CPSC 585 - Artificial Neural Networks Project 1

Optical Recognition of Digits using Convolutionary Neural Networks(CNN)

DataSets

emnist_letter.npz

binaryalphadigits.npz

Part 1 - Warm Up

1. Open this notebook by Francois Chollet, which creates a simple Multilayer Perceptron as described in Section 2.1 of Deep Learning with Python, Second Edition. (Recall that this book is available from the library's O'Reilly database.)

shared link: https://colab.research.google.com/drive/1vGt-pZfHI-P1M87qdjJtRoUu2JNx6Cv-?usp=sharing

Chollet's example uses the simpler MNIST dataset, which includes only handwritten digits. That dataset is included with Keras.

Run the model from this notebook. What accuracy does it achieve for MNIST?

The accuracy was found to be: 0.9884

2. Load the EMNIST Letters dataset, and use plt.imshow() to verify that the image data has been loaded correctly and that the corresponding labels are correct.

```
# Imports
import tensorflow as tf
from sklearn.model_selection import train_test_split
import numpy as np
from sklearn import datasets
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
from google.colab import drive
import datetime, os
from tensorflow.keras.models import load_model
from sklearn.metrics import accuracy_score
from keras.preprocessing.image import ImageDataGenerator
from sklearn.model_selection import KFold
```

Section 1

Simple MNIST convnet

Author: fchollet

Date created: 2015/06/19

Last modified: 2020/04/21

Description: A simple convnet that achieves ~99% test accuracy on MNIST.

Section 2

Loading EMNIST Dataset

Preprocessing and plotting data

```
# Downloading Data
# change this to the method recommended by the professor, where we login to Drive and giv
# this will download the EMNIST data set and save it in the notebook temporarily
!curl -o emnist_letters.npz -L 'https://drive.google.com/uc?export=download&confirm=yes&i
!curl -o binaryalphadigits.npz -L 'https://drive.google.com/uc?export=download&confirm=ye
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100 64.9M  100 64.9M    0     0  42.2M      0  0:00:01  0:00:01 --:--:-- 66.8M
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:--  0:00:01 --:--:--     0
100 44758  100 44758    0     0  23311      0  0:00:01  0:00:01 --:--:-- 1507k
```

```
emnist = np.load("emnist_letters.npz")
```

Here the first 10 training examples will be printed

3. Apply the network architecture from Chollet's MNIST notebook to the EMNIST Letters data. (You will need to modify the numbers of inputs and outputs, but should leave the dense layer intact.)

How does this compare with the accuracy for MNIST?

```
#from tensorflow.keras.datasets import mnist
#(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = emnist.get('train_images')
train_labels = emnist.get('train_labels')
val_images = emnist.get('validate_images')
val_labels = emnist.get('validate_labels')
test_images = emnist.get('test_images')
test_labels = emnist.get('test_labels')


print('Train Shapes - Data: {} , Labels: {} '.format(train_images.shape,train_labels.shap
print('Val Shapes - Data: {} , Labels: {} '.format(val_images.shape,val_labels.shape))
print('Test Shapes - Data: {} , Labels: {} '.format(test_images.shape,test_labels.shape))
```

```
Train Shapes - Data: (104000, 784) , Labels: (104000, 27)
Val Shapes - Data: (20800, 784) , Labels: (20800, 27)
Test Shapes - Data: (20800, 784) , Labels: (20800, 27)
```

