به نام خدا

مستندات پروژه اول

درس هوش محاسباتی

نیما اسلامی ۴۰۰۳۶۶۳۰۰۱

مقدمه

در این پروژه قصد داریم دیتاست MNIST را به روش MLP، آموزش دهیم. و با تعیین پارامترهای مختلف، دقت آن را بررسی کنیم.

دیتاست MNIST شامل عکس هایی از عددهای ۰ تا ۹ است که به شکل دست‌نویس نوشته شده اند و برای آموزش آن‌ها نیاز است که ۱۰ کلاس در نظر بگیریم. پس در نتیجه لیبل‌های ما اعداد ۰ تا ۹ می‌شوند.

بارگذاری دیتاست

ابتدا دیتاست را از طریق کتابخانه TensorFlow دانلود می‌کنیم و به دو بخش train و test تقسیم می‌کنیم.

```
# Load the dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

نرمال‌سازی داده‌ها

در ادامه داده را نرمال‌سازی می‌کنیم؛ به این صورت که آن‌ها را تقسیم بر ۲۵۵ می‌کنیم. **چرا؟**

همانطور که می‌دانیم هر پیکسل در بازه رنگی ۰تا ۲۵۵ قرار دارد، ما با تقسیم داده‌ها به ۲۵۵، آن‌ها را بین ۰ تا ۱ نرمال‌سازی می‌کنیم.

این کار باعث می‌شود که الگوریتم آموزش با محاسبات کمتر و به صورت بهینه انجام پذیرد؛ بدون نرمال‌سازی، ویژگی‌های با مقیاس بزرگ‌تر می‌توانند تاثیر بیشتری بر فرآیند آموزش داشته باشند و منجر به مشکلاتی مانند عدم همگرایی یا کندی در فرآیند آموزش شوند.

```
# Normalizing
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
```

کاهش بعد

همانطور که می‌دانیم، شبکه‌های عصبی برای اعمال توابع فعال‌سازی و وزن‌دهی نیاز به ورودی‌های یک بعدی دارند.

تصاویر موجود در دیتاست به صورت ماتریس‌های ۲۸ در ۲۸ پیکسل هستند و ما در این قسمت آن‌ها را به وکتورهای یک بعدی تبدیل می‌کنیم؛ چراکه شبکه‌های عصبی Fully Connected نیاز به ورودی یک بعدی دارند.

```
# Flatten the images from 28x28 to 784-dimensional vectors
x_train = x_train.reshape((x_train.shape[0], 28 * 28))
x_test = x_test.reshape((x_test.shape[0], 28 * 28))
```

در ادامه لیبل‌ها را به صورت یک وکتور دودویی در نظر می‌گیریم که با توجه به کلاس عدد باینری متناظر با آن ۱ میشود.

```
# Convert labels to categorical one-hot encoding
y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)
```

ساخت مدل

برای ساخت مدل ما از کتابخانه keras کمک گرفتیم و یک مدل شبکه عصبی با سه لایه مخفی Dense (به دلیل Fully Connected بودن) ساختیم.

در اینجا در ابتدا از تابع Relu به عنوان تابع فعالساز لایه‌های مخفی استفاده کردیم و برای تخصیص کلاس‌ها از softmax استفاده نمودیم.

