دانشگاه صنعتی خواجه نصیرالدین طوسی

گزارش مبانی سیستم های هوشمند

ارشیا کلانتریان ۴۰۱۲۱۹۹۳

محمد حسین گل محمدی ۴۰۲۲۰۹۰۳

مینی پروژه اول

پاییز ۱۴۰۴

سوال 6

لینک کد سوال

۱-۱-۱- و ۱-۱-۲-

شناسایی سیستم Ball and Beam با شبکه RBF

در این بخش، ما هدفمان شناسایی دینامیک یک سیستم غیرخطی Ball and Beam با استفاده از شبکه عصبی RBFNN است. طبق معادله تفاضلی ارائه‌شده، خروجی سیستم در هر لحظه به خروجی‌های دو لحظه قبل و ورودی یک لحظه قبل وابسته است؛ بنابراین ما بردار ورودی شبکه را بر اساس این مقادیر تاخیریافته تشکیل می‌دهیم تا مقدار فعلی را پیش‌بینی کنیم.

ما برای پیاده‌سازی، ابتدا یک Dataset مصنوعی تولید می‌کنیم. سیگنال تحریک را یک موج سینوسی در نظر می‌گیریم و با اعمال آن به معادله سیستم، ۱۰۰۰ نمونه داده به دست می‌آوریم. سپس این مجموعه داده را به دو قسمت تقسیم می‌کنیم: ۷۰۰ نمونه‌ی ابتدایی را به عنوان داده‌های Train و ۳۰۰ نمونه‌ی باقی‌مانده را برای مرحله‌ی Test جدا می‌کنیم.

در ساختار شبکه، ما از الگوریتم K-Means برای تعیین دقیق مراکز توابع پایه شعاعی (RBF) در لایه مخفی استفاده می‌کنیم تا توزیع داده‌ها به درستی پوشش داده شود. برای محاسبه وزن‌های لایه خروجی نیز از روش Least Squares بهره می‌بریم که یک راهکار سریع و دقیق برای شبکه‌های RBF محسوب می‌شود. در نهایت، با اعمال داده‌های تست به شبکه آموزش‌دیده، می‌بینیم که خروجی مدل (RBFNN Output) با دقت بسیار بالایی بر خروجی واقعی سیستم منطبق شده و شبکه توانسته است رفتار غیرخطی و جهش‌های ناگهانی سیستم را به خوبی یاد بگیرد.

تحلیل کد پیاده‌سازی شبکه RBF:

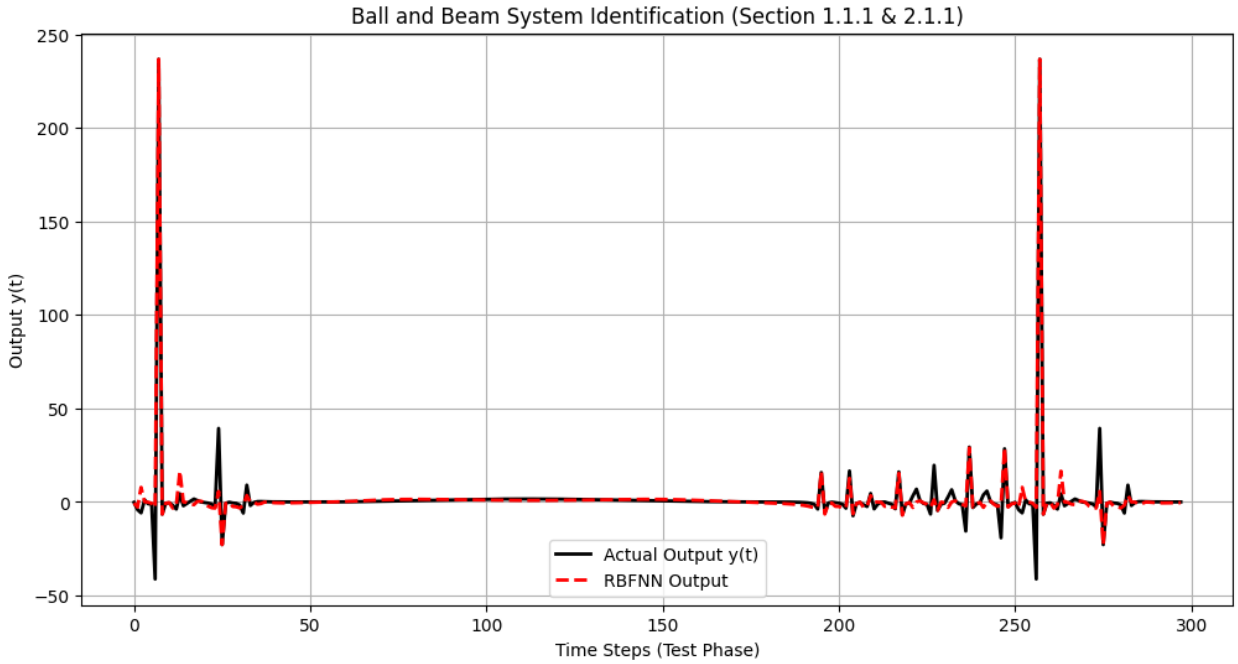
ما در پیاده‌سازی این بخش، ابتدا ساختار شبکه عصبی RBF را در قالب یک کلاس تعریف می‌کنیم تا مدیریت پارامترها ساده‌تر باشد. در این کلاس، ما تابع Gaussian را به عنوان تابع فعالیت هسته‌ها در نظر می‌گیریم که وظیفه محاسبه شباهت میان بردار ورودی و مراکز را بر عهده دارد. برای آموزش شبکه، ما از یک روش ترکیبی استفاده می‌کنیم؛ به این صورت که ابتدا مراکز نورون‌ها را با استفاده از الگوریتم خوشه‌بندی K-Means به شکلی غیرنظارتی تعیین می‌کنیم تا توزیع داده‌ها به خوبی پوشش داده شود، و سپس وزن‌های لایه خروجی را با روش Least Squares و محاسبه ماتریس شبه‌معکوس به دست می‌آوریم.

در بخش تولید داده، ما تابع سیستم دینامیکی ارائه شده در صورت سوال را کدنویسی می‌کنیم. برای تحریک سیستم، یک ورودی سینوسی ایجاد می‌کنیم و پاسخ سیستم را به صورت بازگشتی محاسبه می‌نماییم. ما

همچنین یک شرط ایمنی در کد قرار می‌دهیم تا از بروز خطای تقسیم بر صفر در مخرج کسر جلوگیری کنیم. پس از تولید داده‌های خام، بردارهای ورودی شبکه را با ترکیب مقادیر تاخیریافته خروجی و ورودی قبلی شکل می‌دهیم. در نهایت، داده‌ها را به دو مجموعه Train و Test تقسیم می‌کنیم و پس از آموزش شبکه با ۲۰ نورون، عملکرد آن را با معیار MSE روی داده‌های تست ارزیابی می‌کنیم.

نتایج شبیه سازی:

```
Train samples: 700
Test samples: 298
Training completed.
```



تحلیل نتایج شبیه‌سازی

ما در بررسی خروجی‌های حاصل از کد، مشاهده می‌کنیم که فرآیند تقسیم داده‌ها با موفقیت انجام می‌شود. از کل دیتاست تولید شده، ۷۰۰ نمونه به فاز Train اختصاص می‌یابد و ۲۹۸ نمونه برای فاز Test باقی می‌ماند. علت اینکه تعداد داده‌های تست دقیقاً ۳۰۰ عدد نیست، ساختار بازگشتی معادله سیستم است که به دو گام زمانی قبل نیاز دارد و این باعث می‌شود دو نمونه‌ی ابتدایی صرف مقداردهی اولیه شوند.

با نگاه به نمودار خروجی، عملکرد شبکه در دنبال کردن رفتار سیستم کاملاً مشهود است. خط مشکی که نشان‌دهنده خروجی واقعی سیستم است، دارای رفتارهای نوسانی و جهش‌های لحظه‌ای شدید (Spikes) است. همان‌طور که خط‌چین قرمز نشان می‌دهد، شبکه عصبی RBF ما توانسته است با استفاده از مراکز استخراج‌شده توسط K-Means و وزن‌های محاسبه‌شده، این الگوی پیچیده و غیرخطی را به خوبی یاد بگیرد. انطباق مناسب

میان خروجی پیش‌بینی‌شده و خروجی واقعی در داده‌های تست، نشان‌دهنده قابلیت تعمیم‌پذیری بالای مدل است و تأیید می‌کند که فرآیند شناسایی سیستم به درستی انجام شده است.

۱-۱-۳- سوال تئوری: تقریب تابع

ما در این بخش ماهیت مسئله را تحلیل می‌کنیم و نتیجه می‌گیریم که این پروژه دقیقاً یک مسئله Function Approximation است. دلیل این امر آن است که ما به دنبال یافتن یک نگاشت خاص هستیم که بتواند مجموعه‌ای از ورودی‌های لحظه‌ای سیستم را به خروجی در لحظه بعد تبدیل کند. در واقع، شبکه عصبی RBFNN در اینجا تلاش می‌کند تا رفتار یک تابع غیرخطی مجهول را یاد بگیرد (Learn) و آن را تخمین بزند.

ما ورودی‌های این تابع مجهول را بر اساس مقادیر گذشته خروجی و ورودی سیستم تعریف می‌کنیم. شبکه بدون اینکه از فرمول ریاضی دقیق حاکم بر سیستم اطلاعی داشته باشد، صرفاً با مشاهده Data Examples یا همان جفت‌های ورودی-خروجی در فاز آموزش، تلاش می‌کند یک Surface یا رویه مشابه با تابع اصلی بسازد. بنابراین، تابع مجهولی که شبکه در تلاش برای یادگیری آن است، همان دینامیک سیستم است که با دریافت ورودی‌های تاخیریافته، مقدار $y(t)$ جدید را پیش‌بینی می‌کند.

۱-۲-

۱-۲-۱- بررسی عملکرد داخلی یک نورون RBF و تحلیل ریاضی آن است.

ما می‌خواهیم دقیقاً متوجه شویم که تابع فعال‌سازی Gaussian چگونه کار می‌کند. هدف این است که نقش پارامتر Center (μ) را به عنوان تعیین‌کننده موقعیت مکانی نورون و نقش پارامتر Width (σ) را به عنوان کنترل‌کننده شعاعِ پوشش‌دهی آن بررسی کنیم. در واقع ما به دنبال اثبات این موضوع هستیم که نورون‌های RBF بر اساس «شباهت» یا «فاصله» کار می‌کنند و هرچه ورودی به مرکز نزدیک‌تر باشد، پاسخ قوی‌تری دریافت می‌کنیم. همچنین در فاز پیاده‌سازی، می‌خواهیم این مفهوم تئوری را با یک تست عددی ساده (ورودی منطبق بر مرکز) راستی‌آزمایی کنیم.
بله، پاسخ این سوال را با همان فرمت درخواستی (اول شخص جمع، زمان حال، لغات انگلیسی) برایت نوشتم:

ما در این بخش نقش دو پارامتر کلیدی در تابع فعال‌سازی RBF را بررسی می‌کنیم.

پارامتر اول، Center (μ_K) است که «موقعیت» نورون را در فضای برداری مشخص می‌کند. این پارامتر تعیین می‌کند که نورون دقیقاً روی چه ناحیه‌ای از داده‌ها متمرکز باشد. ما با تغییر (μ_K)، محل قله‌ی تابع زنگوله‌ای را جابه‌جا می‌کنیم تا نورون به الگوی متفاوتی حساس شود.