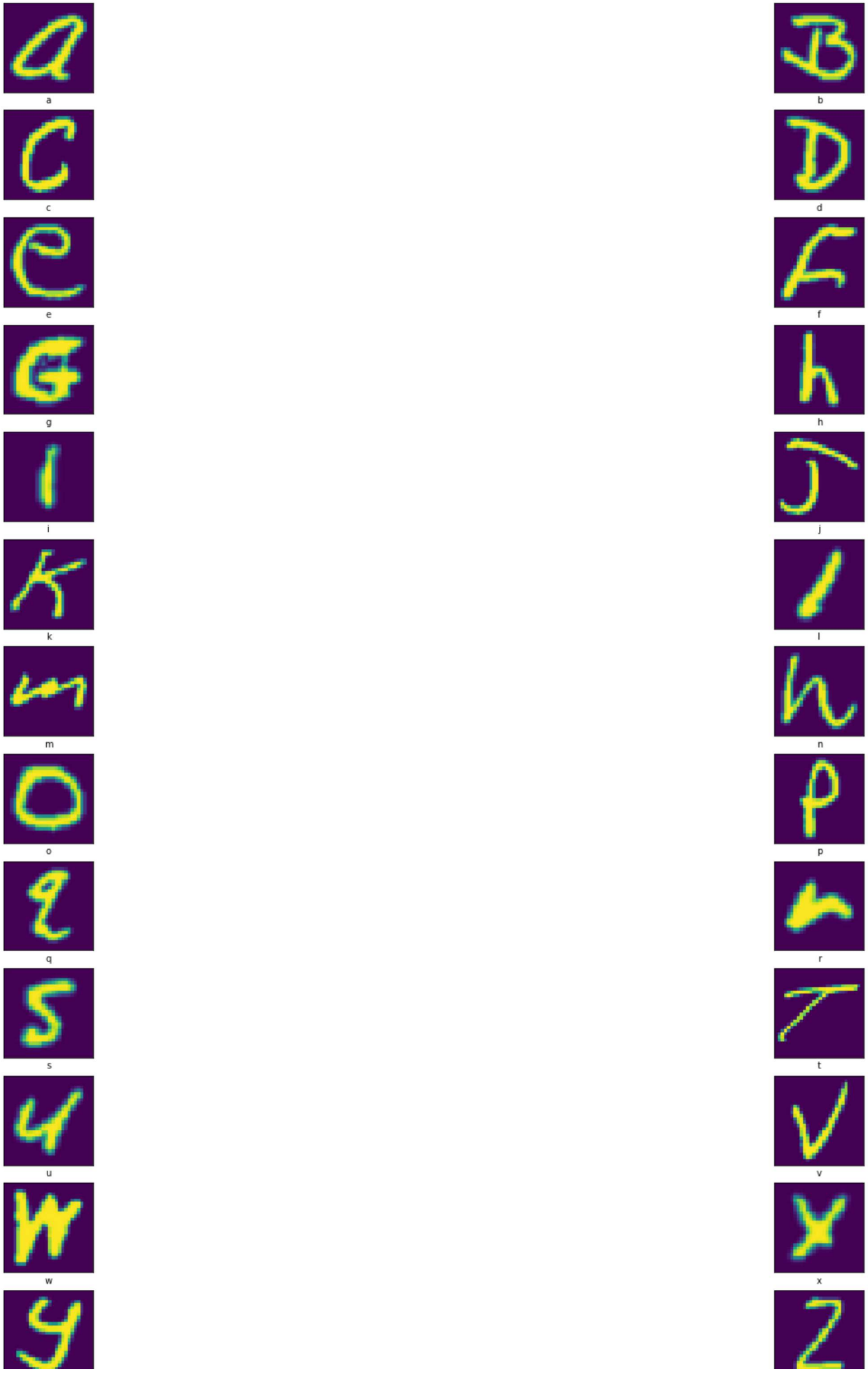
```
indexs=[None]*27
for i,label in enumerate(train_labels):
  if indexs[np.argmax(label)] == None:
    indexs[np.argmax(label)]=i
  if None not in indexs[1:]:
    break
print(indexs)
alphas='$abcdefghijklmnopqrstuvwxyz'
```

```
[None, 29, 20, 42, 37, 19, 51, 1, 36, 22, 10, 7, 28, 6, 11, 3, 2, 5, 12, 18, 45, 13,
```

```
np.argmax(train_labels[29])
```

```
1
```

```
plt.figure(figsize=(28,28))

for i,v in enumerate(indexs[1:]):
  plt.subplot(13,2,i+1)
  plt.xticks([])
  plt.yticks([])
  plt.grid(False)
  plt.imshow(train_images[v].reshape(28,28))
  plt.xlabel(alphas[np.argmax(train_labels[v])])
plt.show()
```


a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x

Section 3

The network architecture

```
cholletModel = keras.Sequential([
    layers.Dense(512, activation="relu",input_shape=(784,)),
    layers.Dense(27, activation="softmax")
])

cholletModel.compile(loss="sparse_categorical_crossentropy", optimizer="adam", metrics=["

cholletModel.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 512)               401920

 dense_1 (Dense)             (None, 27)                13851

=================================================================
Total params: 415,771
Trainable params: 415,771
```

```
Non-trainable params: 0
_________________________________________________________________
```

Before feeding to Francois Chollet's Network Reshapping Data

Preparing the image data

```
# Reshapping Labels
train_labels=np.argmax(train_labels,axis=1)
val_labels=np.argmax(val_labels,axis=1)
test_labels=np.argmax(test_labels,axis=1)
```

Add Tensorboard extension for notebook

```
%load_ext tensorboard
```

Specify the location for logDirectory and callback function to log various metrics and visualize them in TensorBoard.

```
logdir = os.path.join("logs", datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
tensorboard_callback = tf.keras.callbacks.TensorBoard(logdir, histogram_freq=1)
```

Fit the model with Tensorboard callback for visualization Purpose

```
train_labels.shape
```

