Training and deploying a deep learning model on web

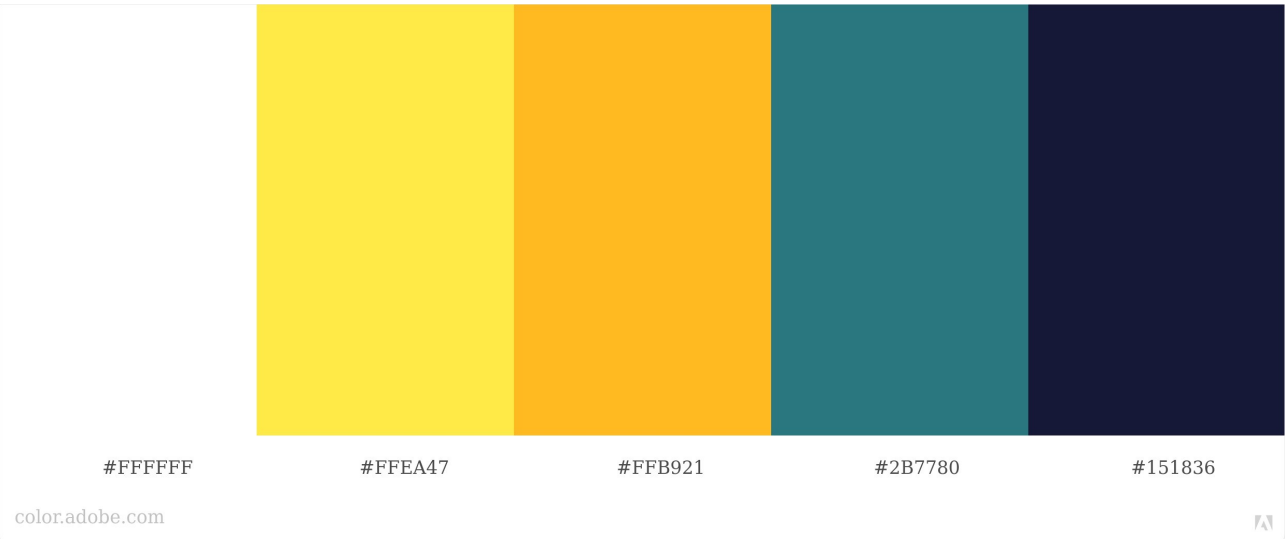
This document is a step by step guide to deploy a trained deep learning model on github pages. For reference, training a GAN to generate color palettes is used as a example.

Prerequisites

Before starting it is assumed that the person has knowledge of working in python environment, deep learning theory, model training using Keras/Tensorflow, jupyter notebook and machine learning data manipulation libraries including numpy, pandas, matplotlib, opencv and some background theory of image processing.

Data acquisition

For training a GAN which will generate color palettes, a dataset of available color palettes is required. Till date there is no curated dataset available for this purpose. Therefore as a first step, data is collected. For this purpose, Adobe color is used for collecting color palettes. The color palettes available at Adobe color website are designed by users around the world. The color palettes available on this website are in 5-color pattern. Users can download them by creating an account on Adobe color website. By sorting via trending, a user can get 4526 different palettes by removing duplicates. Though for real life purposes, this dataset is not enough to train a GAN but for educational purposes one can apply data synthesis techniques to get more data points (image processing knowledge required).



Once collected, the images of color palettes can be converted into a pandas dataframe. This is useful for reducing space required for dataset and for model training as providing digits instead of images is many times less resource intensive. One image of color palette is converted into one data point in dataframe. Each row in dataframe contains one data point having 15 columns with each column representing either red, green or blue value for 1 color in 5-color palette image. To acquire RGB

values of each color, K-nearest-neighbor method can be used. The code for getting RGB values using K-nearest-neighbor is given below:

```
def paletts(image):
    red, green, blue = [], [], []
    for line in image:
        for pixel in line:
            r, g, b = pixel
            red.append(r)
            green.append(g)
            blue.append(b)

    df = pd.DataFrame({
        'red': red,
        'green': green,
        'blue': blue})

    df['standardized_red'] = cluster.vq.whiten(df['red'])
    df['standardized_green'] = cluster.vq.whiten(df['green'])
    df['standardized_blue'] = cluster.vq.whiten(df['blue'])

    color_palette, distortion = cluster.vq.kmeans(df[['standardized_red',
'standardized_green',
                                                      'standardized_blue']],5)
    colors = []
    red_std, green_std, blue_std = df[['red', 'green', 'blue']].std()
    for color in color_palette:
        scaled_red, scaled_green, scaled_blue = color
        colors.append((
            math.ceil(scaled_red * red_std) ,
            math.ceil(scaled_green * green_std) ,
            math.ceil(scaled_blue * blue_std)
        ))
    return colors
```