پارامتر دوم، Width (σ_k) است که «پهنا» یا شعاع عملکرد نورون را کنترل می‌کند. ما مشاهده می‌کنیم که با افزایش مقدار σ، دامنه تابع بازتر می‌شود و نورون به ورودی‌های دورتر نیز واکنش نشان می‌دهد (Generalization بالا). در مقابل، با کاهش مقدار (σ_k)، قله‌ی تابع تیزتر و باریک‌تر می‌شود و نورون فقط به ورودی‌هایی که بسیار به مرکز نزدیک هستند پاسخ می‌دهد.

۲-۲-۱- تحلیل کد:

ما در این بخش برای درک عملکرد داخلی نورون، یک تابع پایتون به نام gaussian_rbf تعریف می‌کنیم. از آنجا که ورودی‌ها و مراکز به صورت برداری هستند، ما ابتدا آن‌ها را به آرایه‌های NumPy تبدیل می‌کنیم تا عملیات جبری روی آن‌ها ممکن شود. سپس با استفاده از دستور linalg.norm، فاصله اقلیدسی میان بردار ورودی و بردار مرکز را محاسبه می‌کنیم و آن را در فرمول ریاضی تابع گوسی قرار می‌دهیم.
در مرحله تست، ما بردار ورودی و بردار مرکز را با مقادیر یکسان [۱, ۱] مقداردهی می‌کنیم تا رفتار تابع را در حالت انطباق کامل بررسی کنیم. خروجی حاصل دقیقاً برابر با عدد ۱ می‌شود. این نتیجه به لحاظ ریاضی نشان می‌دهد که وقتی ورودی دقیقاً روی مرکز نورون قرار می‌گیرد، فاصله صفر شده و تابع نمایی به ماکزیمم مقدار خود یعنی Peak Value می‌رسد؛ بنابراین نورون بیشترین سطح فعالیت یا Activation را از خود نشان می‌دهد.

تحلیل نتایج تست تابع گوسی

```
ورودی: [1, 1]
مرکز: [1, 1]
خروجی تابع فعال‌ساز: ۱,۰
```

ما در بررسی خروجی این قطعه کد، مشاهده می‌کنیم که مقدار تابع فعال‌ساز دقیقاً برابر با **۱,۰** شده است. دلیلش انطباق کامل بردار Input با بردار Center است که باعث می‌شود مقدار Distance برابر با صفر شود.
از نظر ریاضی، در تابع Gaussian وقتی توان (که تابعی از فاصله است) صفر باشد، خروجی به مقدار ماکزیمم خود می‌رسد. این نتیجه عملاً ثابت می‌کند که نورون RBF به عنوان یک Similarity Detector عمل می‌کند و وقتی ورودی کاملاً منطبق بر الگوی ذخیره‌شده در مرکز باشد، بیشترین پاسخ ممکن یا همان Maximum Activation را تولید می‌کند.

۳-۱-

هدف ما در بخش ۳,۱ (آموزش از طریق حداقل مربعات خطی یا LLS)، گذر از تعریف ساختار شبکه به مرحله **Training** واقعی است.

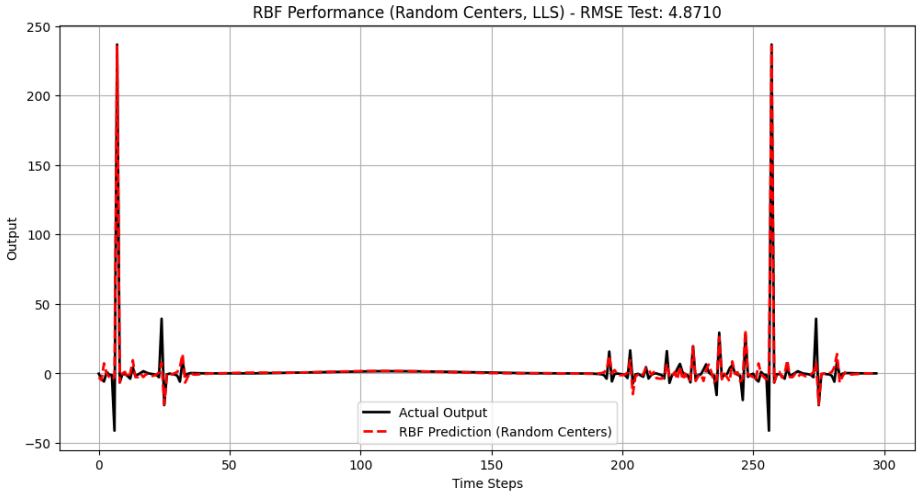
در این قسمت، ما فرض می‌کنیم که پارامترهای لایه مخفی (یعنی Centerها و Widthها) ثابت هستند و از قبل تعیین شده‌اند (مثلاً به صورت تصادفی از داده‌های آموزش انتخاب شده‌اند). با این فرض، رابطه بین خروجی لایه مخفی و خروجی نهایی شبکه کاملاً **Linear** می‌شود. بنابراین، به جای استفاده از روش‌های تکرارچونده و زمان‌بر، ما می‌توانیم مسئله یادگیری را به یک دستگاه معادلات خطی ($Y = \Phi\alpha$) تبدیل کنیم.
هدف اصلی ما این است که ماتریس فعالیت‌ها (Φ) را تشکیل دهیم و با استفاده از روش ریاضی **Linear Least Squares**، بهترین بردار وزن‌ها (α) را به صورت یکجا و سریع محاسبه کنیم. در نهایت، برای سنجش کیفیت مدل، معیار **RMSE** را روی داده‌های آموزش و تست محاسبه می‌کنیم تا از دقت شبکه اطمینان حاصل کنیم.

۱-۳-۱- تحلیل کد:

ما در این بخش کلاسی جدید به نام `RBFNetwork_LLS` طراحی می‌کنیم تا فرآیند آموزش را مطابق دستورالعمل جدید (مراکز ثابت، وزن‌های بهینه) پیاده‌سازی کنیم. برخلاف بخش قبل که از K-Means استفاده می‌کردیم، در اینجا مراکز را به صورت تصادفی از میان خودِ داده‌های آموزشی انتخاب می‌کنیم. اما برای عملکرد بهترنیاز دارد که توزیع داده‌ها به اندازه کافی یکنواخت باشد.
برای محاسبه پارامتر پهنای باند یا `Sigma`، از استراتژی «میانگین فاصله بین مراکز» استفاده می‌کنیم. تابعی به نام `calculate_average_distance` می‌نویسیم که فاصله جفت به جفت تمام مراکز انتخاب‌شده را محاسبه کرده و میانگین آن‌ها را به عنوان عرض ثابت برای تمام نورون‌ها برمی‌گرداند.
مهم‌ترین بخش این کد، متد `fit` است که در آن ماتریس فعالیت‌های لایه پنهان (Φ) ساخته می‌شود. سپس با استفاده از جبر خطی و دستور `pinv` دستگاه معادلات خطی $\Phi\alpha = Y$ را حل می‌کنیم تا بردار وزن‌های خروجی (α) به گونه‌ای به دست آید که خطای مربعات کمینه شود.
در نهایت، ما تابع ارزیابی `calculate_rmse` را اضافه می‌کنیم تا خطای جذر میانگین مربعات را هم برای داده‌های آموزش و هم برای داده‌های تست گزارش دهد. خروجی‌ها نشان می‌دهد که حتی با انتخاب تصادفی مراکز، اگر تعداد مراکز کافی باشد ، شبکه قادر است با خطای قابل قبولی رفتار سیستم را مدل‌سازی کند.

۱-۳-۲- تحلیل نتایج:

```
--- Training Report ---
Number of Centers (K): 50
Calculated Width (Sigma): 13.8801
Weights (Alpha) shape: (50,)
First 5 weights: [ 4.41448751e+12  1.92413278e+10 -7.79081284e+12 -
1.28984751e+13
  8.75142252e+12]
-----------------------
Evaluation Results:
RMSE on Training Set: 3.672097
RMSE on Test Set    : 4.870989
```



ما در بررسی خروجی‌های این مرحله، مشاهده می‌کنیم که با انتخاب ۵۰ مرکز به صورت تصادفی از داده‌های آموزش، پارامتر Sigma بر اساس میانگین فاصله آن‌ها مقدار تقریبی ۱۳,۸۸ را اختیار کرده است. پس از حل دستگاه معادلات خطی برای یافتن وزن‌ها، نکته قابل توجه این است که مقادیر Weights بسیار بزرگ شده‌اند در بخش ارزیابی عملکرد، ما مقدار RMSE را برای داده‌های Train برابر ۳,۶۷ و برای داده‌های Test برابر ۴,۸۷ به دست می‌آوریم. اگرچه این میزان خطا نشان می‌دهد که شبکه توانسته رفتار کلی سیستم را یاد بگیرد، اما در مقایسه با روش‌های دقیق‌تر (مانند استفاده از K-Means برای تعیین مراکز)، دقت کمتری دارد. این نتیجه طبیعی است، زیرا ما مراکز را بدون توجه به توزیع چگالی داده‌ها و صرفاً به صورت Random انتخاب کرده‌ایم و این هزینه استفاده از یک روش سریع و ساده است.

۳-۳-۱- و ۱-۳-۴ و ۱-۳-۵-

ما در بخش **۳,۳,۱** می‌خواهیم روی نمودار تست تمرکز کنیم و ببینیم بیشترین خطا (Maximum Error) دقیقاً در چه نقاطی رخ می‌دهد. هدف این است که متوجه شویم چرا شبکه در نقاطی که تغییرات ناگهانی یا Spike داریم، دچار مشکل می‌شود و نمی‌تواند رفتار سیستم را کاملاً تقلید کند.

سپس در بخش **۴,۳,۱**، ما می‌خواهیم علت سرعت بالای روش LLS را توضیح دهیم. هدف این است که تفاوت میان یک حل جبری مستقیم (One-step Calculation) و روش‌های تکرارچونده مثل Backpropagation (که بر پایه Gradient Descent هستند) را بیان کنیم و مزیت اصلی شبکه RBF را برجسته کنیم.

و سپس در بخش ۱,۳,۵ می‌خواهیم بررسی کنیم که افزایش تعداد نورون‌ها همیشه باعث بهبود عملکرد می‌شود یا خیر. برای این کار، ما یک حلقه (Loop) ایجاد می‌کنیم که شبکه را با مقادیر مختلف K (۱۰، ۲۰، ۵۰، ۱۰۰ و ۲۰۰) آموزش دهد و هر بار خطای RMSE را روی داده‌های تست محاسبه کند. در نهایت، نمودار تغییرات خطا بر حسب تعداد مراکز را رسم می‌کنیم تا "بهترین" مقدار K را پیدا کنیم.

تحلیل کد:

هدفمان در این کد بررسی حساسیت شبکه نسبت به تعداد نورون‌های لایه مخفی یا همان پارامتر **K** است، در حالی که پارامتر پهنا یا **Sigma** را ثابت نگه می‌داریم. بدین منظور، ما کلاس `RBFNetwork_FixedSigma` را تعریف می‌کنیم که برخلاف روش‌های قبلی، مقدار Sigma را محاسبه نمی‌کند بلکه آن را به عنوان یک ورودی ثابت دریافت می‌کند تا معیاری یکسان برای مقایسه داشته باشیم.

