پروژه اول مبانی هوش محاسباتی

استاد: دکتر تابع الحجه

ایلیا شعبان پور فولادی

4003613041

لینک گیت‌هاب:

https://github.com/ilya-shabanpour/MNIST_MLP_Keras

نیم سال دوم ۱۴۰۳ - ۱۴۰۲

فهرست مطالب

مقدمه

در این مستند به شرح پروژه اول درس مبانی هوش محاسباتی می‌پردازم.

هدف از این پروژه کلاس بندی داده‌های دیتاست MNIST با استفاده از مدل MLP می‌باشد.

در چند مرحله این مدل را پیاده سازی میکنیم.

در پیاده سازی از کتابخانه ها و فریم ورک های زیر استفاده شده:

- Numpy
- Pandas
- Keras
- MatPlotLib

ایمپورت کردن کتابخانه های لازم:

```
import numpy as np

import pandas as pd


import matplotlib.pyplot as plt
%matplotlib inline


from keras.models import Sequential
from keras.optimizers import Adam ,RMSprop
from keras import  backend as K
from keras.layers import Dense, Activation, Dropout, BatchNormalization
from keras.utils import to_categorical, plot_model
from keras.callbacks import EarlyStopping
```

در ابتدا کتابخانه های لازم را ایمپورت میکنیم. از نامپای برای کار با تنسورها، از پانداز برای کار با دیتاست، از MatPlotLib برای مصورسازی داده‌ها و از keras برای پیاده سازی شبکه عصبی (MLP) استفاده میکنیم.

آماده سازی داده‌ها:

MNIST dataset

```
# import dataset
from keras.datasets import mnist

# load dataset
(x_train, y_train),(x_test, y_test) = mnist.load_data()

# count the number of unique train labels
unique, counts = np.unique(y_train, return_counts=True)
print("Train labels: ", dict(zip(unique, counts)))

# count the number of unique test labels
unique, counts = np.unique(y_test, return_counts=True)
print("\nTest labels: ", dict(zip(unique, counts)))
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 2s 0us/step
Train labels:  {0: 5923, 1: 6742, 2: 5958, 3: 6131, 4: 5842, 5: 5421, 6: 5918, 7: 6265, 8: 5851, 9: 5949}

Test labels:  {0: 980, 1: 1135, 2: 1032, 3: 1010, 4: 982, 5: 892, 6: 958, 7: 1028, 8: 974, 9: 1009}
```

دیتاست MNIST در مجموعه دیتاست های keras وجود دارد پس آن را import میکنیم.

سپس دیتاست را لود کرده و مجموعه داده های زیر را ایجاد میکنیم:

X_train: مجموعه داده های ورودی برای آموزش مدل است.

Y_train: مجموعه برچسب های داده های ورودی برای آموزش مدل است.

X_test: مجموعه داده های ورودی برای آزمایش مدل است.

Y_test: مجموعه برچسب های داده های آموزش است.

در انتها تعداد داده های هر کلاس در مجموعه های test و train نمایش داده شده اند.

مصور سازی داده‌های MNIST:

Data visualization

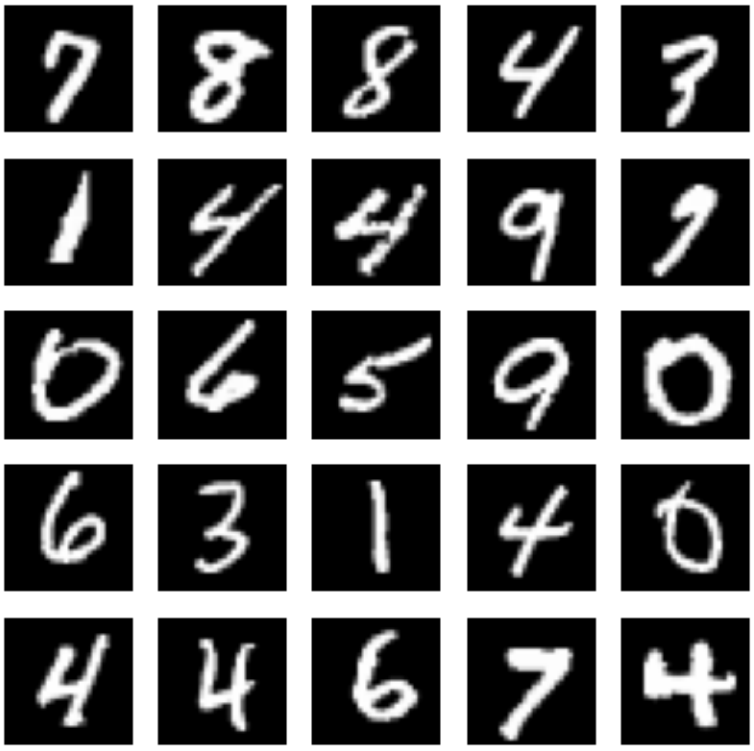
```
# sample 25 mnist digits from train dataset
indexes = np.random.randint(0, x_train.shape[0], size=25)
images = x_train[indexes]
labels = y_train[indexes]


# plot the 25 mnist digits
plt.figure(figsize=(5,5))
for i in range(len(indexes)):
    plt.subplot(5, 5, i + 1)
    image = images[i]
    plt.imshow(image, cmap='gray')
    plt.axis('off')

plt.show()
plt.savefig("mnist-samples.png")
plt.close('all')
```