In the above code, an image is provided to the function and it returns series of tuples with each tuple containing red, green and blue values of each color (each element of series). Output for a random image is given below:

```
[(56, 25, 21), (126, 120, 44), (34, 186, 253), (73, 234, 255), (255, 255, 255)]
```

Iterating through all images and saving them in pandas dataframe gives a dataset with each data point in a row containing 15 different values.

	red-0	green-0	blue-0	red-1	green-1	blue-1	red-2	green-2	blue-2	red-3	
count	90520.000000	90520.000000	90520.000000	90520.000000	90520.000000	90520.000000	90520.000000	90520.000000	90520.000000	90520.000000	90520.
mean	109.212970	98.540234	96.544134	109.212970	98.540234	96.544134	109.212970	98.540234	96.544134	109.212970	98.
std	71.133358	49.897345	49.459821	71.133358	49.897345	49.459821	71.133358	49.897345	49.459821	71.133358	49.
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.
25%	53.000000	61.000000	60.000000	53.000000	61.000000	60.000000	53.000000	61.000000	60.000000	53.000000	61.
50%	93.000000	92.000000	90.000000	93.000000	92.000000	90.000000	93.000000	92.000000	90.000000	93.000000	92.
75%	154.000000	129.000000	126.000000	154.000000	129.000000	126.000000	154.000000	129.000000	126.000000	154.000000	129.
max	255.000000	255.000000	255.000000	255.000000	255.000000	255.000000	255.000000	255.000000	255.000000	255.000000	255.

Figure 1: Dataset

For synthesizing additional data points from the image dataset, contrast and brightness of an image can be changed to get a new data point. In this example 3 additional data points are created for each image by changing brightness and contrast of that image. Appending original and synthesized data points, total 18105 data points are acquired. As the values for each RGB point varies from 0 to 255, the whole dataframe is normalized by dividing with 255. This gives data points a range of values ranging from 0 to 1. This data normalization helps in training the model and helps in avoiding exploding gradients problem.

Model training

As each data point in dataset contains 15 different values, the input for GAN is 15 units. The architecture of generator model is given below:

- 5 unit Input layer
- 100 unit Hidden layer
- Batch Normalization
- 200 unit Hidden layer
- Batch Normalization
- 15 unit Output layer

The architecture of discriminator model is given below:

- 15 unit Input layer
- 100 unit Hidden layer
- Batch Normalization
- 200 unit Hidden layer
- Batch Normalization
- 1 unit Output layer

```
Model: "functional_13"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_5 (InputLayer)         [(None, 5)]               0
_________________________________________________________________
dense_16 (Dense)             (None, 100)               600
_________________________________________________________________
batch_normalization_12 (Batc (None, 100)               400
_________________________________________________________________
dense_17 (Dense)             (None, 200)               20200
_________________________________________________________________
batch_normalization_13 (Batc (None, 200)               800
_________________________________________________________________
dense_18 (Dense)             (None, 15)                3015
=================================================================
Total params: 25,015
Trainable params: 24,415
Non-trainable params: 600
```

Figure 2: Generator

```
Model: "functional_15"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_6 (InputLayer)         [(None, 15)]              0
_________________________________________________________________
dense_19 (Dense)             (None, 100)               1600
_________________________________________________________________
batch_normalization_14 (Batc (None, 100)               400
_________________________________________________________________
dense_20 (Dense)             (None, 200)               20200
_________________________________________________________________
batch_normalization_15 (Batc (None, 200)               800
_________________________________________________________________
dense_21 (Dense)             (None, 1)                 201
=================================================================
Total params: 23,201
Trainable params: 22,601
Non-trainable params: 600
```

Figure 3: Discriminator

```
Model: "functional_17"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_5 (InputLayer)         [(None, 5)]               0
_________________________________________________________________
functional_13 (Functional)   (None, 15)                25015
_________________________________________________________________
functional_15 (Functional)   (None, 1)                 23201
=================================================================
Total params: 48,216
Trainable params: 24,415
Non-trainable params: 23,801
_________________________________________________________________
```

Figure 4: GAN

For model design more depth and complex architecture is avoided as created dataset is not large enough to justify complex design and comparatively shallow architecture helps in decreasing training time which helps in prototyping and hyperparameter tuning. During experimenting it is observed that with this dataset, increasing the depth of model results in mode collapse with default hyperparameter settings.

