karan sindwani Follow
Sep 24, 2020 · 3 min read · Listen

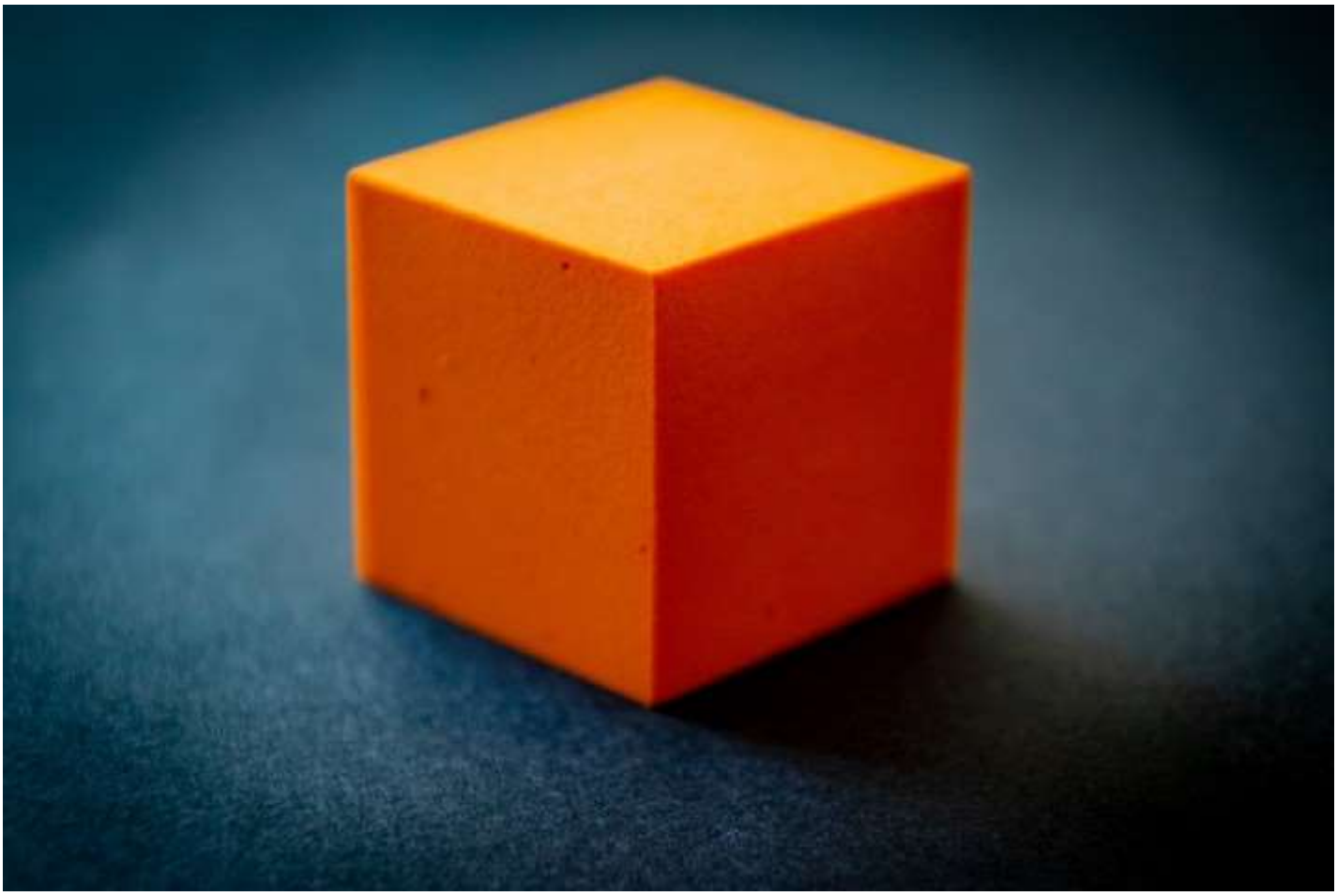

Photo by Magda Ehlers from Pexels

How to train an Image Classifier on TFRecord files

Improving the performance of pipelines using TFRecords

Introduction to TFRecords

TFRecords store a sequence of binary records, which are read linearly. They are useful format for storing data because they can be read efficiently. You can learn more about TFRecords here.