سپس برای کسب اطلاعات بیشتر از دیتاست ۲۵ عدد از عکس های این دیتاست را چاپ میکنیم.

همانگونه که مشاهده میشود این دیتاست شامل ۷۰ هزار عکس از اعداد دست نویس میباشد که به صورت ماتریس های دو بعدی ۲۸ در ۲۸ ذخیره شده اند:

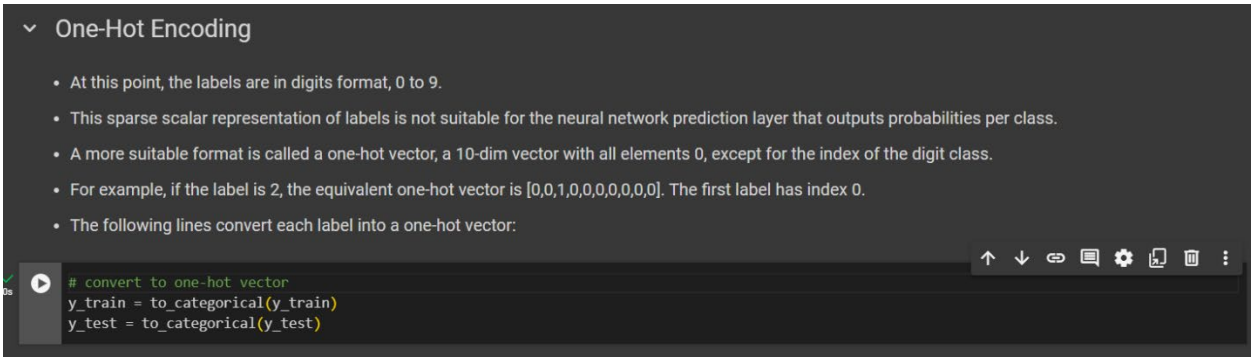


در درایه های این ماتریس اعداد ۰ تا ۲۵۵ قرار دارد که نشان دهنده سیاه و سفیدی تصویر میباشد.