The compiling steps for GAN are given below:

- Compiling discriminator
- Compiling GAN with both generator and discriminator (discriminator weights are freezed/ not trained)

The learning rate of discriminator model is kept slightly higher than learning rate of GAN model. This is to give discriminator slightly upper hand as it pushes GAN to create more and more realistic data points.

The training steps for GAN are given below:

- Training discriminator model on real data points

Output label for real data points is selected to be 1.

- Generating fake data points using GAN

This is done by providing random generated noise to GAN as input and getting its output.

- Training discriminator model on fake data points

Output label for fake data points is selected to be 0.

- Training GAN

Training GAN is done by providing random generated points as input and setting the output label as 1. The given GAN is trained for 20,000 epochs with batch size of 32. Training the model for more epochs results in mode collapse as at 20,000 epoch mark, the accuracy of model is maximum and

starts to decrease slowly after training (using default hyperparameters). The code for training GAN is given below:

```
def train_disc(df, batch_size):
    batch_size = int(batch_size/2)
    x_real= []
    x_generated = []
    y_real = np.ones((batch_size,1))
    y_generated = np.zeros((batch_size,1))

    #Selecting random data points from dataset

    data_pt_idx = np.random.randint(0, df.shape[0], batch_size)
    for idx in data_pt_idx:
        x = df.loc[idx]
        x = np.array(x).astype('float32')
        x_real.append(x)
    x_real = np.array(x_real)

    #Training on real data
    disc_out_real = discriminator.train_on_batch(x_real, y_real)

    #Training on generated data
    for i in range(batch_size):
        x = np.random.random_sample((15))
        x_generated.append(x)
    x_generated = np.array(x_generated).astype('float32')
    x_generated = np.reshape(x_generated, (batch_size,15))

    disc_out_gen = discriminator.train_on_batch(x_generated, y_generated)

    return (disc_out_real, disc_out_gen)

def train_gan(batch_size):
    x_gan = []
    y = np.ones((batch_size,1))
    for i in range(batch_size):
```

```
        x = np.random.random_sample((5))
        x_gan.append(x)
    x_gan = np.array(x_gan)
    gan_out = gan.train_on_batch(x_gan,y)
    return gan_out


epoch = 20000
batch_size = 32
for ep in range(epoch):
    start = time.time()
    x,y = train_disc(df, batch_size)
    z = train_gan(batch_size)
    end = time.time()
    print('Epoch: {}, in {:.2f}s'.format(ep,(end-start)))
train_end = time.time()
total_time = train_end-train_start
total_time = total_time/3600
print('Training of {} epochs completed in {:.3f}hrs'.format(epoch,(total_time)))
```

Model inference

After training the GAN, output from the model can be visualized by providing random array of 5 values to GAN. The output of GAN is array of 15 values ranging from 0 to 1. These values multiplied by 255 gives the actual RGB values in the format in which dataset is saved. These values can then be plotted to get visual representation of the color palette generated by GAN. The code for model inference is given below:

```
def gan_out_to_colors(x):
    x = np.ceil(np.reshape(x,(15,)) *255)
    x = x.astype(int)
    ls = []
    rgb = x[0],x[1],x[2]
    ls.append(rgb)
    rgb = x[3],x[4],x[5]
    ls.append(rgb)
    rgb = x[6],x[7],x[8]
    ls.append(rgb)
    rgb = x[9],x[10],x[11]
    ls.append(rgb)
```

```
    rgb = x[12],x[13],x[14]
    ls.append(rgb)
    #ls.sort(key=lambda x: x[0]+x[1]+x[2])
    fig, (ax1, ax2, ax3, ax4, ax5) = plt.subplots(1, 5, figsize=(20,20))
    ax1.axis('off')
    ax2.axis('off')
    ax3.axis('off')
    ax4.axis('off')
    ax5.axis('off')
    ax1.imshow([[ls[0]]])
    ax2.imshow([[ls[1]]])
    ax3.imshow([[ls[2]]])
    ax4.imshow([[ls[3]]])
    ax5.imshow([[ls[4]]])
noise = np.random.random(5)
noise = np.array(noise)
noise = np.reshape(noise, (1,5))
x = generator.predict(noise)
gan_out_to_colors(x)
```

```
noise = np.random.random(5)
noise = np.array(noise)
noise = np.reshape(noise, (1,5))
x = generator.predict(noise)
gan_out_to_colors(x)
print(noise)
#print(x)
```

```
[(255, 58, 1), (255, 251, 255), (1, 255, 255), (229, 1, 255), (255, 254, 1)]
[[0.69891704 0.00863536 0.34524818 0.53296743 0.49830206]]
```

