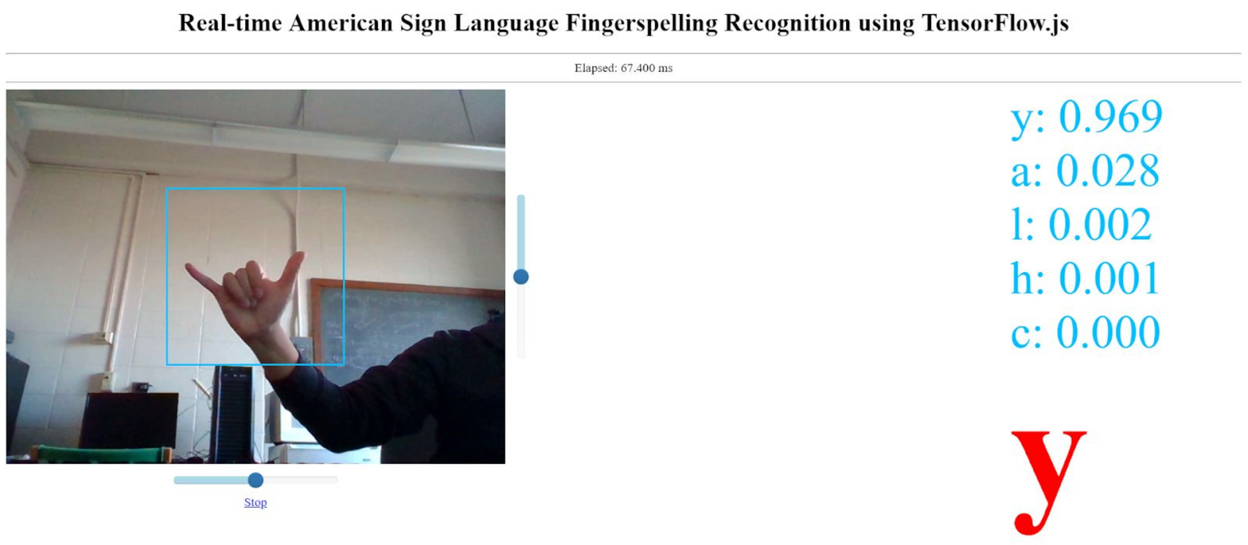
Real-time American Sign Language (ASL) fingerspelling recognition using TensorFlow.js

By Wei Cao (GitHub)

In this post we will be using Google Colab to train a model that can recognize an American Sign Language (ASL) fingerspelled alphabet, then deploy it in the browser using TensorFlow.js. You can find the demo and the code on GitHub.


Real-time American Sign Language Fingerspelling Recognition using TensorFlow.js
Elapsed: 67.400 ms
y: 0.969
a: 0.028
l: 0.002
h: 0.001
c: 0.000
y
Stop

The fast and lightweight neural network, MobileNetV2, will be our classification model. The model will be trained on a dataset of 3000 images sampled from the ASL FingerSpelling Dataset from the University of Surrey and the Massey University Gesture Dataset 2012 (I learned about both datasets via a great article from Stanford's CS231n).

First, we make sure that the Colab runtime type is Python 3 with GPU. Then we download the dataset to Colab.

```
!wget https://github.com/CoserU/coseru.github.io/raw/master/ASLtest/dataset_ASL.zip
!unzip -q dataset_ASL.zip
```

Preparing the Data

Import Tensorflow, Keras layers and other packages.

```
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense, Conv2D, DepthwiseConv2D,
Flatten, BatchNormalization, ReLU, AveragePooling2D, Add

import os
import numpy as np
from matplotlib import pyplot as plt
%matplotlib inline
```

Now let's take a look at the images. They are of different sizes and different aspect ratios.

```
# Training directory and validation directory
train_path = 'dataset_ASL/train/'
valid_path = 'dataset_ASL/validation/'

def show_image_from_path(data_path, alphabet):
    folder_path = os.path.join(data_path, alphabet)
    img_files = os.listdir(folder_path)
    img_file = img_files[np.random.choice(len(img_files), 1)[0]]
    img = plt.imread(os.path.join(folder_path, img_file))
    plt.imshow(img)

# Show a image randomly picked with label 'a'
show_image_from_path(train_path, 'a')
```

