KHUSHI J. H.

ISHANI SINGH

[GROUP 2]

**CODES**

## Data collection

!wget -q https://git.io/J0fjL -O IAM\_Words.zip

!unzip -qq IAM\_Words.zip

!

!mkdir data

!mkdir data/words

!tar -xf IAM\_Words/words.tgz -C data/words

!mv IAM\_Words/words.txt data

Preview how the dataset is organized. Lines prepended by "#" are just metadata information.

!head -50 data/words.txt

## Imports

from tensorflow.keras.layers.experimental.preprocessing import StringLookup

from tensorflow import keras

import matplotlib.pyplot as plt

import tensorflow as tf

import numpy as np

import os

np.random.seed(42)

tf.random.set\_seed(42)

## Dataset splitting

base\_path = "data"

words\_list = []

words = open(f"{base\_path}/words.txt", "r").readlines()

for line in words:

if line[0] == "#":

continue

if line.split(" ")[1] != "err": # We don't need to deal with errored entries.

words\_list.append(line)

len(words\_list)

np.random.shuffle(words\_list)

We will split the dataset into three subsets with a 90:5:5 ratio (train:validation:test).

split\_idx = int(0.9 \* len(words\_list))

train\_samples = words\_list[:split\_idx]

test\_samples = words\_list[split\_idx:]

val\_split\_idx = int(0.5 \* len(test\_samples))

validation\_samples = test\_samples[:val\_split\_idx]

test\_samples = test\_samples[val\_split\_idx:]

assert len(words\_list) == len(train\_samples) + len(validation\_samples) + len(

test\_samples

)

print(f"Total training samples: {len(train\_samples)}")

print(f"Total validation samples: {len(validation\_samples)}")

print(f"Total test samples: {len(test\_samples)}")

## Data input pipeline

We start building our data input pipeline by first preparing the image paths.

base\_image\_path = os.path.join(base\_path, "words")

def get\_image\_paths\_and\_labels(samples):

paths = []

corrected\_samples = []

for (i, file\_line) in enumerate(samples):

line\_split = file\_line.strip()

line\_split = line\_split.split(" ")

# Each line split will have this format for the corresponding image:

# part1/part1-part2/part1-part2-part3.png

image\_name = line\_split[0]

partI = image\_name.split("-")[0]

partII = image\_name.split("-")[1]

img\_path = os.path.join(

base\_image\_path, partI, partI + "-" + partII, image\_name + ".png"

)

if os.path.getsize(img\_path):

paths.append(img\_path)

corrected\_samples.append(file\_line.split("\n")[0])

return paths, corrected\_samples

train\_img\_paths, train\_labels = get\_image\_paths\_and\_labels(train\_samples)

validation\_img\_paths, validation\_labels = get\_image\_paths\_and\_labels(validation\_samples)

test\_img\_paths, test\_labels = get\_image\_paths\_and\_labels(test\_samples)

Then we prepare the ground-truth labels.

# Find maximum length and the size of the vocabulary in the training data.

train\_labels\_cleaned = []

characters = set()

max\_len = 0

for label in train\_labels:

label = label.split(" ")[-1].strip()

for char in label:

characters.add(char)

max\_len = max(max\_len, len(label))

train\_labels\_cleaned.append(label)

characters = sorted(list(characters))

print("Maximum length: ", max\_len)

print("Vocab size: ", len(characters))

# Check some label samples.

train\_labels\_cleaned[:10]

Now we clean the validation and the test labels as well.

def clean\_labels(labels):

cleaned\_labels = []

for label in labels:

label = label.split(" ")[-1].strip()

cleaned\_labels.append(label)

return cleaned\_labels

validation\_labels\_cleaned = clean\_labels(validation\_labels)

test\_labels\_cleaned = clean\_labels(test\_labels)

### Building the character vocabulary

Keras provides different preprocessing layers to deal with different modalities of data.

[This guide](https://keras.io/guides/preprocessing\_layers/) provides a comprehensive introduction.

Our example involves preprocessing labels at the character

level. This means that if there are two labels, e.g. "cat" and "dog", then our character

vocabulary should be {a, c, d, g, o, t} (without any special tokens). We use the

[`StringLookup`](https://keras.io/api/layers/preprocessing\_layers/categorical/string\_lookup/)

layer for this purpose.