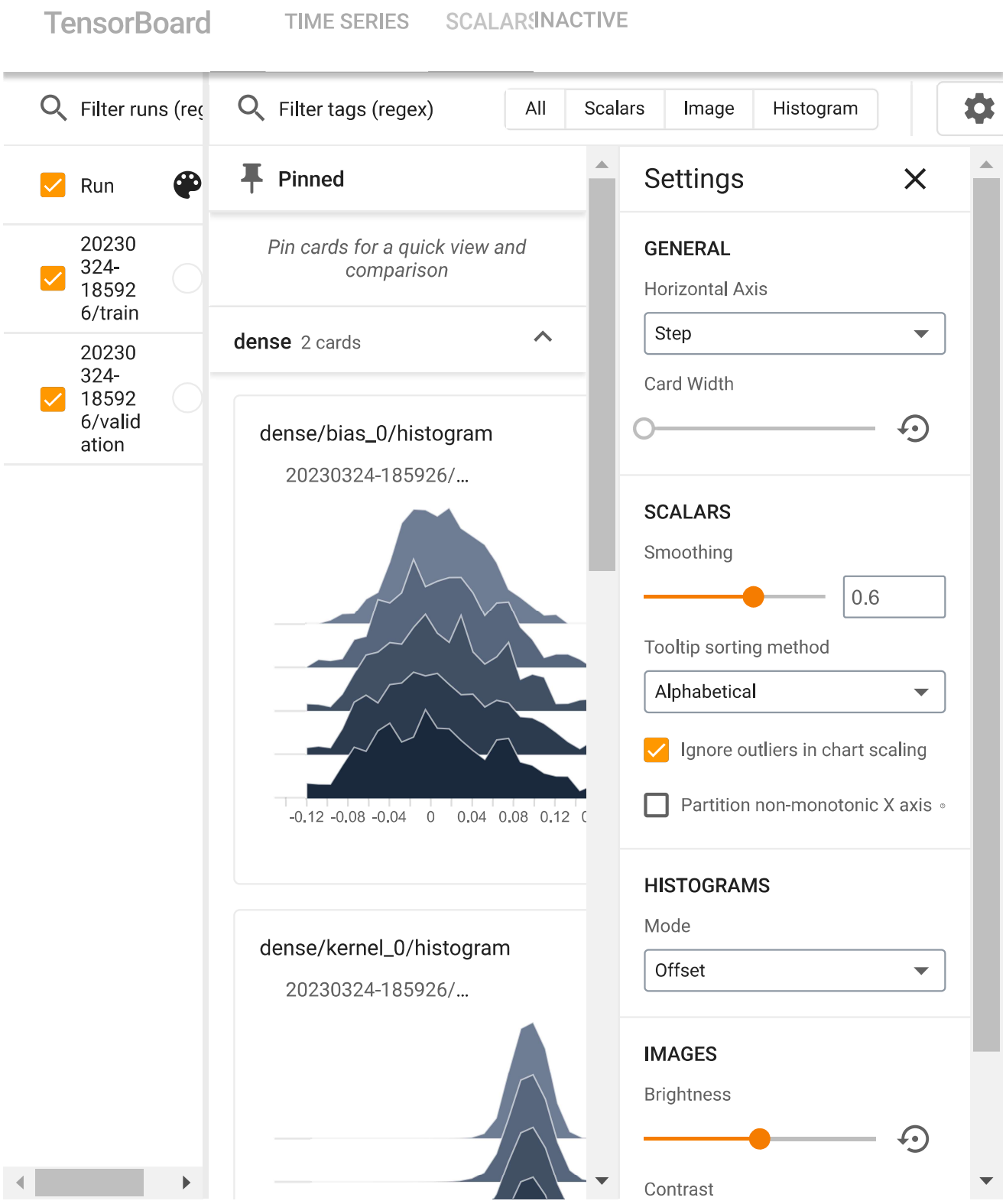
```
(104000,)
```

```
cholletHistory = cholletModel.fit(train_images,
                                  train_labels,
                                  batch_size=128,
                                  epochs=5,
                                  validation_data=(val_images,val_labels),
                                  callbacks=[tensorboard_callback]
                                  )
```

```
Epoch 1/5
813/813 [==============================] - 15s 18ms/step - loss: 0.7592 - accuracy: 0
Epoch 2/5
813/813 [==============================] - 14s 17ms/step - loss: 0.3916 - accuracy: 0
Epoch 3/5
813/813 [==============================] - 15s 18ms/step - loss: 0.3098 - accuracy: 0
Epoch 4/5
813/813 [==============================] - 15s 18ms/step - loss: 0.2651 - accuracy: 0
Epoch 5/5
813/813 [==============================] - 15s 18ms/step - loss: 0.2338 - accuracy: 0
```

Launch Tensorboard

```
%tensorboard --logdir logs
```



Predictions using chollet's Model

Using the model to make predictions

```
cholletPredictions = np.argmax(cholletModel.predict(test_images),axis=1)
cholletPredictions.shape
```

```
650/650 [==============================] - 7s 10ms/step
(20800,)
```

```
print(f'EMNIST data using Chollet\'s model, Accuracy: {accuracy_score(test_labels,chollet
```

```
EMNIST data using Chollet's model, Accuracy: 0.8988461538461539
```

Section 4

The Keras examples include a Simple MNIST convnet. Note the accuracy obtained by that code compared to the previous example from Chollet.

Applying Same to ConvNet from keras

Prepare the data

```
# Model / data parameters
num_classes = 27
input_shape = (28, 28, 1)

# Reshaping to input data to 28x28 (2D)
train_images2d = train_images.reshape((-1,28,28,1))
val_images2d = val_images.reshape((-1,28,28,1))
test_image2d = test_images.reshape((-1,28,28,1))
```

```
print(train_images2d.shape)
print(train_labels.shape)
print(val_images2d.shape)
print(val_labels.shape)
```

```
(104000, 28, 28, 1)
(104000,)
(20800, 28, 28, 1)
(20800,)
```

Build the model

```
convnet = keras.Sequential(
    [
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation="softmax"),
    ]
)

convnet.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2D  (None, 13, 13, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 5, 5, 64)         0
 2D)

 flatten (Flatten)           (None, 1600)              0

 dropout (Dropout)           (None, 1600)              0

 dense_2 (Dense)             (None, 27)                43227

=================================================================
Total params: 62,043
Trainable params: 62,043
Non-trainable params: 0
_________________________________________________________________
```