```
model = Sequential([
    Dense(512, activation='relu', input_shape=(784,)),
    Dense(256, activation='relu'),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])
```

کامپایل مدل

در این قسمت برای کامپایل مدل ما هایپر پارامترها را به صورت زیر تعیین کردیم:

از الگوریتم بهینه‌سازی Adam ، برای تابع هزینه از categorical crossentropy و برای محاسبه دقت از متریک Accuracy استفاده کردیم.

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

فیت کردن مدل

در ادامه مدل را با ۲۰ اپوک و هر batch را با سایز ۱۲۸ در نظر می‌گیریم و مدل را روی داده‌های آموزشی فیت می‌کنیم.

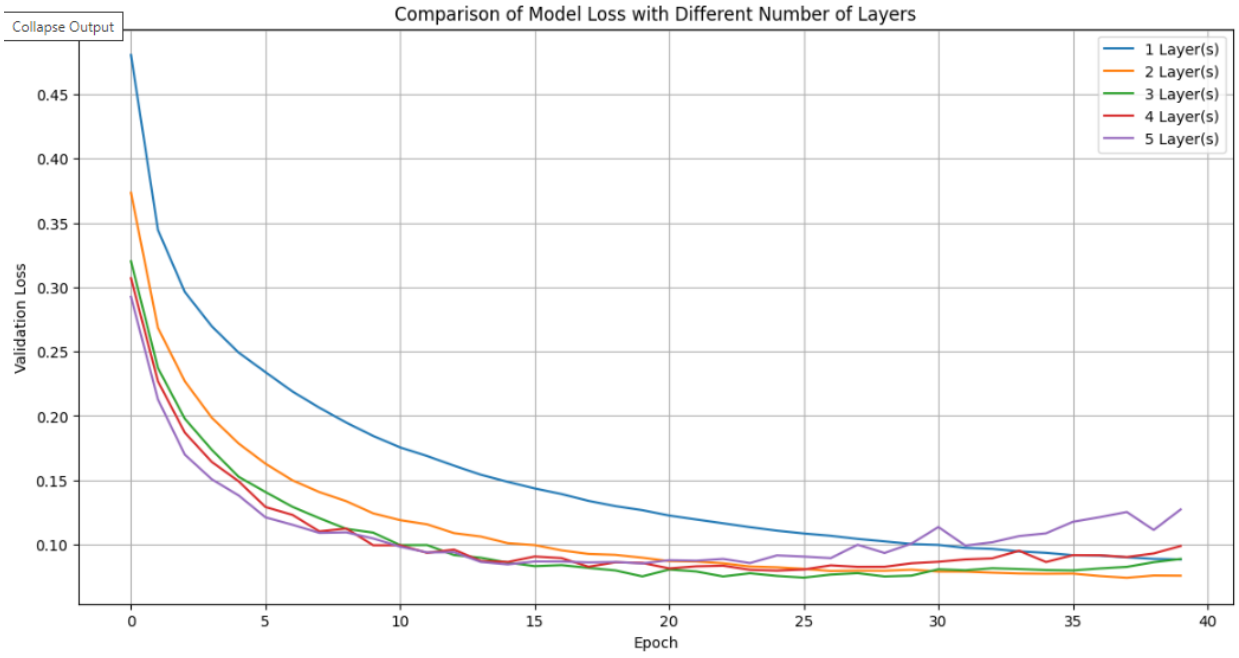
```
mlp = model.fit(x_train, y_train, epochs=20, batch_size=128, validation_split=0.2)
```

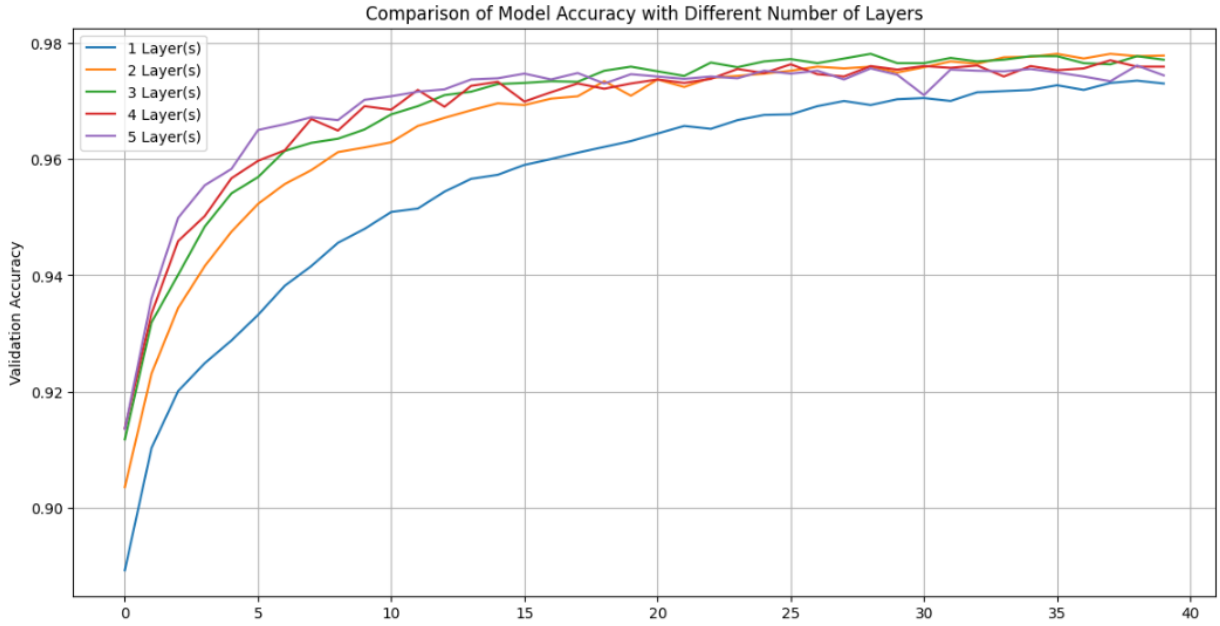
ارزیابی مدل

در قسمت‌های قبل یک مرور بر کلیات کد داشتیم اما در ادامه می‌خواهیم به بررسی جزئی‌تر کد بپردازیم و نتایج را با حالات مختلف بررسی کنیم.

۱. تعداد لایه:

ابتدا آمدیم و مدل را با تعداد لایه های مختلف (۱ تا ۵ لایه) آموزش دادیم.





همانطور که از نمودار مشاهده می‌شود، با افزایش تعداد لایه‌ها، مدل‌ها به سرعت‌تر و با شیب بیشتر از داده‌های آموزش عبور می‌کنند و این باعث افزایش سرعت یادگیری می‌شود. با این حال، اگر تعداد لایه‌ها زیاد شود، همانطور که در نمودار ۴ و ۵ لایه مشاهده می‌شود، مدل به داده‌های آموزشی بسیار بیش‌برازش می‌کند، که موجب افزایش تغییرات ناپایدار و ناگهانی در نمودارها شده است.

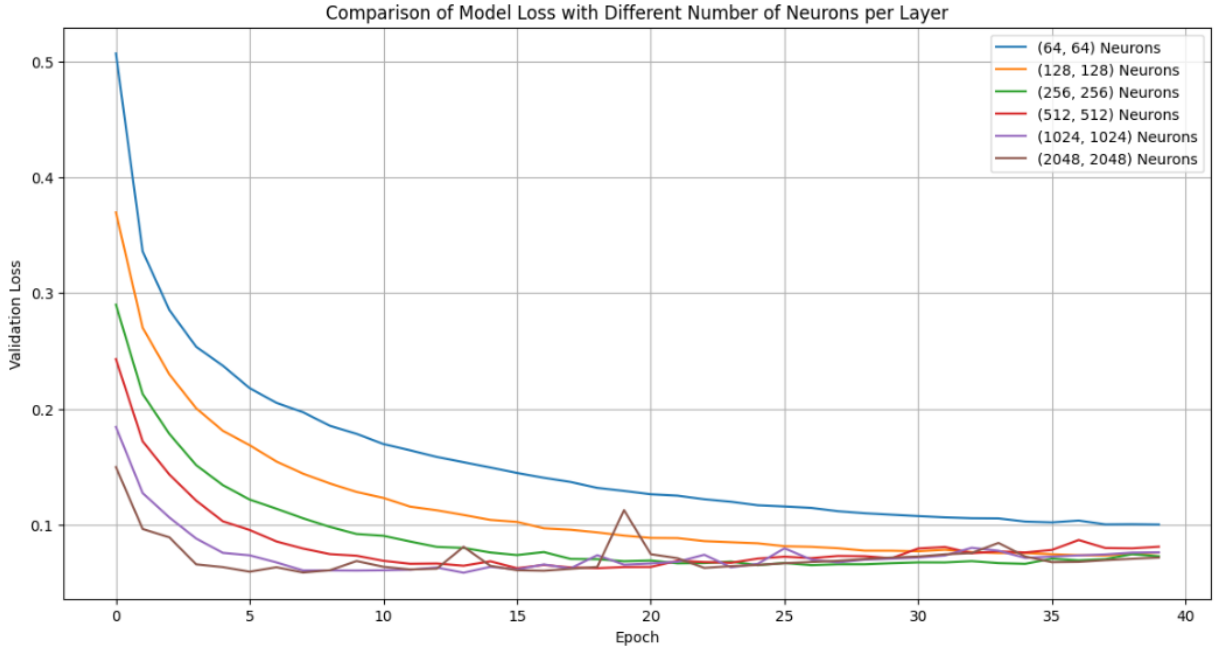
```
1 Layer(s):
Final Training Accuracy: 0.9845499992370605