A binary file format for storage while working with voluminous data can have a significant impact on the performance of import pipeline and as a consequence on the training time of your model. Binary data takes up less space on disk, takes less time to copy and can be read much more efficiently from disk. This is especially true if your data is stored on spinning disks, due to the much lower read/write performance in comparison with SSDs.

Apart from performance, TFRecords are optimized for use with Tensorflow in multiple ways. Firstly, it makes it easy to combine multiple datasets and integrates seamlessly with the data import and preprocessing functionality provided by the library. Especially for datasets that are too large to be stored fully in memory this is an advantage as only the data that is required at the time (e.g. a

The next few sections focus on converting an image dataset to tfrecords, loading the data, model training and prediction.

Data Download

For this illustration, I shall be using a ubiquitous image dataset CIFAR-10.More details on the same can be found here. The code snippet below loads the CIFAR10 dataset directly from tensorflow.

```
import tensorflow as tf
import matplotlib.pyplot as plt
import wget
import tarfile
import os
from tensorflow.keras.datasets import cifar10

(x_train, y_train), (x_test, y_test) = cifar10.load_data()
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
```

download_data hosted with ❤ by GitHub view raw

Convert dataset to TFRecords

A TFRecord file stores the data as a sequence of binary strings. We need to specify the structure of our data before writing it to the file.

We shall be using tf.train.Example for the same. While writing an image to a TFRecord, we need the image itself and the corresponding label . In the code snippet below , we are defining two features namely image and label within the tf.train.Example.

The image feature stores the image array as bytes and the label feature stores the label as int. You may choose to store the label as string also or you could store additional information such as height, width and depth.

```
if not os.path.exists('./data/validation'):
    os.makedirs('./data/validation')

if not os.path.exists('./data/train'):
    os.makedirs('./data/train')

def write_tfrecords(x, y, filename):
    writer = tf.io.TFRecordWriter(filename)

    for image, label in zip(x, y):
        example = tf.train.Example(features=tf.train.Features(
            feature={
                'image': tf.train.Feature(bytes_list=tf.train.BytesList(value=[image.tobytes()])),
                'label': tf.train.Feature(int64_list=tf.train.Int64List(value=[label])),
            }))
        writer.write(example.SerializeToString())

write_tfrecords(x_test, y_test, './data/validation/validation.tfrecords')
```

Load the dataset

The process of reading TFRecords is straightforward:

1. Read the TFRecord using a tf.data.TFRecordDataset
2. Define the features you expect in the TFRecord by using tf.FixedLenFeature and tf.VarLenFeature, depending on what has been defined during the definition of tf.train.Example.
3. Parse one tf.train.Example (one file) a time using tf.parse_single_example.
4. Shuffle the dataset and extract by batch_size

```
def _parse_image_function(example_proto):
    features = tf.io.parse_single_example(example_proto, image_feature_description)
    image = tf.io.decode_raw(features['image'], tf.uint8)
    image.set_shape([3 * 32 * 32])
    image = tf.reshape(image, [32, 32, 3])

    label = tf.cast(features['label'], tf.int32)
    label = tf.one_hot(label, 10)

    return image, label
```

parse_image hosted with ❤ by GitHub

```
def read_dataset(epochs, batch_size, channel, channel_name):

    filenames = [os.path.join(channel, channel_name + '.tfrecords')]
    dataset = tf.data.TFRecordDataset(filenames)

    image_feature_description = {
        'image': tf.io.FixedLenFeature([], tf.string),
        'label': tf.io.FixedLenFeature([], tf.int64),
    }

    dataset = dataset.map(_parse_image_function, num_parallel_calls=10)
    dataset = dataset.prefetch(10)
    dataset = dataset.repeat(epochs)
    dataset = dataset.shuffle(buffer_size=10 * batch_size)
    dataset = dataset.batch(batch_size, drop_remainder=True)

    return dataset
```