Train the model

```
batch_size = 128
epochs = 15

convnet.compile(loss="sparse_categorical_crossentropy", optimizer="adam", metrics=["accur

convnet.fit(train_images2d,
            train_labels,
            batch_size=batch_size,
            epochs=epochs,
            validation_data=(val_images2d,val_labels),
            callbacks=[tensorboard_callback]
            )
```

```
Epoch 1/15
813/813 [==============================] - 119s 145ms/step - loss: 0.8342 - accuracy:
Epoch 2/15
813/813 [==============================] - 115s 141ms/step - loss: 0.4352 - accuracy:
Epoch 3/15
813/813 [==============================] - 121s 149ms/step - loss: 0.3699 - accuracy:
Epoch 4/15
813/813 [==============================] - 126s 156ms/step - loss: 0.3366 - accuracy:
Epoch 5/15
813/813 [==============================] - 125s 154ms/step - loss: 0.3114 - accuracy:
Epoch 6/15
813/813 [==============================] - 118s 146ms/step - loss: 0.2933 - accuracy:
Epoch 7/15
813/813 [==============================] - 118s 145ms/step - loss: 0.2788 - accuracy:
Epoch 8/15
813/813 [==============================] - 114s 140ms/step - loss: 0.2674 - accuracy:
Epoch 9/15
813/813 [==============================] - 115s 141ms/step - loss: 0.2612 - accuracy:
Epoch 10/15
813/813 [==============================] - 122s 150ms/step - loss: 0.2513 - accuracy:
Epoch 11/15
813/813 [==============================] - 118s 146ms/step - loss: 0.2449 - accuracy:
Epoch 12/15
813/813 [==============================] - 119s 147ms/step - loss: 0.2407 - accuracy:
Epoch 13/15
813/813 [==============================] - 117s 143ms/step - loss: 0.2359 - accuracy:
Epoch 14/15
813/813 [==============================] - 110s 136ms/step - loss: 0.2293 - accuracy:
Epoch 15/15
813/813 [==============================] - 117s 144ms/step - loss: 0.2275 - accuracy:
<keras.callbacks.History at 0x7ff1bd9595e0>
```

Evaluate the trained model

```
score = convnet.evaluate(test_image2d, test_labels, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

```
Test loss: 0.21137887239456177
Test accuracy: 0.932692289352417
```

Keras Model gave better performace than chollet's model

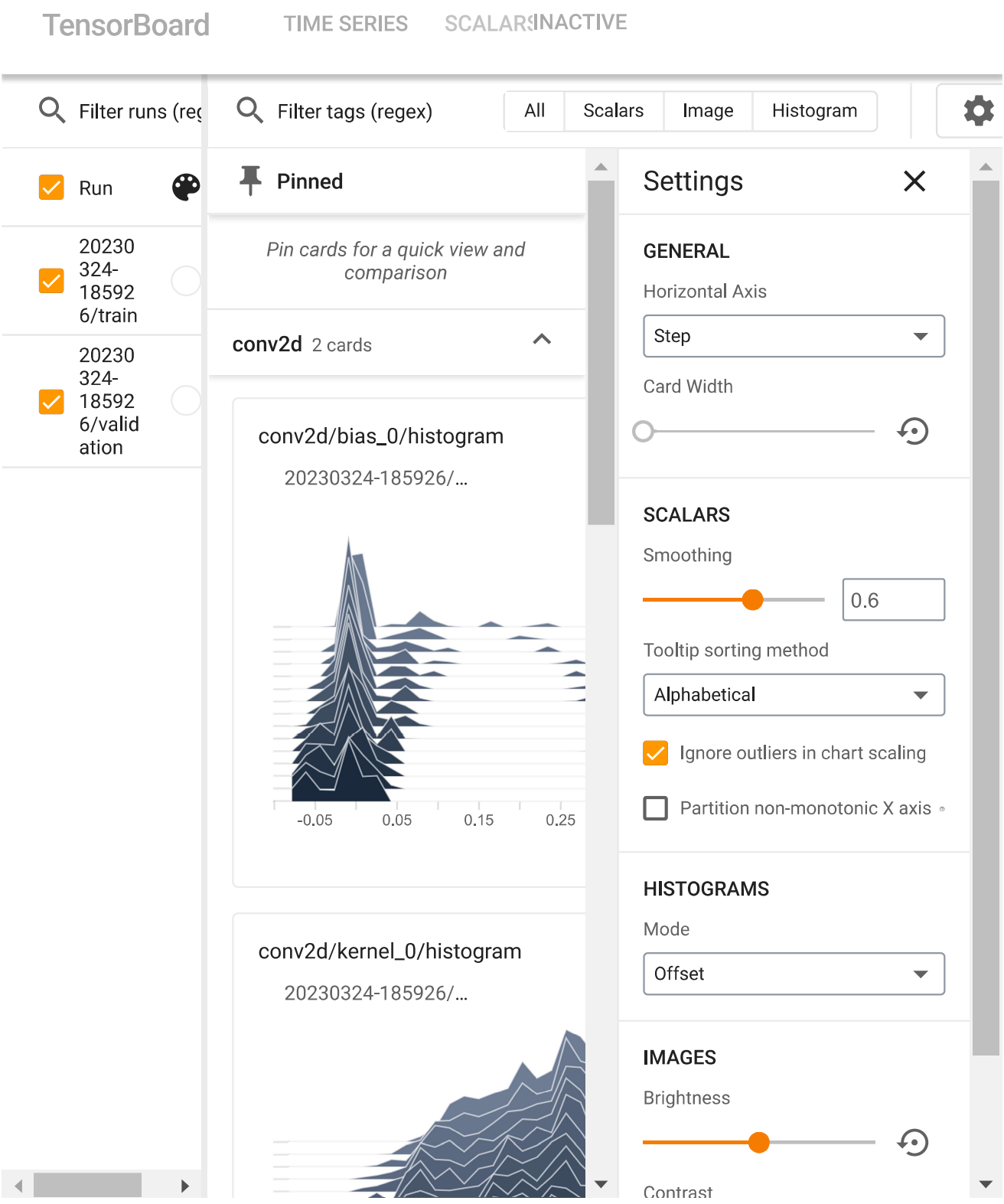
Part 2

Add TensorBoard support to the CNN model you run in Part 1, and add TensorBoard to your notebook to visualize the training process.

Section 5

