

به نام خدا



دانشگاه تهران



دانشکده مهندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

تمرین اول

علی صفری	نام و نام خانوادگی	پرسش ۱
۸۱۰۲۰۲۱۵۳	شماره دانشجویی	
حمیدرضا نادی مقدم	نام و نام خانوادگی	پرسش ۲
۸۱۰۱۰۳۲۶۴	شماره دانشجویی	
۱۴۰۳۰۸۱۵	مهلت ارسال پاسخ	

فهرست

۱ مقدمه
۱ پرسش ۱. طراحی و تحلیل شبکه عصبی چند لایه (MLP)
۱ ۱. طراحی MLP
۴ -۱-۱-۱ ترسیم ماتریس آشفتگی
۶ ۱-۱-۲- کلاس‌هایی که با هم اشتباه گرفته می‌شوند
۷ -۱-۱-۳ تاثیر افزایش نورون بر حل مسائل پیچیده
۸ ۱-۱-۴- معیارهایی برای پیکربندی
۸ ۲-۱ آموزش دو مدل متفاوت
۱۱ ۱-۲-۱- هیستوگرام وزن‌ها
۱۴ ۲-۲-۱ تاثیر بهینه‌ساز
۲۲ ۳-۱ الگوریتم بازگشت به عقب
۲۳ ۳-۱-۱- پیاده‌سازی سه الگوریتم بهینه‌ساز
۳۸ ۳-۱-۲- چگونگی اثر جستجوی بیزی
۴۳ ۴-۱ بررسی هایپرپارامترهای مختلف
۶۳ ۴-۱-۱- استفاده از Random Search
۶۹ ۴-۱-۲- بررسی ماتریس آشفتگی با تغییر هایپرپارامترها
۸۹ ۲- آموزش و ارزیابی یک شبکه عصبی ساده
۸۹ ۱-۲ آموزش یک شبکه عصبی
۹۴ ۲-۲ آزمون شبکه عصبی بر روی یک مجموعه داده
۱۰۵ پرسش ۳ Madaline –
۱۰۵ ۱-۳ الگوریتم‌های MRI و MII
۱۰۵ ۲-۳ نمودار پراکندگی داده‌ها
۱۰۶ ۳-۳ آموزش مدل

۱۰۹	۴-۳. تحلیل نتایج
۱۱۱	پرسش ۴ – MLP
۱۱۱	۴-۱. نمایش تعداد ستون
۱۱۱	۴-۲. ماتریس همبستگی
۱۱۳	۴-۳. رسم نمودار
۱۱۴	۴-۴. پیش پردازش داده
۱۱۵	۴-۵. پیاده سازی مدل
۱۱۶	۴-۶. آموزش مدل
۱۱۷	تحلیل نتایج

شکل‌ها

- شکل ۱ – import کتابخانه‌های مورد نیاز ۱
- شکل ۲ – استخراج دیتا و خطی کردن تصاویر از دیتابیس Fasion MNIST ۱
- شکل ۳ – تفاوت ابعادی X_{train_flan} و X_{trian} ۲
- شکل ۴ – تعریف مدل، لایه‌ها و کامپایل کردن مدل با استفاده از بهینه‌ساز Adam و تابع loss ۳
- شکل ۵ – نتیجه فیت کردن مدل در ۵ epoch ۳
- شکل ۶ – محاسبه و ترسیم ماتریس آشفتگی ۴
- شکل ۷ – ماتریس آشفتگی ۵
- شکل ۸ – محاسبه و نمایش کلاس‌های با بیشترین تشابه ۶
- شکل ۹ – تشخیص تعداد کلاس و اجرای تابع تشخیص کلاس‌های مشابه ۶
- شکل ۱۰ – خروجی کلاس‌های با بیشترین تشابه ۷
- شکل ۱۱ – کتابخانه‌های مورد نیاز ۸
- شکل ۱۲ – تعریف مدل با ساختار اولیه ۱۲۸ نورون و بدون dropout و نهایتاً اجرای مدل ۹
- شکل ۱۳ – تعریف مدل با ساختار ثانویه ۴۸ نورون و یک لایه dropout و نهایتاً اجرای مدل ۹
- شکل ۱۴ – نتیجه اجرای مدل اول ۱۰
- شکل ۱۵ – نتیجه اجرای مدل دوم ۱۱
- شکل ۱۶ – تابع نوشته شده برای توزیع وزن‌ها ۱۲
- شکل ۱۷ – فراخواندن تابع و ارسال دو مدل - بدون optimizer ۱۳
- شکل ۱۸ – هیستوگرام وزن‌های دو مدل در لایه ورودی و مخفی ۱۳
- شکل ۱۹ – مدل اولیه با استفاده از بهینه‌ساز Adam ۱۴
- شکل ۲۰ – نتیجه اجرای مدل اول با بهینه‌ساز Adam ۱۵
- شکل ۲۱ – مدل ثانویه با استفاده از بهینه‌ساز Adam ۱۶
- شکل ۲۲ – نتیجه اجرای مدل دوم با بهینه‌ساز Adam ۱۶
- شکل ۲۳ – توزیع وزن‌ها در مدل یک و دو با حضور بهینه‌ساز Adam ۱۷
- شکل ۲۴ – مدل اولیه با استفاده از بهینه‌ساز rmsprop ۱۸
- شکل ۲۵ – نتیجه اجرای مدل اول با بهینه‌ساز rmsprop – مراحل اولیه ۱۸
- شکل ۲۶ – نتیجه اجرای مدل اول با بهینه‌ساز rmsprop – مراحل پایانی ۱۹

۱۹ شکل ۲۷ - مدل ثانویه با استفاده از بهینهساز rmsprop
۲۰ شکل ۲۸ - نتیجه اجرای مدل دوم با بهینهساز rmsprop
۲۱ شکل ۲۹ - توزیع وزن‌ها در مدل یک و دو با حضور بهینهساز RMSProp
۲۳ شکل ۳۰ - فرمول محاسباتی در روش RMSProp [۱]
۲۴ شکل ۳۱ - فرمول محاسباتی در روش Adam [۲]
۲۴ شکل ۳۲ - فرمول محاسباتی روش Nadam [۳]
۲۵ شکل ۳۳ - کردن import کتابخانه‌های مورد نیاز
۲۶ شکل ۳۴ - کردن دیتای train و test
۲۶ شکل ۳۵ - خروجی load کردن دیتا
۲۷ شکل ۳۶ - تعریف کلاس کاستوم برای callback
۲۸ شکل ۳۷ - پیاده‌سازی مدل
۲۹ شکل ۳۸ - تعریف مدل‌ها و بهینه‌ساز برای هر کدام و ذخیره اطلاعات هر کدام در لیست result
۳۰ شکل ۳۹ - مدل یک - Adam
۳۰ شکل ۴۰ - مدل یک - RMSprop
۳۰ شکل ۴۱ - مدل یک - Nadam
۳۱ شکل ۴۲ - مدل دو - Adam
۳۱ شکل ۴۳ - مدل دو - RMSprop
۳۲ شکل ۴۴ - مدل دو - Nadam
۳۳ شکل ۴۵ -تابع نمایش accuracy و loss هر نتیجه به ازای گام محاسباتی
۳۴ شکل ۴۶ - تابع confusion matrix
۳۴ شکل ۴۷ - مشاهده نتایج و مقایسه هر بهینه ساز در هر مدل
۳۵ شکل ۴۸ - نتایج و بررسی بهینه‌سازها روی نمودار
۳۶ شکل ۴۹ - ماتریس آشفتگی مدل یک - Nadam
۳۷ شکل ۵۰ - ایجاد فایل csv
۳۸ شکل ۵۱ - خلاصه نتایج
۳۹ شکل ۵۲ - کتابخانه‌ها و تقسیم دیتای تست و ولیدیشن
۴۰ شکل ۵۳ - ساخت مدل برای بهینه سازی پارامترهای مدل به روش Beysian - بخش اول
۴۱ شکل ۵۴ - بخش compile مدل
۴۲ شکل ۵۵ - تعریف Bayesian Optimizer

شکل ۵۶ - ران کردن Beysian optimizer	۴۲
شکل ۵۷ - نتیجه Beysian optimization	۴۳
شکل ۵۸ - تعریف کلاس برای callback	۴۴
شکل ۵۹ - تعریف مدل‌های متفاوت برای بررسی هایپرپارامترها - بخش اول	۴۵
شکل ۶۰ - تعریف مدل‌های متفاوت برای بررسی هایپرپارامترها - بخش دوم	۴۶
شکل ۶۱ - تعریف مدل‌های متفاوت برای بررسی هایپرپارامترها - بخش سوم	۴۷
شکل ۶۲ - تعریف مدل‌های متفاوت برای بررسی هایپرپارامترها - تعریف بهینه‌سازهای مختلف	۴۷
شکل ۶۳ - بررسی هایپرپارامترها - بخش اول - learning rate	۴۸
شکل ۶۴ - نتیجه مدل اول - learning rate 0.0001	۴۹
شکل ۶۵ - نتایج مدل اول - learning rate 0.001	۵۰
شکل ۶۶ - نتایج مدل اول - learning rate 0.01	۵۰
شکل ۶۷ - نمودار تغییرات loss و accuracy با تغییر learning rate	۵۱
شکل ۶۸ - بررسی هایپرپارامترها - بخش دوم - کنترل تعداد نورون‌ها + حضور یا عدم حضور dropout	۵۲
شکل ۶۹ - بررسی هایپرپارامترها - بخش دوم - مدل شماره یک	۵۴
شکل ۷۰ - بررسی هایپرپارامترها - بخش دوم - مدل شماره دو	۵۴
شکل ۷۱ - بررسی هایپرپارامترها - بخش دوم - مدل شماره سه	۵۴
شکل ۷۲ - بررسی هایپرپارامترها - بخش دوم - مدل شماره چهار	۵۵
شکل ۷۳ - بررسی هایپرپارامترها - بخش دوم - مدل شماره پنج	۵۵
شکل ۷۴ - بررسی هایپرپارامترها - بخش دوم - مدل شماره شش	۵۵
شکل ۷۵ - بررسی هایپرپارامترها - بخش دوم - مدل شماره هفت	۵۶
شکل ۷۶ - نمودار تغییرات loss و accuracy با تغییر تعداد نورون و وجود و عدم وجود dropout	۵۷
شکل ۷۷ - بررسی هایپرپارامترها - بخش سوم - تعداد لایه‌ها	۵۸
شکل ۷۸ - بررسی هایپرپارامترها - بخش سوم - مدل شماره هشت	۶۰
شکل ۷۹ - بررسی هایپرپارامترها - بخش سوم - مدل شماره نه	۶۰
شکل ۸۰ - بررسی هایپرپارامترها - بخش سوم - مدل شماره ده	۶۰
شکل ۸۱ - بررسی هایپرپارامترها - بخش سوم - مدل شماره یازده	۶۱
شکل ۸۲ - دقیقت مدل	۶۲
شکل ۸۳ - نمودار تغییرات loss و accuracy با تغییر تعداد لایه‌ها	۶۲

..... ۶۴	شكل ۸۴ - نصب کتابخانه kares-tuner
..... ۶۴ شکل ۸۵ - تعریف کلاس dataloader
..... ۶۵ شکل ۸۶ - تعریف کلاس ایجاد مدل
..... ۶۶ شکل ۸۷ - کامپایل و تعریف HyperparameterTuner
..... ۶۷ شکل ۸۸ - تعریف تابع fine_best_model
..... ۶۸ شکل ۸۹ - اجرای random search برای هایپرپارامترها
..... ۶۹ شکل ۹۰ - نتیجه Random Search
..... ۷۰ شکل ۹۱ - ماتریس آشفتگی مدل یک با نرخ یادگیری ۰.۰۰۰۱
..... ۷۲ شکل ۹۲ - ماتریس آشفتگی مدل یک با نرخ یادگیری ۰.۰۰۱
..... ۷۳ شکل ۹۳ - ماتریس آشفتگی مدل یک با نرخ یادگیری ۰.۰۱
..... ۷۵ شکل ۹۴ - ماتریس آشفتگی مدل یک
..... ۷۶ شکل ۹۵ - نتایج ماتریس آشفتگی مدل دو
..... ۷۷ شکل ۹۶ - نتایج ماتریس آشفتگی مدل سه
..... ۷۸ شکل ۹۷ - نتایج ماتریس آشفتگی مدل چهار
..... ۷۹ شکل ۹۸ - نتایج ماتریس آشفتگی مدل پنج
..... ۸۰ شکل ۹۹ - نتایج ماتریس آشفتگی مدل شش
..... ۸۱ شکل ۱۰۰ - نتایج ماتریس آشفتگی مدل هفت
..... ۸۳ شکل ۱۰۱ - نتایج ماتریس آشفتگی مدل هشت
..... ۸۵ شکل ۱۰۲ - نتایج ماتریس آشفتگی مدل نه
..... ۸۶ شکل ۱۰۳ - نتایج ماتریس آشفتگی مدل ده
..... ۸۷ شکل ۱۰۴ - نتایج ماتریس آشفتگی مدل یازده
..... ۸۹ شکل ۱۰۵ - معادلات بکار رفته در این سوال با توجه به کتاب بیشап [۴]
..... ۹۰ شکل ۱۰۶ - تابع forward
..... ۹۲ شکل ۱۰۷ - تابع backward - بخش اول
..... ۹۳ شکل ۱۰۸ - تابع backward - بخش دوم
..... ۹۴ شکل ۱۰۹ - تابع backward جدید - بخش اول
..... ۹۵ شکل ۱۱۰ - آپلود فایل csv به عنوان دیتای ورودی
..... ۹۵ شکل ۱۱۱ - استخراج اطلاعات و نرمالیزیشن و اضافه کردن ستون bias به دیتا
..... ۹۶ شکل ۱۱۲ - تابع backward - بخش دوم

شکل ۱۱۳ - تابع backward	- بخش سوم	۹۷
شکل ۱۱۴ - تابع محاسبه accuracy و squared_error	۹۸	
شکل ۱۱۵ - تابع ترسیم تغییرات loss و accuracy بر حسب iteration	۹۹	
شکل ۱۱۶ - تابع ترسیم error بر حسب iteration - خروجی تابع backward	۹۹	
شکل ۱۱۷ - توابع محاسبه RMS و کنترل مراحل train	۱۰۰	
شکل ۱۱۸ - اجرای الگوریتم بر روی دیتا	۱۰۱	
شکل ۱۱۹ - نمودار تغییرات loss و accuracy با learning rate 1e-06	۱۰۱	
شکل ۱۲۰ - نمودار تغییرات loss و accuracy با learning rate 1e-05	۱۰۲	
شکل ۱۲۱ - نمودار تغییرات loss و accuracy با learning rate 1e-04	۱۰۲	
شکل ۱۲۲ - نمودار تغییرات خطای iteration بر حسب learning rate با های متفاوت	۱۰۴	
شکل ۱۲۳ - کد نمودار پراکندگی	۱۰۵	
شکل ۱۲۴ - نمودار پراکندگی	۱۰۶	
شکل ۱۲۵ - آموزش مدل‌ها	۱۰۶	
شکل ۱۲۶ - ارزیابی مدل‌ها	۱۰۷	
شکل ۱۲۷ - کد خطاهای جداکننده	۱۰۷	
شکل ۱۲۸ - مرز تصمیم در ۳ نورون	۱۰۸	
شکل ۱۲۹ - مرز تصمیم در ۴ نورون	۱۰۸	
شکل ۱۳۰ - مرز تصمیم در ۸ نورون	۱۰۹	
شکل ۱۳۱ - کد رسم نمودار برای هر مدل	۱۰۹	
شکل ۱۳۲ - نمودار دقیق برای هر مدل	۱۱۰	
شکل ۱۳۳ - دقیق هر مدل	۱۱۰	
شکل ۱۳۴ - دقیق هر مدل	۱۱۰	
شکل ۱۳۵ - کد و نتیجه ستون‌های NaN	۱۱۱	
شکل ۱۳۶ - کد ماتریس همبستگی و نمایش ویژگی-های correlation که بیشترین را با قیمت دارد	۱۱۲	
شکل ۱۳۷ - ماتریس همبستگی	۱۱۳	
شکل ۱۳۸ - کد رسم نمودار	۱۱۳	
شکل ۱۳۹ - نمودار توزیع قیمت خانه	۱۱۴	
شکل ۱۴۰ - نمودار توزیع قیمت و ویژگی sqft_living با بیشترین همبستگی	۱۱۴	

شکل ۱۴۱ - حذف ستون تاریخ و نمایش سه سطر اول	۱۱۵
شکل ۱۴۲ - تقسیم داده‌ها به دو بخش تست و آموزش.....	۱۱۵
شکل ۱۴۳ - مقیاس گذاری داده‌های تست و آموزش.....	۱۱۵
شکل ۱۴۴ - پیاده سازی شبکه عصبی با یک لایه مخفی.....	۱۱۶
شکل ۱۴۵ - پیاده سازی شبکه عصبی با دو لایه مخفی	۱۱۶
شکل ۱۴۶ - آموزش با یک لایه مخفی.....	۱۱۶
شکل ۱۴۷ - آموزش با دو لایه مخفی.....	۱۱۶
شکل ۱۴۸ - کد نمایش مقایسه loss تست و آموزش	۱۱۷
شکل ۱۴۹ - نمودار مقایسه loss تست و آموزش.....	۱۱۷
شکل ۱۵۰ - انتخاب و پیش‌بینی ۵ نمونه تصادفی.....	۱۱۸

جدول‌ها

۲۱	جدول ۱. مقایسه شبکه‌ها بدون استفاده از بهینه‌ساز
۲۲	جدول ۲. مقایسه شبکه‌ها با استفاده از Adam
۲۲	س ۳. مقایسه شبکه‌ها با استفاده از RMSProp
۵۰	جدول ۴ - مقایسه اثر learning rate
۵۳	جدول ۵ - انواع مدل با تعداد نورون‌های متفاوت و حضور و عدم حضور dropout
۵۶	جدول ۶ - مقایسه دقت مدل‌های یک تا هفت با معماری متفاوت
۵۸	جدول ۷ - انواع مدل با لایه‌های مخفی متفاوت
۶۱	جدول ۸ - مقایسه دقت مدل‌های هشت تا یازده با معماری متفاوت
۷۴	جدول ۹ - اثر نرخ یادگیری در ماتریس آشفتگی
۸۲	جدول ۱۰ - اثر تعداد نورون و Dropout در ماتریس آشفتگی
۸۸	جدول ۱۱ - اثر تعداد لایه‌های مخفی در ماتریس آشفتگی
۱۰۴	جدول ۱۲ - مقایسه RMSE

برای پیاده‌سازی پروژه از بستر Google Colab برای کد نویسی استفاده شده است. تمامی مراحل کد و اجرای آن در این گزارش به تفصیل شرح داده شده است.

کدهای نوشته شده همگی در پوشه‌ی Code و با پسوند .ipynd ذخیره شده است.

در بخش‌هایی که اشاره به استفاده یا عدم استفاده از Optimizer یا تابع loss خاصی نشده است فرض بر آزاد بودن استفاده از هر کدام شده است. به عنوان مثال در مسئله اول از تابع Cross entropy برای تابع loss استفاده شده که مناسب multi-class classification است.

در بخش اجرای کد مربوط به شبکه در تمامی سوالات ما شاهد چهار مقدار، accuracy, loss, val_loss و val_accuracy هستیم. عملکرد به این صورت است که یک بخش به صورت اتوماتیک به اختصاص داده می‌شود و دو مقدار دوم با پیشوند val مربوط به این بخش هستند. به جهت پر تکرار بودن این موضوع و جلوگیری از تکرار توضیح آن در این بخش به تشریح این مورد پرداخته شد.

پرسش ۱. طراحی و تحلیل شبکه عصبی چند لایه (MLP)

۱-۱. طراحی MLP

در ابتدا برای آشنایی بیشتر با دیتا به وبسایت Kaggle مراجعه شد. دیتابست Fasion MNIST حاوی ۶۰۰۰۰ نمونه به همراه ۱۰۰۰۰ تست از تصاویر grayscale با ابعاد 28×28 است که به هر کدام از تصاویر یک کلاس از بین ۱۰ کلاس موجود اختصاص داده شده است.

در ابتدا import های لازم برای انجام سوال را صورت گرفت. با توجه به اینکه در فضای google colab تمامی کتابخانه‌های مورد نیاز از پیش نصب شده است نیازی به نصب هیچ یک از پکیج‌ها در این فضا نیست.

```
import tensorflow as tf
from tensorflow.keras import layers, models, regularizers
from tensorflow.keras.datasets import fashion_mnist
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns
import matplotlib.pyplot as plt
from tensorflow.keras.optimizers import Adam
```

شکل ۱ – کردن کتابخانه‌های مورد نیاز

در گام بعد دیتا را از استخراج کردیم. همانطور که گفتیم هر دیتا حاوی یک تصویر 28×28 و در حالت سیاه سفید است. در ابتدا برای اینکه کار با سرعت و دقیق بالاتری پیش برود نرمالیزیشن انجام می‌دهیم. با توجه به اینکه ماکریم عدد ممکن برای هر پیکس عدد ۲۵۵ می‌باشد تمامی دیتا ورودی به این عدد تقسیم شده‌اند. در گام بعد برای اینکه بتوانیم لایه ورودی را به صورت خطی در معماری خود قرار دهیم ماتریس 28×28 به یک ماتریس خطی $28 \times 28 = 784$ عضوی تبدیل شد. به عبارت دیگر لایه ورودی ما حاوی ۷۸۴ نورون خواهد بود.

```
# Load Fashion-MNIST dataset
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

# Normalize pixel values to between 0 and 1
x_train, x_test = x_train / 255.0, x_test / 255.0

# Flatten the 28x28 images into 784-dimensional vectors
x_train_flat = x_train.reshape(-1, 28*28)
x_test_flat = x_test.reshape(-1, 28*28)
```

شکل ۲ – استخراج دیتا و خطی کردن تصاویر از دیتابیس Fasion MNIST

```
print(x_train.shape)
print(x_train_flat.shape)

(60000, 28, 28)
(60000, 784)
```

X_train_flan و X_trian شکل ۳ - تفاوت ابعادی

در بخش بعد، که مهم ترین بخش کد از نظر محاسباتی است، ابتدا یک مدل sequential تعريف می‌کنیم. این مدل حاوی لایه‌های خطی است که هر لایه به لایه بعدی خود مرتبط است. به عبارت دیگر برای پیاده‌سازی ساختار FCNN مناسب است. در گام بعد لایه‌ها مطابق فرضیات سوال ایجاد شده است. در نهایت برای بهینه ساز از بهینه ساز Adam استفاده شده و برای تابع loss هم به جهت اینکه مسئله sparse_categorical_crossentropy است از multi-class classification استفاده شده است. در نهایت برای بررسی دقیق مدل از metrics باید استفاده شود که ما اینجا از accuracy استفاده کردیم که به نوعی ساده ترین نوع است و درصد دقیق شبکه را نشان می‌دهد.

نکته مهم در این بخش استفاده از بهینه ساز است. می‌توان بدون بهینه ساز یک عدد ثابت برای learning rate در نظر گرفت و مراحل را جلو برد اما این کار نیاز به سعی و خطای زیاد تا رسیدن به یک مقدار بهینه و مناسب برای این مقدار است. به همین جهت برای تسريع عملیات محاسباتی و بدست آوردن بهینه و سریع تر learning rate از بهینه‌سازها استفاده می‌کنیم. همانطور که ذکر شد اینجا از بهینه ساز Adam استفاده شد که یکی از معروف ترین توابع است. با توجه به عدم پرسش درباره چگونگی عملکرد دقیق و ریاضیاتی بهینه‌ساز، از توضیح مفصل‌تر در این باره صرف نظر می‌کنیم.

نکته بعدی انتخاب activation function است که برای لایه‌های مخفی از ReLU و برای لایه خروجی از SoftMax استفاده شده است. دلیل استفاده از ReLU در لایه‌های مخفی چندین مورد است. اول اینکه این تابع یک ساختار غیرخطی ارائه می‌کند که به شبکه کمک می‌کنه الگوهای مخفی و غیر خطی موجود در دیتا را بهتر شناسایی کند. مورد بعدی فعال شدن این تابع فقط در شرایط مثبت است که سرعت محاسباتی را به شدت افزایش می‌دهد. در واقع به ازای مقادیر منفی نورون اکتیو نمی‌شود. نکته اخر اما کم کردن اثر vanishing gradient هست. به این معنا که با مشتق گیری‌های پیاپی، و نزدیک شدن به لایه‌های ورودی اثر گرادیان مانند تابع Activation sigmoid باعث پدیده vanishing gradient نخواهد شد. اما چرا از softmax استفاده می‌کنیم؟ زیرا در لایه اخر خروجی ما باید عددی بین صفر و یک باشد. به عنوان مثال تصویر ورودی ما با چه احتمالی مربوط به کلاس شماره n است. هر چه عددی به یک نزدیک‌تر باشد به معنای تعلق ورودی ما به آن کلاس است.

```

# Define the MLP model
model = models.Sequential()
model.add(layers.Input(shape=(28*28,))) # Define input shape here

# Input layer (Flatten is applied earlier)
# Hidden layer with 100 neurons, ReLU activation, L2 regularization, and Dropout
model.add(layers.Dense(100, activation='relu',
                      kernel_regularizer=regularizers.l2(0.0001)))
model.add(layers.Dropout(0.3)) # Dropout with 30%

# Output layer with 10 neurons (one for each class), Softmax activation
model.add(layers.Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
history = model.fit(x_train_flat, y_train, epochs=10, validation_data=(x_test_flat, y_test))

```

شکل ۴ - تعریف مدل، لایه ها و کامپایل کردن مدل با استفاده از بهینه ساز Adam و تابع cross entropy loss

در ادامه خروجی کد را داریم:

```

Epoch 1/10
1875/1875 13s 7ms/step - accuracy: 0.7408 - loss: 0.7627 - val_accuracy: 0.8445 - val_loss: 0.4621
Epoch 2/10
1875/1875 11s 6ms/step - accuracy: 0.8381 - loss: 0.4738 - val_accuracy: 0.8525 - val_loss: 0.4373
Epoch 3/10
1875/1875 21s 6ms/step - accuracy: 0.8509 - loss: 0.4459 - val_accuracy: 0.8603 - val_loss: 0.4232
Epoch 4/10
1875/1875 14s 3ms/step - accuracy: 0.8605 - loss: 0.4253 - val_accuracy: 0.8481 - val_loss: 0.4424
Epoch 5/10
1875/1875 10s 3ms/step - accuracy: 0.8647 - loss: 0.4104 - val_accuracy: 0.8553 - val_loss: 0.4442
Epoch 6/10
1875/1875 8s 4ms/step - accuracy: 0.8657 - loss: 0.4090 - val_accuracy: 0.8574 - val_loss: 0.4304
Epoch 7/10
1875/1875 9s 3ms/step - accuracy: 0.8666 - loss: 0.4068 - val_accuracy: 0.8686 - val_loss: 0.4238
Epoch 8/10
1875/1875 10s 5ms/step - accuracy: 0.8715 - loss: 0.3968 - val_accuracy: 0.8687 - val_loss: 0.4074
Epoch 9/10
1875/1875 6s 3ms/step - accuracy: 0.8709 - loss: 0.3974 - val_accuracy: 0.8723 - val_loss: 0.4035
Epoch 10/10
1875/1875 8s 4ms/step - accuracy: 0.8704 - loss: 0.3972 - val_accuracy: 0.8670 - val_loss: 0.4080

```

شکل ۵ - نتیجه فیت کردن مدل در epoch

شکل ۵ نشان میدهد شبکه با دقت خوبی در هر epoch مقدار loss را کاهش داده است و نهایتا در گام آخر با دقت قابل قبول حدود ۸۷ درصد به کار خود پایان داده است. البته گام اخر شاهد افزایش بسیار اندک loss هستیم. برای رفع این مشکل می‌توان از بهینه ساز های دیگر و یا learning rate اولیه متفاوت و حتی افزایش epoch استفاده کرد اما این مقدار اندک با توجه به دقت قابل قبول امری طبیعی قلمداد می‌شود. روند کاهشی تابع loss به این معنا است که در هر مرحله ما یک قدم به نقطه optimum نزدیک تر شدیم. به عبارت دیگر اگر تابع loss تعریف شده برای این مسئله convex در نظر بگیریم (چنانچه که همینطور هست با توجه به روند کاهش دائمی و همینطور جنس مسئله و مطابقت با تابع loss انتخاب شده) در هر گام محاسباتی به نقطه optimum نزدیک و نزدیکتر شدیم. از طرفی چون مقدار accuracy در هر گام محاسباتی به نقطه optimum نزدیک و نزدیکتر شدیم.

و validation accuracy بسیار به هم نزدیک است، می‌توان نتیجه گرفت مدل به خوبی قابلیت تعمیم دارد و از overfitting فاصله دارد.

۱-۱-۱ - ترسیم ماتریس آشتفتگی

در گام بعد برای بررسی بهتر دقت کار ماتریس آشتفتگی را با کد زیر ترسیم می‌کنیم.

```
# Predict on the test set
y_pred = model.predict(x_test_flat)
y_pred_classes = y_pred.argmax(axis=1)

# Generate the confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred_classes)

# Plot the confusion matrix as a heatmap using Seaborn
plt.figure(figsize=(10,8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', cbar=False)
plt.xlabel('Predicted')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()
```

شکل ۶ - محاسبه و ترسیم ماتریس آشتفتگی

اتفاقی که در کد موجود در تصویر ۶ افتاده این است که ما در نهایت برای هر عکس یک لیست حاوی ۱۰ مقدار داریم که بیانگر میزان نزدیکی تصویر اولیه به هر کلاس است. برای اینکه پیشビینی نهایی انجام شود باید ماکزیمم گرفته شود و این کلاس به عنوان کلاس پیشビینی شده برای تصویر ورودی در نظر گرفته می‌شود. در نهایت ماتریس ما برای مقادیر تست بین مقادیر واقعی (y_{test}) و مقادیر پیشビینی شده توسط شبکه ($y_{pred_classes}$) ماتریس ترسیم می‌کنیم.

Confusion Matrix											
True Label	0	1	2	3	4	5	6	7	8	9	
	0	825	2	25	44	4	0	91	0	9	0
	1	2	969	2	20	3	0	3	0	1	0
	2	14	1	827	16	102	0	39	0	1	0
	3	18	16	11	900	31	0	21	0	3	0
	4	0	1	141	46	789	0	21	0	2	0
	5	0	0	0	1	0	968	0	20	2	9
	6	148	1	157	40	110	0	534	0	10	0
	7	0	0	0	0	0	24	0	950	0	26
	8	5	0	3	7	4	2	12	3	964	0
	9	1	0	0	0	0	10	0	45	0	944
Predicted											

شکل ۷ - ماتریس آشفتگی

قطر ماتریس آشفتگی نشان دهنده تعداد آیتم های درست پیشبینی شده توسط مدل ماست و هر چه به رنگ تیره نزدیک تر باشد این عدد بیشتر بوده است و همانطور که در ماتریس شکل ۷ می بینیم، مدل با دقت خوبی پیشبینی کرده است. مقادیر موجود در بالا و پایین قطر اصلی نمایانگر کلاس های به اشتباہ شناسایی شده اند به عنوان مثال تصاویری که برچسب ۸ را داشته اند توسط شبکه ما ۷ بار به اشتباہ برچسب ۳ را دریافت کرده اند.

۱-۱-۲ - کلاس‌هایی که با هم اشتباه گرفته می‌شوند

```
[32] def most_confused_class(classes_number, conf_matrix):
    # Corrected logic to find the most confused classes
    for i in range(classes_number):
        conf_row = conf_matrix[i].copy() # Copy the row to avoid changing the original matrix
        conf_row[i] = -1 # Set diagonal element to -1 to ignore it (so we can find the second-largest)
        most_confused_class = conf_row.argmax() # Now find the argmax, ignoring diagonal
        print(f"Class {i} is most often ({conf_matrix[i][most_confused_class]} times)"
              "confused with Class {most_confused_class}.")  
  
    print("\n")  
  
    # Identify the two classes most confused with each other
    max_confusion = 0
    confused_pair = (0, 0)
    for i in range(10):
        for j in range(10):
            if i != j and conf_matrix[i][j] + conf_matrix[j][i] > max_confusion:
                max_confusion = conf_matrix[i][j] + conf_matrix[j][i]
                confused_pair = (i, j)  
  
print(f"The two classes most confused with each other are: {confused_pair}")
print(f"The two classes most confused with each other are: {max_confusion}")
```

شکل ۸ - محاسبه و نمایش کلاس‌های با بیشترین تشابه

در تصویر ۸ کد مربوط به پیدا کردن کلاس‌هایی که بیشتر از همه با هم اشتباه گرفته شده اند را داریم. برای این کار ابتدا یکتابع نوشته شد که تعداده کلاس‌های موجود و ماتریس آشغتگی را به عنوان ورودی گرفته و نهایتا محاسبات را بر این اساس انجام می‌دهد.

```
# Find unique classes in the training labels
unique_classes = np.unique(y_train)
num_classes = unique_classes.size  
  
most_confused_class(num_classes, conf_matrix)
```

شکل ۹ - تشخیص تعداد کلاس و اجرای تابع تشخیص کلاس‌های مشابه

بر اساس y_train تعداد کلاس‌های موجود هر دیتاست استخراج شده و نتیجه به تابع نوشته ارسال گردیده است.

```
Class 0 is most often (91 times)confused with Class 6.  
Class 1 is most often (20 times)confused with Class 3.  
Class 2 is most often (102 times)confused with Class 4.  
Class 3 is most often (31 times)confused with Class 4.  
Class 4 is most often (141 times)confused with Class 2.  
Class 5 is most often (20 times)confused with Class 7.  
Class 6 is most often (157 times)confused with Class 2.  
Class 7 is most often (26 times)confused with Class 9.  
Class 8 is most often (12 times)confused with Class 6.  
Class 9 is most often (45 times)confused with Class 7.
```

The two classes most confused with each other are: (2, 4)

The two classes most confused with each other are: 243

شکل ۱۰ - خروجی کلاس‌های با بیشترین تشابه

نتیجه کد موجود در شکل ۹ را در شکل ۱۰ می‌بینیم. در بخش اول بر اساس نتایج مشاهده می‌شود که هر کلاس بیشتر با کدام کلاس اشتباه گرفته شده. برای اینکار کافی است سطر به سطر جلو برویم، هر سطر بیانگر لیبل واقعی ورودی و مقادیر پیش‌بینی شده توسط شبکه است، ماکزیمم مقدار در هر کدام از ستون‌ها، غیر از لیبل درست، به این معنا است که آن کلاس با کدام کلاس بیشتر اشتباه گرفته شده است. به عنوان مثال اگر سطر چهارم (لیبل شماره ۳) را بررسی کنیم می‌بینیم که ستون پنجم (لیبل شماره ۴) بیشترین عدد را دارد (غیر از لیبل درست که همان ستون چهارم یا لیبل شماره ۳ است). بنابراین کلاس ۳ بیشتر از همه با کلاس ۴ اشتباه گرفته شده است. در بخش دوم اما بیشترین خطای کلی بین دو کلاس را داریم. بیشترین میزان خطا برای کلاس ۲ و ۴ بوده است. نحوه محاسبه به این صورت بوده که تعداد تصاویری که واقعاً لیبل ۲ داشتند اما شبکه به آنها لیبل ۴ داده است به علاوهٔ تعداد تصاویری که واقعاً لیبل ۴ داشته‌اند و شبکه به آنها لیبل ۲ داده است با هم جمع‌زده شده‌اند و مشاهده شده که ۲۴۳ بار به اشتباه تشخیص داده شده است. این کار برای تمامی کلاس‌ها انجام شده تا تشخیص داده شود هر یک بیشتر با کدام اشتباه گرفته شده و نهایتاً بیشترین میزان خطا از بین تمامی کلاس‌ها بررسی و گزارش شده است.

۱-۱-۳ - تاثیر افزایش نورون بر حل مسائل پیچیده

تعداد نورون‌های بیشتر می‌تواند ظرفیت شبکه ما را برای کشف روابط پیچیده تر و پنهان موجود در داده را بالا ببرد. به عبارتی با تعداد نورون بیشتر شناسی ما برای رمزگشایی پیچیدگی‌ها بالا تر می‌رود اما از طرفی احتمال وقوع پدیده‌هایی مانند overfitting هم وجود دارد چرا که ممکن است انقدر شبکه روی دیتای train خوب آموزش دیده باشد که قابلیت تعمیم برای کل داده را نداشته باشد که این مورد هم

مطلوب ما نیست برای همین از لایه regularization و Dropout استفاده می‌کنیم تا مدل قابلیت تعمیم بهتری داشته باشد.

۱-۱-۴ - معیارهایی برای پیکربندی

بهترین پیکربندی معمولاً با استفاده از معیارهای عملکردی مانند دقت اعتبارسنجی یا میزان خطا تعیین می‌شود. اگر یک مجموعه تست جداگانه در دسترس باشد، دقت تست نیز می‌تواند مورد استفاده قرار گیرد. علاوه بر این، تکنیک‌هایی مانند cross-validation می‌توانند به ارزیابی عملکرد مدل در زیرمجموعه‌های مختلف داده‌ها کمک کنند. برای جلوگیری از overfitting، تعادل بین پیچیدگی مدل و توانایی تعمیم‌دهی بسیار مهم است. تکنیک‌های تنظیم، نرخ یادگیری و توقف زودهنگام هم از روش‌های مناسب برای این مورد هستند.

