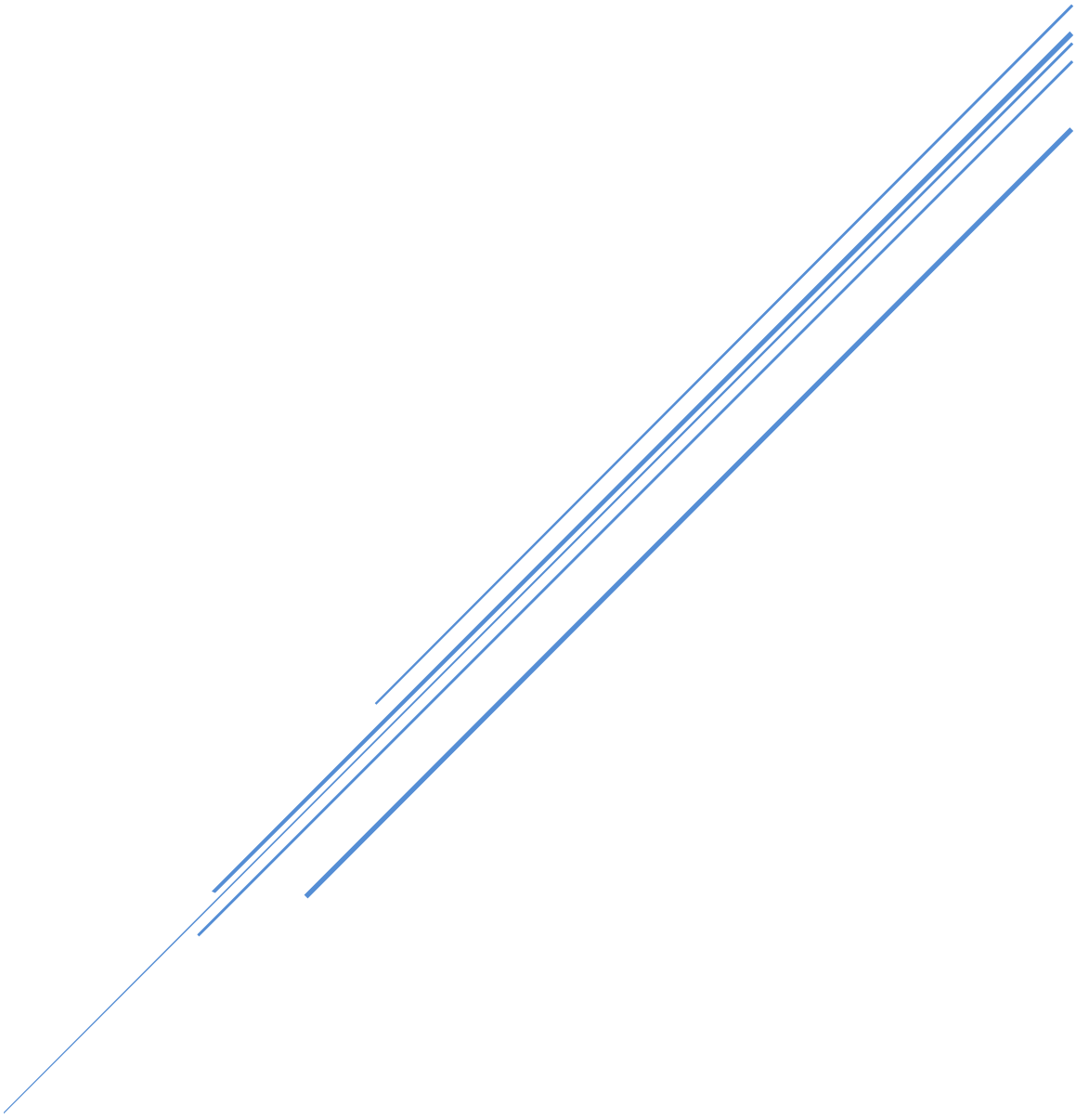
مستندات پروژه مدل شناسایی MNIST

آرمان خلیلی – 4003623016

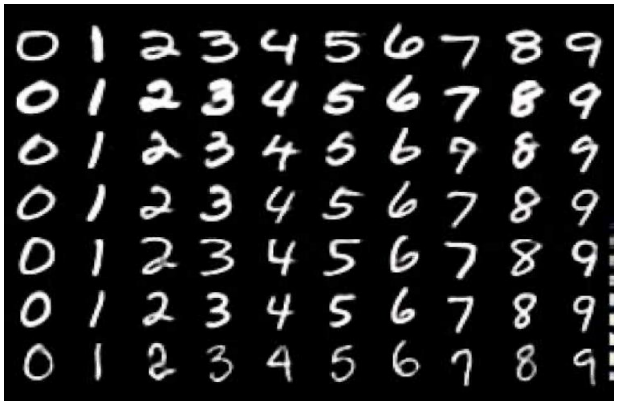


دانشگاه اصفهان

درس مبانی هوش محاسباتی – دکتر هادی تابع الحجه

مقدمه

MNIST مخفف Modified National Institute of Standards and Technology است و یک پایگاه داده بزرگ از اعداد دست‌نویس است که به طور گسترده‌ای برای آموزش و آزمایش در زمینه یادگیری ماشین استفاده می‌شود. این مجموعه شامل 60,000 تصویر آموزشی و 10,000 تصویر آزمایشی است که هر کدام به صورت تصاویر خاکستری 28*28 پیکسلی هستند. این تصاویر از پایگاه داده‌های اصلی NIST بازآمیخته شده‌اند تا برای آزمایش‌های یادگیری ماشین مناسب‌تر باشند و به دلیل سادگی و اندازه مناسب، برای شروع کار با یادگیری عمیق و پردازش تصویر بسیار محبوب است.



نمونه ای از ارقام دستنویس در MNIST

ما نیز در این پروژه از این مجموعه داده معتبر برای آموزش و تست مدل پیشبینی ارقام خود استفاده کرده‌ایم، برای این امر ابتدا این مجوعه داده را از وبسایت **Yann LeCun** دانلود کرده و بصورت زیر آنرا در دو دسته داده های آموزش و داده های آزمایش بارگیری کرده ایم:

```
import idx2numpy as dxpy
RESOURCES = [
        '.\\train-images.idx3-ubyte', '.\\train-labels.idx1-ubyte',
        '.\\t10k-images.idx3-ubyte',  '.\\t10k-labels.idx1-ubyte']
train_images = dxpy.convert_from_file(RESOURCES[0])
train_labels = dxpy.convert_from_file(RESOURCES[1])
test_images = dxpy.convert_from_file(RESOURCES[2])
test_labels = dxpy.convert_from_file(RESOURCES[3])
```

پیش‌پردازش داده‌ها

پیش‌پردازش داده ها به مدل کمک می‌کنند تا الگوهای موجود در تصاویر را بهتر یاد بگیرد و دقت بالاتری در تشخیص اعداد دست‌نویس داشته باشد. همچنین، این مراحل به اطمینان از اینکه مدل قابلیت تعمیم به داده‌های جدید را دارد، کمک می‌کنند. پیش‌پردازش داده‌ها در مجموعه داده‌های MNIST شامل چندین مرحله است که این مراحل عبارتند از:

۱) اصلاح ابعاد داده ها

برای استفاده از داده ها در یک شبکه عصبی معمولی (مانند شبکه‌های کاملاً متصل)، نیاز است که این تصاویر به آرایه‌های یک بعدی تبدیل شوند پس لازم است در قدم اول تصاویر از حالت دو بعدی ۲۸*۲۸ به فرمت یک بعدی یعنی یک ردیف ۷۸۴ تایی دربیاوریم. برای این امر از تابع reshape موجود در کتابخانه numpy استفاده میکنیم:

```
train_images = train_images.reshape(60000, -1)
test_images = test_images.reshape(10000, -1)
```

به این ترتیب تصاویر آموزشی و آزمایشی را به ترتیب به آرایه‌هایی با اندازه‌های (۶۰۰۰۰, ۷۸۴) و (۱۰۰۰۰, ۷۸۴) تبدیل می‌کنند. این کار برای آماده‌سازی داده‌ها برای ورود به مدل یادگیری ماشین ضروری است.

پس از اجرای این کد میتوانیم بصورت زیر ابعاد جدید داده هارا مشاهده کنیم:

```
print(f"Train images shape: {train_images.shape}   |    Train labels shape:
{train_labels.shape}")
print(f"Test images shape: {test_images.shape}    |   Test labels shape:
{test_labels.shape}")
```

```
Train images shape: (60000, 784)   |    Train labels shape: (60000,)
Test images shape: (10000, 784)    |   Test labels shape: (10000,)
```

۲) نرمال‌سازی داده‌ها:

نرمالایز کردن یک مرحله مهم در پیش‌پردازش داده‌ها است که به مقیاس‌بندی داده‌ها کمک می‌کند تا مقادیر پیکسل‌ها در بازه استانداردی قرار گیرند، که این امر به الگوریتم‌های یادگیری ماشین کمک می‌کند تا بهتر عمل کنند. در مجموعه دادهMNIST ، مقادیر پیکسل‌ها بین ۰ تا ۲۵۵ هستند، که نشان‌دهنده شدت خاکستری هستند. با تقسیم تمام مقادیر پیکسل‌ها بر ۲۵۵، مقادیر نرمال‌شده بین ۰ تا ۱ قرار می‌گیرند. این کار باعث می‌شود که شبکه عصبی بهتر و سریع‌تر همگرا شود، زیرا وزن‌ها می‌توانند در مقیاس‌های کوچکتر و با گام‌های کوچکتر به‌روزرسانی شوند.

```
train_images = train_images / 255
test_images = test_images / 255
```

با اجرای کد زیر میتوانیم مقدار حداکثر و حداقل داده ها در مقیاس جدید را مشاهده کنیم:

```
print("Training images pixel values range from", train_images.min(), "to",
train_images.max())
print("Test images pixel values range from", test_images.min(), "to",
test_images.max())
```

```
Training images pixel values range from 0.0 to 1.0
Test images pixel values range from 0.0 to 1.0
```

۳) کد گذاری برچسب‌ها :

یکی دیگر از مراحل پیش‌پردازش تبدیل برچسب‌های داده‌های آموزشی و آزمایشی از حالت عددی به حالت آرایه‌ای است. این تبدیل برای آماده‌سازی داده‌ها برای مدل‌های شبکه عصبی است که از تابع فعال‌سازی softmax در لایه خروجی استفاده می‌کنند، که می‌توانند احتمال تعلق به هر کلاس را برای یک نمونه پیش‌بینی کنند.

برای اینکار از روش کدگذاری one-hot encoding استفاده میکنیم تا هر برچسب به یک آرایه تبدیل شود که در آن تمام مقادیر به جز یکی صفر هستند. این مقدار غیر صفر نشان‌دهنده کلاس فعلی برچسب است. به عنوان مثال، اگر برچسب عددی ۳ باشد، پس از تبدیل به فرمت one-hot آرایه به صورت [0, 0, 0, 1, 0, 0, 0, 0, 0, 0] خواهد بود، که در آن عدد ۱ در موقعیت چهارم (شمارش از صفر) قرار دارد که نشان‌دهنده کلاس ۳ است. این کدگذاری بصورت زیر انجام میشود:

```
from keras import utils
train_labels =utils.to_categorical(train_labels, num_classes=10)
test_labels = utils.to_categorical(test_labels, num_classes=10)
```

برچسب‌های one-hot با خروجی‌های مدل مطابقت دارند و می‌توان از آن‌ها برای محاسبه تابع هزینه و بهینه‌سازی مدل استفاده کرد.