```
%tensorboard --logdir logs
```

```
Reusing TensorBoard on port 6006 (pid 2292), started 0:35:39 ago. (Use '!kill 2292'
to kill it.)
```



Section 6

6. Now that you have a baseline convolutional network for comparison, begin experimenting with alternative architectures (e.g. adding additional filters to learn features and additional hidden layers to learn combinations of features) and with adjusting hyperparameters. Your team's goal is to obtain as high an accuracy as possible on the validation set. Use what

you've learned in Chapters 2 through 5 of the textbook to obtain the highest accuracy you can, including:

- Weight initialization
- Choice of activation function
- Choice of optimizer
- Batch normalization
- Data augmentation
- Regularization
- Dropout
- Early Stopping
- Pooling

```
#Add Imports if any

#Initialize variables
batchSize=512 #128
epochs=30

early_stopping = keras.callbacks.EarlyStopping(
        # Stop training when `val_loss` is no longer improving
        monitor='val_loss',
        # "no longer improving" being defined as "no better than 1e-2 less"
        min_delta=1e-3,
        # "no longer improving" being further defined as "for at least 10 epochs"
        patience=7,
        verbose=1)
```

Description of our model. We found that these didn't help increase the validation accuracy. There was a challenge in getting the validation accuracy above 93%.

1. Data Augmentation - This seemed to lower validation accuracy by about 5-10%
2. Regularization (L1, L2) - This lowered validation accuracy by about 5-10%
3. Other activation functions such as sigmoid didn't make improvements
4. Using other weight initializations other than the default. These changes made little differences.
5. Increasing the number of layers beyond 2 Conv2D and 2 Dense didn't seem to help
6. The pool size for max pooling was changed to 3x3, but that didn't help

These did help increase training accurracy and/or validation accuracy:

1. Increasing batch size to 512
2. Batch Normalization
3. Dropout (both on Conv23 and Dense layers)
4. Specifing valid padding showed some improvement, but same didn't improve further.
5. The 5x5 filter size was an improvment over 3x3

Lastly, early stopping was useful.

Our model is below (newCNNmodel), while a previous iteration is newCNNmodel_previous.

With the help of our previous model newCNNmodel_previous, we were able to achieve upto 96% training accuracy and 92% validation accuracy.In order to reduce overfitting,we added few more layers in our newCNNmodel which is the final model.

You can go through the graphs for epoch loss and epoch accuracy using Tensorboard to visualize the results of newCNNmodel on training and validation set.

```
newCNNmodel_previous = keras.Sequential(
    [
    layers.Conv2D(32, (5, 5), padding='valid', input_shape=(28, 28, 1), activation='relu'
         layers.BatchNormalization(),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Dropout(0.2),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(27, activation='softmax'),
     ]
   )

newCNNmodel = keras.Sequential(
    [
    layers.Conv2D(32, (5, 5), padding='valid', input_shape=(28, 28, 1), activation='relu'
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(32, (5, 5), padding='valid', activation='relu'),
    layers.BatchNormalization(),
        layers.SpatialDropout2D(0.2),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Flatten(),
        layers.Dropout(0.2),
    layers.Dense(128, activation='relu'),
    layers.Dense(27, activation='softmax'),
     ]
   )
newCNNmodel.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_8 (Conv2D)           (None, 24, 24, 32)        832

 max_pooling2d_8 (MaxPooling  (None, 12, 12, 32)       0
 2D)

 conv2d_9 (Conv2D)           (None, 8, 8, 32)          25632

 batch_normalization_3 (Batc  (None, 8, 8, 32)         128
 hNormalization)

 spatial_dropout2d_1 (Spatia  (None, 8, 8, 32)         0
 lDropout2D)
```

```
 max_pooling2d_9 (MaxPooling  (None, 4, 4, 32)         0
 2D)

 flatten_5 (Flatten)         (None, 512)               0

 dropout_5 (Dropout)         (None, 512)               0

 dense_10 (Dense)            (None, 128)               65664

 dense_11 (Dense)            (None, 27)                3483

=================================================================
Total params: 95,739
Trainable params: 95,675
Non-trainable params: 64
_________________________________________________________________
```

Compile the model trained above

```
#newCNNmodel.compile(loss='sparse_categorical_crossentropy', optimizer='rmsprop', metrics
newCNNmodel.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['a
```

Train the improved CNN model and add tensorboard support for better visualization

```
# previous
#history=newCNNmodel.fit(train_images2d, train_labels, batch_size=batchSize, epochs=epoch
#            callbacks=[tensorboard_callback])

history=newCNNmodel.fit(
    train_images2d, train_labels,
    batch_size=batchSize, epochs=epochs,
    validation_data=(val_images2d, val_labels),
    callbacks=[tensorboard_callback, early_stopping]
)
```

