Convolution Neural Networks

Shan-Hung Wu & DataLab
Fall 2023

In this lab, we introduce two datasets, **MNIST** and **CIFAR**, then we will talk about how to implement CNN models for these two datasets using tensorflow. The major difference between mnist and cifar is their size. Due to the limit of memory size and time issue, we offer a guide to illustrate typical **input pipeline** of tensorflow. Let's dive into tensorflow!

In [1]:
```
import warnings
import os
warnings.filterwarnings("ignore")
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

import tensorflow as tf
from tensorflow.keras import utils, datasets, layers, models
import matplotlib.pyplot as plt
import numpy as np

import matplotlib as mpl
import pickle
import csv
import itertools
from collections import defaultdict
import time
import pandas as pd
import math
from tqdm import tqdm
import dill
```

```
2023-10-28 19:03:51.509111: E tensorflow/stream_executor/cuda/cuda_blas.cc:2981] Unable
to register cuBLAS factory: Attempting to register factory for plugin cuBLAS when one ha
s already been registered
```

In [2]:
```
gpus = tf.config.experimental.list_physical_devices('GPU')
if gpus:
    try:
        # Currently, memory growth needs to be the same across GPUs
        for gpu in gpus:
            tf.config.experimental.set_memory_growth(gpu, True)
        tf.config.experimental.set_visible_devices(gpus[1], 'GPU')
        logical_gpus = tf.config.experimental.list_logical_devices('GPU')
        print(len(gpus), "Physical GPUs,", len(logical_gpus), "Logical GPUs")
    except RuntimeError as e:
        # Memory growth must be set before GPUs have been initialized
        print(e)
```

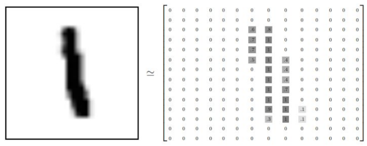
```
4 Physical GPUs, 1 Logical GPUs
```

In [3]:
```
import urllib.request
if not os.path.exists("lab11_1_lib.py"):
    urllib.request.urlretrieve("https://nthu-datalab.github.io/ml/labs/11-1_CNN/lab11_1

from lab11_1_lib import draw_timeline
```

MNIST

We start from a simple dataset. MNIST is a simple computer vision dataset. It consists of image of handwritten digits like:

It also includes label for each image, telling us which digit it is. For example, the label for the above image are 5, 0, 4, and 1. Each image is 28 pixels by 28 pixels. We can interpret this as a big array of numbers:



The MNIST data is hosted on Yann LeCun's website. We can directly import MNIST dataset from Tensorflow.

In [4]:
```
# Download and prepare the MNIST dataset
(train_image, train_label), (test_image, test_label) = datasets.mnist.load_data()

# Normalize pixel values to be between 0 and 1
train_image, test_image = train_image / 255.0, test_image / 255.0
print('shape of train_image:', train_image.shape)
print('shape of train_label:', train_label.shape)
```

```
shape of train_image: (60000, 28, 28)
shape of train_label: (60000,)
```

Softmax Regression on MNIST

Before jumping to *Convolutional Neural Network* model, we're going to start with a very simple model with a single layer and softmax regression.

We know that every image in MNIST is a handwritten digit between zero and nine. So there are only ten possible digits that a given image can be. We want to give the probability of the input image for being each digit. That is, input an image, the model outputs a ten-dimension vector.

This is a classic case where a softmax regression is a natural, simple model. If you want to assign probabilities to an object being one of several different things, softmax is the thing to do.

In [5]:
```
# flating the training data for dense layers
train_image_1 = train_image.reshape((60000, -1))
test_image_1 = test_image.reshape((10000, -1))
print(train_image_1.shape)
print(test_image_1.shape)
```

```
(60000, 784)
(10000, 784)
```

In [6]:
```
model_1 = models.Sequential()
model_1.add(layers.Dense(10, activation='softmax',input_shape=(784,)))
model_1.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 10)                7850
```

```
=================================================================
Total params: 7,850
Trainable params: 7,850
Non-trainable params: 0
_________________________________________________________________
```

In [7]:
```
# compile the model and train it for 5 epochs
model_1.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model_1.fit(train_image_1, train_label, epochs=5)
```

```
Epoch 1/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.4647 - accuracy: 0.87
91
Epoch 2/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3031 - accuracy: 0.91
56
Epoch 3/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.2830 - accuracy: 0.92
11
Epoch 4/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.2729 - accuracy: 0.92
35
Epoch 5/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.2668 - accuracy: 0.92
59
```

Out[7]:
```
<keras.callbacks.History at 0x7fd74031a490>
```

In [8]:
```
_, test_acc_1 = model_1.evaluate(test_image_1, test_label, verbose=0)
print('Testing Accuracy : %.4f'%test_acc_1)
```

```
Testing Accuracy : 0.9244
```

From the above result, we got about 92% accuracy for *Softmax Regression* on MNIST. In fact, it's not so good. This is because we're using a very simple model.

Multilayer Convolutional Network on MNIST

We're now jumping from a very simple model to something moderately sophisticated: a small *Convolutional Neural Network*. This will get us to over 99% accuracy, not state of the art, but respectable.

In [9]:
```
# reshaping the training data to 3 dimensions
train_image_2 = train_image.reshape((60000, 28, 28, 1))
test_image_2 = test_image.reshape((10000, 28, 28, 1))
print(train_image_2.shape)
print(test_image_2.shape)
```

```
(60000, 28, 28, 1)
(10000, 28, 28, 1)
```

Create the convolutional base

As input, a CNN takes tensors of shape (image_height, image_width, color_channels), ignoring the batch size. If you are new to color channels, MNIST has one (because the image are grayscale), whereas a color image has three (R,G,B). In this example, we will configure our CNN to process inputs of shape (28, 28, 1), which is the format of MNIST image. We do this by passing the argument **input_shape** to our first layer.