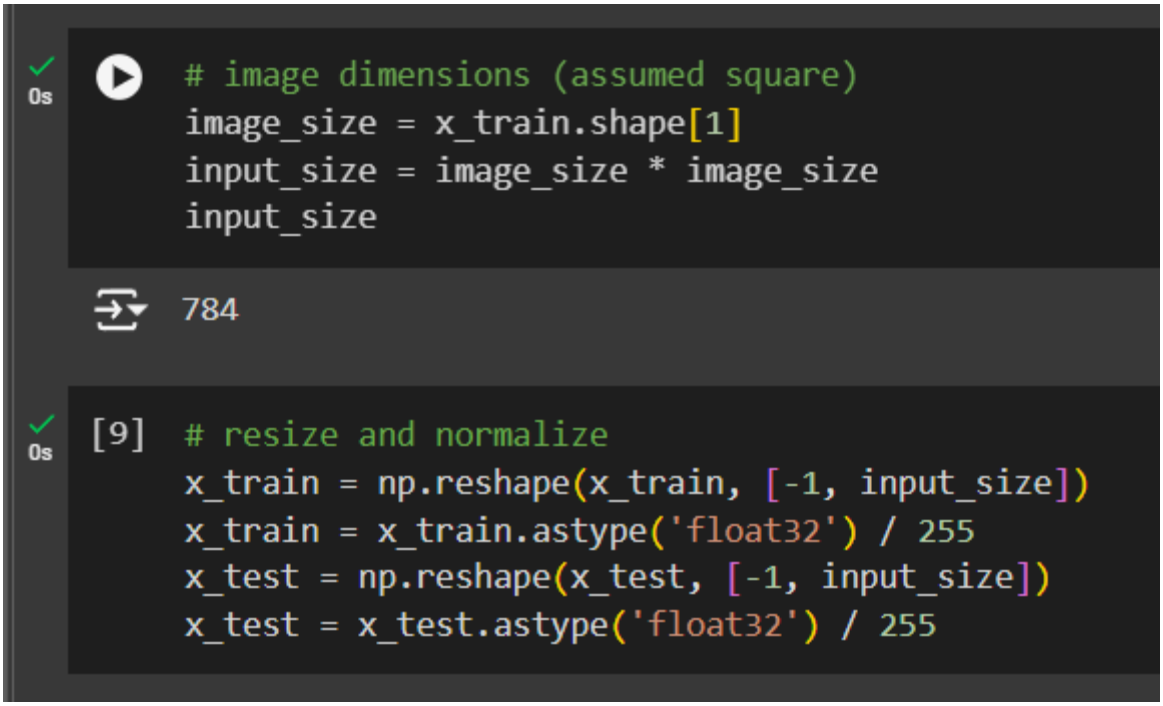
پیش پردازش داده‌ها:

در ابتدا باید برچسب خروجی هر تصویر که یک عدد بین ۰ تا ۹ میباشد را به برداری از ۰ و ۱ تبدیل کنیم. این کار باعث میشود که مدل برچسب ۹ را با ارزش تر از برچسب ۰ نبیند و آنها را صرفا یک دسته بندی ببیند. برای این کار از روش one hot encoding استفاده میکنیم.

پیاده سازی این انکودر به این صورت است:



سپس ماتریس هر عکس را به اصطلاح flatten میکنیم و آن را به یک بردار ۷۸۴ تایی تبدیل میکنیم. در انتها داده ها را نرمال سازی میکنیم. همانطور که اشاره شد داده ها در بازه ۰ تا ۲۵۵ قرار دارند پس با تقسیم کردن آنها بر ۲۵۵ میتوانیم بازه آن ها را به ۰ تا ۱ ببریم که به این کار نرمال سازی داده ها میگویند:



```
# image dimensions (assumed square)
image_size = x_train.shape[1]
input_size = image_size * image_size
input_size
```

```
784
```

```
[9] # resize and normalize
x_train = np.reshape(x_train, [-1, input_size])
x_train = x_train.astype('float32') / 255
x_test = np.reshape(x_test, [-1, input_size])
x_test = x_test.astype('float32') / 255
```

تنظیم ابرپارامترهای مدل:

ابرپارامترهای مدل شامل موارد زیر میشود که هر یک را شرح خواهم داد:

- Batch size: نشان دهنده تعداد نمونه‌های انتخاب شده برای آموزش در هر Epoch است. که مقدار آن را ۱۲۸ انتخاب کرده ام. هر چه این عدد را بزرگ تر قرار دهیم محاسبات هر epoch سنگین تر شده و زمان بیشتری میبرد اما در کل مدل زودتر converge میشود.

- Hidden units: تعداد نورون های هر لایه نهان از شبکه را نشان میدهد. این مقدار را ۲۵۶ قرار داده ام زیرا در ۱۲۸ مدل زود همگرا میشود اما دچار underfit میشود. در مقادیر بالاتر همچون ۵۱۲ و ۱۰۲۴ مدل سنگین تر میشود و زمان بیشتری برای پردازش میبرد اما دقت نهایی تفاوت قابل توجهی نمیکند.

- Drop_out: مقدار این عدد نشان دهنده درصد خروجی نورن هاییست که در انتهای هر لایه نهان drop میشوند. تاثیر drop out کم کردن احتمال وقوع overfit است. بدین صورت که در انتهای هر لایه نهان و بعد از activation function یک لایه drop out میگذاریم تا درصدی از داده ها را دراپ کند تا در هر epoch مدل به تمام داده های train توجه نکند و به گونه ای چالش برای آموزش مدل ایجاد کنیم تا در نهایت وقتی به سراغ آزمایش مدل رفتیم مدل روی داده های آموزش بیش برازش نشده باشد و تاثیر overfitting کاهش یابد. با استفاده از drop out و batch normalization که در ادامه توضیح خواهم داد، نیازی به regularization نیز نخواهد بود. در نهایت میتوان گفت drop out از overfit جلوگیری میکند.

