Week 2 Assignment: CIFAR-10 Autoencoder

For this week, you will create a convolutional autoencoder for the CIFAR10 dataset. You are free to choose the architecture of your autoencoder provided that the output image has the same dimensions as the input image.

After training, your model should meet loss and accuracy requirements when evaluated with the test dataset. You will then download the model and upload it in the classroom for grading.

Let's begin!

Important: *This colab notebook has read-only access so you won't be able to save your changes. If you want to save your work periodically, please click* `File -> Save a Copy in Drive` *to create a copy in your account, then work from there.*

Imports

[1]
```
try:
  # %tensorflow_version only exists in Colab.
  %tensorflow_version 2.x
except Exception:
  pass

import tensorflow as tf
import tensorflow_datasets as tfds

from keras.models import Sequential
```

Load and prepare the dataset

The CIFAR 10 dataset already has train and test splits and you can use those in this exercise. Here are the general steps:

- Load the train/test split from TFDS. Set `as_supervised` to `True` so it will be convenient to use the preprocessing function we provided.
- Normalize the pixel values to the range [0,1], then return `image, image` pairs for training instead of `image, label`. This is because you will check if the output image is successfully regenerated after going through your autoencoder.
- Shuffle and batch the train set. Batch the test set (no need to shuffle).

[3]
```
# preprocessing function
def map_image(image, label):
  image = tf.cast(image, dtype=tf.float32)
  image = image / 255.0

  return image, image # dataset label is not used. replaced with the same image input.

# parameters
BATCH_SIZE = 128
SHUFFLE_BUFFER_SIZE = 1024


### START CODE HERE (Replace instances of `None` with your code) ###

# use tfds.load() to fetch the 'train' split of CIFAR-10
train_dataset = tfds.load('cifar10', as_supervised=True, split="train")

# preprocess the dataset with the `map_image()` function above
train_dataset = train_dataset.map(map_image)

# shuffle and batch the dataset
train_dataset = train_dataset.shuffle(SHUFFLE_BUFFER_SIZE).batch(BATCH_SIZE)


# use tfds.load() to fetch the 'test' split of CIFAR-10
test_dataset = tfds.load('cifar10', as_supervised=True, split="test")

# preprocess the dataset with the `map_image()` function above
test_dataset = test_dataset.map(map_image)

# batch the dataset
test_dataset = test_dataset.batch(BATCH_SIZE)

### END CODE HERE ###
```

```
Downloading and preparing dataset cifar10/3.0.2 (download: 162.17 MiB, generated: 132.40 MiB, total: 294.58 MiB) to /root/tensorflow_datasets/cifar10/3.0.2...
Dl Completed...: 100% 1/1 [00:06<00:00, 4.53s/ url]
Dl Size...: 100% 162/162 [00:06<00:00, 46.15 MiB/s]
Extraction completed...: 100% 1/1 [00:06<00:00, 6.34s/ file]


Shuffling and writing examples to /root/tensorflow_datasets/cifar10/3.0.2.incompleteTQ8UMP/cifar10-train.tfrecord
100% 49999/50000 [00:00<00:00, 165091.50 examples/s]
Shuffling and writing examples to /root/tensorflow_datasets/cifar10/3.0.2.incompleteTQ8UMP/cifar10-test.tfrecord
100% 9999/10000 [00:00<00:00, 85531.31 examples/s]
Dataset cifar10 downloaded and prepared to /root/tensorflow_datasets/cifar10/3.0.2. Subsequent calls will reuse this data.
```

Build the Model

Create the autoencoder model. As shown in the lectures, you will want to downsample the image in the encoder layers then upsample it in the decoder path. Note that the output layer should be the same dimensions as the original image. Your input images will have the shape `(32, 32,`

`3)`. If you deviate from this, your model may not be recognized by the grader and may fail.