read_dataset hosted with ❤ by GitHub view raw

Visualize input images

```
train_dataset = read_dataset(10, 100, 'data/train', 'train')
validation_dataset = read_dataset(10, 100, 'data/validation', 'validation')

def show_batch(image_batch, label_batch):
    plt.figure(figsize=(50, 50))
    for n in range(100):
        ax = plt.subplot(5, 5, n + 1)
        plt.imshow(image_batch[n] / 255.0)
        plt.axis("off")

image_batch, label_batch = next(iter(train_dataset))

show_batch(image_batch.numpy(), label_batch.numpy())
```

visualize_images hosted with ❤ by GitHub view raw

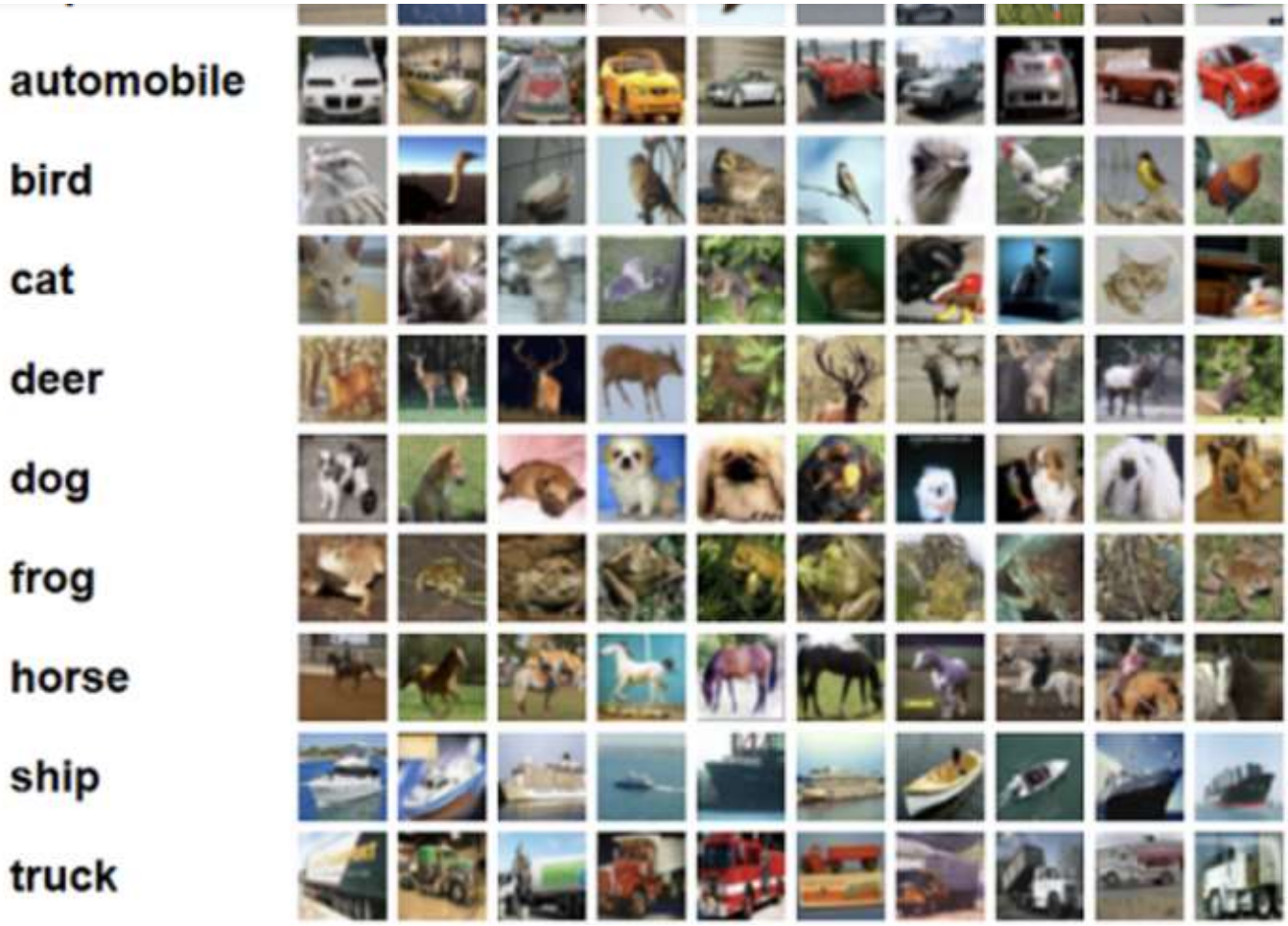

Photo from CIFAR10 dataset (https://www.cs.toronto.edu/~kriz/cifar.html)

Model Training and Prediction

Since we are using a relatively small and balanced dataset with 10 classes, we decided to write a custom model architecture similar to Lenet.

Our evaluation metric will be accuracy, but one may choose to have other metrics like average precision or area under roc curve

```
def keras_model_fn(learning_rate, optimizer):
    model = Sequential()
    model.add(Conv2D(32, (3, 3), padding='same', name='inputs', input_shape=(HEIGHT, WIDTH, DEPTH)))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(Conv2D(32, (3, 3)))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.2))

    model.add(Conv2D(64, (3, 3), padding='same'))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(Conv2D(64, (3, 3)))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
```

```
24      model.add(Conv2D(128, (3, 3)))
25      model.add(BatchNormalization())
26      model.add(Activation('relu'))
27      model.add(MaxPooling2D(pool_size=(2, 2)))
28      model.add(Dropout(0.4))
29
30      model.add(Flatten())
31      model.add(Dense(512, kernel_constraint=max_norm(2.)))
32      model.add(Activation('relu'))