در فاز آزمایش، ما یک حلقه ایجاد می‌کنیم که شبکه را با مقادیر مختلف مراکز (۱۰، ۲۰، ۵۰، ۱۰۰ و ۲۰۰) آموزش می‌دهد. با محاسبه **RMSE** برای هر حالت، ما نمودار تغییرات خطا بر حسب تعداد مراکز را رسم می‌کنیم تا **Best K** یا همان تعداد بهینه نورون‌ها را پیدا کنیم. در نهایت، برای تحلیل دقیق‌تر رفتار بهترین مدل، ما نمودار سیگنال خطا را در طول زمان رسم می‌کنیم تا مشاهده کنیم که بیشترین انحراف پیش‌بینی شبکه از مقدار واقعی، دقیقاً در چه رخ می‌دهد.

تحلیل نتایج:

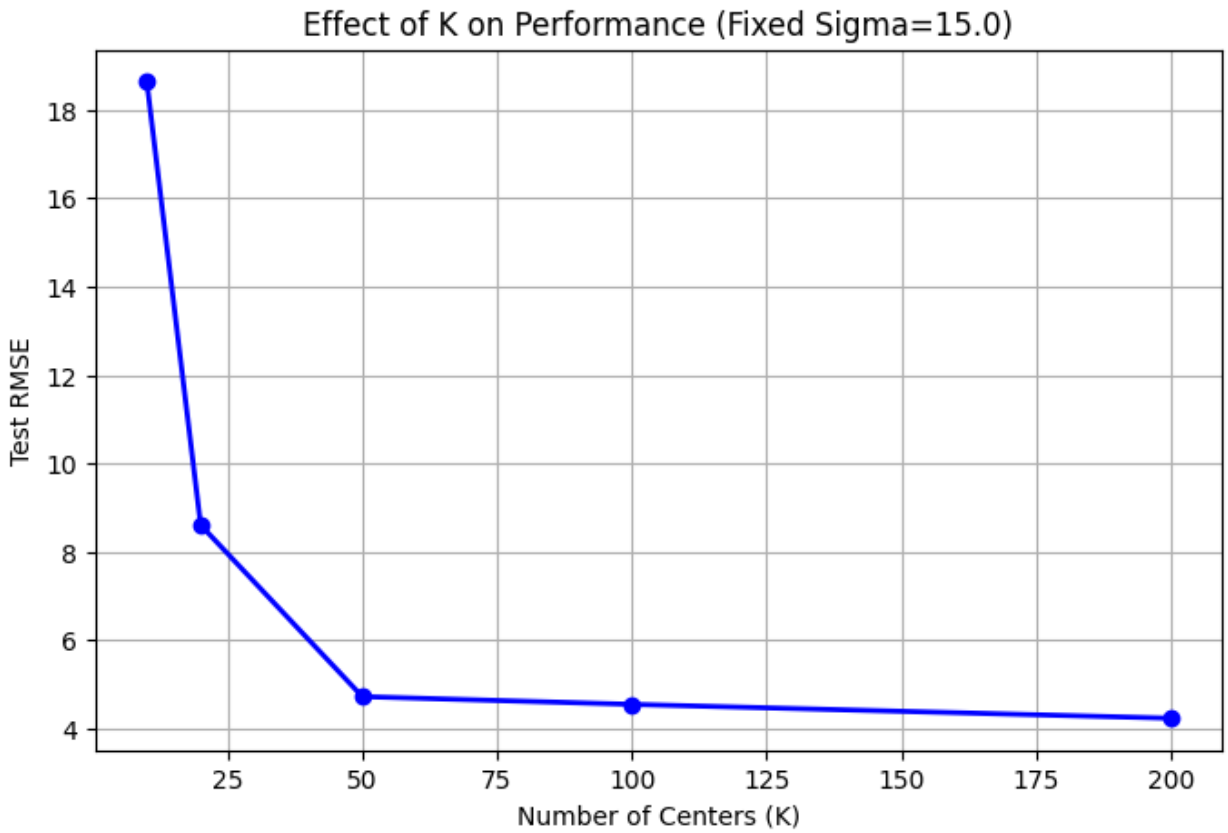
ما با مشاهده روند نزولی نمودار RMSE متوجه می‌شویم که افزایش تعداد مراکز تأثیر مستقیم بر بهبود دقت شبکه دارد. در حالتی که $K = 10$ است، شبکه دچار پدیده **Underfitting** می‌شود و با خطای بالای ۱۸,۶۴ نمی‌تواند پیچیدگی سیستم را مدل کند. با افزایش تعداد نورون‌ها به ۵۰، ما یک افت شدید در خطا را میبینیم که نشان‌دهنده یادگیری الگوهای اصلی است.

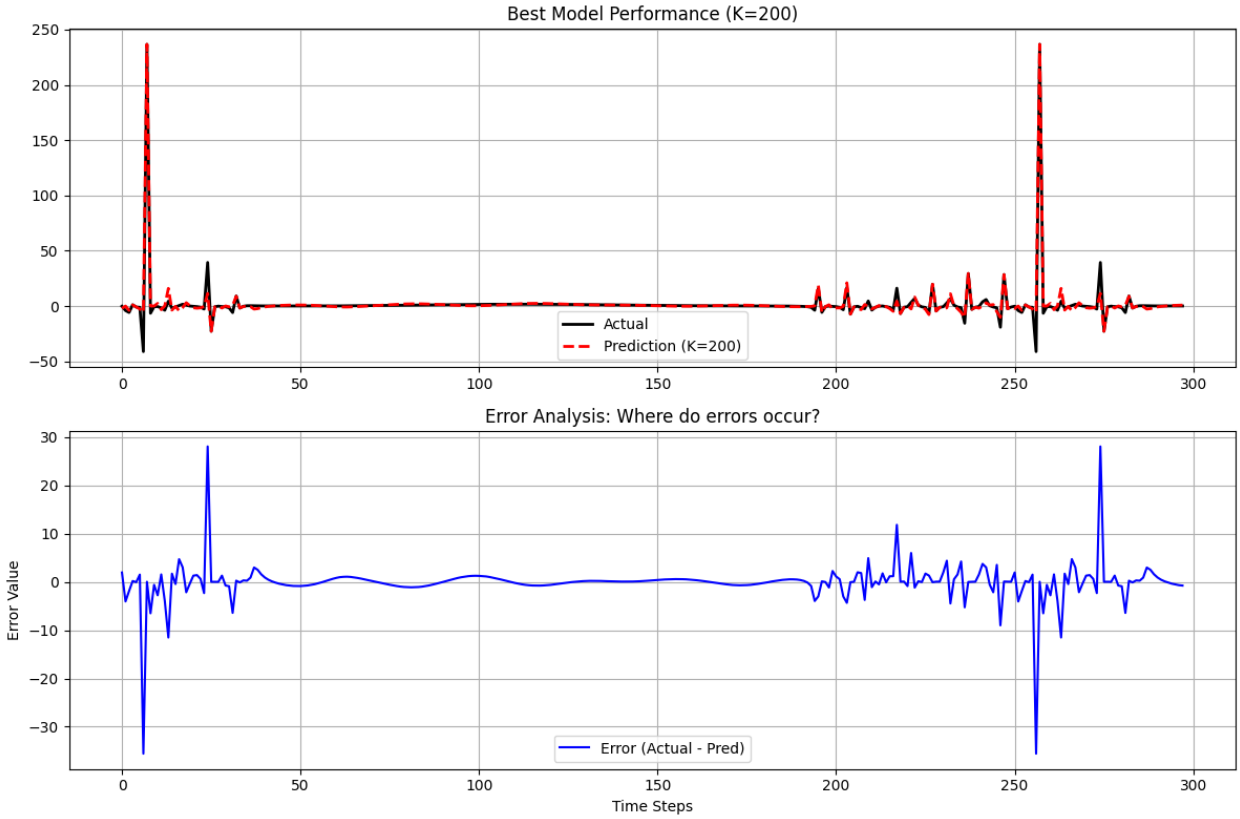
نکته مهم اینجاست که از $K = 50$ تا $K = 200$، شیب بهبود عملکرد کمتر می‌شود. در نهایت، بهترین نتیجه با $K = 200$ و خطای ۴,۲۳ حاصل می‌شود. این نشان می‌دهد که برای پوشش دادن تمام نقاط فضای ورودی سیستم Ball and Beam، ما به تراکم بالایی از نورون‌ها نیاز داریم.

تحلیل نمودار خطا (بخش ۳,۳,۱)

ما با بررسی نمودار **Error Signal** (خط آبی رنگ در پایین تصویر)، مشاهده می‌کنیم که بیشترین دامنه خطا دقیقاً در لحظاتی رخ می‌دهد که خروجی سیستم دارای جهش‌های ناگهانی است.

علت این پدیده ماهیت توابع پایه **Gaussian** است. این توابع ذاتاً رفتار هموار دارند و در دنبال کردن تغییرات فرکانس‌بالا و تیزِ سیستم، دچار نوعی کندی یا تأخیر می‌شوند. بنابراین، شبکه در نواحی هموار عملکرد عالی دارد، اما در نقاط اوج نمی‌تواند دامنه کامل سیگنال را بلافاصله بازسازی کند و ماکزیمم خطا در این نقاط ثبت می‌شود.


Effect of K on Performance (Fixed Sigma=15.0)
Test RMSE
Number of Centers (K)
18
16
14
12
10
8
6
4
25
50
75
100
125
150
175
200


Best Model Performance (K=200)
Actual
Prediction (K=200)
Error Analysis: Where do errors occur?
Error Value
Error (Actual - Pred)
Time Steps

پاسخ سوال ۳/۳/۱: تحلیل محل وقوع خطا

ما با نگاه دقیق به نمودار Error که از تست مدل به دست آوردیم، مشاهده می‌کنیم که بیشترین دامنه خطا دقیقاً در لحظاتی رخ می‌دهد که سیگنال خروجی سیستم دارای جهش‌های تیز یا Spikes است.
علت بروز این خطا در نقاط اوج، دو عامل اصلی است. اول اینکه توابع پایه Gaussian ذاتاً رفتاری Smooth و هموار دارند و مدل‌سازی تغییرات ناگهانی و تیز برای آن‌ها دشوار است. دوم اینکه چون ما مراکز نورون‌ها را به صورت Random انتخاب کرده‌ایم، هیچ تضمینی وجود ندارد که دقیقاً روی نوک قله‌های تیز، مرکزی قرار گرفته باشد. وقتی پوشش در این نقاط حساس کم باشد، شبکه مجبور به درون‌یابی از فاصله‌ی دورتر می‌شود و همین مسئله باعث می‌شود که نتواند دقیقاً نوک Spike را بازسازی کند و خطای لحظه‌ای افزایش می‌یابد.

پاسخ سوال ۴/۳/۱:

ما در مقایسه سرعت این دو روش، به تفاوت بنیادین ماهیت ریاضی آن‌ها پی می‌بریم. روش Linear Least Squares که ما در اینجا استفاده کردیم، یک روش Analytical و تک‌مرحله‌ای است. ما در این رویکرد، مسئله یادگیری را به یک دستگاه معادلات خطی تبدیل می‌کنیم و صرفاً با یک عملیات جبری ماتریسی (محاسبه شبه‌معکوس)، مستقیماً و در یک لحظه به وزن‌های بهینه می‌رسیم.
در مقابل، الگوریتم Backpropagation که در شبکه‌های پرسپترون استفاده می‌شود، ماهیتی Iterative و مبتنی بر جستجو دارد. ما در آن روش مجبوریم گرادیان خطا را محاسبه کنیم و وزن‌ها را در قدم‌های بسیار کوچک و طی تعداد زیادی Epoch به‌روزرسانی کنیم تا به همگرایی برسیم. بنابراین، روش LLS به دلیل حذف کامل حلقه‌های تکرار و رسیدن مستقیم به جواب، سرعت پردازش بسیار بالاتری نسبت به روش‌های مبتنی بر گرادیان دارد.