In [10]:
```
#The 6 lines of code below define the convolutional base using a common pattern: a stac
model_2 = models.Sequential()
model_2.add(layers.Conv2D(32, (3, 3), strides=(1,1), padding='valid', activation='relu'
model_2.add(layers.MaxPooling2D((2, 2)))
model_2.add(layers.Conv2D(64, (3, 3), strides=(1,1), padding='valid', activation='relu'
model_2.add(layers.MaxPooling2D((2, 2)))
model_2.add(layers.Conv2D(64, (3, 3), strides=(1,1), padding='valid', activation='relu'
```

Let's display the architecture of our model so far.

In [11]:
```
model_2.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320       
                                                                 
 max_pooling2d (MaxPooling2D  (None, 13, 13, 32)       0         
 )                                                               
                                                                 
 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496     
                                                                 
 max_pooling2d_1 (MaxPooling  (None, 5, 5, 64)         0         
 2D)                                                             
                                                                 
 conv2d_2 (Conv2D)           (None, 3, 3, 64)          36928     
                                                                 
=================================================================
Total params: 55,744
Trainable params: 55,744
Non-trainable params: 0
_________________________________________________________________
```

Above, you can see that the output of every Conv2D and MaxPooling2D layer is a 3D tensor of shape (height, width, channels). The width and height dimensions tend to shrink as we go deeper in the network. The number of output channels for each Conv2D layer is controlled by the first argument (e.g., 32 or 64). Typically, as the width and height shrink, we can afford (computationally) to add more output channels in each Conv2D layer.

Add Dense layers on top

To complete our model, we will feed the last output tensor from the convolutional base (of shape (3, 3, 64)) into one or more Dense layers to perform classification. Dense layers take vectors as input (which are 1D), while the current output is a 3D tensor. First, we will flatten (or unroll) the 3D output to 1D, then add one or more Dense layers on top. MNIST has 10 output classes, so we use a final Dense layer with 10 outputs and a softmax activation.

To reduce overfitting, we will apply *dropout* before the readout layer. The idea behind dropout is to train an ensemble of model instead of a single model. During training, we drop out neurons with probability p, i.e., the probability to keep is $1-p$. When a neuron is dropped, its output is set to zero. These dropped neurons do not contribute to the training phase in forward pass and backward pass. For each training phase, we train the network slightly different from the previous one. It's just like we train different networks in each training phrase. However, during testing phase, we **don't** drop any neuron, and thus, implement dropout is kind of like doing ensemble. Also, randomly drop units in training phase can prevent units from co-adapting too much. Thus, dropout is a powerful regularization techique to deal with *overfitting*.

In [12]:
```
model_2.add(layers.Flatten())
```

```
model_2.add(layers.Dense(64, activation='relu'))
model_2.add(layers.Dropout(0.5))
model_2.add(layers.Dense(10, activation='softmax'))
```

Here's the complete architecture of our model.

In [13]:
```
model_2.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2D  (None, 13, 13, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 5, 5, 64)         0
 2D)

 conv2d_2 (Conv2D)           (None, 3, 3, 64)          36928

 flatten (Flatten)           (None, 576)               0

 dense_1 (Dense)             (None, 64)                36928

 dropout (Dropout)           (None, 64)                0

 dense_2 (Dense)             (None, 10)                650

=================================================================
Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
_________________________________________________________________
```

As you can see, our (3, 3, 64) outputs were flattened into vectors of shape (576) before going through two Dense layers.

Compile and train the model

In [14]:
```
model_2.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model_2.fit(train_image_2, train_label, epochs=5)
```

```
Epoch 1/5
1875/1875 [==============================] - 13s 6ms/step - loss: 0.2837 - accuracy: 0.9
125
Epoch 2/5
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0905 - accuracy: 0.9
751
Epoch 3/5
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0646 - accuracy: 0.9
819
Epoch 4/5
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0508 - accuracy: 0.9
856
Epoch 5/5
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0418 - accuracy: 0.9
876
```

Out[14]:
```
<keras.callbacks.History at 0x7fd7400adcd0>
```

In [15]:
```
_, test_acc_2 = model_2.evaluate(test_image_2, test_label, verbose=0)
print('Testing Accuracy : %.4f'%test_acc_2)
```

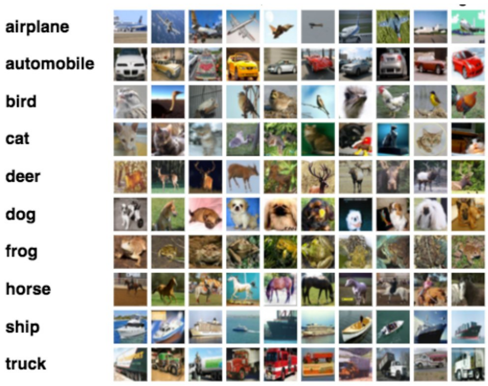
```
Testing Accuracy : 0.9912
```

As you can see, our simple CNN has achieved a test accuracy of 99%. Not bad for a few lines of code! For another style of writing a CNN (using the Keras Subclassing API and a GradientTape) head here.

Cifar-10

Actually MNIST is a easy dataset for the beginner. To demonstrate the power of *Neural Networks*, we need a larger dataset *CIFAR-10*.

CIFAR-10 consists of 60000 32x32 color image in 10 classes, with 6000 image per class. There are 50000 training image and 10000 test image. Here are the classes in the dataset, as well as 10 random image from each:



Before jumping to a complicated neural network model, we're going to start with **KNN** and **SVM**. The motivation here is to compare neural network model with traditional classifiers, and highlight the performance of neural network model.