AUTOTUNE = tf.data.AUTOTUNE

# Mapping characters to integers.

char\_to\_num = StringLookup(vocabulary=list(characters), mask\_token=None)

# Mapping integers back to original characters.

num\_to\_char = StringLookup(

vocabulary=char\_to\_num.get\_vocabulary(), mask\_token=None, invert=True

)

### Resizing images without distortion

Instead of square images, many OCR models work with rectangular images. This will become

clearer in a moment when we will visualize a few samples from the dataset. While

aspect-unaware resizing square images does not introduce a significant amount of

distortion this is not the case for rectangular images. But resizing images to a uniform

size is a requirement for mini-batching. So we need to perform our resizing such that

the following criteria are met:

\* Aspect ratio is preserved.

\* Content of the images is not affected.

def distortion\_free\_resize(image, img\_size):

w, h = img\_size

image = tf.image.resize(image, size=(h, w), preserve\_aspect\_ratio=True)

# Check tha amount of padding needed to be done.

pad\_height = h - tf.shape(image)[0]

pad\_width = w - tf.shape(image)[1]

# Only necessary if you want to do same amount of padding on both sides.

if pad\_height % 2 != 0:

height = pad\_height // 2

pad\_height\_top = height + 1

pad\_height\_bottom = height

else:

pad\_height\_top = pad\_height\_bottom = pad\_height // 2

if pad\_width % 2 != 0:

width = pad\_width // 2

pad\_width\_left = width + 1

pad\_width\_right = width

else:

pad\_width\_left = pad\_width\_right = pad\_width // 2

image = tf.pad(

image,

paddings=[

[pad\_height\_top, pad\_height\_bottom],

[pad\_width\_left, pad\_width\_right],

[0, 0],

],

)

image = tf.transpose(image, perm=[1, 0, 2])

image = tf.image.flip\_left\_right(image)

return image

If we just go with the plain resizing then the images would look like so:

![](https://i.imgur.com/eqq3s4N.png)

Notice how this resizing would have introduced unnecessary stretching.

### Putting the utilities together

batch\_size = 64

padding\_token = 99

image\_width = 128

image\_height = 32

def preprocess\_image(image\_path, img\_size=(image\_width, image\_height)):

image = tf.io.read\_file(image\_path)

image = tf.image.decode\_png(image, 1)

image = distortion\_free\_resize(image, img\_size)

image = tf.cast(image, tf.float32) / 255.0

return image

def vectorize\_label(label):

label = char\_to\_num(tf.strings.unicode\_split(label, input\_encoding="UTF-8"))

length = tf.shape(label)[0]

pad\_amount = max\_len - length

label = tf.pad(label, paddings=[[0, pad\_amount]], constant\_values=padding\_token)

return label

def process\_images\_labels(image\_path, label):

image = preprocess\_image(image\_path)

label = vectorize\_label(label)

return {"image": image, "label": label}

def prepare\_dataset(image\_paths, labels):

dataset = tf.data.Dataset.from\_tensor\_slices((image\_paths, labels)).map(

process\_images\_labels, num\_parallel\_calls=AUTOTUNE

)

return dataset.batch(batch\_size).cache().prefetch(AUTOTUNE)

## Prepare `tf.data.Dataset` objects

train\_ds = prepare\_dataset(train\_img\_paths, train\_labels\_cleaned)

validation\_ds = prepare\_dataset(validation\_img\_paths, validation\_labels\_cleaned)

test\_ds = prepare\_dataset(test\_img\_paths, test\_labels\_cleaned)

## Visualize a few samples

for data in train\_ds.take(1):

images, labels = data["image"], data["label"]

\_, ax = plt.subplots(4, 4, figsize=(15, 8))

for i in range(16):

img = images[i]

img = tf.image.flip\_left\_right(img)

img = tf.transpose(img, perm=[1, 0, 2])

img = (img \* 255.0).numpy().clip(0, 255).astype(np.uint8)

img = img[:, :, 0]

# Gather indices where label!= padding\_token.

label = labels[i]

indices = tf.gather(label, tf.where(tf.math.not\_equal(label, padding\_token)))

# Convert to string.

label = tf.strings.reduce\_join(num\_to\_char(indices))

label = label.numpy().decode("utf-8")

ax[i // 4, i % 4].imshow(img, cmap="gray")

ax[i // 4, i % 4].set\_title(label)

ax[i // 4, i % 4].axis("off")

plt.show()

You will notice that the content of original image is kept as faithful as possible and has

been padded accordingly.