2 Layer(s):
Final Training Accuracy: 0.9937833547592163

3 Layer(s):
Final Training Accuracy: 0.9981833100318909

4 Layer(s):
Final Training Accuracy: 0.9987666606903076

5 Layer(s):
Final Training Accuracy: 0.9993333220481873
```

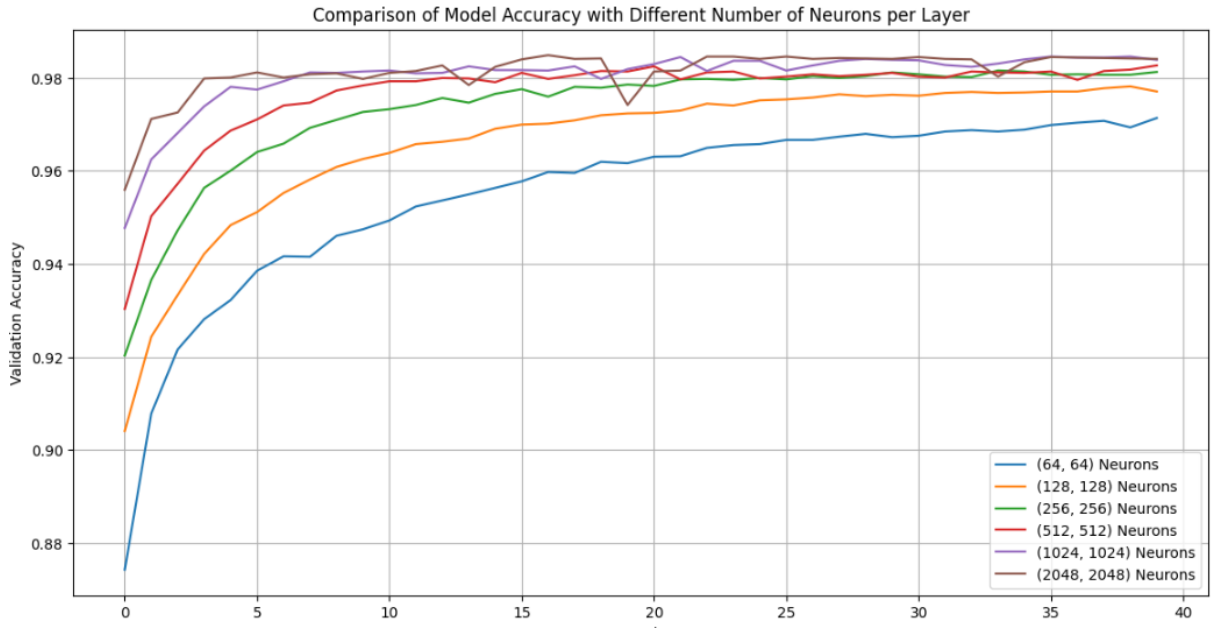
مورد دیگری که با توجه به تصویر بالا و نمودار می‌توان فهمید، این است که علی‌رغم افزایش دقت روی داده‌های آموزش در تعداد لایه‌های بیشتر، میزان loss نیز از یک جا به بعد افزایش می‌یابد و مدل دچار بیش برازش می‌شود.

در آخر باتوجه به نتایج ۲ یا ۳ لایه بهترین و پایدارترین نتایج را حاصل می‌کنند.

۲. تعداد نورون:

باتوجه به نتایج بدست آمده از مورد ۱، ما ۲ لایه را در نظر گرفتیم و تعداد نورون‌های مختلف را برای هر لایه آزمایش می‌کنیم.



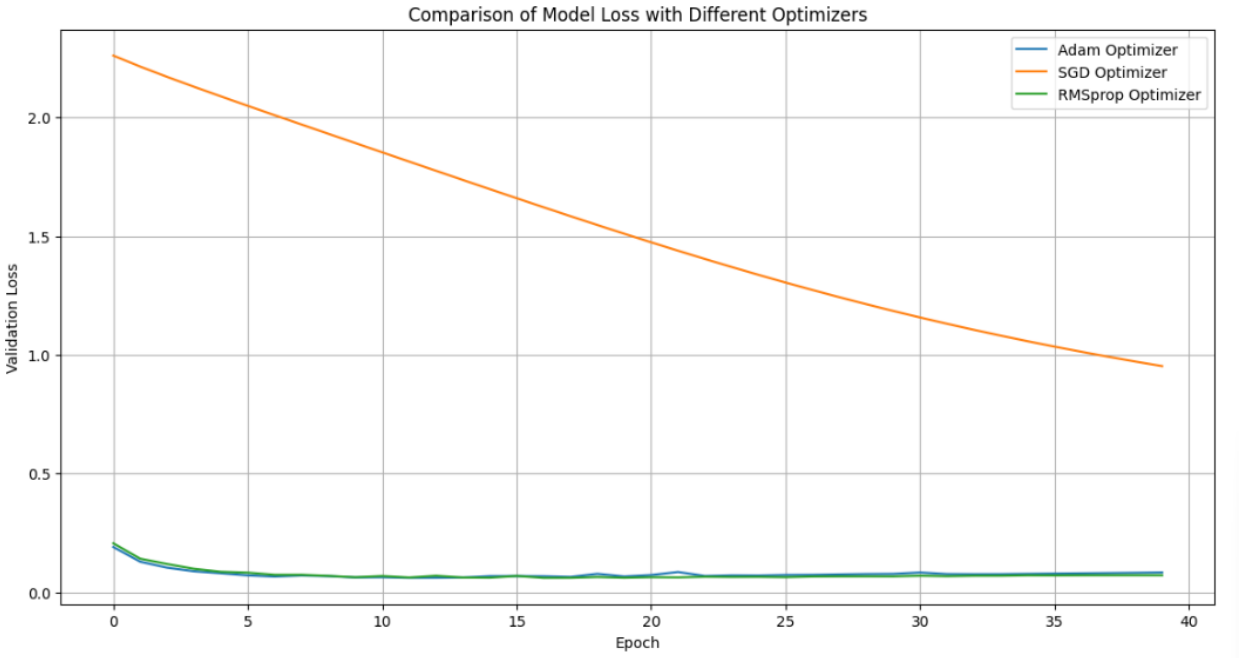


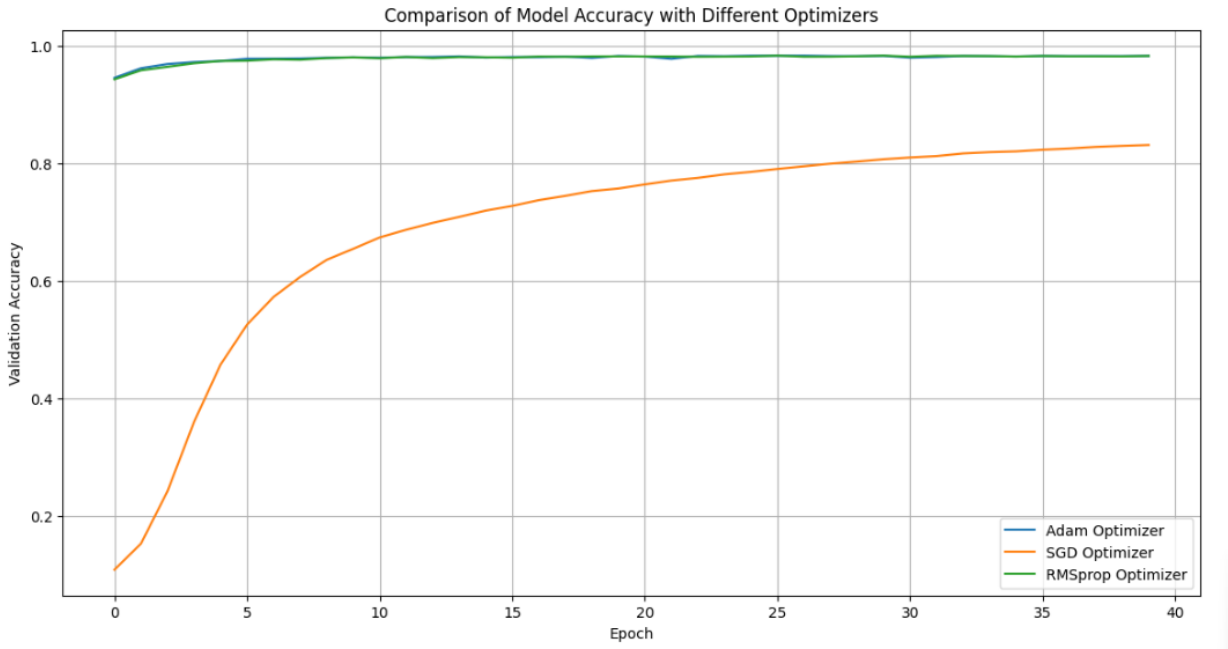
همانطور که مشاهده شد، مطابق افزایش تعداد لایه‌ها، افزایش تعداد نورون‌ها نیز باعث افزایش دقت می‌شود اما مطابق قبل افزایش بیش از حد تعداد نورون، بی تاثیر و کم کم دارای تاثیر منفی می‌شود.

باتوجه به نمودارها مقدار مطلوب با ۲ لایه مخفی ۱۰۲۴ نورون بود و ما در ادامه از این مقدار استفاده می‌کنیم.

۳. الگوریتم بهینه سازی:

ما در این قسمت از ۳ الگوریتم Adam, SGD, RMSprop استفاده کردیم که نتایج آن را در جداول زیر مشاهده می‌کنید.





باتوجه به نمودارها، مشاهده شد که الگوریتم Adam, RMSprop عملکرد بسیار بهتر و نزدیک به هم داشتند، چراکه هر دو تقریبا از الگوهای بهینه سازی مشابهی استفاده می‌کنند.

در Adam, RMSprop، برای هر وزن شبکه، یک نرخ یادگیری مخصوص وجود دارد که با استفاده از زمان اجرا تطبیق می‌یابد. این نرخ یادگیری تا حدی است که بتواند میزان تغییرات گرادیان را مدیریت کند.

یکی از مزیت‌های مهم این دو، این است که دارای یک نرخ یادگیری adaptive هستند که به طور اتوماتیک تطبیق می‌یابند و می‌توانند باعث سریع‌تر رسیدن به نقطه همگرایی شوند. علاوه بر این، به دلیل اینکه میانگین ریشه مربعات گرادیان‌ها را برای هر وزن حساب می‌کنند، از مشکلاتی مانند متوقف شدن در قله‌های محلی که در SGD وجود دارد، جلوگیری می‌کنند.