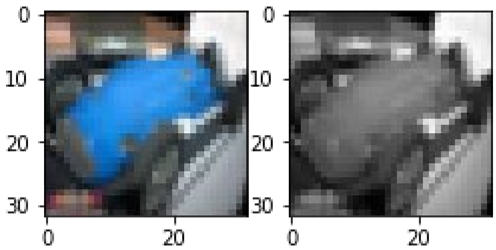
tf.keras.datasets offers convenient facilities that automatically access some well-known datasets. Let's load the CIFAR-10 in tf.keras.datasets:

In [16]:
```
# Loading Data

(X_train, y_train), (X_test, y_test) = datasets.cifar10.load_data()

# normalize inputs from 0-255 to 0.0-1.0
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train = X_train / 255.0
X_test = X_test / 255.0

# convert class vectors to binary vectors
Y_train = utils.to_categorical(y_train)
Y_test = utils.to_categorical(y_test)

print('X_train shape:', X_train.shape)
print('Y_train shape:', Y_train.shape)
print('X_test shape:', X_test.shape)
print('Y_test shape:', Y_test.shape)
```

```
X_train shape: (50000, 32, 32, 3)
Y_train shape: (50000, 10)
X_test shape: (10000, 32, 32, 3)
Y_test shape: (10000, 10)
```

For simplicity, we also convert the image into the grayscale. We use the Luma coding that is common in video systems:

In [17]:
```
# transform a 3-channel image into one channel
def grayscale(data, dtype='float32'):
    # luma coding weighted average in video systems
    r = np.asarray(.3, dtype=dtype)
    g = np.asarray(.59, dtype=dtype)
    b = np.asarray(.11, dtype=dtype)
    rst = r * data[:, :, :, 0] + g * data[:, :, :, 1] + b * data[:, :, :, 2]
    # add channel dimension
    rst = np.expand_dims(rst, axis=3)
    return rst

X_train_gray = grayscale(X_train)
X_test_gray = grayscale(X_test)

# plot a randomly chosen image
img = round(np.random.rand() * X_train.shape[0])
plt.figure(figsize=(4, 2))
plt.subplot(1, 2, 1)
plt.imshow(X_train[img], interpolation='none')
plt.subplot(1, 2, 2)
plt.imshow(
    X_train_gray[img, :, :, 0], cmap=plt.get_cmap('gray'), interpolation='none')
plt.show()
```



As we can see, the objects in grayscale image can still be recognizable.

Feature Selection

When coming to object detection, HOG (histogram of oriented gradients) is often extracted as a feature for classification. It first calculates the gradients of each image patch using sobel filter, then use the magnitudes and orientations of derived gradients to form a histogram per patch (a vector). After normalizing these histograms, it concatenates them into one HOG feature. For more details, read this tutorial.

> Note. one can directly feed the original image for classification; however, it will take lots of time to train and get worse performance.

In [18]:
```
def getHOGfeat(image,
               stride=8,
               orientations=8,
               pixels_per_cell=(8, 8),
               cells_per_block=(2, 2)):
    cx, cy = pixels_per_cell
    bx, by = cells_per_block
    sx, sy, sz = image.shape
    n_cellsx = int(np.floor(sx // cx))  # number of cells in x
    n_cellsy = int(np.floor(sy // cy))  # number of cells in y
```

```
    n_blocksx = (n_cellsx - bx) + 1
    n_blocksy = (n_cellsy - by) + 1
    gx = np.zeros((sx, sy), dtype=np.double)
    gy = np.zeros((sx, sy), dtype=np.double)
    eps = 1e-5
    grad = np.zeros((sx, sy, 2), dtype=np.double)
    for i in range(1, sx - 1):
        for j in range(1, sy - 1):
            gx[i, j] = image[i, j - 1] - image[i, j + 1]
            gy[i, j] = image[i + 1, j] - image[i - 1, j]
            grad[i, j, 0] = np.arctan(gy[i, j] / (gx[i, j] + eps)) * 180 / math.pi
            if gx[i, j] < 0:
                grad[i, j, 0] += 180
            grad[i, j, 0] = (grad[i, j, 0] + 360) % 360
            grad[i, j, 1] = np.sqrt(gy[i, j] ** 2 + gx[i, j] ** 2)
    normalised_blocks = np.zeros((n_blocksy, n_blocksx, by * bx * orientations))
    for y in range(n_blocksy):
        for x in range(n_blocksx):
            block = grad[y * stride:y * stride + 16, x * stride:x * stride + 16]
            hist_block = np.zeros(32, dtype=np.double)
            eps = 1e-5
            for k in range(by):
                for m in range(bx):
                    cell = block[k * 8:(k + 1) * 8, m * 8:(m + 1) * 8]
                    hist_cell = np.zeros(8, dtype=np.double)
                    for i in range(cy):
                        for j in range(cx):
                            n = int(cell[i, j, 0] / 45)
                            hist_cell[n] += cell[i, j, 1]
                    hist_block[(k * bx + m) * orientations:(k * bx + m + 1) * orientati
            normalised_blocks[y, x, :] = hist_block / np.sqrt(
                hist_block.sum() ** 2 + eps)
    return normalised_blocks.ravel()
```

Once we have our *getHOGfeat* function, we then get the HOG features of all image.

In [19]:

```
X_train_hog = []
X_test_hog = []

print('This will take some minutes.')

for img in tqdm(X_train_gray):
    img_hog = getHOGfeat(img)
    X_train_hog.append(img_hog)

for img in tqdm(X_test_gray):
    img_hog = getHOGfeat(img)
    X_test_hog.append(img_hog)

X_train_hog_array = np.asarray(X_train_hog)
X_test_hog_array = np.asarray(X_test_hog)
```

```
This will take some minutes.
100%|██████████████████████████████████████| 50000/50000 [05:31<00:00, 150.88it/s]
100%|██████████████████████████████████████| 10000/10000 [01:06<00:00, 151.50it/s]
```

K Nearest Neighbors (KNN) on CIFAR-10

scikit-learn provides off-the-shelf libraries for classification. For KNN and SVM classifiers, we can just import from scikit-learn to use.