۴) K-Fold Cross-Validation :

یکی دیگر از عملیات‌های قابل انجام در مرحله پیش پردازش اعتبار سنجی متقابل یا Cross-Validation است، که یک تکنیک ارزیابی مدل در یادگیری ماشین است که برای اطمینان از قابلیت تعمیم مدل به داده‌های جدید استفاده می‌شود. در این تکنیک ، داده‌ها به **k**بخش تقسیم می‌شوند و در هر دور، یک بخش به عنوان داده‌های آزمایشی و بقیه به عنوان داده‌های آموزشی استفاده می‌شوند. این فرآیند برای **k**دور تکرار می‌شود و هر بار یک بخش جدید به عنوان داده‌های آزمایشی انتخاب می‌شود.

در مورد مجموعه داده‌هایMNIST ، استفاده از K-Fold Cross-Validation به ما این امکان را می‌دهد که دقت مدل خود را بر روی بخش‌های مختلفی از داده‌ها ارزیابی کنیم و از این طریق، اطمینان حاصل کنیم که مدل فقط بر روی یک بخش خاص از داده‌ها خوب عمل نمی‌کند، بلکه قادر است به طور کلی تعمیم پیدا کند. نحوه پیاده سازی این تکنیک بصورت زیر است:

```
from sklearn.model_selection import KFold

n_folds = 5
# prepare cross validation
kfold = KFold(n_folds, shuffle=True, random_state=1)
# assigning data
for train_ix, test_ix in kfold.split(train_images):
    X_train, X_test = train_images[train_ix], train_images[test_ix]
    y_train, y_test = train_labels[train_ix], train_labels[test_ix]
```

n_folds= 5 به این معنی است که داده‌ها به ۵ بخش تقسیم می‌شوند و مدل برای ۵ دور مختلف آموزش داده می‌شود. در هر دور، یک بخش به عنوان داده‌های آزمایشی و ۴ بخش دیگر به عنوان داده‌های آموزشی استفاده می‌شوند. این کار باعث می‌شود که مدل بتواند بر روی داده‌های متنوع‌تری آزمایش شود و نتایج دقیق‌تری ارائه دهد.

پیاده‌سازی مدل

مدل پیاده سازی شده در این پروژه یک پرسپترون چند لایه (MLP) است که به یک نوع خاص از شبکه عصبی با نورون‌های تماما متصل (Fully Connected Neural Network) و سه نوع لایه ورودی، نهان و خروجی می‌باشد. پیاده سازی این مدل با استفاده از کتابخانه Keras در فریم ورک TensorFlow انجام گرفته و مدل نهایی از ۶ عدد لایه با تعداد نرون های متفاوت تشکیل شده است. کد این مدل از طریق لینک گیت‌هاب قابل دسترس است. که قسمت های مختلف آن در ادامه توضیح داده شده است.

```
from keras import Sequential, layers, optimizers, regularizers

# Define the model
model = Sequential()
model.add(layers.Dense(2048, activation='relu', input_dim=784 ,
kernel_regularizer=regularizers.l2(0.001)))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1024, activation='relu'))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))


opt = optimizers.SGD(learning_rate = 0.001, momentum = 0.9)


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

# Train the model
history = model.fit(train_images, train_labels,
validation_data=(test_images, test_labels), epochs=25, batch_size=100)

# Make predictions
predictions_probabilities = model.predict(test_images)
predictions = np.argmax(predictions_probabilities, axis=1)
```

مدل : مدل به صورت ترتیبی (Sequential) تعریف شده است، به این معنی که لایه‌ها یکی پس از دیگری اضافه می‌شوند.

- **لایه‌ها** : هر Denseنشان‌دهنده یک لایه کاملاً متصل است که هر نورون در آن به تمام نورون‌های لایه قبلی متصل است. تعداد نورون‌ها در هر لایه به ترتیب۲۰۴۸، ۱۰۲۴, ۵۱۲, ۲۵۶, ۱۲۸, و ۶۴ است. تابع فعال‌سازی reluبرای افزودن خاصیت غیرخطی به مدل استفاده می‌شود.

- **ورودی**: input_dim=۷۸۴ نشان‌دهنده تعداد ویژگی‌های ورودی است که برابر با تعداد پیکسل‌های تصاویر (۲۸x۲۸) MNIST است.

- **لایه خروجی** :آخرین لایه دارای ۱۰ نورون است که هر کدام نمایانگر یکی از اعداد ۰ تا ۹ هستند. تابع فعال‌سازی softmax احتمالاتی را برای هر کلاس تولید می‌کند و کلاسی که بیشترین تکرار یا احتمال را دارد انتخاب میکند.

- **بهینه‌ساز**SGD : با نرخ یادگیری ۰.۰۱ برای به‌روزرسانی وزن‌ها در طول آموزش استفاده می‌شود.

- **کامپایل**: مدل با استفاده از بهینه‌ساز ، تابع زیان categorical_crossentropy (مناسب برای مسائل طبقه‌بندی چندکلاسه) و معیار accuracyکامپایل می‌شود.

- **آموزش** : مدل با استفاده از داده‌های آموزشی و آزمایشی برای ۲۵ دوره (epochs) و با اندازه دسته ۱۰۰ (batch_size) آموزش داده می‌شود.

- **پیش‌بینی** : پس از آموزش، مدل برای تولید احتمالات کلاس‌ها بر روی داده‌های آزمایشی استفاده می‌شود و سپس با استفاده از np.argmax، کلاس با بیشترین احتمال به عنوان پیش‌بینی نهایی انتخاب می‌شود.

ارزیابی مدل

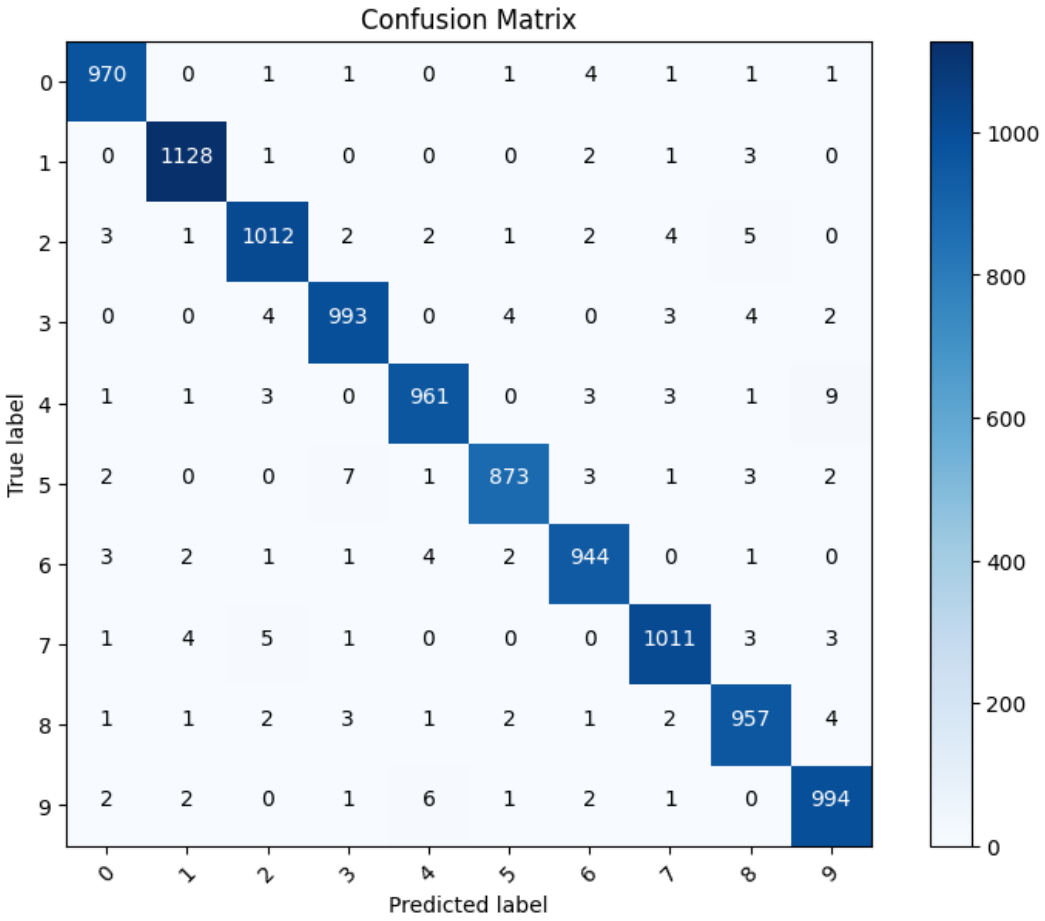
ارزیابی مدل یکی از مراحل کلیدی در فرآیند یادگیری ماشین است که به ما امکان می‌دهد تا عملکرد مدل را در مقابله با داده‌های جدید بسنجیم. این ارزیابی به ما کمک می‌کند تا درک کنیم که مدل چقدر خوب می‌تواند پیش‌بینی‌های دقیق انجام دهد. در این پروژه، از معیارهای مختلفی برای ارزیابی مدل MNIST استفاده شده است:

- **دقت (Accuracy)** نشان‌دهنده درصد نمونه‌هایی است که به درستی طبقه‌بندی شده‌اند.
- **دقت (Precision)** نشان‌دهنده درصد پیش‌بینی‌های صحیح در میان تمام پیش‌بینی‌هایی است که به عنوان یک کلاس خاص انجام شده‌اند.
- **حساسیت (Recall)** نشان‌دهنده درصد نمونه‌های واقعی یک کلاس خاص است که به درستی به عنوان آن کلاس پیش‌بینی شده‌اند.
- **نمره F۱ (F۱-Score)** میانگین هارمونیک دقت و حساسیت است که تعادل بین این دو را فراهم می‌کند.

که مقادیر این چهار معیار عبارتست از:

```
Accuracy: 98.18%    |   Recall: 98.18%
Precision: 98.18%   |   F1-Score: 98.18%
```

همچنین یکی دیگر از روش های ارزیابی دقیق‌تر مدل محاسبه و نمایش **ماتریس درهم‌ریختگی (Confusion Matrix)** است که یک جدول است که نشان می‌دهد مدل چگونه پیش‌بینی‌های خود را بر روی یک مجموعه داده آزمایشی انجام داده است. این ماتریس نشان می‌دهد که چه تعداد از نمونه‌ها به درستی یا نادرستی به عنوان هر کلاس پیش‌بینی شده‌اند.



ماتریس درهم‌ریختگی (Confusion Matrix)

همانطور که در کد زیر مشاهده میکنید در این پروژه، ابتدا مدل با استفاده از داده‌های آزمایشی ارزیابی می‌شود و دقت محاسبه می‌گردد. سپس، معیارهای دقت، حساسیت و نمره F۱ با استفاده از برچسب‌های واقعی و پیش‌بینی‌های مدل محاسبه می‌شوند. این معیارها به صورت وزن‌دار محاسبه می‌شوند، به این معنی که به هر کلاس بر اساس تعداد نمونه‌های آن در مجموعه داده وزن داده می‌شود.

