایگاه داده سوم

تمام موارد قبلی گزارش شده برای این پایگاه داده نیز اجرا شد با این تفاوت که در اینجا از تبدیل **RBF (Radial Basis Function)** استفاده کردیم. این تبدیل هر نقطه را به فاصله **pairwise distance** آن با دیگر نقاط تبدیل می‌کند.حال نتایج را به صورت نمودار مشاهده می‌کنیم:

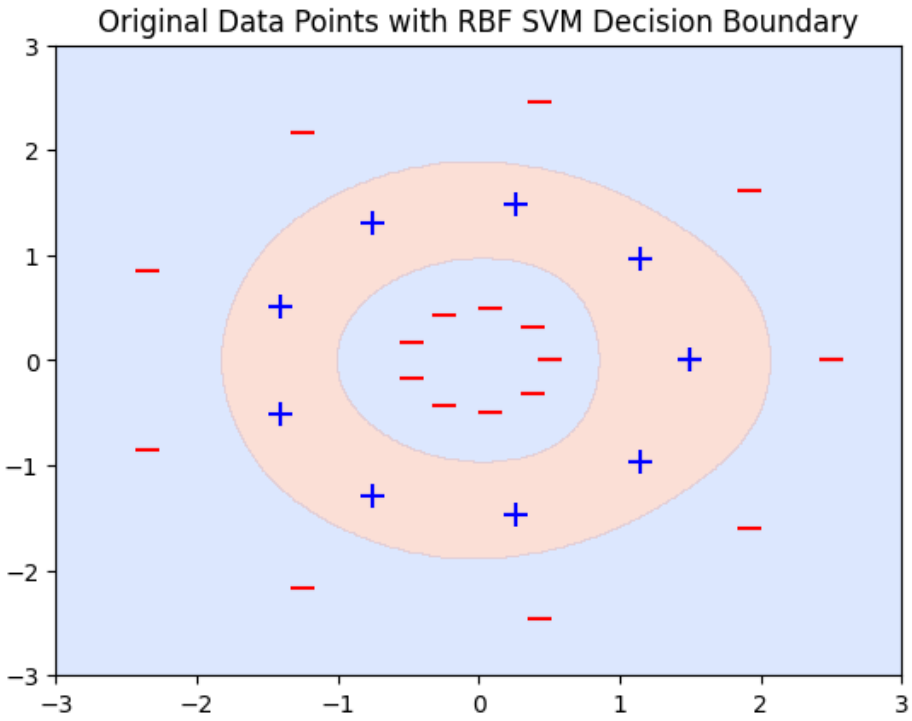
نمودار فضای اولیه:



نقاط در فضای جدید:



مرز تصمیم در فضای اولیه با کمک SVM بر روی داده‌ای فضای جدید که دقت برابر ۱ حاصل شد:



نگاشت‌های استفاده‌شده برای هر پایگاه داده

پایگاه داده اول: Polynomial Transformation

```
poly = PolynomialFeatures(degree=3)
X_transformed = poly.fit_transform(X)[:, 1:]
```

پایگاه داده دوم: Norm-2 یا Euclidean Norm Transformation

```
def norm2_transform(X):
    return np.array([np.sqrt(X[:, 0]**2 + X[:, 1]**2)]).T
X_transformed = norm2_transform(X)
```

پایگاه داده سوم: تبدیل RBF (Radial Basis Function)

```
def rbf_transform(X, gamma=1.0):
    pairwise_sq_dists = np.square(np.linalg.norm(X[:, np.newaxis] - X[np.newaxis, :], axis=2))
    return np.exp(-gamma * pairwise_sq_dists)

X_transformed = rbf_transform(X)
```

در هر یک از این موارد، نگاشت‌های متفاوتی برای تبدیل داده‌ها به فضای ویژگی‌های بالاتر استفاده شد تا داده‌های غیرخطی در فضای اولیه به صورت خطی جداپذیر شوند. استفاده از این نگاشت‌ها و آموزش مدل SVM خطی روی داده‌های تبدیل شده، دقت بالایی را نشان داد و به خوبی داده‌ها را تفکیک کرد.

۵_پاسخ سوال ۵

۵-۱_ مفهوم کرنل

کرنل، یک تابع ریاضی است که برای نگاشت داده‌ها به یک فضای ویژگی بالاتر استفاده می‌شود، به گونه‌ای که در این فضای جدید، داده‌ها ممکن است به صورت خطی جداپذیر شوند. کرنل‌ها اجازه می‌دهند که الگوریتم‌های یادگیری ماشین مثل SVM بدون نیاز به محاسبه صریح مختصات داده‌ها در فضای بالاتر، به طور موثری با داده‌های غیرخطی کار کنند.