۱-۲. آموزش دو مدل متفاوت

در این بخش بدون استفاده از learning rate اولیه شروع به کار می‌کنیم و هر دو معماری گفته شده را پیاده سازی می‌کنیم. برای رسیدن به accuracy مناسب باید چندین learning rate را امتحان کنیم تا به مقدار مناسب برسیم. اما پیش از پیاده‌سازی پیش‌بینی می‌شود برای حالتی که dropout نداریم، به دلیل وجود نورون‌های بیشتر، وزن بیشتری برای هر نورون در نظر گرفته شود و به عبارتی گویا مدل رفتار چکشی تری خواهد داشت. این موضوع می‌تواند به overfitting هم منجر شود. در ادامه با پیاده‌سازی مدل و بررسی هیستوگرام‌ها صحت این گزاره بررسی خواهد شد.

```
import tensorflow as tf
from tensorflow.keras import layers, models, regularizers, optimizers
from tensorflow.keras.optimizers import Adam, RMSprop
from tensorflow.keras.datasets import fashion_mnist
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
```

شکل ۱۱ - کتابخانه‌های مورد نیاز

ابتدا import های لازم را انجام دادیم و سپس مشابه بخش اول سوال و عکس ؟؟؟ دیتابی مورد نیاز را استخراج و به حالت خطی در می‌آوریم. در گام بعد شبکه را پیاده‌سازی می‌کنیم. توجه شود که برای این بخش از سوال از الگوریتم بهینه‌سازی gradient descent استفاده شده و یک عدد ثابت برای learning rate در نظر گرفته شده است. این عدد باید عدد کوچکی باشد (عموماً به این صورت است) به همین دلیل عدد ۰.۰۰۰۰۱ به عنوان learning rate اولیه و ثابت ما در این حالت انتخاب شد. برای پیاده‌سازی الگوریتم

ثبت استفاده شده learning rate و optimizer با عنوان SGD از gradien descent موجود در کتابخانه به است.

```
# Define Model 1 with constant learning rate
model1 = models.Sequential([
    layers.Input(shape=(28*28,)),
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')
])

# Compile Model 1 with a simple SGD optimizer using a fixed learning rate
initial_lr = 0.01 # Set the learning rate you want to use initially
model1.compile(optimizer=optimizers.SGD(learning_rate=initial_lr),
               loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])

# Train Model 1 for 40 epochs
history1 = model1.fit(x_train_flat, y_train, epochs=40, validation_data=(x_test_flat, y_test))
```

شکل ۱۲- تعریف مدل با ساختار اولیه ۱۲۸ نورون و بدون dropout و نهایتاً اجرای مدل

برای ساختار دوم هم به همین صورت پیاده سازی انجام شده است.

```
# Define Model 2 with constant learning rate
model2 = models.Sequential([
    layers.Input(shape=(28*28,)),
    layers.Dense(48, activation='relu', kernel_regularizer=regularizers.l2(0.0001)),
    layers.Dropout(0.2),
    layers.Dense(10, activation='softmax')
])

# Compile Model 2 with SGD and fixed learning rate
model2.compile(optimizer=optimizers.SGD(learning_rate=initial_lr),
               loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])

# Train Model 2 for 40 epochs
history2 = model2.fit(x_train_flat, y_train, epochs=40, validation_data=(x_test_flat, y_test))
```

شکل ۱۳- تعریف مدل با ساختار ثانویه ۴۸ نورون و یک لایه dropout و نهایتاً اجرای مدل

مشابه بخش اول و به همان دلایل انتخاب شده است. تعداد epochs و نورون و Activation function باقی موارد در صورت سوال مطرح شده است.
در ادامه نتیجه اجرای این دو کد را داریم.

```
Epoch 24/40  
1875/1875 6s 3ms/step - accuracy: 0.8903 - loss: 0.3102 - val_accuracy: 0.8661 - val_loss: 0.3717  
Epoch 25/40  
1875/1875 5s 3ms/step - accuracy: 0.8898 - loss: 0.3090 - val_accuracy: 0.8663 - val_loss: 0.3755  
Epoch 26/40  
1875/1875 4s 2ms/step - accuracy: 0.8958 - loss: 0.2998 - val_accuracy: 0.8673 - val_loss: 0.3742  
Epoch 27/40  
1875/1875 6s 3ms/step - accuracy: 0.8945 - loss: 0.2987 - val_accuracy: 0.8708 - val_loss: 0.3596  
Epoch 28/40  
1875/1875 4s 2ms/step - accuracy: 0.8934 - loss: 0.3009 - val_accuracy: 0.8689 - val_loss: 0.3622  
Epoch 29/40  
1875/1875 5s 2ms/step - accuracy: 0.8946 - loss: 0.2997 - val_accuracy: 0.8734 - val_loss: 0.3552  
Epoch 30/40  
1875/1875 6s 3ms/step - accuracy: 0.8978 - loss: 0.2895 - val_accuracy: 0.8667 - val_loss: 0.3731  
Epoch 31/40  
1875/1875 4s 2ms/step - accuracy: 0.9000 - loss: 0.2858 - val_accuracy: 0.8704 - val_loss: 0.3576  
Epoch 32/40  
1875/1875 6s 3ms/step - accuracy: 0.8977 - loss: 0.2877 - val_accuracy: 0.8716 - val_loss: 0.3527  
Epoch 33/40  
1875/1875 9s 2ms/step - accuracy: 0.9004 - loss: 0.2831 - val_accuracy: 0.8721 - val_loss: 0.3528  
Epoch 34/40  
1875/1875 7s 3ms/step - accuracy: 0.8988 - loss: 0.2829 - val_accuracy: 0.8713 - val_loss: 0.3568  
Epoch 35/40  
1875/1875 10s 3ms/step - accuracy: 0.9000 - loss: 0.2806 - val_accuracy: 0.8705 - val_loss: 0.3555  
Epoch 36/40  
1875/1875 10s 2ms/step - accuracy: 0.9022 - loss: 0.2766 - val_accuracy: 0.8743 - val_loss: 0.3505  
Epoch 37/40  
1875/1875 7s 3ms/step - accuracy: 0.9045 - loss: 0.2718 - val_accuracy: 0.8742 - val_loss: 0.3470  
Epoch 38/40  
1875/1875 4s 2ms/step - accuracy: 0.9028 - loss: 0.2690 - val_accuracy: 0.8732 - val_loss: 0.3477  
Epoch 39/40  
1875/1875 4s 2ms/step - accuracy: 0.9047 - loss: 0.2676 - val_accuracy: 0.8735 - val_loss: 0.3456  
Epoch 40/40  
1875/1875 6s 3ms/step - accuracy: 0.9049 - loss: 0.2646 - val_accuracy: 0.8763 - val_loss: 0.3428
```

شکل ۱۴- نتیجه اجرای مدل اول

```

Epoch 24/40
1875/1875 - 4s 2ms/step - accuracy: 0.8700 - loss: 0.3836 - val_accuracy: 0.8647 - val_loss: 0.3911
Epoch 25/40
1875/1875 - 4s 2ms/step - accuracy: 0.8698 - loss: 0.3785 - val_accuracy: 0.8664 - val_loss: 0.3898
Epoch 26/40
1875/1875 - 5s 2ms/step - accuracy: 0.8701 - loss: 0.3761 - val_accuracy: 0.8636 - val_loss: 0.3892
Epoch 27/40
1875/1875 - 6s 2ms/step - accuracy: 0.8716 - loss: 0.3753 - val_accuracy: 0.8663 - val_loss: 0.3880
Epoch 28/40
1875/1875 - 4s 2ms/step - accuracy: 0.8692 - loss: 0.3777 - val_accuracy: 0.8667 - val_loss: 0.3828
Epoch 29/40
1875/1875 - 5s 2ms/step - accuracy: 0.8728 - loss: 0.3689 - val_accuracy: 0.8638 - val_loss: 0.3905
Epoch 30/40
1875/1875 - 5s 3ms/step - accuracy: 0.8684 - loss: 0.3775 - val_accuracy: 0.8669 - val_loss: 0.3836
Epoch 31/40
1875/1875 - 4s 2ms/step - accuracy: 0.8712 - loss: 0.3674 - val_accuracy: 0.8696 - val_loss: 0.3812
Epoch 32/40
1875/1875 - 5s 2ms/step - accuracy: 0.8730 - loss: 0.3737 - val_accuracy: 0.8687 - val_loss: 0.3808
Epoch 33/40
1875/1875 - 5s 3ms/step - accuracy: 0.8750 - loss: 0.3666 - val_accuracy: 0.8670 - val_loss: 0.3783
Epoch 34/40
1875/1875 - 3s 2ms/step - accuracy: 0.8756 - loss: 0.3663 - val_accuracy: 0.8679 - val_loss: 0.3792
Epoch 35/40
1875/1875 - 4s 2ms/step - accuracy: 0.8764 - loss: 0.3579 - val_accuracy: 0.8681 - val_loss: 0.3808
Epoch 36/40
1875/1875 - 4s 2ms/step - accuracy: 0.8789 - loss: 0.3584 - val_accuracy: 0.8679 - val_loss: 0.3800
Epoch 37/40
1875/1875 - 4s 2ms/step - accuracy: 0.8798 - loss: 0.3553 - val_accuracy: 0.8686 - val_loss: 0.3769
Epoch 38/40
1875/1875 - 3s 2ms/step - accuracy: 0.8781 - loss: 0.3583 - val_accuracy: 0.8718 - val_loss: 0.3746
Epoch 39/40
1875/1875 - 5s 2ms/step - accuracy: 0.8788 - loss: 0.3548 - val_accuracy: 0.8707 - val_loss: 0.3758
Epoch 40/40
1875/1875 - 4s 2ms/step - accuracy: 0.8799 - loss: 0.3537 - val_accuracy: 0.8727 - val_loss: 0.3725

```

شکل ۱۵- نتیجه اجرای مدل دوم

همانطور که از نتایج مشخص است مقدار loss به شکل قابل قبولی در حال کاهش است. در مدل اول تقریباً در تمامی مراحل شاهد کاهش مقدار loss هستیم که نشان دهنده معماری درست و دیتای informative ما است. در مدل دوم در بخش هایی شاهد بالا و پایین شدن هایی برای loss هستیم که در range محدود و قابل قبولی است.

نکته بسیار حائز اهمیت دیگری که در نتایج مشاهده می شود نزدیک بودن accuracy برای داده های اولیه و داده های validation در مدل دوم است. (نزدیک تر بودن val_accuracy و accuracy در مدل دوم نسبت به اول) این موضوع نشان می دهد علی رغم اینکه دقت در مدل دوم بنظر کمتر از مدل اول با نورون های بیشتر است اما این دقت ناشی از overfit شدن داده ها است. به عبارتی این موضوع که تفاوت دقت بین دیتای train و validation وجود دارد این موضوع را به ما خاطر نشان می کند. پس مدل دوم به دلیل وجود لایه dropout علی رغم دقت کمتر اما قابلیت تعمیم بهتری نسبت به مدل اول دارد.س

۱-۲-۱- هیستوگرام وزن ها

حال با ترسیم هیستوگرام وزن ها، وزن اختصاص داده شده به نورون ها در هر دو مدل را مورد تحلیل و بررسی قرار می دهیم. برای این کار یکتابع نوشته شده است که دو مدل را به عنوان ورودی دریافت کرده و نمودار توزیع وزن ها برای لایه ورودی و مخفی را ترسیم می کند.

```

def weight_distribution_plot(model1, model2):
    # Get weights from Model 1
    weights_model1_input = model1.layers[0].get_weights()[0].flatten() # Input layer weights
    weights_model1_hidden = model1.layers[1].get_weights()[0].flatten() # Hidden layer weights
    # Get weights from Model 2
    weights_model2_input = model2.layers[0].get_weights()[0].flatten() # Input layer weights
    weights_model2_hidden = model2.layers[2].get_weights()[0].flatten() # Hidden layer weights
    plt.figure(figsize=(12, 8))
    # Model 1 Input Layer Weights
    plt.subplot(2, 2, 1)
    plt.hist(weights_model1_input, bins=30, color='blue', alpha=0.7)
    plt.title('Model 1 - Input Layer Weights')
    # Model 1 Hidden Layer Weights
    plt.subplot(2, 2, 2)
    plt.hist(weights_model1_hidden, bins=30, color='blue', alpha=0.7)
    plt.title('Model 1 - Hidden Layer Weights')
    # Model 2 Input Layer Weights
    plt.subplot(2, 2, 3)
    plt.hist(weights_model2_input, bins=30, color='green', alpha=0.7)
    plt.title('Model 2 - Input Layer Weights')
    # Model 2 Hidden Layer Weights
    plt.subplot(2, 2, 4)
    plt.hist(weights_model2_hidden, bins=30, color='green', alpha=0.7)
    plt.title('Model 2 - Hidden Layer Weights')
    plt.tight_layout()
    plt.show()

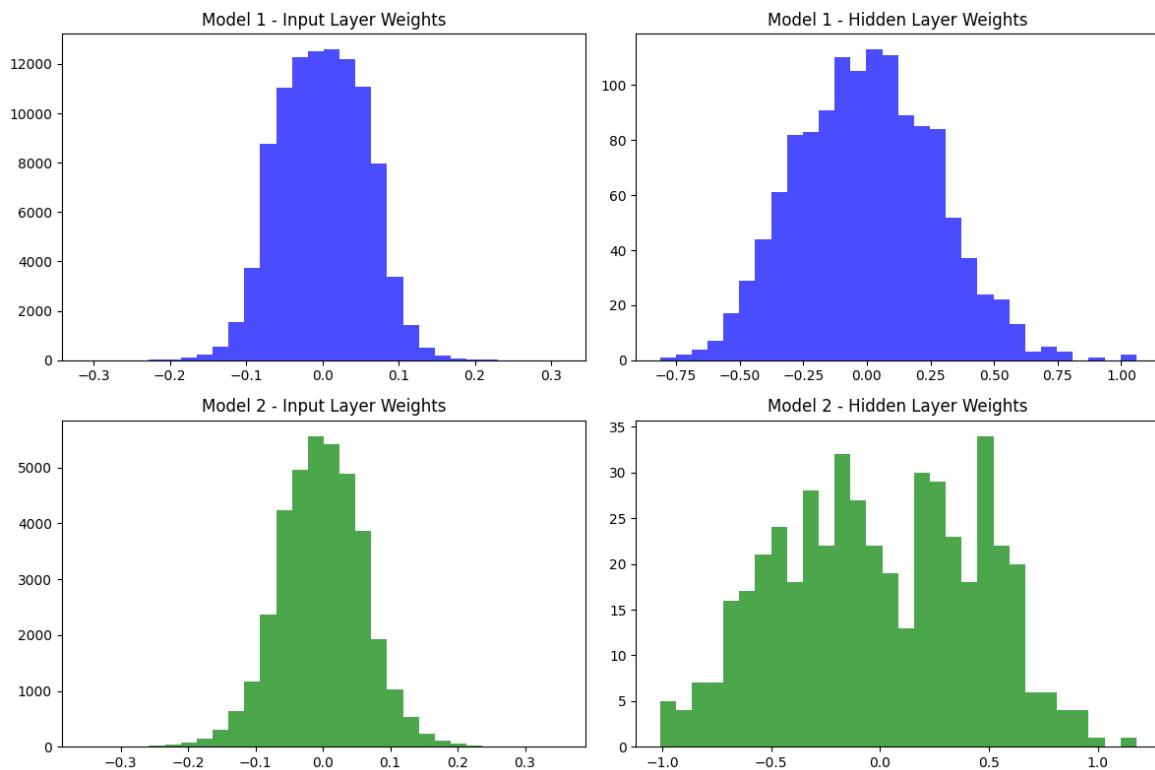
```

شکل ۱۶- تابع نوشته شده برای توزیع وزن‌ها

در مدل اول لایه با ایندکس صفر مربوط به لایه input و ایندکس یک مربوط به لایه مخفی است اما در مدل دوم به دلیل وجود لایه dropout لایه با ایندکس دوم مربوط به لایه مخفی خواهد بود. در ادامه دو مدل اول را به عنوان ورودی به تابع ارسال و خروجی را مورد تحلیل قرار داده ایم.

```
weight_distribution_plot(model1, model2)
```

شکل ۱۷- فراخواندن تابع و ارسال دو مدل - بدون **optimizer**



شکل ۱۸- هیستوگرام وزن‌های دو مدل در لایه ورودی و مخفی

در محور قائم تعداد نورون‌ها یا فراوانی را داریم و در محور افقی وزن اختصاص داده شده. در ابتدا با توجه به وجود نورون‌های بیشتر در مدل اول محور قائم عدد بزرگتری را خواهد داشت. اما در لایه ورودی مشاهده می‌کنیم هر دو مدل توزیع نسبتاً مشابه و نرمال دارند اما هیستوگرام مدل اول اندکی variance بیشتری را دارد یا به اصطلاح چاق‌تر است. این بدان معناست که تعداد نورون‌های بیشتری عددهای بزرگتر گرفته اند. این با پیش‌بینی ما مطابقت دارد. به دو دلیل: ۱) تفاوت در ساختار شبکه دو مدل و تعداد نورون ۲) که می‌تواند دلیل اصلی باشد، در شرایط عدم وجود dropout احتمال وابسته بودن به نورون‌ها بیشتر است و وزن بیشتری به آنها داده خواهد شد. البته لازم به ذکر است که لایه dropout به صورت مستقیم روی لایه backpropagation چرا که این لایه بعد از dense layer تعریف شده اما با توجه به الگوریتم ما شاهد اثر این کاهش وزن‌ها به صورت غیر مستقیم در لایه ورودی هستیم.

در لایه مخفی اما شاهد یک توزیع نسبتاً نرمال برای مدل اول و یک توزیع نسبتاً یکنواخت تر (uniform) در مدل دوم هستیم. به این معنا که در لایه مخفی مدل دوم یکنواختی بیشتری دارد. دلیل این موضوع می‌تواند با وجود Dropout توجیه شود. با حضور این لایه در واقع به صورت رندوم تعدادی از نورون‌ها

خواهند شد و باقی نورون‌ها می‌بایست وزن مناسبی دریافت کنند. از طرفی شاهد کاهش وزن Deactive به صورت خیلی مشخصی در این لایه نیستیم. دلیل این امر می‌تواند به optimizer هم مرتبط باشد.

۲-۲-۱- تاثیر بهینه‌ساز

برای بررسی اثر بهینه‌ساز هر دو بهینه‌ساز گفته شده در صورت سوال را برای هر دو مدل به کار گرفتیم و عملکرد آنها را در ۴۰ epoch مورد بررسی قرار دادیم. در ادامه کد هر مدل و نتایج آن را مشاهده خواهید کرد.

ابتدا از Adam برای هر دو مدل استفاده شده است.

```
# Define Model 1
model1_adam = models.Sequential()
model1_adam.add(layers.Input(shape=(28*28,)))

model1_adam.add(layers.Dense(128, activation='relu'))
model1_adam.add(layers.Dense(10, activation='softmax'))

# Compile Model 1
model1_adam.compile(optimizer='adam',
                     loss='sparse_categorical_crossentropy',
                     metrics=['accuracy'])

# Train Model 1 for 40 epochs
history1 = model1_adam.fit(x_train_flat, y_train, epochs=40, validation_data=(x_test_flat, y_test))
```

شکل ۱۹- مدل اولیه با استفاده از بهینه‌ساز Adam

```

Epoch 24/40
1875/1875 10s 3ms/step - accuracy: 0.9395 - loss: 0.1590 - val_accuracy: 0.8896 - val_loss: 0.3536
Epoch 25/40
1875/1875 8s 4ms/step - accuracy: 0.9398 - loss: 0.1558 - val_accuracy: 0.8882 - val_loss: 0.3730
Epoch 26/40
1875/1875 9s 3ms/step - accuracy: 0.9412 - loss: 0.1526 - val_accuracy: 0.8915 - val_loss: 0.3877
Epoch 27/40
1875/1875 10s 3ms/step - accuracy: 0.9427 - loss: 0.1489 - val_accuracy: 0.8880 - val_loss: 0.4057
Epoch 28/40
1875/1875 9s 5ms/step - accuracy: 0.9452 - loss: 0.1439 - val_accuracy: 0.8905 - val_loss: 0.3734
Epoch 29/40
1875/1875 7s 3ms/step - accuracy: 0.9465 - loss: 0.1439 - val_accuracy: 0.8892 - val_loss: 0.3929
Epoch 30/40
1875/1875 8s 4ms/step - accuracy: 0.9460 - loss: 0.1384 - val_accuracy: 0.8865 - val_loss: 0.3839
Epoch 31/40
1875/1875 6s 3ms/step - accuracy: 0.9511 - loss: 0.1312 - val_accuracy: 0.8883 - val_loss: 0.3920
Epoch 32/40
1875/1875 8s 4ms/step - accuracy: 0.9491 - loss: 0.1325 - val_accuracy: 0.8860 - val_loss: 0.4210
Epoch 33/40
1875/1875 11s 5ms/step - accuracy: 0.9528 - loss: 0.1275 - val_accuracy: 0.8876 - val_loss: 0.3992
Epoch 34/40
1875/1875 8s 3ms/step - accuracy: 0.9539 - loss: 0.1256 - val_accuracy: 0.8783 - val_loss: 0.4284
Epoch 35/40
1875/1875 10s 3ms/step - accuracy: 0.9532 - loss: 0.1237 - val_accuracy: 0.8892 - val_loss: 0.4176
Epoch 36/40
1875/1875 11s 4ms/step - accuracy: 0.9553 - loss: 0.1219 - val_accuracy: 0.8901 - val_loss: 0.4168
Epoch 37/40
1875/1875 8s 4ms/step - accuracy: 0.9551 - loss: 0.1214 - val_accuracy: 0.8894 - val_loss: 0.4235
Epoch 38/40
1875/1875 6s 3ms/step - accuracy: 0.9552 - loss: 0.1157 - val_accuracy: 0.8916 - val_loss: 0.4330
Epoch 39/40
1875/1875 6s 3ms/step - accuracy: 0.9566 - loss: 0.1161 - val_accuracy: 0.8869 - val_loss: 0.4379
Epoch 40/40
1875/1875 6s 3ms/step - accuracy: 0.9563 - loss: 0.1156 - val_accuracy: 0.8839 - val_loss: 0.4738

```

شکل ۲۰ - نتیجه اجرای مدل اول با بهینه‌ساز Adam

با استفاده از این بهینه‌ساز شاهد افزایش accuracy در گام پایانی هستیم نسبت به مدل یک بدون این بهینه‌ساز. در حالت اول دقت حدود ۹۰.۴۹ درصد اما اینجا دقت ۹۵.۶۳ درصدی را شاهد هستیم. س

```

# Define Model 2
model2_adam = models.Sequential()
model2_adam.add(layers.Input(shape=(28*28,)))

model2_adam.add(layers.Dense(48, activation='relu',
                            kernel_regularizer=regularizers.l2(0.0001)))
model2_adam.add(layers.Dropout(0.2))
model2_adam.add(layers.Dense(10, activation='softmax'))

# Compile Model 2
model2_adam.compile(optimizer='adam',
                     loss='sparse_categorical_crossentropy',
                     metrics=['accuracy'])

# Train Model 2 for 40 epochs
history2 = model2_adam.fit(x_train_flat, y_train, epochs=40, validation_data=(x_test_flat, y_test))

```

شکل ۲۱ - مدل ثانویه با استفاده از بهینه‌ساز Adam

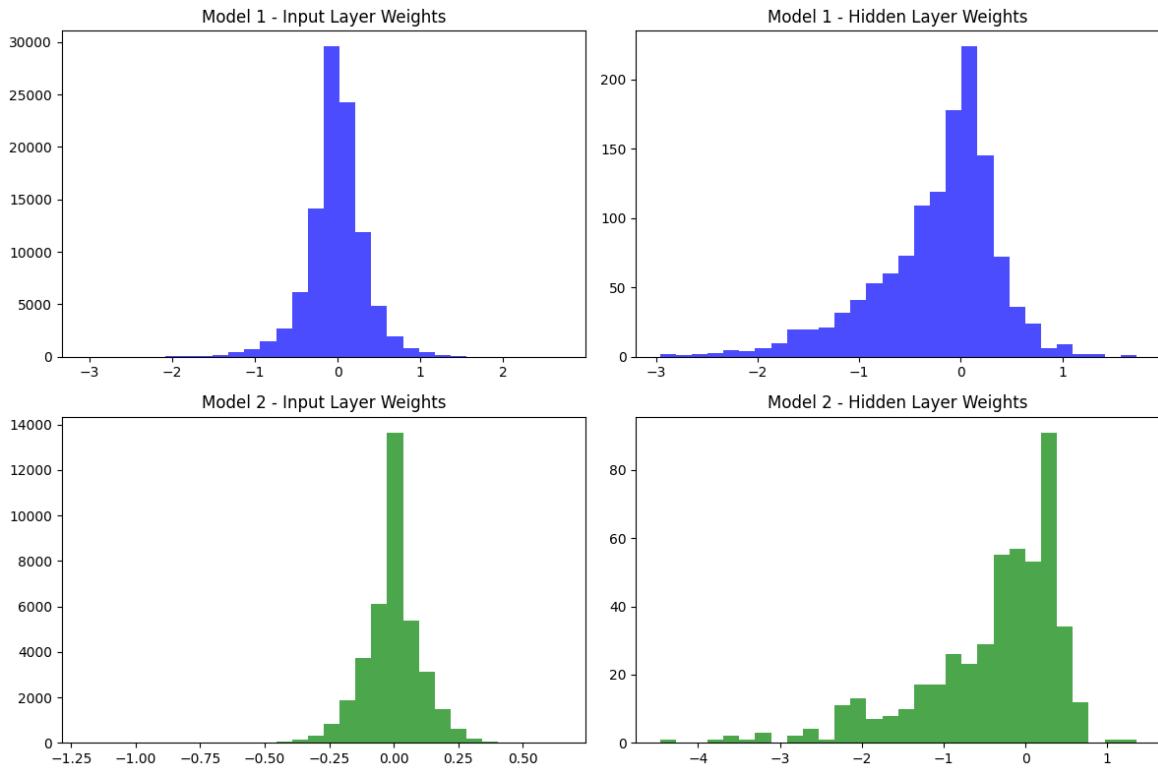
```

1875/1875 ━━━━━━━━━━ 4s 2ms/step - accuracy: 0.8749 - loss: 0.3738 - val_accuracy: 0.8702 - val_loss: 0.4046
Epoch 24/40
1875/1875 ━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8774 - loss: 0.3631 - val_accuracy: 0.8628 - val_loss: 0.4176
Epoch 25/40
1875/1875 ━━━━━━━━━━ 4s 2ms/step - accuracy: 0.8777 - loss: 0.3672 - val_accuracy: 0.8710 - val_loss: 0.4059
Epoch 26/40
1875/1875 ━━━━━━━━━━ 7s 3ms/step - accuracy: 0.8778 - loss: 0.3686 - val_accuracy: 0.8635 - val_loss: 0.4259
Epoch 27/40
1875/1875 ━━━━━━━━━━ 9s 2ms/step - accuracy: 0.8761 - loss: 0.3702 - val_accuracy: 0.8698 - val_loss: 0.4062
Epoch 28/40
1875/1875 ━━━━━━━━━━ 7s 3ms/step - accuracy: 0.8813 - loss: 0.3651 - val_accuracy: 0.8729 - val_loss: 0.4064
Epoch 29/40
1875/1875 ━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8833 - loss: 0.3608 - val_accuracy: 0.8729 - val_loss: 0.4017
Epoch 30/40
1875/1875 ━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8806 - loss: 0.3608 - val_accuracy: 0.8665 - val_loss: 0.4067
Epoch 31/40
1875/1875 ━━━━━━━━━━ 11s 3ms/step - accuracy: 0.8786 - loss: 0.3644 - val_accuracy: 0.8665 - val_loss: 0.4081
Epoch 32/40
1875/1875 ━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8802 - loss: 0.3637 - val_accuracy: 0.8607 - val_loss: 0.4271
Epoch 33/40
1875/1875 ━━━━━━━━━━ 4s 2ms/step - accuracy: 0.8813 - loss: 0.3656 - val_accuracy: 0.8704 - val_loss: 0.4094
Epoch 34/40
1875/1875 ━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8830 - loss: 0.3583 - val_accuracy: 0.8688 - val_loss: 0.4142
Epoch 35/40
1875/1875 ━━━━━━━━━━ 4s 2ms/step - accuracy: 0.8809 - loss: 0.3627 - val_accuracy: 0.8699 - val_loss: 0.4043
Epoch 36/40
1875/1875 ━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8820 - loss: 0.3594 - val_accuracy: 0.8697 - val_loss: 0.4096
Epoch 37/40
1875/1875 ━━━━━━━━━━ 7s 3ms/step - accuracy: 0.8833 - loss: 0.3608 - val_accuracy: 0.8704 - val_loss: 0.4197
Epoch 38/40
1875/1875 ━━━━━━━━━━ 9s 2ms/step - accuracy: 0.8817 - loss: 0.3664 - val_accuracy: 0.8755 - val_loss: 0.4000
Epoch 39/40
1875/1875 ━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8820 - loss: 0.3595 - val_accuracy: 0.8742 - val_loss: 0.3998
Epoch 40/40
1875/1875 ━━━━━━━━━━ 10s 3ms/step - accuracy: 0.8855 - loss: 0.3528 - val_accuracy: 0.8695 - val_loss: 0.4123

```

شکل ۲۲- نتیجه اجرای مدل دوم با بهینه‌ساز Adam

در مدل دوم هم شاهد افزایش اندک دقت هستیم. مقدار افزایش در این حالت کمتر از مدل اول است.



شکل ۲۳ - توزیع وزن‌ها در مدل یک و دو با حضور بهینه‌ساز **Adam**

با استفاده از تابعی که برای ترسیم توزیع وزن داشتیم، این بار مدل یک و دو را با حضور Adam optimizer مورد بررسی قرار دادیم. نتایج به شکل ملموسی نشان‌دهنده کاهش وزن نورون‌ها در مدل دوم و با حضور dropout است. به عبارت دیگر وجود بهینه‌ساز به توزیع بهتر وزن‌ها کمک کرده است.

حال از rmsprop برای هر دو مدل استفاده شده است.

```

# Define Model 1 with rmsprop optimizer
model1_rmsprop = models.Sequential()
model1_rmsprop.add(layers.Input(shape=(28*28,)))

model1_rmsprop.add(layers.Dense(128, activation='relu'))
model1_rmsprop.add(layers.Dense(10, activation='softmax'))

# Compile Model 1
model1_rmsprop.compile(optimizer='rmsprop',
                       loss='sparse_categorical_crossentropy',
                       metrics=['accuracy'])

# Train Model 1 for 10 epochs
history1 = model1_rmsprop.fit(x_train_flat, y_train, epochs=40, validation_data=(x_test_flat, y_test))

```

شکل ۲۴- مدل اولیه با استفاده از بهینه‌ساز **rmsprop**

```

Epoch 1/40
1875/1875 7s 4ms/step - accuracy: 0.7789 - loss: 0.6396 - val_accuracy: 0.8262 - val_loss: 0.4613
Epoch 2/40
1875/1875 9s 3ms/step - accuracy: 0.8624 - loss: 0.3820 - val_accuracy: 0.8657 - val_loss: 0.3803
Epoch 3/40
1875/1875 5s 3ms/step - accuracy: 0.8737 - loss: 0.3498 - val_accuracy: 0.8639 - val_loss: 0.3992
Epoch 4/40
1875/1875 5s 3ms/step - accuracy: 0.8824 - loss: 0.3268 - val_accuracy: 0.8581 - val_loss: 0.4044
Epoch 5/40
1875/1875 7s 3ms/step - accuracy: 0.8898 - loss: 0.3109 - val_accuracy: 0.8766 - val_loss: 0.3653
Epoch 6/40
1875/1875 10s 3ms/step - accuracy: 0.8943 - loss: 0.3020 - val_accuracy: 0.8696 - val_loss: 0.3655
Epoch 7/40
1875/1875 5s 3ms/step - accuracy: 0.8973 - loss: 0.2913 - val_accuracy: 0.8727 - val_loss: 0.3831
Epoch 8/40
1875/1875 5s 3ms/step - accuracy: 0.8960 - loss: 0.2938 - val_accuracy: 0.8684 - val_loss: 0.4484
Epoch 9/40
1875/1875 6s 3ms/step - accuracy: 0.9023 - loss: 0.2784 - val_accuracy: 0.8750 - val_loss: 0.3986
Epoch 10/40
1875/1875 5s 3ms/step - accuracy: 0.9034 - loss: 0.2764 - val_accuracy: 0.8645 - val_loss: 0.4545
Epoch 11/40
1875/1875 6s 3ms/step - accuracy: 0.9070 - loss: 0.2695 - val_accuracy: 0.8660 - val_loss: 0.4253
Epoch 12/40
1875/1875 9s 3ms/step - accuracy: 0.9098 - loss: 0.2571 - val_accuracy: 0.8751 - val_loss: 0.4347
Epoch 13/40

```

شکل ۲۵- نتیجه اجرای مدل اول با بهینه‌ساز **rmsprop** – مراحل اولیه

```

Epoch 24/40
1875/1875 5s 3ms/step - accuracy: 0.9296 - loss: 0.2150 - val_accuracy: 0.8761 - val_loss: 0.5237
Epoch 25/40
1875/1875 6s 3ms/step - accuracy: 0.9295 - loss: 0.2087 - val_accuracy: 0.8772 - val_loss: 0.5404
Epoch 26/40
1875/1875 11s 4ms/step - accuracy: 0.9318 - loss: 0.2065 - val_accuracy: 0.8736 - val_loss: 0.5422
Epoch 27/40
1875/1875 5s 2ms/step - accuracy: 0.9313 - loss: 0.2064 - val_accuracy: 0.8767 - val_loss: 0.5157
Epoch 28/40
1875/1875 5s 3ms/step - accuracy: 0.9314 - loss: 0.2053 - val_accuracy: 0.8773 - val_loss: 0.5260
Epoch 29/40
1875/1875 6s 3ms/step - accuracy: 0.9338 - loss: 0.2046 - val_accuracy: 0.8736 - val_loss: 0.5533
Epoch 30/40
1875/1875 11s 3ms/step - accuracy: 0.9356 - loss: 0.1950 - val_accuracy: 0.8681 - val_loss: 0.6106
Epoch 31/40
1875/1875 9s 3ms/step - accuracy: 0.9338 - loss: 0.2003 - val_accuracy: 0.8729 - val_loss: 0.5354
Epoch 32/40
1875/1875 6s 3ms/step - accuracy: 0.9382 - loss: 0.1901 - val_accuracy: 0.8703 - val_loss: 0.5930
Epoch 33/40
1875/1875 5s 3ms/step - accuracy: 0.9381 - loss: 0.1878 - val_accuracy: 0.8726 - val_loss: 0.5685
Epoch 34/40
1875/1875 6s 3ms/step - accuracy: 0.9389 - loss: 0.1895 - val_accuracy: 0.8709 - val_loss: 0.6354
Epoch 35/40
1875/1875 8s 2ms/step - accuracy: 0.9395 - loss: 0.1910 - val_accuracy: 0.8779 - val_loss: 0.5795
Epoch 36/40
1875/1875 6s 3ms/step - accuracy: 0.9428 - loss: 0.1774 - val_accuracy: 0.8751 - val_loss: 0.5863
Epoch 37/40
1875/1875 5s 3ms/step - accuracy: 0.9422 - loss: 0.1781 - val_accuracy: 0.8785 - val_loss: 0.5728
Epoch 38/40
1875/1875 7s 4ms/step - accuracy: 0.9439 - loss: 0.1770 - val_accuracy: 0.8794 - val_loss: 0.5845
Epoch 39/40
1875/1875 5s 3ms/step - accuracy: 0.9439 - loss: 0.1714 - val_accuracy: 0.8823 - val_loss: 0.6013
Epoch 40/40
1875/1875 11s 3ms/step - accuracy: 0.9448 - loss: 0.1742 - val_accuracy: 0.8713 - val_loss: 0.7175

```

شکل ۲۶ - نتیجه اجرای مدل اول با بهینهساز **rmsprop** – مراحل پایانی

در نتیجه استفاده از این روش هم شاهد افزایش مقدار دقت به ۹۴.۴۸ درصد هستیم که نشان دهنده عملکرد مثبت استفاده از بهینهساز است.

```

# Define Model 2
model2_adam = models.Sequential()
model2_adam.add(layers.Input(shape=(28*28,)))

model2_adam.add(layers.Dense(48, activation='relu',
                           kernel_regularizer=regularizers.l2(0.0001)))
model2_adam.add(layers.Dropout(0.2))
model2_adam.add(layers.Dense(10, activation='softmax'))

# Compile Model 2
model2_adam.compile(optimizer='adam',
                     loss='sparse_categorical_crossentropy',
                     metrics=['accuracy'])

# Train Model 2 for 40 epochs
history2 = model2_adam.fit(x_train_flat, y_train, epochs=40, validation_data=(x_test_flat, y_test))

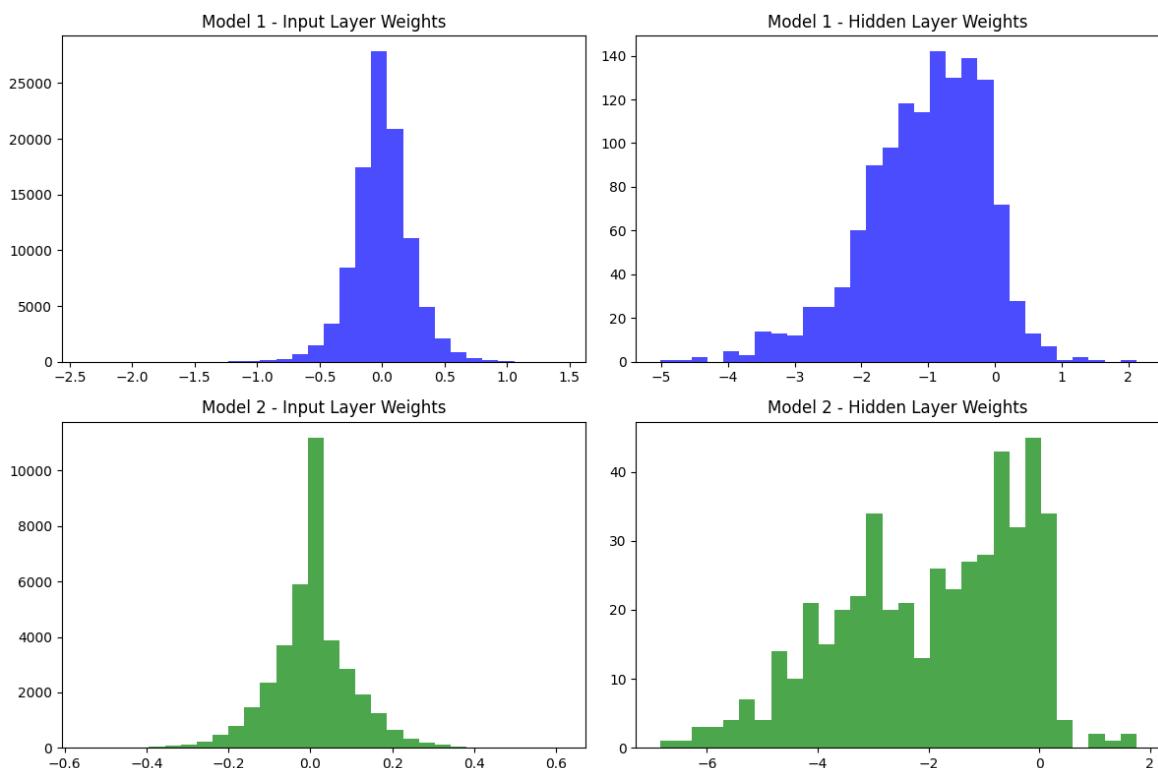
```

شکل ۲۷ - مدل ثانویه با استفاده از بهینهساز **rmsprop**

```
1875/1875 ━━━━━━━━ 5s 2ms/step - accuracy: 0.8679 - loss: 0.4105 - val_accuracy: 0.8624 - val_loss: 0.4631
Epoch 25/40
1875/1875 ━━━━━━━━ 5s 2ms/step - accuracy: 0.8674 - loss: 0.4093 - val_accuracy: 0.8624 - val_loss: 0.4589
Epoch 26/40
1875/1875 ━━━━━━━━ 5s 2ms/step - accuracy: 0.8686 - loss: 0.4147 - val_accuracy: 0.8517 - val_loss: 0.4857
Epoch 27/40
1875/1875 ━━━━━━ 6s 2ms/step - accuracy: 0.8686 - loss: 0.4174 - val_accuracy: 0.8627 - val_loss: 0.4694
Epoch 28/40
1875/1875 ━━━━━━ 4s 2ms/step - accuracy: 0.8718 - loss: 0.4088 - val_accuracy: 0.8611 - val_loss: 0.4796
Epoch 29/40
1875/1875 ━━━━━━ 4s 2ms/step - accuracy: 0.8683 - loss: 0.4176 - val_accuracy: 0.8660 - val_loss: 0.4718
Epoch 30/40
1875/1875 ━━━━━━ 6s 2ms/step - accuracy: 0.8715 - loss: 0.4129 - val_accuracy: 0.8638 - val_loss: 0.4919
Epoch 31/40
1875/1875 ━━━━━━ 4s 2ms/step - accuracy: 0.8698 - loss: 0.4145 - val_accuracy: 0.8665 - val_loss: 0.4650
Epoch 32/40
1875/1875 ━━━━━━ 5s 2ms/step - accuracy: 0.8673 - loss: 0.4184 - val_accuracy: 0.8575 - val_loss: 0.4688
Epoch 33/40
1875/1875 ━━━━━━ 5s 2ms/step - accuracy: 0.8691 - loss: 0.4233 - val_accuracy: 0.8556 - val_loss: 0.5078
Epoch 34/40
1875/1875 ━━━━━━ 4s 2ms/step - accuracy: 0.8679 - loss: 0.4231 - val_accuracy: 0.8584 - val_loss: 0.4885
Epoch 35/40
1875/1875 ━━━━━━ 5s 2ms/step - accuracy: 0.8705 - loss: 0.4223 - val_accuracy: 0.8692 - val_loss: 0.4771
Epoch 36/40
1875/1875 ━━━━━━ 5s 3ms/step - accuracy: 0.8710 - loss: 0.4183 - val_accuracy: 0.8637 - val_loss: 0.4878
Epoch 37/40
1875/1875 ━━━━━━ 4s 2ms/step - accuracy: 0.8701 - loss: 0.4218 - val_accuracy: 0.8620 - val_loss: 0.4641
Epoch 38/40
1875/1875 ━━━━━━ 5s 2ms/step - accuracy: 0.8704 - loss: 0.4138 - val_accuracy: 0.8667 - val_loss: 0.4727
Epoch 39/40
1875/1875 ━━━━━━ 7s 3ms/step - accuracy: 0.8718 - loss: 0.4113 - val_accuracy: 0.8603 - val_loss: 0.4954
Epoch 40/40
1875/1875 ━━━━━━ 4s 2ms/step - accuracy: 0.8690 - loss: 0.4250 - val_accuracy: 0.8675 - val_loss: 0.4792
```

شکل ۲۸ - نتیجه اجرای مدل دوم با بهینه‌ساز **rmsprop**

در این حالت اما شاهد تفاوت چشم گیری در دقیقیت مدل نیستیم.