In [20]:

```
# KNN
```

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# p=2 and metric='minkowski' means the Euclidean Distance
knn = KNeighborsClassifier(n_neighbors=11, p=2, metric='minkowski')

knn.fit(X_train_hog_array, y_train.ravel())
y_pred = knn.predict(X_test_hog_array)
print('[KNN]')
print('Misclassified samples: %d' % (y_test.ravel() != y_pred).sum())
print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
```

```
[KNN]
Misclassified samples: 5334
Accuracy: 0.47
```

We can observe that the accuracy of KNN on CIFAR-10 is embarrassingly bad.

Support Vector Machine (SVM) on CIFAR-10

In [21]:
```
# SVM
from sklearn.svm import SVC

print('This will take some minutes.')
start_time = time.time()

# C is the hyperparameter for the error penalty term
# gamma is the hyperparameter for the rbf kernel
svm_linear = SVC(kernel='linear', random_state=0, gamma=0.2, C=10.0)

svm_linear.fit(X_train_hog_array, y_train.ravel())
y_pred = svm_linear.predict(X_test_hog_array)
print('[Linear SVC]')
print('Misclassified samples: %d' % (y_test.ravel() != y_pred).sum())
print('Accuracy: %.2f' % accuracy_score(y_test.ravel(), y_pred))

print('{:.2f} sec.'.format(time.time()-start_time))
```

```
This will take some minutes.
[Linear SVC]
Misclassified samples: 4940
Accuracy: 0.51
495.08 sec.
```

By above, SVM is slightly better than KNN, but still poor. Next, we'll design a CNN model using tensorflow.

CNN on CIFAR-10

In [22]:
```
model_3 = models.Sequential()

#The 6 lines of code below define the convolutional base using a common pattern: a stac
model_3.add(layers.Conv2D(64, (5, 5), padding='same', activation='relu', input_shape=(3
model_3.add(layers.MaxPool2D(pool_size=3,strides=2,padding='same'))
model_3.add(layers.BatchNormalization())
model_3.add(layers.Conv2D(64, (5, 5), padding='same', activation='relu'))
model_3.add(layers.MaxPool2D(pool_size=3,strides=2,padding='same'))
model_3.add(layers.BatchNormalization())

model_3.add(layers.Flatten())
model_3.add(layers.Dense(384, activation='relu'))
model_3.add(layers.Dropout(0.5))
model_3.add(layers.Dense(192, activation='relu'))
```

```
model_3.add(layers.Dense(10, activation='softmax'))
model_3.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model_3.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_3 (Conv2D)           (None, 32, 32, 64)        4864

 max_pooling2d_2 (MaxPooling  (None, 16, 16, 64)       0
 2D)

 batch_normalization (BatchN  (None, 16, 16, 64)       256
 ormalization)

 conv2d_4 (Conv2D)           (None, 16, 16, 64)        102464

 max_pooling2d_3 (MaxPooling  (None, 8, 8, 64)         0
 2D)

 batch_normalization_1 (Batc  (None, 8, 8, 64)         256
 hNormalization)

 flatten_1 (Flatten)         (None, 4096)              0

 dense_3 (Dense)             (None, 384)               1573248

 dropout_1 (Dropout)         (None, 384)               0

 dense_4 (Dense)             (None, 192)               73920

 dense_5 (Dense)             (None, 10)                1930

=================================================================
Total params: 1,756,938
Trainable params: 1,756,682
Non-trainable params: 256
_________________________________________________________________
```

In [23]:

```
model_3.fit(X_train, y_train, epochs=5, validation_data=(X_test, y_test), verbose=1)
_, test_acc_3 = model_3.evaluate(X_test, y_test, verbose=0)
print('Testing Accuracy : %.4f'%test_acc_3)
```

```
Epoch 1/5
1563/1563 [==============================] - 13s 8ms/step - loss: 1.6064 - accuracy: 0.4
281 - val_loss: 1.3853 - val_accuracy: 0.5148
Epoch 2/5
1563/1563 [==============================] - 12s 8ms/step - loss: 1.1151 - accuracy: 0.6
074 - val_loss: 1.1946 - val_accuracy: 0.5959
Epoch 3/5
1563/1563 [==============================] - 12s 8ms/step - loss: 0.9340 - accuracy: 0.6
749 - val_loss: 1.0923 - val_accuracy: 0.6232
Epoch 4/5
1563/1563 [==============================] - 12s 8ms/step - loss: 0.8215 - accuracy: 0.7
152 - val_loss: 1.0149 - val_accuracy: 0.6511
Epoch 5/5
1563/1563 [==============================] - 11s 7ms/step - loss: 0.7306 - accuracy: 0.7
488 - val_loss: 0.8527 - val_accuracy: 0.7087
Testing Accuracy : 0.7087
```

Although Cifar10 is larger than Mnist, it's not large enough for the dataset you will meet in the following lessons. For large datasets, we can't feed all training data to the model due to the limit of GPU memory size. Even if we can feed all data into the model, we still want the process of loading data is efficient. **Input pipeline** is the common way to solve these.

Input Pipeline

Structure of an input pipeline

A typical TensorFlow training input pipeline can be framed as an ETL process:

1. Extract: Read data from memory (NumPy) or persistent storage -- either local (HDD or SSD) or remote (e.g. GCS or HDFS).
2. Transform: Use CPU to parse and perform preprocessing operations on the data such as shuffling, batching, and domain specific transformations such as image decompression and augmentation, text vectorization, or video temporal sampling.
3. Load: Load the transformed data onto the accelerator device(s) (e.g. GPU(s) or TPU(s)) that execute the machine learning model.

This pattern effectively utilizes the CPU, while reserving the accelerator for the heavy lifting of training your model. In addition, viewing input pipelines as an ETL process provides a framework that facilitates the application of performance optimizations.

tf.data API

To build a data input pipeline with **tf.data**, here are the steps that you can follow:

1. Define data source and initialize your Dataset object
2. Apply transformations on the dataset, following are some common useful techniques
   - map
   - shuffle
   - batch
   - repeat
   - prefetch
3. Create iterator

Construct your Dataset

To create an input pipeline, you must start with a data source. For example, to construct a **Dataset** from data in memory, you can use tf.data.Dataset.from_tensors() or tf.data.Dataset.from_tensor_slices(). Alternatively, if your input data is stored in a file in TFRecord format, you can use tf.data.TFRecordDataset().

Once you have a **Dataset** object, you can *transform* it into a new **Dataset** by chaining method calls on the **tf.data.Dataset** object. For example, you can apply per-element transformations such as **Dataset.map()**, and multi-element transformations such as **Dataset.batch()**. See the documentation for tf.data.Dataset for a complete list of transformations.

Now suppose we have simple data sources:

In [24]:
```
# number of samples
n_samples = 200
```