اما برتری آدام در این است که Adam از یک مفهوم به نام momentum استفاده می‌کند که به الگوریتم کمک می‌کند از گرادیان‌های گذشته برای بهبود سرعت و عملکرد آموزش استفاده کند. در حقیقت آدام ترکیبی از momentum و RMSprop است.

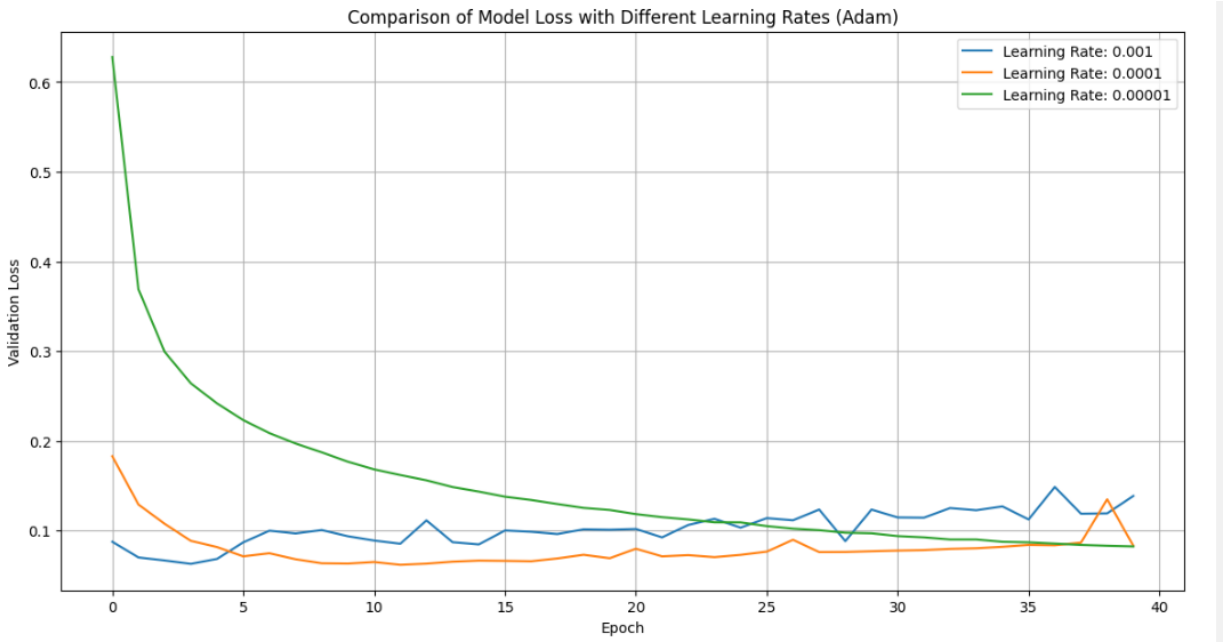
اما درSGD ، به جای استفاده از کل دسته داده‌ها، فقط یک نمونه تصادفی انتخاب می‌شود و برای به‌روزرسانی وزن‌ها استفاده می‌شود. که باعث سرعت و سادگی الگوریتم می‌شود و نیاز به منابع سخت افزاری پیچیده نداشته باشد.

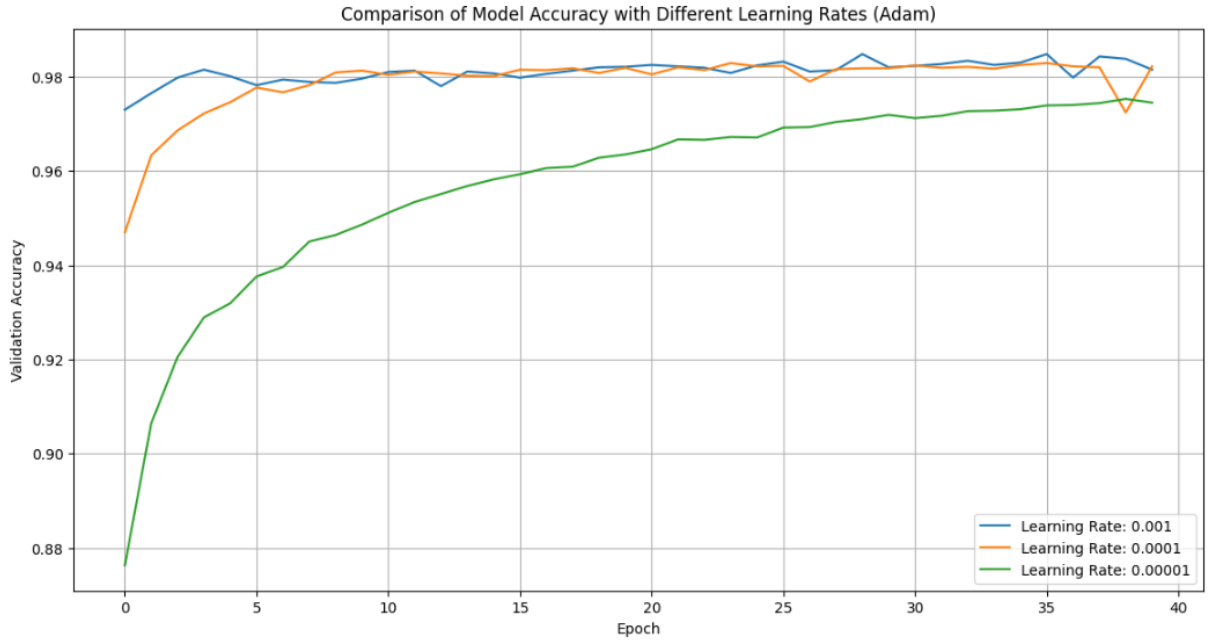
اما علی‌رغم سادگی و سرعت اجرای بالا، SGD مشکلاتی را داراست که در نمودار هم واضح است، مثل حرکت آشفته و پرش‌های ناگهانی در فضای وزن‌ها که می‌تواند به کند شدن فرآیند یادگیری و یا برخورد با نقاط محلی برساند.

همچنین به دلیل ثابت بودن نرخ یادگیری در SGD، نیاز هست که نرخ یادگیری با دقت بالا انتخاب شود.

۴. نرخ یادگیری:

ما در این قسمت از ۳ نرخ یادگیری استفاده کردیم و آن‌ها را با هم مقایسه کردیم.





نرخ یادگیری ۰.۰۰۱:

در ابتدا دقت سریعا افزایش می‌یابد و به سرعت به بیش از ۰.۹۸ می‌رسد. دقت به طور کلی ثابت می‌ماند و فقط کمی نوسان دارد.

این نرخ یادگیری به خوبی به مدل اجازه می‌دهد تا به نقطه بهینه برسد و دقت بالایی در طول اپوک‌ها حفظ شود.

نرخ یادگیری ۰.۰۰۰۱:

دقت در ابتدا سریع‌تر از نرخ یادگیری ۰.۰۰۱ افزایش می‌یابد و به سرعت به بیش از ۰.۹۶ می‌رسد. در طول اپوک‌ها، دقت به طور پیوسته افزایش می‌یابد و به حدود ۰.۹۸ نزدیک می‌شود.

این نرخ یادگیری نیز به مدل اجازه می‌دهد تا به دقت بالایی برسد، هرچند که کمی آهسته‌تر از نرخ ۰.۰۰۱ به نقطه اوج می‌رسد.

نرخ یادگیری ۰.۰۰۰۰۱:

دقت به آرامی افزایش می‌یابد و در ابتدا به سرعت نرخ‌های یادگیری دیگر نمی‌رسد. دقت به طور پیوسته و آهسته افزایش می‌یابد و به حدود ۰.۹۵ می‌رسد.

این نرخ یادگیری به دلیل کوچک بودن، باعث می‌شود که مدل به کندی به نقطه بهینه برسد و به دقت بالایی نرسد.

در نتیجه نرخ یادگیری ۰.۰۰۱ نتیجه بهتری دارد.

۵. تاثیر بیش‌برازش و کم‌برازش:

همانطور که در قبل اشاره شد. در قسمت‌هایی مشاهده می‌شود که علی رغم افزایش دقت در داده‌های آموزش، دقت در داده‌های اعتبارسنجی، روندی نزولی دارد که این نشان از فیت شدن بیش از حد روی داده‌های آموزش است و مدل برای داده‌های تست، به خوبی عمل نمی‌کند.