RMSProp شکل ۲۹ - توزیع وزن‌ها در مدل یک و دو با حضور بهینه‌ساز

با استفاده از تابعی که برای ترسیم توزیع وزن داشتیم، این بار مدل یک و دو را با حضور RMSProp optimizer مورد بررسی قرار دادیم. نتایج به شکل ملموسی نشان‌دهنده کاهش وزن نورون‌ها در مدل دوم و با حضور dropout است. به عبارت دیگر وجود بهینه‌ساز به توزیع بهتر وزن‌ها کمک کرده است. در لایه دوم اما اندکی مشابه بدون بهینه‌ساز است و یکنواخت است.

در نهایت برای مقایسه شبکه‌ها بدون استفاده از جدول استفاده شده است.

جدول ۱. مقایسه شبکه‌ها بدون استفاده از بهینه‌ساز

دقت validation (گام آخر)	دقت (گام آخر)	
۸۷.۶۳	۹۰.۴۹	شبکه‌ی اول
۸۷.۲۷	۸۷.۹۷	شبکه‌ی دوم

جدول ۲. مقایسه شبکه‌ها با استفاده از **Adam**

دقت validation (گام آخر)	دقت (گام آخر)	شبکه‌ی اول	شبکه‌ی دوم
۸۸.۳۹	۹۵.۶۳		
۸۶.۹۵	۸۸.۵۵		

س. ۳. مقایسه شبکه‌ها با استفاده از **RMSProp**

دقت validation (گام آخر)	دقت (گام آخر)	شبکه‌ی اول	شبکه‌ی دوم
۸۸.۱۳	۹۴.۴۸		
۸۶.۷۵	۸۶.۹۰		

در نهایت با توجه به جداول ارائه شده بنظر می‌رسد استفاده از بهینه‌ساز نقش بسزایی در افزایش دقت مدل دارد. از طرفی بررسی نتایج نشان می‌دهد با استفاده از بهینه‌سازها از همان ابتدا خیلی سریع تر به optimum نزدیک بوده و نزدیک تر می‌شویم ولی بدون استفاده از آنها در ابتدا مقدار leaning rate به گونه‌ای است که احتمال دور بودن از جواب نهایی در گام‌های اولیه زیاد است و در نهایت کم کم به جواب خواهیم رسید. به صورت کلی با کمک این بهینه‌سازها سریع تر و دقیق تر به جواب رسیدیم. البته این موضوع در مدل اول ملموس تر بوده و در مدل دوم کمتر شاهد افزایش و تغییر در دقت هستیم.

۱-۳. الگوریتم بازگشت به عقب

روند این الگوریتم به این صورت است که ابتدا دیتای ورودی وارد می‌شود، و به صورت forward pass جلو می‌رویم. حال با توجه به مقادیر در نظر گرفته شده برای وزن و bias مقدار loss را محاسبه می‌کنیم تا میزان خطای شبکه را نسبت به واقعیت محاسبه کنیم. در گام بعد با استفاده از روش gradian و به صورت عقب گرد (backpropagation) مقادیر وزن را بروز می‌کنیم.

حال در این بخش می‌بایست سه بهینه‌ساز متفاوت را برای آموزش شبکه به کار بیندیم و نتایج را باهم مقایسه کنیم. در ابتدای کار به توضیح هر کدام از بهینه‌سازهای بکار رفته و نحوه عملکرد آنها می‌پردازیم و سپس نحوه پیاده سازی و نتایج را گزارش کرده و آنها را باهم مقایسه می‌کنیم.

۱-۳-۱ - پیاده‌سازی سه الگوریتم بهینه‌ساز

در این روش learning rate در هر مرحله تغییر خواهد کرد. روند کار به این صورت است که مقدار learning rate را بر اساس میانگین گرادیان‌های قبلی محاسبه می‌کند. عملاً قابلیت تطبیقی این بهینه‌ساز نقطه حائز اهمیت آن است. فرمول ریاضیاتی مورد استفاده در این روش به صورت زیر است. در این روش یک ضریب افت هم تعریف می‌شود که عموماً برابر ۰.۹ در نظر گرفته می‌شود.

- **Formula:**

$$v_t = \beta v_{t-1} + (1 - \beta) g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t} + \epsilon} g_t$$

where g_t is the gradient, v_t is the moving average of squared gradients, and η is the learning rate.

[۱] RMSProp - فرمول محاسباتی در روش

Adam: مزایای دو توسعه دیگر نزول گرادیان تصادفی یعنی آداغراد و آرام اس پر اپ رو ترکیب می‌کنه. این روش نرخ‌های یادگیری انطباقی برای هر پارامتر رو از برآوردهایی از لحظه‌های اول و دوم محاسبه گرادیان‌ها و بیزگی‌های کلیدی: مومنتوم: از میانگین نمایی گذشته گرادیان‌ها (first momentum) استفاده می‌کنه. نرخ یادگیری انطباقی: میانگین متحرک گرادیان‌های مربع شده (second momentum) رو نگه می‌داره.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

شکل ۳۱ - فرمول محاسباتی در روش [۲] Adam

این روش همان روش آدام به علاوه استفاده از Nesterov momentum است. به عبارتی همان الگوریتم اولیه آدام استفاده شده اما از روش Nesterov استفاده شده تا به شکل سریع تری به همگرایی برسیم.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$g' = g_t + (\beta_1 m_{t-1})$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t} + \epsilon} (\beta_1 m_t + g')$$

شکل ۳۲ - فرمول محاسباتی روش [۳] Nadam

لازم به ذکر است چگونگی عملکرد ریاضیاتی بهینه‌سازها در برنامه ارائه شده درس نبوده به همین جهت از توضیح بیشتر درباره چگونگی عملکرد ریاضیاتی و اثبات آنها صرف نظر می‌کنیم.

در ادامه به چگونگی پیاده‌سازی آنها می‌پردازیم. در این بخش مقداری کد بهینه‌تر شده که در ادامه به توضیح هر بخش به صورت مجزا پرداخته خواهد شد.

ابتدا کتابخانه‌های مورد نیاز را استخراج کرده ایم.

```
import tensorflow as tf
from tensorflow.keras import layers, models, regularizers, optimizers
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix
!pip install bayesian-optimization
from bayes_opt import BayesianOptimization
import pandas as pd
```

شکل ۳۳ - **import** کردن کتابخانه‌های مورد نیاز

در گام بعد استخراج دیتاست توسط یک تابع انجام شده است. تا پیش از این به صورت مستقیم این کار صورت می‌گرفت. نکته دیگر که در این تابع برای استخراج دیتا تغییر کرده استفاده از متدهای `to_categorical` برای تبدیل خروجی ما از حالت یک عدد بین صفر تا ۹ به حالت one hot است. به عبارت دیگر اگر به ازای مقادیر ورودی کلاس ۹ به عنوان خروجی گزارش شده باشد به جای گزارش عدد ۹، آن را به فرم `[0,0,0,0,0,0,0,0,1]` خواهیم دید. این کار به سبب افزایش کارایی و عملکرد بهتر شبکه انجام شده است و الزاماً به این مورد نبوده است.

```

def load_and_preprocess_data():
    """Load and preprocess Fashion-MNIST dataset"""
    (x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()

    # Normalize and reshape
    x_train = x_train.reshape(-1, 28*28).astype('float32') / 255.0
    x_test = x_test.reshape(-1, 28*28).astype('float32') / 255.0

    # Convert labels to categorical
    y_train = tf.keras.utils.to_categorical(y_train, 10)
    y_test = tf.keras.utils.to_categorical(y_test, 10)

    return (x_train, y_train), (x_test, y_test)

# Load data
(x_train, y_train), (x_test, y_test) = load_and_preprocess_data()

```

شکل - ۳۴ - **load** و **train** و **test** کردن دیتای

```

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
29515/29515          0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26421880/26421880    0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
5148/5148             0s 1us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4422102/4422102      0s 0us/step

```

شکل - ۳۵ - خروجی **load** کردن دیتا

در بخش بعد کلاسی را برای کنترل و ثبت دیتا بعد از هر epoch تعریف می‌کنیم. این کلاس از **Callback** پیش فرض کتابخانه tensorflow است برای می‌کند و عمکرد تابع `on_epoch_end` بدین صورت است که در انتهای هر گام محاسباتی اجرا می‌شود و اطراعات مربوط به مقدار `accuracy` و `loss` را در لیست‌های از پیش تعریف شده ما ذخیره می‌کند. این کار در نمایش بهتر داده‌ها در پایان به ما کمک خواهد کرد و ساختار `clean` و بهتری به کد ما می‌دهد.

```

class TrainingMonitor(tf.keras.callbacks.Callback):
    def __init__(self):
        super(TrainingMonitor, self).__init__()
        self.train_losses = []
        self.val_losses = []
        self.train_accuracies = []
        self.val_accuracies = []

    def on_epoch_end(self, epoch, logs=None):
        self.train_losses.append(logs.get('loss'))
        self.val_losses.append(logs.get('val_loss'))
        self.train_accuracies.append(logs.get('accuracy'))
        self.val_accuracies.append(logs.get('val_accuracy'))

```

شکل ۳۶ - تعریف کلاس کاستوم برای **callback**

در بخش بعد تابعی را تعریف می‌کنیم که بخش اصلی پیاده سازی مدل ماست. در این تابع امکان تعریف هر دو مدل بخش‌های قبل، مدل اول با ۱۲۸ نورون و بدون dropout و مدل دوم با ۴۸ نورون و یک لایه dropout مهیا است. مقادیر ورودی به گونه‌ای تنظیم شده اند که داخل این تابع بسته به نوع مدل و البته optimizer انتخابی، شبکه ایجاد و نهایتا compile و fit شده است. در بخش Create optimizer به وضوح نحوه پیاده‌سازی مشخص است. ابتدا انواع optimizer مورد نظر را استخراج کردیم، سپس در این مرحله آنها را فراخوانی کرده و مقدار اولیه learning rate هم به آنها داده‌ایم. در نهایت در هنگام compile کردن مدل آن را به عنوان optimizer در آرگومان متدهای قرار داده‌ایم.

```

def create_and_train_model(model_type, optimizer_name, learning_rate, x_train, y_train, x_test, y_test):
    """Create and train a model with specified configuration"""
    if model_type == 1:
        model = models.Sequential([
            layers.Input(shape=(28*28,)),
            layers.Dense(128, activation='relu'), layers.Dense(10, activation='softmax')
        ])
    else:
        model = models.Sequential([
            layers.Input(shape=(28*28,)),
            layers.Dense(48, activation='relu',
                        kernel_regularizer=regularizers.l2(0.0001)),
            layers.Dropout(0.2), layers.Dense(10, activation='softmax')
        ])
    # Create optimizer
    if optimizer_name == 'Adam':
        opt = optimizers.Adam(learning_rate=learning_rate)
    elif optimizer_name == 'RMSprop':
        opt = optimizers.RMSprop(learning_rate=learning_rate)
    else: # Nadam
        opt = optimizers.Nadam(learning_rate=learning_rate)
    monitor = TrainingMonitor()
    model.compile(
        optimizer=opt,
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )
    history = model.fit(
        x_train, y_train,
        epochs=10, batch_size=32, validation_data=(x_test, y_test), callbacks=[monitor], verbose=1)

    return model, history, monitor

```

شکل ۳۷ - پیاده‌سازی مدل

در انتها برای فیت کردن مدل x و y مدل، آرگومان‌های ابتدایی هستند، سپس ۱۰ گام محاسباتی در نظر گرفته شده و در بخش بعد `batch_size` برابر ۳۲ در نظر گرفته شده است. این کار به این معنا است به جای استفاده از کل دیتا یا به صورت نقطه‌ای یک mini batch به تعداد ۳۲ دیتا در هر گام به صورت تصادفی انتخاب و در محاسبات ما بکار گرفته می‌شود. بر اساس مطالب گفته شده در کلاس این روش در حال حاضر بسیار پرکاربرد است و از این نظر که اندکی هم تفاوت با کد بخش قبل داشته باشد این کار انجام شده است. در گام بعد دیتای `validation` فرستاده شده است و در بخش بعدی از کلاس `NoShit` شده در قسمت قبل به عنوان `callback` استفاده کردیم تا دیتا مورد نظر را به شکل راحت تری ذخیره کنیم. و در انتها `verbose` برابر یک قرار داده شده است. این ترم در این متاد، چگونگی نمایش خروجی در هر epoch را مشخص می‌کند به عنوان مثال اگر عدد صفر قرار بدھیم حالت `silent` پیدا می‌کند و هیچ چیزی نمایش داده نمی‌شود اما عدد یک به معنای نمایش مرحله به مرحله هر epoch و مشخص بودن مقدار عددی `loss` و `accuracy` در هر گام محاسباتی است.

در مرحله بعد یک `configuration` تعریف شده است که مشخص کننده هر مدل و هر بهینه‌ساز است. لازم به ذکر است در صورت سوال اشاره‌ای به طراحی هر دو مدل نشده بود اما به جهت بررسی بهتر هر دو مدل با تمام `optimizer`‌ها اجرا شده و نتایج مورد بررسی قرار گرفته اند.

```

# Define configurations
configs = [
    (1, 'Adam', 0.001),
    (1, 'RMSprop', 0.001),
    (1, 'Nadam', 0.001),
    (2, 'Adam', 0.001),
    (2, 'RMSprop', 0.001),
    (2, 'Nadam', 0.001)
]

# Train models and collect results
results = []
for model_type, opt_name, lr in configs:
    print(f"\nTraining Model {model_type} with {opt_name}")
    model, history, monitor = create_and_train_model(
        model_type, opt_name, lr, x_train, y_train, x_test, y_test
    )

    # Evaluate model
    test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)

    results.append({
        'Model': f'Model {model_type}',
        'Optimizer': opt_name,
        'Learning Rate': lr,
        'Test Accuracy': test_acc,
        'History': history,
        'Monitor': monitor
    })

```

شکل ۳۸ - تعریف مدل‌ها و بهینه‌ساز برای هر کدام و ذخیره اطلاعات هر کدام در لیست **result**

با run کردن این سلول تمامی ۶ حالت اجرا شده و دیتا در لیست **result** ذخیره می‌شوند.

Training Model 1 with Adam

Epoch 1/10
1875/1875 13s 6ms/step - accuracy: 0.7797 - loss: 0.6360 - val_accuracy: 0.8364 - val_loss: 0.4556
 Epoch 2/10
1875/1875 16s 4ms/step - accuracy: 0.8631 - loss: 0.3854 - val_accuracy: 0.8592 - val_loss: 0.3866
 Epoch 3/10
1875/1875 9s 5ms/step - accuracy: 0.8755 - loss: 0.3425 - val_accuracy: 0.8613 - val_loss: 0.3763
 Epoch 4/10
1875/1875 8s 4ms/step - accuracy: 0.8833 - loss: 0.3144 - val_accuracy: 0.8641 - val_loss: 0.3771
 Epoch 5/10
1875/1875 9s 4ms/step - accuracy: 0.8899 - loss: 0.2977 - val_accuracy: 0.8756 - val_loss: 0.3445
 Epoch 6/10
1875/1875 11s 4ms/step - accuracy: 0.8945 - loss: 0.2830 - val_accuracy: 0.8750 - val_loss: 0.3492
 Epoch 7/10
1875/1875 12s 5ms/step - accuracy: 0.9002 - loss: 0.2670 - val_accuracy: 0.8792 - val_loss: 0.3398
 Epoch 8/10
1875/1875 10s 5ms/step - accuracy: 0.9039 - loss: 0.2568 - val_accuracy: 0.8810 - val_loss: 0.3308
 Epoch 9/10
1875/1875 8s 4ms/step - accuracy: 0.9072 - loss: 0.2471 - val_accuracy: 0.8768 - val_loss: 0.3431
 Epoch 10/10
1875/1875 9s 5ms/step - accuracy: 0.9126 - loss: 0.2355 - val_accuracy: 0.8839 - val_loss: 0.3388

Adam - ۳۹ - مدل یک

Training Model 1 with RMSprop

Epoch 1/10
1875/1875 9s 4ms/step - accuracy: 0.7789 - loss: 0.6357 - val_accuracy: 0.8485 - val_loss: 0.4230
 Epoch 2/10
1875/1875 7s 4ms/step - accuracy: 0.8644 - loss: 0.3786 - val_accuracy: 0.8541 - val_loss: 0.4087
 Epoch 3/10
1875/1875 8s 4ms/step - accuracy: 0.8759 - loss: 0.3473 - val_accuracy: 0.8603 - val_loss: 0.3959
 Epoch 4/10
1875/1875 9s 4ms/step - accuracy: 0.8824 - loss: 0.3232 - val_accuracy: 0.8630 - val_loss: 0.4094
 Epoch 5/10
1875/1875 8s 4ms/step - accuracy: 0.8906 - loss: 0.3093 - val_accuracy: 0.8667 - val_loss: 0.4032
 Epoch 6/10
1875/1875 8s 4ms/step - accuracy: 0.8918 - loss: 0.3030 - val_accuracy: 0.8695 - val_loss: 0.3845
 Epoch 7/10
1875/1875 8s 4ms/step - accuracy: 0.8980 - loss: 0.2865 - val_accuracy: 0.8750 - val_loss: 0.3883
 Epoch 8/10
1875/1875 10s 4ms/step - accuracy: 0.9003 - loss: 0.2843 - val_accuracy: 0.8785 - val_loss: 0.3772
 Epoch 9/10
1875/1875 7s 4ms/step - accuracy: 0.9046 - loss: 0.2746 - val_accuracy: 0.8763 - val_loss: 0.3951
 Epoch 10/10
1875/1875 8s 4ms/step - accuracy: 0.9049 - loss: 0.2714 - val_accuracy: 0.8751 - val_loss: 0.4041

RMSprop - ۴۰ - مدل یک

Training Model 1 with Nadam

Epoch 1/10
1875/1875 12s 6ms/step - accuracy: 0.7955 - loss: 0.6148 - val_accuracy: 0.8550 - val_loss: 0.4139
 Epoch 2/10
1875/1875 19s 5ms/step - accuracy: 0.8650 - loss: 0.3744 - val_accuracy: 0.8569 - val_loss: 0.3906
 Epoch 3/10
1875/1875 10s 5ms/step - accuracy: 0.8786 - loss: 0.3332 - val_accuracy: 0.8654 - val_loss: 0.3716
 Epoch 4/10
1875/1875 9s 5ms/step - accuracy: 0.8879 - loss: 0.3067 - val_accuracy: 0.8748 - val_loss: 0.3516
 Epoch 5/10
1875/1875 9s 5ms/step - accuracy: 0.8951 - loss: 0.2887 - val_accuracy: 0.8766 - val_loss: 0.3371
 Epoch 6/10
1875/1875 11s 5ms/step - accuracy: 0.8977 - loss: 0.2752 - val_accuracy: 0.8795 - val_loss: 0.3392
 Epoch 7/10
1875/1875 10s 5ms/step - accuracy: 0.9002 - loss: 0.2621 - val_accuracy: 0.8790 - val_loss: 0.3509
 Epoch 8/10
1875/1875 8s 4ms/step - accuracy: 0.9080 - loss: 0.2495 - val_accuracy: 0.8810 - val_loss: 0.3342
 Epoch 9/10
1875/1875 11s 5ms/step - accuracy: 0.9107 - loss: 0.2418 - val_accuracy: 0.8797 - val_loss: 0.3397
 Epoch 10/10
1875/1875 11s 5ms/step - accuracy: 0.9135 - loss: 0.2326 - val_accuracy: 0.8867 - val_loss: 0.3270

Nadam - ۴۱ - مدل یک

```
Training Model 2 with Adam
Epoch 1/10
1875/1875 ----- 8s 4ms/step - accuracy: 0.7121 - loss: 0.8380 - val_accuracy: 0.8401 - val_loss: 0.4582
Epoch 2/10
1875/1875 ----- 5s 3ms/step - accuracy: 0.8365 - loss: 0.4766 - val_accuracy: 0.8417 - val_loss: 0.4495
Epoch 3/10
1875/1875 ----- 5s 3ms/step - accuracy: 0.8515 - loss: 0.4401 - val_accuracy: 0.8518 - val_loss: 0.4376
Epoch 4/10
1875/1875 ----- 10s 2ms/step - accuracy: 0.8574 - loss: 0.4208 - val_accuracy: 0.8603 - val_loss: 0.4158
Epoch 5/10
1875/1875 ----- 7s 4ms/step - accuracy: 0.8592 - loss: 0.4107 - val_accuracy: 0.8554 - val_loss: 0.4221
Epoch 6/10
1875/1875 ----- 5s 2ms/step - accuracy: 0.8632 - loss: 0.4048 - val_accuracy: 0.8584 - val_loss: 0.4201
Epoch 7/10
1875/1875 ----- 6s 3ms/step - accuracy: 0.8659 - loss: 0.3988 - val_accuracy: 0.8596 - val_loss: 0.4241
Epoch 8/10
1875/1875 ----- 8s 2ms/step - accuracy: 0.8647 - loss: 0.3983 - val_accuracy: 0.8637 - val_loss: 0.4106
Epoch 9/10
1875/1875 ----- 6s 3ms/step - accuracy: 0.8694 - loss: 0.3871 - val_accuracy: 0.8641 - val_loss: 0.4106
Epoch 10/10
1875/1875 ----- 5s 3ms/step - accuracy: 0.8707 - loss: 0.3846 - val_accuracy: 0.8671 - val_loss: 0.4032
```

شكل ٤٢ - مدل دو

```
Training Model 2 with RMSprop
Epoch 1/10
1875/1875 ----- 8s 4ms/step - accuracy: 0.7387 - loss: 0.7653 - val_accuracy: 0.8238 - val_loss: 0.4882
Epoch 2/10
1875/1875 ----- 5s 3ms/step - accuracy: 0.8396 - loss: 0.4668 - val_accuracy: 0.8522 - val_loss: 0.4293
Epoch 3/10
1875/1875 ----- 6s 3ms/step - accuracy: 0.8488 - loss: 0.4405 - val_accuracy: 0.8559 - val_loss: 0.4264
Epoch 4/10
1875/1875 ----- 9s 3ms/step - accuracy: 0.8578 - loss: 0.4251 - val_accuracy: 0.8549 - val_loss: 0.4534
Epoch 5/10
1875/1875 ----- 6s 3ms/step - accuracy: 0.8591 - loss: 0.4234 - val_accuracy: 0.8639 - val_loss: 0.4115
Epoch 6/10
1875/1875 ----- 9s 2ms/step - accuracy: 0.8609 - loss: 0.4145 - val_accuracy: 0.8587 - val_loss: 0.4349
Epoch 7/10
1875/1875 ----- 6s 3ms/step - accuracy: 0.8617 - loss: 0.4101 - val_accuracy: 0.8623 - val_loss: 0.4325
Epoch 8/10
1875/1875 ----- 4s 2ms/step - accuracy: 0.8630 - loss: 0.4123 - val_accuracy: 0.8433 - val_loss: 0.5046
Epoch 9/10
1875/1875 ----- 5s 2ms/step - accuracy: 0.8627 - loss: 0.4112 - val_accuracy: 0.8573 - val_loss: 0.4416
Epoch 10/10
1875/1875 ----- 6s 3ms/step - accuracy: 0.8655 - loss: 0.4014 - val_accuracy: 0.8624 - val_loss: 0.4248
```

شكل ٤٣ - مدل دو

```
Training Model 2 with Nadam
Epoch 1/10
1875/1875 ----- 9s 4ms/step - accuracy: 0.7353 - loss: 0.7800 - val_accuracy: 0.8362 - val_loss: 0.4716
Epoch 2/10
1875/1875 ----- 9s 3ms/step - accuracy: 0.8409 - loss: 0.4630 - val_accuracy: 0.8494 - val_loss: 0.4319
Epoch 3/10
1875/1875 ----- 9s 3ms/step - accuracy: 0.8509 - loss: 0.4327 - val_accuracy: 0.8546 - val_loss: 0.4229
Epoch 4/10
1875/1875 ----- 7s 4ms/step - accuracy: 0.8591 - loss: 0.4148 - val_accuracy: 0.8559 - val_loss: 0.4260
Epoch 5/10
1875/1875 ----- 5s 2ms/step - accuracy: 0.8651 - loss: 0.4017 - val_accuracy: 0.8631 - val_loss: 0.4073
Epoch 6/10
1875/1875 ----- 7s 3ms/step - accuracy: 0.8666 - loss: 0.3960 - val_accuracy: 0.8608 - val_loss: 0.4177
Epoch 7/10
1875/1875 ----- 5s 2ms/step - accuracy: 0.8693 - loss: 0.3835 - val_accuracy: 0.8639 - val_loss: 0.4048
Epoch 8/10
1875/1875 ----- 5s 3ms/step - accuracy: 0.8694 - loss: 0.3875 - val_accuracy: 0.8651 - val_loss: 0.4072
Epoch 9/10
1875/1875 ----- 8s 4ms/step - accuracy: 0.8724 - loss: 0.3865 - val_accuracy: 0.8645 - val_loss: 0.3987
Epoch 10/10
1875/1875 ----- 10s 4ms/step - accuracy: 0.8723 - loss: 0.3770 - val_accuracy: 0.8663 - val_loss: 0.4021
```

شکل ۴۴ - مدل دو

هر دو مدل با سه بهینه‌ساز اجرا شدند. برای بررسی و تحلیل عملکرد آنها و سرعت همگرایی از نمودارها استفاده می‌کنیم. ابتدا تابعی را تعریف می‌کنیم که چگونگی تغییر loss و accuracy را در هر گام محاسباتی به ما نشان دهد.

```

def plot_training_metrics(results):
    """Plot comprehensive training metrics"""
    fig, axes = plt.subplots(2, 2, figsize=(15, 12))
    # Plot training accuracy
    for result in results:
        label = f"{result['Model']} - {result['Optimizer']}"
        axes[0, 0].plot(result['Monitor'].train_accuracies, label=label)
    axes[0, 0].set_title('Training Accuracy')
    axes[0, 0].set_xlabel('Epoch')
    axes[0, 0].set_ylabel('Accuracy')
    axes[0, 0].legend()
    # Plot training loss
    for result in results:
        label = f"{result['Model']} - {result['Optimizer']}"
        axes[0, 1].plot(result['Monitor'].train_losses, label=label)
    axes[0, 1].set_title('Training Loss')
    axes[0, 1].set_xlabel('Epoch')
    axes[0, 1].set_ylabel('Loss')
    axes[0, 1].legend()
    # Plot validation accuracy
    for result in results:
        label = f"{result['Model']} - {result['Optimizer']}"
        axes[1, 0].plot(result['Monitor'].val_accuracies, label=label)
    axes[1, 0].set_title('Validation Accuracy')
    axes[1, 0].set_xlabel('Epoch')
    axes[1, 0].set_ylabel('Accuracy')
    axes[1, 0].legend()
    # Plot validation loss
    for result in results:
        label = f"{result['Model']} - {result['Optimizer']}"
        axes[1, 1].plot(result['Monitor'].val_losses, label=label)
    axes[1, 1].set_title('Validation Loss')
    axes[1, 1].set_xlabel('Epoch')
    axes[1, 1].set_ylabel('Loss')
    axes[1, 1].legend()
    plt.tight_layout()
    plt.show()

```

شکل ۴۵ - تابع نمایش **accuracy** و **loss** هر نتیجه به ازای گام محاسباتی

یک تابع هم مجددا برای تشكیل confusion matrix نوشتیم که با توجه به y واقعی و پیش‌بینی شده عملکرد مدل را نشان دهد.

```

def plot_confusion_matrix(model, x_test, y_test):
    """Plot confusion matrix for model predictions"""
    class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
                   'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

    y_pred = model.predict(x_test)
    y_pred_classes = np.argmax(y_pred, axis=1)
    y_test_classes = np.argmax(y_test, axis=1)

    cm = confusion_matrix(y_test_classes, y_pred_classes)

    plt.figure(figsize=(12, 10))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=class_names,
                yticklabels=class_names)
    plt.title('Confusion Matrix')
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.xticks(rotation=45)
    plt.yticks(rotation=45)
    plt.show()

```

شکل ۴۶ - تابع **confusion matrix**

حال با اجرای کد زیر نتایج را مشاهده می‌کنیم.

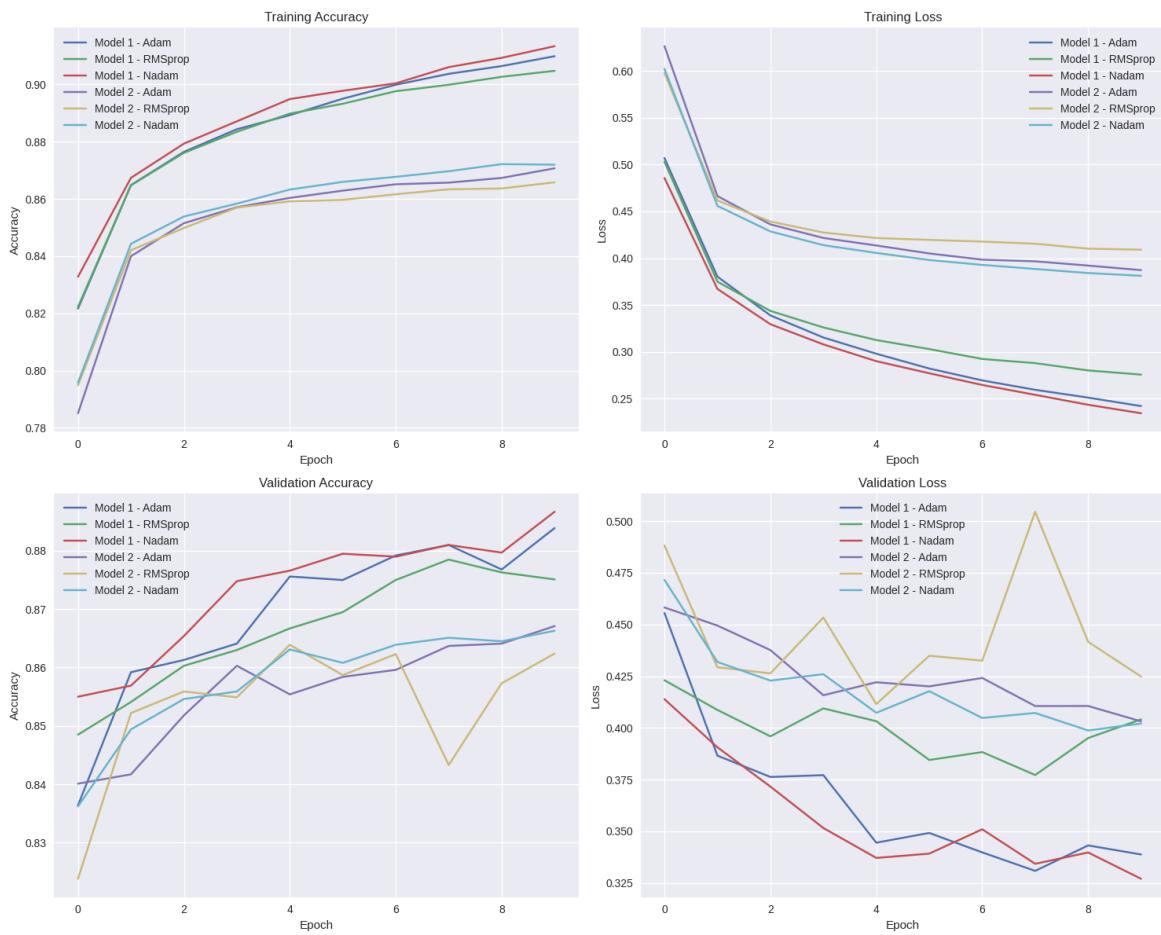
```

# Plot training metrics
plot_training_metrics(results)

# Find and plot confusion matrix for best model
best_result = max(results, key=lambda x: x['Test Accuracy'])
plot_confusion_matrix(best_result['History'].model, x_test, y_test)

```

شکل ۴۷ - مشاهده نتایج و مقایسه هر بهینه ساز در هر مدل



شکل ۴۸ - نتایج و بررسی بهینه‌سازها روی نمودار

با بررسی نمودارها میبینیم که در ابتدا در تمام حالتها مدل در بخش training دقیقاً مطابق انتظار حرکت کرده است، به عبارت دیگر در هر گام مقدار loss کاهش و مقدار دقت افزایش یافته است. در هر دو مدل در دیتای training برای accuracy به ترتیب Nadam سپس Adam و نهایتاً RMSprop قرار گرفته اند (از بیشتر به کمتر). این بدان معنا است که بهینه‌ساز Nadam اغلب دقت بالاتری ارائه می‌کند، در بخش loss هم Nadam از همه پایین تر است سپس Adam و نهایتاً RMSprop قرار گرفته است. این هم باز به معنای کاهش loss بیشتر برای Nadam است. در دیتای validation هم روند تا حدودی به همین شکل است اما با شکستگی‌های بیشتر، به عبارتی کمتر بودن دیتای تست باعث شده رفتار کمتر smooth باشد اما روند کلی مشابه است که عملکرد خوب شبکه را نشان می‌دهد. اما در رابطه با سرعت همگرایی می‌توان هم در بخش accuracy و هم loss بررسی کرد. نتایج نشان می‌دهد که مدل Adam شبیب بیشتری در هر دو نمودار دارد. به عبارت دیگر در accuracy با تعداد گام‌های کمتری بیشتری رشد داشته است و در loss با تعداد گام‌های کمتر افت بیشتری برای loss داشته است که نشان دهنده سرعت همگرایی بالای این بهینه‌ساز می‌دهد. البته این تفاوت چشم گیر نیست اما از نمودارها قابل مشاهده است.