## Model

Our model will use the CTC loss as an endpoint layer. For a detailed understanding of the

CTC loss, refer to [this post](https://distill.pub/2017/ctc/).

class CTCLayer(keras.layers.Layer):

def \_\_init\_\_(self, name=None):

super().\_\_init\_\_(name=name)

self.loss\_fn = keras.backend.ctc\_batch\_cost

def call(self, y\_true, y\_pred):

batch\_len = tf.cast(tf.shape(y\_true)[0], dtype="int64")

input\_length = tf.cast(tf.shape(y\_pred)[1], dtype="int64")

label\_length = tf.cast(tf.shape(y\_true)[1], dtype="int64")

input\_length = input\_length \* tf.ones(shape=(batch\_len, 1), dtype="int64")

label\_length = label\_length \* tf.ones(shape=(batch\_len, 1), dtype="int64")

loss = self.loss\_fn(y\_true, y\_pred, input\_length, label\_length)

self.add\_loss(loss)

# At test time, just return the computed predictions.

return y\_pred

def build\_model():

# Inputs to the model

input\_img = keras.Input(shape=(image\_width, image\_height, 1), name="image")

labels = keras.layers.Input(name="label", shape=(None,))

# First conv block.

x = keras.layers.Conv2D(

32,

(3, 3),

activation="relu",

kernel\_initializer="he\_normal",

padding="same",

name="Conv1",

)(input\_img)

x = keras.layers.Conv2D(

128,

(3, 3),

activation="relu",

kernel\_initializer="he\_normal",

padding="same",

name="Conv4",

)(x)

x = keras.layers.Conv2D(

256,

(3, 3),

activation="relu",

kernel\_initializer="he\_normal",

padding="same",

name="Conv6",

)(x)

x = keras.layers.MaxPooling2D((2, 2), name="pool1")(x)

#first max pooling layer

x = keras.layers.MaxPooling2D((2, 2), name="pool2")(x)

x = keras.layers.Conv2D(

1024,

(3, 3),

activation="relu",

kernel\_initializer="he\_normal",

padding="same",

name="Conv8",

)(x)

#first max pooling layer

x = keras.layers.Conv2D(

64,

(3, 3),

activation="relu",

kernel\_initializer="he\_normal",

padding="same",

name="Conv7",

)(x)

# x=keras.layers.MaxPooling2D((2, 2), name="pool3")(x)

# We have used two max pool with pool size and strides 2.

# Hence, downsampled feature maps are 4x smaller. The number of

# filters in the last layer is 64. Reshape accordingly before

# passing the output to the RNN part of the model.

new\_shape = ((image\_width // 4), (image\_height // 4) \* 64)

x = keras.layers.Reshape(target\_shape=new\_shape, name="reshape")(x)

x = keras.layers.Dense(64, activation="relu", name="dense1")(x)

x = keras.layers.Dropout(0.2)(x)

# RNNs.

x = keras.layers.Bidirectional(

keras.layers.LSTM(1024, return\_sequences=True, dropout=0.2)

)(x)

x = keras.layers.Bidirectional(

keras.layers.LSTM(512, return\_sequences=True, dropout=0.25)

)(x)

x = keras.layers.Bidirectional(

keras.layers.LSTM(64, return\_sequences=True, dropout=0.25)

)(x)

# +2 is to account for the two special tokens introduced by the CTC loss.

# The recommendation comes here: https://git.io/J0eXP.

x = keras.layers.Dense(

len(char\_to\_num.get\_vocabulary()) + 2, activation="softmax", name="dense2"

)(x)

# Add CTC layer for calculating CTC loss at each step.

output = CTCLayer(name="ctc\_loss")(labels, x)

# Define the model.

model = keras.models.Model(

inputs=[input\_img, labels], outputs=output, name="handwriting\_recognizer"

)

# Optimizer.

opt = keras.optimizers.Adam()

# Compile the model and return.

model.compile(optimizer=opt)

return model

# Get the model.

model = build\_model()

model.summary()

## Evaluation metric

[Edit Distance](https://en.wikipedia.org/wiki/Edit\_distance)

is the most widely used metric for evaluating OCR models. In this section, we will

implement it and use it as a callback to monitor our model.