- Number of hidden layers: انتخاب لایه های نهان مسئله چالشی است که به دیتاست ما بستگی دارد. هر چه تعداد لایه های نهان افزایش پیدا کند زمان پردازش نیز طولانی تر خواهد شد. در این پروژه تعداد لایه های نهان یک عدد در نظر گرفته شده است چون مسئله جداناپذیر خطی است ولی خیلی هم پیچیده نیست!

- Learning rate: در الگوریتم بهینه سازی adam نرخ یادگیری به طور adaptive تغییر میکند و نیازی به تنظیم آن نیست اما در sgd نیاز داریم تا آن را تنظیم کنیم. در ادامه بیشتر درباره این موضوع توضیح خواهم داد.

طراحی معماری مدل:

از مدل sequential کراس برای طراحی شبکه عصبی استفاده شده که یک شبکه عصبی تماما متصل برای ما ایجاد میکند. برای افزودن لایه ها از متود add استفاده میکنیم. در نهایت یک مدل ۳ لایه ایجاد کرده ایم که هر لایه دارای یک dense ۲۵۶ نورونی سپس یک activation function و سپس یک batch normalization و در انتها لایه drop out میباشد. لایه آخر نیز یک dense ۱۰ نورونی با تابع فعال سازی softmax میباشد. این انتخاب به این دلیل صورت گرفته است که تعداد کلاس های ما ۱۰ تا میباشد و هر نورون لایه آخر که مقدار بیشتری داشته باشد توسط سافت مکس انتخاب میشود و به عنوان برچسب نهایی داده نشان داده میشود.

معماری نهایی شبکه به صورت زیر آمده است:

Designing the model architecture

```
[11] model = Sequential()

     model.add(Dense(hidden_units, input_dim=input_size, use_bias=False))
     model.add(BatchNormalization())
     model.add(Activation('relu'))
     model.add(Dropout(dropout))

     model.add(Dense(hidden_units, use_bias=False))
     model.add(BatchNormalization())
     model.add(Activation('relu'))
     model.add(Dropout(dropout))

     model.add(Dense(num_labels))
     model.add(Activation('softmax'))
```

دلیل انتخاب relu به عنوان تابع فعال سازی نهایی این است که این مدل در هر لایه ترکیبات خطی بدست می اورد و در نتیجه خروجی نهایی نیز یک خروجی خطی خواهد بود. حال آنکه دیتاست MNIST برای پیش بینی نیاز به یک مدل غیرخطی دارد که relu این کار را برای ما انجام میدهد.

Batch normalization نیز برای این منظور استفاده میشود که ما وقتی داده ها را نرمال سازی میکنیم و به لایه اول میدهیم در خروجی لایه اول اعداد بدست آمده دیگر نرمال نیستند و بهتر است دوباره نرمال سازی شوند. این کار توسط لایه batch normalization صورت گرفته که منجر به افزایش سرعت پردازش و همگرایی مدل میشود.

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

برای تابع loss مدل از categorical cross entropy استفاده شده زیرا که هدف پروژه کلاس بندی است و نه رگرسیون. کراس انتروپی برای داده هایی که برچسب کلاسی دارند مناسب میباشد.