```
 Epoch 1/30
 204/204 [==============================] - 122s 590ms/step - loss: 0.8436 - accuracy:
 Epoch 2/30
 204/204 [==============================] - 121s 593ms/step - loss: 0.3485 - accuracy:
 Epoch 3/30
 204/204 [==============================] - 121s 593ms/step - loss: 0.2825 - accuracy:
 Epoch 4/30
 204/204 [==============================] - 117s 574ms/step - loss: 0.2464 - accuracy:
 Epoch 5/30
 204/204 [==============================] - 117s 572ms/step - loss: 0.2244 - accuracy:
 Epoch 6/30
 204/204 [==============================] - 115s 562ms/step - loss: 0.2115 - accuracy:
 Epoch 7/30
 204/204 [==============================] - 117s 577ms/step - loss: 0.1988 - accuracy:
 Epoch 8/30
 204/204 [==============================] - 121s 591ms/step - loss: 0.1896 - accuracy:
 Epoch 9/30
 204/204 [==============================] - 121s 592ms/step - loss: 0.1837 - accuracy:
 Epoch 10/30
```

```
204/204 [==============================] - 124s 608ms/step - loss: 0.1772 - accuracy:
Epoch 11/30
204/204 [==============================] - 131s 641ms/step - loss: 0.1706 - accuracy:
Epoch 12/30
204/204 [==============================] - 123s 601ms/step - loss: 0.1658 - accuracy:
Epoch 13/30
204/204 [==============================] - 118s 581ms/step - loss: 0.1617 - accuracy:
Epoch 14/30
204/204 [==============================] - 126s 618ms/step - loss: 0.1561 - accuracy:
Epoch 15/30
204/204 [==============================] - 119s 585ms/step - loss: 0.1514 - accuracy:
Epoch 16/30
204/204 [==============================] - 119s 585ms/step - loss: 0.1509 - accuracy:
Epoch 17/30
204/204 [==============================] - 126s 621ms/step - loss: 0.1476 - accuracy:
Epoch 18/30
204/204 [==============================] - 121s 596ms/step - loss: 0.1440 - accuracy:
Epoch 19/30
204/204 [==============================] - 120s 590ms/step - loss: 0.1404 - accuracy:
Epoch 20/30
204/204 [==============================] - 127s 620ms/step - loss: 0.1385 - accuracy:
Epoch 20: early stopping
```

Early Stopping worked and stopped at 20 epochs

Section 7

saving the model and evaluate the results on the test set.

As we encountered issues while loading data for binaryalpha digits from downloaded npz file, we uploaded them to google drive and it's working.

```
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.m
```

```
newCNNmodel.save('/content/drive/MyDrive/mynewCNNmodel.h5')
```

Evaluate the test accuracy and test loss from CNN model trained above

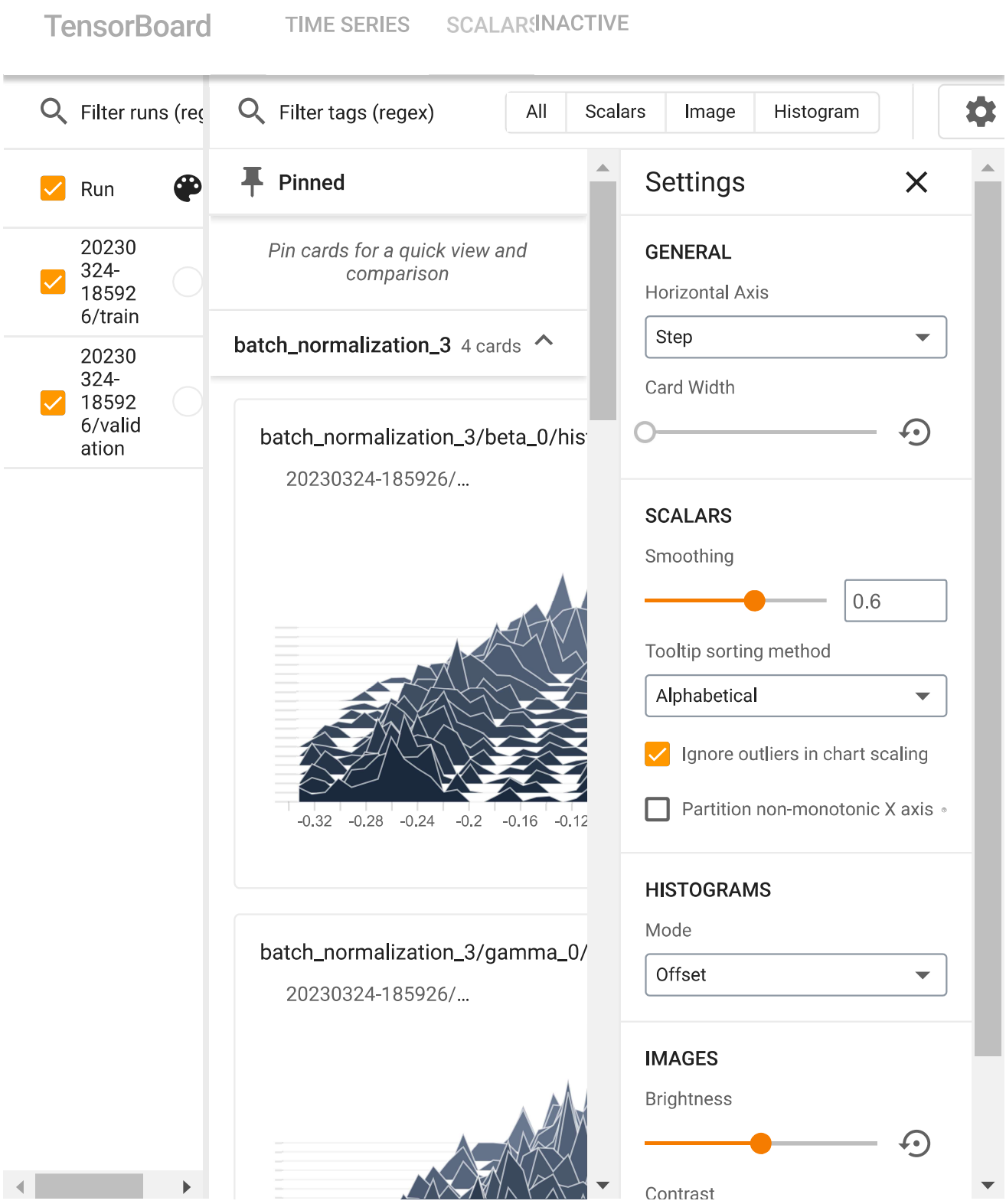
```
newscore = newCNNmodel.evaluate(test_image2d, test_labels, verbose=0)
print("Test loss:", newscore[0])
print("Test accuracy:", newscore[1])
```