```
# an array with shape (n_samples, 5)
raw_data_a = np.random.rand(n_samples, 5)
# a list with length of n_samples from 0 to n_samples-1
raw_data_b = np.arange(n_samples)
print(raw_data_a.shape, raw_data_b.shape)
```

```
(200, 5) (200,)
```

We can create our tensorflow Dataset object with these two data using **tf.data.Dataset.from_tensor_slices**, which will automatically cut your data into slices:

In [25]:
```
# this tells the dataset that each row of raw_data_a is corresponding to each element o
raw_dataset = tf.data.Dataset.from_tensor_slices((raw_data_a, raw_data_b))
```

Consume elements

The **Dataset** object is a Python iterable. This makes it possible to consume its elements using a for loop:

In [26]:
```
# Here, we print the first 8 batches.
for i,elem in enumerate(raw_dataset):
    print("Batch ", i, ", b are ", elem)
    if i==7:
        break
```

```
Batch  0 , b are  (<tf.Tensor: shape=(5,), dtype=float64, numpy=array([0.66931552, 0.604
61851, 0.48526867, 0.35751424, 0.42699349])>, <tf.Tensor: shape=(), dtype=int64, numpy=0
>)
Batch  1 , b are  (<tf.Tensor: shape=(5,), dtype=float64, numpy=array([0.84717025, 0.399
1303 , 0.76538914, 0.67931079, 0.23154431])>, <tf.Tensor: shape=(), dtype=int64, numpy=1
>)
Batch  2 , b are  (<tf.Tensor: shape=(5,), dtype=float64, numpy=array([0.25540675, 0.808
12385, 0.04000756, 0.45713614, 0.97573978])>, <tf.Tensor: shape=(), dtype=int64, numpy=2
>)
Batch  3 , b are  (<tf.Tensor: shape=(5,), dtype=float64, numpy=array([0.0938586 , 0.659
4527 , 0.02701433, 0.31752666, 0.44325064])>, <tf.Tensor: shape=(), dtype=int64, numpy=3
>)
Batch  4 , b are  (<tf.Tensor: shape=(5,), dtype=float64, numpy=array([0.73322097, 0.489
46239, 0.3686729 , 0.04317379, 0.74173232])>, <tf.Tensor: shape=(), dtype=int64, numpy=4
>)
Batch  5 , b are  (<tf.Tensor: shape=(5,), dtype=float64, numpy=array([0.02034684, 0.051
58959, 0.71902387, 0.61182723, 0.23348519])>, <tf.Tensor: shape=(), dtype=int64, numpy=5
>)
Batch  6 , b are  (<tf.Tensor: shape=(5,), dtype=float64, numpy=array([0.18205827, 0.700
72316, 0.37018672, 0.11622035, 0.43881617])>, <tf.Tensor: shape=(), dtype=int64, numpy=6
>)
Batch  7 , b are  (<tf.Tensor: shape=(5,), dtype=float64, numpy=array([0.70428993, 0.008
45075, 0.02440134, 0.90915499, 0.30298183])>, <tf.Tensor: shape=(), dtype=int64, numpy=7
>)
```

Or by explicitly creating a Python iterator using **iter** and consuming its elements using **next**:

In [27]:
```
# Here, we print the first 8 batches.
it = iter(raw_dataset)
for i in range(8):
    print("Batch ", i, ", b are ", next(it))
```

```
Batch  0 , b are  (<tf.Tensor: shape=(5,), dtype=float64, numpy=array([0.66931552, 0.604
61851, 0.48526867, 0.35751424, 0.42699349])>, <tf.Tensor: shape=(), dtype=int64, numpy=0
>)
Batch  1 , b are  (<tf.Tensor: shape=(5,), dtype=float64, numpy=array([0.84717025, 0.399
1303 , 0.76538914, 0.67931079, 0.23154431])>, <tf.Tensor: shape=(), dtype=int64, numpy=1
>)
Batch  2 , b are  (<tf.Tensor: shape=(5,), dtype=float64, numpy=array([0.25540675, 0.808
12385, 0.04000756, 0.45713614, 0.97573978])>, <tf.Tensor: shape=(), dtype=int64, numpy=2
```

```
>)
Batch  3 , b are  (<tf.Tensor: shape=(5,), dtype=float64, numpy=array([0.0938586 , 0.659
4527 , 0.02701433, 0.31752666, 0.44325064])>, <tf.Tensor: shape=(), dtype=int64, numpy=3
>)
Batch  4 , b are  (<tf.Tensor: shape=(5,), dtype=float64, numpy=array([0.73322097, 0.489
46239, 0.3686729 , 0.04317379, 0.74173232])>, <tf.Tensor: shape=(), dtype=int64, numpy=4
>)
Batch  5 , b are  (<tf.Tensor: shape=(5,), dtype=float64, numpy=array([0.02034684, 0.051
58959, 0.71902387, 0.61182723, 0.23348519])>, <tf.Tensor: shape=(), dtype=int64, numpy=5
>)
Batch  6 , b are  (<tf.Tensor: shape=(5,), dtype=float64, numpy=array([0.18205827, 0.700
72316, 0.37018672, 0.11622035, 0.43881617])>, <tf.Tensor: shape=(), dtype=int64, numpy=6
>)
Batch  7 , b are  (<tf.Tensor: shape=(5,), dtype=float64, numpy=array([0.70428993, 0.008
45075, 0.02440134, 0.90915499, 0.30298183])>, <tf.Tensor: shape=(), dtype=int64, numpy=7
>)
```

Apply transformations

Next, according to your needs, you can preprocess your data in this step.

map

For example, **Dataset.map()** provide element-wise customized data preprocessing.

In [28]:

```python
def preprocess_function(one_row_a, one_b):
    """
        Input: one slice of the dataset
        Output: modified slice
    """
    # Do some data preprocessing, you can also input filenames and load data in here
    # Here, we transform each row of raw_data_a to its sum and mean
    one_row_a = [tf.reduce_sum(one_row_a), tf.reduce_mean(one_row_a)]

    return one_row_a, one_b

raw_dataset = raw_dataset.map(preprocess_function, num_parallel_calls=tf.data.experimen
```

In [29]:

```python
it = iter(raw_dataset)
for i in range(8):
    print("Batch ", i, ", b are ", next(it))
```

```
Batch  0 , b are  (<tf.Tensor: shape=(2,), dtype=float64, numpy=array([2.54371044, 0.508
74209])>, <tf.Tensor: shape=(), dtype=int64, numpy=0>)
Batch  1 , b are  (<tf.Tensor: shape=(2,), dtype=float64, numpy=array([2.92254479, 0.584
50896])>, <tf.Tensor: shape=(), dtype=int64, numpy=1>)
Batch  2 , b are  (<tf.Tensor: shape=(2,), dtype=float64, numpy=array([2.53641408, 0.507
28282])>, <tf.Tensor: shape=(), dtype=int64, numpy=2>)
Batch  3 , b are  (<tf.Tensor: shape=(2,), dtype=float64, numpy=array([1.54110293, 0.308
22059])>, <tf.Tensor: shape=(), dtype=int64, numpy=3>)
Batch  4 , b are  (<tf.Tensor: shape=(2,), dtype=float64, numpy=array([2.37626237, 0.475
25247])>, <tf.Tensor: shape=(), dtype=int64, numpy=4>)
Batch  5 , b are  (<tf.Tensor: shape=(2,), dtype=float64, numpy=array([1.63627272, 0.327
25454])>, <tf.Tensor: shape=(), dtype=int64, numpy=5>)
Batch  6 , b are  (<tf.Tensor: shape=(2,), dtype=float64, numpy=array([1.80800466, 0.361
60093])>, <tf.Tensor: shape=(), dtype=int64, numpy=6>)
Batch  7 , b are  (<tf.Tensor: shape=(2,), dtype=float64, numpy=array([1.94927884, 0.389
85577])>, <tf.Tensor: shape=(), dtype=int64, numpy=7>)
```

shuffle

Dataset.shuffle(buffer_size) maintains a fixed-size buffer and chooses the next element uniformly at random from that buffer. This way, you can see your data coming with different order in different epoch. This can prevent your model overfit on the order of your training data.

In [30]:
```
dataset = raw_dataset.shuffle(16)
```

In [31]:
```
idxs = []
for i,elem in enumerate(dataset):
    print("Batch ", i, ", b are ", elem)
    idxs.append(elem[1].numpy())
    if i==7:
        break

print("\nThe order of the first 8 shuffle from [0, 1, 2, 3, 4, 5, 6, 7] to ",idxs)
```

```
Batch  0 , b are  (<tf.Tensor: shape=(2,), dtype=float64, numpy=array([2.24929143, 0.449
85829])>, <tf.Tensor: shape=(), dtype=int64, numpy=13>)
Batch  1 , b are  (<tf.Tensor: shape=(2,), dtype=float64, numpy=array([2.51418351, 0.502
8367 ])>, <tf.Tensor: shape=(), dtype=int64, numpy=8>)
Batch  2 , b are  (<tf.Tensor: shape=(2,), dtype=float64, numpy=array([2.52106856, 0.504
21371])>, <tf.Tensor: shape=(), dtype=int64, numpy=17>)
Batch  3 , b are  (<tf.Tensor: shape=(2,), dtype=float64, numpy=array([2.01846541, 0.403
69308])>, <tf.Tensor: shape=(), dtype=int64, numpy=10>)
Batch  4 , b are  (<tf.Tensor: shape=(2,), dtype=float64, numpy=array([3.15692468, 0.631
38494])>, <tf.Tensor: shape=(), dtype=int64, numpy=12>)
Batch  5 , b are  (<tf.Tensor: shape=(2,), dtype=float64, numpy=array([2.27690284, 0.455
38057])>, <tf.Tensor: shape=(), dtype=int64, numpy=16>)
Batch  6 , b are  (<tf.Tensor: shape=(2,), dtype=float64, numpy=array([2.54371044, 0.508
74209])>, <tf.Tensor: shape=(), dtype=int64, numpy=0>)
Batch  7 , b are  (<tf.Tensor: shape=(2,), dtype=float64, numpy=array([3.28908268, 0.657
81654])>, <tf.Tensor: shape=(), dtype=int64, numpy=19>)

The order of the first 8 shuffle from [0, 1, 2, 3, 4, 5, 6, 7] to  [13, 8, 17, 10, 12, 1
6, 0, 19]
```

batch

Now our dataset is one example by one example. However, in reality, we usually want to read one batch at a time, thus we can call **Dataset.batch(batch_size)** to stack batch_size elements together.

Note: Be careful that if you apply **Dataset.shuffle** after **Dataset.batch**, you'll get shuffled batch but data in a batch remains the same.

In [32]:
```
dataset = dataset.batch(2, drop_remainder=False)
```

In [33]:
```
idxs = []
for i,elem in enumerate(dataset):
    print("Batch ", i, ", b are ", elem)
    print("")
    idxs.append(elem[1].numpy())
    if i==7:
        break

print("\nAfter `dataset.batch(2)`,\nBatch 0 is combined by %dth and %dth,\nBatch 1 is c
```

```
Batch  0 , b are  (<tf.Tensor: shape=(2, 2), dtype=float64, numpy=
array([[2.51418351, 0.5028367 ],
       [2.24929143, 0.44985829]])>, <tf.Tensor: shape=(2,), dtype=int64, numpy=array([
8, 13])>)