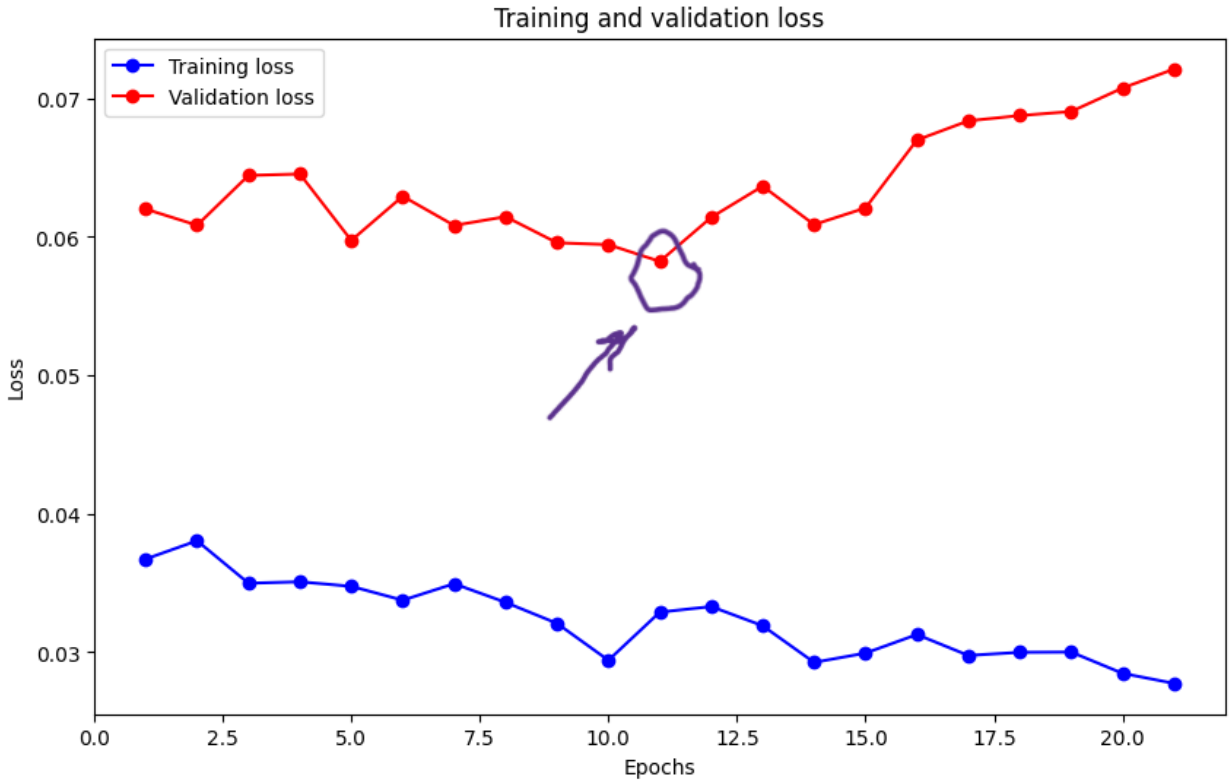
معیار سنجش مدل را نیز accuracy قرار داده ایم و استفاده از متریک های دیگر تفاوتی در نتیجه نمیکند. در رابطه با optimizer مدل در ادامه توضیح میدهم.

```
usualCallback = EarlyStopping(monitor='val_loss', restore_best_weights=True, patience=10)
history = model.fit(x_train, y_train, epochs=100, batch_size=batch_size, validation_split=0.1, shuffle=True, callbacks=[usualCallback])
```

سپس نوبت به fit کردن مدل است.

شرط پایان آموزش را پیدا کردن بهترین validation loss در بین epoch ها قرار داده ام. در واقع تا ده epoch سعی میکند validation loss کمتری بدست بیاورد اگر پیدا نکند مدل همگرا شده و متوقف میشود اگر پیدا شود دوباره به جستجو ادامه میدهد.

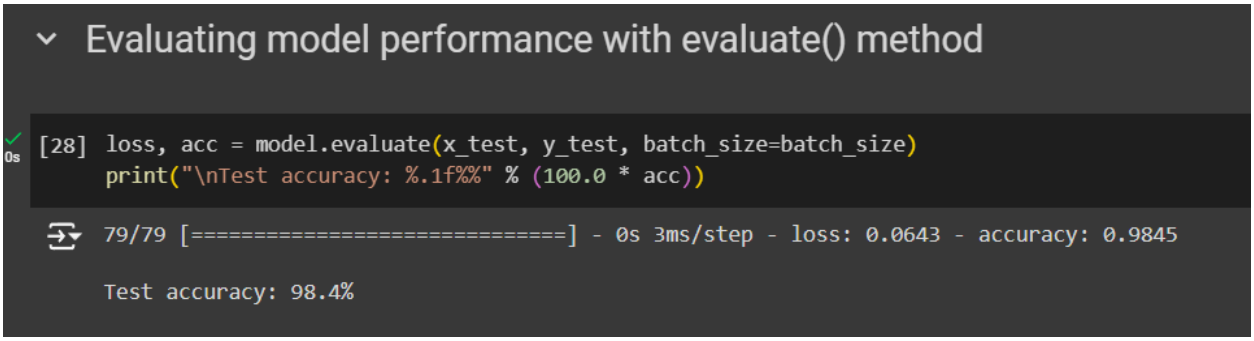
تعداد epoch ها را بالا قرار داده ام تا توقف به شرط همگرایی باشد.