۵-۲_ دلایل استفاده از روش‌های مبتنی بر کرنل

جداپذیری خطی یا Linearly Seperable: این مورد را به طور کامل در سوال قبلی بررسی کردیم و مفهوم آن این است که کرنل‌ها داده‌هایی که در فضای اولیه غیرخطی جداپذیر هستند را به فضای ویژگی بالاتری نگاشت می‌کنند که در آن به صورت خطی جداپذیر می‌شوند.

محاسبه کارا: به دلیل محاسبه کرنل بین جفت داده‌ها به جای محاسبه صریح مختصات در فضای بالاتر، کارایی محاسبات افزایش می‌یابد.

انعطاف‌پذیری: کرنل‌ها انواع مختلفی دارند (مانند RBF، Polynomial و..) که هر کدام برای نوع خاصی از داده‌ها مناسب است که برخی از آن‌ها را در سوال قبلی بررسی کردیم.

تعمیم بهتر: استفاده از کرنل‌ها به الگوریتم‌های یادگیری ماشین اجازه می‌دهد که بر روی داده‌های دیده‌نشده تعمیم یا Generalization بهتری داشته باشند و با داده‌های پیچیده و نویزی بهتر کار کنند.

۳-۵_ اثبات

$$K(x,y)^2 \le K(x,x)K(y,y)$$

برای اثبات این خاصیت، از نامساوی کوشی-شوارتز استفاده می‌کنیم. فرض می کنیم ϕ تابع نگاشت به فضای ویژگی بالاتر باشد که با استفاده از کرنل K تعریف می‌شود:

$$K(x,y) = (\phi(x), \phi(y))$$

از طرفی نامساوی کوشی-شوارتز به شکل زیر است:

$$|\phi(x), \phi(y)|^2 \le \big(\phi(x), \phi(x)\big)\big(\phi(y), \phi(y)\big)$$

با جایگذاری $K(x,y)$ به جای $\phi(x), \phi(y)$، داریم:

$$K(x,y)^2 \le K(x,x)K(y,y)$$

بنابراین، نشان داده شد که اگر یک کرنل K به صورت$K(x,y)=(\phi(x),\phi(y))$ تعریف شود، این کرنل معتبر خواهدبود.

۴-۵_ اثبات دوم

پاسخ به صورت فایل جداگانه پیوست شد.

۶_پاسخ سوال ۶

در این مسئله، ما استفاده از ماشین‌های بردار پشتیبان SVM برای طبقه‌بندی تصاویر مجموعه داده MNIST را بررسی می‌کنیم. ابتدا کتابخانه‌های مورد نیاز را Importمیکنیم. داده‌ها از OpenML دریافت شده‌اند و خروجی تعداد داده‌ها نیز مشابه صورت سوال و ۷۰۰۰۰ است.

پس از آن ۱۰۰۰ نمونه تصادفی از مجموعه داده اصلی انتخاب کرده و داده‌های نمونه‌برداری شده را به مجموعه‌های آموزش و تست تقسیم‌بندی می‌کنیم. ۲۰ درصد از داده‌ها به عنوان مجموعه تست و ۸۰ درصد به عنوان مجموعه آموزش استفاده می‌شوند. پارامتر stratify برای اطمینان از این که توزیع برچسب‌ها در هر دو مجموعه آموزش و تست متناسب است، استفاده می‌شود.

پس از تقسیم‌بندی داده‌ها، تعداد نمونه‌های هر برچسب در مجموعه آموزشی را با استفاده از np.unique و return_counts میشماریم و نتایج را در قالب یک دیکشنری ذخیره می‌کنیم. این موضوع به ما اطمینان می‌دهد که هر برچسب حداقل ۱۰ نمونه در مجموعه آموزشی دارد.

```
X_train, X_test, y_train, y_test = train_test_split(X_sample, y_sample, test_size=0.2, random_state=42, stratify=y_sample)
unique_train, counts_train = np.unique(y_train, return_counts=True)
label_counts_train = dict(zip(unique_train, counts_train))
```

+ Code + Markdown

```
print(f'Training set size: {X_train.shape[0]}')
print(f'Test set size: {X_test.shape[0]}')
print('Training set label distribution:', label_counts_train)
```

```
Training set size: 800
Test set size: 200
Training set label distribution: {0: 76, 1: 83, 2: 76, 3: 95, 4: 70, 5: 72, 6: 78, 7: 82, 8: 81, 9: 87}
```