We first segregate the validation images and their labels for convenience.

validation\_images = []

validation\_labels = []

for batch in validation\_ds:

validation\_images.append(batch["image"])

validation\_labels.append(batch["label"])

Now, we create a callback to monitor the edit distances.

def calculate\_edit\_distance(labels, predictions):

# Get a single batch and convert its labels to sparse tensors.

saprse\_labels = tf.cast(tf.sparse.from\_dense(labels), dtype=tf.int64)

# Make predictions and convert them to sparse tensors.

input\_len = np.ones(predictions.shape[0]) \* predictions.shape[1]

predictions\_decoded = keras.backend.ctc\_decode(

predictions, input\_length=input\_len, greedy=True

)[0][0][:, :max\_len]

sparse\_predictions = tf.cast(

tf.sparse.from\_dense(predictions\_decoded), dtype=tf.int64

)

# Compute individual edit distances and average them out.

edit\_distances = tf.edit\_distance(

sparse\_predictions, saprse\_labels, normalize=False

)

return tf.reduce\_mean(edit\_distances)

class EditDistanceCallback(keras.callbacks.Callback):

def \_\_init\_\_(self, pred\_model):

super().\_\_init\_\_()

self.prediction\_model = pred\_model

def on\_epoch\_end(self, epoch, logs=None):

edit\_distances = []

for i in range(len(validation\_images)):

labels = validation\_labels[i]

predictions = self.prediction\_model.predict(validation\_images[i])

edit\_distances.append(calculate\_edit\_distance(labels, predictions).numpy())

print(

f"Mean edit distance for epoch {epoch + 1}: {np.mean(edit\_distances):.4f}"

)

## Training

Now we are ready to kick off model training.

epochs = 50 # To get good results this should be at least 50.

model = build\_model()

prediction\_model = keras.models.Model(

model.get\_layer(name="image").input, model.get\_layer(name="dense2").output

)

edit\_distance\_callback = EditDistanceCallback(prediction\_model)

# Train the model.

history = model.fit(

train\_ds,

validation\_data=validation\_ds,

epochs=epochs,

callbacks=[edit\_distance\_callback],

)

## Inference

# A utility function to decode the output of the network.

def decode\_batch\_predictions(pred):

input\_len = np.ones(pred.shape[0]) \* pred.shape[1]

# Use greedy search. For complex tasks, you can use beam search.

results = keras.backend.ctc\_decode(pred, input\_length=input\_len, greedy=True)[0][0][

:, :max\_len

]

# Iterate over the results and get back the text.

output\_text = []

for res in results:

res = tf.gather(res, tf.where(tf.math.not\_equal(res, -1)))

res = tf.strings.reduce\_join(num\_to\_char(res)).numpy().decode("utf-8")

output\_text.append(res)

return output\_text

# Let's check results on some test samples.

for batch in test\_ds.take(1):

# print(batch["label"])

batch\_images,batch\_labels = batch["image"],batch["label"]

print("len is : ",len(batch))

print("batch img shape: ",batch\_images[1].shape)

# print(batch)

\_, ax = plt.subplots(4, 4, figsize=(15, 8))

preds = prediction\_model.predict(batch\_images)

pred\_texts = decode\_batch\_predictions(preds)

for i in range(16):

img = batch\_images[i]

print(img.shape)

img = tf.image.flip\_left\_right(img)

img = tf.transpose(img, perm=[1, 0, 2])

img = (img \* 255.0).numpy().clip(0, 255).astype(np.uint8)

img = img[:, :, 0]

label = batch\_labels[i]

indices = tf.gather(label, tf.where(tf.math.not\_equal(label, padding\_token)))

# Convert to string.

label = tf.strings.reduce\_join(num\_to\_char(indices))

label = label.numpy().decode("utf-8")

title = f"Prediction: {pred\_texts[i]} \n\n Original : {label}"

ax[i // 4, i % 4].imshow(img, cmap="gray")

ax[i // 4, i % 4].set\_title(title)

ax[i // 4, i % 4].axis("off")

plt.show()

# Precision

total\_count = 0

correct\_count = 0

for batch in test\_ds.take(75):

batch\_images,batch\_labels = batch["image"],batch["label"]

preds = prediction\_model.predict(batch\_images)

pred\_texts = decode\_batch\_predictions(preds)

for i in range(64):

label = batch\_labels[i]

indices = tf.gather(label, tf.where(tf.math.not\_equal(label, padding\_token)))

# Convert to string.

label = tf.strings.reduce\_join(num\_to\_char(indices))

label = label.numpy().decode("utf-8")

title = f"Prediction: {pred\_texts[i]} \n\n Original : {label}"

total\_count = total\_count+1

if(label == pred\_texts[i]):

correct\_count = correct\_count+1

print("Total count : ",total\_count)

print("Correct count : ",correct\_count)

percentage = (correct\_count/total\_count)\*100

print(f"Percentage : {percentage}% .")