33      model.add(Dropout(0.5))
34      model.add(Dense(NUM_CLASSES))
35      model.add(Activation('softmax'))
36
37      opt = SGD(learning_rate=learning_rate)
38
39
40      model.compile(loss='categorical_crossentropy',
41                    optimizer=opt,
42                    metrics=['accuracy'])
43      return model
```

model_def hosted with ❤ by GitHub view raw

The next step would be fitting the model , saving it and predicting for test set.

```
1   model.fit(x=train_dataset,
2                 epochs=epochs,
3                 batch_size=batch_size)
4
5   # Saving trained model
6   model.save(args.model_output + '/1')
7
8   validation_array = np.array(list(validation_dataset.unbatch().take(-1).as_numpy_iterator()))
9   test_x = np.stack(validation_array[:,0])
10  test_y = np.stack(validation_array[:,1])
11
```

```
17  # Evaluating model accuracy and logging it as a scalar for Tensorboard hyperparameter visualization.
18  accuracy = sklearn.metrics.accuracy_score(test_y_true, test_y_pred)
19  tf.summary.scalar(METRIC_ACCURACY, accuracy, step=1)
20  logging.info('Test accuracy:{}'.format(accuracy))
```

model_fit hosted with ❤ by GitHub view raw

Conclusion

Though the task of converting raw data to TFRecords may seem arduous but it is worth the effort. Few notable advantages are:

- TFRecords occupy less disk space than raw jpegs/pngs/csvs.
- TFRecords improves I/O performance as it takes less time to copy and can be read much more efficiently from disk
- Tensorflow provides a seemless integration of TFRecords with import pipelines
- It has the ability to store sequential data
- There is no change required in the model fit and definition statements while working with TFRecords
- As a consequence of improved I/O performance, TFRecords reduces the training time.