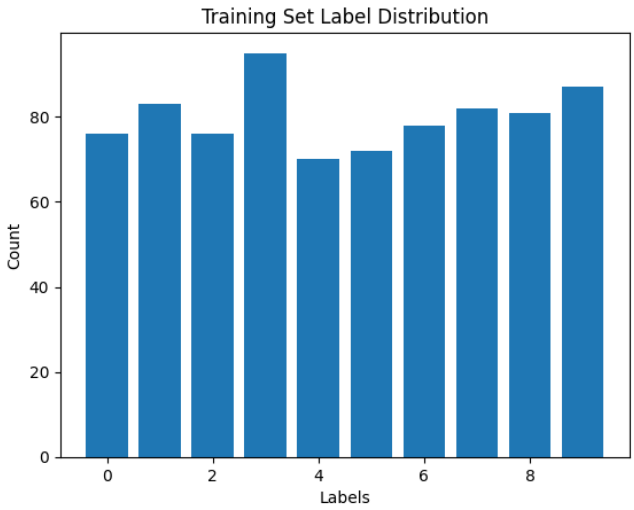
با استفاده از یک شرط بررسی می‌کنیم که آیا همه برچسب‌ها حداقل ۱۰ نمونه در مجموعه آموزشی دارند یا خیر که با انتخاب ۱۰۰۰ نمونه این موضوع حل شده است.

Check 10 Samples in Train Data

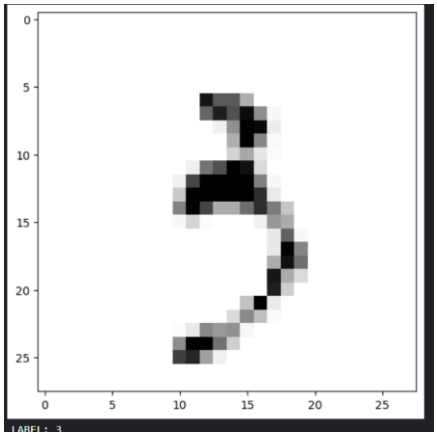
```
dataset_ok = all(count >= 10 for count in counts_train)
print('Dataset is okay:', dataset_ok)
if not dataset_ok:
    print('Not all labels have at least 10 samples in the training set.')
```

```
Dataset is okay: True
```

همچنین نمودار توزیع برچسب‌ها در شکل پایین نشان می‌دهد که تعداد نمونه‌های هر برچسب به طور کلی متوازن است، که برای آموزش مدل‌های یادگیری ماشین قابل قبول است.



در قسمت بعد، تابعی برای نمایش یک تصویر از دیتاست پیاده‌سازی می‌کنیم. این تابع یک تصویر را از مجموعه داده‌ها به همراه برچسب مربوطه به صورت زیر برای نمونه عدد 3 نمایش می‌دهد.



پس از آن برای هر کرنل ذکر شده در سوال، با استفاده از تکنیک grid search، بهترین پارامترها را پیدا کرده و دقت آن‌ها را ارزیابی می‌کنیم. این روش با امتحان کردن ترکیب‌های مختلف پارامترها و ارزیابی آن‌ها با استفاده از cross-validation بهترین پارامترها را پیدا می‌کند. فضای جستجو را به صورت زیر قرار دادیم:

gamma: [0.001, 0.01, 0.1, 1, 'scale', 'auto']

C : [0.1, 1, 10, 100],

نتایج جستجو به صورت زیر است:

کرنل خطی (Linear):

بهترین پارامترها: {'C': 0.1}

دقت: ۰.۸۶۸۷۵

کرنل چندجمله‌ای (Polynomial degree 2):

بهترین پارامترها: {'C': 0.1, 'degree': 2, 'gamma': 0.001}

دقت: ۰.۸۸۲۵

کرنل RBF:

بهترین پارامترها: {'C': 10, 'gamma': 'scale'}

دقت: ۰.۹۰۵

بهترین مدل SVM

کرنل RBF:

پارامتر C: این پارامتر میزان جریمه برای خطاهای آموزش را کنترل می‌کند. مقدار **۱۰** برای C به این مدل اجازه می‌دهد تا پیچیدگی بیشتری داشته و در نتیجه دقت بالاتری به دست آورد.

پارامتر gamma: این پارامتر تعیین می‌کند که چقدر یک نمونه آموزش بر سایر نمونه‌ها تاثیر می‌گذارد. مقدار **scale** به طور خودکار این پارامتر را بر اساس تعداد ویژگی‌ها تنظیم می‌کند.