در نهایت، ماتریس درهم‌ریختگی محاسبه و نمایش داده می‌شود. این ماتریس به ما اجازه می‌دهد تا ببینیم که کدام کلاس‌ها به درستی پیش‌بینی شده‌اند و کدام یک از کلاس‌ها با یکدیگر اشتباه

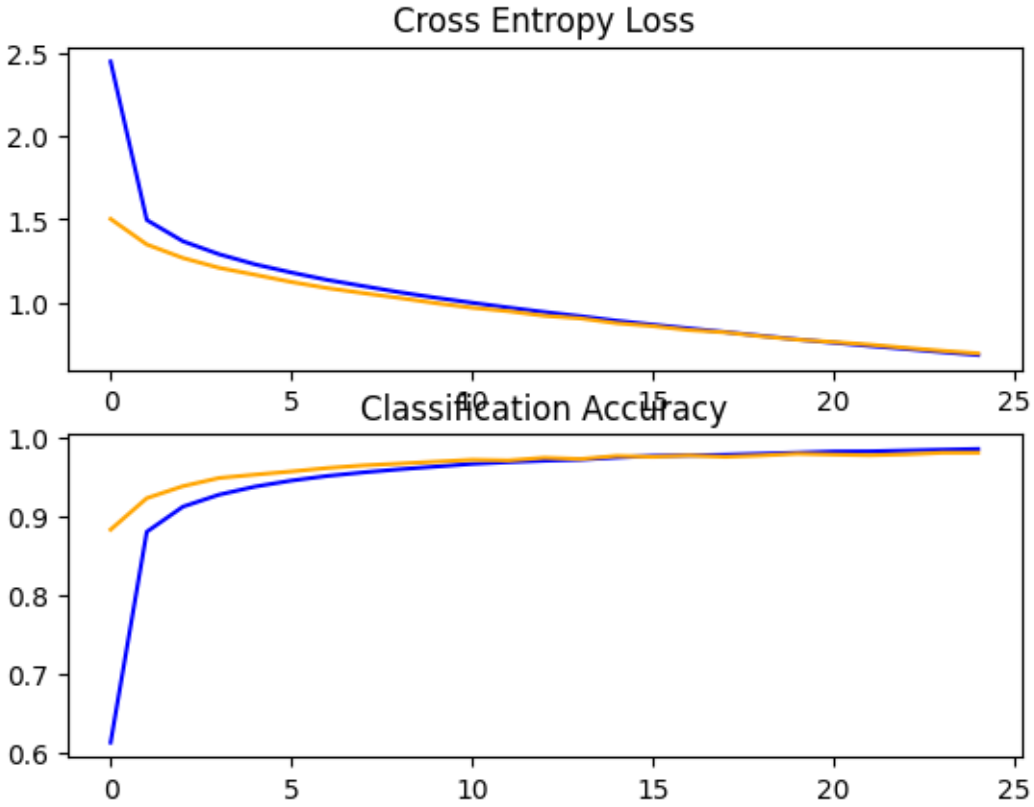
گرفته شده‌اند. این اطلاعات می‌توانند به بهبود مدل کمک کنند، زیرا می‌توانیم بر روی کلاس‌هایی که مدل در پیش‌بینی آن‌ها دشواری دارد، تمرکز کنیم:

```
from sklearn.metrics import confusion_matrix, precision_score, recall_score,
f1_score
import itertools
import numpy as np
from sklearn import metrics


# Evaluate the model
_, acc = model.evaluate(test_images, test_labels)
precision = precision_score(test_labels_integer_encoded, predictions,
average='weighted')
recall = recall_score(test_labels_integer_encoded, predictions,
average='weighted')
f1 = f1_score(test_labels_integer_encoded, predictions, average='weighted')

print(f'Accuracy: {acc * 100:.2f}%')
print(f"Precision: {precision * 100:.2f}%")
print(f"Recall (Sensitivity): {recall * 100:.2f}%")
print(f"F1-Score: {f1 * 100:.2f}%")


# Calculate and print the confusion matrix
class_names = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
cm = confusion_matrix(np.argmax(test_labels, axis=1), predictions)
plt.figure(figsize=(8, 6))
plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
plt.title('Confusion Matrix')
plt.colorbar()
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks, class_names, rotation=45)
plt.yticks(tick_marks, class_names)
thresh = cm.max() / 2.
for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, int(cm[i, j]),
             horizontalalignment="center",
             color="white" if cm[i, j] > thresh else "black")
plt.tight_layout()
plt.ylabel('True label')
plt.xlabel('Predicted label')
plt.show()
```

رسم نمودارهای فراگیری

نمودارهای Learning Curves ابزارهای بصری هستند که برای ارزیابی چگونگی یادگیری مدل در طول زمان استفاده می‌شوند. این نمودارها می‌توانند به شناسایی مشکلاتی مانند بیش‌برازش (overfitting) یا کم‌برازش (underfitting) کمک کنند و به تحلیل‌گران داده اجازه می‌دهند تا تنظیمات مدل را بهینه‌سازی کنند.

در اینجا نمودارهای **خطای متقاطع** (Cross Entropy Loss) و **دقت طبقه‌بندی** (Classification Accuracy) برای داده‌های آموزشی و آزمایشی مدل پیاده سازی شده رسم شده اند که در ادامه توضیحاتی برای هر مورد آورده شده است:



نمودار خطای متقاطع (Cross Entropy Loss) : نمودار خطا نشان‌دهنده میزان خطای مدل در طول زمان است. اگر نمودار خطای آموزشی به مرور زمان کاهش یابد و نمودار خطای آزمایشی ثابت بماند یا افزایش یابد، این می‌تواند نشانه‌ای از بیش‌برازش باشد.

نمودار دقت طبقه‌بندی(Classification Accuracy) : نمودار دقت نشان‌دهنده میزان دقت مدل در طول زمان است. اگر دقت آموزشی به مرور زمان افزایش یابد و دقت آزمایشی ثابت بماند یا کاهش یابد، این نیز می‌تواند نشانه‌ای از بیش‌برازش باشد.

با استفاده از این نمودارها، می‌توانید تصمیم بگیریم که آیا باید تعداد دوره‌های آموزشی (epochs) را افزایش دهید، تنظیمات مدل را تغییر دهید، یا از تکنیک‌هایی مانند **Dropout**یا **Regularization**برای جلوگیری از بیش‌برازش استفاده کنیم. این نمودارها ابزارهای قدرتمندی برای بهبود عملکرد مدل‌های یادگیری ماشین هستند.

در ادامه به توضیحاتی دقیق‌تر برای هر یک از مراحل پیاده سازی مدل و همچنین آزمایش انواع حالات پیاده سازی و نتایج آنها خواهیم پرداخت.

انتخاب تعداد لایه ها

انتخاب تعداد لایه‌ها در شبکه‌های عصبی (NN) یکی از جنبه‌های مهم طراحی مدل است که می‌تواند تأثیر زیادی بر عملکرد مدل داشته باشد. در اینجا به بررسی کلی این موضوع و سپس توجیه نحوه انتخاب لایه‌ها در مدل ارائه شده می‌پردازیم:

- پیچیدگی مسئله :برای مسائل پیچیده‌تر، ممکن است به تعداد بیشتری لایه نیاز باشد تا مدل بتواند ویژگی‌های پیچیده‌تری را یاد بگیرد.
- مقدار داده :با افزایش تعداد داده‌های آموزشی، می‌توان از مدل‌های عمیق‌تر استفاده کرد بدون اینکه نگران بیش‌برازش باشیم.
- قدرت محاسباتی :مدل‌های عمیق‌تر نیاز به منابع محاسباتی بیشتری دارند و زمان آموزش طولانی‌تری خواهند داشت.
- بیش‌برازش و کم‌برازش :افزایش تعداد لایه‌ها می‌تواند منجر به بیش‌برازش شود، در حالی که تعداد کم لایه‌ها ممکن است منجر به کم‌برازش شود.

در این پروژه مدل پیاده سازی شده دارای ۶ لایه است که به ترتیب دارای ۲۰۴۸، ۱۰۲۴، ۵۱۲، ۲۵۶، ۱۲۸، ۶۴، ۱۰ عدد نرون میباشند:

```
# Define the model
model = Sequential()
model.add(layers.Dense(2048, activation='relu', input_dim=784 ,
                       kernel_regularizer=regularizers.l2(0.001)))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1024, activation='relu'))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

بطور کلی لایه های این مدل به سه دسته تقسیم میشوند:

- **۱ عدد لایه ورودی** : یک لایه ورودی با تعداد نورون‌های۷۸۴ لازم است که متناسب با تعداد پیکسل‌های تصاویر MNIST است.
- **۵ عدد لایه‌های پنهان** : از آنجایی که MNIST یک مجموعه داده نسبتاً ساده است اما هنوز هم نیاز به یک مدل دارد که بتواند ویژگی‌های مختلف اعداد دست‌نویس را تشخیص دهد. پنج لایه پنهان به مدل اجازه می‌دهد تا ویژگی‌های مختلف را در سطوح مختلف انتزاعی‌تر یاد بگیرد.
- **۱ عدد لایه خروجی** : یک لایه خروجی در شبکه‌های عصبی هم برای تعیین نتیجه نهایی مدل ضروری است.

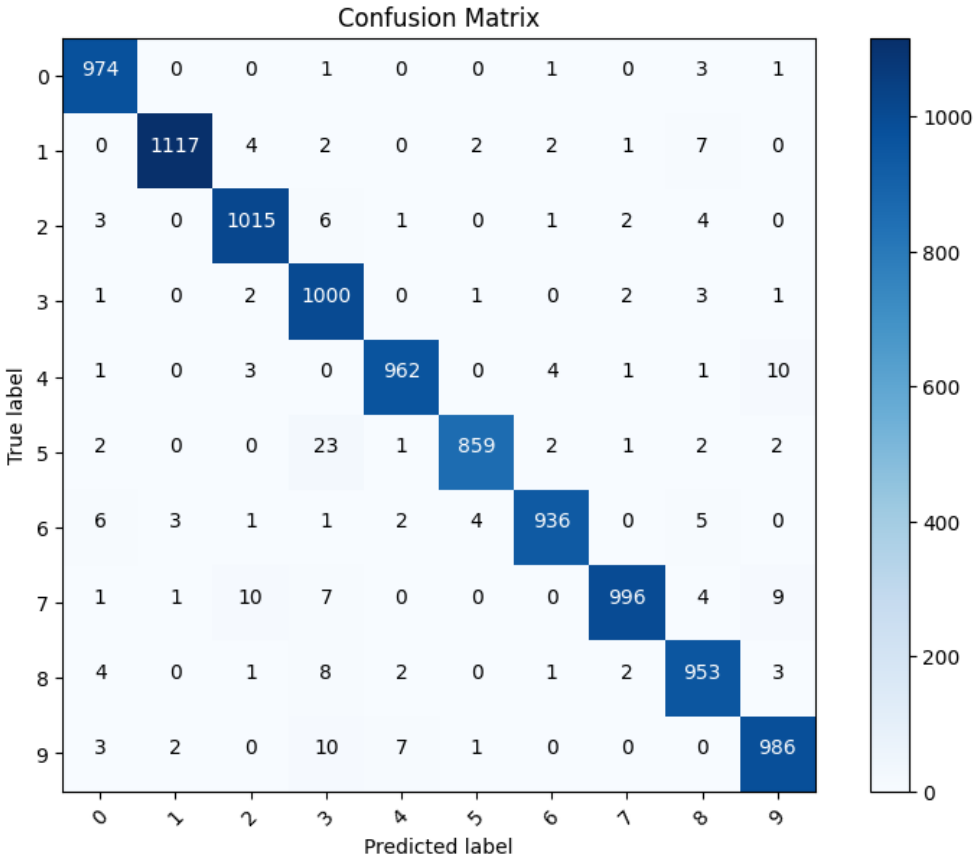
این معماری می‌تواند به مدل کمک کند تا ویژگی‌های مختلف تصاویر را در سطوح مختلف انتزاعی‌تر یاد بگیرد و به این ترتیب، توانایی تشخیص اعداد دست‌نویس را بهبود بخشد. با این حال، انتخاب تعداد لایه‌ها و تعداد نورون‌ها در هر لایه می‌تواند بر اساس آزمایش و خطا و تنظیمات مختلف بهینه‌سازی شود تا بهترین عملکرد را برای مسئله مورد نظر فراهم آورد.

برای مثال میتوانیم حالت ۳ لایه ای مدل را نیز به صورت زیر امتحان کنیم و سپس تاثیر آن در دقت مدل را مورد بررسی قرار دهیم:

```
# Define the model
model = Sequential()
model.add(layers.Dense(1024, activation='relu', input_dim=784,
                       kernel_regularizer=l2(0.001)))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

نتیجه:

```
Accuracy: 97.98%    |   Recall: 97.98%
Precision: 98.01%   |   F1-Score: 97.69%
```



همانطور که مشاهده کردید کاهش تعداد لایه های مدل موجب کم شدن پیچیدگی مدل و در نتیجه کاهش دقت آن شد، پس نتیجه میگیریم تعداد ۶ لایه در صورت کنترل و جلوگیری از بیش برازش انتخاب بهتری است.