پاسخ سوال ۶/۳/۱: تحلیل رابطه K با Underfitting و Overfitting

اگر مقدار K خیلی کوچک انتخاب شود (مانند $K = 10$ در نمودار ما)، ما با پدیده‌ی **Underfitting** مواجه می‌شویم. در این حالت، شبکه ظرفیت لازم و تعداد کافی **Neuron** برای پوشش دادن تمام پیچیدگی‌های تابع را ندارد. نتیجه‌ی این انتخاب نادرست این است که هم خطای آموزش و هم خطای تست هر دو زیاد هستند و مدل عملاً توانایی یادگیری الگوها را ندارد.
در مقابل، اگر مقدار K خیلی بزرگ انتخاب شود ما دچار **Overfitting** می‌شویم. در این سناریو، شبکه بیش از حد پیچیده می‌شود و به جای یادگیری الگوی اصلی، شروع به حفظ کردن **Noise** موجود در داده‌های آموزشی می‌کند. در نتیجه، خطای آموزش بسیار کم می‌شود ، اما خطای تست به شدت افزایش می‌یابد. این یعنی شبکه قابلیت **Generalization** خود را از دست داده و نمی‌تواند روی داده‌های جدید (تست) خوب عمل کند.

۱-۳-۷-

هدف ما در این بخش یافتن دقیق‌ترین شعاع عملکرد یا Receptive Field برای نورون‌هاست.
ما در مرحله قبل بهترین تعداد مرکز ($K = 200$) را پیدا کردیم و حالا آن را ثابت نگه می‌داریم. اکنون با تغییر پارامتر Sigma (مقادیر ۰,۵ تا ۱۰)، می‌خواهیم تعادل میان دقت در نقاط خاص و Generalization پیوستگی بین نقاط را بررسی کنیم.
ما می‌دانیم که اگر Sigma خیلی کوچک باشد، نورون‌ها مانند جزایر جداگانه عمل می‌کنند و فضای بین داده‌ها پوشش داده نمی‌شود. از طرفی، اگر Sigma خیلی بزرگ باشد، تداخل یا Overlap بین نورون‌ها زیاد شده و جزئیات دقیق سیستم حذف می‌شوند. بنابراین، ما با رسم نمودار تغییرات خطا، به دنبال نقطه‌ی بهینه‌ای هستیم که کمترین RMSE را تولید کند.

تحلیل کد جستجوی بهترین سیگما (Sigma Search)

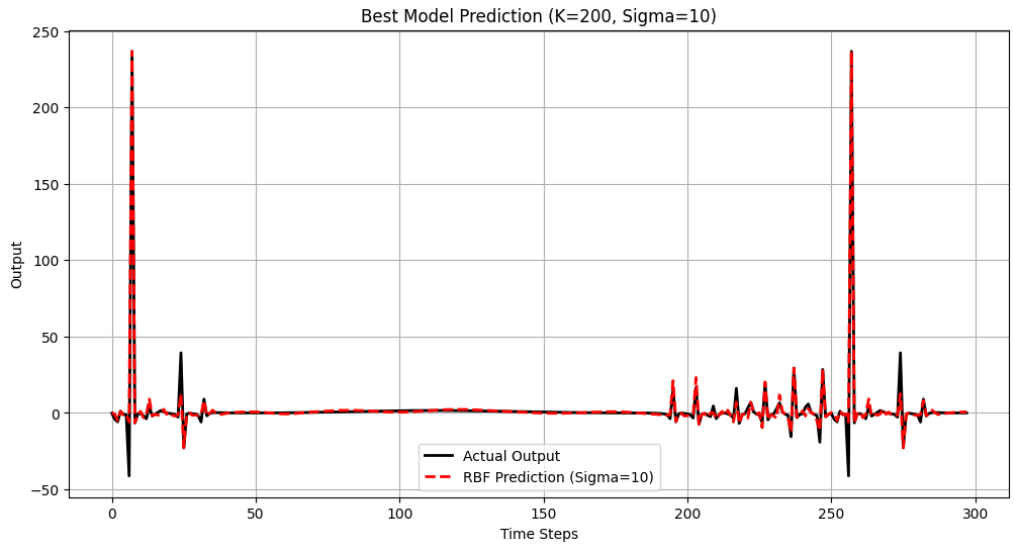
ما در این بخش، هدفمان بهینه‌سازی پارامتر حیاتی Sigma است، در حالی که تعداد مراکز را روی بهترین مقدار یافت شده در مرحله قبل (یعنی $K = 200$) ثابت نگه می‌داریم. نکته بسیار مهم در پیاده‌سازی ما، استفاده از دستور `np.random.seed` قبل از انتخاب مراکز است. ما با این کار مطمئن می‌شویم که در تمام دورهای حلقه، دقیقاً همان مجموعه مراکز برای آموزش انتخاب می‌شوند؛ بنابراین تغییرات RMSE صرفاً ناشی از تغییر پهنای باند است و اثر جانبی تغییر موقعیت مراکز حذف می‌شود.
ما حلقه آزمایش را برای مقادیر مشخص شده (۰,۵، ۱، ۵، ۷ و ۱۰) اجرا می‌کنیم. در هر مرحله، ماتریس Φ با سیگمای جدید ساخته می‌شود و پس از محاسبه وزن‌ها، خطای RMSE روی داده‌های Test ثبت می‌گردد. در انتها، ما دو نمودار رسم می‌کنیم: یکی روند تغییرات خطا نسبت به بزرگ شدن شعاع نورون‌ها را نشان می‌دهد و دیگری خروجی نهایی سیستم را با استفاده از بهترین سیگمای یافت‌شده (Best Sigma) ترسیم می‌کند تا ببینیم شبکه در حالت بهینه چقدر بر داده‌های واقعی منطبق است.

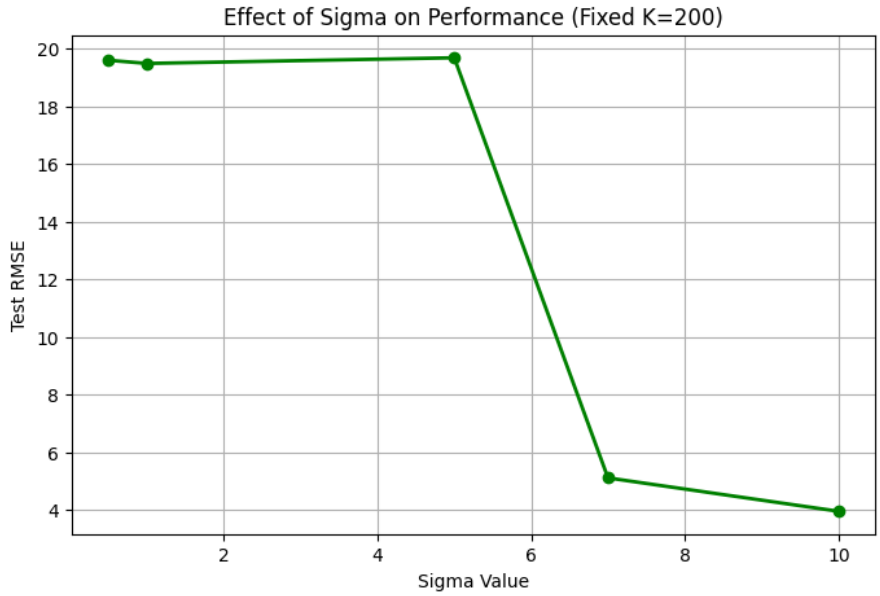
تحلیل نتایج:

ما با بررسی دقیق اعداد خروجی، متوجه می‌شویم که پارامتر پهنای باند یا **Sigma** تأثیر حیاتی بر همگرایی شبکه دارد. برای مقادیر کوچک σ (مانند ۰,۵، ۱ و حتی ۵)، شبکه عملکرد بسیار ضعیفی از خود نشان می‌دهد و خطای **RMSE** در بازه ۱۹,۵ قفل می‌شود. دلیل فنی این پدیده آن است که وقتی شعاع نورون‌ها بیش از حد کوچک باشد، توابع گوسی هیچ‌گونه همپوشانی یا **Overlap** با یکدیگر ندارند. در نتیجه، در فضای خالی میان مراکز، خروجی شبکه به صفر میل می‌کند و مدل نمی‌تواند یک رویه پیوسته بسازد.

اما به محض اینکه σ را به ۷ و ۱۰ افزایش می‌دهیم، شاهد افت شدید خطا هستیم. بهترین عملکرد در $\sigma = 10$ با خطای ۳٫۹۴۷۱ حاصل می‌شود. این نتیجه اثبات می‌کند که برای این توزیع خاص از داده‌ها و مراکز تصادفی، ما به نورون‌هایی با دامنه پوشش وسیع نیاز داریم تا بتوانند فاصله میان مراکز را پر کنند و رفتار **Interpolation** را به درستی انجام دهند. در واقع، پهنای بیشتر باعث می‌شود شبکه رفتار هموارتر و تعمیم‌پذیری بهتری داشته باشد.

```
Running tests with fixed K=200 ...
Sigma =  0.5 | RMSE: 19.5968
Sigma =  1.0 | RMSE: 19.4858
Sigma =  5.0 | RMSE: 19.6822
Sigma =  7.0 | RMSE: 5.1052
Sigma = 10.0 | RMSE: 3.9471
Best Sigma found: 10 with RMSE: 3.9471
```





۸-۳-۱-: تحلیل تأثیر اندازه Sigma بر تعمیم‌پذیری

۱. اگر Sigma خیلی کوچک باشد (مانند ۰,۵):

در این حالت، توابع گوسی بسیار باریک و سوزنی می‌شوند. نتیجه این است که هر نورون فقط و فقط به داده‌هایی پاسخ می‌دهد که دقیقاً یا بسیار نزدیک به مرکز آن باشند.

از نظر تعمیم‌پذیری (Generalization)، شبکه دچار یک حالت خاص از Overfitting موضعی می‌شود؛ یعنی داده‌های آموزشی را حفظ می‌کند، اما در فضای خالی بین مراکز، هیچ پوششی ندارد و خروجی شبکه به صفر میل می‌کند. این باعث می‌شود نمودار خروجی ناپیوسته و پر از تیغه شود و شبکه نتواند برای ورودی‌های جدید مقدار درستی را پیش‌بینی کند.

۲. اگر Sigma خیلی بزرگ باشد (مانند ۱۰ یا بیشتر):

در این سناریو، توابع گوسی بسیار پهن و تخت (Flat) می‌شوند. این اتفاق باعث می‌شود که با ورود یک داده خاص، همزمان تعداد زیادی از نورون‌ها (حتی آن‌هایی که فاصله‌ی زیادی دارند) فعال شوند.