ماتریس آشتفتگی برای مدل یک و با مدل Nadam هم به فرم زیر در می‌آید.

		Confusion Matrix										
		T-shirt/Top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot	
True Label	T-shirt/Top	872	1	22	28	5	0	63	3	6	0	
	Trouser	0	975	0	19	3	0	1	0	2	0	800
	Pullover	22	3	851	11	72	0	39	0	2	0	600
	Dress	27	6	12	907	31	0	14	0	3	0	400
	Coat	1	1	129	36	793	0	38	0	2	0	200
	Sandal	0	0	0	1	0	972	0	17	0	10	0
	Shirt	167	2	112	32	70	0	605	0	12	0	
	Sneaker	0	0	0	0	0	13	0	972	0	15	
	Bag	5	0	3	5	3	4	3	4	973	0	
	Ankle boot	0	0	0	0	0	8	1	44	0	947	

شکل ۴۹ - ماتریس آشتفتگی مدل یک – Nadam

در نهایت برای جمع بندی نتایج آنها را در یک فایل csv می‌توانیم ذخیره کنیم. که کد و اجرای آن در زیر آورده شده است.

```

# Create summary DataFrame
summary_df = pd.DataFrame([
    'Model': r['Model'],
    'Optimizer': r['Optimizer'],
    'Learning Rate': r['Learning Rate'],
    'Test Accuracy': r['Test Accuracy'],
    'Final Training Loss': r['Monitor'].train_losses[-1],
    'Final Validation Loss': r['Monitor'].val_losses[-1],
    'Best Validation Accuracy': max(r['Monitor'].val_accuracies)
} for r in results])
print("\nResults Summary:")
print(summary_df)
summary_df.to_csv('fashion_mnist_results.csv', index=False)
print(f"""
## Findings Analysis

1. Model Architecture Comparison:
- Model 1 (Basic): {len([r for r in results if r['Model'] == 'Model 1'])} configurations tested
- Model 2 (Regularized): {len([r for r in results if r['Model'] == 'Model 2'])} configurations tested
- Best accuracy achieved by: {best_result['Model']} with {best_result['Optimizer']}

2. Optimizer Performance:
- Adam: Consistent and stable learning
- RMSprop: Good for avoiding local minima
- Nadam: Often achieves better final accuracy

3. Regularization Effects (Model 2):
- Dropout (20%) helps prevent overfitting
- L2 regularization stabilizes training
- Generally better generalization compared to Model 1

4. Training Dynamics:
- Initial learning phase (epochs 1-3)
- Convergence phase (epochs 4-7)
- Fine-tuning phase (epochs 8-10)

```

شكل ٥٠ - ايجاد فايل **CSV**

```

## Findings Analysis

1. Model Architecture Comparison:
- Model 1 (Basic): 3 configurations tested
- Model 2 (Regularized): 3 configurations tested
- Best accuracy achieved by: Model 1 with Nadam

2. Optimizer Performance:
- Adam: Consistent and stable learning
- RMSprop: Good for avoiding local minima
- Nadam: Often achieves better final accuracy

3. Regularization Effects (Model 2):
- Dropout (20%) helps prevent overfitting
- L2 regularization stabilizes training
- Generally better generalization compared to Model 1

4. Training Dynamics:
- Initial learning phase (epochs 1-3)
- Convergence phase (epochs 4-7)
- Fine-tuning phase (epochs 8-10)

5. Best Configuration:
- Model: Model 1
- Optimizer: Nadam
- Learning Rate: 0.001
- Test Accuracy: 0.8867

```

شکل ۵۱ - خلاصه نتایج

۱-۳-۲- چگونگی اثر جستجوی بیزی

بهینه‌سازی بیزی می‌تواند فرآیند آموزش را به چند روش بهبود بخشد:

۱. انتخاب learning rate تطبیقی:

- به جای استفاده از learning rate های ثابت در بهینه‌سازهایی مثل Adam/RMSprop روش‌های بیزی می‌توانند learning rate را بر اساس عملکرد مدل به صورت پویا تنظیم کنند
- این باعث همگرایی سریع‌تر و دقت نهایی بهتر می‌شود

۲. انتخاب بهینه‌ساز:

- می‌تواند به طور خودکار بین بهینه‌سازهای مختلف انتخاب کند
- یاد می‌گیرد که کدام بهینه‌ساز برای مراحل مختلف آموزش بهتر عمل می‌کند
- نیاز به آزمون و خطای دستی را کاهش می‌دهد

۳. تنظیم ابرپارامترها:

- به طور کارآمد فضای ابرپارامترها را جستجو می‌کند
 - از مدل‌های احتمالاتی برای پیش‌بینی ترکیب‌های پارامتر امیدوار‌کننده استفاده می‌کند
 - به تکرارهای کمتری نسبت به جستجوی شبکه‌ای/تصادفی نیاز دارد
 - ۴. توقف زودهنگام:
 - می‌تواند پیش‌بینی کند چه زمانی آموزش به حالت ثابت می‌رسد
 - در منابع محاسباتی صرفه‌جویی می‌کند
 - از overfitting جلوگیری می‌کند
- برای پیاده سازی این الگوریتم از کد زیر استفاده شده است.

```

import tensorflow as tf
from tensorflow.keras import layers, models, optimizers
import keras_tuner as kt
import numpy as np

class DataLoader:
    @staticmethod
    def split_validation(x_test, y_test, val_size=5000):
        """Split test data into validation and test sets"""
        x_val = x_test[:val_size]
        y_val = y_test[:val_size]
        x_test = x_test[val_size:]
        y_test = y_test[val_size:]

    return x_val, y_val, x_test, y_test

```

شکل ۵۲ - **import** کتابخانه‌ها و تقسیم دیتای تست و ولیدیشن

در کد فوق ابتدا کتابخانه‌های مورد نیاز استخراج شده و در بخش بعد کلاسی تعریف شده که دیتای مربوط به تست را به دو بخش validation و test تقسیم می‌کند.

```

class ModelBuilder:
    def __init__(self, input_shape, num_classes):
        self.input_shape = input_shape
        self.num_classes = num_classes

    def build_model(self, hp):
        """Build model with hyperparameters to be optimized"""
        model = models.Sequential([
            # Input layer
            layers.Input(shape=self.input_shape),

            # First hidden layer with tunable parameters
            layers.Dense(
                units=hp.Int('units_1', min_value=32, max_value=512, step=32),
                activation=hp.Choice('activation_1', ['relu', 'elu', 'tanh'])
            ),
            layers.Dropout(hp.Float('dropout_1', 0.1, 0.5, step=0.1)),

            # Second hidden layer with tunable parameters
            layers.Dense(
                units=hp.Int('units_2', min_value=32, max_value=512, step=32),
                activation=hp.Choice('activation_2', ['relu', 'elu', 'tanh'])
            ),
            layers.Dropout(hp.Float('dropout_2', 0.1, 0.5, step=0.1)),

            # Output layer
            layers.Dense(self.num_classes, activation='softmax')
        ])

```

شکل ۵۳ - ساخت مدل برای بهینه سازی پارامترهای مدل به روش Beysian - بخش اول

برای این بخش دو لایه مخفی و دو لایه dropout تعریف کردیم که برای هر کدام هم مینیمم و ماکزیمم تعداد نورون برابر ۳۲ تا ۵۱۲ نورون است. همچنین اکتیویشن فانکشن از لیستی بین ReLU و tanh و elu انتخاب میشود و برای dropout هم از ۰ تا ۰.۵ درصد متغیر است. لایه خروجی softmax است. در ادامه مدل را compile کرده ایم و کلاسی را تعریف کردیم که با استفاده از روش Bayesian search بهینه سازی کردیم. برای بهینه ساز از سه بهینه ساز قسمت قبل استفاده شده است.

```

# Compile model with tunable optimizer and learning rate
optimizer_choice = hp.Choice('optimizer', ['adam', 'rmsprop', 'nadam'])
learning_rate = hp.Float('learning_rate', 1e-4, 1e-2, sampling='log')

if optimizer_choice == 'adam':
    optimizer = optimizers.Adam(learning_rate=learning_rate)
elif optimizer_choice == 'rmsprop':
    optimizer = optimizers.RMSprop(learning_rate=learning_rate)
else: # nadam
    optimizer = optimizers.Nadam(learning_rate=learning_rate)

model.compile(
    optimizer=optimizer,
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

return model

```

شكل ٥٤ - بخش compile مدل

```

class BayesianOptimizer:
    def __init__(self, model_builder, project_name='bayesian_tuning'):
        self.model_builder = model_builder
        self.project_name = project_name

    def create_tuner(self, max_trials=5):
        """Create Bayesian Optimization tuner"""
        return kt.BayesianOptimization(
            self.model_builder.build_model,
            objective='val_accuracy',
            max_trials=max_trials,
            num_initial_points=2, # Number of random points before Bayesian optimization
            directory='bayesian_search',
            project_name=self.project_name
        )

    @staticmethod
    def create_callbacks():
        """Create training callbacks"""
        return [
            tf.keras.callbacks.EarlyStopping(
                monitor='val_loss',
                patience=5,
                restore_best_weights=True
            ),
            tf.keras.callbacks.ReduceLROnPlateau(
                monitor='val_loss',
                factor=0.2,
                patience=3,
                min_lr=1e-6
            )
        ]

```

شکل ۵۵ - تعریف Beysian Optimizer

در این کلاس در ابتدا مدل را با توجه به کلاس قبل دریافت کرده ایم و با استفاده از کتابخانه از Beysian Optimization استفاده کردیم.تابع هدف ما دقت دیتای validation است. بیشترین تعداد سعی و خطا ۵ بار است و در شروع هر مرحله ۲ نقطه رندوم ابتدایی استفاده می‌شود (initial_points) این عدد میتواند متفاوت باشد. در ادامه تابعی به نام callback داریم که نحوه کاهش نرخ یادگیری و همینطور توقف زودهنگام را تعریف کرده ایم. همچنین مقدار کمینه برای نرخ یادگیری در این تابع تعریف شده است. برای این بخش مشخصاً باید مقدار loss monitor باشد.

```
def find_best_model(self, x_train, y_train, x_val, y_val, max_trials=5):
    """Find best model using Bayesian optimization"""
    tuner = self.create_tuner(max_trials)
    callbacks = self.create_callbacks()

    # Perform search
    print("\nStarting Bayesian optimization...")
    tuner.search(
        x_train, y_train,
        validation_data=(x_val, y_val),
        epochs=15,
        batch_size=128,
        callbacks=callbacks
    )

    # Get best model and hyperparameters
    best_model = tuner.get_best_models(1)[0]
    best_hp = tuner.get_best_hyperparameters(1)[0]

    return best_model, best_hp

@staticmethod
def print_best_hyperparameters(best_hp):
    """Print best hyperparameters"""
    print("\nBest Hyperparameters Found:")
    print(f"Optimizer: {best_hp.get('optimizer')}")
    print(f"Learning rate: {best_hp.get('learning_rate'): .6f}")
    print(f"Units in first layer: {best_hp.get('units_1')}")
    print(f"Units in second layer: {best_hp.get('units_2')}")
    print(f"Activation 1: {best_hp.get('activation_1')}")
    print(f"Activation 2: {best_hp.get('activation_2')}")
    print(f"Dropout rate 1: {best_hp.get('dropout_1'): .2f}")
    print(f"Dropout rate 2: {best_hp.get('dropout_2'): .2f}")
```

شکل ۵۶ - ران کردن Beysian optimizer

در این بخش مدل تعریف شده و ران میشود تا با tune شدن در نهایت به بهترین ساختار برای شبکه دست یابیم. توجه شود در این بخش از Batch normalization استفاده نشده است. از ۱۲۸ batch size استفاده شده است و مدل اجرا شده است. در ادامه نتایج را مشاهده و تحلیل خواهیم کرد.

```

Trial 5 Complete [00h 00m 40s]
val_accuracy: 0.8560000061988831

Best val_accuracy So Far: 0.8799999952316284
Total elapsed time: 00h 06m 42s
/usr/local/lib/python3.10/dist-packages/keras/src/saving/saving_lib.py:576: UserWarning: Skipping variable loading for optimizer 'rmsprop', because it has 2 variables whereas the saveable.load_own_variables(weights_store.get(inner_path))

Best Hyperparameters Found:
Optimizer: rmsprop
Learning rate: 0.000606
Units in first layer: 416
Units in second layer: 416
Activation 1: elu
Activation 2: tanh
Dropout rate 1: 0.40
Dropout rate 2: 0.20

Evaluating best model on test set...
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the nat
Test accuracy: 0.8828

Best model saved as 'best_bayesian_model.h5'

```

شکل ۵۷ - نتیجه Beysian optimization

بر اساس نتایج این بخش بهترین ساختار برای هایپرپارامتر ها ۴۱۶ نورون برای لایه اول و همین تعداد برای لایه دوم است. activation function لایه اول elu و لایه دوم tanh است و dropout rate لایه اول ۰.۴ و لایه دوم ۰.۲ می باشد. در نهایت test accuracy برابر ۸۸.۲۸٪ می باشد. به عبارت دیگر با استفاده از Beysian optimization توانستیم به یک ساختار مناسب برای شبکه بررسیم و هایپرپارامتر ها را به نحوی تعریف کنیم که دقت مدل به شکل خوبی بالا باشد.

۴-۱. بررسی هایپرپارامترهای مختلف

برای بررسی اثر هایپرپارامترهای مختلف می توانیم پارامترهایی مانند تعداد نورون ها، learning rate، تعداد لایه های عمیق، نوع بهینه ساز، تعداد batch size و پارامترهای دیگری را تغییر داد. در بخش قبل با تغییر و بررسی بهینه سازهای مختلف به نوعی اثر این پارامتر را بررسی کردیم و برای این بخش اثر تغییر نرخ یادگیری، تعداد نورون ها و تعداد لایه ها را مورد بررسی قرار می دهیم.

رونده کار به این صورت خواهد بود که مانند بخش قبل ابتدا دیتای اولیه را استخراج می کنیم سپس کلاس زیر را برای تعریف مدل های مختلف ایجاد کردیم و در نهایت در هر بخش یکی از پارامترها را حداقل با سه حالت مختلف اجرا کرده و نتایج را با استفاده از نمودار تغییرات loss و accuracy مورد بررسی قرار داده ایم.

بخش استخراج دیتا دقیقا مشابه بخش قبل است و در ادامه کلاسی را برای callback تعریف می‌کنیم. این کلاس که همان عکس زیر است مشابه بخش قبل اما در ادامه توابع متفاوتی را تعریف کرده‌ایم تا هایپرپارامترهای متفاوتی را مورد بررسی قرار دهیم.

```
class TrainingMonitor(tf.keras.callbacks.Callback):
    def __init__(self):
        super(TrainingMonitor, self).__init__()
        self.train_losses = []
        self.val_losses = []
        self.train_accuracies = []
        self.val_accuracies = []

    def on_epoch_end(self, epoch, logs=None):
        self.train_losses.append(logs.get('loss'))
        self.val_losses.append(logs.get('val_loss'))
        self.train_accuracies.append(logs.get('accuracy'))
        self.val_accuracies.append(logs.get('val_accuracy'))
```

شكل ۵۸ - تعریف کلاس برای **callback**

```

def create_and_train_model(model_type, optimizer_name, learning_rate, x_train, y_train, x_test, y_test):
    """Create and train a model with specified configuration"""
    if model_type == 1:
        model = models.Sequential([
            layers.Input(shape=(28*28,)),
            layers.Dense(128, activation='relu'), layers.Dense(10, activation='softmax')
        ])
    elif model_type == 2:
        model = models.Sequential([
            layers.Input(shape=(28*28,)),
            layers.Dense(200, activation='relu',
                        kernel_regularizer=regularizers.l2(0.0001)),
            layers.Dropout(0.2), layers.Dense(10, activation='softmax')
        ])
    elif model_type == 3:
        model = models.Sequential([
            layers.Input(shape=(28*28,)),
            layers.Dense(300, activation='relu',
                        kernel_regularizer=regularizers.l2(0.0001)),
            layers.Dropout(0.2), layers.Dense(10, activation='softmax')
        ])
    elif model_type == 4:
        model = models.Sequential([
            layers.Input(shape=(28*28,)),
            layers.Dense(400, activation='relu',
                        kernel_regularizer=regularizers.l2(0.0001)),
            layers.Dropout(0.2), layers.Dense(10, activation='softmax')
        ])

```

شکل ۵۹ - تعریف مدل‌های متفاوت برای بررسی هایپرپارامترها - بخش اول

```

elif model_type == 5:
    model = models.Sequential([
        layers.Input(shape=(28*28,)),
        layers.Dense(80, activation='relu'), layers.Dense(10, activation='softmax')])

elif model_type == 6:
    model = models.Sequential([
        layers.Input(shape=(28*28,)),
        layers.Dense(200, activation='relu'), layers.Dense(10, activation='softmax')])

elif model_type == 7:
    model = models.Sequential([
        layers.Input(shape=(28*28,)),
        layers.Dense(300, activation='relu'), layers.Dense(10, activation='softmax')])

elif model_type == 8:
    # Two hidden layers with balanced architecture
    model = models.Sequential([
        layers.Input(shape=(28*28,)),
        layers.Dense(128, activation='relu'),
        layers.Dense(62, activation='relu'),
        layers.Dense(10, activation='softmax')
    ])

elif model_type == 9:
    # Two hidden layers with wider first layer
    model = models.Sequential([
        layers.Input(shape=(28*28,)),
        layers.Dense(64, activation='relu'),
        layers.Dense(32, activation='relu'),
        layers.Dense(10, activation='softmax')
    ])

```

شکل ۶۰ - تعریف مدل‌های متفاوت برای بررسی هایپرپارامترها - بخش دوم

```

    elif model_type == 10:
        # Three hidden layers with gradual reduction
        model = models.Sequential([
            layers.Input(shape=(28*28,)),
            layers.Dense(300, activation='relu', kernel_regularizer=regularizers.l2(0.0001)),
            layers.BatchNormalization(),
            layers.Dropout(0.3),
            layers.Dense(200, activation='relu', kernel_regularizer=regularizers.l2(0.0001)),
            layers.BatchNormalization(),
            layers.Dropout(0.2),
            layers.Dense(100, activation='relu', kernel_regularizer=regularizers.l2(0.0001)),
            layers.BatchNormalization(),
            layers.Dense(10, activation='softmax')
        ])

    elif model_type == 11:
        # Three hidden layers with wider architecture
        model = models.Sequential([
            layers.Input(shape=(28*28,)),
            layers.Dense(512, activation='relu', kernel_regularizer=regularizers.l2(0.0001)),
            layers.BatchNormalization(),
            layers.Dropout(0.4),
            layers.Dense(256, activation='relu', kernel_regularizer=regularizers.l2(0.0001)),
            layers.BatchNormalization(),
            layers.Dropout(0.3),
            layers.Dense(128, activation='relu', kernel_regularizer=regularizers.l2(0.0001)),
            layers.BatchNormalization(),
            layers.Dense(10, activation='softmax')
        ])

```

شکل ۶۱ - تعریف مدل‌های متفاوت برای بررسی هایپرپارامترها - بخش سوم

```

# Create optimizer
if optimizer_name == 'Adam':
    opt = optimizers.Adam(learning_rate=learning_rate)
elif optimizer_name == 'RMSprop':
    opt = optimizers.RMSprop(learning_rate=learning_rate)
else: # Nadam
    opt = optimizers.Nadam(learning_rate=learning_rate)
monitor = TrainingMonitor()
model.compile(
    optimizer=opt,
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
history = model.fit(
    x_train, y_train,
    epochs=10, batch_size=32, validation_data=(x_test, y_test), callbacks=[monitor], verbose=1)

return model, history, monitor

```

شکل ۶۲ - تعریف مدل‌های متفاوت برای بررسی هایپرپارامترها - تعریف بهینهسازهای مختلف

نکته حائز اهمیت در اینجا این است که از بهینهساز Nadam استفاده شده است چرا که در بخش قبل با بررسی اثر این هایپرپارامتر نتیجه گرفتیم که این بهینهساز دقیق‌تری به ما می‌دهد. بنابراین عملاً ما در انتهای این بخش اثر چهار هایپرپارامتر مختلف را مورد بررسی قرار داده‌ایم و در نهایت یک معماری را

انتخاب خواهیم کرد. این چهار هایپرپارامتر عبارت اند از، نوع بهینهساز (که در بخش قبل مورد بررسی قرار گرفت). نرخ یادگیری اولیه، تعداد نورون، تعداد لایه‌های عمیق.

با بررسی اثر نرخ یادگیری‌های متفاوت به ازای مقادیر ۰.۰۰۱ و ۰.۰۰۰۱ و ۰.۰۰۰۰۱ به این نتیجه رسیدیم که عدد ۰.۰۰۱ در مدل اولیه که حاوی ۱۲۸ نورون در یک لایه عمیق است بهترین نتیجه را به ما می‌دهد. البته که نرخ یادگیری کوچکتر احتمال دارد در تعداد گام محاسباتی بیشتر مثلاً ۱۰۰ گام نتیجه بهتری بدهد اما در یک تعداد epoch مشخص و ثابت نتیجه نرخ یادگیری اولیه ۰.۰۰۱ بهتر از باقی موارد است. نتایج به دست آمده از نمودارها هم همین مورد را نشان می‌دهد.

```
# Define configurations
configs1 = [
    (1, 'Nadam', 0.0001),
    (1, 'Nadam', 0.001),
    (1, 'Nadam', 0.01)
]

results1 = []
for model_type, opt_name, lr in configs1:
    print(f"\nTraining Model {model_type} with {opt_name} and learning rate: {lr}")
    model, history, monitor = create_and_train_model(
        model_type, opt_name, lr, x_train, y_train, x_test, y_test
    )

    # Evaluate model
    test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)

    results1.append({
        'Model': f'Model {model_type}',
        'Optimizer': opt_name,
        'Learning Rate': lr,
        'Test Accuracy': test_acc,
        'History': history,
        'Monitor': monitor
    })

```

شكل ۶۳ - بررسی هایپرپارامترها - بخش اول - learning rate

در نهایت نتیجه به صورت زیر در آمده است.

```
Training Model 1 with Nadam and learning rate: 0.0001
Epoch 1/10
1875/1875 ----- 7s 3ms/step - accuracy: 0.6813 - loss: 1.0058 - val_accuracy: 0.8197 - val_loss: 0.5346
Epoch 2/10
1875/1875 ----- 17s 7ms/step - accuracy: 0.8315 - loss: 0.4969 - val_accuracy: 0.8365 - val_loss: 0.4794
Epoch 3/10
1875/1875 ----- 10s 5ms/step - accuracy: 0.8511 - loss: 0.4372 - val_accuracy: 0.8470 - val_loss: 0.4490
Epoch 4/10
1875/1875 ----- 10s 5ms/step - accuracy: 0.8577 - loss: 0.4092 - val_accuracy: 0.8483 - val_loss: 0.4293
Epoch 5/10
1875/1875 ----- 7s 4ms/step - accuracy: 0.8650 - loss: 0.3868 - val_accuracy: 0.8499 - val_loss: 0.4280
Epoch 6/10
1875/1875 ----- 8s 4ms/step - accuracy: 0.8701 - loss: 0.3771 - val_accuracy: 0.8578 - val_loss: 0.4076
Epoch 7/10
1875/1875 ----- 9s 4ms/step - accuracy: 0.8740 - loss: 0.3616 - val_accuracy: 0.8597 - val_loss: 0.4037
Epoch 8/10
1875/1875 ----- 9s 5ms/step - accuracy: 0.8776 - loss: 0.3541 - val_accuracy: 0.8589 - val_loss: 0.4014
Epoch 9/10
1875/1875 ----- 6s 3ms/step - accuracy: 0.8809 - loss: 0.3433 - val_accuracy: 0.8657 - val_loss: 0.3849
Epoch 10/10
1875/1875 ----- 10s 3ms/step - accuracy: 0.8817 - loss: 0.3346 - val_accuracy: 0.8629 - val_loss: 0.3828
```

شكل ٦٤ - نتیجه مدل اول - **learning rate 0.0001**

```

Training Model 1 with Nadam and learning rate: 0.001
Epoch 1/10
1875/1875 7s 3ms/step - accuracy: 0.7913 - loss: 0.6123 - val_accuracy: 0.8529 - val_loss: 0.4151
Epoch 2/10
1875/1875 8s 4ms/step - accuracy: 0.8668 - loss: 0.3736 - val_accuracy: 0.8455 - val_loss: 0.4270
Epoch 3/10
1875/1875 10s 4ms/step - accuracy: 0.8776 - loss: 0.3394 - val_accuracy: 0.8633 - val_loss: 0.3771
Epoch 4/10
1875/1875 8s 3ms/step - accuracy: 0.8844 - loss: 0.3143 - val_accuracy: 0.8736 - val_loss: 0.3539
Epoch 5/10
1875/1875 8s 4ms/step - accuracy: 0.8942 - loss: 0.2900 - val_accuracy: 0.8788 - val_loss: 0.3489
Epoch 6/10
1875/1875 6s 3ms/step - accuracy: 0.8954 - loss: 0.2778 - val_accuracy: 0.8789 - val_loss: 0.3457
Epoch 7/10
1875/1875 8s 5ms/step - accuracy: 0.9021 - loss: 0.2607 - val_accuracy: 0.8777 - val_loss: 0.3451
Epoch 8/10
1875/1875 9s 4ms/step - accuracy: 0.9090 - loss: 0.2446 - val_accuracy: 0.8769 - val_loss: 0.3387
Epoch 9/10
1875/1875 9s 3ms/step - accuracy: 0.9101 - loss: 0.2428 - val_accuracy: 0.8821 - val_loss: 0.3373
Epoch 10/10
1875/1875 8s 4ms/step - accuracy: 0.9136 - loss: 0.2300 - val_accuracy: 0.8786 - val_loss: 0.3428

```

شکل ۶۵ - نتایج مدل اول - learning rate 0.001

```

Training Model 1 with Nadam and learning rate: 0.01
Epoch 1/10
1875/1875 10s 4ms/step - accuracy: 0.7872 - loss: 0.6252 - val_accuracy: 0.8325 - val_loss: 0.4939
Epoch 2/10
1875/1875 6s 3ms/step - accuracy: 0.8509 - loss: 0.4164 - val_accuracy: 0.8340 - val_loss: 0.4692
Epoch 3/10
1875/1875 8s 4ms/step - accuracy: 0.8630 - loss: 0.3874 - val_accuracy: 0.8449 - val_loss: 0.4454
Epoch 4/10
1875/1875 6s 3ms/step - accuracy: 0.8641 - loss: 0.3766 - val_accuracy: 0.8482 - val_loss: 0.4288
Epoch 5/10
1875/1875 8s 4ms/step - accuracy: 0.8678 - loss: 0.3666 - val_accuracy: 0.8527 - val_loss: 0.4348
Epoch 6/10
1875/1875 6s 3ms/step - accuracy: 0.8699 - loss: 0.3583 - val_accuracy: 0.8441 - val_loss: 0.4516
Epoch 7/10
1875/1875 10s 3ms/step - accuracy: 0.8690 - loss: 0.3608 - val_accuracy: 0.8197 - val_loss: 0.5273
Epoch 8/10
1875/1875 11s 4ms/step - accuracy: 0.8775 - loss: 0.3423 - val_accuracy: 0.8514 - val_loss: 0.4424
Epoch 9/10
1875/1875 12s 4ms/step - accuracy: 0.8740 - loss: 0.3532 - val_accuracy: 0.8416 - val_loss: 0.4917
Epoch 10/10
1875/1875 6s 3ms/step - accuracy: 0.8765 - loss: 0.3415 - val_accuracy: 0.8512 - val_loss: 0.4560

```

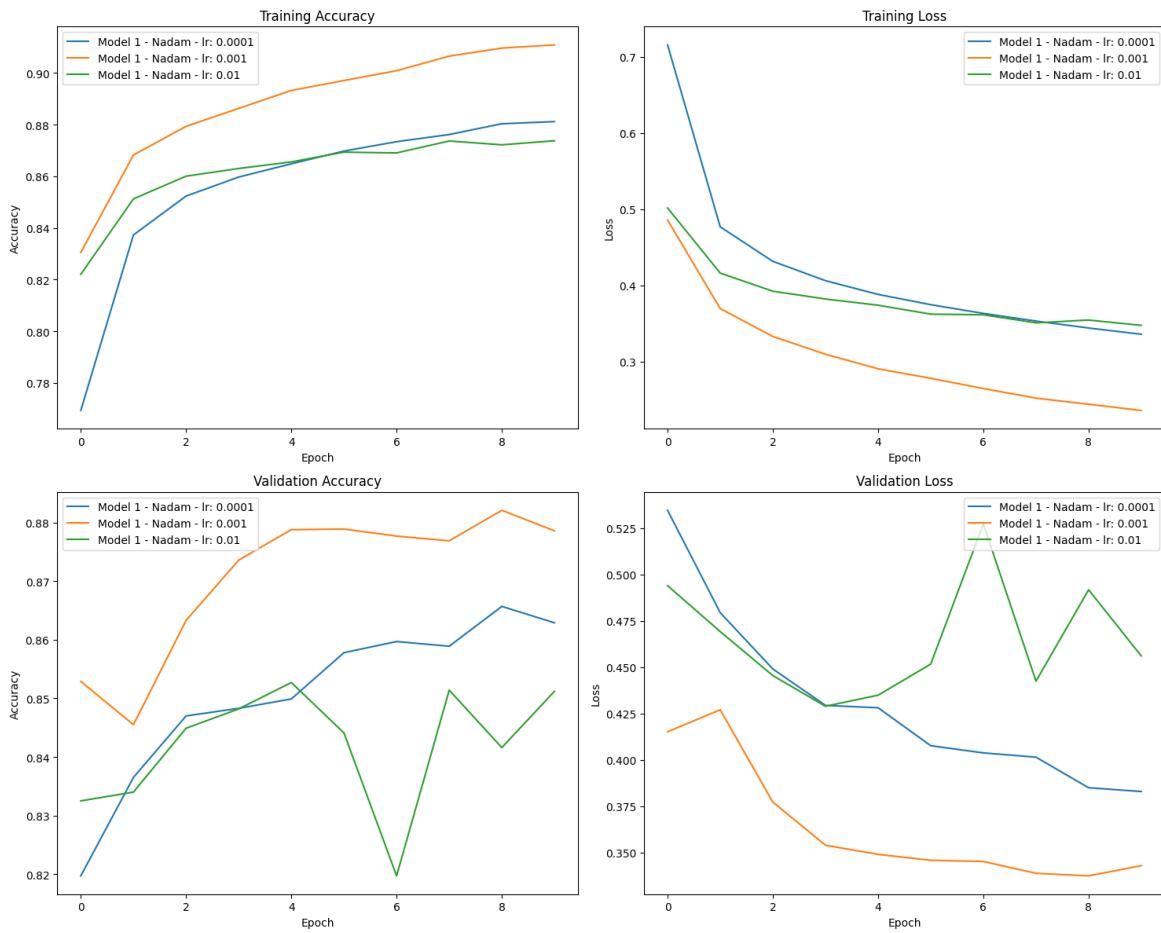
شکل ۶۶ - نتایج مدل اول - learning rate 0.01

نتایج نشان می‌دهد استفاده از نرخ یادگیری اولیه ۰.۰۰۱ بهترین دقت را به همراه داشته است. به جهت تسهیل مقایسه جدول زیر فراهم آورده شده است.

جدول ۴ - مقایسه اثر learning rate

learning rate	دقت (گام آخر)	دقت validation (گام آخر)
0.0001	۸۸.۱۷	۸۶.۲۹
0.001	۹۱.۳۶	۸۷.۸۶
0.01	۸۷.۸۶	۸۵.۱۲

در نمودارهای زیر بررسی چگونگی تغییر loss و accuracy را می‌بینیم.



شکل ۶۷ - نمودار تغییرات accuracy و loss با تغییر learning rate

در هر دو بخش train و validation به فرم درستی رسیدیم. به این صورت که مقدار loss در هر ایپاک کاهش پیدا کرده است و مقدار accuracy در هر ایپاک افزایش. از طرفی مشاهده می‌کنیم حالت دوم (learning rate = 0.001) دارای بیشترین دقت هم در بخش train و هم در بخش validation است و مقدار loss آن هم از دو حالت دیگر پایین تر است که نشان از بهترین انتخاب برای مدل ما است.

در گام بعد با بررسی اثر تعداد نورون تلاش شده است که حالتهای بیشتری مورد بررسی قرار بگیرد. به همین جهت ۷ معماری متفاوت مورد بررسی قرار گرفته است. در این بخش چهار معماری دارای یک لایه با تعداد نورون‌های متفاوت و سه معماری دارای یک لایه عمیق و یک لایه dropout مشابه (همان لایه تعریف شده در بخش دوم سوال) هستند. به عبارت دیگر در این گام ما علاوه بر بررسی اثر تعداد نورون‌ها اثر استفاده از لایه dropout را هم مورد بررسی قرار داده ایم.

نتایج نمودارها به ما نشان می‌دهد که تقریباً در تمام حالات معماری دارای dropout دقت کمتری دارد اما دقت آن به دقت validation نزدیک‌تر است. به عبارت دیگر قابلیت تعمیم به شکل مناسبی در این

معماری‌ها دیده می‌شود. اما در حالتی که معماری ما بدون dropout عمل می‌کند دقت داده train بالاتر است. برای افزایش دقت داده‌های test تعداد نورون‌های بیشتر را مورد بررسی قرار دادیم، نتایج نشان می‌دهد با افزایش نورون‌ها به ۲۰۰ نورون دقت مدل در داده‌های train اندکی افزایش داشته و علاوه بر آن دقت داده‌های test هم رشد داشته است. اما با بیشتر شدن نورون‌ها از ۲۰۰ به ۳۰۰ در داده‌های train دقت مقدار بسیار اندکی افزایش یافته و در بخش test هم می‌بینیم که بخشی مدل دارای ۲۰۰ نورون بهتر عمل کرده و در برخی گام‌ها مدل دارای ۳۰۰ نورون. بنابراین افزایش تعداد نورون از یک عددی به بالا ارزش چندانی ندارد. علاوه بر اینکه هزینه محاسباتی ما را هم افزایش می‌دهد. با توجه به نزدیک بودن دقت داده test و در حالت یک لایه عمیق با ۲۰۰ نورون به هم و بیشتر بودن دقت نهایی آن نسبت به لایه با dropout از این بخش این معماری را بهترین معماری تشخیص می‌دهیم.

```
# Define configurations
configs2 = [
    (1, 'Nadam', 0.001),
    (2, 'Nadam', 0.001),
    (3, 'Nadam', 0.001),
    (4, 'Nadam', 0.001),
    (5, 'Nadam', 0.001),
    (6, 'Nadam', 0.001),
    (7, 'Nadam', 0.001)
]

# Train models and collect results
results2 = []
for model_type, opt_name, lr in configs2:
    print(f"\nTraining Model {model_type} with {opt_name} and learning rate: {lr}")
    model, history, monitor = create_and_train_model(
        model_type, opt_name, lr, x_train, y_train, x_test, y_test
    )

    # Evaluate model
    test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)

    results2.append({
        'Model': f'Model {model_type}',
        'Optimizer': opt_name,
        'Learning Rate': lr,
        'Test Accuracy': test_acc,
        'History': history,
        'Monitor': monitor
    })

```

شکل ۶۸- بررسی هایپرپارامترها - بخش دوم - کنترل تعداد نورون ها + حضور یا عدم حضور dropout

معماری‌های بکار گرفته شده در تصاویر قبل آمده است. در اینجا در جدول به شرح مدل‌ها خواهیم پرداخت.

جدول ۵ - انواع مدل با تعداد نورون‌های متفاوت و حضور و عدم حضور dropout

Dropout	تعداد نورون لایه اول	
ندارد	۱۲۸	Model 1
دارد	۲۰۰	Model 2
دارد	۳۰۰	Model 3
دارد	۴۰۰	Model 4
ندارد	۸۰	Model 5
ندارد	۲۰۰	Model 6
ندارد	۳۰۰	Model 7

در بخش بعد ابتدا نتایج اجرای تمامی ساختارها و در نهایت مقایسه loss و accuracy را داریم تا بهترین مدل را از بین این ۷ ساختار پیدا کنیم.