تعداد نرون‌های هر لایه

انتخاب تعداد نورون‌ها در هر لایه از شبکه عصبی نیز مانند انتخاب تعداد لایه ها یکی از مهم‌ترین تصمیمات در طراحی مدل است که بر پیچیدگی و قدرت یادگیری مدل تأثیر می‌گذارد. ابتدا به بررسی کلی این موضوع و سپس توجیه نحوه انتخاب تعداد نورون‌ها در مدل ارائه شده می‌پردازیم:

- **قدرت تمثیل :** تعداد نورون‌ها باید به اندازه کافی باشد تا مدل بتواند ویژگی‌های داده‌ها را به خوبی تمثیل کند.
- **بیش‌برازش و کم‌برازش :** تعداد زیاد نورون‌ها می‌تواند منجر به بیش‌برازش شود، در حالی که تعداد کم ممکن است باعث کم‌برازش شود.
- **هزینه محاسباتی :** تعداد نورون‌های بیشتر نیازمند منابع محاسباتی بیشتری است و ممکن است زمان آموزش را افزایش دهد.

در این پروژه تعداد نرون های مدل پیاده سازی شده به ترتیب ۲۰۴۸، ۱۰۲۴، ۵۱۲، ۲۵۶، ۱۲۸، ۶۴، ۱۰ عدد نرون میباشند:

```
# Define the model
model = Sequential()
model.add(layers.Dense(2048, activation='relu', input_dim=784 ,
                       kernel_regularizer=regularizers.l2(0.001)))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1024, activation='relu'))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

نورون های این شبکه در سه نوع لایه هستند:

- **لایه ورودی** : با تعداد نورون‌های ۷۸۴، متناسب با تعداد پیکسل‌های تصاویر MNIST است.
- **لایه‌های پنهان** : تعداد نورون‌ها در هر لایه به تدریج کاهش می‌یابد (۲۰۴۸، ۱۰۲۴, ۵۱۲, ۲۵۶, ۱۲۸, ۶۴). این معماری به شکل یک قیف است که به مدل اجازه می‌دهد تا ویژگی‌های سطح بالا را در لایه‌های اولیه استخراج کند و سپس این ویژگی‌ها را به تدریج ترکیب و تصفیه کند تا به تصمیم‌گیری دقیق‌تری در لایه‌های بعدی برسد.
- **لایه خروجی** : دارای ۱۰ نورون است که متناسب با تعداد کلاس‌های دست‌نویس MNIST است و از تابع فعال‌سازی **softmax**استفاده می‌کند تا احتمال تعلق هر تصویر به یکی از این ۱۰ کلاس را محاسبه کند.

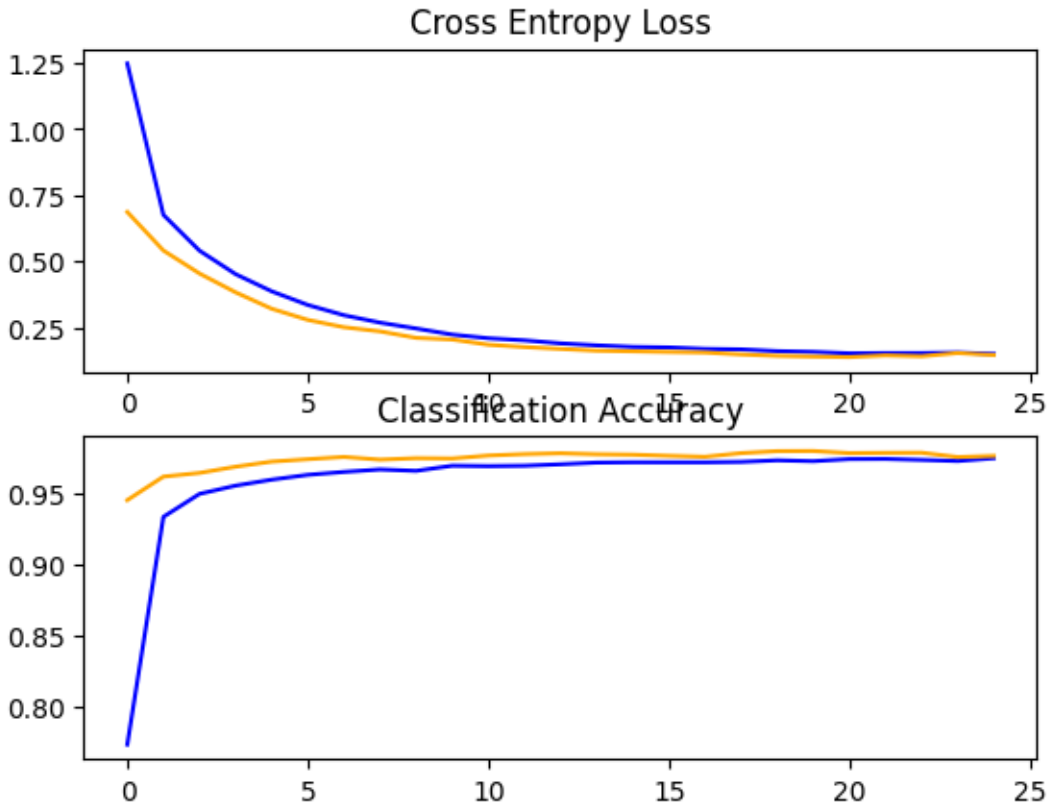
این تعداد نرون ها به مدل اجازه می‌دهد تا ویژگی‌های مختلف تصاویر را در سطوح مختلف آموزش ببیند و به این ترتیب، توانایی خوبی در تشخیص اعداد دست‌نویس برخوردار شود. با این حال، انتخاب تعداد نورون‌ها در هر لایه می‌تواند بر اساس آزمایش و خطا و تنظیمات مختلف بهینه‌سازی شود تا بهترین عملکرد را برای مسئله مورد نظر فراهم آورد.

برای مثال میتوانیم تعداد نرون هارا بصورت زیر تنظیم کنیم و سپس نتیجه را بررسی نماییم:

```
# Define the model
model = Sequential()
model.add(layers.Dense(512, activation='relu', input_dim=784 ,
                       kernel_regularizer=l2(0.001)))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

نتیجه:

```
Accuracy: 97.69%    |   Recall: 97.69%
Precision: 97.72%   |   F1-Score: 97.69%
```



همانطور که مشاهده میکنید دقت مدل به دلیل کاهش تعداد نورون ها کاهش یافته است، در نمودار هم به محض رسیدن دقت داده های آموزش به داده های تست پردازش متوقف شده است، این به این معنیست که اگر روند پردازش ادامه پیدا میکرد احتمالا دقت افزایش می یافت.

Dropout و Regularization

Dropout و Regularization دو تکنیک مهم برای جلوگیری از بیش‌برازش (Overfitting) در شبکه‌های عصبی هستند. بیش‌برازش زمانی رخ می‌دهد که مدل بیش از حد به داده‌های آموزشی خود وابسته شود و نتواند به خوبی بر روی داده‌های جدید تعمیم یابد. در ادامه به توضیح دقیق‌تر این دو تکنیک میپردازیم:

- **Dropout**: این تکنیک شامل تصادفی حذف کردن (یا "خاموش کردن") برخی از نورون‌ها در طول فرآیند آموزش است. این کار باعث می‌شود که شبکه نتواند به شدت به هر نورون خاصی وابسته شود و در نتیجه، مدل قوی‌تر و قابل تعمیم‌تر می‌شود.
- **Regularization**: این تکنیک شامل افزودن یک جمله جریمه به تابع هزینه است که وزن‌های بزرگ را مجازات می‌کند. این امر باعث می‌شود که مدل ساده‌تر شود و کمتر به داده‌های آموزشی وابسته باشد.

همانطور که مشاهده میکنید در این پروژه، در لایه اول شبکه از Dropout(۰.5) و Regularization با ضریب ۰.۰۰۱ استفاده شده است. Dropout(۰.5) به این معنی است که در هر دوره آموزش، ۵۰٪ از نورون‌ها به صورت تصادفی خاموش می‌شوند. همچنین، kernel_regularizer=l2(۰.۰۰۱) به لایه اول اضافه شده است که نشان‌دهنده استفاده از L۲ Regularization با ضریب ۰.۰۰۱ است. این تغییرات به مدل کمک می‌کنند تا عمومی‌تر شود و بر روی داده‌های جدید بهتر عمل کند .

```
# Define the model
model = Sequential()
model.add(layers.Dense(1024, activation='relu', input_dim=784,
                                    kernel_regularizer=regularizers.l2(0.001)))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(128, activation='relu'))
```

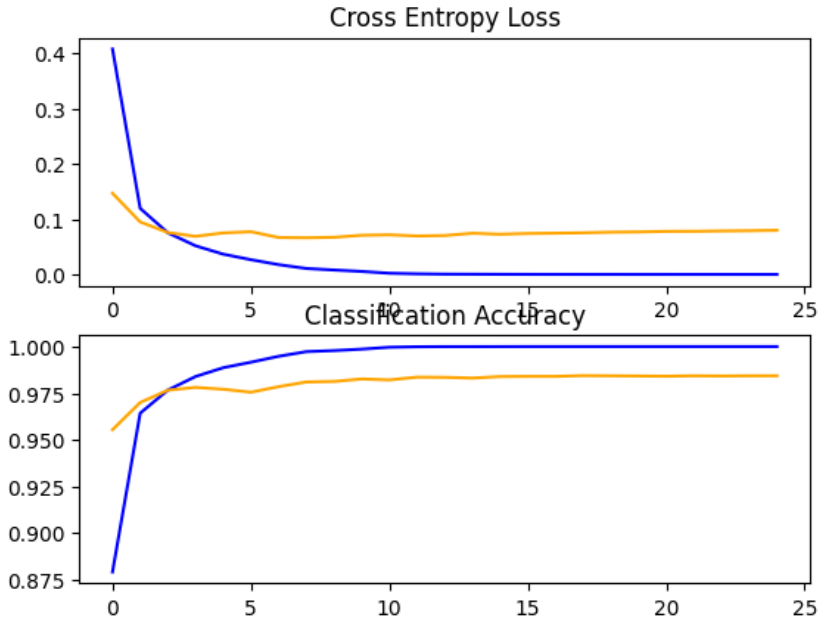
```
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

برای مثال اگر از Dropout و Regularization استفاده نکنیم دقت مدل به طور چشمگیری افزایش میابد اما مدل دچار بیش برازش بر روی داده ها میشود:

```
model = Sequential()
model.add(layers.Dense(2048, activation='relu', input_dim=784))
model.add(layers.Dense(1024, activation='relu'))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

نتیجه:

```
Accuracy: 98.44%   |   Recall: 98.44%
Precision: 98.44%  |   F1-Score: 98.44%
```



بیش برازش مدل روی داده ها بدلیل نبود Dropout و Regularization

الگوریتم بهینه سازی

الگوریتم‌های بهینه‌سازی در یادگیری ماشین برای تنظیم پارامترهای مدل به منظور کاهش تابع هزینه استفاده می‌شوند. این الگوریتم‌ها به مدل کمک می‌کنند تا از داده‌های آموزشی برای پیش‌بینی دقیق‌تر استفاده کند. انواع مختلفی از این الگوریتم ها با عملکردهای متفاوت وجود دارد و بر اساس مشخصات مسئله و محدودیت‌های محاسباتی انتخاب می‌شوند. هر کدام مزایا و معایب خاص خود را دارند و ممکن است در شرایط مختلف بهتر عمل کنند. انتخاب الگوریتم بهینه‌سازی مناسب می‌تواند تأثیر زیادی بر سرعت و کیفیت یادگیری مدل داشته باشد. در ادامه چهار مدل از این الگئوریتم هارا پیاده سازی و نتایج آنهارا مقایسه میکنیم:

الگوریتم SGD

Stochastic Gradient Descent (SGD): نسخه‌ای تصادفی از Gradient Descent است که در آن گرادیان تابع هزینه بر اساس تنها یک نمونه یا یک دسته کوچک از نمونه‌ها (mini-batch) محاسبه می‌شود. این کار باعث می‌شود که به‌روزرسانی‌ها سریع‌تر و کم‌هزینه‌تر باشند . .الگوریتم بهینه ساز استفاده شده در پیاده سازی مدل همین الگوریت است و نحوه پیاده سازی و نتایج آنرا قبلا مشاهده نمودید.