There are 24 fingerspelled alphabets in the dataset. We need to convert the alphabetical labels to integer labels for model training.

```
# Alphabet to integer
alphabet2int = {}
for i, char in enumerate('abcdefghiklmnopqrstuvwxy'):
    alphabet2int[char] = i
print(alphabet2int)

# Integer to alphabet
int2alphabet = {val: key for key, val in alphabet2int.items()}
print(int2alphabet)
```

Here, we will use the tf.data API to preprocess the images and build the training/validation datasets. First we get the paths for all images and the corresponding integer labels.

```python
# Get the image paths and the corresponding integer labels
def get_paths_labels(data_path):
    paths = []
    labels = []
    for class_folder in os.listdir(data_path):
        folder_path = os.path.join(data_path, class_folder)
        label = alphabet2int[class_folder]
        for img_path in os.listdir(folder_path):
            paths.append(os.path.join(folder_path, img_path))
            labels.append(label)
    return np.array(paths), np.array(labels)

train_paths, train_labels = get_paths_labels(train_path)
valid_paths, valid_labels = get_paths_labels(valid_path)
```

Then we randomly shuffle the training paths/labels and validation paths/labels.

```python
# Shuffle the paths/labels
p = np.random.permutation(len(train_labels))
train_paths, train_labels = train_paths[p], train_labels[p]

p = np.random.permutation(len(valid_labels))
valid_paths, valid_labels = valid_paths[p], valid_labels[p]

n_train, n_valid = len(train_labels), len(valid_labels)
print('The size of the training set is {}'.format(n_train)) # 2400 images
print('The size of the validation set is {}'.format(n_valid)) # 600 images
```

Here we set the size of the input image as 224*224 for MobileNetV2. For simplicity, we assume the image mean is 0.5 for preprocessing and use 32 as the batch size for both training and validation.

```python
image_size = 224
RGB_mean = np.array([0.5, 0.5, 0.5])
BATCH_SIZE = 32
```

Training Dataset

Preprocessing steps for training:

- Decode the png-encoded image.

- Resize the image to 256*256 with padding to keep the same aspect ratio.
- Randomly crop the image to 224*224. The camera sometimes cannot capture the whole gesture, so this data augmentation method lets our model to learn to predict without a complete gesture.
- Randomly flip the image horizontally to learn both right-handed and left-handed gestures.
- Normalize the image tensor.

```
# Load and preprocess the images for the training dataset
def preprocess_train_image(image):
    image = tf.image.decode_png(image, channels=3)
    image = tf.image.resize_image_with_pad(image, 256, 256)
    image = tf.image.random_crop(image, [image_size, image_size, 3])
    image = tf.image.random_flip_left_right(image)
    image /= 255.0
    image -= RGB_mean
    return image

def load_and_preprocess_train_image(path):
    image = tf.read_file(path)
    return preprocess_train_image(image)

train_path_ds = tf.data.Dataset.from_tensor_slices(train_paths)
train_image_ds = train_path_ds.map(load_and_preprocess_train_image)
train_label_ds = tf.data.Dataset.from_tensor_slices(train_labels)

# Zip the images and labels together
train_image_label_ds = tf.data.Dataset.zip((train_image_ds,
train_label_ds))
```

Then we set the buffer size and the batch size for training.

```
# Prepare for training
tr_ds = train_image_label_ds.repeat()
tr_ds = tr_ds.shuffle(buffer_size=1000)
tr_ds = tr_ds.batch(BATCH_SIZE)
tr_ds = tr_ds.prefetch(1)
```

Validation Dataset

Preprocessing steps for validation:

- Decode the png-encoded image.
- Resize the image to 224*224 with padding to keep the same aspect ratio.
- Normalize the image tensor.

```
# Load and preprocess the images for the validation dataset
def preprocess_test_image(image):
    image = tf.image.decode_png(image, channels=3)
    image = tf.image.resize_image_with_pad(image, image_size, image_size)
    image /= 255.0
    image -= RGB_mean
    return image

def load_and_preprocess_test_image(path):
    image = tf.read_file(path)
    return preprocess_test_image(image)

valid_path_ds = tf.data.Dataset.from_tensor_slices(valid_paths)
valid_image_ds = valid_path_ds.map(load_and_preprocess_test_image)
valid_label_ds = tf.data.Dataset.from_tensor_slices(valid_labels)

# Zip the images and labels together
valid_image_label_ds = tf.data.Dataset.zip((valid_image_ds,
valid_label_ds))
```