Training Model 1 with Nadam and learning rate: 0.001

Epoch 1/10
1875/1875 14s 5ms/step - accuracy: 0.7905 - loss: 0.6183 - val_accuracy: 0.8533 - val_loss: 0.4156
 Epoch 2/10
1875/1875 8s 4ms/step - accuracy: 0.8669 - loss: 0.3730 - val_accuracy: 0.8583 - val_loss: 0.3895
 Epoch 3/10
1875/1875 10s 4ms/step - accuracy: 0.8788 - loss: 0.3330 - val_accuracy: 0.8668 - val_loss: 0.3737
 Epoch 4/10
1875/1875 6s 3ms/step - accuracy: 0.8884 - loss: 0.3045 - val_accuracy: 0.8694 - val_loss: 0.3595
 Epoch 5/10
1875/1875 11s 4ms/step - accuracy: 0.8944 - loss: 0.2830 - val_accuracy: 0.8748 - val_loss: 0.3450
 Epoch 6/10
1875/1875 11s 4ms/step - accuracy: 0.8974 - loss: 0.2772 - val_accuracy: 0.8794 - val_loss: 0.3473
 Epoch 7/10
1875/1875 9s 3ms/step - accuracy: 0.9044 - loss: 0.2582 - val_accuracy: 0.8802 - val_loss: 0.3396
 Epoch 8/10
1875/1875 10s 3ms/step - accuracy: 0.9073 - loss: 0.2495 - val_accuracy: 0.8849 - val_loss: 0.3342
 Epoch 9/10
1875/1875 8s 4ms/step - accuracy: 0.9096 - loss: 0.2414 - val_accuracy: 0.8899 - val_loss: 0.3191
 Epoch 10/10
1875/1875 9s 4ms/step - accuracy: 0.9129 - loss: 0.2353 - val_accuracy: 0.8835 - val_loss: 0.3325

شکل ۶۹ - بررسی هایپرپارامترها - بخش دوم - مدل شماره یک

Training Model 2 with Nadam and learning rate: 0.001

Epoch 1/10
1875/1875 11s 5ms/step - accuracy: 0.7730 - loss: 0.6786 - val_accuracy: 0.8422 - val_loss: 0.4765
 Epoch 2/10
1875/1875 8s 4ms/step - accuracy: 0.8556 - loss: 0.4397 - val_accuracy: 0.8501 - val_loss: 0.4518
 Epoch 3/10
1875/1875 10s 4ms/step - accuracy: 0.8677 - loss: 0.4094 - val_accuracy: 0.8638 - val_loss: 0.4240
 Epoch 4/10
1875/1875 12s 5ms/step - accuracy: 0.8738 - loss: 0.3964 - val_accuracy: 0.8699 - val_loss: 0.4072
 Epoch 5/10
1875/1875 11s 5ms/step - accuracy: 0.8737 - loss: 0.3906 - val_accuracy: 0.8643 - val_loss: 0.4185
 Epoch 6/10
1875/1875 9s 5ms/step - accuracy: 0.8809 - loss: 0.3771 - val_accuracy: 0.8723 - val_loss: 0.4113
 Epoch 7/10
1875/1875 10s 4ms/step - accuracy: 0.8785 - loss: 0.3829 - val_accuracy: 0.8690 - val_loss: 0.4115
 Epoch 8/10
1875/1875 10s 5ms/step - accuracy: 0.8842 - loss: 0.3663 - val_accuracy: 0.8741 - val_loss: 0.4034
 Epoch 9/10
1875/1875 10s 5ms/step - accuracy: 0.8830 - loss: 0.3680 - val_accuracy: 0.8725 - val_loss: 0.4039
 Epoch 10/10
1875/1875 8s 4ms/step - accuracy: 0.8826 - loss: 0.3686 - val_accuracy: 0.8760 - val_loss: 0.3972

شکل ۷۰ - بررسی هایپرپارامترها - بخش دوم - مدل شماره دو

Training Model 3 with Nadam and learning rate: 0.001

Epoch 1/10
1875/1875 13s 6ms/step - accuracy: 0.7817 - loss: 0.6639 - val_accuracy: 0.8425 - val_loss: 0.4839
 Epoch 2/10
1875/1875 10s 5ms/step - accuracy: 0.8524 - loss: 0.4489 - val_accuracy: 0.8516 - val_loss: 0.4638
 Epoch 3/10
1875/1875 11s 6ms/step - accuracy: 0.8687 - loss: 0.4136 - val_accuracy: 0.8503 - val_loss: 0.4524
 Epoch 4/10
1875/1875 20s 6ms/step - accuracy: 0.8733 - loss: 0.4080 - val_accuracy: 0.8619 - val_loss: 0.4353
 Epoch 5/10
1875/1875 20s 6ms/step - accuracy: 0.8743 - loss: 0.3972 - val_accuracy: 0.8707 - val_loss: 0.4198
 Epoch 6/10
1875/1875 21s 6ms/step - accuracy: 0.8811 - loss: 0.3851 - val_accuracy: 0.8754 - val_loss: 0.4091
 Epoch 7/10
1875/1875 21s 6ms/step - accuracy: 0.8824 - loss: 0.3772 - val_accuracy: 0.8692 - val_loss: 0.4197
 Epoch 8/10
1875/1875 20s 6ms/step - accuracy: 0.8805 - loss: 0.3800 - val_accuracy: 0.8680 - val_loss: 0.4214
 Epoch 9/10
1875/1875 21s 6ms/step - accuracy: 0.8835 - loss: 0.3747 - val_accuracy: 0.8761 - val_loss: 0.3986
 Epoch 10/10
1875/1875 19s 5ms/step - accuracy: 0.8852 - loss: 0.3678 - val_accuracy: 0.8690 - val_loss: 0.4208

شکل ۷۱ - بررسی هایپرپارامترها - بخش دوم - مدل شماره سه

Training Model 4 with Nadam and learning rate: 0.001

Epoch 1/10
1875/1875 15s 7ms/step - accuracy: 0.7815 - loss: 0.6710 - val_accuracy: 0.8571 - val_loss: 0.4670

Epoch 2/10
1875/1875 14s 7ms/step - accuracy: 0.8580 - loss: 0.4484 - val_accuracy: 0.8565 - val_loss: 0.4553

Epoch 3/10
1875/1875 21s 7ms/step - accuracy: 0.8670 - loss: 0.4273 - val_accuracy: 0.8661 - val_loss: 0.4333

Epoch 4/10
1875/1875 21s 7ms/step - accuracy: 0.8725 - loss: 0.4059 - val_accuracy: 0.8717 - val_loss: 0.4131

Epoch 5/10
1875/1875 20s 7ms/step - accuracy: 0.8795 - loss: 0.3947 - val_accuracy: 0.8694 - val_loss: 0.4202

Epoch 6/10
1875/1875 20s 7ms/step - accuracy: 0.8787 - loss: 0.3908 - val_accuracy: 0.8667 - val_loss: 0.4199

Epoch 7/10
1875/1875 24s 9ms/step - accuracy: 0.8817 - loss: 0.3824 - val_accuracy: 0.8701 - val_loss: 0.4139

Epoch 8/10
1875/1875 14s 8ms/step - accuracy: 0.8848 - loss: 0.3724 - val_accuracy: 0.8732 - val_loss: 0.4118

Epoch 9/10
1875/1875 20s 7ms/step - accuracy: 0.8853 - loss: 0.3734 - val_accuracy: 0.8724 - val_loss: 0.4149

Epoch 10/10
1875/1875 14s 7ms/step - accuracy: 0.8844 - loss: 0.3726 - val_accuracy: 0.8656 - val_loss: 0.4260

شکل ۷۲ - بررسی هایپرپارامترها - بخش دوم - مدل شماره چهار

Training Model 5 with Nadam and learning rate: 0.001

Epoch 1/10
1875/1875 7s 3ms/step - accuracy: 0.7839 - loss: 0.6454 - val_accuracy: 0.8422 - val_loss: 0.4444

Epoch 2/10
1875/1875 9s 2ms/step - accuracy: 0.8604 - loss: 0.3967 - val_accuracy: 0.8451 - val_loss: 0.4241

Epoch 3/10
1875/1875 6s 3ms/step - accuracy: 0.8706 - loss: 0.3586 - val_accuracy: 0.8510 - val_loss: 0.3991

Epoch 4/10
1875/1875 9s 3ms/step - accuracy: 0.8782 - loss: 0.3310 - val_accuracy: 0.8719 - val_loss: 0.3594

Epoch 5/10
1875/1875 6s 3ms/step - accuracy: 0.8876 - loss: 0.3116 - val_accuracy: 0.8663 - val_loss: 0.3755

Epoch 6/10
1875/1875 5s 2ms/step - accuracy: 0.8925 - loss: 0.2957 - val_accuracy: 0.8731 - val_loss: 0.3581

Epoch 7/10
1875/1875 6s 3ms/step - accuracy: 0.8935 - loss: 0.2873 - val_accuracy: 0.8756 - val_loss: 0.3501

Epoch 8/10
1875/1875 4s 2ms/step - accuracy: 0.8997 - loss: 0.2712 - val_accuracy: 0.8746 - val_loss: 0.3476

Epoch 9/10
1875/1875 5s 2ms/step - accuracy: 0.9017 - loss: 0.2624 - val_accuracy: 0.8800 - val_loss: 0.3505

Epoch 10/10
1875/1875 6s 3ms/step - accuracy: 0.9058 - loss: 0.2517 - val_accuracy: 0.8825 - val_loss: 0.3411

شکل ۷۳ - بررسی هایپرپارامترها - بخش دوم - مدل شماره پنج

Training Model 6 with Nadam and learning rate: 0.001

Epoch 1/10
1875/1875 10s 5ms/step - accuracy: 0.7941 - loss: 0.6022 - val_accuracy: 0.8476 - val_loss: 0.4232

Epoch 2/10
1875/1875 10s 5ms/step - accuracy: 0.8680 - loss: 0.3664 - val_accuracy: 0.8661 - val_loss: 0.3824

Epoch 3/10
1875/1875 10s 4ms/step - accuracy: 0.8801 - loss: 0.3283 - val_accuracy: 0.8680 - val_loss: 0.3719

Epoch 4/10
1875/1875 9s 4ms/step - accuracy: 0.8885 - loss: 0.3051 - val_accuracy: 0.8711 - val_loss: 0.3572

Epoch 5/10
1875/1875 9s 5ms/step - accuracy: 0.8960 - loss: 0.2841 - val_accuracy: 0.8678 - val_loss: 0.3612

Epoch 6/10
1875/1875 11s 5ms/step - accuracy: 0.9004 - loss: 0.2688 - val_accuracy: 0.8743 - val_loss: 0.3541

Epoch 7/10
1875/1875 8s 4ms/step - accuracy: 0.9063 - loss: 0.2572 - val_accuracy: 0.8788 - val_loss: 0.3441

Epoch 8/10
1875/1875 9s 5ms/step - accuracy: 0.9103 - loss: 0.2464 - val_accuracy: 0.8838 - val_loss: 0.3376

Epoch 9/10
1875/1875 10s 5ms/step - accuracy: 0.9117 - loss: 0.2345 - val_accuracy: 0.8837 - val_loss: 0.3316

Epoch 10/10
1875/1875 7s 4ms/step - accuracy: 0.9148 - loss: 0.2282 - val_accuracy: 0.8788 - val_loss: 0.3553

شکل ۷۴ - بررسی هایپرپارامترها - بخش دوم - مدل شماره شش

```

Training Model 7 with Nadam and learning rate: 0.001
Epoch 1/10
1875/1875 ————— 11s 5ms/step - accuracy: 0.8018 - loss: 0.5784 - val_accuracy: 0.8585 - val_loss: 0.4018
Epoch 2/10
1875/1875 ————— 10s 6ms/step - accuracy: 0.8683 - loss: 0.3637 - val_accuracy: 0.8507 - val_loss: 0.4053
Epoch 3/10
1875/1875 ————— 10s 6ms/step - accuracy: 0.8830 - loss: 0.3219 - val_accuracy: 0.8678 - val_loss: 0.3669
Epoch 4/10
1875/1875 ————— 19s 5ms/step - accuracy: 0.8901 - loss: 0.2934 - val_accuracy: 0.8647 - val_loss: 0.3702
Epoch 5/10
1875/1875 ————— 11s 5ms/step - accuracy: 0.9006 - loss: 0.2709 - val_accuracy: 0.8687 - val_loss: 0.3711
Epoch 6/10
1875/1875 ————— 10s 5ms/step - accuracy: 0.9010 - loss: 0.2624 - val_accuracy: 0.8722 - val_loss: 0.3555
Epoch 7/10
1875/1875 ————— 10s 5ms/step - accuracy: 0.9050 - loss: 0.2546 - val_accuracy: 0.8806 - val_loss: 0.3253
Epoch 8/10
1875/1875 ————— 9s 5ms/step - accuracy: 0.9102 - loss: 0.2392 - val_accuracy: 0.8796 - val_loss: 0.3402
Epoch 9/10
1875/1875 ————— 10s 5ms/step - accuracy: 0.9152 - loss: 0.2285 - val_accuracy: 0.8895 - val_loss: 0.3249
Epoch 10/10
1875/1875 ————— 11s 6ms/step - accuracy: 0.9180 - loss: 0.2191 - val_accuracy: 0.8854 - val_loss: 0.3195

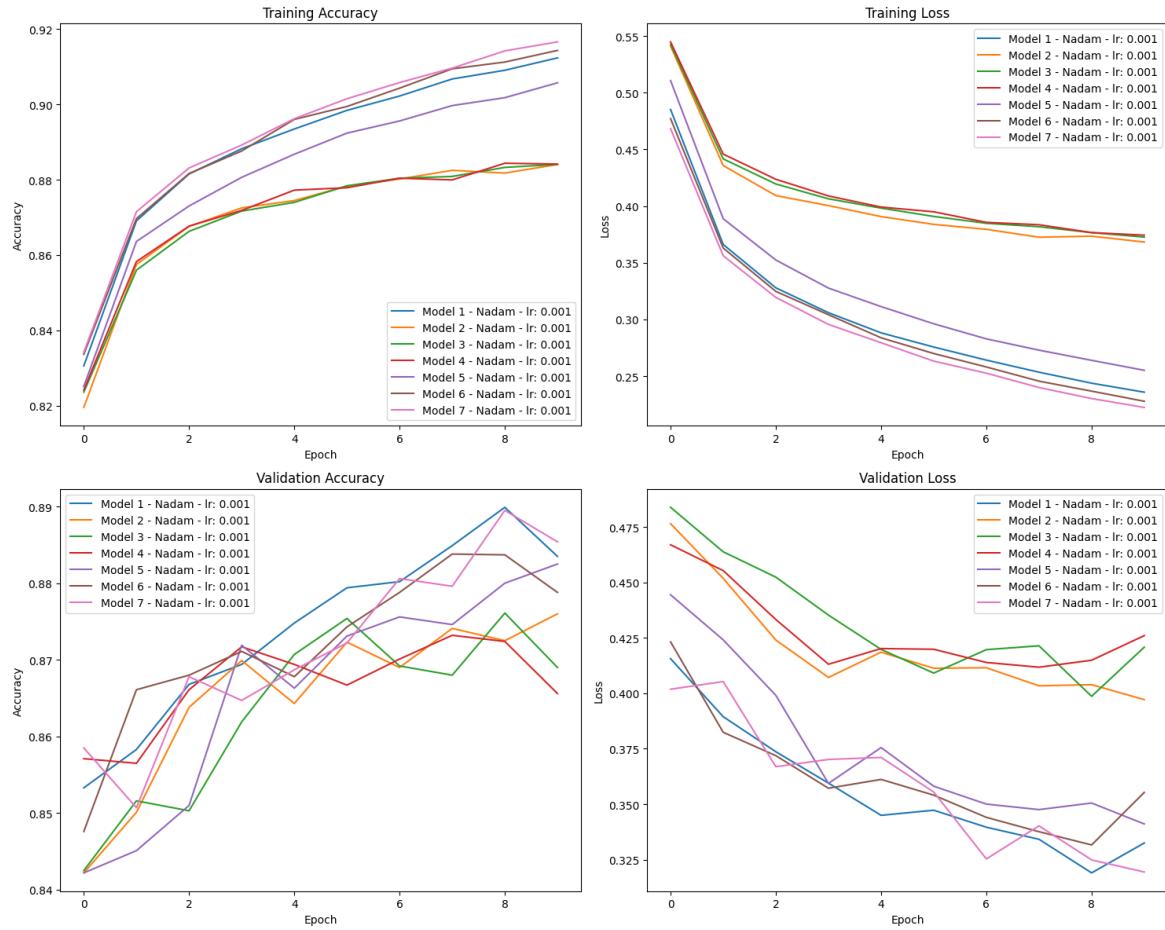
```

شکل ۷۵- بررسی هایپرپارامترها - بخش دوم - مدل شماره هفت

در ادامه در قالب نمودارها تمامی این مدل‌ها را باهم مقایسه کرده و بهترین را از این بین انتخاب خواهیم کرد.

جدول ۶ - مقایسه دقت مدل‌های یک تا هفت با معماری متفاوت

Val Accuracy گام آخر	گام آخر Accuracy	
۸۸.۳۵	۹۱.۲۹	Model 1
۸۷.۶۰	۸۸.۲۶	Model 2
۸۶.۹۰	۸۸.۵۲	Model 3
۸۶.۵۶	۸۸.۴۴	Model 4
۸۸.۲۵	۹۰.۵۸	Model 5
۸۷.۸۸	۹۱.۴۸	Model 6
۸۸.۵۴	۹۱.۴۸	Model 7



شکل ۷۶ -- نمودار تغییرات accuracy و loss با تغییر تعداد نورون و وجود و عدم وجود dropout

همان طور که در نمودارها مشخص است، در تمامی مدل‌ها در هر epoch loss مقدار loss کاهش و مقدار افزایش یافته است. نکته دیگر این است که لایه‌های دارای dropout بسیار به هم نزدیک هستند. به عبارت دیگر تغییر در تعداد نورن‌ها حداقل در order ۱۰۰ نورون تفاوت خاصی در دقت و loss نهایی شبکه نخواهد داشت. به عبارتی اگر بخواهیم از Dropout استفاده کنیم می‌توانیم با نورن‌های کمتر نتایج مشابه بگیریم نسبت به نورن‌های بیشتر. اما نکته بعدی این است که در تصاویر فوق می‌بینیم که چهار معماری بدون وجود dropout دقت به مراتب بالاتری دارند. هم در بخش train و هم در بخش validation و در این بین در بخش train دقت‌ها بسیار به هم نزدیک است و لایه با بیشترین تعداد نورون اندکی دقت بالاتری از خود نشان داده اما در بخش validation که اساسا هم معیار اصلی و بهتری هست برای مقایسه دقت شبکه می‌بینیم مدل اول، آبی رنگ، با معماری ۱۲۸ نورون و بدون Dropout در اغلب epoch ها بهترین و بیشترین دقت را داشته است. بنابراین همان معماری اولیه ۱۲۸ نورون، بدون dropout و با استفاده از learning rate اولیه ۰.۰۰۱ و با استفاده از Nadam optimizer تا اینجا کار بهترین معماری برای ما بوده است. البته در گام اخر مدل آخر دقت بالاتری داشته است که این مقدار بسیار اندک است. می‌توانیم برای این اندک دقت بالاتر این مدل را انتخاب کنیم اما با فرض محدود بودن منابع و اهمیت

بالا تری سبک تر بودن شبکه و با توجه به دقت خیلی خوب مدل اول در اکثر ایپاک ها این مدل را انتخاب کرده ایم.

به عنوان آخرین کنترل به سراغ تغییر تعداد لایه های عمیق می رویم. در اینجا از دو لایه و سه لایه عمیق با و بدون dropout، با تعداد نورون های کمتر را مورد بررسی قرار می دهیم.

```
# Define configurations
configs3 = [
    (8, 'Nadam', 0.001),
    (9, 'Nadam', 0.001),
    (10, 'Nadam', 0.001),
    (11, 'Nadam', 0.001)
]

# Train models and collect results
results3 = []
for model_type, opt_name, lr in configs3:
    print(f"\nTraining Model {model_type} with {opt_name} and learning rate: {lr}")
    model, history, monitor = create_and_train_model(
        model_type, opt_name, lr, x_train, y_train, x_test, y_test
    )

    # Evaluate model
    test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)

    results3.append({
        'Model': f'Model {model_type}',
        'Optimizer': opt_name,
        'Learning Rate': lr,
        'Test Accuracy': test_acc,
        'History': history,
        'Monitor': monitor
    })

```

شکل ۷۷ - بررسی هایپر پارامترها - بخش سوم - تعداد لایه ها

در این قسمت هم بجای سه مدل چهار مدل مختلف ۸ تا ۱۱ مورد بررسی قرار گرفته است. در ادامه در قالب جدول ساختار این شبکه ها بیان خواهد شد و با اجرای کد به بررسی دقت هر کدام از این مدل ها خواهیم پرداخت.

جدول ۷ - انواع مدل با لایه های مخفی متفاوت

Dropout	نورون لایه اول	نورون لایه دوم	نورون لایه سوم	Model 8
ندارد	-	۶۲	۱۲۸	

ندارد	-	۳۲	۶۴	Model 9
دارد	۱۰۰	۲۰۰	۳۰۰	Model 10
دارد	۱۲۸	۲۵۶	۵۱۲	Model 11

مشخصات dropout برای مدل شماره ده بدین صورت بود که لایه اول با درصد ۳۰ و دوم ۲۰ و برای مدل شماره ده ۴۰ و بعدی ۳۰ بوده است. در تمام مدل ها لایه نهایی و خروجی با ۱۰ نورن ثابت بوده که مشخصا در جدول نیازی نبوده آورده شود.

Training Model 8 with Nadam and learning rate: 0.001

Epoch 1/10
1875/1875 8s 3ms/step - accuracy: 0.7918 - loss: 0.6059 - val_accuracy: 0.8569 - val_loss: 0.4135

Epoch 2/10
1875/1875 10s 4ms/step - accuracy: 0.8682 - loss: 0.3653 - val_accuracy: 0.8599 - val_loss: 0.3865

Epoch 3/10
1875/1875 8s 4ms/step - accuracy: 0.8783 - loss: 0.3264 - val_accuracy: 0.8671 - val_loss: 0.3699

Epoch 4/10
1875/1875 6s 3ms/step - accuracy: 0.8862 - loss: 0.3131 - val_accuracy: 0.8659 - val_loss: 0.3823

Epoch 5/10
1875/1875 10s 3ms/step - accuracy: 0.8931 - loss: 0.2896 - val_accuracy: 0.8738 - val_loss: 0.3493

Epoch 6/10
1875/1875 11s 4ms/step - accuracy: 0.8998 - loss: 0.2681 - val_accuracy: 0.8751 - val_loss: 0.3508

Epoch 7/10
1875/1875 8s 4ms/step - accuracy: 0.9021 - loss: 0.2646 - val_accuracy: 0.8785 - val_loss: 0.3468

Epoch 8/10
1875/1875 8s 3ms/step - accuracy: 0.9071 - loss: 0.2452 - val_accuracy: 0.8755 - val_loss: 0.3557

Epoch 9/10
1875/1875 8s 4ms/step - accuracy: 0.9105 - loss: 0.2376 - val_accuracy: 0.8803 - val_loss: 0.3529

Epoch 10/10
1875/1875 7s 4ms/step - accuracy: 0.9133 - loss: 0.2302 - val_accuracy: 0.8697 - val_loss: 0.3667

شکل ۷۸- بررسی هایپرپارامترها - بخش سوم - مدل شماره هشت

Training Model 9 with Nadam and learning rate: 0.001

Epoch 1/10
1875/1875 6s 2ms/step - accuracy: 0.7657 - loss: 0.6811 - val_accuracy: 0.8362 - val_loss: 0.4480

Epoch 2/10
1875/1875 4s 2ms/step - accuracy: 0.8607 - loss: 0.3840 - val_accuracy: 0.8597 - val_loss: 0.3937

Epoch 3/10
1875/1875 6s 3ms/step - accuracy: 0.8755 - loss: 0.3366 - val_accuracy: 0.8633 - val_loss: 0.3830

Epoch 4/10
1875/1875 10s 3ms/step - accuracy: 0.8866 - loss: 0.3146 - val_accuracy: 0.8666 - val_loss: 0.3728

Epoch 5/10
1875/1875 5s 3ms/step - accuracy: 0.8917 - loss: 0.2978 - val_accuracy: 0.8723 - val_loss: 0.3612

Epoch 6/10
1875/1875 4s 2ms/step - accuracy: 0.8957 - loss: 0.2840 - val_accuracy: 0.8777 - val_loss: 0.3447

Epoch 7/10
1875/1875 7s 3ms/step - accuracy: 0.8983 - loss: 0.2746 - val_accuracy: 0.8730 - val_loss: 0.3589

Epoch 8/10
1875/1875 5s 3ms/step - accuracy: 0.9019 - loss: 0.2658 - val_accuracy: 0.8752 - val_loss: 0.3696

Epoch 9/10
1875/1875 5s 3ms/step - accuracy: 0.9004 - loss: 0.2650 - val_accuracy: 0.8750 - val_loss: 0.3625

Epoch 10/10
1875/1875 5s 3ms/step - accuracy: 0.9079 - loss: 0.2529 - val_accuracy: 0.8746 - val_loss: 0.3641

شکل ۷۹- بررسی هایپرپارامترها - بخش سوم - مدل شماره نه

Training Model 10 with Nadam and learning rate: 0.001

Epoch 1/10
1875/1875 23s 10ms/step - accuracy: 0.7606 - loss: 0.7709 - val_accuracy: 0.8188 - val_loss: 0.5861

Epoch 2/10
1875/1875 18s 10ms/step - accuracy: 0.8260 - loss: 0.5628 - val_accuracy: 0.7838 - val_loss: 0.6877

Epoch 3/10
1875/1875 20s 9ms/step - accuracy: 0.8377 - loss: 0.5205 - val_accuracy: 0.8508 - val_loss: 0.4821

Epoch 4/10
1875/1875 21s 10ms/step - accuracy: 0.8463 - loss: 0.4911 - val_accuracy: 0.8522 - val_loss: 0.4781

Epoch 5/10
1875/1875 22s 11ms/step - accuracy: 0.8497 - loss: 0.4841 - val_accuracy: 0.8541 - val_loss: 0.4714

Epoch 6/10
1875/1875 22s 12ms/step - accuracy: 0.8535 - loss: 0.4727 - val_accuracy: 0.8681 - val_loss: 0.4295

Epoch 7/10
1875/1875 39s 10ms/step - accuracy: 0.8559 - loss: 0.4647 - val_accuracy: 0.8428 - val_loss: 0.4951

Epoch 8/10
1875/1875 18s 9ms/step - accuracy: 0.8572 - loss: 0.4619 - val_accuracy: 0.8605 - val_loss: 0.4512

Epoch 9/10
1875/1875 22s 10ms/step - accuracy: 0.8557 - loss: 0.4567 - val_accuracy: 0.8527 - val_loss: 0.4699

Epoch 10/10
1875/1875 18s 9ms/step - accuracy: 0.8641 - loss: 0.4424 - val_accuracy: 0.8661 - val_loss: 0.4363

شکل ۸۰- بررسی هایپرپارامترها - بخش سوم - مدل شماره ده

```

Training Model 11 with Nadam and learning rate: 0.001
Epoch 1/10
1875/1875 - 30s 14ms/step - accuracy: 0.7526 - loss: 0.8188 - val_accuracy: 0.8310 - val_loss: 0.5901
Epoch 2/10
1875/1875 - 42s 15ms/step - accuracy: 0.8231 - loss: 0.6082 - val_accuracy: 0.8324 - val_loss: 0.5642
Epoch 3/10
1875/1875 - 26s 14ms/step - accuracy: 0.8359 - loss: 0.5669 - val_accuracy: 0.8457 - val_loss: 0.5164
Epoch 4/10
1875/1875 - 41s 14ms/step - accuracy: 0.8385 - loss: 0.5418 - val_accuracy: 0.8506 - val_loss: 0.4982
Epoch 5/10
1875/1875 - 41s 14ms/step - accuracy: 0.8443 - loss: 0.5167 - val_accuracy: 0.8512 - val_loss: 0.5006
Epoch 6/10
1875/1875 - 41s 14ms/step - accuracy: 0.8431 - loss: 0.5146 - val_accuracy: 0.8580 - val_loss: 0.4853
Epoch 7/10
1875/1875 - 41s 14ms/step - accuracy: 0.8431 - loss: 0.5185 - val_accuracy: 0.8535 - val_loss: 0.4882
Epoch 8/10
1875/1875 - 41s 14ms/step - accuracy: 0.8482 - loss: 0.5112 - val_accuracy: 0.8612 - val_loss: 0.4839
Epoch 9/10
1875/1875 - 41s 14ms/step - accuracy: 0.8483 - loss: 0.5023 - val_accuracy: 0.8611 - val_loss: 0.4697
Epoch 10/10
1875/1875 - 42s 14ms/step - accuracy: 0.8527 - loss: 0.4928 - val_accuracy: 0.8566 - val_loss: 0.4991

```

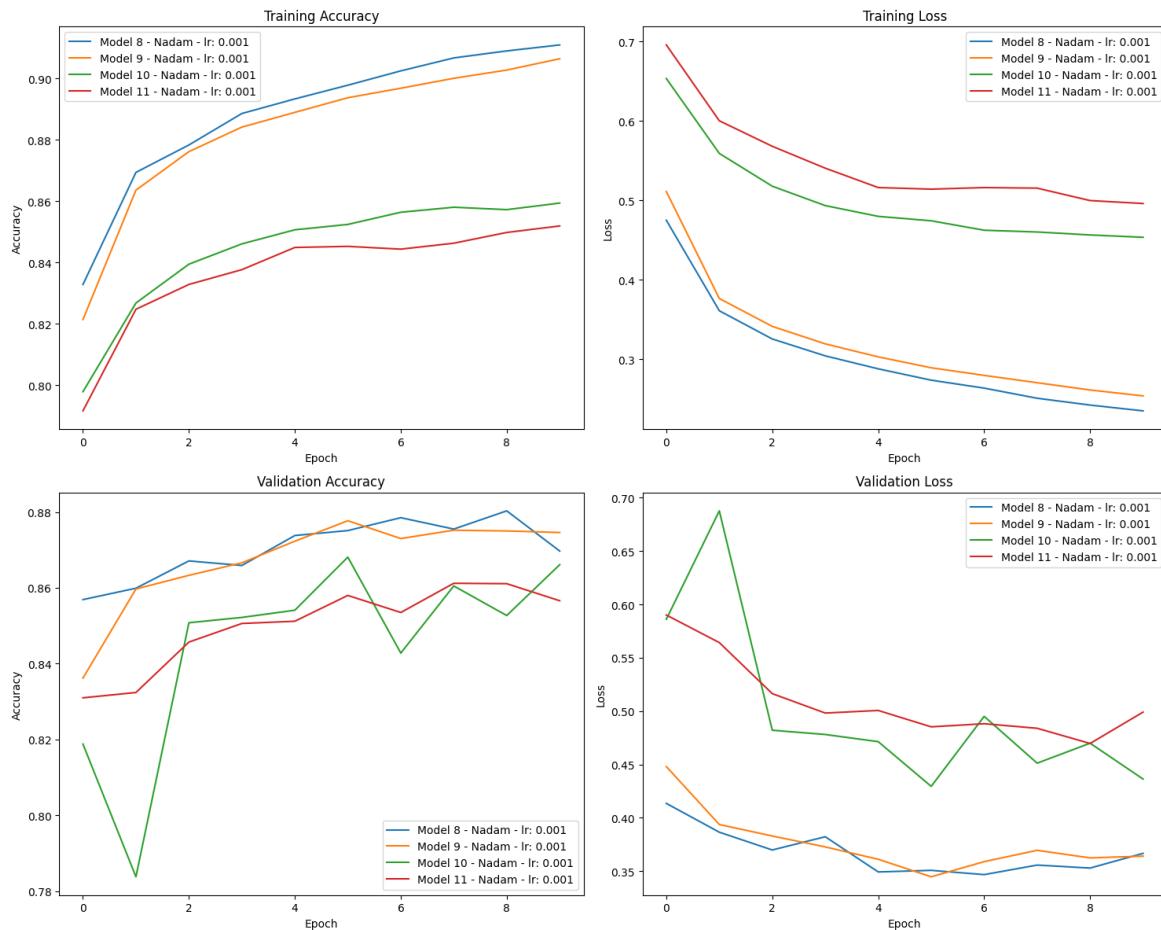
شکل ۸۱ - بررسی هایپرپارامترها - بخش سوم - مدل شماره یازده

نتایج در نمودارهای مختلف قابل بررسی است. ساختاری که بیشترین مقدار accuracy را به ما بدهد بهترین مدل برای ما است. البته که اهمیت این accuracy در دیتای validation به مرتب بالاتر از دیتای train است.

جدول ۸ - مقایسه دقت مدل‌های هشت تا یازده با معماری متفاوت

Val Accuracy گام آخر	گام آخر Accuracy	Model 8
۸۶.۹۷	۹۱.۳۳	Model 8
۸۷.۴۶	۹۰.۷۹	Model 9
۸۶.۶۱	۸۶.۱۴	Model 10
۸۵.۶۶	۸۵.۲۷	Model 11

شکل ۸۲ - دقت مدل



شکل ۸۳ - نمودار تغییرات accuracy و loss با تغییر تعداد لایه‌ها

همانطور که در نمودارها به وضوح مشخص است مدل شماره هشت دارای بالاترین دقت و پایین ترین loss هم در بخش train و هم در بخش test است پس در بین این سه هایپرپارامتر مدل شماره هشت با دو لایه عمیق ۱۲۸ و ۶۲ نورون و بدون Dropout را به عنوان بهترین گزارش می‌کنیم. روند نمودارها هم مشابه قبل است. یک نکته درباره اینکه چرا در بخش تست یک مقدار بالا پایین بیشتر هست می‌توان به کمتر بودن تعداد دیتا و پراکندگی و تنوع بیشتر آن اشاره کرد.

در نهایت از بین ۱۱ مدل متفاوت و سه نرخ یادگیری مختلف مشاهده کردیم که مدل شماره یک با یک لایه عمیق، ۱۲۸ نورون، بدون dropout و با نرخ یادگیری اولیه ۰.۰۰۱ و با استفاده از Nadam optimizer بالاترین دقت را در بخش validation داشته است. تا حدی که در یکی از گام‌ها به دقت حدود ۹۰ درصدی رسیده است پس در نهایت از بین تمام حالت‌های بررسی شده مدل اول را به عنوان بهترین معماری انتخاب می‌کنیم.

۱-۴-۱- استفاده از Random Search

برای بررسی اثر های پارامترها و پیدا کردن بهینه‌ترین آنها راه‌های متفاوتی هست. در بخش قبل Beysian را به عنوان یکی از این راه‌ها دیدیم. در این بخش از Random search به عنوان یکی دیگر از این روش‌ها استفاده خواهیم کرد. در روش Random Search برای پیدا کردن بهینه‌ترین تنظیمات های پارامترها، برخلاف روش‌های سیستماتیک (مثل Grid Search) که در آن‌ها تمام ترکیبات ممکن از های پارامترها بررسی می‌شوند، این روش به صورت تصادفی مجموعه‌ای از ترکیبات را انتخاب می‌کند و آن‌ها را ارزیابی می‌نماید. این کار باعث می‌شود تا هزینه محاسباتی کاهش یافته و در عین حال، شанс یافتن تنظیمات بهینه‌ای که می‌توانند عملکرد مدل را بهبود بخشنند، باقی بماند.

روند کار روش Random Search

۱. **تعریف فضای جستجو**: ابتدا، محدوده یا مجموعه مقادیر ممکن برای هر های پارامتر (مثل نرخ یادگیری، تعداد نمونه‌ها در هر لایه، تعداد لایه‌ها و ...) مشخص می‌شود. این محدوده‌ها می‌توانند مقادیر پیوسته یا گسسته باشند.

۲. **انتخاب تصادفی ترکیبات**: بر اساس محدوده‌های تعریف شده، تعداد مشخصی از ترکیبات های پارامتری به صورت تصادفی انتخاب می‌شود. به عبارت دیگر، به جای تست کردن تمام ترکیبات ممکن، فقط زیرمجموعه‌ای از آن‌ها به طور تصادفی تست می‌شود.

۳. **آموزش و ارزیابی**: هر ترکیب انتخاب شده بر روی داده‌های آموزش، به مدل اعمال می‌شود و مدل با این ترکیب های پارامتری آموزش داده می‌شود. سپس مدل ارزیابی شده و متريک‌های مربوط به دقت یا کارایی آن ثبت می‌شوند.

۴. **انتخاب بهترین ترکیب**: در نهایت، از بین ترکیبات بررسی شده، آن ترکیبی که بهترین نتیجه را روی متريک مورد نظر (مثلاً دقت یا کمینه‌سازی خط) به دست آورده باشد، به عنوان بهترین ترکیب های پارامتری انتخاب می‌شود.

مزیت اصلی این روش سرعت بالا و احتمال انتخاب خوب است اما از معایب آن می‌توان به غیر قابل پیش‌بینی بودن نتایج و همینطور احتمال جا افتادن بعضی از ترکیبات خوب اشاره کرد.