الگوریتم Adagrad

این الگوریتم نرخ یادگیری را برای هر پارامتر به صورت انطباقی تنظیم می‌کند، که به ویژه برای داده‌هایی با توزیع نامتوازن مفید است. در ادامه عملکرد این الگوریتم را در این مساله بررسی میکنیم:

```
from keras.optimizers import Adagrad

# Define the model
model = Define_model()
```

```
opt = optimizers.Adagrad(learning_rate=0.01)

# Compile the model
model.compile(optimizer=opt, loss='categorical_crossentropy',
metrics=['accuracy'])
```

```
Accuracy: 98.01%      |    Precision: 98.02%
Recall : 98.01%       |    F1-Score: 98.01%
```

با توجه به تغییرات دقت مدل درمیابیم این الگوریتم در کل عمکرد خوبی داشته اما همچنان نسبت به الگوریتم SGD ضعیف تر عملکرده است.

الگوریتم RMSprop

شامل یک تغییر در Adagrad است که مشکل کاهش شدید نرخ یادگیری را حل می‌کند. این الگوریتم با استفاده از میانگین مربعات گرادیان‌های اخیر برای تنظیم نرخ یادگیری کار می‌کند.

```
# Define the model
model = Model_Define()

#optimizers
opt = optimizers.RMSprop(learning_rate=0.001)

# Compile the model
model.compile(optimizer=opt, loss='categorical_crossentropy',
metrics=['accuracy'])
```

نتیجه:

```
Accuracy: 97.32%      |    Recall: 97.32%
Precision: 97.34%     |    F1-Score: 97.32%
```

الگوریتم Adam

Adam (Adaptive Moment Estimation) ترکیبی از ایده‌های Momentum و RMSprop است. این الگوریتم نه تنها میانگین مربعات گرادیان‌ها را حساب می‌کند بلکه میانگین متحرک گرادیان‌ها را نیز محاسبه می‌کند، که به آن اجازه می‌دهد تا نرخ یادگیری را برای هر پارامتر به صورت تطبیقی تنظیم کند. پیاده سازی این الگوریتم بصورت زیر است:

```
from keras.optimizers import Adam

# Define the model
model = Define_model()

opt = Adam()

# Compile the model
model.compile(optimizer=opt, loss='categorical_crossentropy',
metrics=['accuracy'])
```

```
Accuracy: 97.54%      |    Precision: 97.55%
Recall : 97.54%       |    F1-Score: 97.54%
```

همانطور که مشاهده میکنید دقت مدل با استفاده از این الگوریتم از همه ی حالات قبلی کمتر شد، دلیل این اتفاق میتواند نامناسب بود ابرپارامترهای پیشفرض این الگوریتم برای این مساله باشد. همچنین آدام گرادیان‌ها را مقیاس‌پذیر می‌کند که می‌تواند در داده‌هایی با ویژگی‌های نامتوازن مفید باشد، اما در برخی موارد ممکن است به کاهش دقت منجر شود.

نرخ یادگیری

نرخ یادگیری در شبکه‌های عصبی یک پارامتر کلیدی است که اندازه‌ی گام‌های به‌روزرسانی وزن‌ها در فرآیند یادگیری را تعیین می‌کند .این پارامتر به صورت یک مقدار اسکالر است که با گرادیان تابع زیان ضرب می‌شود تا میزان تغییر وزن‌ها را در هر تکرار مشخص کند.

برای تنظیم نرخ یادگیری مناسب، معمولاً باید بین سرعت همگرایی و خطر پرتاب شدن تعادل برقرار کرد. برخی از روش‌های پیشرفته مانند نرخ یادگیری تطبیقی، نرخ یادگیری را در طول زمان تغییر می‌دهند تا بهینه‌سازی را بهبود ببخشند و از گیر افتادن در حداقل‌های محلی جلوگیری کنند.

در ادامه ما دو نرخ یادگیری متفاوت را بر روی مدل امتحان میکنیم و نتایج را مقایسه میکنیم:

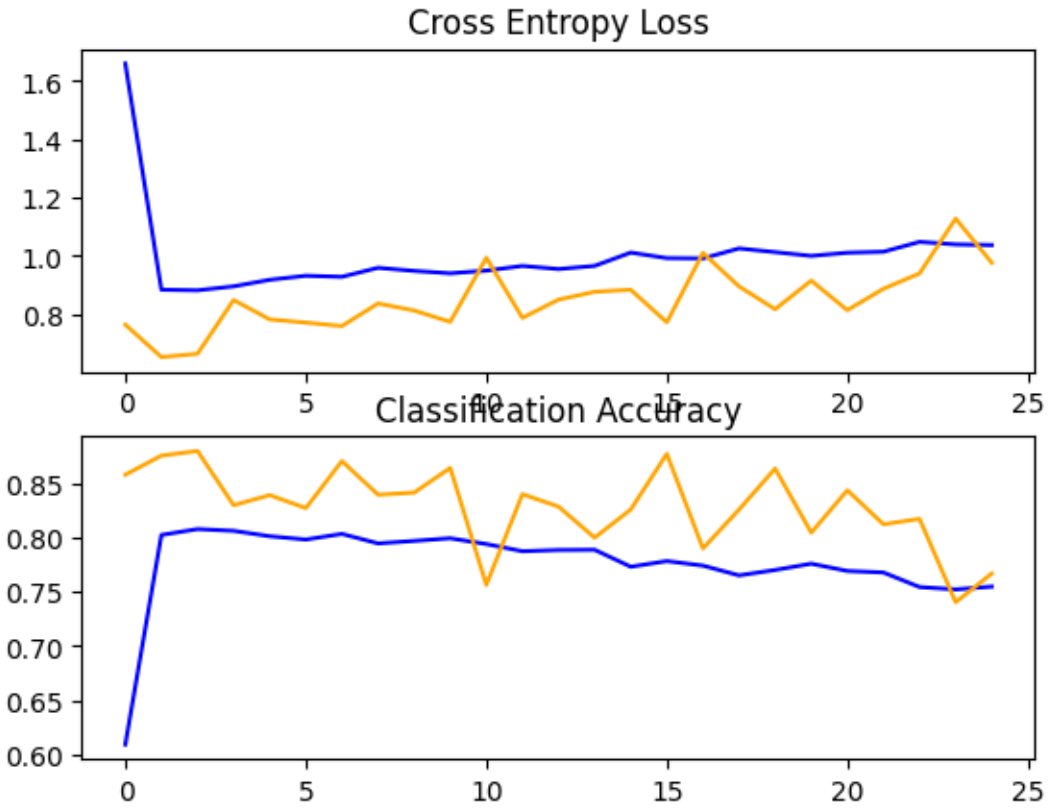
نرخ یادگیری 0.01:

تابحال از الگوریتم بهینه سازی SGD با نرخ یادگیری 0.001 استفاده میکردیم و نتایج آن را نیز در بخش ارزیابی مدل مشاهده کردیم، اکنون بیاید نرخ یادگیری را ده برابر کنیم و به مقدار 0.01 افزایش دهیم، بنظر شما چه تاثیری در دقت مدل خواهد داشت؟

```
# Define the model
model = Model_Define()

#optimizers
opt = optimizers.SGD(learning_rate = 0.01, momentum = 0.9)

# Compile the model
model.compile(optimizer=opt, loss='categorical_crossentropy',
                              metrics=[‘accuracy’])
```

نتیجه:

```
Accuracy: 76.68%      |     Precision: 81.97%
Recall  : 76.68%      |     F1-Score: 76.09%
```



شاید شما هم انتظار داشتید دقت مدل تنها مقدار کاهش یابد، اما همانطور که مشاهده میکند بیش از ۲۰ درصد دقت مدل کم شده است، و همچنین نمودارهای دقت و خطای مدل بر روی داده های تست بشدت نوسانی و ناموزون شده است. این میتواند به دلیل پرش تابع زیان از روی مقادیر کمینه و عبور از آنها بدلیل گام های بزرگ بروزرسانی وزن ها که منجر به عدم همگرایی مدل و نوسانات شدید در معیارهای ارزیابی شده است.

نرخ یادگیری 0.005:

اکنون میخواهیم نرخ یادگیری را ۵ برابر کرده و به مقدار 0.005افزایش دهیم.

```
# Define the model
model = Model_Define()
#optimizers
opt = optimizers.SGD(learning_rate = 0.005, momentum = 0.9)
# Compile the model
model.compile(optimizer=opt, loss='categorical_crossentropy',
                              metrics=['accuracy'])
```

نتیجه:

```
Accuracy: 93.93%      |       Precision: 94.02%
Recall : 93.93%       |       F1-Score: 93.93%
```

همانطور که مشاهده میکنید بازهم دقت مدل کاهش یافته است و پرش تابع زیان از روی مقادیر کمینه اتفاق افتاده است. اما همچنان نسبت به حالت قبل بصورت مقدار معقولانه‌تری داده از دست داده ایم. پس بطورکلی نتیجه میگیریم شاید نرخ یادگیری بزرگتر سرعت یادگیری خوبی به ما ارائه کند و همچنین ریسک افتادن در کمینه های محلی را کاهش دهد اما خطر پرتاب شدن تعادل مدل را بوجود میاورد و ممکن است دقت مدل را به شدت کاهش دهد.

بیش‌برازش و کم‌برازش

بیش‌برازش یا Overfitting زمانی رخ می‌دهد که یک مدل شبکه عصبی بیش از حد بر روی داده‌های آموزشی تنظیم شده و نتواند به خوبی عمومیت پیدا کند. این مدل‌ها ممکن است دقت بالایی روی داده‌های آموزشی داشته باشند، اما روی داده‌های جدید یا تست عملکرد ضعیفی نشان دهند.

کم‌برازش یا Underfitting زمانی اتفاق می‌افتد که مدل نتواند الگوهای موجود در داده‌های آموزشی را به خوبی یاد بگیرد. این مدل‌ها نه تنها روی داده‌های آموزشی بلکه روی داده‌های تست نیز عملکرد ضعیفی دارند.