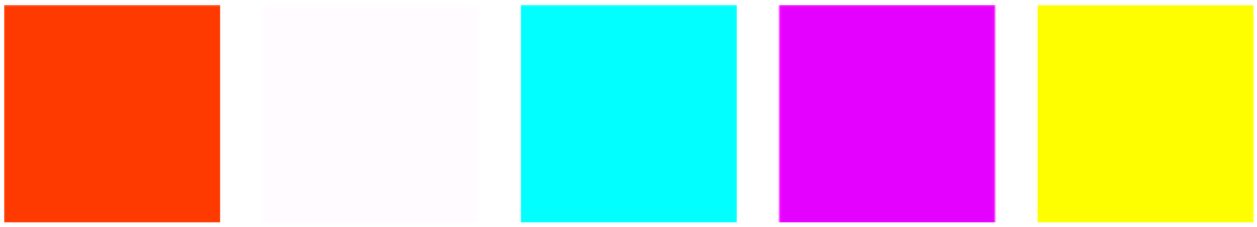


Figure 5: GAN output

Converting model into Tensorflow.js format

After training GAN, it can be saved into .h5 format. This saved model contains complete architecture and trained weights of GAN. At this point it should be noted that discriminator and generator should be separately saved. As the model used for inference is not the complete GAN but

the generator part of it therefore both generator and discriminator should be saved separately for future reference. Once saved into `.h5` file, generator model can be converted into Tensorflow.js model format. The code for converting model is given below:

```
import tensorflowjs as tfjs

tfjs.converters.save_keras_model(generator,'tfjs_model/')
```

The model should be saved in a separate folder as there are more than one file for model. After saving the model, navigate to the folder containing model files. Open `model.json` file in text editor and change `"class_name": "functional"` to `"class_name": "Model".` In Tensorflow.js version `'2.8.2'` or lower, functional models cannot be used for inference as-is. This is due to a bug in Tensorflow.js. Changing the `model.json` accordingly solves the problem.

Embedding model into HTML file

Any HTML file can use Javascript file for additional styling, features and backend processing. Loading a Tensorflow.js model in a browser using Javascript returns a promise which can be resolved by calling this function inside an async function. Example code is given below:

```
async function loadmodel() {

  model = await tf.loadLayersModel("assets/tfjs_model/model.json");

}
```

Calling `loadmodel()` function loads a trained model in `model` variable using `model,json` file. A user can then use this model for inference purpose.

The input for model is array of 5 random generated float numbers ranging from 0 to 1. A HTML button is used for this purpose. Once the user clicks on the button, an array of 5 randomly generated floats is created and is passed as input to the model. The model then generates an array of 15 floats which are converted to integers ranging from 0 to 255. Each of these values in array corresponds to either red, green or blue value for a color palette. With basic number manipulation, 5 different color palettes are received as output. These color palettes can be displayed on the HTML page using another Javascript function.

With additional styling following HTML page is designed:

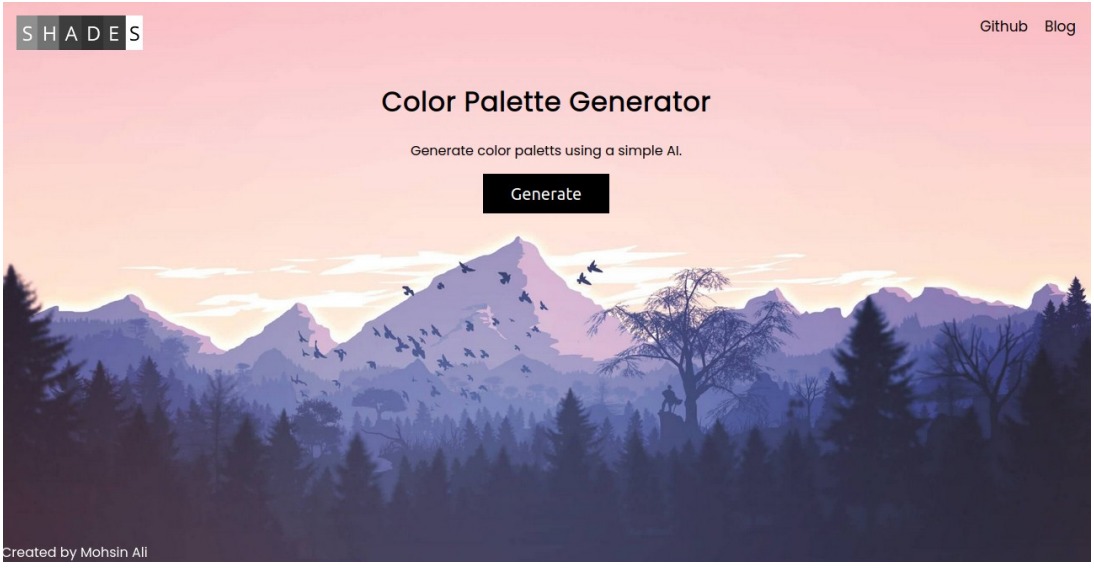


Figure 6: Landing page of Shades web app

HTML page after generating output:

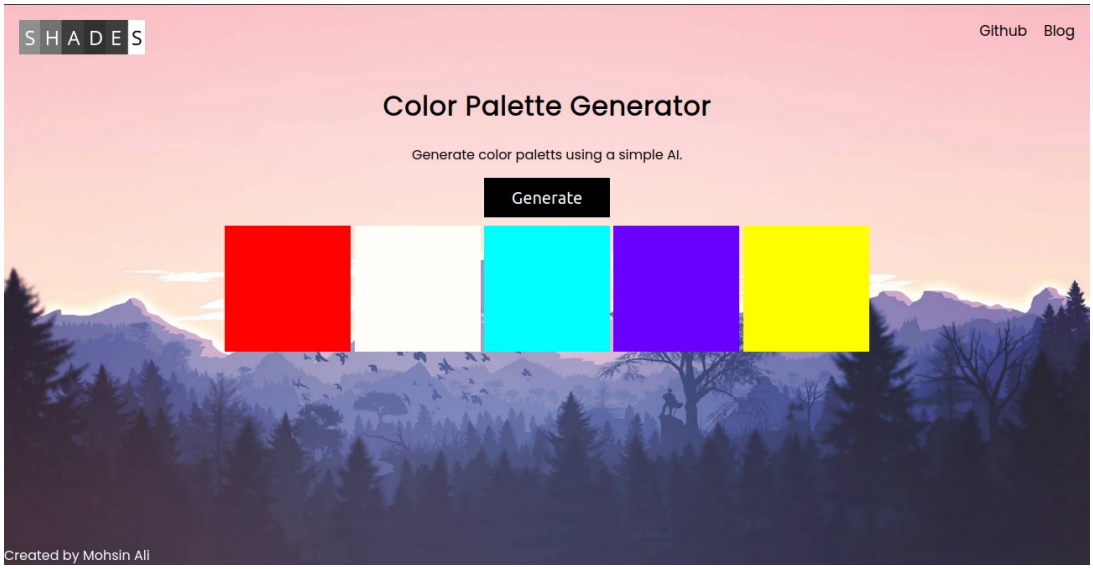


Figure 7: Output of Shades web app

Deploying app on github pages

After creating necessary HTML and Javascript files to generate color palettes in browser, these files can be published on internet using github pages. Github pages allows user to upload their static pages as separate repositories and it can host them without any additional user defined pipeline. Though there are many other platforms to deploy a machine learning/deep learning app on the web but if web app is simple and can work using Javascript then github pages is one of the best solutions available to deploy a machine learning/deep learning model. Other methods for model deployment include using PAAS (Platform As A Service) websites to host their models. A user can either select github pages or PAAS website but there are trade offs in selecting either.

PAAS websites

PAAS websites provide user with limited storage and compute capacity per app depending upon the packages they offer. One of the most user friendly PAAS is Heroku. A user can link his/her Heroku app directly with one of his/her repositories and the platform will take care of the rest for deployment of app. Though it is one of the more user friendly approach, a user might face some trouble in actually deploying process of app. Heroku uses container approach to deploy web app. The dependencies of the web app are mentioned in a separate file inside github repository and during initialization process Heroku installs all the packages mentioned in that file. If machine learning/deep learning model is created using Tensorflow, a user might see web app exceeding resource limits because Tensorflow python library takes more than 100MBs space by itself. Installing additional packages may leave actual web app with around 50-100MBs.

Github pages

Github can be used to host static web pages. User can convert his/her Tensorflow model into Tensorflow.js which can be run by browser without requiring additional dependencies. In this case when visiting the web app page, browser first loads the complete model into the client machine. If

the model size is large, the loading and inference process might take a lot of time. Therefore to make web app robust, one is left with small models which can be downloaded and loaded by browser in considerable time. This method has additional work overhead of converting Tensorflow model to Javascript code and developing front-end pipeline to make model work in a web page.