برای پیاده سازی این روش در کد از کتابخانه دیگری استفاده کردیم که در فضای گوگل کولب به صورت پیش فرض نصب نبود پس ابتدا به نصب آن پرداختیم.

```

!pip install keras-tuner

Collecting keras-tuner
  Downloading keras_tuner-1.4.7-py3-none-any.whl.metadata (5.4 kB)
Requirement already satisfied: keras in /usr/local/lib/python3.10/dist-packages (from keras-tuner) (3.4.1)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages (from keras-tuner) (24.1)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from keras-tuner) (2.32.3)
Collecting kt-legacy (from keras-tuner)
  Downloading kt_legacy-1.0.5-py3-none-any.whl.metadata (221 bytes)
Requirement already satisfied: absolv in /usr/local/lib/python3.10/dist-packages (from keras->keras-tuner) (1.4.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from keras->keras-tuner) (1.26.4)
Requirement already satisfied: rich in /usr/local/lib/python3.10/dist-packages (from keras->keras-tuner) (13.9.3)
Requirement already satisfied: namex in /usr/local/lib/python3.10/dist-packages (from keras->keras-tuner) (0.0.8)
Requirement already satisfied: h5py in /usr/local/lib/python3.10/dist-packages (from keras->keras-tuner) (3.12.1)
Requirement already satisfied: optree in /usr/local/lib/python3.10/dist-packages (from keras->keras-tuner) (0.13.0)
Requirement already satisfied: ml-dtypes in /usr/local/lib/python3.10/dist-packages (from keras->keras-tuner) (0.4.1)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->keras-tuner) (3.4.0)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->keras-tuner) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->keras-tuner) (2.2.3)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->keras-tuner) (2024.8.30)
Requirement already satisfied: typing-extensions>4.5.0 in /usr/local/lib/python3.10/dist-packages (from optree->keras->keras-tuner) (4.12.2)
Requirement already satisfied: markdown-it-py>=2.2.0 in /usr/local/lib/python3.10/dist-packages (from rich->keras->keras-tuner) (3.0.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in /usr/local/lib/python3.10/dist-packages (from rich->keras->keras-tuner) (2.18.0)
Requirement already satisfied: mdurl~0.1 in /usr/local/lib/python3.10/dist-packages (from markdown-it-py>=2.2.0->rich->keras->keras-tuner) (0.1.2)
Downloading keras_tuner-1.4.7-py3-none-any.whl (129 kB)
  129.1/129.1 kB 4.3 MB/s eta 0:00:00
Downloaded kt_legacy-1.0.5-py3-none-any.whl (9.6 kB)
Installing collected packages: kt-legacy, keras-tuner
Successfully installed keras-tuner-1.4.7 kt-legacy-1.0.5

```

شکل ۸۴ - نصب کتابخانه kares-tuner

در نهایت با بررسی ماتریس آشفتگی هر یک از مدل‌ها بررسی کرده ایم که با تغییر هر یک از هایپر پارامترها هر کدام از لیبل‌هایی که با هم اشتباه گرفته می‌شده اند چه تغییری کرده اند و در نهایت کدام ساختار باعث شده است که مدل‌های مشابه کمتر با هم اشتباه گرفته شوند.

ابتدا کلاسی را تعریف کردیم که دیتای تست را به دو بخش validation و تست تقسیم کند.

```

class DataLoader:
    @staticmethod
    def split_validation(x_test, y_test, val_size=5000):
        """Split test data into validation and test sets"""
        x_val = x_test[:val_size]
        y_val = y_test[:val_size]
        x_test = x_test[val_size:]
        y_test = y_test[val_size:]

        return x_val, y_val, x_test, y_test

```

شکل ۸۵ - تعریف کلاس dataloader

```

class ModelBuilder:
    def __init__(self, input_shape, num_classes):
        self.input_shape = input_shape
        self.num_classes = num_classes

    def build_model(self, hp):
        """Build model with hyperparameters"""
        model = models.Sequential([
            # Input layer
            layers.Input(shape=self.input_shape),

            # First hidden layer
            layers.Dense(
                units=hp.Int('units_1', min_value=32, max_value=256, step=32),
                activation='relu'
            ),
            layers.Dropout(hp.Float('dropout_1', 0.1, 0.5, step=0.1)),

            # Second hidden layer
            layers.Dense(
                units=hp.Int('units_2', min_value=32, max_value=256, step=32),
                activation='relu'
            ),
            layers.Dropout(hp.Float('dropout_2', 0.1, 0.5, step=0.1)),

            # Output layer
            layers.Dense(self.num_classes, activation='softmax')
        ])

```

شکل ۸۶ - تعریف کلاس ایجاد مدل

در کلاس تولید مدل لایه‌ها مرز و بازه جستجو برای تعداد نورون و همینطور dropout rate هم تعریف می‌شود. در این بخش دو لایه عمیق مخفی داریم و دو لایه dropout که برای هر دو حداقل و حداقل تعداد نورن ۳۲ و ۲۵۶ و گام حرکتی ۳۲ در نظر گرفته شده است و برای dropout rate هم از ۰.۱ تا ۰.۵ با گام ۰.۱ در نظر گرفته شده است. اکتیویشن فانکشن برای لایه مخفی ReLU و برای لایه خروجی soft max در بخش بعدی صورت گرفته و یک بازه هم برای learning rate ها در نظر گرفته شده است.

```

# Compile model
model.compile(
    optimizer=optimizers.Adam(
        hp.Float('learning_rate', 1e-4, 1e-2, sampling='log')
    ),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

return model

class HyperparameterTuner:
    def __init__(self, model_builder, project_name='model_tuning'):
        self.model_builder = model_builder
        self.project_name = project_name

    def create_tuner(self, max_trials=5):
        """Create random search tuner"""
        return kt.RandomSearch(
            self.model_builder.build_model,
            objective='val_accuracy',
            max_trials=max_trials,
            directory='random_search',
            project_name=self.project_name
        )

```

شکل ۸۷ - کامپایل و تعریف **HyperparameterTuner**

در ادامه عکس فوق کلاسی برای tune کردن هایپرپارامترها تعریف شده است. نکته حائز اهمیت در این بخش تعریف val_accuracy به عنوان تابع هدف است چون این پارامتر از اهمیت ویژه‌ای برخوردار است.

```

def find_best_model(self, x_train, y_train, x_val, y_val, max_trials=5):
    """Find best model using random search"""
    # Create tuner and callbacks
    tuner = self.create_tuner(max_trials)
    callbacks = self.create_callbacks()

    # Perform search
    print("Starting random search...")
    tuner.search(
        x_train, y_train,
        validation_data=(x_val, y_val),
        epochs=10,
        batch_size=128,
        callbacks=callbacks
    )

    # Get best model and hyperparameters
    best_model = tuner.get_best_models(1)[0]
    best_hp = tuner.get_best_hyperparameters(1)[0]

    return best_model, best_hp

```

شکل ۸۸ - تعریف تابع **fine_best_model**

در بخش فوق تابعی تعریف شده است تا بهترین مدل را انتخاب کند. بخش اصلی کار شروع فرایند سرچ است که می‌بینیم از ۱۰ ایپاک برای هر گام و ۱۲۸ batch size استفاده شده است. در نهایت مدل بهترین را انتخاب و گزارش خواهد کرد.

```

# Load data
print("Loading MNIST dataset...")
data_loader = DataLoader()
(x_train, y_train), (x_test, y_test) = load_and_preprocess_data()
x_val, y_val, x_test, y_test = data_loader.split_validation(x_test, y_test)

# Create model builder
model_builder = ModelBuilder(input_shape=(784,), num_classes=10)

# Create and run tuner
tuner = HyperparameterTuner(model_builder, project_name='mnist_tuning')
best_model, best_hp = tuner.find_best_model(
    x_train, y_train,
    x_val, y_val,
    max_trials=5
)

# Print results
tuner.print_best_hyperparameters(best_hp)

# Evaluate best model
print("\nEvaluating best model on test set...")
test_loss, test_accuracy = best_model.evaluate(x_test, y_test, verbose=0)
print(f"Test accuracy: {test_accuracy:.4f}")

# Save best model
best_model.save('best_mnist_model.h5')
print("\nBest model saved as 'best_mnist_model.h5'")

```

شکل ۸۹ - اجرای random search برای هایپرپارامترها

ما پنج تلاش یا trial را برای سرج انتخاب کرده ایم اما برای دقت و کار بیشتر می‌توان این عدد را بیشتر کرد.

```
Trial 5 Complete [00h 00m 30s]
val_accuracy: 0.8601999878883362

Best val_accuracy So Far: 0.876800000667572
Total elapsed time: 00h 03m 33s
/usr/local/lib/python3.10/dist-packages/keras/src/saving/saving_lib.py:576: UserWarning: Skipping variable loa
    saveable.load_own_variables(weights_store.get(inner_path))

Best Hyperparameters:
Learning rate: 0.003993
Units in first layer: 96
Units in second layer: 64
Dropout rate 1: 0.10
Dropout rate 2: 0.10

Evaluating best model on test set...
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`.
Test accuracy: 0.8778

Best model saved as 'best_mnist_model.h5'
```

شکل ۹۰ - نتیجه Random Search

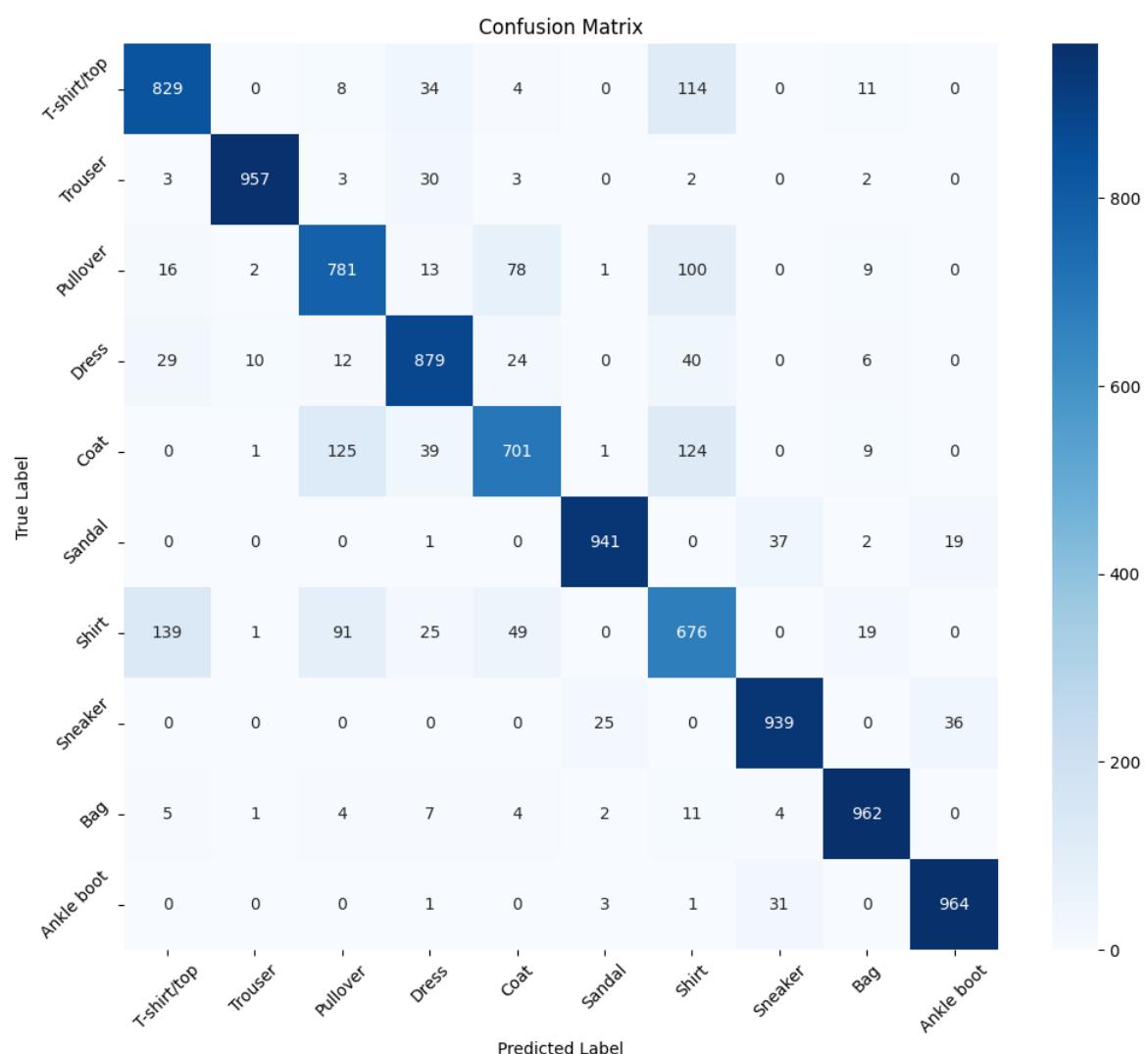
در نهایت آنچه که این جستجو به عنوان هایپر پارامتر به ما داد به معماری زیر است.

dropout rate = 0.003993 و تعداد نورون لایه اول ۹۶، تعداد نورون لایه دوم ۶۴، Learning rate = 0.003993 برای هر دو لایه. در نهایت دقت این معماری برای شبکه برابر با عدد ۸۷.۷۸ درصد شد. این عدد عدد بسیار خوبی است در شرایط حضور dropout.

۲-۴-۱- بررسی ماتریس آشفتگی با تغییر هایپرپارامترها

برای بررسی این بخش در انتهای ران شدن هر مدل ماتریس آشفتگی را هم ترسیم کردم و بررسی کردیم کدام کلاس ها بیشتر با هم اشتباہ گرفته می شوند. با این کار و مقایسه مدل های متفاوت با معماری و هایپرپارامترهای متفاوت می توانیم تاثیر تغییر هایپرپارامترها را بررسی کنیم.

ابتدا به بررسی تغییر learning rate و اثر آن در ماتریس آشفتگی می پردازیم.



Class 0 is most often (114 times) confused with Class 6.
 Class 1 is most often (30 times) confused with Class 3.
 Class 2 is most often (100 times) confused with Class 6.
 Class 3 is most often (40 times) confused with Class 6.
 Class 4 is most often (125 times) confused with Class 2.
 Class 5 is most often (37 times) confused with Class 7.
 Class 6 is most often (139 times) confused with Class 0.
 Class 7 is most often (36 times) confused with Class 9.
 Class 8 is most often (11 times) confused with Class 6.
 Class 9 is most often (31 times) confused with Class 7.

The two classes most confused with each other are: (0, 6)
 The two classes most confused with each : 253 times.

شكل ۹۱ - ماتریس آشفتگی مدل یک با نرخ یادگیری ۰.۰۰۰۱

Confusion Matrix										
True Label	T-shirt/top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot
	884	1	13	9	6	0	83	0	4	0
	7	963	0	20	6	0	2	0	1	1
	23	0	735	6	169	0	67	0	0	0
	46	2	12	836	71	0	29	0	3	1
	1	1	63	11	892	0	31	0	1	0
	0	0	0	0	0	957	0	24	1	18
	151	2	70	24	112	0	635	0	6	0
	0	0	0	0	0	14	0	958	1	27
	4	0	3	1	8	1	5	4	974	0
	0	0	0	0	0	8	1	39	0	952

Class 0 is most often (83 times)confused with Class 6.
Class 1 is most often (20 times)confused with Class 3.
Class 2 is most often (169 times)confused with Class 4.
Class 3 is most often (71 times)confused with Class 4.
Class 4 is most often (63 times)confused with Class 2.
Class 5 is most often (24 times)confused with Class 7.
Class 6 is most often (151 times)confused with Class 0.
Class 7 is most often (27 times)confused with Class 9.
Class 8 is most often (8 times)confused with Class 4.
Class 9 is most often (39 times)confused with Class 7.

The two classes most confused with each other are: (0, 6)
The two classes most confused with each : 234 times.

شکل ۹۲ - ماتریس آشفتگی مدل یک با نرخ یادگیری ۰.۰۰۱

Confusion Matrix										
True Label	T-shirt/top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot
	800	3	22	36	0	0	132	0	7	0
	3	964	1	23	3	0	4	0	2	0
	15	2	727	13	110	0	133	0	0	0
	27	13	2	888	13	0	52	0	5	0
	1	2	95	55	673	0	171	0	3	0
	0	0	0	0	0	966	1	14	0	19
	119	1	97	34	41	0	698	0	10	0
	0	0	0	0	0	67	0	890	0	43
	6	1	2	5	3	12	16	3	952	0

Class 0 is most often (132 times) confused with Class 6.
 Class 1 is most often (23 times) confused with Class 3.
 Class 2 is most often (133 times) confused with Class 6.
 Class 3 is most often (52 times) confused with Class 6.
 Class 4 is most often (171 times) confused with Class 6.
 Class 5 is most often (19 times) confused with Class 9.
 Class 6 is most often (119 times) confused with Class 0.
 Class 7 is most often (67 times) confused with Class 5.
 Class 8 is most often (16 times) confused with Class 6.
 Class 9 is most often (23 times) confused with Class 5.

The two classes most confused with each other are: (0, 6)
 The two classes most confused with each : 251 times.

شكل ٩٣ - ماتریس آشفتگی مدل یک با نرخ یادگیری ۱۰۰%

جدول ۹ - اثر نرخ یادگیری در ماتریس آشфтگی

کلاس هایی که بیشترین تعداد بار اشتباه شباخت	بیشترین شباخت گرفته شدن	دو کلاس با هم باهم را داشته اند	Model 1
۲۵۳	۶ و ۰	۰۰۰۰۱	
۲۳۴	۶ و ۰	۰۰۰۱	
۲۵۱	۶ و ۰	۰۰۱	

نرخ یادگیری مناسب، نقش کلیدی در بهبود عملکرد مدل و کاهش اشتباهات در ماتریس آشфтگی دارد، زیرا در بهروزرسانی وزن‌ها طی فرآیند آموزش تأثیر مستقیم دارد. وقتی نرخ یادگیری بسیار پایین است (مثل ۰۰۰۰۱)، تغییرات وزن در هر تکرار بسیار کم است و همگرایی مدل به کندی اتفاق می‌افتد و ممکن است در کمینه‌های محلی گیر کند و نتواند به کمینه بهینه برسد. این باعث افزایش اشتباهات در ماتریس آشфтگی می‌شود، زیرا مدل تفاوت‌های بین کلاس‌ها را به خوبی یاد نمی‌گیرد. از طرف دیگر، نرخ یادگیری بالا (مثل ۰۰۱) باعث می‌شود که تغییرات وزن‌ها بیش از حد بزرگ باشد و مدل از روی کمینه‌های مناسب بپردازد و نتواند به همگرایی برسد؛ این مسئله باعث نوسانات شدید در فرآیند یادگیری و افزایش اشتباهات بین کلاس‌ها می‌شود، چراکه مدل نمی‌تواند الگوهای تمایزدهنده هر کلاس را به درستی بیاموزد. اما نرخ یادگیری متوسط ۰۰۰۱ تعادل مناسبی بین سرعت همگرایی و دقت یادگیری فراهم می‌کند و به مدل امکان می‌دهد تفاوت‌های پیچیده بین کلاس‌ها را بهتر درک کند. این نرخ یادگیری بهینه باعث کاهش اشتباهات در ماتریس آشфтگی می‌شود، زیرا مدل با بهروزرسانی‌های پیوسته و مناسب می‌تواند مرزهای تصمیم‌گیری بهتری بین کلاس‌ها ایجاد کند و آن‌ها را با دقت بیشتری از هم تفکیک نماید. در شکل‌های فوق هم می‌بینیم که کلاس‌هایی که بیشتر با هم اشتباه گرفته می‌شوند یعنی کلاس شماره صفر و شش که همان کلاس‌های shirt و کلاس T-shirt/top هستند با نرخ یادگیری ۰۰۰۱ که بهترین نرخ یادگیری در این بین است کمترین میزان اشتباه گرفته شدن با هم را دارند که عدد ۲۳۴ را نشان می‌دهد.

در گام بعد بررسی اثر تعداد نورون و اثر حضور و عدم حضور لایه dropout را مقایسه می‌کنیم.

Confusion Matrix												
True Label	T-shirt/top	12	25	30	4	2	109	0	3	0		
	Trouser	815	984	1	9	2	0	2	0	1	0	- 800
	Pullover	1	14	2	783	13	102	0	84	0	2	0
	Dress	984	8	35	16	873	42	0	23	0	3	0
	Coat	1	0	2	85	23	816	0	73	0	1	0
	Sandal	2	0	0	0	1	0	962	0	20	0	17
	Shirt	0	89	8	82	28	59	0	727	0	7	0
	Sneaker	0	0	0	0	0	21	0	958	0	21	- 200
	Bag	0	7	0	8	5	7	2	8	4	959	0
	Ankle boot	0	0	0	1	0	7	1	33	0	958	
Predicted Label												

Class 0 is most often (109 times) confused with Class 6.
 Class 1 is most often (9 times) confused with Class 3.
 Class 2 is most often (102 times) confused with Class 4.
 Class 3 is most often (42 times) confused with Class 4.
 Class 4 is most often (85 times) confused with Class 2.
 Class 5 is most often (20 times) confused with Class 7.
 Class 6 is most often (89 times) confused with Class 0.
 Class 7 is most often (21 times) confused with Class 5.
 Class 8 is most often (8 times) confused with Class 2.
 Class 9 is most often (33 times) confused with Class 7.

The two classes most confused with each other are: (0, 6)
 The two classes most confused with each : 198 times.

شکل ۹۴ - ماتریس آشفتگی مدل یک

		Confusion Matrix										
		T-shirt/top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot	
True Label	T-shirt/top	805	1	10	27	3	2	139	0	13	0	- 800
	Trouser	2	970	1	20	4	0	1	0	2	0	- 600
	Pullover	9	0	778	11	106	0	87	0	9	0	- 400
	Dress	25	4	14	896	25	0	31	0	5	0	- 200
	Coat	0	0	104	44	787	0	63	0	2	0	- 0
	Sandal	0	0	0	1	0	968	0	19	0	12	
	Shirt	111	0	89	28	78	0	673	0	21	0	
	Sneaker	0	0	0	0	0	13	0	960	0	27	
	Bag	2	0	3	5	4	1	4	4	977	0	
	Ankle boot	0	0	0	0	0	9	1	44	0	946	

Class 0 is most often (139 times) confused with Class 6.
 Class 1 is most often (20 times) confused with Class 3.
 Class 2 is most often (106 times) confused with Class 4.
 Class 3 is most often (31 times) confused with Class 6.
 Class 4 is most often (104 times) confused with Class 2.
 Class 5 is most often (19 times) confused with Class 7.
 Class 6 is most often (111 times) confused with Class 0.
 Class 7 is most often (27 times) confused with Class 9.
 Class 8 is most often (5 times) confused with Class 3.
 Class 9 is most often (44 times) confused with Class 7.

The two classes most confused with each other are: (0, 6)
 The two classes most confused with each : 250 times.

شكل ٩٥ - نتائج ماتریس آشفتگی مدل دو

Confusion Matrix											
True Label	T-shirt/top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot	
	873	0	9	27	1	2	81	0	7	0	800
	7	963	3	20	3	0	3	0	1	0	600
	19	1	765	9	90	0	115	0	1	0	400
	28	6	5	904	20	0	36	0	1	0	200
	1	0	100	45	734	0	119	0	1	0	0
	0	0	0	0	0	968	0	18	0	14	
	186	0	74	33	49	0	653	0	5	0	
	0	0	0	0	0	26	0	933	0	41	
	7	0	6	2	8	3	25	5	944	0	

Class 0 is most often (81 times) confused with Class 6.
 Class 1 is most often (20 times) confused with Class 3.
 Class 2 is most often (115 times) confused with Class 6.
 Class 3 is most often (36 times) confused with Class 6.
 Class 4 is most often (119 times) confused with Class 6.
 Class 5 is most often (18 times) confused with Class 7.
 Class 6 is most often (186 times) confused with Class 0.
 Class 7 is most often (41 times) confused with Class 9.
 Class 8 is most often (25 times) confused with Class 6.
 Class 9 is most often (31 times) confused with Class 7.

The two classes most confused with each other are: (0, 6)
 The two classes most confused with each : 267 times.

شکل ۹۶ - نتایج ماتریس آشفتگی مدل سه

Confusion Matrix										
True Label	T-shirt/top	0	29	17	8	0	141	1	5	0
	Trouser	4	963	0	24	3	0	5	1	0
	Pullover	16	1	759	7	177	0	40	0	0
	Dress	33	3	17	838	80	0	26	0	0
	Coat	0	0	65	13	890	0	32	0	0
	Sandal	0	0	0	0	0	944	0	26	1
	Shirt	130	0	110	21	117	0	613	0	9
	Sneaker	0	0	0	0	0	16	0	919	0
	Bag	5	0	19	2	8	3	8	3	951
	Ankle boot	1	0	0	0	0	2	0	17	0
Predicted Label										

Class 0 is most often (141 times) confused with Class 6.
 Class 1 is most often (24 times) confused with Class 3.
 Class 2 is most often (177 times) confused with Class 4.
 Class 3 is most often (80 times) confused with Class 4.
 Class 4 is most often (65 times) confused with Class 2.
 Class 5 is most often (29 times) confused with Class 9.
 Class 6 is most often (130 times) confused with Class 0.
 Class 7 is most often (65 times) confused with Class 9.
 Class 8 is most often (19 times) confused with Class 2.
 Class 9 is most often (17 times) confused with Class 7.

The two classes most confused with each other are: (0, 6)
 The two classes most confused with each : 271 times.

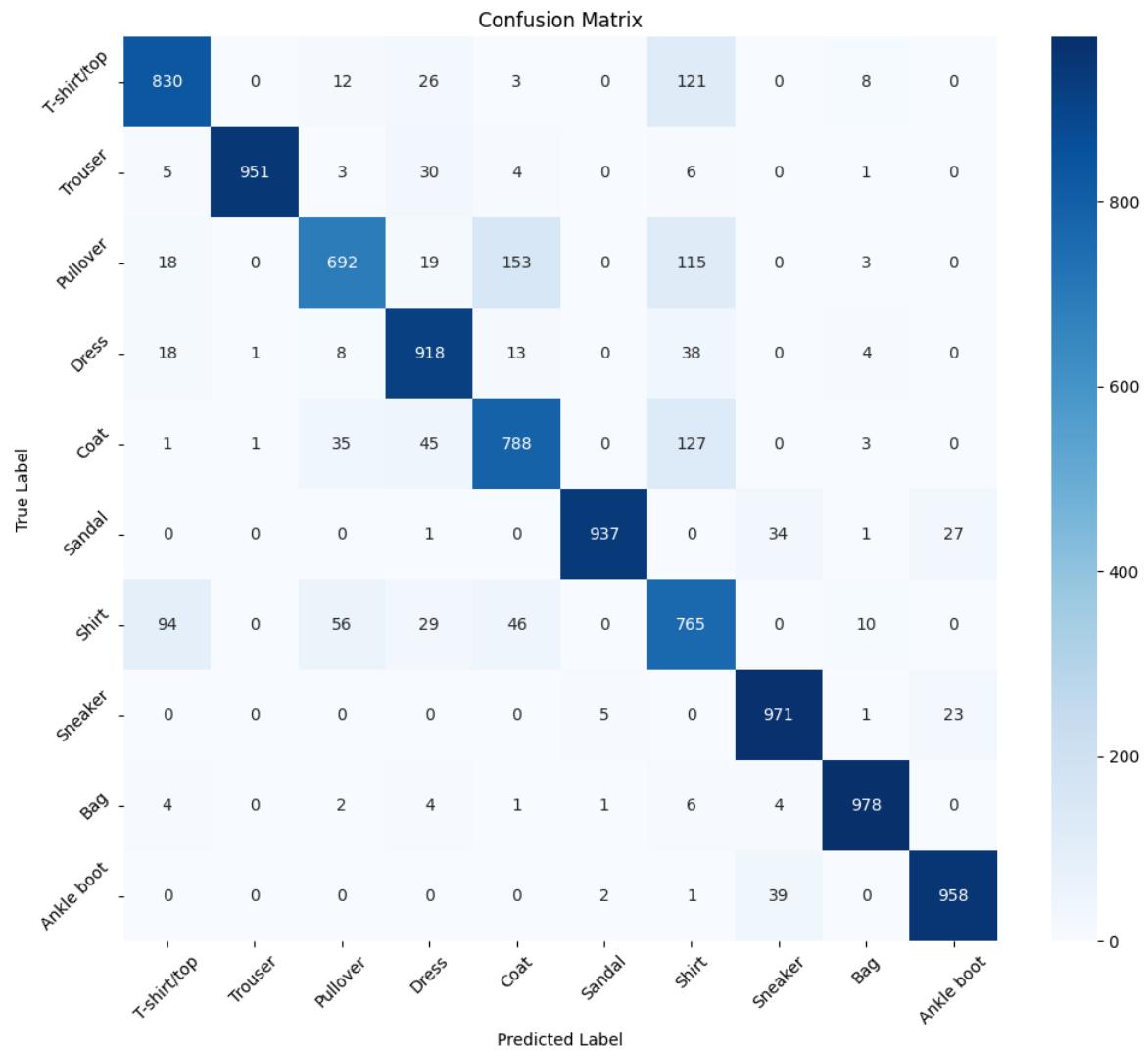
شکل ۹۷ - نتایج ماتریس آشفتگی مدل چهار

		Confusion Matrix										
		T-shirt/top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot	
True Label	T-shirt/top	792	0	12	42	9	0	137	2	6	0	- 800
	Trouser	1	975	0	16	4	0	4	0	0	0	- 600
	Pullover	11	0	832	12	90	0	55	0	0	0	- 400
	Dress	18	5	12	891	36	0	35	0	3	0	- 200
	Coat	1	1	105	28	826	0	38	0	1	0	- 0
	Sandal	0	0	0	1	0	956	0	28	0	15	- 0
	Shirt	90	3	101	33	96	1	671	0	5	0	- 0
	Sneaker	0	0	0	0	0	11	0	978	0	11	- 0
	Bag	4	0	6	6	5	5	5	4	965	0	- 0
	Ankle boot	0	0	0	1	0	7	1	52	0	939	- 0

Class 0 is most often (137 times) confused with Class 6.
 Class 1 is most often (16 times) confused with Class 3.
 Class 2 is most often (90 times) confused with Class 4.
 Class 3 is most often (36 times) confused with Class 4.
 Class 4 is most often (105 times) confused with Class 2.
 Class 5 is most often (28 times) confused with Class 7.
 Class 6 is most often (101 times) confused with Class 2.
 Class 7 is most often (11 times) confused with Class 5.
 Class 8 is most often (6 times) confused with Class 2.
 Class 9 is most often (52 times) confused with Class 7.

The two classes most confused with each other are: (0, 6)
 The two classes most confused with each : 227 times.

شكل ٩٨ - نتایج ماتریس آشنتگی مدل پنج



Class 0 is most often (121 times) confused with Class 6.
 Class 1 is most often (30 times) confused with Class 3.
 Class 2 is most often (153 times) confused with Class 4.
 Class 3 is most often (38 times) confused with Class 6.
 Class 4 is most often (127 times) confused with Class 6.
 Class 5 is most often (34 times) confused with Class 7.
 Class 6 is most often (94 times) confused with Class 0.
 Class 7 is most often (23 times) confused with Class 9.
 Class 8 is most often (6 times) confused with Class 6.
 Class 9 is most often (39 times) confused with Class 7.

The two classes most confused with each other are: (0, 6)
 The two classes most confused with each : 215 times.

شکل ۹۹ - نتایج ماتریس آشفتگی مدل شش

Confusion Matrix										
True Label	T-shirt/top	0	13	38	6	0	145	0	9	0
	Trouser	2	954	1	37	3	0	3	0	0
	Pullover	16	0	770	15	123	1	75	0	0
	Dress	9	1	7	922	33	0	25	0	0
	Coat	0	1	64	36	841	0	58	0	0
	Sandal	0	0	0	1	0	960	0	20	2
	Shirt	89	0	78	37	66	0	725	0	5
	Sneaker	0	0	0	0	0	8	0	957	2
	Bag	3	0	4	6	5	3	3	6	970
	Ankle boot	0	0	0	1	0	5	1	27	0
Predicted Label										

Class 0 is most often (145 times) confused with Class 6.
 Class 1 is most often (37 times) confused with Class 3.
 Class 2 is most often (123 times) confused with Class 4.
 Class 3 is most often (33 times) confused with Class 4.
 Class 4 is most often (64 times) confused with Class 2.
 Class 5 is most often (20 times) confused with Class 7.
 Class 6 is most often (89 times) confused with Class 0.
 Class 7 is most often (33 times) confused with Class 9.
 Class 8 is most often (6 times) confused with Class 3.
 Class 9 is most often (27 times) confused with Class 7.

The two classes most confused with each other are: (0, 6)
 The two classes most confused with each : 234 times.

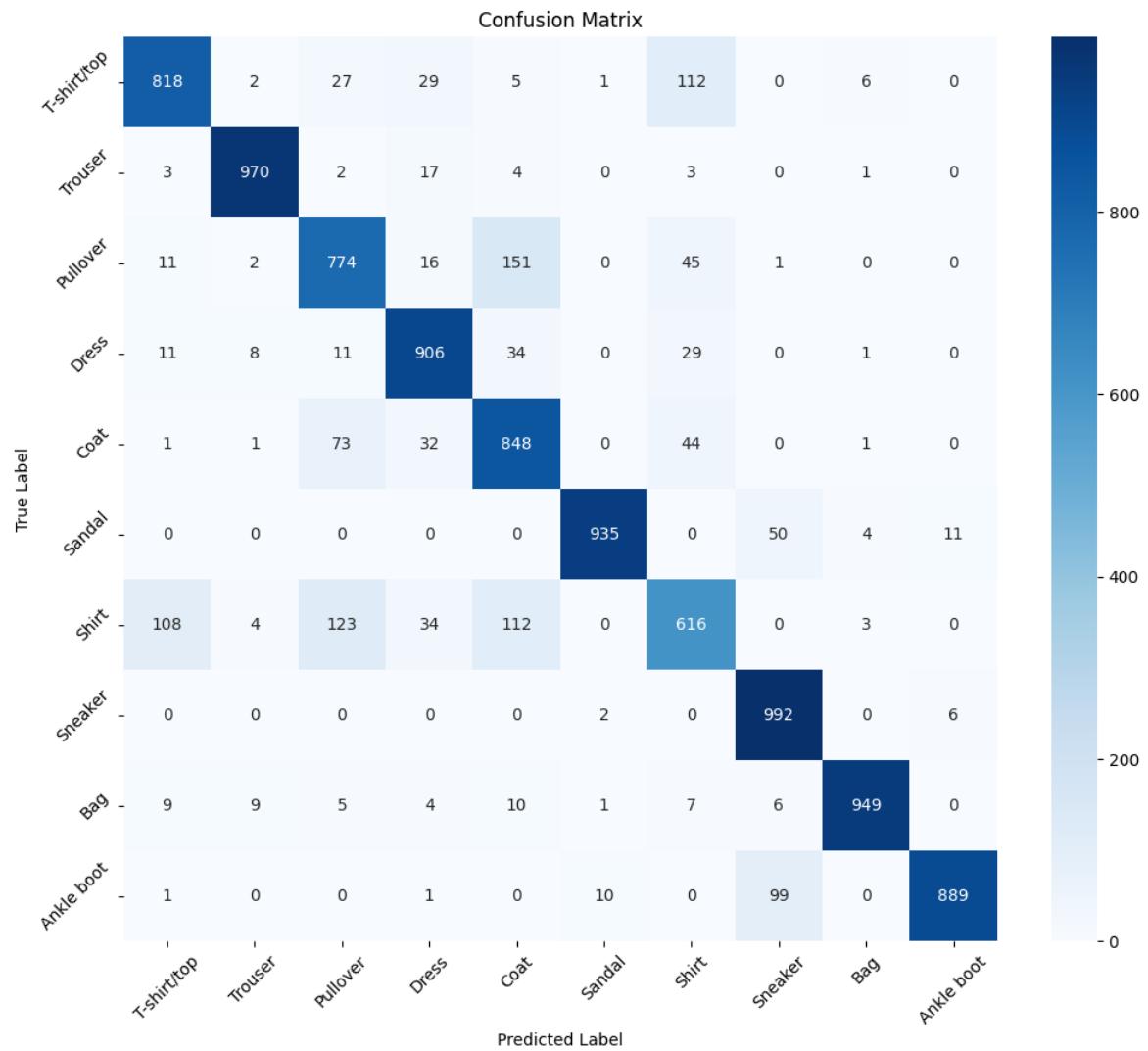
شكل ١٠٠ - نتایج ماتریس آشتفتگی مدل هفت

جدول ۱۰- اثر تعداد نورون و Dropout در ماتریس آشفتگی

کلاس هایی که بیشترین تعداد بار اشتباه شباخت	بیشترین شباخت باهم را داشته اند	
۱۹۸	۶ و ۰	Model 1
۲۵۰	۶ و ۰	Model 2
۲۶۸	۶ و ۰	Model 3
۲۷۱	۶ و ۰	Model 4
۲۲۷	۶ و ۰	Model 5
۲۱۵	۶ و ۰	Model 6
۲۳۴	۶ و ۰	Model 7

بررسی نتایج مدل‌ها با توجه به تعداد نورون‌ها و استفاده از دراپ‌اوت نشان می‌دهد که مدل ۱، با ۱۲۸ نورون در لایه اول و بدون دراپ‌اوت، بهترین عملکرد را داشته است، زیرا کمترین تعداد اشتباهات بین کلاس‌های مشابه ۰ و ۶ (تعداد ۱۹۸ بار اشتباه) را نشان می‌دهد. مدل‌هایی که تعداد نورون‌های بیشتری دارند (مثل مدل ۳ و ۴) یا از دراپ‌اوت استفاده می‌کنند، دچار افزایش اشتباهات بین این دو کلاس شده‌اند. دلیل این موضوع این است که مدل ۱ با تعداد نورون‌های متعادل و بدون استفاده از دراپ‌اوت، تعادل مناسبی بین یادگیری ویژگی‌های اساسی داده‌ها و جلوگیری از پیچیدگی‌های غیرضروری ایجاد می‌کند، در حالی که مدل‌های با نورون‌های بیشتر یا دراپ‌اوت ممکن است با نوساناتی در یادگیری مواجه شوند یا نتوانند به همان دقت در تمایز کلاس‌های مشابه برسند.