پارامترهایی که ممکن است موجب overfitting یا underfitting شوند عبارتند از:

- **تعداد لایه‌ها و نورون‌ها** در شبکه
- **نرخ یادگیری**
- **تعداد تکرارها** (epochs)
- **اندازه دسته‌های آموزشی** (batch size)
- **تنظیمات مقداردهی اولیه وزن‌ها** (weight initialization)
- **تکنیک‌های** Regularization مانند **L1/L2** regularization و Dropout
- **تعداد ویژگی‌ها** (features) در داده‌های آموزشی
- **کیفیت و تنوع داده‌های آموزشی**

ایجاد Underfitting

در ادامه میخواهیم یک حالت کم‌برازش یا Underfitting ایجاد کنیم و نتیجه آنرا بررسی کنیم. مدل را بصورت زیر پیاده سازی میکنیم:

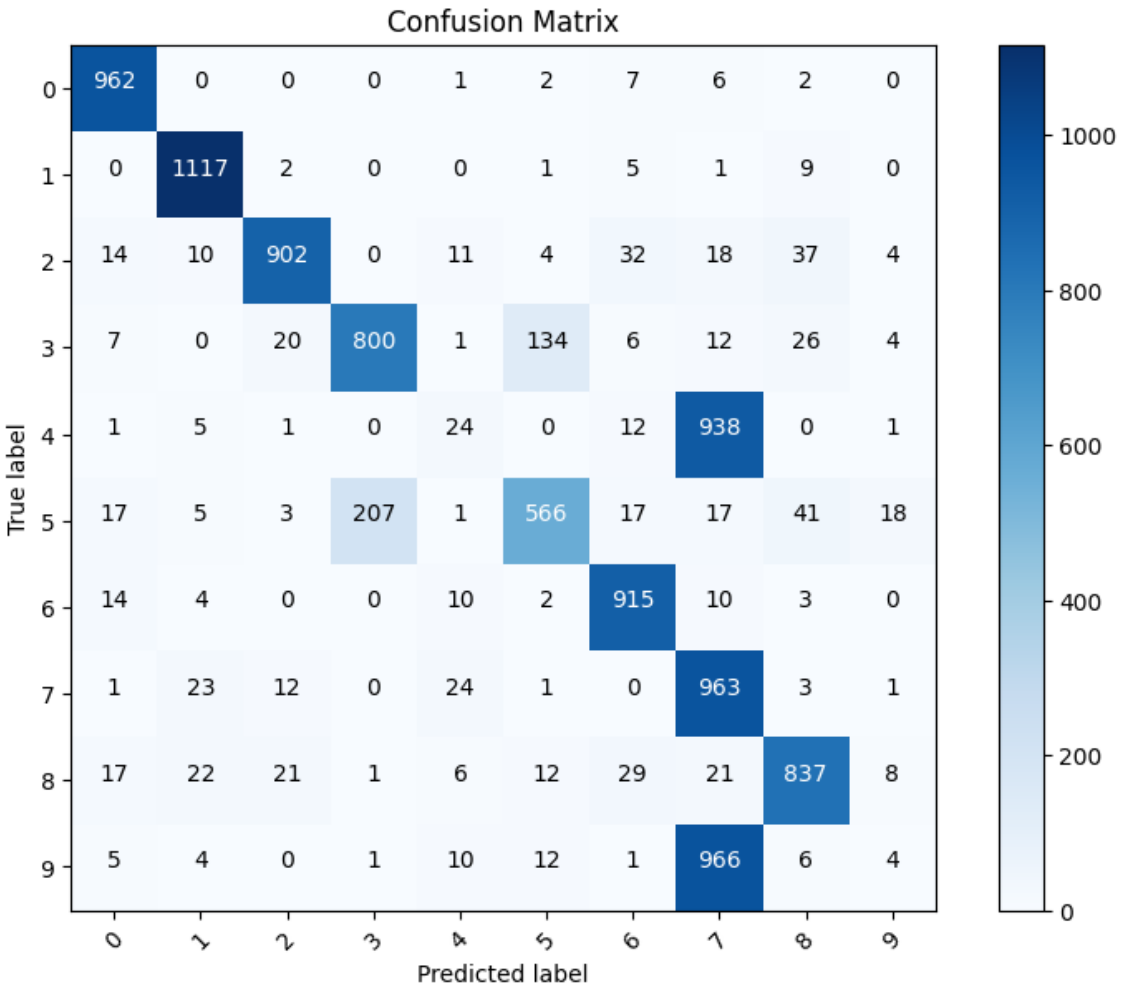
```
# Define the model
model = Sequential()

model.add(Dense(512, activation='relu', input_dim=784 ,
kernel_regularizer=l2(0.001)))
model.add(Dropout(0.5))
model.add(layers.Dense(256, activation='relu',
kernel_regularizer=l2(0.001)))
model.add(Dropout(0.5))
model.add(layers.Dense(128, activation='relu',
kernel_regularizer=l2(0.001)))
model.add(Dropout(0.5))
model.add(layers.Dense(64, activation='relu',
kernel_regularizer=l2(0.001)))
model.add(Dropout(0.5))
model.add(layers.Dense(32, activation='relu',
kernel_regularizer=l2(0.001)))
model.add(Dropout(0.5))
model.add(layers.Dense(16, activation='relu',
kernel_regularizer=l2(0.001)))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))


opt = optimizers.SGD(learning_rate = 0.001, momentum = 0.9)

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

# Train the model
history = model.fit(train_images, train_labels,
validation_data=(test_images, test_labels), epochs=5, batch_size=256)

# Make predictions
predictions_probabilities = model.predict(test_images)
predictions = np.argmax(predictions_probabilities, axis=1)
```

همانطور که مشاهده میکنید تعداد لایه های مدل را یک عدد کمتر، نرون های هر لایه را نصف، مقدار نرخ یادگیری را ۵ برابر، تعداد تکرارها (epochs) را ۵ و اندازه دسته‌های آموزشی (batch size) را ۲۵۶ مقداردهی کرده ایم و نتیجه بصورت زیر است

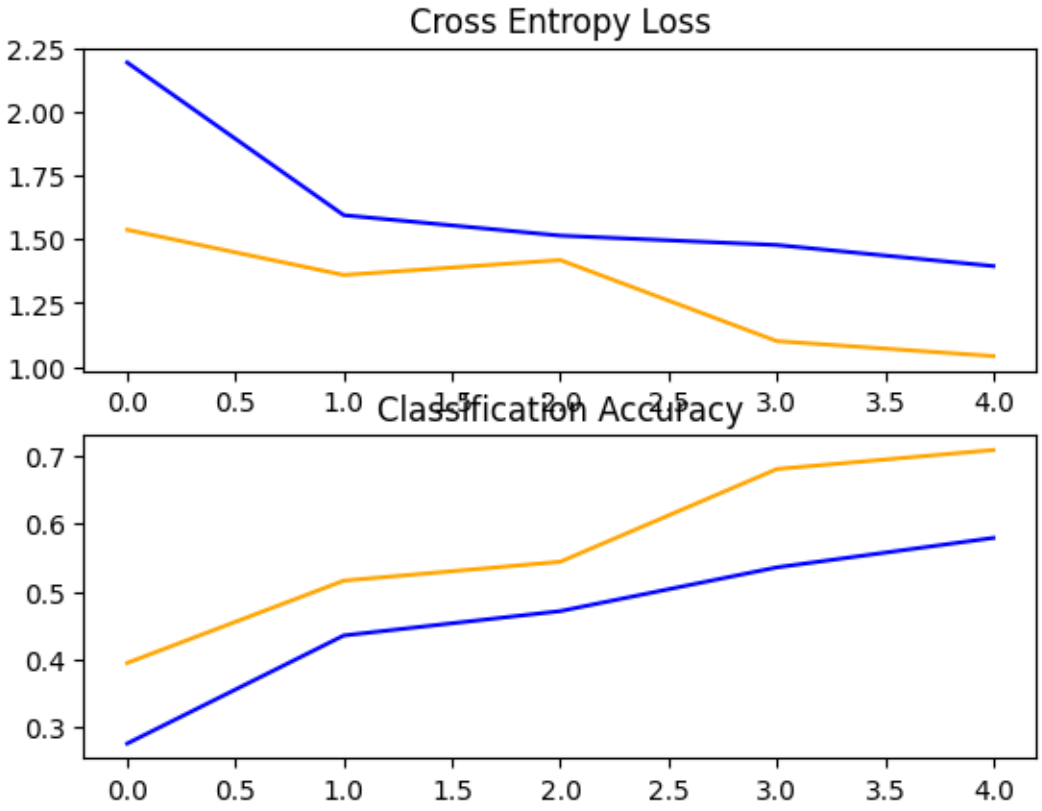
معیارهای مدل:

```
Accuracy: 70.90%        |     Precision: 68.37%
Recall  : 70.90%        |     F1-Score: 66.56%
```

ماتریس درهم‌ریختگی (Confusion Matrix):



نمودارهای فراگیری:



همانطور که مشاهده میکنید شرایط مدل نمونه بارز یک کم برازش را نشان میدهد. دقت مدل بسیار کاهش یافته، اختلاف منحنی های داده های آموزش و تست بسیار زیاد است، و ماتریس درهم ریختگی نیز از حالت قطری خارج شده و پیش بینی مدل در کلاس ارقام بسیار پرخطا شده است.

ایجاد Overfitting

اکنون میخواهیم یک حالت بیش‌برازش یا Overfitting ایجاد کنیم و نتیجه آنرا بررسی کنیم.

مدل را بصورت زیر پیاده سازی میکنیم:

```
# Define the model
model = Sequential()
model.add(Dense(2048, activation='relu', input_dim=784))
model.add(layers.Dense(2048, activation='relu'))
model.add(layers.Dense(1024, activation='relu'))
model.add(layers.Dense(1024, activation='relu'))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(64, activation='relu'))
model.add(Dense(10, activation='softmax'))


opt = optimizers.SGD(learning_rate = 0.001, momentum = 0.9)


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

# Train the model
history = model.fit(train_images, train_labels,
validation_data=(test_images, test_labels), epochs=35, batch_size=64)

# Make predictions
predictions_probabilities = model.predict(test_images)
predictions = np.argmax(predictions_probabilities, axis=1)
```

در کد بالا سعی کردیم مدل را تا حد ممکن پیچیده کنیم تا به اورفیتینگ برسیم، بطوری که ۸ لایه با تعداد نرون های بالا، بدون هیچ دراپ‌اوت و رگولایزری درست کردیم، همچنین تعداد تکرارها (epochs) را ۳۵ و اندازه دسته‌های آموزشی (batch size) را ۶۴ مقداردهی کرده ایم.

در ادامه نتیجه این اعمال را مشاهده و تحلیل میکنیم.