نمونه train loss و validation در بالا اورده شده.

در نقطه مشخص شده بهترین پارامتر ها بدست امده و در انتها نیز از همان ها استفاده میشود.

محاسبه دقت:



دقت نهایی مدل روی داده های تست 98.4% شده است.

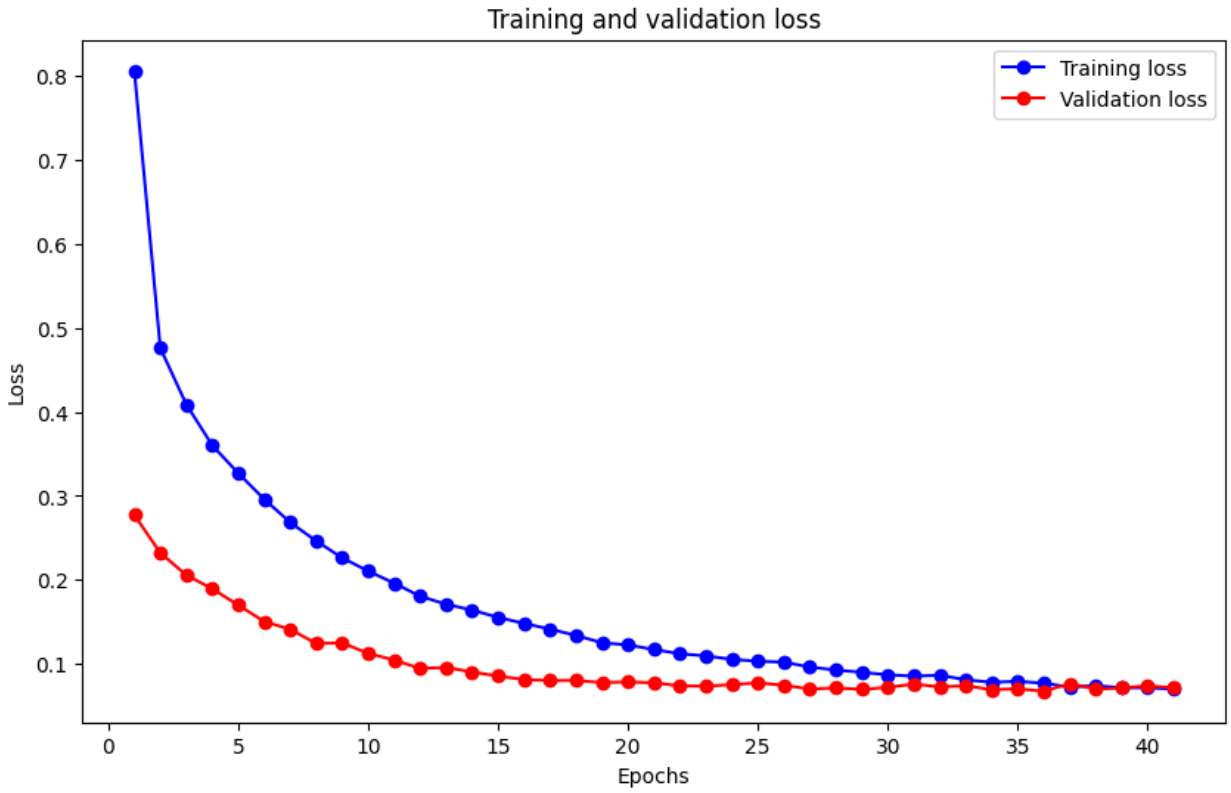
همانگونه که از دقت نهایی روی داده تست مشخص است دچار overfit و underfit نیز نشده ایم.

مقایسه الگوریتم های بهینه سازی و توابع فعال سازی:

در انتها مقایسه بین الگوریتم های sgd و adam و مقایسه توابع فعال سازی relu و sigmoid را شرح میدهم.