به عنوان سومین آیتم تغییری به سراغ تغییر در تعداد لایه‌های مخفی رفتیم و در این بخش هم ماتریس آشفتگی هر مدل را گزارش و نتایج را بررسی می‌کنیم.



Class 0 is most often (112 times) confused with Class 6.
 Class 1 is most often (17 times) confused with Class 3.
 Class 2 is most often (151 times) confused with Class 4.
 Class 3 is most often (34 times) confused with Class 4.
 Class 4 is most often (73 times) confused with Class 2.
 Class 5 is most often (50 times) confused with Class 7.
 Class 6 is most often (123 times) confused with Class 2.
 Class 7 is most often (6 times) confused with Class 9.
 Class 8 is most often (10 times) confused with Class 4.
 Class 9 is most often (99 times) confused with Class 7.

The two classes most confused with each other are: (2, 4)
 The two classes most confused with each : 224 times.

شکل ۱۰۱ - نتایج ماتریس آشناگی مدل هشت

Confusion Matrix										
True Label	T-shirt/top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot
	919	2	12	19	3	1	35	0	9	0
	8	966	0	20	4	0	2	0	0	0
	26	0	862	15	56	1	36	0	4	0
	40	5	14	893	33	0	10	0	5	0
	4	1	154	32	782	0	24	0	3	0
	0	0	0	0	0	956	0	17	2	25
	237	2	125	29	106	0	486	0	15	0
	0	0	0	0	0	16	0	938	4	42
	4	0	4	6	4	2	1	4	975	0
	0	0	0	0	0	8	1	22	0	969

Class 0 is most often (35 times)confused with Class 6.
Class 1 is most often (20 times)confused with Class 3.
Class 2 is most often (56 times)confused with Class 4.
Class 3 is most often (40 times)confused with Class 0.
Class 4 is most often (154 times)confused with Class 2.
Class 5 is most often (25 times)confused with Class 9.
Class 6 is most often (237 times)confused with Class 0.
Class 7 is most often (42 times)confused with Class 9.
Class 8 is most often (6 times)confused with Class 3.
Class 9 is most often (22 times)confused with Class 7.

The two classes most confused with each other are: (0, 6)
The two classes most confused with each : 272 times.

شكل ١٠٢ - نتایج ماتریس آشفتگی مدل نه

		Confusion Matrix										
		T-shirt/top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot	
True Label	T-shirt/top	747	0	18	63	2	1	148	0	21	0	- 800
	Trouser	3	954	4	31	3	0	3	0	2	0	- 600
	Pullover	13	0	820	14	105	1	43	0	4	0	- 400
	Dress	12	4	12	903	27	1	30	0	11	0	- 200
	Coat	1	1	115	40	795	0	38	0	10	0	- 0
	Sandal	0	1	0	0	0	949	0	26	1	23	- 800
	Shirt	88	1	130	45	84	1	624	0	27	0	- 600
	Sneaker	0	0	0	0	0	26	0	961	0	13	- 400
	Bag	0	1	3	5	1	5	2	6	977	0	- 200
	Ankle boot	0	0	0	0	0	8	0	60	1	931	- 0

Class 0 is most often (148 times) confused with Class 6.
 Class 1 is most often (31 times) confused with Class 3.
 Class 2 is most often (105 times) confused with Class 4.
 Class 3 is most often (30 times) confused with Class 6.
 Class 4 is most often (115 times) confused with Class 2.
 Class 5 is most often (26 times) confused with Class 7.
 Class 6 is most often (130 times) confused with Class 2.
 Class 7 is most often (26 times) confused with Class 5.
 Class 8 is most often (6 times) confused with Class 7.
 Class 9 is most often (60 times) confused with Class 7.

The two classes most confused with each other are: (0, 6)
 The two classes most confused with each : 236 times.

شكل ١٠٣ - نتایج ماتریس آشفتگی مدل ده

Confusion Matrix										
True Label	T-shirt/top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot
	790	2	38	31	1	1	127	1	9	0
	2	965	6	19	4	0	4	0	0	0
	12	0	911	8	43	0	25	0	1	0
	15	11	18	879	41	0	32	0	4	0
	0	1	285	26	641	0	46	0	1	0
	0	0	0	1	0	947	0	30	0	22
	97	1	205	26	57	1	604	0	9	0
	0	0	0	0	0	22	0	930	0	48
	1	0	28	3	2	14	9	5	938	0

Class 0 is most often (127 times) confused with Class 6.
 Class 1 is most often (19 times) confused with Class 3.
 Class 2 is most often (43 times) confused with Class 4.
 Class 3 is most often (41 times) confused with Class 4.
 Class 4 is most often (285 times) confused with Class 2.
 Class 5 is most often (30 times) confused with Class 7.
 Class 6 is most often (205 times) confused with Class 2.
 Class 7 is most often (48 times) confused with Class 9.
 Class 8 is most often (28 times) confused with Class 2.
 Class 9 is most often (33 times) confused with Class 7.

The two classes most confused with each other are: (2, 4)
 The two classes most confused with each : 328 times.

شکل ۱۰۴ - نتایج ماتریس آشتگی مدل یازده

جدول ۱۱- اثر تعداد لایه‌های مخفی در ماتریس آشفتگی

کلاس هایی که بیشترین تعداد بار اشتباه شباخت	بیشترین گرفته شدن باهم را داشته اند	
۲۲۴	۰ و ۶	Model 8
۲۷۲	۰ و ۶	Model 9
۲۳۶	۰ و ۶	Model 10
۳۲۸	۰ و ۶	Model 11

نتایج نشان می‌دهد که تعداد لایه‌ها، تعداد نورون‌ها، و استفاده یا عدم استفاده از دراپ‌اوت تأثیر قابل توجهی بر دقت مدل و میزان خطا در ماتریس آشفتگی دارند. بهویژه، مدل ۸ با پیکربندی سه لایه و ۱۲۸، ۱۲۴ و ۳۲ نورون (بدون دراپ‌اوت) بهترین عملکرد را داشته و تعداد اشتباها بین کلاس‌های ۰ و ۶ (که بیشترین شباهت را با هم دارند) در این مدل پایین‌تر بوده است. دلیل این عملکرد بهتر این است که مدل ۸، با تعداد متناسبی از نورون‌ها، قادر است ویژگی‌های مهم داده را بدون پیچیدگی اضافی استخراج کند. در مدل‌های پیچیده‌تر، مانند مدل ۱۱ که دارای تعداد نورون‌های بیشتری در سه لایه و دراپ‌اوت است، بهدلیل تعداد بالای نورون‌ها و اعمال دراپ‌اوت، مدل در تمایز بین کلاس‌ها (بهویژه کلاس‌های مشابه مانند ۰ و ۶) دچار مشکل شده است و دقت کاهش یافته و اشتباها بین این دو کلاس افزایش یافته است.

پرشن ۲ - آموزش و ارزیابی یک شبکه عصبی ساده

۱-۲. آموزش یک شبکه عصبی

برای این بخش همانطور که گفته شد از معادلات موجود در کتاب Bishop استفاده شده است. یک تابع فقط برای نمایش معادلات موجود نوشته شده است تا رفع ابهام کاملی باشد بر معادلات استفاده شده. در کل کد فقط از numpy برای بخش محاسبات و در نهایت از matplotlib برای نمایش اطلاعات در صورت نیاز استفاده شده است.

Bishop's Backpropagation Equations Breakdown:

1. Forward Propagation:

```
zj = Σ(wi,j * xi) # Weighted sum for hidden layer  
aj = tanh(zj)      # Hidden layer activation (hyperbolic tangent)  
zk = Σ(wj,k * aj) # Weighted sum for output layer  
yk = zk            # Linear output
```

2. Error Calculation:

```
E = 1/2 * Σ(tk - yk)^2 # Squared error loss
```

3. Gradient Calculations:

```
δk = yk - tk          # Output layer error  
δj = (1 - aj^2) * Σ(wj,k * δk) # Hidden layer error
```

4. Weight Update Rules:

```
Δwj,k = -η * δk * aj    # Output layer weight update  
Δwi,j = -η * δj * xi    # Hidden layer weight update
```

5. Key Parameters:

```
η (eta) = Learning Rate  
tk = Target value  
yk = Predicted value  
aj = Hidden layer activations  
xi = Input features
```

شکل ۱۰۵ - معادلات بکار رفته در این سوال با توجه به کتاب بیشап [۴]

معادلات نوشته شده همگی از فصل پنجم و بخش ۳-۵ نسخه سال ۲۰۱۶ استخراج شده اند.

نکته مهم دیگر در این سوال استفاده از activation function تانژانت هایپربولیک برای لایه اول مخفی و استفاده از identity function در لایه خروجی است. بدین معنا که در لایه خروجی خود نتیجه را نمایش می‌دهیم.

الف) برای نوشته تابع forward به صورت زیر عمل می‌کنیم. ساختار بسیار ساده است. مقادیر ورودی ما ابعاد $N \times D$ دارند که N تعداد نورون‌های ورودی و D تعداد feature‌های ما است. در ادامه W1 دارای ابعاد $D \times M$ است که M تعداد نورون‌های لایه مخفی اولیه ما است و در نهایت W2 ماتریس وزن‌های لایه نهایی ما است که به ما خروجی منتهی می‌شود که دارای ابعاد $M \times 1$ است. خروجی ما هم مطابق صورت سوال خواهد بود.

کد این بخش به صورت زیر خواهد بود.

```
def forward(X, W1, W2):
    """
    Perform forward pass through a neural network with a hidden layer.

    Parameters:
    X (numpy.ndarray): Input features matrix of shape (N, D)
    W1 (numpy.ndarray): Weight matrix between input and hidden layer of shape (D, M)
    W2 (numpy.ndarray): Weight matrix between hidden and output layer of shape (M, 1)

    Returns:
    tuple: (y_pred, Z)
        y_pred (numpy.ndarray): Predicted outputs of shape (N, 1)
        Z (numpy.ndarray): Hidden layer activations of shape (N, M)
    """
    # Number of training samples
    N = X.shape[0]

    # Calculate hidden layer activations using hyperbolic tangent
    Z = np.tanh(np.dot(X, W1))

    # Calculate output layer (linear/identity activation)
    y_pred = np.dot(Z, W2)

    return y_pred, Z
```

شکل ۱۰۶ - تابع forward

در این تابع ابتدا تعداد داده‌هارا مشخص کرده ایم سپس با ضرب داخلی X و W1 و سپس قرار دادن آن در تابع فعالساز tanh به مقدار Z می‌رسیم. در نهایت با ضرب داخلی Z و W2 که وزن لایه اخراست به مقدار Y_pred خواهیم رسید. لازم به ذکر است اگر در لایه خروجی هم یک تابع activation داشتیم این مقدار میباشد ابتدا در آن قرار می‌گرفت، اما در این حالت مستقیماً حاصل این ضرب داخلی پیش‌بینی مدل را به ما خواهد داد. در نهایت خروجی‌های مطرح شده در صورت سوال را به بازگردانده ایم. این تابع از این جهت کاربرد دارد که با پیش فرض دانستن مقادیر وزن لایه‌ها و همینطور X به پیش‌بینی مدل دست پیدا خواهیم کرد. اینکه چگونه وزن نورون‌ها در شبکه تعیین می‌شود با توجه به تابع backward تعریف خواهد شد که در بخش بعد به تشریح آن خواهیم پرداخت.

ب) در این بخش که کمی پیچیده‌تر از بخش قبل است و روند کار به این صورت است که ابتدا مقادیر فرضی برای وزن‌ها در نظر گرفته می‌شود و از تابع forward برای محاسبه z و y_{pred} استفاده کرده‌ایم. در مرحله بعد وارد یک حلقه می‌شویم و به ازای هر گام، ابتدا ترم تغییرات (گرادیان) وزن برای لایه اخیر را محاسبه می‌کنیم و سپس از این عبارت استفاده کرده و وزن لایه قبل را آپدیت می‌کنیم. لازم به ذکر است در صورت حضور لایه‌های بیشتر هم روند کار به همین صورت خواهد بود. به صورت خلاصه روند آپدیت کردن وزن‌ها از اخر به اول خواهد بود.

```

def backward_1(X, y, M, iters, lr):
    """
    Train neural network using backpropagation.

    Parameters:
    X (numpy.ndarray): Input features matrix of shape (N, D)
    y (numpy.ndarray): True labels of shape (N, 1)
    M (int): Number of neurons in hidden layer
    iters (int): Number of training iterations
    lr (float): Learning rate

    Returns:
    tuple: (W1, W2, error_over_time)
        W1: Updated weight matrix for input to hidden layer
        W2: Updated weight matrix for hidden to output layer
        error_over_time: Array of errors for each iteration
    """
    # Get input dimensions
    N, D = X.shape

    # Initialize weights randomly with small values
    np.random.seed(42)
    W1 = np.random.randn(D, M) * 0.01
    W2 = np.random.randn(M, 1) * 0.01

    # Initialize error tracking array
    error_over_time = np.zeros(iters)

```

شكل ١٠٧ - تابع **backward** - بخش اول

```

# Training loop
for t in range(iters):
    # Forward pass
    y_pred, Z = forward(X, W1, W2)

    # Compute error (squared error)
    error = y_pred - y
    # error_over_time[t] = np.mean(error**2) # this is MSE
    error_over_time[t] = np.sum(error**2) # this is Squared Error

    # Backward propagation
    # Derivative of output layer (squared error loss)
    dL_dypred = 2 * error

    # Gradient for hidden to output weights
    dW2 = np.dot(Z.T, dL_dypred)

    # Gradient for hidden layer
    # Use derivative of tanh: 1 - ta Loading...
    delta_hidden = dL_dypred.dot(W2.T) * (1 - Z**2)

    # Gradient for input to hidden weights
    dW1 = np.dot(X.T, delta_hidden)

    # Update weights
    W2 -= lr * dW2
    W1 -= lr * dW1

return W1, W2, error_over_time

```

شکل ۱۰۸ - تابع backward - بخش دوم

نحوه عملکرد تابع دقیقا مشابه فرمول‌هایی است که از کتاب بیشاب استخراج شده و در بالا هم نوشته شده است. روند کلی بدین صورت است که ابتدا به صورت رندوم مقادیر اولیه برای وزن‌ها اختصاص داده خواهد شد. برای این کار از seed هم استفاده شده تا با هر بار اجرا مقادیر تصادفی یکسانی بگیریم. سپس وارد یک حلقه شده ایم که در این بخش ابتدا با فراخوانی تابع forward، مقدار پیش‌بینی شبکه را بدست آورده و با کمک این مقدار و مقدار واقعی که از قبیل داریم خطای محاسبه کرده و تابع loss را تشکیل می‌دهیم که اینجا همان squared error هست. برای محاسبه ترم‌های derivative از فرمول‌ها استفاده شده و ابتدا وزن W_2 و سپس با توجه به آن وزن لایه قبل W_1 محاسبه شده است. در نهایت با توجه به مقادیر learning rate و عدد iteration وزن‌ها آپدیت شده اند. در هر مرحله خطای محاسبه شده و عدد derivative ذخیره می‌شود. در پایان وزن‌های محاسبه شده پایانی و لیست خطای بر می‌گردد. برای استفاده از این تابع در بخش بعدی سوال تغییراتی را در ورودی اعمال خواهیم کرد.

```

def backward(X, y, M, iters, lr, X_val, y_val):
    """
    Train neural network using backpropagation.

    Parameters:
    X (numpy.ndarray): Input features matrix of shape (N, D)
    y (numpy.ndarray): True labels of shape (N, 1)
    M (int): Number of neurons in hidden layer
    iters (int): Number of training iterations
    lr (float): Learning rate

    Returns:
    tuple: (W1, W2, error_over_time)
        W1: Updated weight matrix for input to hidden layer
        W2: Updated weight matrix for hidden to output layer
        error_over_time: Array of errors for each iteration
    """
    # Get input dimensions
    N, D = X.shape

    # Initialize weights randomly with small values
    np.random.seed(42)
    W1 = np.random.randn(D, M) * 0.01
    W2 = np.random.randn(M, 1) * 0.01

    # Initialize error tracking array
    error_over_time = np.zeros(iters)

    # Initialize metric tracking arrays
    train_losses = np.zeros(iters)
    train_accuracies = np.zeros(iters)
    val_losses = np.zeros(iters)
    val_accuracies = np.zeros(iters)

```

شکل ۱۰.۹ - تابع **backward** جدید - بخش اول

۲-۲. آزمون شبکه عصبی بر روی یک مجموعه داده

در این بخش ابتدا فایل دیتا را به عنوان ورودی در کد آپلود کردیم و سپس با استفاده از کتابخانه Pandas مدیریت اولیه روی دیتا را انجام داده‌ایم. در گام بعد با توجه به خواسته سوال، عملیات نرمالیزیشن انجام شده است به این صورت که برای هر فیچر میانگین و انحراف معیار محاسبه شده است و سپس مقدار هر دیتا برای هر فیچر منهای میانگین شده و بر انحراف معیار تقسیم شده است.

```
from google.colab import files  
uploaded = files.upload()
```

Choose Files winequality-red.csv
• **winequality-red.csv**(text/csv) - 100951 bytes, last modified: 9/21/2019 - 100% done
Saving winequality-red.csv to winequality-red (1).csv

شکل ۱۱۰ - آپلود فایل CSV به عنوان دیتای ورودی

```
def load_and_prepare_data(filename):  
    """  
    Load and preprocess the wine quality dataset according to the requirements.  
    """  
  
    # Load data  
    data = pd.read_csv(filename)  
  
    # Separate features and target  
    X = data.drop('quality', axis=1).values  
    y = data['quality'].values.reshape(-1, 1)  
  
    # Split data (50% train, 50% test)  
    N = X.shape[0]  
    train_size = int(0.5 * N)  
  
    X_train = X[:train_size]  
    y_train = y[:train_size]  
    X_test = X[train_size:]  
    y_test = y[train_size:]  
  
    # Standardize features using only training data  
    X_mean = np.mean(X_train, axis=0)  
    X_std = np.std(X_train, axis=0)  
  
    X_train = (X_train - X_mean) / X_std  
    X_test = (X_test - X_mean) / X_std  
  
    # Add bias column (ones) to the beginning of feature matrices  
    X_train = np.hstack([np.ones((X_train.shape[0], 1)), X_train])  
    X_test = np.hstack([np.ones((X_test.shape[0], 1)), X_test])  
  
    return X_train, X_test, y_train, y_test
```

شکل ۱۱۱ - استخراج اطلاعات و نرمالیزیشن و اضافه کردن ستون bias به دیتا

در بخش فوق تمامی ستون‌ها ابتدا خوانده شده سپس ستون آخر به عنوان خروجی y در نظر گرفته شده و باقی ستون‌ها همگی فیچرهای ما برای ورودی است و به عبارتی ابعاد ماتریس ورودی X را تشکیل می‌دهد. در گام بعد ۵۰ درصد دیتای ورودی را به عنوان دیتای train و باقی را به عنوان دیتای test در نظر گرفتیم. سپس با محاسبه مقدار میانگین و انحراف معیار دیتای ورودی را نرمالایز کردیم. در خط بعد برای هر دو دیتای تست و ترین یک ستون به عنوان bias اضافه شده است.

تابع backward نوشته شده در بخش قبل برای بخش محاسبات کفایت خواهد کرد اما به جهت تسهیل در ترسیم نمودارهای loss و accuracy در این تابع مقادیر x_val و y_val هم به عنوان آرگومان ورودی ارسال کردیم تا بتوانیم در همین قسمت و بدون کد اضافه accuracy را هم محاسبه کنیم. روند اضافه بدین صورت است که علاوه بر محاسبات فوق مقدار loss را با توجه به تابع squared error و همینطور accuracy را علاوه بر train برای test هم پیاده‌سازی کردیم. چندین لیست با مقدار اولیه صفر ایجاد کردیم که حاوی مقادیر لاس و دقت برای بخش TRAIN و TEST است. سپس در ادامه با آپدیت کردن مقادیر وزن لایه‌ها این لیست‌ها را آپدیت کردیم.

```
# Training loop
for t in range(iters):
    # Forward pass
    y_pred, Z = forward(X, W1, W2)

    # Compute error (squared error)
    error = y_pred - y
    # error_over_time[t] = np.mean(error**2) # this is MSE
    error_over_time[t] = np.sum(error**2) # this is Squared Error

    # Backward propagation
    # Derivative of output layer (squared error loss)
    dL_dypred = 2 * error

    # Gradient for hidden to output weights
    dW2 = np.dot(Z.T, dL_dypred)

    # Gradient for hidden layer
    # Use derivative of tanh: 1 - tanh^2
    delta_hidden = dL_dypred.dot(W2.T) * (1 - Z**2)

    # Gradient for input to hidden weights
    dW1 = np.dot(X.T, delta_hidden)

    # Update weights
    W2 -= lr * dW2
    W1 -= lr * dW1
```

شکل ۱۱۲ - تابع backward - بخش دوم

در پایان که مقادیر وزن‌ها را آپدیت کردیم. در بخش بعد با توجه به مقادیر آپدیت شده وزن‌ها می‌توانیم تابع forward را فراخوانی کنیم و با توجه به مقادیر x_train و x_test و $loss$ و $accuracy$ دست یابیم.

```

# Forward pass on training data
y_train_pred, _ = forward(X, W1, W2)
train_losses[t], train_accuracies[t] = compute_metrics(y, y_train_pred)

# Forward pass on validation data
y_val_pred, _ = forward(X_val, W1, W2)
val_losses[t], val_accuracies[t] = compute_metrics(y_val, y_val_pred)

# Plot individual metrics
fig = plot_all_metrics(train_losses, train_accuracies, val_losses, val_accuracies, rms_errors, lr)
plt.show()

return W1, W2, error_over_time

```

شکل ۱۱۳ - تابع backward - بخش سوم

در این بخش علاوه بر محاسبه مقادیر loss و accuracy نمودار تغییرات هر یک را هم بر حسب iteration ترسیم کردیم. در نهایت خروجی ما علاوه بر وزن‌های W1 و W2 مقدار error_over_time است که از استفاده شده است برای این بخش Squared error.

قبل از استفاده از توابع forward و backward چندین تابع برای محاسبه مقادیر RMS و ترسیم نمودارها نوشته شده است که در ادامه به تشریح و توضیح این توابع خواهیم پرداخت و در نهایت با توجه به تعداد نورون و ۱۰۰۰ تکرار برنامه را اجرا کرده و نتایج را به دقت توضیح خواهیم داد.

```

def compute_metrics(y_true, y_pred):
    """
    Compute squared error loss and simple accuracy metrics.

    Parameters:
    y_true (numpy.ndarray): True labels
    y_pred (numpy.ndarray): Predicted labels

    Returns:
    tuple: (squared_error, accuracy)
        squared_error: Sum of squared differences between predictions and true values
        accuracy: Proportion of predictions within 0.5 of true value
    """
    squared_error = np.sum((y_true - y_pred) ** 2)

    # Simple accuracy: predictions within 0.5 of true value
    accuracy = np.mean(np.abs(y_true - y_pred) < 0.5)
    return squared_error, accuracy

```

شکل ۱۱۴ - تابع محاسبه **accuracy** و **squared_error**

برای محاسبه loss صورت سوال اشاره کرده است که از چه تابعی استفاده کنیم اما برای محاسبه Accuracy اشاره به استفاده از روش خاصی نشده است. لذا به ساده ترین شکل ممکن عمل کردیم. برای محاسبه accuracy ابتدا یک threshold تعريف کردیم. هر چه عدد کوچکتر باشد دقت بدست آمده واقعی تر و دقیق تر خواهد بود. اینجا از 0.5 به این عنوان استفاده شده است. سپس با بررسی قدرمطلق تفاوت مقدار واقعی و مقدار پیشビینی شده توسط شبکه مقایسه با threshold مقدار true یا false را گزارش کرده‌ایم به این معنی که برای هر کدام از مقادیر اگر شبکه درست پیشビینی کرده باشد true یا عدد یک و اگر اشتباه پیشビینی کرده باشد false یا عدد صفر را گزارش خواهیم کرد. در نهایت با میانگین گیری به مقدار accuracy برای آن iteration خاص رسیده ایم. این محاسبه به همین شکل برای تمام مراحل محاسبه و در لیستی ذخیره می‌شود و در نهایت نمودار تغییرات accuracy و loss بر حسب iteration برای هر learning rate ترسیم می‌شود. برای ترسیم نمودارها از توابع زیر استفاده شده است.

```

def plot_all_metrics(train_losses, train_accuracies, val_losses, val_accuracies, rms_errors, lr):
    """
    Plot all metrics including the original RMS plot.
    """
    fig = plt.figure(figsize=(20, 5))

    # Plot 1: Losses
    ax1 = plt.subplot(131)
    ax1.plot(train_losses, label='Training Loss')
    ax1.plot(val_losses, label='Validation Loss')
    ax1.set_xlabel('Iteration')
    ax1.set_ylabel('Squared Error')
    ax1.set_title(f'Loss Over Time (lr={lr})')
    ax1.legend()
    ax1.grid(True)

    # Plot 2: Accuracies
    ax2 = plt.subplot(132)
    ax2.plot(train_accuracies, label='Training Accuracy')
    ax2.plot(val_accuracies, label='Validation Accuracy')
    ax2.set_xlabel('Iteration')
    ax2.set_ylabel('Accuracy')
    ax2.set_title(f'Accuracy Over Time (lr={lr})')
    ax2.legend()
    ax2.grid(True)

    plt.tight_layout()
    return fig

```

شکل ۱۱۵ - تابع ترسیم تغییرات accuracy و loss بر حسب iteration

```

def plot_learning_curves(errors_list, learning_rates):
    """
    Plot error over time for different learning rates.
    """
    plt.figure(figsize=(10, 6))
    for i, (errors, lr) in enumerate(zip(errors_list, learning_rates)):
        plt.plot(errors, label=f'Learning Rate = {lr}')

    plt.xlabel('Iteration')
    plt.ylabel('Error')
    plt.title('Error vs. Iteration for Different Learning Rates')
    plt.legend()
    plt.grid(True)
    plt.show()

```

شکل ۱۱۶ - تابع ترسیم error بر حسب iteration - خروجی تابع backward

از تابع فوق برای نمایش تغییرات خطای زمان یا همان در هر iteration استفاده شده است. خطای backward در این تابع همان خروجی تابع backward است.

```

def calculate_rmse(y_true, y_pred):
    """
    Calculate Root Mean Squared Error
    """
    return np.sqrt(np.mean((y_true - y_pred) ** 2))

def train_and_evaluate(X_train, X_test, y_train, y_test, learning_rates):
    """
    Train and evaluate the model with different learning rates
    """
    M = 30 # Number of hidden neurons
    iterations = 1000
    errors_list = []
    rmse_list = []

    for lr in learning_rates:
        print(f"\nTraining with learning rate: {lr}")

        # Train the model
        W1, W2, errors = backward(X_train, y_train, M, iterations, lr, X_test, y_test)
        errors_list.append(errors)

        # Make predictions
        y_pred, _ = forward(X_test, W1, W2)

        # Calculate RMSE
        rmse = calculate_rmse(y_test, y_pred)
        rmse_list.append(rmse)
        print(f"RMSE for learning rate {lr}: {rmse:.4f}")

    return errors_list, rmse_list

```

شکل ۱۱۷ - توابع محاسبه **RMS** و کنترل مراحل **train**

در تابع اول به سادگی فرمول RMS نوشته شده است و با توجه به مقدار واقعی و پیش‌بینی شده خروجی این مقدار محاسبه می‌شود. لازم به ذکر است این بخش تنها برای پایان iteration و وزن‌های نهایی به دست آمده به ازای یک learning rate ثابت هستند. در تابع train_and_evaluate مراحل پیاده سازی به دقت نوشته شده است. برای اجرای این برنامه ابتدا ۳۰ نورون برای لایه مخفی در نظر گرفته شده است و از ۱۰۰۰ تکرار استفاده شده است. در ادامه این تابع با توجه به مقادیر x و y برای train و test و همینطور learning rate های مختلف یک حلقه بکار گرفته‌ایم. در این حلقه با توجه به مقدار نرخ یادگیری مقادیر ورودی به تابع backward فرستاده می‌شود و مطابق مراحل از پیش گفته شده مقادیر وزن‌ها آپدیت شده و در پایان وزن لایه‌ها به علاوه‌ی مقدار error_over_time به عنوان خروجی گزارش خواهد شد. سپس با توجه به W1، W2 تابع forward را صدای زده ایم و عدد پیش‌بینی شده مقدار مدل با توجه به این وزن‌ها learning rate محاسبه کرده ایم. در نهایت با استفاده از تابع محاسبه RMS مقدار این خط را هم به ازای این rate محاسبه کرده ایم و در یک لیست نگه داشته ایم.

```

# Load and prepare data
filename = "winequality-red.csv"
X_train, X_test, y_train, y_test = load_and_prepare_data(filename)

# Define learning rates to test
learning_rates = [0.000001, 0.00001, 0.0001]

# Train and evaluate for each learning rate
errors_list, rmse_list = train_and_evaluate(X_train, X_test, y_train, y_test, learning_rates)

# Plot learning curves
plot_learning_curves(errors_list, learning_rates)

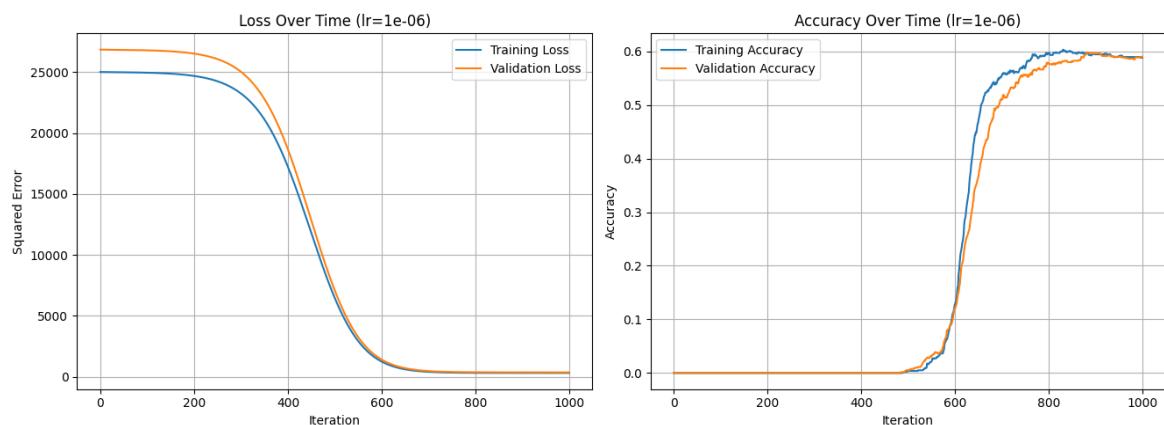
# Print best learning rate
best_lr_idx = np.argmin(rmse_list)
print(f"\nBest learning rate: {learning_rates[best_lr_idx]}")
print(f"Best RMSE: {rmse_list[best_lr_idx]:.4f}")

```

شکل ۱۱۸ - اجرای الگوریتم بر روی دیتا

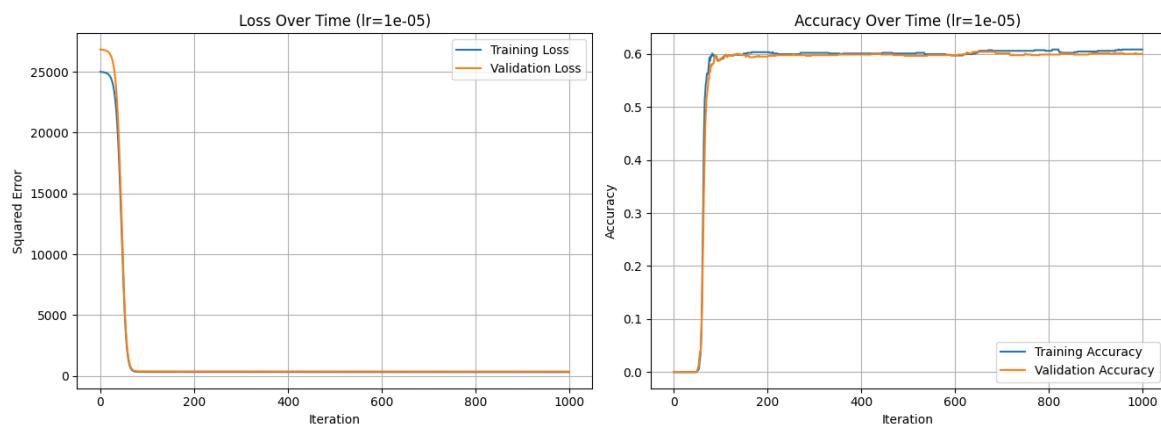
در این مرحله ابتدا با استفاده از تابع `load_and_prepare_data` مقادیر را از فایل استخراج کرده ایم. سپس با انتخاب سه `learning rate` متفاوت که در صورت سوال هم از ما خواسته شده بود به اجرای الگوریتم پرداخته ایم و در نهایت نمودارها را ترسیم کرده ایم. در ادامه در قالب شکل و جدول مقادیر مختلف `learning rate` و نحوه عملکرد شبکه ساخته شده را مورد نقد و بررسی قرار خواهیم داد.

برای خطای پایانی از روش Root Mean Squared Error استفاده شده است. مقدار واقعی لیبل ها همان `y` می هست. روابط گرادیان مطابق روابط کتاب بیشتر نوشته شده و در پایان حلقه وزن ها با توجه به نرخ یادگیری آپدیت شده اند. در پایان حلقه مقدار نهایی `W1` و `W2` و مقدار پایانی خطا را به عنوان خروجی گزارش کرده ایم.



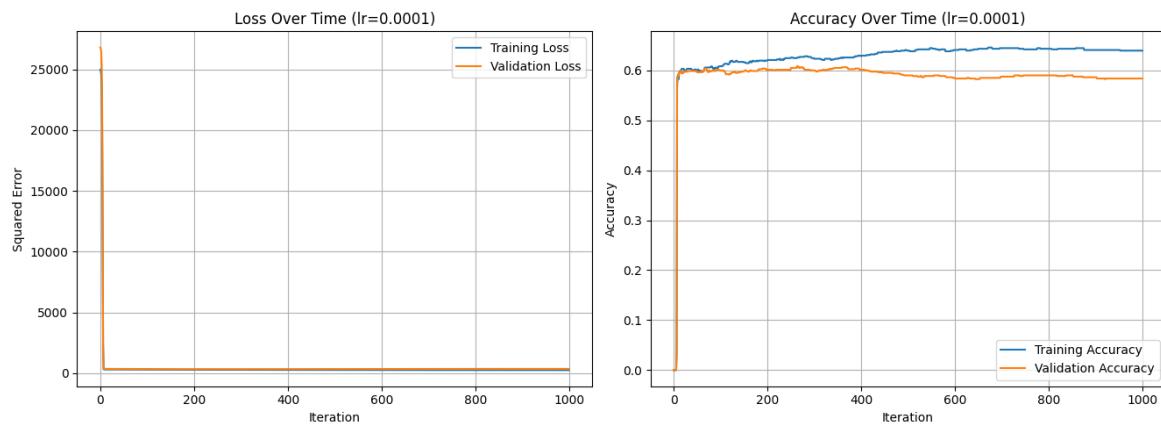
شکل ۱۱۹ - نمودار تغییرات loss و accuracy با learning rate 1e-06

در این حالت مشاهده می‌کنیم که از در حدود ۴۰۰ iteration loss شروع به افت زیاد و accuracy شروع به رشد کرده است.



شکل ۱۲۰ - نمودار تغییرات accuracy و loss با learning rate 1e-05

در این حالت مشاهده می‌کنیم که از در حدود ۱۰۰ iteration loss شروع به افت زیاد و accuracy شروع به رشد کرده است.



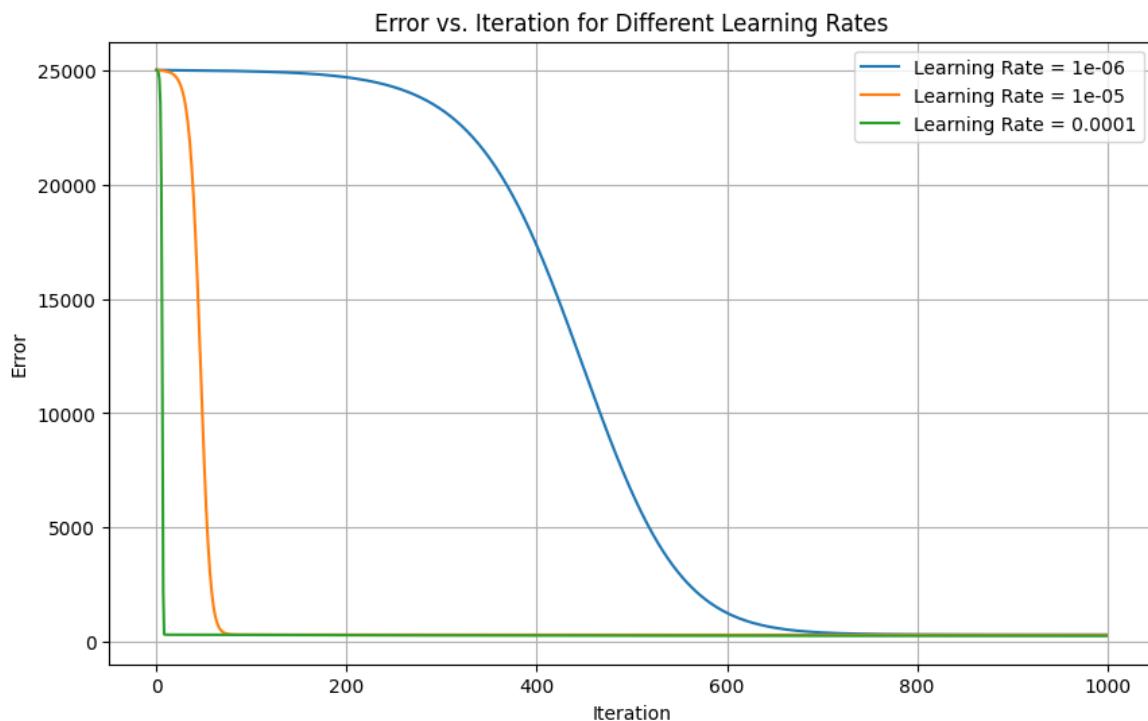
شکل ۱۲۱ - نمودار تغییرات accuracy و loss با learning rate 1e-04

در این حالت اما مشاهده می‌کنیم که در همان iteration های ابتدایی loss شروع به افت زیاد و accuracy شروع به رشد کرده است.