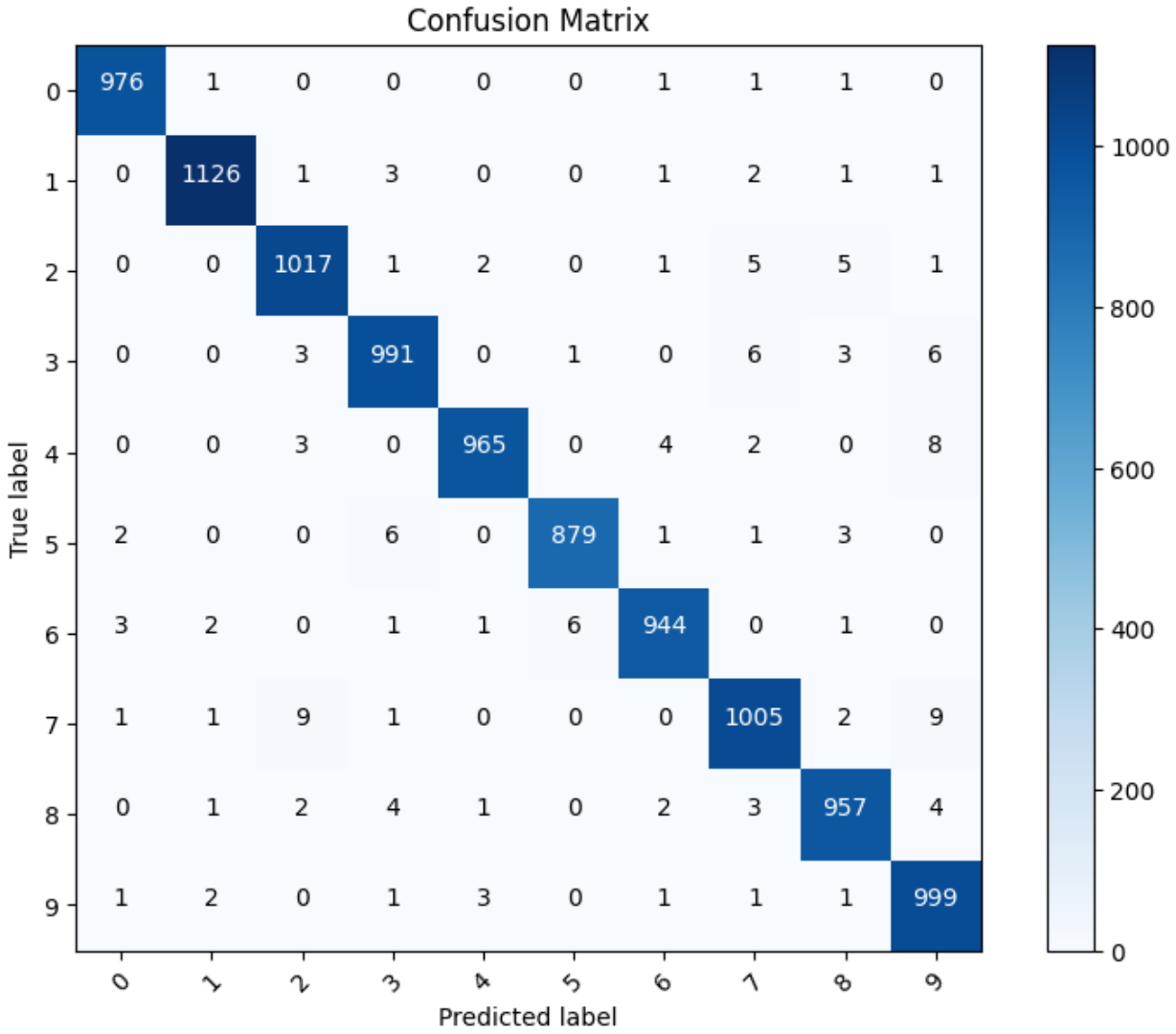
دقت مدل روی داده‌های آموزشی:

```
loss: 0.0050 - accuracy: 0.9994
```

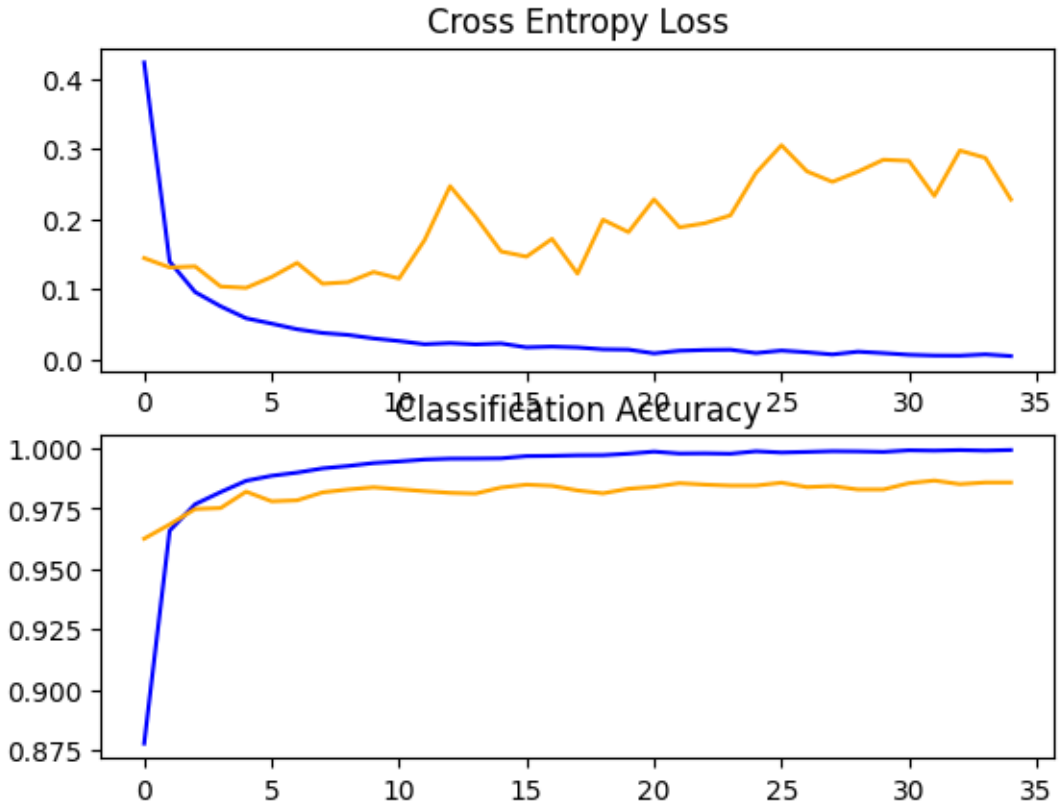
معیارهای مدل:

```
Accuracy: 98.59%      |      Precision: 98.59%
Recall : 98.59%       |      F1-Score: 98.59%
```

ماتریس درهم‌ریختگی (Confusion Matrix):



نمودارهای فراگیری:



همانطور که مشاهده میکنید خروجی مدل رخ داد بیش‌برازش را به وضوح نمایش میدهد. دقت مدل روی داده های آموزش بسیار افزایش یافته است اما روی داده های تست کمتر است، همچنین اختلاف منحنی های داده های آموزش و تست بسیار زیاد است و با تمام شدن هر دور دقت مدل در داده های آموزش بیشتر شده ولی روی داده های تست کمتر و کمتر میشود، همچنین خطای مدل روی داده های آموزش رو به کم شدن است اما روی داده های تست افزایش شدید داشته است.

شرایط توقف

شرایط توقف یا (Early Stopping) یک تکنیک مدیریتی است که برای جلوگیری از overfittingدر حین آموزش مدل‌های یادگیری ماشین استفاده می‌شود. این روش به مدل اجازه می‌دهد تا در زمانی که عملکرد روی داده‌های اعتبارسنجی (validation) دیگر بهبود نمی‌یابد، آموزش را متوقف کند. به عبارت دیگر، اگر دقت مدل روی داده‌های اعتبارسنجی برای تعداد مشخصی از epochs افزایش نیابد یا کاهش یابد، آموزش متوقف می‌شود و وزن‌هایی که بهترین عملکرد را داشته‌اند، ذخیره می‌شوند.

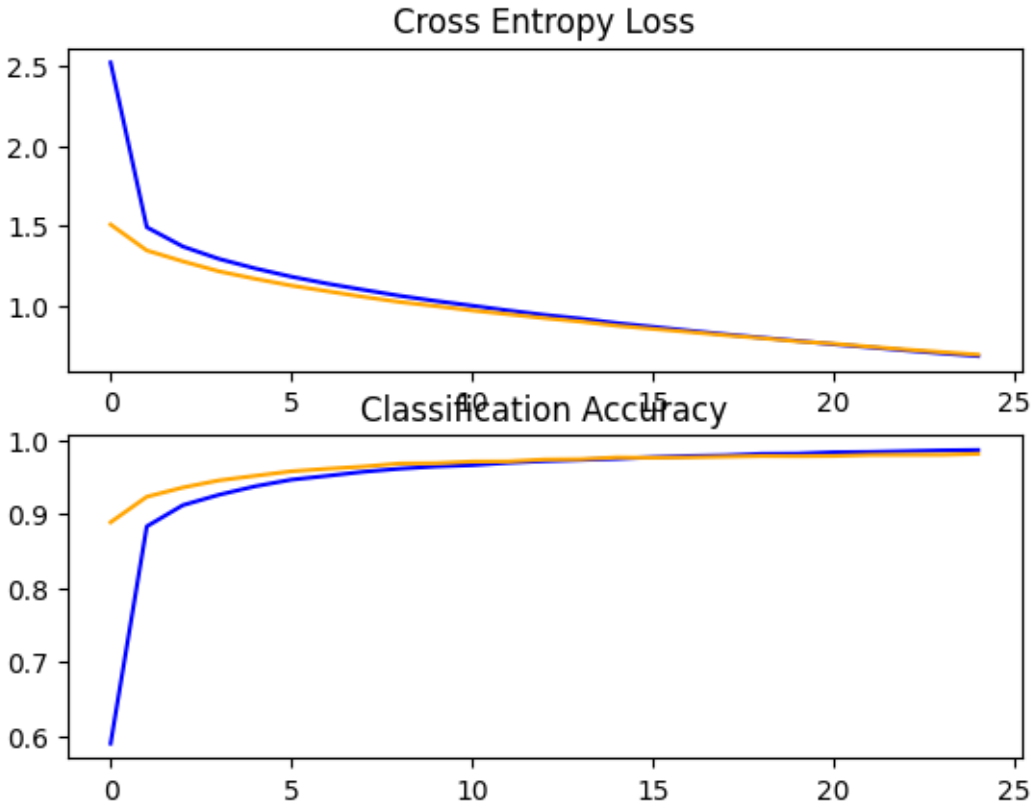
برای پیاده سازی این تکنیک بصورت زیر عمل میکنیم:

```
from tensorflow.keras.callbacks import EarlyStopping

# Define the model
model = Model_Define()
opt = optimizers.SGD(learning_rate = 0.001, momentum = 0.9)
# Compile the model
model.compile(optimizer=opt, loss='categorical_crossentropy',
metrics=['accuracy'])

# early stop conditions
early_stopping = EarlyStopping(monitor='val_loss', patience=2)


# Train the model
history = model.fit(train_images, train_labels,
                    validation_data=(test_images, test_labels), epochs=25,
                    batch_size=100,  callbacks=[early_stopping])
```

در این کد، EarlyStopping به گونه‌ای تنظیم شده است که اگر val_loss(خطای اعتبارسنجی) برای دو epoch متوالی بهبود نیابد، آموزش متوقف شود. پس از پایان آموزش، عملکرد مدل را روی داده‌های تست ارزیابی میکنیم:

```
Epoch 25/25

600/600 [==============================] - 56s 93ms/step
- loss: 0.6846 - accuracy: 0.9863
- val_loss: 0.6936 - val_accuracy: 0.9812
313/313 [==============================] - 3s 10ms/step
313/313 [==============================] - 4s 14ms/step
- loss: 0.6936 - accuracy: 0.9812
```

Accuracy: 98.12% | Precision: 98.13%
Recall : 98.12% | F1-Score: 98.12%



مشاهدات نشان میدهد شرایط توقفی که پیاده سازی کردیم اجرا نشده است و تابع توقف زودهنگام تاثیری در عملکرد مدل نداشته است زیرا تمام 25 دور آموزش اجرا شده است. پس نتیجه میگیریم تعداد دور ها و اندازه دسته های آموزشی مقدار مناسبی تنظیم شده و خطای اعتبارسنجی همواره در حال بهبود یافتن بوده است، زیرا در غیر این صورت تابع توقف زودهنگام فعال می‌شد.