در سمت مقابل کم برازش وجود دارد که زمانی رخ می‌دهد که هنوز مدل به خوبی آموزش ندیده و داده‌های تست و آموزش هردو دقت پایینی دارند.

۶. شرایط توقف:

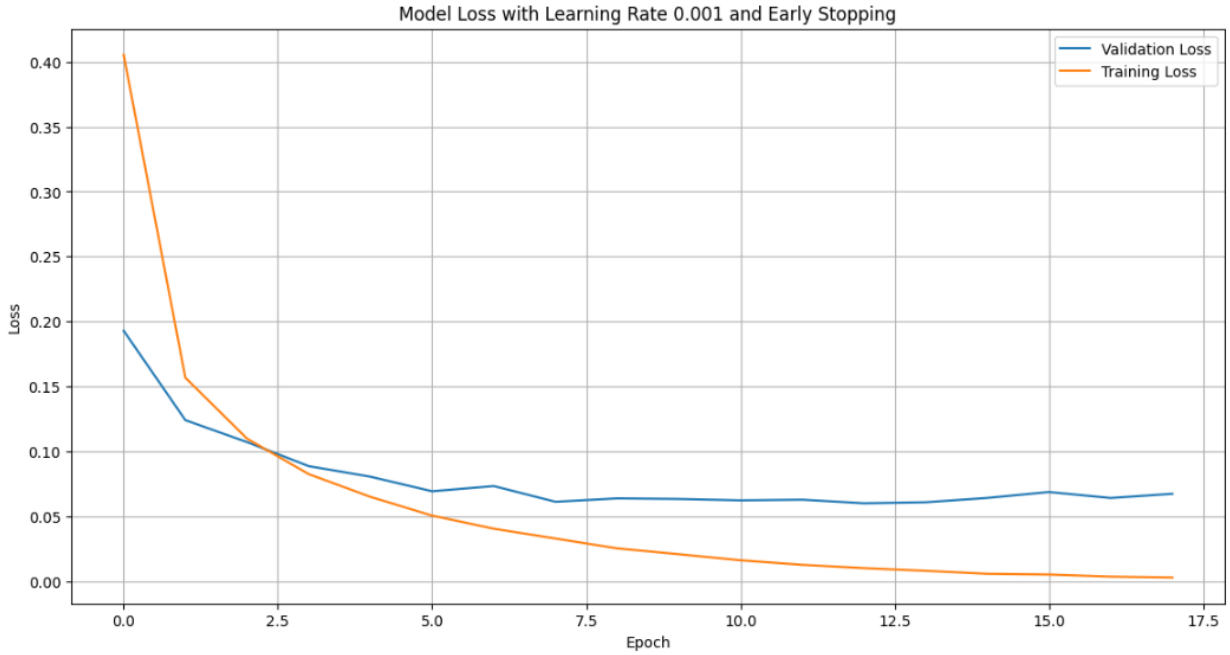
در این قسمت ما باتوجه به روند دقت آموزش در مقایل دادهای اعتبارسنجی یک شرط توقف قرار دادیم.

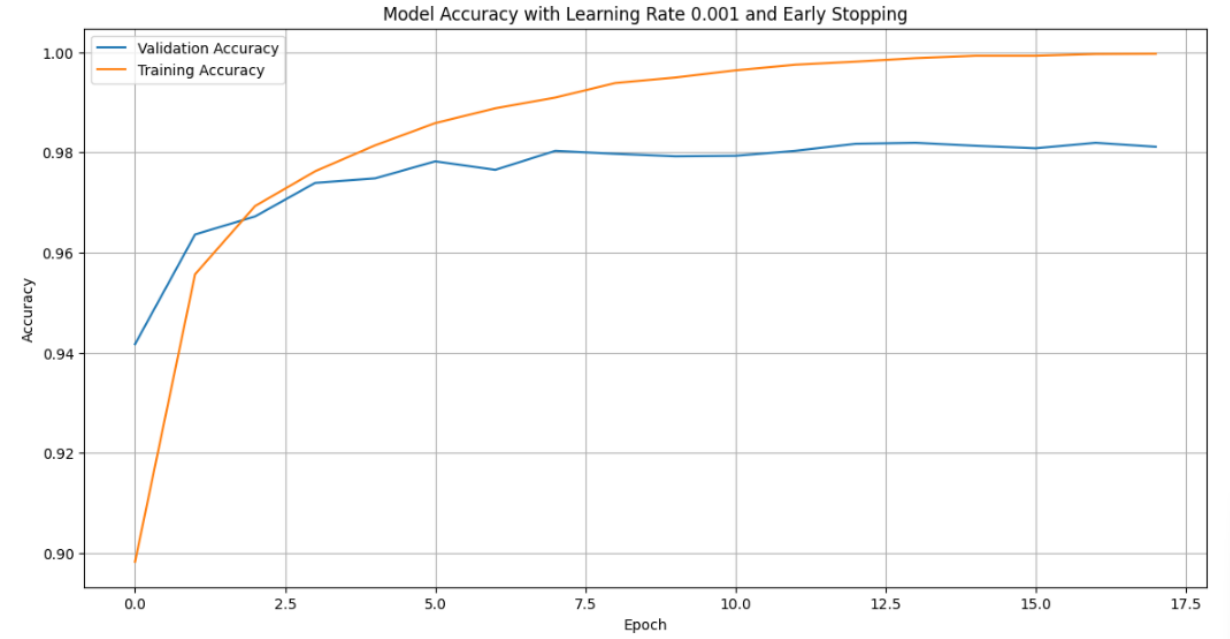
```
Epoch 8/40
469/469 - 5s - 11ms/step - accuracy: 0.9909 - loss: 0.0331 - val_accuracy: 0.9803 - val_loss: 0.0614
Epoch 9/40
469/469 - 5s - 11ms/step - accuracy: 0.9938 - loss: 0.0255 - val_accuracy: 0.9797 - val_loss: 0.0640
Epoch 10/40
469/469 - 5s - 11ms/step - accuracy: 0.9949 - loss: 0.0210 - val_accuracy: 0.9792 - val_loss: 0.0635
Epoch 11/40
469/469 - 5s - 11ms/step - accuracy: 0.9964 - loss: 0.0163 - val_accuracy: 0.9793 - val_loss: 0.0624
Epoch 12/40
469/469 - 5s - 11ms/step - accuracy: 0.9975 - loss: 0.0129 - val_accuracy: 0.9803 - val_loss: 0.0630
Epoch 13/40
469/469 - 5s - 11ms/step - accuracy: 0.9981 - loss: 0.0103 - val_accuracy: 0.9817 - val_loss: 0.0602
Epoch 14/40
469/469 - 5s - 11ms/step - accuracy: 0.9988 - loss: 0.0082 - val_accuracy: 0.9819 - val_loss: 0.0610
Epoch 15/40
469/469 - 5s - 11ms/step - accuracy: 0.9993 - loss: 0.0059 - val_accuracy: 0.9813 - val_loss: 0.0643
Epoch 16/40
469/469 - 6s - 13ms/step - accuracy: 0.9993 - loss: 0.0054 - val_accuracy: 0.9808 - val_loss: 0.0689
Epoch 17/40
469/469 - 6s - 13ms/step - accuracy: 0.9996 - loss: 0.0038 - val_accuracy: 0.9819 - val_loss: 0.0643
Epoch 18/40
469/469 - 6s - 13ms/step - accuracy: 0.9997 - loss: 0.0031 - val_accuracy: 0.9811 - val_loss: 0.0675
```

همانطور که در شکل بالا مشخص است در یک بازه علی‌رغم رشد دقت داده‌های آموزش تا نزدیک ۱، داده های اعتبارسنجی رشد خاصی نکردند. پس ما از یک شرط توقف مطابق شکل زیر استفاده کردیم که باعث افزایش سرعت و بهینه شدن روند آموزش می‌شود.