We included a few hints to use the Sequential API below but feel free to remove it and use the Functional API just like in the ungraded labs if you're more comfortable with it. Another reason to use the latter is if you want to visualize the encoder output. As shown in the ungraded labs, it will be easier to indicate multiple outputs with the Functional API. That is not required for this assignment though so you can just stack layers sequentially if you want a simpler solution.

[5]
```
# suggested layers to use. feel free to add or remove as you see fit.
from keras.layers import Conv2D, UpSampling2D, MaxPooling2D, Dropout

# use the Sequential API (you can remove if you want to use the Functional API)
model = Sequential()

### START CODE HERE ###
# use `model.add()` to add layers (if using the Sequential API)
model.add(Conv2D(64,input_shape=(32,32,3),kernel_size=(3,3),activation='relu',padding='same'))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Conv2D(128,kernel_size=(3,3),activation='relu',padding='same'))
model.add(MaxPooling2D(pool_size=(2,2)))


model.add(Conv2D(256,kernel_size=(3,3),activation='relu',padding='same'))

model.add(Conv2D(128,kernel_size=(3,3),activation='relu',padding='same'))
model.add(UpSampling2D(size=(2,2)))

model.add(Conv2D(64,kernel_size=(3,3),activation='relu',padding='same'))
model.add(UpSampling2D(size=(2,2)))


model.add(Conv2D(3,kernel_size=(3,3),activation='sigmoid',padding='same'))









### END CODE HERE ###

model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 32, 32, 64)        1792
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 16, 16, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 16, 16, 128)       73856
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 8, 8, 128)         0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 8, 8, 256)         295168
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 8, 8, 128)         295040
_________________________________________________________________
up_sampling2d (UpSampling2D) (None, 16, 16, 128)       0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 16, 16, 64)        73792
_________________________________________________________________
up_sampling2d_1 (UpSampling2 (None, 32, 32, 64)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 32, 32, 3)         1731
=================================================================
Total params: 741,379
Trainable params: 741,379
Non-trainable params: 0
_________________________________________________________________
```

Configure training parameters

We have already provided the optimizer, metrics, and loss in the code below.

[6]
```
# Please do not change the model.compile() parameters
model.compile(optimizer='adam', metrics=['accuracy'], loss='mean_squared_error')
```

Training

You can now use model.fit() to train your model. You will pass in the `train_dataset` and you are free to configure the other parameters. As with any training, you should see the loss generally going down and the accuracy going up with each epoch. If not, please revisit the previous sections to find possible bugs.

Note: If you get a `dataset length is infinite` error. Please check how you defined `train_dataset`. You might have included a method that repeats the dataset indefinitely.