تابع فعالسازی

تابع فعال‌سازی (Activation Function) یکی از اجزای اصلی در شبکه‌های عصبی است که نقش مهمی در تعیین خروجی نورون‌ها دارد. این توابع به شبکه کمک می‌کنند تا الگوهای پیچیده‌تر و غیرخطی را یاد بگیرند و از این طریق، قابلیت تعمیم به داده‌های جدید را افزایش می‌دهند.

در اینجا به توضیح چند نوع پرکاربرد از توابع فعال‌سازی و سپس به پیاده سازی آنها و ارزیابی نتایج می‌پردازیم:

تابع واحد یک‌سوشده‌ی خطی (ReLU / Rectified Linear Unit):

این تابع برای مقادیر مثبت خطی است و برای مقادیر منفی صفر را برمی‌گرداند. ReLU به دلیل سادگی و کارایی بالا در شبکه‌های عمیق بسیار محبوب است. تابع مورد استفاده در این مساله همین تابع است که نحوه پیاده سازی و نتایج آنرا قبلا بررسی نموده ایم.

```
Accuracy: 98.18%    |  Recall: 98.18%
Precision: 98.18%   |  F1-Score: 98.18%
```

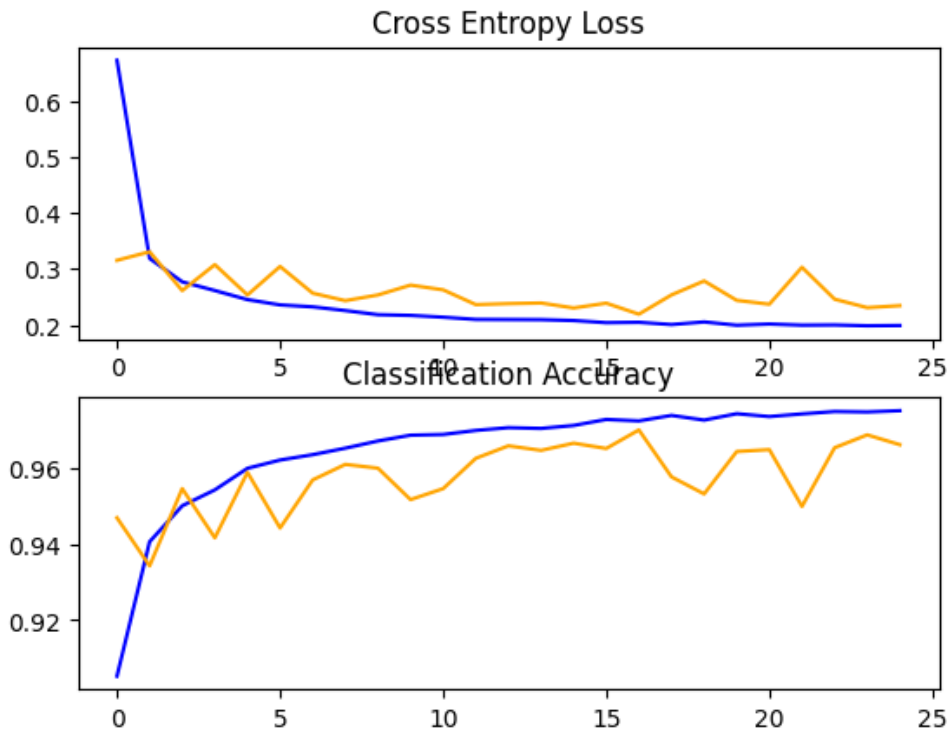
تابع سیگموید (Sigmoid):

این تابع مقادیر ورودی را به بازه (۰,۱) محدود می‌کند و برای مسائل طبقه‌بندی دو کلاسه مناسب است. با این حال، ممکن است با مشکل گرادیان‌های ناپدید شونده مواجه شود. برای استفاده از این تابع لازم است از تکنیک نرمال سازی دسته ای نیز استفاده کنیم زیرا افزودن لایه های نرمال سازی دسته ای می تواند به کاهش مشکل تغییر توزیع های ورودی کمک کند و همچنین مشکل ناپدید شدن گرادیان را حل کند. و اگر بدون استفاده از لایه های نرمال سازی دسته ای این تابع را پیاده سازی کنیم دقت مدل بشده کاهش میابد.

```
# Define the model
model = Sequential()
model.add(Dense(2048, activation='sigmoid', input_dim=784 ,
                                        kernel_regularizer=l2(0.001)))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(layers.Dense(1024, activation='sigmoid'))
model.add(BatchNormalization())
model.add(layers.Dense(512, activation='sigmoid'))
model.add(BatchNormalization())
model.add(layers.Dense(256, activation='sigmoid'))
model.add(BatchNormalization())

model.add(layers.Dense(128, activation='sigmoid'))
model.add(BatchNormalization())
model.add(layers.Dense(64, activation='sigmoid'))
model.add(BatchNormalization())

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

نتایج مدل:

Accuracy: 96.63% | Precision: 96.64%
Recall : 96.63% | F1-Score: 96.63%



مشاهدات نشان میدهد تابع فعالسازی در این مساله عملکرد خوبی نداشته است و باعث کاهش دقت مدل و تا حدودی کم برازش شده است. دلیل این اتفاق میتواند مربوط به ایجاد گرادیان های بسیار کوچک در طول اجرای الگوریتم پس انتشار خطا باشد.

تابع تانژانت هایپربولیک (Tanh):

این تابع مقادیر ورودی را به بازه (-۱,۱) محدود می‌کند و به دلیل مرکزیت در صفر، اغلب بهتر از سیگموید عمل می‌کند.

```
# Define the model
model = Sequential()
model.add(Dense(2048, activation='tanh', input_dim=784 ,
kernel_regularizer=l2(0.001)))
```

```
model.add(Dropout(0.5))
model.add(layers.Dense(1024, activation='tanh'))
model.add(layers.Dense(512, activation='tanh'))
model.add(layers.Dense(256, activation='tanh'))
model.add(layers.Dense(128, activation='tanh'))
model.add(layers.Dense(64, activation='tanh'))
model.add(Dense(10, activation='softmax'))
```

نتایج:

Accuracy: 95.64% | Precision: 95.70%
Recall : 95.64% | F1-Score: 95.65%

با توجه به کاهش دقت مدل نتیجه میگیریم تابع فعال‌سازی ReLU در مساله MNIST ، به دلیل ویژگی‌های خاص خود، بهتر عمل می‌کند. زیرا این تابع به صورت غیرخطی عمل می‌کند و برای مقادیر مثبت، خروجی را به همان مقدار ورودی تنظیم می‌کند. این ویژگی باعث می‌شود که تابع ReLU به خوبی با مسائلی که الگوهای پیچیده‌تری دارند، سازگار باشد.

نرمال‌سازی دسته‌ای

Batch Normalization یا نرمال‌سازی دسته‌ای، در آموزش شبکه‌های عصبی عمیق، به استانداردسازی ورودی‌های هر لایه برای هر دسته از داده‌ها مربوط می‌شود. این فرآیند به ثبات بخشیدن به روند یادگیری کمک کرده و تعداد دوره‌های آموزشی (epochs) لازم برای آموزش شبکه‌های عمیق را به طور قابل توجهی کاهش می‌دهد.

از جمله مزایای اصلی Batch Normalization می‌توان به موارد زیر اشاره کرد:

- افزایش سرعت آموزش: با کاهش تعداد epochs مورد نیاز برای همگرایی.
- بهبود دقت: با کمک به کاهش مشکلات مربوط به گرادیان‌های ناپدید شونده یا منفجر شونده.
- اثر معادل‌سازی: با کاهش حساسیت مدل به مقداردهی اولیه وزن‌ها.
- تسهیل در استفاده از نرخ‌های یادگیری بالاتر: بدون خطر پرش از حداقل‌های جهانی تابع هزینه.
- کاهش اثر انتقال داخلی: با کمک به کاهش تغییرات توزیع ورودی‌ها در طول آموزش.

با این حال، دلایل دقیق اثربخشی Batch Normalization هنوز مورد بحث است. برخی محققان معتقدند که این تکنیک باعث کاهش انتقال داخلی می‌شود، در حالی که برخی دیگر می‌گویند که این تکنیک باعث صاف شدن تابع هدف می‌شود که در نتیجه عملکرد را بهبود می‌بخشد. همچنین، برخی دیگر بیان می‌کنند که Batch Normalization به جداسازی طول و جهت در شبکه‌های عصبی کمک کرده و در نتیجه سرعت آن‌ها را افزایش می‌دهد.

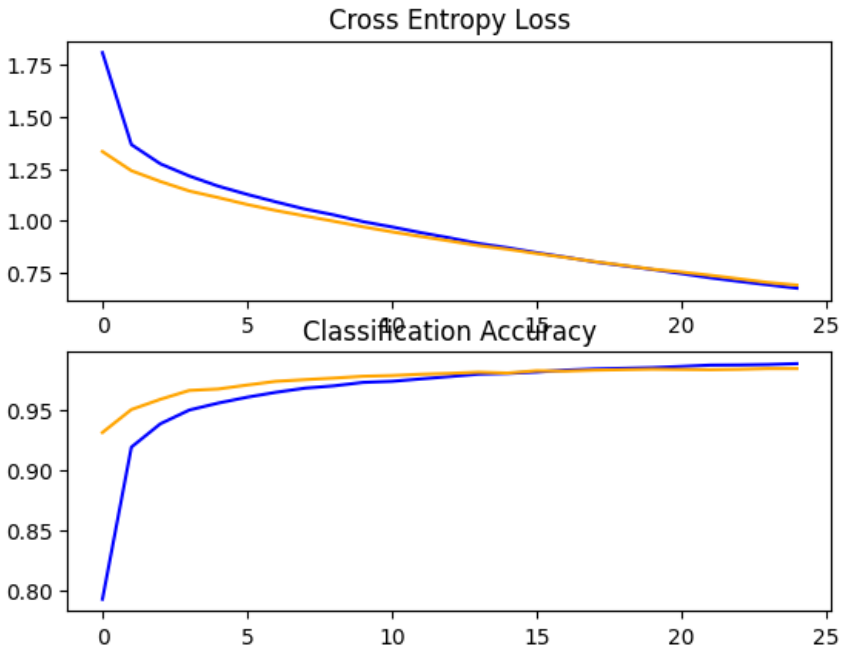
برای درک بهتر تأثیر Batch Normalization در ادامه آنرا در مدل خود پیاده سازی کرده و نتایج را با معیارهای قبلی مقایسه میکنیم:

```
# Define the model
model = Sequential()
model.add(layers.Dense(2048, activation='relu', input_dim=784 ,
kernel_regularizer=regularizers.l2(0.001)))
model.add(layers.Dropout(0.5))
model.add(layers.BatchNormalization())

model.add(layers.Dense(1024, activation='relu'))
model.add(layers.BatchNormalization())

model.add(layers.Dense(512, activation='relu'))
model.add(layers.BatchNormalization())

model.add(layers.Dense(256, activation='relu'))
model.add(layers.BatchNormalization())

model.add(layers.Dense(128, activation='relu'))
model.add(layers.BatchNormalization())

model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

معیارهای ارزیابی خواهد بود:

Accuracy: 98.44% | Precision: 98.44%
Recall : 98.44% | F1-Score: 98.44%

نمودار فراگیری:



همانطور که مشاهده میکنید دقت مدل بیشتر شد! اما ذره ای نیز بیش‌برازش در نمودار اتفاق افتاده است، اما با توجه به ناچیز بودن این موضوع میتوانیم نتیجه بگیریم نرمال سازی دسته ای داده ها در هر مرحله از آموزش میتواند مفید واقع شود پس این تکنیک را در پروژه استفاده میکنیم.