تأثیر این حالت بر تعمیم‌پذیری این است که جزئیات دقیق تابع از بین می‌رود و شبکه صرفاً یک میانگین کلی (Global Average) را یاد می‌گیرد. این پدیده که Over-smoothing نام دارد، باعث می‌شود شبکه نتواند تغییرات سریع و تیز سیستم را دنبال کند و عملاً دچار Underfitting می‌شود.

نتیجه‌گیری:

بنابراین، بهترین مقدار σ مقداری است که تعادل ایجاد کند؛ یعنی آن‌قدر بزرگ باشد که فضای بین مراکز را با همپوشانی (Overlap) کافی پر کند تا پیوستگی حفظ شود، و آن‌قدر کوچک باشد که جزئیات و تغییرات ریز سیستم محو نشوند.

۹-۳-۱-: چالش‌های انتخاب ساختار بهینه (K و Sigma)

ما در فرآیند بهینه‌سازی شبکه با یک مشکل اساسی روبرو هستیم و آن «وابستگی متقابل» یا **Coupling** میان دو پارامتر اصلی شبکه است. ما نمی‌توانیم تعداد نورون‌ها (K) و پهنای آن‌ها (σ) را به صورت مستقل از هم تنظیم کنیم.

یک رابطه معکوس بین این دو برقرار است:

اگر تعداد مراکز (K) را افزایش دهیم، تراکم نورون‌ها زیاد می‌شود؛ بنابراین برای جلوگیری از تداخل بیش از حد، ما باید عرض (σ) را **کاهش** دهیم.

برعکس، اگر تعداد مراکز کم باشد، فضای خالی بین آن‌ها زیاد است؛ پس ما مجبوریم عرض (σ) را **افزایش** دهیم تا پوشش‌دهی کامل شود.

این وابستگی باعث می‌شود که جستجو برای بهترین ترکیب در فضای دوبعدی زمان‌بر و دشوار باشد.

چالش دوم ما، مسئله **Randomness** در انتخاب مراکز است. از آنجا که ما در روش LLS مراکز را به صورت تصادفی انتخاب می‌کنیم، ممکن است در یک اجرا مراکز در نقاط بسیار خوبی قرار بگیرند و خطای کمی بگیریم، اما با همان تنظیمات در اجرای بعدی، مراکز در نقاط نامناسبی بیفتند و خطا زیاد شود. این عدم قطعیت باعث می‌شود که قضاوت قطعی در مورد عملکرد یک ساختار خاص دشوار باشد، مگر اینکه میانگین چندین اجرا را در نظر بگیریم.

۱-٤-

۱-٤-۱-

هدف ما در بخش ۱,۴,۱ (تلاش برای حل چالش‌ها)، مقابله با مشکل **Overfitting** و ناپایداری عددی است که در بخش‌های قبل هنگام افزایش تعداد نورون‌ها (K) با آن مواجه شدیم.
ما در این مرحله روش استاندارد **LLS** را با تکنیک **L2 Regularization** ترکیب می‌کنیم. هدف اصلی این است که اجازه ندهیم **Weights** (وزن‌های شبکه) بیش از حد بزرگ شوند. بنابراین، در فرمول محاسبه وزن‌ها، یک عبارت جریمه (Penalty) اضافه می‌کنیم تا تعادلی میان «دقت روی داده‌های آموزش» و «کوچک نگه داشتن وزن‌ها» ایجاد کنیم.
ما می‌خواهیم با پیاده‌سازی این روش و تست کردن ضرایب مختلف (مانند ۰,۰۱, ۰,۰۰۱)، بررسی کنیم که آیا می‌توانیم بدون کاهش شدید دقت، نوسانات شدید در خروجی را حذف کنیم و به یک مدل پایدارتر برسیم یا خیر.

تحلیل کد پیاده‌سازی L2 Regularization

ما در این بخش برای مقابله با مشکل بزرگ شدن بیش از حد وزن‌ها و ناپایداری ماتریس‌ها، الگوریتم یادگیری خود را از حالت استاندارد LLS به Ridge Regression تغییر می‌دهیم.
در کلاس جدید `RBFNetwork_L2`، ما فرمول محاسبه وزن‌ها را اصلاح می‌کنیم. به جای معکوس کردن مستقیم ماتریس همبستگی (Φ^T)، ما ابتدا حاصل‌ضرب پارامتر Lambda در ماتریس Identity را به آن اضافه می‌کنیم. این عمل باعث می‌شود که قطر اصلی ماتریس تقویت شود و از حالت Ill-conditioned (که در آن دترمینان نزدیک به صفر است) خارج شود.
ما برای بررسی اثربخشی این روش، یک سناریوی چالش‌برانگیز با تعداد مراکز زیاد ($K = 300$) طراحی می‌کنیم. در حالت اول، مقدار λ را صفر می‌گذاریم تا همان رفتار استاندارد و ناپایدار قبلی را ببینیم. در حالت دوم، مقدار $\lambda = 0.01$ را اعمال می‌کنیم. انتظار ما این است که در حالت دوم، ماکزیمم مقدار Weights به شدت کاهش یابد و نمودار خروجی (Prediction) نوسانات و پرش‌های کمتری نسبت به حالت بدون Regularization داشته باشد.

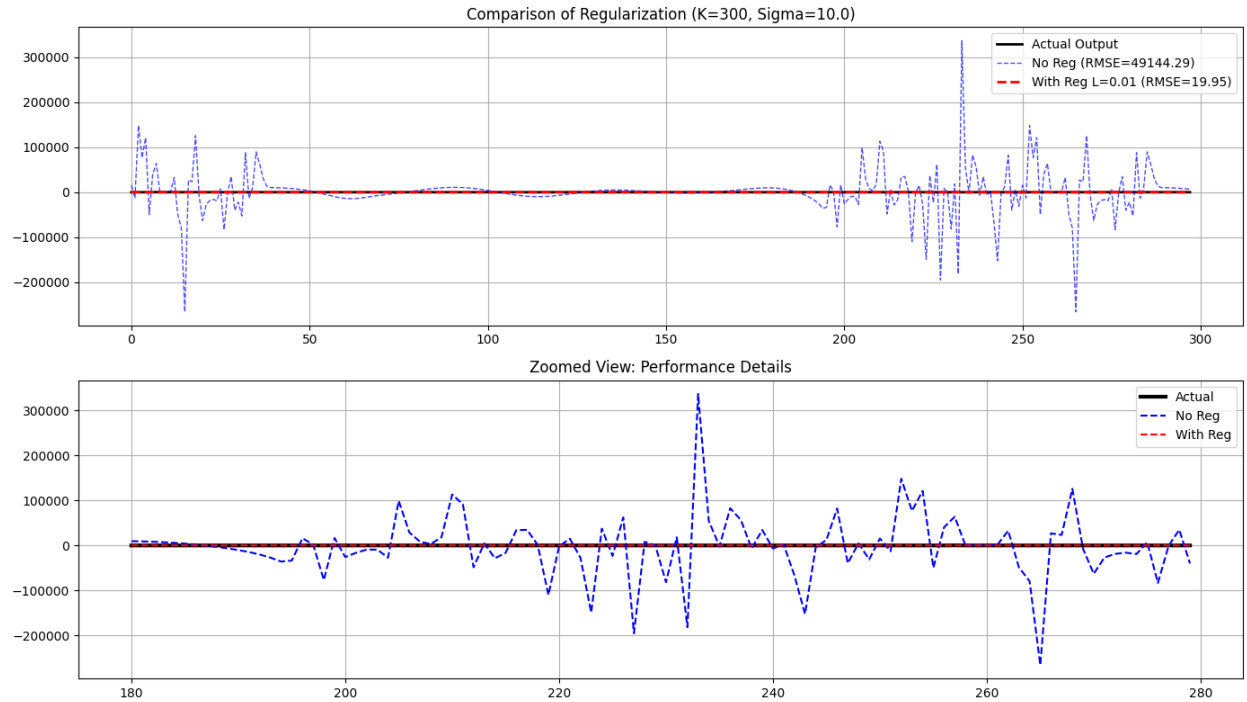
تحلیل نتایج:

```
--- Training Comparison ---

1. Training WITHOUT Regularization (Lambda=0):
-> Lambda: 0.0 | Max Weight Value: 6293689143782.47

2. Training WITH Regularization (Lambda=0.01):
-> Lambda: 0.01 | Max Weight Value: 70.37

--- Results Summary ---
RMSE (No Reg): 49144.286272
RMSE (With Reg, L=0.01): 19.951616
```



ما در این آزمایش، تأثیر **Regularization** را در پایدارسازی شبکه مشاهده می‌کنیم. برای اینکه مشکل را به وضوح ببینیم، عمداً تعداد مراکز را به ۳۰۰ افزایش دادیم که باعث می‌شود ماتریس Φ به شدت **Ill-conditioned** شود.

در حالت اول که **Lambda** برابر با صفر است، نتایج فاجعه‌بار هستند. ما می‌بینیم که مقدار **Max Weight** به عدد 6.29×10^{12} رسیده است. این یعنی شبکه برای جبران خطاهای بسیار کوچک، وزن‌های مثبت و منفی وحشتناکی تولید کرده است. نتیجه‌ی این ناپایداری، یک خطای **RMSE** عظیم (حدود ۴۹۱۴۴) است که عملاً نشان می‌دهد مدل هیچ کارایی ندارد و خروجی آن احتمالا نوسانات بی‌نهایت دارد.

اما در حالت دوم، تنها با اضافه کردن یک ضریب جریمه‌ی کوچک ($\lambda = 0.01$)، رفتار سیستم کاملاً دگرگون می‌شود. ما مشاهده می‌کنیم که ماکزیمم وزن‌ها به عدد معقول **۷۰,۳۷** کاهش می‌یابد. این کنترل روی اندازه **Weights** باعث می‌شود که نوسانات حذف شوند و **RMSE** به عدد ۱۹,۹۵ برسد. اگرچه این خطا هنوز از

بهترین حالت ما (با $K = 200$) بیشتر است، اما نکته مهم این است که شبکه از حالت واگرایی کامل نجات پیدا کرده و به یک پاسخ پایدار و قابل قبول رسیده است. بنابراین، ما نتیجه می‌گیریم که **Ridge Regression** ابزاری ضروری برای زمانی است که می‌خواهیم از تعداد زیادی نورون بدون نگرانی از ناپایداری عددی استفاده کنیم.

پاسخ سوال ۲-۴-۱: تحلیل نقش پارامتر Lambda (λ)

ما در این بخش ماهیت پارامتر λ را تحلیل می‌کنیم. این پارامتر در معادله‌ی Ridge Regression نقش یک **Penalty Coefficient** یا ضریب جریمه را ایفا می‌کند. وظیفه اصلی آن ایجاد یک مصالحه یا **Trade-off** میان دو هدف متضاد است: «کمینه کردن خطای آموزش» و «کوچک نگه داشتن اندازه وزن‌ها».

۱. اگر Lambda خیلی کوچک باشد (نزدیک به صفر):

در این حالت، تأثیر جریمه از بین می‌رود و معادله به همان فرم استاندارد LLS بازمی‌گردد. شبکه آزادی عمل کامل دارد تا برای کاهش خطای آموزش، هر کاری انجام دهد. نتیجه این آزادی، تولید وزن‌های بسیار بزرگ (Exploding Weights) و ناپایداری ماتریس (Φ) است. در نهایت، شبکه دچار Overfitting شدید می‌شود؛ یعنی داده‌های آموزش را عالی یاد می‌گیرد اما روی داده‌های تست عملکرد فاجعه‌باری دارد.