Then we set the buffer size and the batch size for validation.

```
# Prepare for validation
va_ds = valid_image_label_ds.repeat()
va_ds = va_ds.shuffle(buffer_size=1000)
va_ds = va_ds.batch(BATCH_SIZE)
va_ds = va_ds.prefetch(1)
```

MobileNetV2

In this tutorial, we will use Keras functional API to implement MobileNetV2 as our model (just for fun!) However, if you are interested in transfer learning, you can download a pre-trained MobileNetV2 directly from Keras applications instead of writing the model from scratch.

```
def ConvBNReLU(inputs, o, kernel_size=3, strides=1, dwise=False, bn=True,
relu=True):
    padding = 'same'
    if dwise:
        x = DepthwiseConv2D(kernel_size=kernel_size, padding=padding,
strides=strides, use_bias=not bn)(inputs)
    else:
        x = Conv2D(o, kernel_size=kernel_size, padding=padding,
strides=strides, use_bias=not bn)(inputs)

    if bn:
        x = BatchNormalization()(x)

    if relu:
        x = ReLU()(x)

    return x


def BottleNeck(inputs, i, o, expansion, strides):
    add_residual = strides == 1 and i == o
    x = ConvBNReLU(inputs, i*expansion, kernel_size=1)
    x = ConvBNReLU(x, i*expansion, strides=strides, dwise=True)
    x = ConvBNReLU(x, o, kernel_size=1, relu=False)

    if add_residual:
        x = Add()([x, inputs])

    return x


def MobileNetV2(inputs, n_classes, width_multiplier=1.):
    bottleneck_params = [
        # t, c, n, s
        [1, 16, 1, 1],
        [6, 24, 2, 2],
        [6, 32, 3, 2],
        [6, 64, 4, 2],
        [6, 96, 3, 1],
        [6, 160, 3, 2],
        [6, 320, 1, 1],
    ]

    input_channel = int(32*width_multiplier)
    for param in bottleneck_params:
        param[1] = int(param[1]*width_multiplier)

    x = ConvBNReLU(inputs, input_channel, strides=2)

    for (t, c, n, s) in bottleneck_params:
        x = BottleNeck(x, input_channel, c, t, s)
        for _ in range(n-1):
            x = BottleNeck(x, c, c, t, 1)
        input_channel = c

    x = ConvBNReLU(x, int(1280*width_multiplier), kernel_size=1)
    x = AveragePooling2D(pool_size=7, padding='same')(x)
    x = Flatten()(x)
    out = Dense(n_classes, activation='softmax')(x)

    return tf.keras.Model(inputs=inputs, outputs=out)
```

The model will take inputs of the shape [N, 224, 224, 3] and outputs probabilities of the shape [N, 24], where N is the batch size.

```
model = MobileNetV2(Input(shape=(image_size,image_size,3)),
len(alphabet2int))
model.summary()
```

Training the Model

We compile the model with Adam optimizer with 0.001 learning rate and 'sparse_catergorical_crossentropy' loss. The model will be saved after every epoch via tf.keras.callbacks.ModelCheckpoint.

```
# Compile the model with Adam optimizer
model.compile(optimizer=tf.keras.optimizers.Adam(lr=0.001),
loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Directory where the checkpoints will be saved
checkpoint_dir = './training_checkpoints'
!mkdir training_checkpoints

# Name of the checkpoint files
checkpoint_prefix = os.path.join(checkpoint_dir,
"{epoch:02d}-{val_loss:.3f}-{val_acc:.3f}.hdf5")
```

```
checkpoint_callback=tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_
prefix)
```

Let’s say we would like to train the model for 30 epochs. Each epoch takes around 30 seconds with GPU on Colab, so 30 epochs only take 15 to 20 minutes in total. We should notice that the batch size is 32 here.

```
epochs = 30
steps_per_epoch = n_train // BATCH_SIZE if n_train % BATCH_SIZE == 0 else
n_train // BATCH_SIZE + 1 # 75 steps
validation_steps = n_valid // BATCH_SIZE if n_valid % BATCH_SIZE == 0 else
n_valid // BATCH_SIZE + 1 # 19 steps
```

Below are the codes for model training using Keras Model fit method.

```
model_histroy = model.fit(
    tr_ds.make_one_shot_iterator(),
    epochs=epochs,
    steps_per_epoch=steps_per_epoch,
    validation_data=va_ds.make_one_shot_iterator(),
    validation_steps=validation_steps,
    callbacks=[checkpoint_callback],
)
```

After only 30-epoch training, the model can achieve about 60% validation accuracy. You are encouraged to train the model for more epochs. The model in the demo achieved about 80% validation accuracy after 100-epoch training, and it took less than an hour!