```
Test loss: 0.17654144763946533
Test accuracy: 0.9426442384719849
```

Launch Tensorboard and verify the results for epoch_loss and epoch_accuracy

```
%tensorboard --logdir logs
```

```
Reusing TensorBoard on port 6006 (pid 2292), started 1:36:53 ago. (Use '!kill 2292'
to kill it.)
```



Part 3

Using Binaryalpha digits data set.

Section 8

```
bialpha = np.load("binaryalphadigits.npz")
images_bialpha = bialpha.get('images')
labels_bialpha = bialpha.get('labels')
```

```
drive.mount('/content/drive')
```

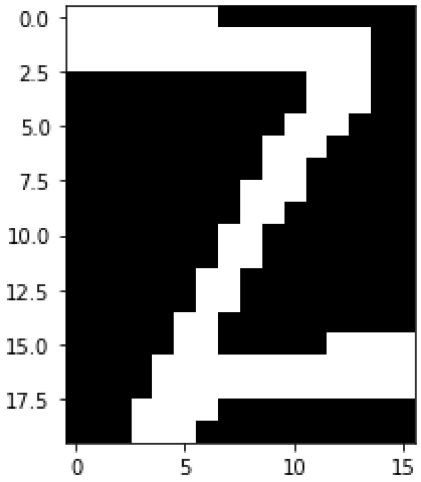
```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.m
```

Loading Labels

Reshaping `images_bialpha` to 2D of 20x16 size

```
images_bialpha = images_bialpha.reshape((-1, 20, 16))
```

```
plt.imshow(images_bialpha[1013], cmap='gray')
plt.show()
```

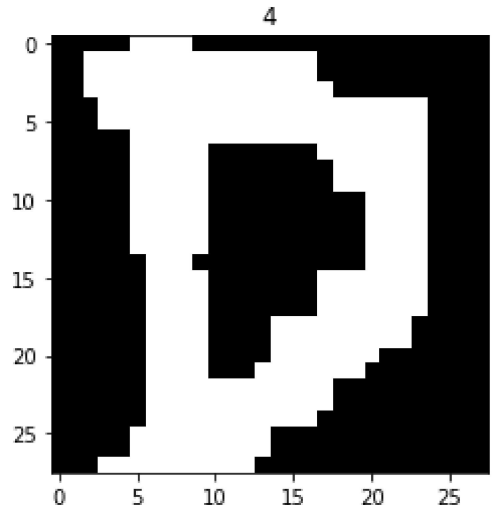


ReSizing Images by expanding images to 28x28 size

```
#from tensorflow.python.framework.tensor_util import constant_value
resized_images = []

# Resize each image to 28x28 and append it to the resized_images list
for image in images_bialpha:
    image_array = np.array(image)
    image_array = np.expand_dims(image_array, axis=-1)
    resized_image = tf.image.resize_with_pad(image_array, 28, 28, method='nearest')
    resized_images.append(np.array(resized_image))

# Convert the list of resized images to a NumPy array
resized_images = np.stack(resized_images)

# Verify the shape of the resized_images array
print(resized_images.shape)
```

```
(1014, 28, 28, 1)
```

```
plt.title(np.argmax(labels_bialpha[150]))
plt.imshow(resized_images[150], cmap='gray')
plt.show()
```




```
resized_images.shape
```

```
(1014, 28, 28, 1)
```

Section 9

model training in Part 2 is saved as `mynewCNNModel.h5` into Drive and link here

```
loadednewCNNmodel = load_model('/content/drive/MyDrive/mynewCNNmodel.h5')
```

```
predictions = loadednewCNNmodel.predict(resized_images)
```

```
32/32 [==============================] - 1s 13ms/step
```

```
predicted_labels = np.argmax(predictions, axis=1)

# Convert the one-hot encoded labels to class labels
true_labels = np.argmax(labels_bialpha, axis=1)

# Calculate the accuracy of the model
accuracy = accuracy_score(true_labels, predicted_labels)

print(f"Accuracy: {accuracy}")
```

```
Accuracy: 0.7159763313609467
```

9. Is the model you trained in Part 2 capable of recognizing letters from this new dataset?

The model trained in part 2 has an accuracy of about 71.5%. It recognized most characters, but not enough. The model is not generalized enough to accomidate this new dataset. We will need to further improve this model.

Section 10

Can you improve the performance on this dataset by adding additional trainable layers and fine tuning the network?

Freezing model from part 2 and fine tuning with aditional layers. This is called the Fine Tuned Model.