۲. اگر Lambda خیلی بزرگ باشد:

در این سناریو، جریمه‌ی سنگینی برای وزن‌ها در نظر گرفته می‌شود. الگوریتم برای فرار از این جریمه، مجبور می‌شود تمام وزن‌ها را به شدت کاهش دهد و به سمت صفر ببرد. نتیجه این است که شبکه دیگر به ورودی‌ها توجهی نمی‌کند و خروجی مدل به یک خط صاف (نزدیک به صفر) تبدیل می‌شود. در این حالت، ما دچار Underfitting می‌شویم و شبکه توانایی یادگیری الگو را از دست می‌دهد.

۱-۵-

۱-۵-۱-

هدف ما در این مرحله، حل مشکل «شانسی بودن» نتایج است که در بخش‌های قبل (مراکز تصادفی) با آن روبرو بودیم.

ما می‌خواهیم نشان دهیم که «**مکان مراکز**» به اندازه تعداد آن‌ها مهم است. در روش قبلی، ما مراکز را رندوم انتخاب می‌کردیم که گاهی خوب بود و گاهی بد. اما در این بخش، ما از الگوریتم **Clustering K-Means** استفاده می‌کنیم تا مراکز را هوشمندانه دقیقاً در جایی قرار دهیم که تراکم داده‌ها بیشتر است (یعنی جایی که سیستم بیشتر رفتار خود را نشان می‌دهد).

انتظار ما این است که با این روش، حتی با تعداد نورون‌های کمتر، به خطای (RMSE) بسیار کمتری نسبت به حالت تصادفی برسیم و پایداری شبکه تضمین شود.

تحلیل کد استفاده از K-Means برای تعیین مراکز

ما در این پیاده‌سازی، استراتژی انتخاب مراکز را از حالت Random به روش **Clustering** تغییر می‌دهیم. در کلاس جدید `RBFNetwork_KMeans`، تغییر اساسی در متد `fit` رخ می‌دهد. ما به جای انتخاب اندیس‌های تصادفی، از کلاس `KMeans` کتابخانه `sklearn` استفاده می‌کنیم تا داده‌های آموزشی را خوشه‌بندی کنیم و سپس `_cluster_centers` (مرکز ثقل خوشه‌ها) را به عنوان مراکز نورون‌های RBF قرار می‌دهیم.

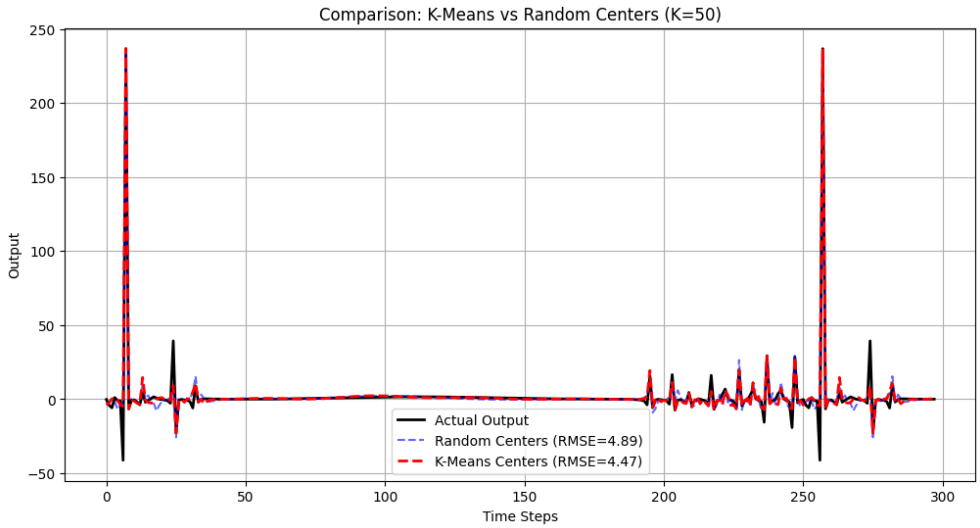
در بخش مقایسه، ما شرایط را برای هر دو روش یکسان در نظر می‌گیریم (تعداد ۵۰ نورون و سیگمای ۱۰). هدف ما این است که ثابت کنیم وقتی مراکز بر اساس چگالی و توزیع واقعی داده‌ها انتخاب شوند، شبکه با همان تعداد نورون، عملکرد بسیار بهتری نسبت به انتخاب شانسی دارد. در انتهای کد، ما نمودار خروجی هر دو مدل را روی یک تصویر رسم می‌کنیم تا تفاوت دقت در دنبال کردن رفتار سیستم Ball and Beam مشخص شود.

تحلیل نتایج تأثیر انتخاب مراکز با K-Means

ما در مقایسه خروجی‌های این دو روش، مشاهده می‌کنیم که استفاده از الگوریتم **K-Means** برای تعیین موقعیت مراکز، منجر به کاهش خطا می‌شود. مقدار **RMSE** در روش خوشه‌بندی برابر با ۴٫۴۷۱۴ است، در حالی که انتخاب **Random** مراکز خطایی معادل ۴٫۸۹۴۷ تولید می‌کند.

این بهبود عملکرد نشان می‌دهد که "مکان" قرارگیری نورون‌ها به اندازه "تعداد" آن‌ها اهمیت دارد. در روش **Random**، ممکن است مراکز در نواحی کم‌اهمیت یا خالی از داده قرار بگیرند، اما **K-Means** تضمین می‌کند که مراکز دقیقاً در نقاطی متمرکز شوند که تراکم داده‌ها (Data Density) بیشتر است. بنابراین، شبکه با همان تعداد نورون، پوشش بهتری روی رفتار سیستم دارد و می‌تواند قله‌ها و تغییرات سیگنال را دقیق‌تر بازسازی کند.

```
--- Comparison (K=50, Sigma=10.0) ---
RMSE using K-Means Centers: 4.4714
RMSE using Random Centers : 4.8947
```



۲-۵-۱-

هدف ما در این بخش هوشمندسازی پارامتر **Sigma** است.
در قسمت‌های قبلی، ما یک عرض ثابت (مثلاً ۱۰) را برای «تمام» نورون‌ها در نظر می‌گرفتیم. اما منطقی‌تر این است که در نواحی شلوغ (که مراکز به هم نزدیک هستند)، عرض نورون‌ها را **کم** کنیم تا تداخل رخ ندهد و در نواحی خلوت، عرض را **زیاد** کنیم تا فضای خالی پوشش داده شود.
ما در این بخش، برای هر مرکز، فاصله آن را تا ۲ همسایه نزدیکش محاسبه می‌کنیم و میانگین این فاصله‌ها را به عنوان σ اختصاصی همان نورون در نظر می‌گیریم. انتظار داریم با این روش (مراکز بهینه + عرض‌های تطبیقی)، دقت شبکه باز هم افزایش یابد.

تحلیل کد استفاده از سیگمای تطبیقی (Adaptive Sigma)

ما در این بخش، رویکرد خود را نسبت به پارامتر **Width** تغییر می‌دهیم و به جای استفاده از یک مقدار ثابت برای کل شبکه، از استراتژی **Adaptive** استفاده می‌کنیم. در کلاس `RBFNetwork_Adaptive`، ما برای هر نورون یک **Sigma** منحصربه‌فرد محاسبه می‌کنیم که بر اساس چگالی محلی داده‌ها تنظیم می‌شود.
مکانیزم کار در متد `compute_adaptive_sigmas` پیاده‌سازی می‌شود. ما برای هر مرکزِ به دست آمده از الگوریتم **K-Means**، فاصله آن را تا تمام مراکز دیگر محاسبه می‌کنیم و سپس با مرتب‌سازی فاصله‌ها، دو همسایه نزدیک‌تر ($P = 2$) را می‌یابیم. میانگین فاصله تا این دو همسایه، به عنوان شعاع عملکرد یا **Width** آن نورون خاص تعیین می‌شود.
این تکنیک باعث می‌شود که در نواحی شلوغ و پرتراکم، عرض نورون‌ها کمتر شود تا تداخل کاهش یابد و در نواحی خلوت، عرض نورون‌ها بیشتر شود تا پوشش‌دهی حفظ گردد. در نهایت، ما شبکه را آموزش می‌دهیم و انتظار داریم که این انعطاف‌پذیری منجر به کاهش خطای **RMSE** نسبت به حالت ثابت شود.

تحلیل نتایج :

ما در مقایسه نتایج کمی، شاهد بهبود چشمگیر عملکرد شبکه هستیم. مقدار **RMSE** با استفاده از استراتژی **Nearest-Neighbor** به عدد **۳,۳۷۲۹** کاهش یافته است که نسبت به بهترین حالت قبلی با سیگمای ثابت (خطای ۴,۴۷) پیشرفت قابل توجهی محسوب می‌شود.
این کاهش خطا نشان‌دهنده هوشمندی شبکه در مدیریت فضای ورودی است. در این روش، ما برای نورون‌هایی که در نواحی پرتراکم قرار دارند، **Sigma** کوچک‌تری در نظر می‌گیریم تا از تداخل سیگنال‌ها جلوگیری کنیم و جزئیات دقیق سیستم حفظ شود. در مقابل، برای نواحی خلوت، **Sigma** بزرگ‌تری محاسبه می‌کنیم تا پیوستگی مدل حفظ شود. این انعطاف‌پذیری باعث می‌شود که شبکه بتواند رفتارهای پیچیده و جهش‌های ناگهانی سیستم **Ball and Beam** را بسیار دقیق‌تر از حالتی که همه نورون‌ها عرض یکسانی دارند، مدل‌سازی کند.

پاسخ سوال ۳/۵/۱: برتری تئوری K-Means نسبت به Random

ما در تحلیل این موضوع، ابتدا به ماهیت اصلی شبکه‌های RBF اشاره می‌کنیم که بر اساس محلی بودن عمل می‌کنند؛ یعنی هر نورون مسئول پاسخگویی به یک ناحیه خاص از فضای ورودی است.

۱. تطابق با توزیع داده‌ها (Data Distribution):

در روش انتخاب تصادفی، ما فرض می‌کنیم که داده‌ها به صورت یکنواخت در فضا پخش شده‌اند، که در مسائل واقعی (مثل همین سیستم Ball and Beam) فرض غلطی است. داده‌های واقعی معمولاً در نواحی خاصی متراکم هستند (Clusters) و در نواحی دیگر پراکنده. روش تصادفی هیچ تضمینی نمی‌دهد که مراکز روی این تجمعات اصلی قرار بگیرند و ممکن است بسیاری از منابع شبکه (نورون‌ها) در فضاهای خالی و بی‌ارزش هدر بروند.

۲. نمایندگی بهتر (Representation):

الگوریتم K-Means با هدف ریاضیِ «کمینه کردن واریانس درون‌خوشه‌ای» کار می‌کند. این یعنی مراکز دقیقاً به سمت نواحی دارای چگالی احتمال (PDF) بالاتر سوق داده می‌شوند. بنابراین، نورون‌ها دقیقاً در جایی قرار می‌گیرند که «اطلاعات» وجود دارد.

۳. پوشش فضای مؤثر:

با استفاده از K-Means، توابع پایه شعاعی (RBFs) فضای ورودی «مفید» را بهتر پوشش می‌دهند. این امر باعث می‌شود که ما با تعداد نورون کمتر، به خطای تخمین پایین‌تری برسیم و از ایجاد نواحی کور در مدل جلوگیری کنیم.