Convert the Model

In this section, we will learn how to install TensorFLow.js and how to save the Keras model in TensorFlow.js Layers format for web application.

First, we should restart the Colab runtime (ctrl+"M"+"."). Then we install Tensorflow.js on Colab using pip.

```
!pip install tensorflowjs
```

Next, we run the codes in ‘Preparing the Data’ and ‘MobileNetV2’ sections again. Then we take a look at the training checkpoints.

```
!ls training_checkpoints/
```

We can pick one of the checkpoints and load its weights. Here we pick the one with the best validation accuracy (the last float number in the file name):

```
model.load_weights('training_checkpoints/28-1.395-0.770.hdf5')
```

The following codes show how to save the metadata and the Keras model in TensorFlow.js Layers format in the model_js folder.

```
import json
import tensorflowjs as tfjs
MODEL_DIR = 'model_js'
if not os.path.exists(MODEL_DIR):
    os.mkdir(MODEL_DIR)

metadata = {
    'alphabet2int': alphabet2int,
    'int2alphabet': int2alphabet,
    'image_size': image_size,
    'RGB_mean': list(RGB_mean)
}

metadata_json_path = os.path.join(MODEL_DIR, 'metadata.json')
json.dump(metadata, open(metadata_json_path, 'wt'))
tfjs.converters.save_keras_model(model, MODEL_DIR)

!ls model_js
```

The model_js folder contains 5 files: (1) model.json (model architecture); (2) 3 'group1-shardXof3' files (model weights); (3) metadata.json (metadata). Then we compress the model_js folder and download it to the local file system.

```
!zip -r model_js.zip model_js

from google.colab import files
downloaded = files.download('model_js.zip')
```

Inference on the Browser

In this section, we will implement a web application for real-time ASL fingerspelling recognition based on the model we just trained. Every second it takes the frame of the camera stream as input and predicts the label of the gesture. The corresponding html, css and javascript files (code) have already been implemented for you, so you can download and use them to serve your own model. Here we focus on the TensorFlow.js codes for preprocessing and predicting.

Preprocessing

After capturing the gesture image from camera stream, we preprocess the image as follows:

- Create a tf.Tensor from the image.
- Cast the tensor to float32.
- Extract a sub-tensor of size 224*224*3
- Normalize the tensor.
- Create a batch that only contains this image tensor.

```
var inputImg = tf.browser.fromPixels(frame);
inputImg = inputImg.asType('float32');
inputImg = inputImg.slice([beginIndex1, beginIndex2, 0], [224, 224, 3]);
inputImg = inputImg.div(255.0);
inputImg = inputImg.sub(RGB_mean);
inputImg = inputImg.expandDims(0);
```

The above codes allows us to capture different regions of size 224*224*3 from the camera frame as the model input (the blue square area in the demo). However, you are welcome to try other sorts of input, e.g., resizing the whole frame to a 224*224*3 tensor as the model input.

Predicting

Below are the codes for predicting the top 5 labels of the input image.

```
const predictOut = model.predict(inputImg);
const result = tf.topk(tf.squeeze(predictOut), 5);
const indices = result['indices'].dataSync();
const probs = result['values'].dataSync();
var labels = []; // Save the top 5 labels
var i;
for (i = 0; i < indices.length; i++) {
    labels.push(int2alphabet[indices[i]]);
}
```

Dispose the tensors from memory.

```
predictOut.dispose();
result['indices'].dispose();
result['values'].dispose();
```

We can predict the top 5 labels and the their probabilities (probs) for the camera frame every second. It's also reasonable to highlight the top 1 label if its probability is greater than the threshold (70% in the demo).

Serving the Model on Github

The size of our MobileNetV2 model saved in Tensorflow.js format is less than 10 MB, so it's definitely okay for us to serve it on GitHub Pages, and it is quite simple! First, create a new repository named username.github.io, where username is your GitHub account name. Next, upload the model_js folder (unzipped) as well as index.html, index.css, index.js files to this repository. Now you can test your ASL fingerspelling recognizer by visiting https://username.github.io on the browser!

This recognizer usually works better with adequate lightening and simple background (e.g., a white wall), and it can be improved by adding other data augmentation methods like random brightness/saturation/rotation. You may check out another ASL fingerspelling recognizer trained on the whole dataset via this link.