[7]
```
# parameters (feel free to change this)
train_steps = len(train_dataset) // BATCH_SIZE
val_steps = len(test_dataset) // BATCH_SIZE

### START CODE HERE ###
model.fit(train_dataset,steps_per_epoch=train_steps,batch_size=128,validation_data=test_dataset,validation_steps=val_steps,epochs=100)
### END CODE HERE ###
```

```
Epoch 72/100
3/3 [==============================] - 0s 48ms/step - loss: 0.0060 - accuracy: 0.7835
Epoch 73/100
3/3 [==============================] - 0s 50ms/step - loss: 0.0056 - accuracy: 0.7813
Epoch 74/100
3/3 [==============================] - 0s 46ms/step - loss: 0.0054 - accuracy: 0.7751
Epoch 75/100
3/3 [==============================] - 0s 50ms/step - loss: 0.0060 - accuracy: 0.7724
```

```
Epoch 76/100
3/3 [==============================] - 0s 50ms/step - loss: 0.0061 - accuracy: 0.7673
Epoch 77/100
3/3 [==============================] - 0s 45ms/step - loss: 0.0055 - accuracy: 0.7905
Epoch 78/100
3/3 [==============================] - 0s 48ms/step - loss: 0.0055 - accuracy: 0.7864
Epoch 79/100
3/3 [==============================] - 0s 47ms/step - loss: 0.0057 - accuracy: 0.7800
Epoch 80/100
3/3 [==============================] - 0s 48ms/step - loss: 0.0056 - accuracy: 0.7788
Epoch 81/100
3/3 [==============================] - 0s 51ms/step - loss: 0.0054 - accuracy: 0.7875
Epoch 82/100
3/3 [==============================] - 0s 47ms/step - loss: 0.0052 - accuracy: 0.7871
Epoch 83/100
3/3 [==============================] - 0s 46ms/step - loss: 0.0056 - accuracy: 0.7834
Epoch 84/100
3/3 [==============================] - 0s 46ms/step - loss: 0.0062 - accuracy: 0.7844
Epoch 85/100
3/3 [==============================] - 0s 48ms/step - loss: 0.0055 - accuracy: 0.7690
Epoch 86/100
3/3 [==============================] - 0s 48ms/step - loss: 0.0056 - accuracy: 0.7738
Epoch 87/100
3/3 [==============================] - 0s 45ms/step - loss: 0.0051 - accuracy: 0.7854
Epoch 88/100
3/3 [==============================] - 0s 49ms/step - loss: 0.0052 - accuracy: 0.7930
Epoch 89/100
3/3 [==============================] - 0s 47ms/step - loss: 0.0051 - accuracy: 0.7916
Epoch 90/100
3/3 [==============================] - 0s 45ms/step - loss: 0.0055 - accuracy: 0.7961
Epoch 91/100
3/3 [==============================] - 0s 49ms/step - loss: 0.0058 - accuracy: 0.7862
Epoch 92/100
3/3 [==============================] - 0s 45ms/step - loss: 0.0050 - accuracy: 0.7967
Epoch 93/100
3/3 [==============================] - 0s 43ms/step - loss: 0.0056 - accuracy: 0.7764
Epoch 94/100
3/3 [==============================] - 0s 44ms/step - loss: 0.0057 - accuracy: 0.7758
Epoch 95/100
3/3 [==============================] - 0s 43ms/step - loss: 0.0049 - accuracy: 0.7900
Epoch 96/100
3/3 [==============================] - 0s 54ms/step - loss: 0.0049 - accuracy: 0.7941
Epoch 97/100
3/3 [==============================] - 0s 46ms/step - loss: 0.0052 - accuracy: 0.7766
Epoch 98/100
3/3 [==============================] - 0s 45ms/step - loss: 0.0050 - accuracy: 0.7801
Epoch 99/100
3/3 [==============================] - 0s 43ms/step - loss: 0.0049 - accuracy: 0.7936
Epoch 100/100
3/3 [==============================] - 0s 50ms/step - loss: 0.0051 - accuracy: 0.7837
<keras.callbacks.History at 0x7f5bf91a9610>
```

Model evaluation

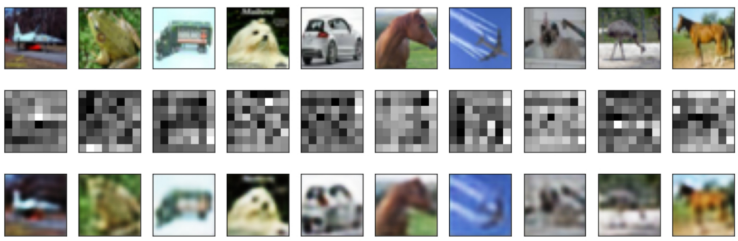
You can use this code to test your model locally before uploading to the grader. To pass, your model needs to satisfy these two requirements:

- loss must be less than 0.01
- accuracy must be greater than 0.6

```
[8]  result = model.evaluate(test_dataset, steps=10)
```

```
10/10 [==============================] - 0s 24ms/step - loss: 0.0048 - accuracy: 0.7848
```

If you did some visualization like in the ungraded labs, then you might see something like the gallery below. This part is not required.



Save your model

Once you are satisfied with the results, you can now save your model. Please download it from the Files window on the left and go back to the Submission portal in Coursera for grading.

```
model.save('mymodel.h5')
```

Congratulations on completing this week's assignment!

✓ 0s completed at 7:57 PM