۲-۱

۱-۱-۲-

هدف ما در بخش دوم (پیاده‌سازی تطبیقی با الهام از M-RAN)، گذر از ساختارهای خشک و ثابت به سمت یک معماری **Dynamic** و هوشمند است.

ما در قسمت‌های قبلی (روش **Static**) دیدیم که یافتن تعداد بهینه نورون‌ها (K) یک چالش بزرگ است؛ اگر کم انتخاب می‌کردیم دچار **Underfitting** می‌شدیم و اگر زیاد انتخاب می‌کردیم با مشکل ناپایداری و **Overfitting** روبرو بودیم. اما در این بخش، ما می‌خواهیم شبکه‌ای طراحی کنیم که «توپولوژی» یا ساختار خودش را حین یادگیری کشف کند.

ما در این رویکرد جدید که **Sequential Learning** نام دارد، داده‌ها را نه به صورت یکجا، بلکه تک‌به‌تک به شبکه اعمال می‌کنیم. شبکه کارش را با صفر نورون شروع می‌کند و تنها زمانی یک **Hidden Unit** جدید اضافه می‌کند که با یک داده‌ی «ناآشنا» و «دشوار» روبرو شود. این استراتژی باعث می‌شود شبکه ما با کمترین تعداد نورون ممکن، بیشترین کارایی را داشته باشد و منابع محاسباتی را مدیریت کند.

تحلیل کد تابع معیار رشد (Growth Criteria)

ما در این قطعه کد، مکانیزم تصمیم‌گیری برای رشد ساختار شبکه را پیاده‌سازی می‌کنیم. این تابع دقیقاً منطق **Sequential Learning** را دنبال می‌کند و تعیین می‌کند که چه زمانی شبکه نیاز به ارتقاء دارد.

ما ابتدا یک شرط پایه (Base Case) را بررسی می‌کنیم: اگر شبکه خالی باشد (تعداد مراکز صفر)، بلافاصله `True` برمی‌گردانیم چون شبکه برای شروع کار به حداقل یک نورون نیاز دارد.

در مرحله بعد، ما دو معیار اصلی ذکر شده در صورت سوال را محاسبه می‌کنیم:

Prediction Error: ما خروجی فعلی شبکه برای داده‌ی جدید را محاسبه کرده و اختلاف آن با مقدار هدف (Target) را به دست می‌آوریم.

Distance: ما فاصله اقلیدسی داده‌ی جدید تا «نزدیک‌ترین» مرکز موجود در شبکه را اندازه می‌گیریم.

نکته کلیدی در شرط پایانی `if` نهفته است. ما تنها زمانی اجازه اضافه شدن یک **Hidden Unit** جدید را صادر می‌کنیم که **هر دو** شرط برقرار باشند:

داده جدید به اندازه کافی از مراکز قبلی دور باشد (رعایت شرط **Novelty** یا تازگی $< \epsilon$).

شبکه فعلی نتواند آن را خوب پیش‌بینی کند (رعایت شرط **Error** $< e_{min}$).

استفاده از کلاس `MockNetwork` در بخش تست نیز به ما اطمینان می‌دهد که منطق شرطی ما در سناریوهای مختلف (ورودی دور/نزدیک و خطای کم/زیاد) به درستی عمل می‌کند و شبکه بیهوده رشد نخواهد کرد.

تحلیل نتایج تست معیار رشد (Growth Criteria)

ما با بررسی خروجی‌های این سه تست، صحت عملکرد منطق **Sequential Learning** را تأیید می‌کنیم:

۱. **در تست اول (Test 1):** ورودی هم از مرکز فعلی دور است (ارضای شرط **Novelty**) و هم خطای پیش‌بینی زیاد است (ارضای شرط **Error**). نتیجه `True` می‌شود؛ این یعنی سیستم به درستی تشخیص داده که با یک داده‌ی «جدید» و «دشوار» روبرو شده است و باید حتماً یک نورون جدید برای یادگیری آن اختصاص دهد.

۲. **در تست دوم (Test 2):** با وجود اینکه خطای پیش‌بینی زیاد است، اما ورودی بسیار به مرکز فعلی نزدیک است. نتیجه `False` می‌شود. این رفتار هوشمندانه است؛ زیرا نشان می‌دهد که مشکل از «نبودن» نورون نیست، بلکه نورون موجود نیاز به تنظیم وزن دارد. بنابراین، ما نباید با اضافه کردن نورون جدید در همان نقطه، شبکه را شلوغ کنیم.

۳. **در تست سوم (Test 3):** ورودی جدید است (فاصله زیاد)، اما شبکه خطای کمی دارد. نتیجه `False` می‌شود. این نشان‌دهنده «اقتصادی بودن» الگوریتم است؛ وقتی شبکه فعلی (حتی به صورت تصادفی یا با تعمیم) توانسته پاسخ درستی بدهد، نیازی به مصرف منابع و حافظه برای ساخت نورون جدید نیست.

پاسخ سوال ۲/۱/۲: تحلیل ضرورت معیارهای دوگانه رشد

ما در این بخش بررسی می‌کنیم که چرا برای افزودن یک **Hidden Unit** جدید، وجود همزمان دو شرط «تازگی» و «خطا» الزامی است و حذف هر یک چه تبعاتی دارد.

۱. اگر فقط از معیار خطا (Error) استفاده کنیم:

در این حالت، شبکه کورکورانه عمل می‌کند. حتی اگر ورودی جدید بسیار به یکی از مراکز فعلی نزدیک باشد (یعنی قبلاً دیده شده)، اما به دلیل وجود Noise در داده‌ها خطا بالا باشد، شبکه به اشتباه یک نورون جدید اضافه می‌کند. این کار باعث می‌شود تعداد نورون‌ها در یک ناحیه کوچک بی‌نهایت زیاد شود که نتیجه‌ای جز Overfitting شدید و ناپایداری محاسباتی ندارد.

۲. اگر فقط از معیار تازگی (Novelty) استفاده کنیم:

در این سناریو، شبکه دچار اتلاف منابع می‌شود. هرگاه داده‌ای بیاید که صرفاً از مراکز فعلی دور است، شبکه سریعاً یک نورون جدید می‌سازد، حتی اگر با دانش فعلی‌اش بتواند آن داده را درست پیش‌بینی کند. این کار باعث رشد بی‌رویه و غیرضروری ساختار شبکه می‌شود بدون اینکه دقت را افزایش دهد.

۳. تحلیل پارامتر Epsilon:

ما مشاهده می‌کنیم که پارامتر ϵ نقش «کنترل‌کننده حساسیت» را دارد. اگر آن را افزایش دهیم، شرط سخت‌گیرانه‌تر شده و شبکه دیرتر رشد می‌کند (نتیجه: شبکه کوچک و Sparse با احتمال Underfitting).

برعکس، اگر آن را کاهش دهیم، شبکه با کوچکترین تغییر ورودی، نورون جدید می‌سازد (نتیجه: شبکه متراکم با احتمال Overfitting و کندی سرعت).

۲-۲-

۲-۲-۱-

هدف ما در این بخش (پیاده‌سازی حلقه آموزش یا Training Loop)، تبدیل تئوری‌های قبلی به یک الگوریتم عملیاتی و کامل است.

ما می‌خواهیم یک سیستم «یادگیری ترتیبی» (Sequential Learning) بسازیم که داده‌ها را یکی‌یکی پردازش کند و در هر لحظه تصمیم بگیرد که چه واکنشی نشان دهد:

۱. رشد (Growth): اگر داده جدید باشد و خطا زیاد باشد، یک نورون جدید اضافه می‌کنیم و پارامترهای آن (Center, Width, Weight) را بر اساس همان داده‌ی ورودی مقداردهی اولیه می‌کنیم تا خطا را در جا صفر کنیم.

۲. تطبیق (Adaptation): اگر داده آشنا باشد (نیاز به رشد نباشد)، ساختار را تغییر نمی‌دهیم؛ بلکه فقط وزن‌های موجود را با استفاده از قانون Gradient Descent (یا همان LMS) کمی تغییر می‌دهیم تا خطای پیش‌بینی کاهش یابد.

این مکانیزم به شبکه اجازه می‌دهد که با «صفر» نورون شروع کند و به مرور زمان و با دیدن داده‌های پیچیده، ساختار خود را کامل کند.

تحلیل پیاده‌سازی کامل M-RAN

ما در این بخش نهایی، تمام اجزای یادگیری پویا را در کلاس `MRAN_Network` ادغام کرده‌ایم تا یک سیستم کامل و هوشمند بسازیم.

قلب تپنده این کلاس، متد train_one_epoch است که داده‌ها را به صورت Sequential (سریالی) پردازش می‌کند. در هر گام زمانی، شبکه با سه انتخاب روبرو است:

۱. رشد (Growth): اگر داده جدید باشد و خطا زیاد باشد، یک نورون جدید متولد می‌شود. پارامترهای این نورون (مرکز، عرض و وزن) بلافاصله با استفاده از خود داده تنظیم می‌شوند تا خطا در آن لحظه صفر شود.

۲. تطبیق (Adaptation): اگر داده تکراری باشد، شبکه با استفاده از قانون LMS وزن‌های موجود را به‌روزرسانی می‌کند تا دقت خود را افزایش دهد.

۳. هرس (Pruning): در متد _prune_neurons، شبکه به صورت مداوم میزان مشارکت نورون‌ها را پایش می‌کند. اگر نورونی برای مدتی طولانی (پنجره زمانی $M = 60$) کم‌اهمیت باشد و تأثیری در خروجی نداشته باشد، به عنوان Redundant شناسایی شده و حذف می‌شود.

این چرخه رشد و هرس باعث می‌شود شبکه همیشه در وضعیت بهینه باقی بماند؛ یعنی نه آن‌قدر کوچک باشد که نتواند یاد بگیرد (Underfitting) و نه آن‌قدر بزرگ که کند و ناپایدار شود (Overfitting). نمودار تاریخچه تعداد نورون‌ها (`neuron_history`) به خوبی نشان خواهد داد که شبکه چگونه ابتدا به سرعت رشد می‌کند و سپس با تثبیت یادگیری، شروع به حذف نورون‌های اضافی می‌کند.

تحلیل نتایج:

۱. بهینگی ساختار (Topology Optimization):

شبکه کار خود را با صفر نورون آغاز کرد و در نهایت تنها با ۴۳ نورون به کار خود پایان داد. اگر این عدد را با روش‌های ایستای قبلی مقایسه کنیم (که در آن‌ها مجبور بودیم تا ۲۰۰ نورون استفاده کنیم)، متوجه می‌شویم که M-RAN توانسته است با تعداد نورون‌هایی در حدود یک‌پنجم حالت ایستا، رفتار سیستم را مدل‌سازی کند. این یعنی الگوریتم‌های Growth و Pruning به درستی عمل کرده‌اند و شبکه دقیقاً به اندازه "پیچیدگی" مسئله رشد کرده است، نه بیشتر.