روند کلی به ما نشان می‌دهد که با افزایش learning rate شاهد کاهش تعداد گام‌های محاسباتی برای رسیدن به همگرایی هستیم. به عبارتی زمانی که learning rate عددی به اندازه کافی بزرگ باشد و از طرفی نه زیاد بزرگ که از نقطه optimum جهش کند، الگوریتم سریع تر جلو رفته و مقدار loss کاهش و accuracy افزایش خواهد یافت. این موضوع از رابطه موجود در کتاب بیشап هم قابل حدس بود چرا که در هر گام محاسباتی در انتهای وزن‌ها منهای learning rate ضرب در ترم گرادیان محاسبه شده می‌شند و بدین ترتیب هر چه این عدد learning rate بزرگ تر باشد، گام‌های محاسباتی بزرگ‌تر بوده و سریع تر به مقدار نهایی و بهینه خواهیم رسید. البته که زیاد از حد بزرگ بودن learning rate هم خود مشکل به وجود می‌آورد و ما را به شکل معنا داری از نقطه بهینه دور خواهد کرد که این مورد هم مد نظر ما نخواهد بود. برای انتخاب بهترین leaning rate باید توجه داشته باشیم که ملاک برای ما چیست؟ نکته اولیه این است که با بزرگ‌ترین leaning rate ممکن ما سریع تر به جواب می‌رسیم پس حجم محاسباتی و گام‌ها می‌تواند کاهش یابد که برای ما مطلوب است. از طرف دیگر نکته بعدی دقت کار شبکه است که باید بررسی کنیم کدام شبکه دارای بیشتری دقت و کمترین خطای است. این هم نکته دیگری است که باید مورد توجه ما قرار گیرد.

در این بخش به وضوح مشخص است که $\text{learning rate} = 0.0001$ سریع ترین همگرایی را ایجاد کرده است و در همان اولین گام‌های محاسباتی مقدار loss به شکل چشم گیری کاهش یافته و accuracy افزایش یافته است. اما از طرفی مقدار $\text{learning rate} = 0.00001$ با اینکه گام‌های بیشتری نیاز داشته است تا به بهترین جواب برسد اما خطای محاسبه شده برای این روش که همان Root mean squared error از باقی موارد کمتر بوده است.

از طرفی چون تعداد گام‌ها چشم گیر نبوده است و خطای کمتر مقدار $\text{learning rate} = 0.00001$ را به عنوان بهترین گزینه انتخاب می‌کنیم. هر چند همانطور که گفته شد مقدار 0.0001 هم انتخاب بسیار مناسب و بهینه‌ای است. نکته مهم توجه به مسئله است. در صورتی که دقت و خطای اهمیت بسیار بالایی داشت ما می‌توانیم از گام‌های محاسباتی بیشتر چشم پوشی کنیم اما اگر هزینه محاسباتی را نمی‌توانیم تامین کنیم و اولویت با سریع تر به جواب رسیدن باشد می‌توانیم مقدار دوم و بزرگ‌تر leaning rate را که همان عدد 0.0001 است را گزارش نماییم.



شکل ۱۲۲ - نمودار تغییرات خطا بر حسب learning rate های متفاوت

درباره این سوال که آیا در رابطه با شبکه‌های عصبی دیگر هم همین منطق صدق می‌کند باید اشاره کنم که نحوه تغییرات و سرعت همگرایی با تغییر learning rate به همین صورت خواهد بود چرا که نحوه آپدیت کردن وزن‌ها در شبکه‌های عصبی هم به همین صورت خواهد بود و اگر تغییری باشد در ترم گرادیان و نحوه مشتق گیری‌ها خواهد بود. اما در رابطه با انتخاب بهترین learning rate همانگونه که اشاره شد، موارد متفاوتی از جمله اولویت مسئله و منابع در دسترس می‌تواند به ما را در انتخاب بهترین گزینه کمک کند.

S

جدول ۱۲ - مقایسه RMSE

RMSE	Learning Rate
0.6763	0.0001
0.6638	0.00001
0.6761	0.000001

۱-۳. الگوریتم‌های MRI و MRII

MADALINE نوعی شبکه عصبی ساخته شده از نورون‌های خطی تطبیق پذیر است که در لایه‌ها سازماندهی شده‌اند. این بخش یک MADALINE را با یک لایه مخفی شامل دو واحد آدلین مخفی و یک واحد آدلین خروجی را پوشش می‌دهد. خروجی واحدهای پنهان بر اساس سیگنال‌های ورودی، با اعمال تابع آستانه محاسبه می‌شود. در حالی که واحدهای پنهان قابلیت‌های شبکه را افزایش می‌دهند، فرآیند آموزش را نیز پیچیده می‌کنند.

دو الگوریتم آموزشی شرح داده شده است: الگوریتم MRI، که تنها وزن واحدهای پنهان را تنظیم می‌کند و وزن واحد خروجی را ثابت نگه می‌دارد، و الگوریتم MRII، که امکان تنظیم همه وزن‌ها را در شبکه فراهم می‌کند. هدف هر دو الگوریتم به حداقل رساندن اختلال در طول آموزش و ایجاد انعطاف‌پذیری در تعریف تابع منطقی واحد خروجی است. الگوریتم‌ها شامل چندین مرحله برای مقداردهی اولیه وزن‌ها، محاسبه خروجی‌ها، تعیین خطاهای و به روز رسانی وزن‌ها بر اساس داده‌های آموزشی است.

۲-۳. نمودار پراکندگی داده‌ها

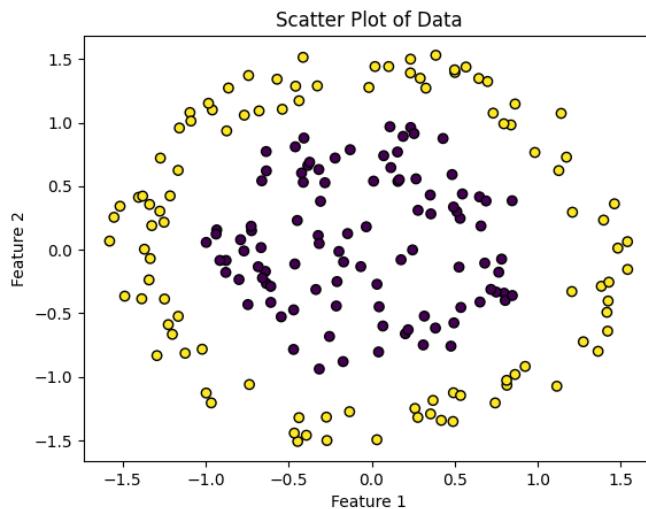
با استفاده از pandas داده‌ها را از فایل مورد نظر بارگذاری می‌کنیم و با استفاده از نمودار پراکندگی داده‌ها را رسم می‌کنیم.

```
# بارگذاری داده‌ها
data = pd.read_csv('Question3.csv')

# استخراج ویژگی‌ها و کلاس
x = data.iloc[:, 0:2] # ستون‌های ویژگی
y = data.iloc[:, 2] # ستون کلاس

# رسم نمودار پراکندگی
plt.scatter(x.iloc[:, 0], x.iloc[:, 1], c=y, cmap='viridis', edgecolor='k')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Scatter Plot of Data')
plt.show()
```

شکل ۱۲۳ - کد نمودار پراکندگی



شکل ۱۲۴ - نمودار پراکندگی

۳-۳. آموزش مدل

تابعی به اسم train_model طراحی کرده‌ایم که یک شبکه عصبی با یک لایه و تعدادی نورون (به صورت پارامتر) با استفاده از تابع فعالساز relu روی ۸۰ درصد داده‌های و ۲۰۰ ایپاک آموزش می‌بیند، پیاده سازی کردیم. و با استفاده از یک حلقه هر سه شبکه با تعداد نورون‌های ۳، ۴ و ۸ نورون را آموزش داده‌ایم. معیار دقت (Accuracy) شبکه عصبی با ۳ نورون ۷۰٪ و با ۴ نورون ۷۸٪ و با ۸ نورون به ۹۰٪ است.

```
# تقسیم داده‌ها به مجموعه آموزش و آزمایش
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)

# تابعی برای آموزش مدل
def train_model(neurons):
    model = Sequential()
    model.add(Input(shape=(x.shape[1],)))
    model.add(Dense(neurons, activation='relu'))
    model.add(Dense(1, activation='sigmoid')) # برای کلاس‌های دودویی

    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

    history = model.fit(X_train, y_train, epochs=200, verbose=0)
    return history, model
```

```
# آموزش مدل‌ها
models = {}
for neurons in [3, 4, 8]:
    print(f'Training model with {neurons} neurons...')
    history, models[neurons] = train_model(neurons)
    models[neurons] = {'model': models[neurons], 'history': history}
```

شکل ۱۲۵ - آموزش مدل‌ها

```

# ارزیابی مدلها
for neurons, model_data in models.items():
    model = model_data['model']
    predictions = model.predict(X_test)
    predictions = (predictions > 0.5).astype(int) # تبدیل به کلاس‌های دودویی
    accuracy = accuracy_score(y_test, predictions)
    print(f'Accuracy for {neurons} neurons: {accuracy:.2f}\n')

```

شکل ۱۲۶ - ارزیابی مدلها

در سه شکل زیر مرزهای جدا کننده با استفاده از کتابخانه‌ها را نشان می‌دهد که چگونه هر شبکه روی داده‌ها آموزشی قرار گرفته‌اند.

```

# نمایش خط‌های جداکننده
def plot_decision_boundary(model, X, y, neurons):
    x_min, x_max = X.iloc[:, 0].min() - 1, X.iloc[:, 0].max() + 1
    y_min, y_max = X.iloc[:, 1].min() - 1, X.iloc[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                          np.arange(y_min, y_max, 0.01))

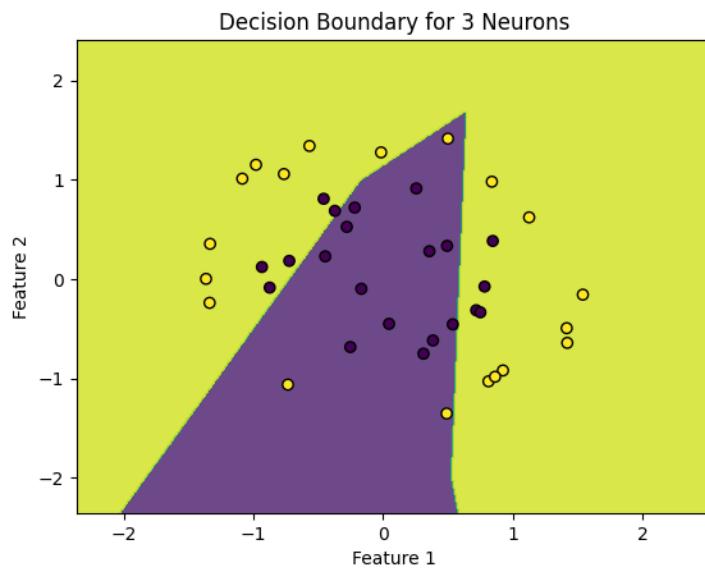
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = (Z > 0.5).astype(int)
    Z = Z.reshape(xx.shape)

    plt.contourf(xx, yy, Z, alpha=0.8)
    plt.scatter(X.iloc[:, 0], X.iloc[:, 1], c=y, edgecolor='k')
    plt.title(f'Decision Boundary for {neurons} Neurons')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.show()

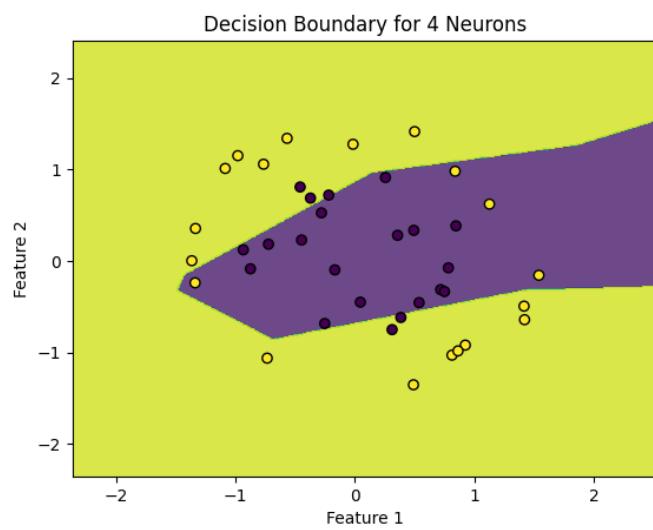
# رسم خط جداکننده برای هر مدل
for neurons, model_data in models.items():
    plot_decision_boundary(model_data['model'], X_test, y_test, neurons)

```

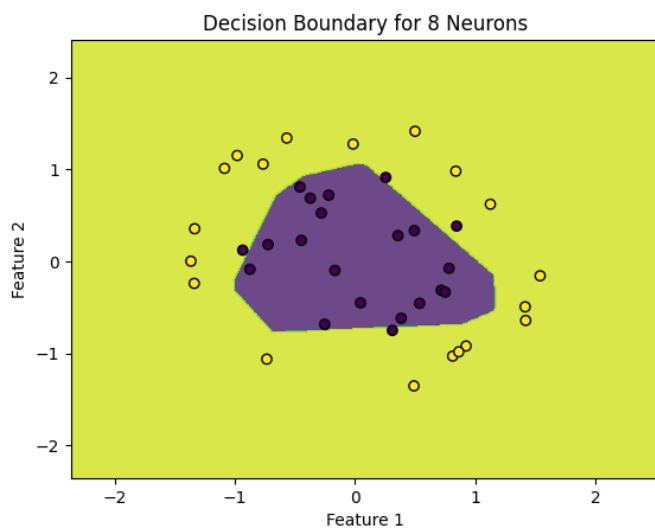
شکل ۱۲۷ - کد خط‌های جداکننده



شکل ۱۲۸ - مرز تصمیم در ۳ نورون



شکل ۱۲۹ - مرز تصمیم در ۴ نورون



شکل ۱۳۰- مرز تصمیم در ۸ نورون

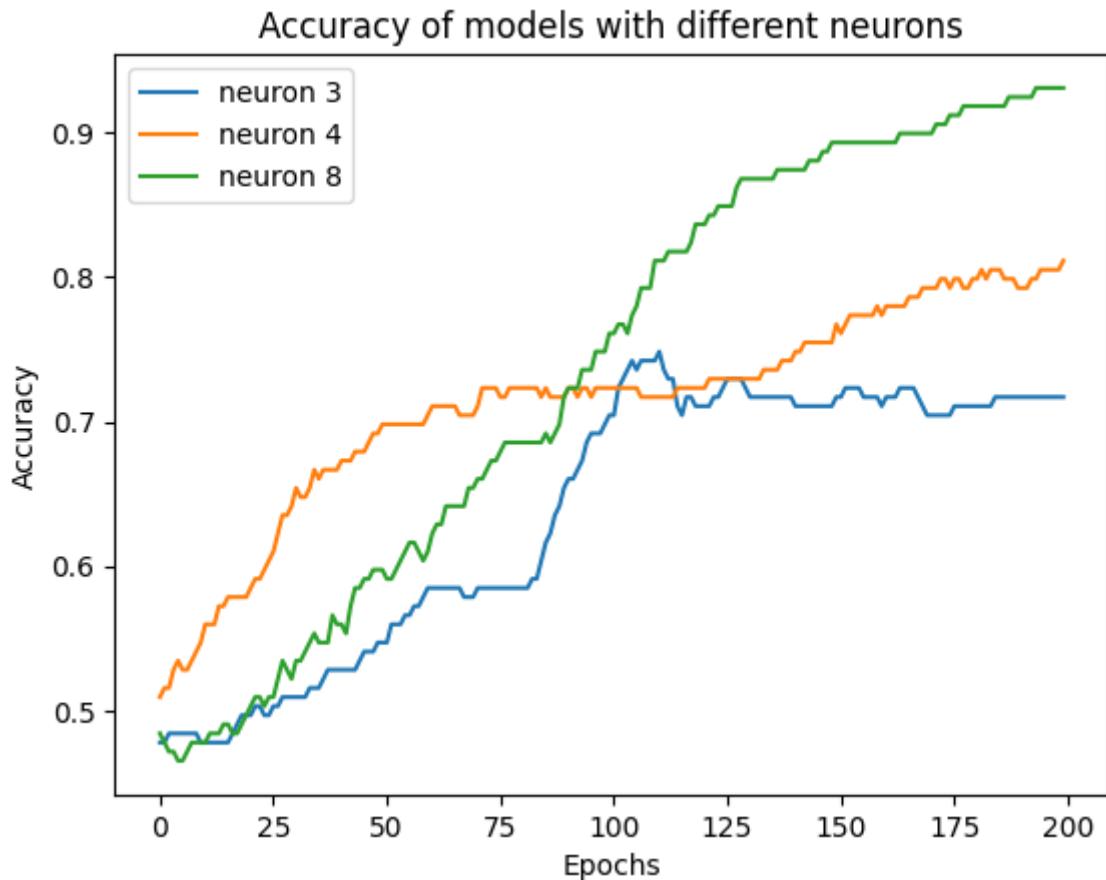
۴-۳. تحلیل نتایج

با توجه به معیار دقت هر شبکه و نمودار زیر می‌توان به این نتیجه رسید که با افزایش تعداد نورون‌ها در شبکه عصبی با یک لایه در این داده‌ها می‌توان میزان دقت خود را بالا برد و لذا شبکه عصبی با ۸ نورون بالاترین دقت را دارد. این نمودار همچنین این را نمایش می‌دهد که در ابتدا معیار دقت در در دو شبکه با ۳ و ۴ نورون بهتر بوده ولی از نقطه‌ای شبکه با ۸ نورون از هردو شبکه دیگر به وضوح بهتر عمل می‌کند.

```
#رسم نمودار دقت برای هر مدل
for neurons, model_data in models.items():
    history = model_data['history']
    plt.plot(history.history['accuracy'], label=f'neuron {neurons}')

plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Accuracy of models with different neurons')
plt.legend()
plt.show()
```

شکل ۱۳۱ - کد رسم نمودار برای هر مدل



شکل ۱۳۲- نمودار دقت برای هر مدل

```

for neurons, h in models.items():
    final_accuracy = h['history'].history['accuracy'][-1] * 100
    print(f'Final accuracy of neurons {neurons} is {final_accuracy:.2f}%')

```

Final accuracy of neurons 3 is 71.70%
 Final accuracy of neurons 4 is 81.13%
 Final accuracy of neurons 8 is 93.08%

شکل ۱۳۴- دقت هر مدل

۱-۱. نمایش تعداد ستون

برای بارگذاری داده‌ها از pd.read_csv استفاده می‌کنیم. و سپس با ()isna().sum() تعداد NaN ها در هر ستون را می‌شماریم. که هیچ داده NaN در ستون‌ها وجود ندارد.

```
# بارگذاری داده‌ها
data = pd.read_csv('Question4.csv')
# ها در هر ستون NaN نمایش تعداد
nan_counts = data.isna().sum()
print(nan_counts)
```

```
id          0
date        0
price       0
bedrooms    0
bathrooms   0
sqft_living 0
sqft_lot    0
floors      0
waterfront  0
view         0
condition    0
grade        0
sqft_above   0
sqft_basement 0
yr_built     0
yr_renovated 0
zipcode      0
lat          0
long         0
sqft_living15 0
sqft_lot15   0
dtype: int64
```

شکل ۱۳۵ - کد و نتیجه ستون‌های NaN

۲-۱. ماتریس همبستگی

یا استفاده از متدهای corr، ماتریس همبستگی را محاسبه می‌کنیم. و سپس با seaborn و matplotlib ماتریس را به صورت بصری نمایش می‌دهیم. ویژگی‌هایی که قدر مطلق همبستگی شان با قیمت بیشتر از

۵. است را پیدا می کنیم و اعلام می کنیم. در این داده ها ویژگی های sqft_above، grade، sqft_living و بیشترین همبستگی را با price دارند.

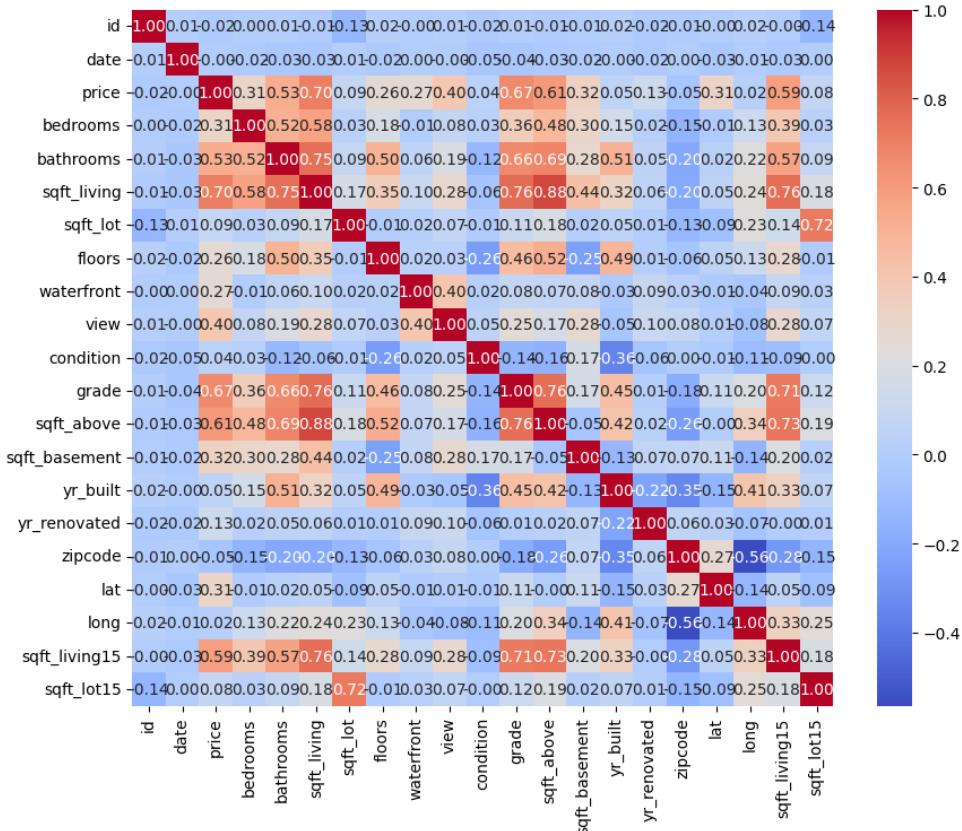
```
# تبدیل ستون تاریخ به فرمت استاندارد
data['date'] = pd.to_datetime(data['date'], format='%Y%m%dT%H%M%S', errors='coerce')

# محاسبه ماتریس همبستگی
correlation_matrix = data.corr()

# رسم ماتریس همبستگی
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, fmt=".2f", cmap='coolwarm')
plt.show()

# ویژگی هایی که بیشترین همبستگی را با قیمت دارند
correlation_with_price = correlation_matrix['price']
high_corr_features = correlation_with_price[correlation_with_price.abs() > 0.5]
high_corr_features = high_corr_features.sort_values(ascending=False)
print("Most correlation features with price:")
print(high_corr_features)
```

شکل ۱۳۶- کد ماتریس همبستگی و نمایش ویژگی هایی که بیشترین correlation را با قیمت دارند



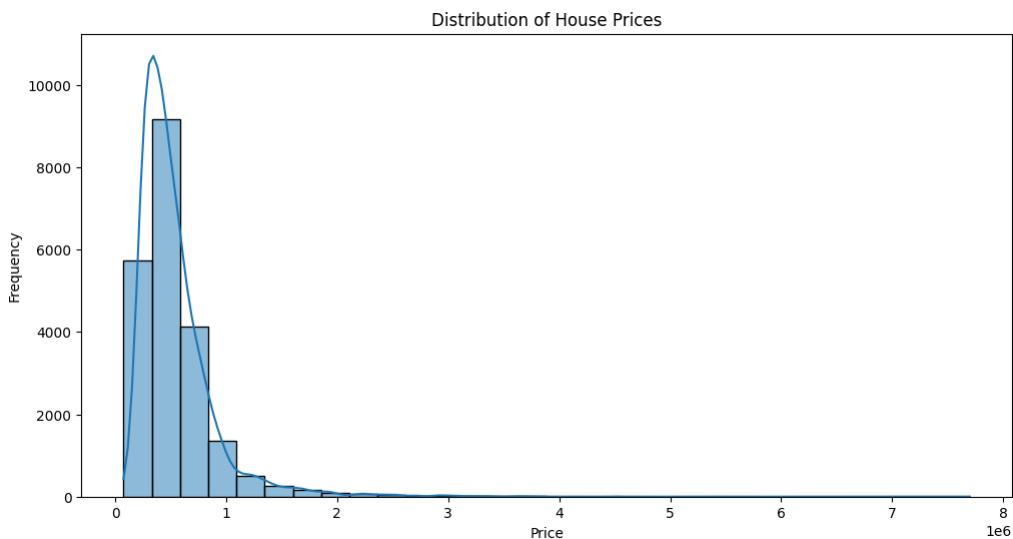
شکل ۱۳۷- ماتریس همبستگی

۴-۳. رسم نمودار

نمودار توزیع قیمت خانه و نمودار قیمت با ویژگی `sqft_living` به عنوان ویژگی که بیشترین همبستگی را با قیمت دارد در این قسمت با استفاده از کتابخانه‌ای `matplotlib` و `seaborn` رسم می‌کنیم.

```
#نمودار توزیع قیمت
plt.figure(figsize=(12, 6))
sns.histplot(data['price'], bins=30, kde=True)
plt.title('Distribution of House Prices')
plt.xlabel('Price')
plt.ylabel('Frequency')
plt.show()
```

شکل ۱۳۸- کد رسم نمودار



شکل ۱۳۹ - نمودار توزیع قیمت خانه



شکل ۱۴۰ - نمودار توزیع قیمت و ویژگی **sqft_living** با بیشترین همبستگی

۴-۴. پیش پردازش داده

در قسمت ۲ قبل از اینکه ماتریس همبستگی را محاسبه کنیم برای جلوگیری از خطا ما ستون تاریخ را به شکل استاندارد آن تبدیل کردیم. ابتدا ستون سال و ماه را به عنوان یک ستون جدا ذخیره می‌کنیم سپس ستون تاریخ را حذف می‌کنیم. و سه سطر اول آن را نمایش می‌دهیم.

```
# تبدیل ستون date
data['date'] = pd.to_datetime(data['date'])
data['month'] = data['date'].dt.month
data['year'] = data['date'].dt.year
data.drop('date', axis=1, inplace=True)
print(data.head(3))

      id   price  bedrooms  bathrooms  sqft_living  sqft_lot  floors \
0  7129300520  221900.0        3      1.00       1180     5650    1.0
1  6414100192  538000.0        3      2.25       2570     7242    2.0
2  5631500400  180000.0        2      1.00       770    10000    1.0

  waterfront  view  condition  ...  sqft_basement  yr_built  yr_renovated \
0          0     0           3  ...            0       1955                  0
1          0     0           3  ...           400       1951                1991
2          0     0           3  ...            0       1933                  0

  zipcode      lat      long  sqft_living15  sqft_lot15  month  year
0    98178  47.5112 -122.257       1340      5650     10  2014
1    98125  47.7210 -122.319       1690      7639     12  2014
2    98028  47.7379 -122.233       2720      8062      2  2015

[3 rows x 22 columns]
```

شکل ۱۴۱- حذف ستون تاریخ و نمایش سه سطر اول

با استفاده از متدهای train_test_split داده‌ها به دو بخش آموزش و تست با نسبت ۷۵ به ۲۳ تقسیم می‌کنیم. با استفاده از تابع MinMaxScaler در کتابخانه sklearn ابتدا روی داده‌های آموزش scale و بعد transform می‌کنیم. و بعد داده‌ای تست را نیز همین روال را انجام می‌دهیم.

```
# تقسیم داده‌ها به train و validation
train_data, validation_data = train_test_split(data, test_size=0.25, random_state=42)
```

شکل ۱۴۲- تقسیم داده‌ها به دو بخش تست و آموزش

```
# مقیاس‌گذاری داده‌های train
scaler = MinMaxScaler()
train_scaled = scaler.fit_transform(train_data.drop('price', axis=1))
train_labels = train_data['price'].values

# مقیاس‌گذاری داده‌های validation
validation_scaled = scaler.transform(validation_data.drop('price', axis=1))
validation_labels = validation_data['price'].values
```

شکل ۱۴۳- مقیاس‌گذاری داده‌های تست و آموزش

۴-۵. پیاده‌سازی مدل

با استفاده از کتابخانه keras دو مدل پیاده‌سازی می‌کنیم. مدل اول یک شبکه عصبی با یک لایه پنهان و تابع فعالساز relu است. و مدل دوم نیز یک شبکه عصبی با دو لایه پنهان و تابع فعالساز relu است.

```
# مدل ۱: یک لایه پنهان
model_1 = Sequential()
model_1.add(Input(shape=(train_scaled.shape[1],)))
model_1.add(Dense(10, activation='relu'))
model_1.add(Dense(1))
```

شکل ۱۴۴- پیاده سازی شبکه عصبی با یک لایه مخفی

```
# مدل ۲: دو لایه پنهان
model_2 = Sequential()
model_2.add(Input(shape=(train_scaled.shape[1],)))
model_2.add(Dense(10, activation='relu'))
model_2.add(Dense(10, activation='relu'))
model_2.add(Dense(1))
```

شکل ۱۴۵- پیاده سازی شبکه عصبی با دو لایه مخفی

۶-۴. آموزش مدل

در این مرحله هر دو مدل را با mean squared error loss function و adam's optimizer و به تعداد ۱۰۰ ایپاک آموزش دادیم. و نمودار آن به صورت زیر رسم شد.

```
model_1.compile(optimizer='adam', loss='mean_squared_error')
history_1 = model_1.fit(train_scaled, train_labels, epochs=100,
                        validation_data=(validation_scaled, validation_labels), verbose=1)
```

شکل ۱۴۶- آموزش با یک لایه مخفی

```
model_2.compile(optimizer='adam', loss='mean_squared_error')
history_2 = model_2.fit(train_scaled, train_labels, epochs=100,
                        validation_data=(validation_scaled, validation_labels), verbose=1)
```

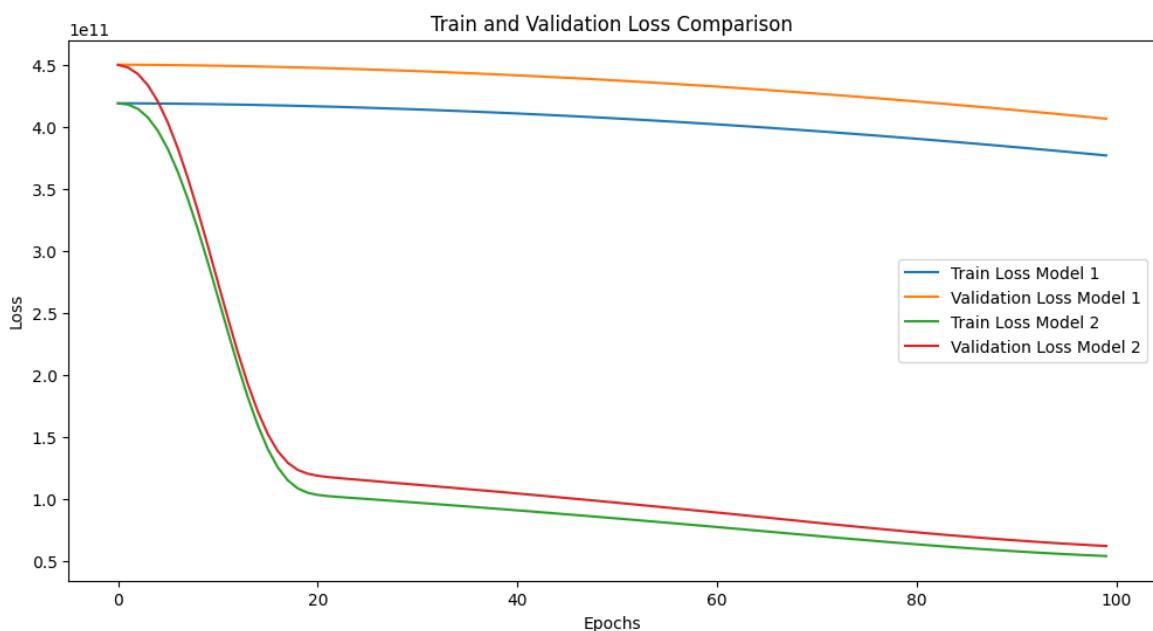
شکل ۱۴۷- آموزش با دو لایه مخفی

```

plt.figure(figsize=(12, 6))
plt.plot(history_1.history['loss'], label='Train Loss Model 1')
plt.plot(history_1.history['val_loss'], label='Validation Loss Model 1')
plt.plot(history_2.history['loss'], label='Train Loss Model 2')
plt.plot(history_2.history['val_loss'], label='Validation Loss Model 2')
plt.title('Train and Validation Loss Comparison')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

```

شکل ۱۴۸ - کد نمایش مقایسه loss تست و آموزش



شکل ۱۴۹ - نمودار مقایسه loss تست و آموزش

تحلیل نتایج

برای مقایسه نتایج نهایی و تعداد ایپاک‌های مناسب برای هر دو مدل (MLP با یک لایه پنهان و MLP با دو لایه پنهان)، باید نمودارهای Loss و دقت هر مدل را بررسی نماییم.

به طور معمول، مدل‌های با لایه‌های بیشتر ممکن است پیچیدگی بیشتری داشته باشند و ممکن است به تعداد بیشتری ایپاک برای همگرا شدن نیاز داشته باشند. در صورتی که مدل با یک لایه پنهان ممکن است سریع‌تر همگرا شود زیرا پیچیدگی کمتری داشته باشد. ولی این موضوع با توجه به تنظیم پارامتر

یادگیری، تابع فعالساز و پیچیدگی داده‌ها در مثال ما بر عکس معمول اتفاق افتاد و مدل با دو لایه پنهان سریع‌تر نسبت به یک لایه پنهان همگرا شده است.

در حین آموزش، دقت مدل‌ها در داده‌های آموزش و تست با هر اپاک تغییر می‌کند. مدل‌هایی با بیش از یک لایه پنهان معمولاً می‌توانند ویژگی‌های پیچیده‌تری را استخراج کنند، بنابراین ممکن است دقت بهتری روی داده‌های آموزش نشان دهد اما خطر overfitting (افت کیفیت مدل روی داده‌های نیز وجود دارد. در مقابل، مدل‌های ساده‌تر معمولاً توانایی کمتری در مدل‌سازی ویژگی‌های پیچیده دارند، اما احتمال overfitting در آنها کمتر است.

برای تعیین تعداد اپاک‌های مناسب، نیاز به بررسی نمودار loss داریم. در اغلب موارد، اگر مدل پس از تعدادی اپاک ثابت شد و کاهش loss به حداقل رسد، این نشان‌دهنده همگرایی مدل است. که در مورد نتایج ما مدل با یک لایه پنهان به اپاک بیشتری برای همگرایی نیاز دارد.

لذا پیچیدگی مدل، ظرفیت یادگیری، داده‌های آموزشی، تنظیمات های پرپارامترها تفاوت‌های اصلی هستند که می‌تواند بر تعداد اپاک‌ها تأثیر بگذارد. با بررسی نمودارهای loss و تغییرات دقت در طول اپاک‌ها می‌توانیم تعداد اپاک‌های مناسب برای هر مدل را مشخص کنیم. و با مقایسه خطای آموزش (training loss) و خطای اعتبارسنجی (validation loss) برای هر مدل، می‌توانیم بفهمیم که آیا مدل به درستی آموزش دیده یا خیر.

```
# به صورت تصادفی validation set انتخاب ۵ داده از
sample_indices = np.random.choice(range(len(validation_labels)), size=5, replace=False)
sample_data = validation_scaled[sample_indices]
sample_prices = validation_labels[sample_indices]

# پیش‌بینی قیمت با مدل بهتر
best_model = model_1 if mse_1 < mse_2 else model_2
print("Best Model is", "One hidden Layer" if mse_1 < mse_2 else "Two hidden Layer")
predicted_prices = best_model.predict(sample_data)
print("\n5 random predictions with best model")
for actual, predicted in zip(sample_prices, predicted_prices):
    print(f'Actual Price: {actual}, Predicted Price: {predicted[0]}')
```

```
Best Model is Two hidden Layer
1/1 ————— 0s 38ms/step

5 random predictions with best model
Actual Price: 915000.0, Predicted Price: 656157.9375
Actual Price: 993500.0, Predicted Price: 726945.0
Actual Price: 385000.0, Predicted Price: 508174.0625
Actual Price: 910000.0, Predicted Price: 835774.75
Actual Price: 691500.0, Predicted Price: 735440.4375
```

شکل ۱۵۰- انتخاب و پیش‌بینی ۵ نمونه تصادفی