ارزیابی بهترین مدل SVM روی داده‌های آموزش و تست

در این قسمت نتایج دقت و خطا برای هر دو مجموعه داده محاسبه شد و ماتریس آشفتگی یا Confusion Matrix و گزارش طبقه‌بندی Classification Report برای مجموعه داده تست نیز خروجی گرفته شد که در ادامه آن را مشاهده می‌کنیم.

Training accuracy: 1.0000

Training error: 0.0000

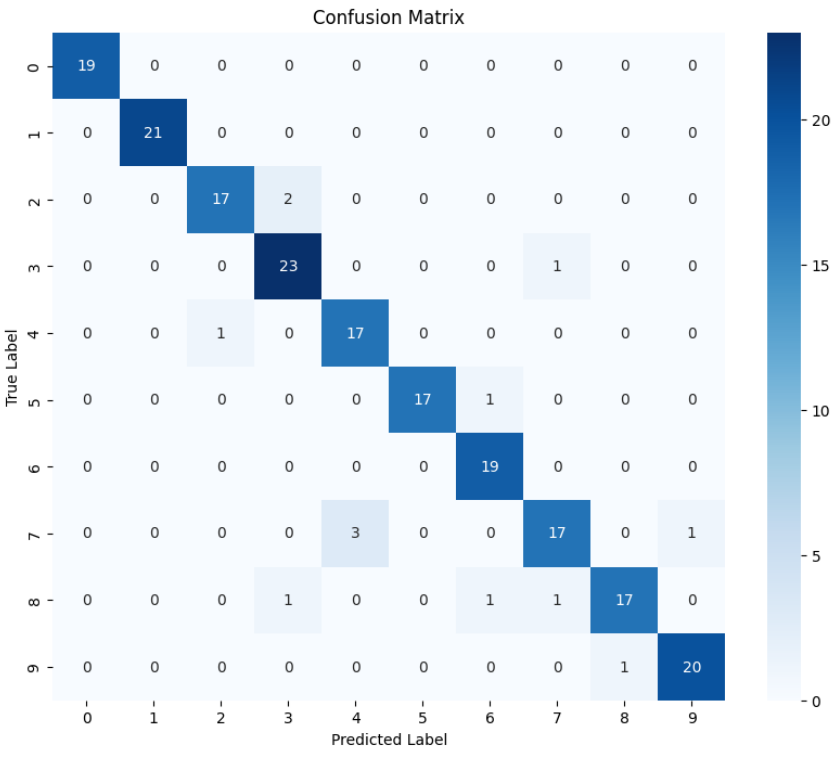
داده‌های تست :

Test accuracy: 0.9350

Test error: 0.0650

Classification report for the test set:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	19
1	1.00	1.00	1.00	21
2	0.94	0.89	0.92	19
3	0.88	0.96	0.92	24
4	0.85	0.94	0.89	18
5	1.00	0.94	0.97	18
6	0.90	1.00	0.95	19
7	0.89	0.81	0.85	21
8	0.94	0.85	0.89	20
9	0.95	0.95	0.95	21
accuracy			0.94	200
macro avg	0.94	0.94	0.94	200
weighted avg	0.94	0.94	0.93	200



دقت و خطای آموزش نشان می‌دهند که مدل بر روی داده‌های آموزشی کاملاً به درستی عمل کرده و هیچ خطایی ندارد. و دقت ۰.۹۳ روی داده‌های تست نشان می‌دهد که مدل عملکرد بسیار خوبی دارد.

مقایسه مدل SVM و لاجستیک رگرسیون

مدل لاجستیک رگرسیون را بر روی داده‌های آموزشی آموزش می‌دهیم و عملکرد آن را بر روی داده‌های تست ارزیابی می‌کنیم.

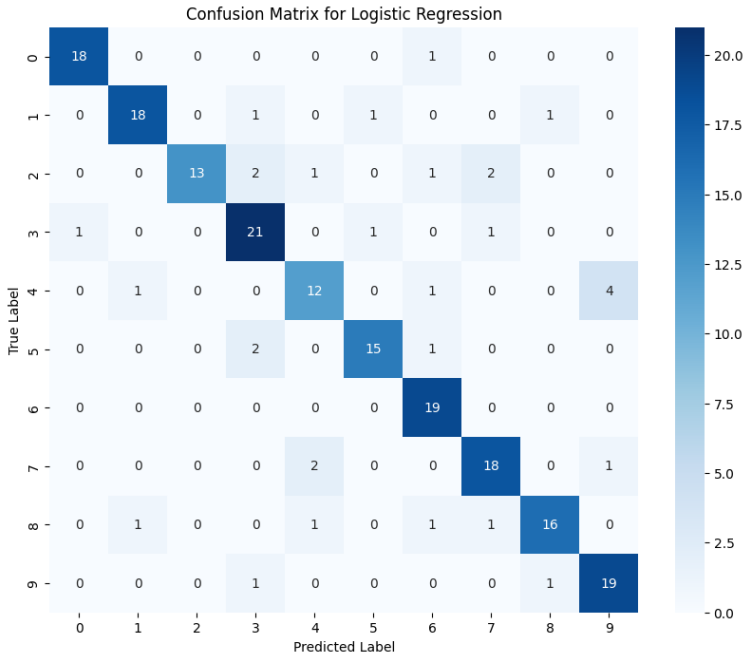
Logistic Regression Test accuracy: 0.8450