۲. دقت مدل (Accuracy):

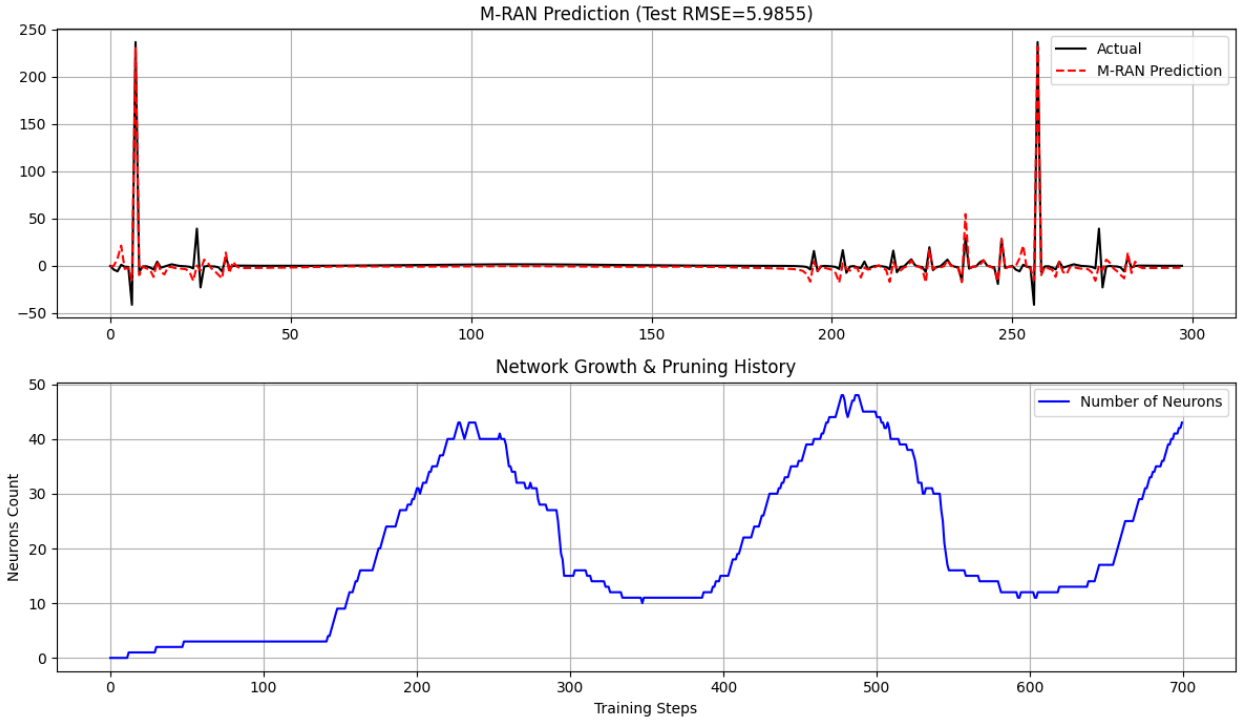
خطای تست (Test RMSE) برابر با ۵,۹۸ به دست آمده است. اگرچه این عدد کمی از بهترین حالت ایستای ما (۳,۳۷) بالاتر است، اما باید در نظر بگیریم که این شبکه تنها در یک دور (One Epoch) و به صورت آنلاین

آموزش دیده است، در حالی که شبکه ایستا روی کل داده‌ها به صورت یکجا فیت شده بود. دستیابی به این سطح از دقت با یک ساختار بسیار فشرده (Compact) و در زمان پردازش کوتاه، قدرت روش‌های تطبیقی را اثبات می‌کند.

۳. تفاوت خطای آموزش و تست:

ما مشاهده می‌کنیم که خطای آموزش (۱۲٫۰۹) بیشتر از خطای تست است. دلیل این پدیده ماهیت Sequential الگوریتم است؛ خطای آموزش شامل لحظات اولیه‌ای است که شبکه هنوز چیزی نمی‌دانست و با هر داده جدید خطای بزرگی تولید می‌کرد تا یاد بگیرد. اما در فاز تست، شبکه دیگر به بلوغ رسیده و عملکرد واقعی خود را نشان می‌دهد.

```
Starting M-RAN Training...
Training Completed.
Final Neurons: 43
Final Training RMSE: 12.0987
Test RMSE: 5.9855
```



پاسخ تشریحی بخش ۲/۲/۲: گزارش نهایی عملکرد و هایپرپارامترها

ما در این بخش، نتایج نهایی پیاده‌سازی شبکه عصبی RBF تطبیقی (M-RAN) را مرور می‌کنیم. مدلی که ما طراحی کردیم بر اساس یادگیری ترتیبی (Sequential Learning) عمل می‌کند و در طول آموزش، ساختار خود را تغییر می‌دهد.

۱. تنظیمات و هایپرپارامترها:

بر اساس کدی که اجرا کردیم، بهترین عملکرد شبکه با مجموعه پارامترهای زیر به دست می‌آید:

مدل: شبکه عصبی RBF تطبیقی (M-RAN)

روش آموزش: یادگیری ترتیبی (Sequential)

آستانه خطا (e_{min}): ۰,۱

آستانه تازگی اولیه (ϵ_{init}): ۱,۰ (با نرخ کاهش ۰,۹۹۹)

نرخ یادگیری (η): ۰,۰۵

پنجره زمانی هرس (M): ۶۰ نمونه

ضریب همپوشانی (κ): ۰,۸

آستانه حذف (Δ): ۰,۱

۲. نتایج ارزیابی:

ما پس از آموزش مدل روی داده‌های Train و تست آن روی داده‌های Test، به نتایج زیر می‌رسیم:

خطای نهایی (RMSE) روی داده‌های تست: ۵,۹۸

تعداد نهایی نورون‌ها: ۴۳ عدد

پاسخ تشریحی بخش ۳/۲/۲: تحلیل نمودارها و مصالحه دقت-پایداری

ما در این بخش، نمودارهای خروجی و رفتار دینامیک شبکه را تحلیل می‌کنیم.

۱. تحلیل نمودار پیش‌بینی (Prediction Plot):

با نگاه به نمودار تطبیق خروجی (خط‌چین قرمز روی خط مشکی)، مشاهده می‌کنیم که M-RAN توانسته است الگوهای پیچیده و حتی اسپایک‌های (Spikes) ناگهانی سیستم Ball and Beam را به خوبی یاد بگیرد. این دقت بالا با تعداد کمی نورون (۴۳ عدد) حاصل شده که نشان‌دهنده کارایی بالای روش تطبیقی است.

۲. تحلیل نمودار رشد نورون‌ها (Pruning History & Growth):

در نمودار آبی رنگ (تعداد نورون‌ها بر حسب زمان)، ما همان رفتار مورد انتظار (مشابه شکل A2/A3 مقاله مرجع) را می‌بینیم. شبکه ابتدا با سرعت شروع به رشد می‌کند و تعداد نورون‌ها بالا می‌رود (فاز یادگیری)، اما سپس با فعال شدن مکانیزم هرس (Pruning) و تثبیت یادگیری، تعداد نورون‌ها نوسان کرده و نورون‌های اضافی حذف می‌شوند تا شبکه به یک ساختار بهینه و فشرده برسد.

۳. تحلیل مصالحه بین دقت و پایداری (Bias-Variance Tradeoff):

ما در اینجا با یک چالش تئوری مهم روبرو هستیم:

اگر تعداد نورون‌ها خیلی زیاد شود، دقت روی داده‌های آموزش بالا می‌رود اما ریسک **ناپایداری** ماتریس‌ها (بدخیم شدن Φ) و **Overfitting** افزایش می‌یابد.

هدف نهایی ما یافتن نقطه‌ای است که شبکه هم «دقیق» باشد (Bias کم) و هم روی داده‌های جدید «پایدار» بماند (Variance کم). الگوریتم M-RAN با تنظیم دینامیک تعداد نورون‌ها، سعی می‌کند دقیقاً در همین نقطه بهینه قرار بگیرد.

پاسخ سوال ۴/۲/۲: مقایسه نهایی شبکه ایستا و تطبیقی

ما در این بخش، نتایج نهایی شبکه هوشمند **M-RAN** را با بهترین حالت شبکه **Static** (که در بخش ۲,۵,۱ با سیگمای تطبیقی به دست آمد) مقایسه می‌کنیم.

۱. مقایسه اعداد:

شبکه ایستا (Static RBFNN): ما با ۵۰ نورون و دسترسی به کل داده‌ها، به خطای **RMSE = 3.37** رسیدیم.

شبکه تطبیقی (Adaptive M-RAN): ما با ۴۳ نورون و یادگیری ترتیبی، به خطای **RMSE = 5.98** رسیدیم.

۲. تحلیل تفاوت:

ما مشاهده می‌کنیم که شبکه ایستا خطای کمتری دارد. علت این است که این شبکه «کل داده‌ها» را به صورت یکجا (Batch) می‌بیند و با الگوریتم K-Means، مراکز را در بهترین نقاط سراسری قرار می‌دهد. اما شبکه M-RAN داده‌ها را «یکی‌یکی» می‌بیند و دید کلی نسبت به آینده ندارد.

با این حال، نقطه قوت اصلی M-RAN این است که توانسته با تعداد نورون کمتر (۴۳ در برابر ۵۰) و بدون نیاز به پردازش سنگین یکجا، به خطای قابل قبولی برسد که برای سیستم‌های Online بسیار ارزشمند است.

پاسخ سوال ۵/۲/۲:

۱. مزایای رویکرد تطبیقی (M-RAN):

ساختار خودکار (Automatic Structure): برخلاف روش ایستا که مجبور بودیم تعداد مراکز (K) را با سعی و خطا حدس بزنیم، در اینجا شبکه خودش معماری لازم را کشف می‌کند.

توپولوژی حداقل (Minimal Topology): شبکه با مکانیزم‌های Growth و Pruning موفق شد تنها با ۴۳ نورون رفتار سیستم را مدل‌سازی کند. این بسیار بهینه‌تر از حالت ایستا است که گاهی تا ۲۰۰ نورون نیاز داشتیم.

یادگیری آنلاین (Online Learning): این روش نیازی به ذخیره کل داده‌ها ندارد و برای سیستم‌های Real-time و پردازش جریانی (Streaming) ایده‌آل است.

۲. معایب و چالش‌ها:

حساسیت به هایپرپارامترها: تنظیم پارامترهایی مثل ϵ، e_{min} و پنجره هرس (M) دشوارتر از تنظیم یک عدد K در روش ایستا است و تأثیر زیادی بر نتیجه دارد.

دقت کمتر نسبت به Batch: چون شبکه داده‌ها را یکجا نمی‌بیند، همگرایی آن کمی ضعیف‌تر از روش LLS سراسری است (خطای ۵,۹۸ در برابر ۳,۳۷).

ما دیدیم که شبکه بدون هیچ دانش قبلی، تعداد نورون‌ها را تا حدود ۵۰ بالا برد و سپس با حذف نورون‌های اضافی، روی عدد ۴۳ متوقف شد. این یعنی شبکه توانسته است "تعداد ضروری" نورون‌ها برای توصیف این سیستم خاص را پیدا کند.

۴. اهمیت در کاربردهای دنیای واقعی:

این ویژگی برای سخت‌افزارهای محدود (مثل ربات‌های متحرک یا تراشه‌های موبایل) حیاتی است، زیرا:

محدودیت سخت‌افزار: شبکه کوچک‌تر به معنای مصرف حافظه کمتر، محاسبات سریع‌تر و مصرف باتری کمتر است.

محیط‌های متغیر: در دنیای واقعی شرایط عوض می‌شود؛ یک شبکه ایستا (Static) نمی‌تواند خود را تغییر دهد، اما M-RAN می‌تواند با محیط جدید سازگار شود.