```
# Freeze the base model layers
for layer in loadednewCNNmodel.layers:
    layer.trainable = False


# Adding classifier layers on top
finetunedModel = keras.Sequential([
    loadednewCNNmodel,
    layers.Flatten(),
    layers.Dense(512, activation='relu'),
    layers.Dense(128, activation='relu'),
    layers.Dense(27, activation='softmax')
])


# Compile the model
finetunedModel.compile(loss="sparse_categorical_crossentropy", optimizer="adam", metrics=
```

Double-click (or enter) to edit

```
finetunedModel.fit(
    resized_images,
    true_labels,
    batch_size=128,
    epochs=15
)
```

```
Epoch 1/15
8/8 [==============================] - 4s 66ms/step - loss: 3.2226 - accuracy: 0.3767
Epoch 2/15
8/8 [==============================] - 1s 69ms/step - loss: 3.0340 - accuracy: 0.6400
Epoch 3/15
8/8 [==============================] - 0s 62ms/step - loss: 2.8003 - accuracy: 0.6450
Epoch 4/15
8/8 [==============================] - 0s 43ms/step - loss: 2.4621 - accuracy: 0.6706
Epoch 5/15
8/8 [==============================] - 0s 37ms/step - loss: 2.0905 - accuracy: 0.6686
Epoch 6/15
8/8 [==============================] - 0s 39ms/step - loss: 1.6811 - accuracy: 0.6657
Epoch 7/15
8/8 [==============================] - 0s 39ms/step - loss: 1.4691 - accuracy: 0.6548
Epoch 8/15
8/8 [==============================] - 0s 39ms/step - loss: 1.2698 - accuracy: 0.6785
Epoch 9/15
8/8 [==============================] - 0s 38ms/step - loss: 1.1362 - accuracy: 0.6953
Epoch 10/15
8/8 [==============================] - 0s 38ms/step - loss: 1.1180 - accuracy: 0.7002
Epoch 11/15
8/8 [==============================] - 0s 37ms/step - loss: 1.1919 - accuracy: 0.6795
Epoch 12/15
8/8 [==============================] - 0s 39ms/step - loss: 1.0772 - accuracy: 0.7101
Epoch 13/15
8/8 [==============================] - 0s 41ms/step - loss: 1.0763 - accuracy: 0.6982
Epoch 14/15
8/8 [==============================] - 0s 37ms/step - loss: 1.0315 - accuracy: 0.7110
Epoch 15/15
8/8 [==============================] - 0s 35ms/step - loss: 1.0360 - accuracy: 0.7081
<keras.callbacks.History at 0x7ff1979f4640>
```

Fine Tuning is faster as only new layers are being trained.

Adding these extra layers did not improve upon the accuracy with this Binary AlphaDigits dataset.

Section 11

Compare the performance of the model you built in step (3) with the performance of a brand-new model trained only on the Binary AlphaDigits dataset.

This is called the Brand New Model.

```
#brand new model only on binary alpha digits

input_shape = (28, 28, 1)
only_binary_model = keras.Sequential(
    [
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(27, activation="softmax"),
    ]
)

only_binary_model.summary()
```

```
Model: "sequential_30"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_22 (Conv2D)          (None, 26, 26, 32)        320

 max_pooling2d_22 (MaxPoolin  (None, 13, 13, 32)       0
 g2D)

 conv2d_23 (Conv2D)          (None, 11, 11, 64)        18496

 max_pooling2d_23 (MaxPoolin  (None, 5, 5, 64)         0
 g2D)

 flatten_29 (Flatten)        (None, 1600)              0

 dropout_27 (Dropout)        (None, 1600)              0

 dense_57 (Dense)            (None, 27)                43227

=================================================================
Total params: 62,043
Trainable params: 62,043
Non-trainable params: 0
_________________________________________________________________
```

Using Data Augementation to as data is small

```
datagen = ImageDataGenerator(
    rotation_range=10,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=False)

datagen.fit(resized_images)
```

```
batch_size = 128
epochs = 15

only_binary_model.compile(loss="sparse_categorical_crossentropy", optimizer="adam", metri
```

Using Cross-validation as dataset is small

```
k = 5
kf = KFold(n_splits=k, shuffle=True)


for fold, (train_idx, val_idx) in enumerate(kf.split(resized_images)):

    print(f"Fold: {fold}")

    # Split the data into training and validation sets
    train_images = resized_images[train_idx]
    train_labels = true_labels[train_idx]
    val_images = resized_images[val_idx]
    val_labels = true_labels[val_idx]

    # Fit the model on the training set
    history = only_binary_model.fit_generator(
        datagen.flow(train_images, train_labels, batch_size=batch_size),
        steps_per_epoch=len(train_images) // batch_size,
        epochs=epochs,
        validation_data=(val_images, val_labels),
    )

    # Evaluate the model on the validation set
    loss, accuracy = only_binary_model.evaluate(val_images, val_labels)
    print(f"Validation loss: {loss}")
    print(f"Validation accuracy: {accuracy}")
```

```
 Fold: 0
```