Logistic Regression Test error: 0.1550

مدل SVM با کرنل RBF عملکرد بهتری نسبت به مدل لاجستیک رگرسیون داشته است. دقت تست مدلSVM **0.9350** بوده که نسبت به دقت **0.8450** مدل لاجستیک رگرسیون بالاتر است. همچنین خطای مدل SVM صفر بوده که از **۰.۱۵۵۰**کمتر است.

Classification report for the logistic regression model:

	precision	recall	f1-score	support
0	0.95	0.95	0.95	19
1	0.90	0.86	0.88	21
2	1.00	0.68	0.81	19
3	0.78	0.88	0.82	24
4	0.75	0.67	0.71	18
5	0.88	0.83	0.86	18
6	0.79	1.00	0.88	19
7	0.82	0.86	0.84	21
8	0.89	0.80	0.84	20
9	0.79	0.90	0.84	21
accuracy			0.84	200
macro avg	0.85	0.84	0.84	200
weighted avg	0.85	0.84	0.84	200

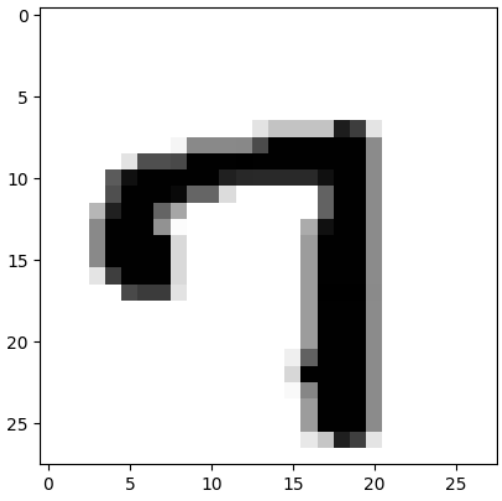


از دلایل این تفاوت می‌توان به این موضوع اشاره کرد که SVM با کرنل RBF قابلیت مدل‌سازی روابط غیرخطی را دارد. این کرنل به مدل اجازه می‌دهد تا مرزهای تصمیم‌گیری پیچیده‌تری ایجاد کند که به بهتر جدا کردن کلاس‌ها کمک می‌کند. در حالی که لاجستیک رگرسیون در حالت پایه تنها یک مدل خطی است و نمی‌تواند به خوبی روابط غیرخطی را مدل کند. این موضوع باعث می‌شود در مجموعه داده‌هایی که روابط پیچیده‌تری دارند، دقت کمتری داشته باشد.

کد زیر به ما کمک می‌کند تا نمونه‌هایی را پیدا کنیم که در آن‌ها مدل SVM بهتر از مدل لاجستیک رگرسیون عمل کرده است و این نمونه‌ها را به صورت دقیق نمایش داده و بررسی کنیم. این کار می‌تواند به شناسایی نقاط ضعف مدل لاجستیک رگرسیون کمک کند.

```
misclassified_indices = np.where((y_test_pred_log_reg != y_test) & (y_test_pred == y_test))[0]

if len(misclassified_indices) > 0:
    index = misclassified_indices[0]

    image = X_test[index].reshape(28, 28)
    true_label = y_test[index]
    log_reg_pred_label = y_test_pred_log_reg[index]
    svm_pred_label = y_test_pred[index]

    plt.imshow(image, cmap='gray_r')
    plt.xticks(np.arange(0, 28, step=5))
    plt.yticks(np.arange(0, 28, step=5))
    plt.show()

    print(f'True Label: {true_label}')
    print(f'Logistic Regression Predicted: {log_reg_pred_label}')
    print(f'SVM Predicted: {svm_pred_label}')
else:
    print("No sample found ")
```

برای نمونه در تصویر زیر که ۷ است، مدل رگرسیون لاجستیک به اشتباه ۹ تشخیص داده است.




```
True Label: 7
Logistic Regression Predicted: 9
SVM Predicted: 7
```