توابع فعال سازی:

همانطور که اشاره شد مسئله ما یک مسئله غیر خطی است و تابع فعال سازی رلو این قضیه را به خوبی handle میکند. نتیجه استفاده از sigmoid در زیر اورده شده که همانطور که مشاهده میشود دقت کمتر در تعداد epoch های خیلی بیشتری دارد. پس relu در اینجا بهتر است:



Test accuracy: 97.9%

:Optimizers

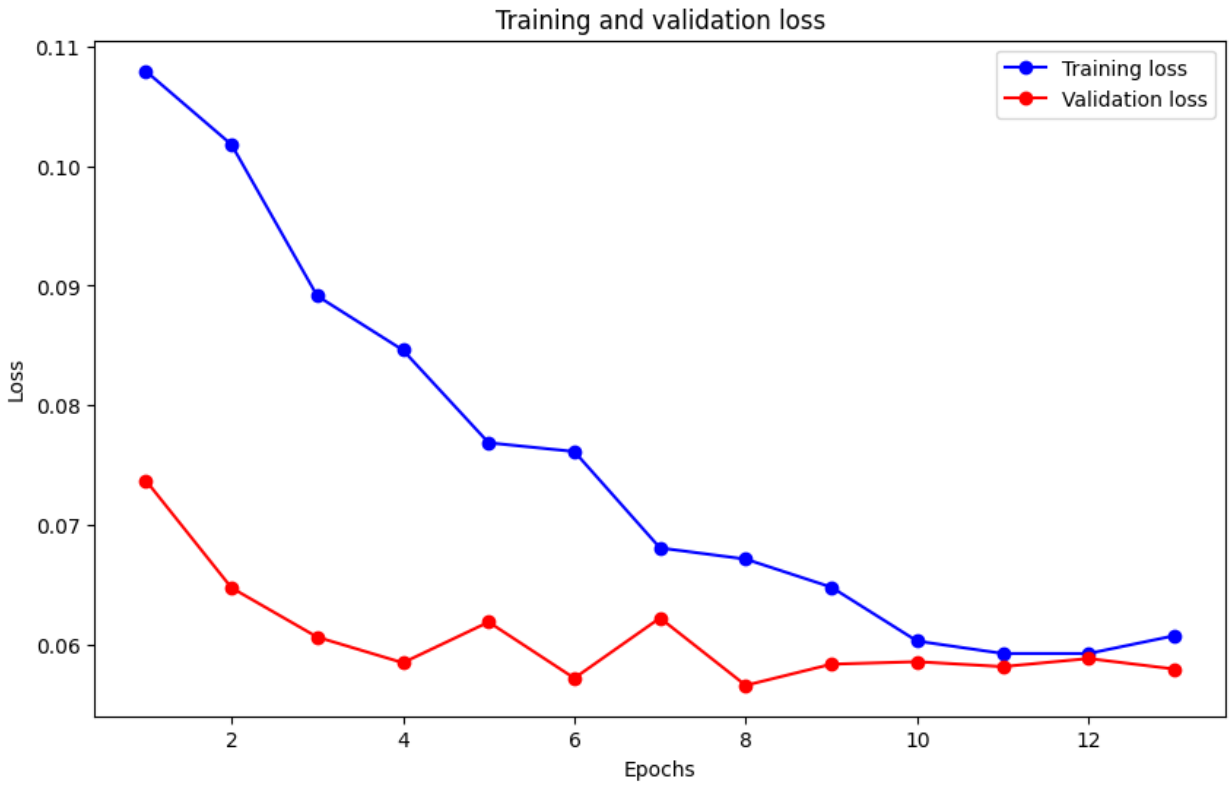
مقایسه دو الگوریتم نام برده شده:

الگوریتم adam میتوان گفت بهتر است زیرا نرخ یادگیری آن به طور adaptive تعیین میشود و در epoch های کمتری converge میشود و دقت نهایی بالایی نیز دارد.

از طرفی sgd نیز الگوریتم خوبیست اما نیاز به تعیین ابرپارامترهای نرخ یادگیری و شتاب میباشد که تعیین آنها میتواند چالشی باشد و یا زمان پردازش اضافه تری بهمراه داشته باشد.

اما در کل دقت نهایی با این الگوریتم روی این دیتاست با تعیین درست پارامتر ها تفاوت چندانی نکرده و دقت و epoch های آن به شرح زیر است:

```
from keras.optimizers import SGD
sgd = SGD(
    learning_rate=0.05,
    momentum=0.9
)
model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])
```



Test accuracy: 98.2%