```
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
```

در اینجا ما معیار شرط توقف را loss گذاشتیم و حداکثر تعداد دورهای بدون پیشرفت را ۵ گذاشتیم.

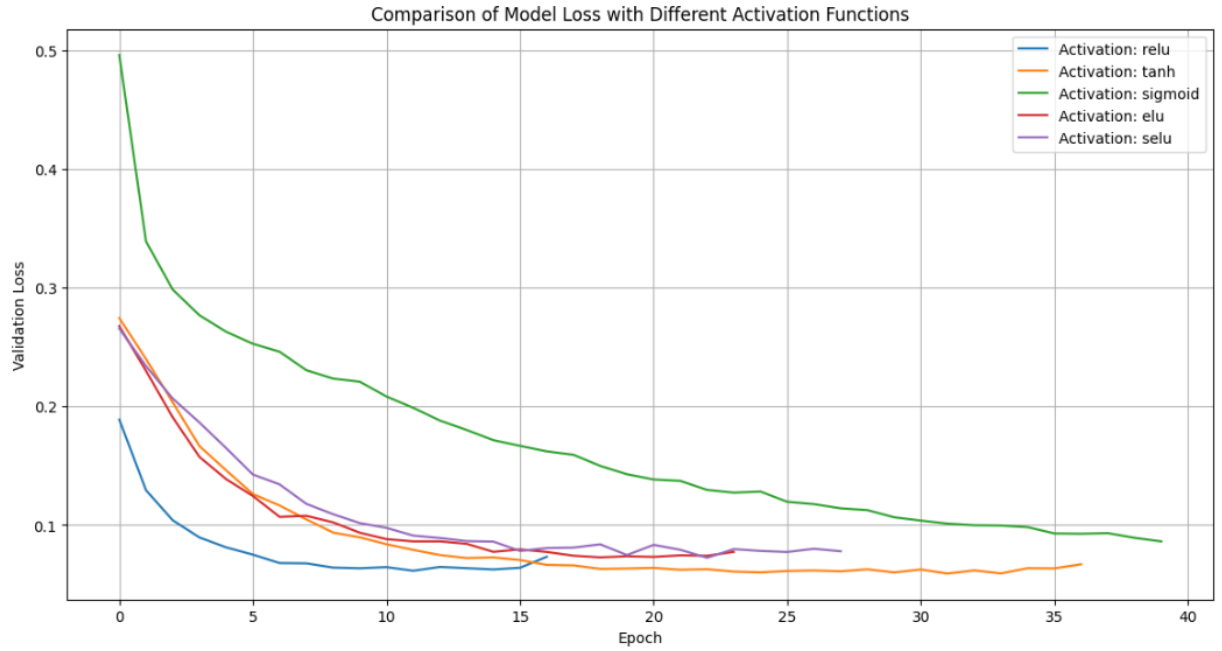


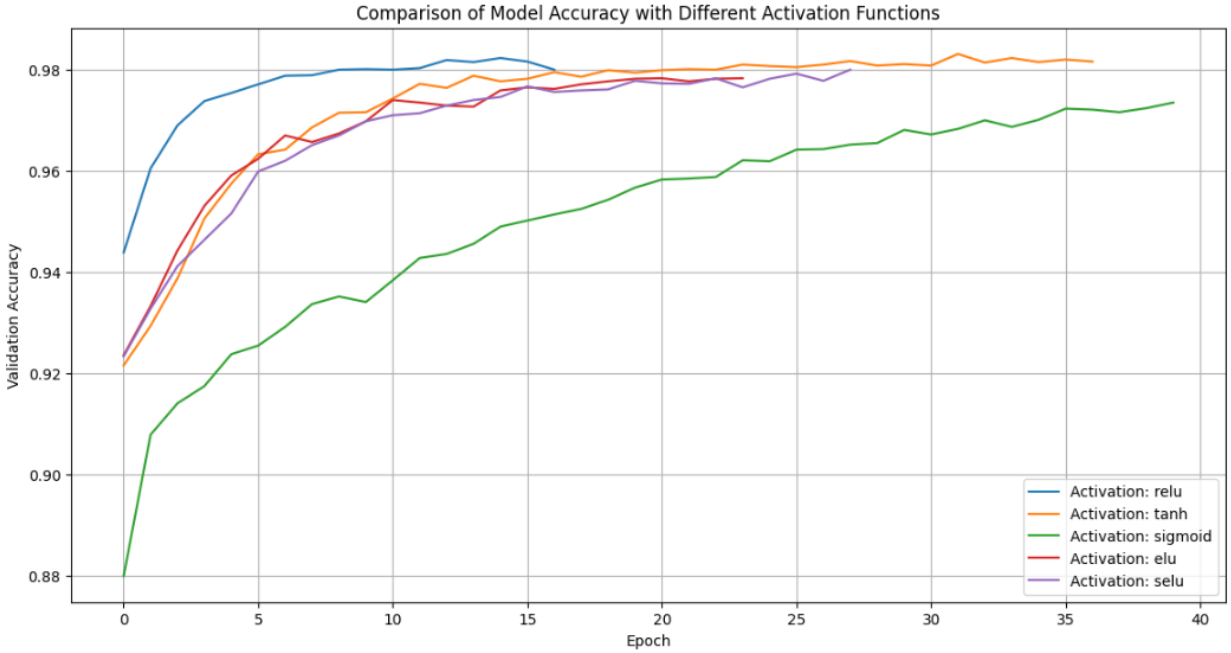


همانطور که مشاهده می‌شود کاهش loss و رشد accuracy برای داده‌های اعتبارسنجی از یک جا به بعد متوقف می‌شود.

۷. توابع فعالساز:

در این قسمت ما از توابع فعالساز مختلف استفاده کردیم که نتایج آن در نمودارهای زیر قابل مشاهده است.





از نمودارهای بالا به طور واضح مشخص است که تابع relu، بهترین عملکرد را ارائه می‌دهد.

اولین نکته سرعت همگرایی در relu است که به دلیل سادگی خود تابع Relu است همچنین به دلیل غیرخطی بودن آن، به سرعت یادگیری روابط پیچیده بین داده‌ها کمک می‌کند.

در ادامه ویژگی‌های هر یک از توابع فعالساز را به طور کلی بررسی می‌کنیم:

ReLU (Rectified Linear Unit):

ReLU به طور معمول بهترین عملکرد را در شبکه‌های عصبی عمیق نشان می‌دهد زیرا مشکلات ناپایداری گرادیان (vanishing gradient) را کاهش می‌دهد.

Tanh (Hyperbolic Tangent):

نزدیک به ReLU عمل می‌کند، اما در برخی از نقاط مقدار Validation Loss کمی بالاتر است.

tanh خروجی‌های متقارن حول صفر تولید می‌کند که ممکن است به آموزش بهتر در بعضی موارد کمک کند.

Sigmoid:

کمترین عملکرد را در میان توابع فعال‌سازی دیگر نشان می‌دهد.

Sigmoid می‌تواند مشکلات ناپایداری گرادیان (vanishing gradient) را در شبکه‌های عمیق‌تر ایجاد کند، که باعث کاهش کارآیی آموزش می‌شود.

ELU (Exponential Linear Unit):

عملکرد خوبی دارد و نزدیک به ReLU عمل می‌کند.

ELU برای مقابله با مشکلات منفی بودن گرادیان و بهبود توانایی مدل در آموزش، طراحی شده است.

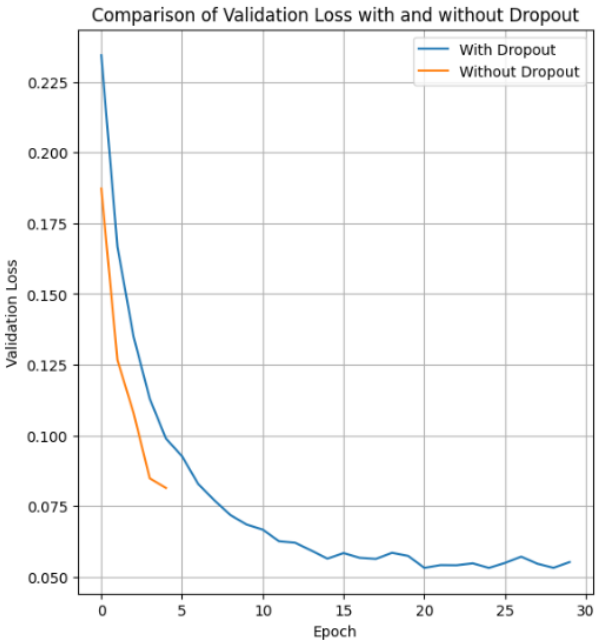
SELU (Scaled Exponential Linear Unit):

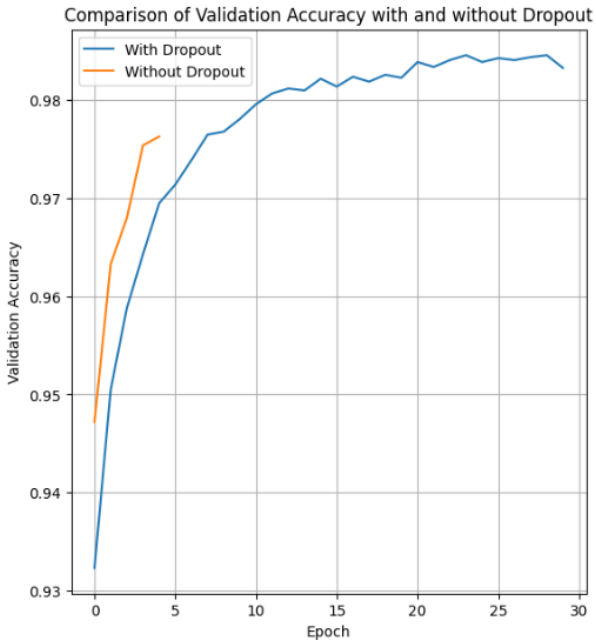
عملکرد خوبی دارد، اما در بعضی نقاط Validation Loss بیشتری نسبت به ReLU و ELU دارد.

SELU با خاصیت خود نرمال‌سازی به تثبیت و تسریع فرآیند آموزش کمک می‌کند.

۸. Dropout:

در این قسمت عملکرد آمورش را با داشتن dropout و بدون داشتن آن مقایسه کرده ایم.





همان‌طور که مشاهده می‌شود، مدل با Dropout در نهایت به دقت بالاتری نسبت به مدل بدون Dropout می‌رسد. این نشان می‌دهد که مدل بهتر قادر به تعمیم‌پذیری و عملکرد روی داده‌های جدید است.

مورد دیگر که در نمودار مشخص است این که ، مدل با Dropout در ابتدا به آهستگی loss و accuracy خود را بهبود می‌دهد که این به دلیل حذف تصادفی نورون‌ها و تغییرات مکرر در ساختار مدل است.

مدل با Dropout به دلیل حذف تصادفی نورون‌ها در طول آموزش، از overfitting جلوگیری می‌کند. این باعث می‌شود که شبکه نتواند ویژگی‌های خاص و دقیق آموزش دیده را به طور کامل حفظ کند، و به جای آن، ویژگی‌های کلی‌تر و برجسته‌تری را کشف کند که برای مجموعه‌داده‌های دیگر نیز قابل تعمیم باشند.

به این صورت می‌تواند به خوبی از فیت شدن مدل روی داده‌های آموزشی جلوگیری کند.

۹. Batch normalization:

در Batch Normalization ورودی‌های هر لایه را با تنظیم و مقیاس‌دهی به گونه‌ای استانداردسازی می‌کند که میانگین آنها صفر و انحراف معیار آنها یک شود.

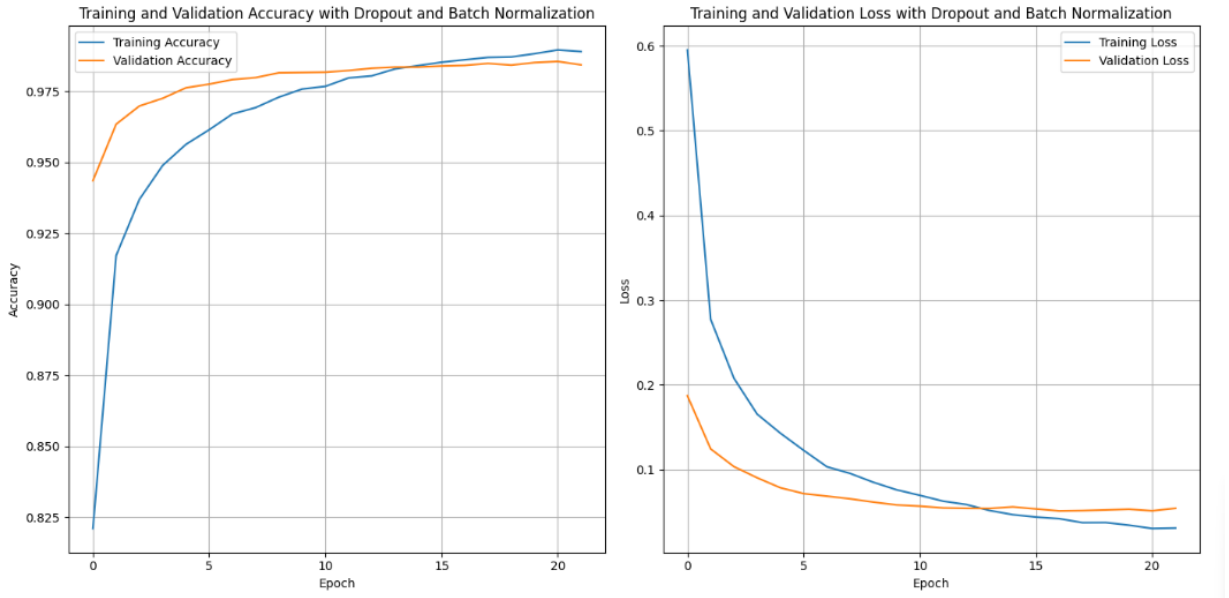
مورد دیگری که در نمودار نهایی قابل مشاهده است پایداری روند یادگیری است که از اثرات batch normalization است که در اثر آن می‌توان از نرخ یادگیری بیشتری استفاده کرد تا روند همگرایی سرعت پیدا کند.

بهبود جریان گرادیان: Batch Normalization مشکل گرادیان‌های ناپایدار (گرادیان‌های ناپایدار و بالا یا پایین) را کاهش می‌دهد. این کار با استفاده از استانداردهای محاسبه شده برای هر مینی‌بچ، ورودی‌ها را استانداردسازی کرده و تغییرات قابل پیش‌بینی‌تری را در گرادیان‌ها در هنگام backpropagation فراهم می‌کند.

چگونگی عملکرد Batch Normalization:

در زمان آموزش: برای هر مینی‌بچ، Batch Normalization میانگین و انحراف معیار فعال‌سازی‌ها را محاسبه می‌کند. سپس فعال‌سازی‌ها را با استفاده از این آمارهای محاسبه شده استانداردسازی می‌کند و پارامترهای آموزش‌یافته (گاما و بتا) را برای مقیاس‌دهی و جابه‌جایی فعال‌سازی‌های استانداردسازی شده استفاده می‌کند.

نمودار نهایی:



روند آموزش:

```
Epoch 1/40
469/469 - 8s - 17ms/step - accuracy: 0.8209 - loss: 0.5953 - val_accuracy: 0.9435 - val_loss: 0.1871
Epoch 2/40
469/469 - 7s - 14ms/step - accuracy: 0.9171 - loss: 0.2774 - val_accuracy: 0.9634 - val_loss: 0.1245
Epoch 3/40
469/469 - 6s - 14ms/step - accuracy: 0.9369 - loss: 0.2080 - val_accuracy: 0.9698 - val_loss: 0.1036
Epoch 4/40
469/469 - 6s - 14ms/step - accuracy: 0.9488 - loss: 0.1655 - val_accuracy: 0.9725 - val_loss: 0.0904
Epoch 5/40
469/469 - 6s - 13ms/step - accuracy: 0.9563 - loss: 0.1430 - val_accuracy: 0.9762 - val_loss: 0.0786
Epoch 6/40
469/469 - 6s - 12ms/step - accuracy: 0.9614 - loss: 0.1228 - val_accuracy: 0.9775 - val_loss: 0.0717
Epoch 7/40
469/469 - 6s - 13ms/step - accuracy: 0.9670 - loss: 0.1034 - val_accuracy: 0.9791 - val_loss: 0.0687
Epoch 8/40
469/469 - 6s - 13ms/step - accuracy: 0.9692 - loss: 0.0955 - val_accuracy: 0.9798 - val_loss: 0.0656
Epoch 9/40
469/469 - 6s - 12ms/step - accuracy: 0.9729 - loss: 0.0851 - val_accuracy: 0.9815 - val_loss: 0.0617
Epoch 10/40
469/469 - 6s - 13ms/step - accuracy: 0.9758 - loss: 0.0762 - val_accuracy: 0.9816 - val_loss: 0.0583
Epoch 11/40
469/469 - 6s - 12ms/step - accuracy: 0.9767 - loss: 0.0697 - val_accuracy: 0.9817 - val_loss: 0.0569
Epoch 12/40
469/469 - 6s - 12ms/step - accuracy: 0.9797 - loss: 0.0628 - val_accuracy: 0.9823 - val_loss: 0.0549
Epoch 13/40
469/469 - 6s - 12ms/step - accuracy: 0.9804 - loss: 0.0588 - val_accuracy: 0.9831 - val_loss: 0.0544
Epoch 14/40
469/469 - 6s - 12ms/step - accuracy: 0.9829 - loss: 0.0518 - val_accuracy: 0.9835 - val_loss: 0.0543
Epoch 15/40
469/469 - 6s - 12ms/step - accuracy: 0.9840 - loss: 0.0469 - val_accuracy: 0.9835 - val_loss: 0.0561
Epoch 16/40
469/469 - 6s - 13ms/step - accuracy: 0.9852 - loss: 0.0441 - val_accuracy: 0.9839 - val_loss: 0.0537
Epoch 17/40
469/469 - 7s - 14ms/step - accuracy: 0.9861 - loss: 0.0421 - val_accuracy: 0.9841 - val_loss: 0.0513
Epoch 18/40
469/469 - 7s - 15ms/step - accuracy: 0.9869 - loss: 0.0375 - val_accuracy: 0.9848 - val_loss: 0.0517
Epoch 19/40
469/469 - 7s - 15ms/step - accuracy: 0.9871 - loss: 0.0376 - val_accuracy: 0.9842 - val_loss: 0.0524
Epoch 20/40
469/469 - 6s - 14ms/step - accuracy: 0.9882 - loss: 0.0345 - val_accuracy: 0.9851 - val_loss: 0.0533
Epoch 21/40
469/469 - 6s - 14ms/step - accuracy: 0.9895 - loss: 0.0305 - val_accuracy: 0.9855 - val_loss: 0.0514
Epoch 22/40
469/469 - 6s - 13ms/step - accuracy: 0.9890 - loss: 0.0311 - val_accuracy: 0.9843 - val_loss: 0.0544
```

بهترین دقت در **validation**: **0.9855**

کمترین میزان خطا در **validation**: **0.0514**

قطعه کد نهایی:

```
# Load MNIST data
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Normalize the images
train_images = train_images / 255.0
test_images = test_images / 255.0

# Convert labels to one-hot encoding
train_labels = tf.keras.utils.to_categorical(train_labels, 10)
test_labels = tf.keras.utils.to_categorical(test_labels, 10)
```

```
#batch normalization

# Define the model with Dropout and Batch Normalization
def create_model():
    model = Sequential()
    model.add(Flatten(input_shape=(28, 28)))
    model.add(Dense(1024, activation='relu'))
    model.add(BatchNormalization())  # Add Batch Normalization
    model.add(Dropout(0.5))  # Add Dropout with rate 0.5

    model.add(Dense(1024, activation='relu'))
    model.add(BatchNormalization())  # Add Batch Normalization
    model.add(Dropout(0.5))  # Add Dropout with rate 0.5

    model.add(Dense(10, activation='softmax'))
    return model

# Hyperparameters
learning_rate = 0.0001
epochs = 40
batch_size = 128
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

# Create the model
model = create_model()
model.compile(optimizer=Adam(learning_rate=learning_rate),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
history = model.fit(train_images, train_labels, epochs=epochs, batch_size=batch_size,
                    validation_data=(test_images, test_labels), callbacks=[early_stopping], verbose=2)
```