Batch  1 , b are  (<tf.Tensor: shape=(2, 2), dtype=float64, numpy=
array([[2.92254479, 0.58450896],
       [2.54371044, 0.50874209]])>, <tf.Tensor: shape=(2,), dtype=int64, numpy=array([1,
0])>)
```

```
Batch  2 , b are  (<tf.Tensor: shape=(2, 2), dtype=float64, numpy=
array([[1.80800466, 0.36160093],
       [1.54110293, 0.30822059]])>, <tf.Tensor: shape=(2,), dtype=int64, numpy=array([6,
3])>)

Batch  3 , b are  (<tf.Tensor: shape=(2, 2), dtype=float64, numpy=
array([[3.28908268, 0.65781654],
       [2.27690284, 0.45538057]])>, <tf.Tensor: shape=(2,), dtype=int64, numpy=array([1
9, 16])>)

Batch  4 , b are  (<tf.Tensor: shape=(2, 2), dtype=float64, numpy=
array([[1.94927884, 0.38985577],
       [2.32019   , 0.464038  ]])>, <tf.Tensor: shape=(2,), dtype=int64, numpy=array([7,
9])>)

Batch  5 , b are  (<tf.Tensor: shape=(2, 2), dtype=float64, numpy=
array([[2.01846541, 0.40369308],
       [2.09043057, 0.41808611]])>, <tf.Tensor: shape=(2,), dtype=int64, numpy=array([1
0, 24])>)

Batch  6 , b are  (<tf.Tensor: shape=(2, 2), dtype=float64, numpy=
array([[2.67089385, 0.53417877],
       [2.37626237, 0.47525247]])>, <tf.Tensor: shape=(2,), dtype=int64, numpy=array([2
6,  4])>)

Batch  7 , b are  (<tf.Tensor: shape=(2, 2), dtype=float64, numpy=
array([[3.49696451, 0.6993929 ],
       [2.53641408, 0.50728282]])>, <tf.Tensor: shape=(2,), dtype=int64, numpy=array([1
4,  2])>)


After `dataset.batch(2)`,
Batch 0 is combined by 8th and 13th,
Batch 1 is combined by 1th and 0th,
Batch 2 is combined by 6th and 3th, etc.
```

repeat

Repeats this dataset count times.

Dataset.repeat(count) allow you iterate over a dataset in multiple epochs. **count = None or -1** will let the dataset repeats indefinitely.

In [34]:
```
dataset = dataset.repeat(2)
```

If you would like to perform a custom computation (e.g. to collect statistics) at the end of each epoch then it's simplest to restart the dataset iteration on each epoch:

In [35]:
```
epochs = 3

for epoch in range(epochs):
    size = 0
    n_batch = 0
    for batch in dataset:
        size += len(batch[1])
        n_batch += 1
    print("End of epoch %d: Total %d batches in this epoch with %d pieces of data"%(epo
```

```
End of epoch 0: Total 200 batches in this epoch with 400 pieces of data
End of epoch 1: Total 200 batches in this epoch with 400 pieces of data
End of epoch 2: Total 200 batches in this epoch with 400 pieces of data
```

Note: Since we `repeat(2)` to the dataset, the code above actually iterates each piece of the dataset **6 times** even though `epochs = 3`.

Therefore, I prefer to set a desired number of epoch rather than using `repeat()`, unless you want the same piece of data to potentially be ordered together, e.g. `dataset.repeat(n).shuffle(n)`.

prefetch

Creates a Dataset that prefetches elements from this dataset.

Dataset.prefetch(buffer_size) allow you decouple the time when data is produced from the time when data is consumed.

In [36]:
```
dataset = dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
```

repeat+batch / batch+repeat

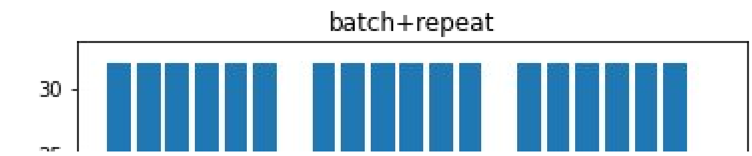
The Dataset.repeat transformation concatenates its arguments without signaling the end of one epoch and the beginning of the next epoch. Because of this a Dataset.batch applied after Dataset.repeat will yield batches that stradle epoch boundaries:

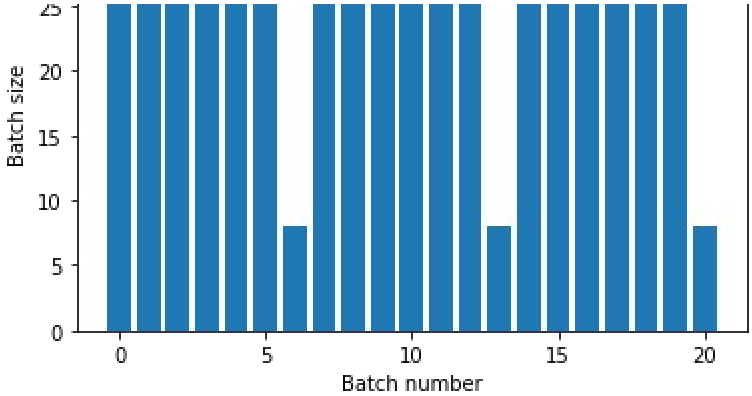
In [37]:
```
# use this function to plot the size of each batch.
def plot_batch_sizes(ds,title):
    batch_sizes = [batch[1].shape[0] for batch in ds]
    plt.bar(range(len(batch_sizes)), batch_sizes)
    plt.xlabel('Batch number')
    plt.ylabel('Batch size')
    plt.title(title)
    plt.show()
```

In [38]:
```
# plot the bar diagram of repeat+batch
repeat_batch_ds = raw_dataset.repeat(3).batch(32)
plot_batch_sizes(repeat_batch_ds,'repeat+batch')
```

If you need clear epoch separation, put Dataset.batch before the repeat:

In [39]:
```
# plot the bar diagram of batch+repeat
batch_repeat_ds = raw_dataset.batch(32).repeat(3)
plot_batch_sizes(batch_repeat_ds,'batch+repeat')
```





shufflt+repeat / repeat+shufflt

As with Dataset.batch the order relative to Dataset.repeat matters.

Dataset.shuffle doesn't signal the end of an epoch until the shuffle buffer is empty. So a shuffle placed before a repeat will show every element of one epoch before moving to the next.

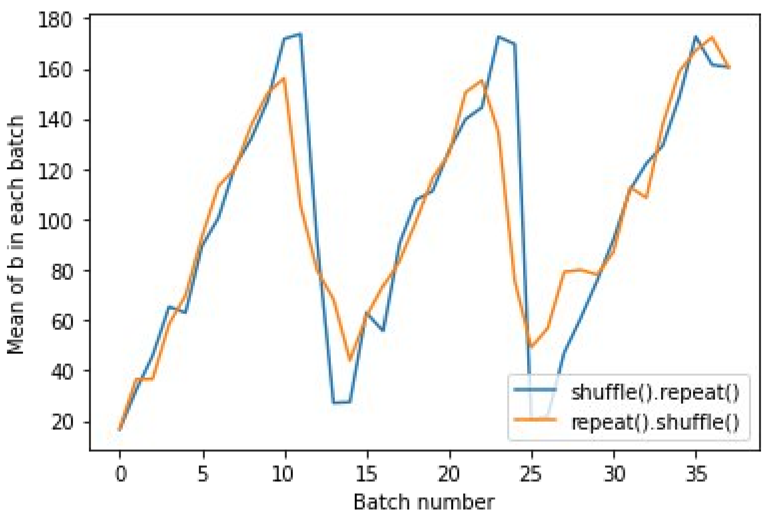
But a repeat before a shuffle mixes the epoch boundaries together.

In [40]:

```
# You can find that a repeat before a shuffle mixes the epoch boundaries in this graph.
shuffle_repeat_ds = raw_dataset.shuffle(32).repeat(3).batch(16)
repeat_shuffle_ds = raw_dataset.repeat(3).shuffle(32).batch(16)

shuffle_repeat = [batch[1].numpy().mean() for batch in shuffle_repeat_ds]
repeat_shuffle = [batch[1].numpy().mean() for batch in repeat_shuffle_ds]

plt.plot(shuffle_repeat, label="shuffle().repeat()")
plt.plot(repeat_shuffle, label="repeat().shuffle()")
plt.xlabel('Batch number')
plt.ylabel("Mean of b in each batch")
plt.legend()
```

Out[40]:

```
<matplotlib.legend.Legend at 0x7fd9111fbb80>
```



Now, let's start designing our cnn model!

CNN Model for CIFAR 10

Loading Data Manually

To know how it works under the hood, let's load CIFAR-10 by our own (not using tf.keras). According the descripion, the dataset file is divided into five training batches and one test batch, each with 10000 image. The test batch contains exactly 1000 randomly-selected image from each class.

In [41]:
```
# the url to download CIFAR-10 dataset (binary version)
# see format and details here: http://www.cs.toronto.edu/~kriz/cifar.html
DATA_URL = 'https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz'
# the image size we want to keep
IMAGE_SIZE_CROPPED = 24
IMAGE_HEIGHT = 32
IMAGE_WIDTH = 32
IMAGE_DEPTH = 3
```

In [42]:
```
# donwnload data
if not os.path.exists("cifar-10-batches-py/"):
    cifar10 = utils.get_file('cifar-10-python.tar.gz',
                                  cache_subdir=os.path.abspath('.'),
                                  origin = DATA_URL,
                                  extract = True)
```

In [43]:
```
DEST_DIRECTORY = 'cifar-10-batches-py'
filenames_train = [os.path.join(DEST_DIRECTORY,'data_batch_%d' % i) for i in range(1,6)
filenames_test = [os.path.join(DEST_DIRECTORY,'test_batch')]
```

In [44]:
```
# save (img_path, label) pairs
with open('cifar10_train.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(['filenames'])
    writer.writerows(np.array(filenames_train).reshape(-1,1))

with open('cifar10_test.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(['filenames'])
    writer.writerows(np.array(filenames_test).reshape(-1,1))
```

In [45]:
```
# read data
def read_file(file):
    with open(file, 'rb') as fo:
        raw_data = pickle.load(fo, encoding='bytes')
    return raw_data[b'data'],raw_data[b'labels']

# parse training data
@tf.function
def map_fun(image,label):
    image = tf.reshape(image,[IMAGE_DEPTH,IMAGE_HEIGHT,IMAGE_WIDTH])
    image = tf.divide(tf.cast(tf.transpose(image,[1,2,0]),tf.float32),255.0)
    label = tf.one_hot(label,10)
    distorted_image = tf.image.resize_with_crop_or_pad(image, IMAGE_SIZE_CROPPED,IMAGE_
    distorted_image = tf.image.random_flip_left_right(distorted_image)
    distorted_image = tf.image.random_brightness(distorted_image, max_delta=63)
    distorted_image = tf.image.random_contrast(distorted_image, lower=0.2, upper=1.8)
    distorted_image = tf.image.per_image_standardization(distorted_image)
    return distorted_image, label

# parse testing data
@tf.function
def map_fun_test(image,label):
    image = tf.reshape(image,[IMAGE_DEPTH,IMAGE_HEIGHT,IMAGE_WIDTH])
    image = tf.divide(tf.cast(tf.transpose(image,[1,2,0]),tf.float32),255.0)
    label = tf.one_hot(label,10)
```

```
    distorted_image = tf.image.resize_with_crop_or_pad(image, IMAGE_SIZE_CROPPED,IMAGE_
    distorted_image = tf.image.per_image_standardization(distorted_image)
    return distorted_image, label
```

In [46]:

```
X_train = None
Y_train = None
X_test = None
Y_test = None

for filename in filenames_train:
    image,label = read_file(filename)
    X_train = image if X_train is None else np.concatenate((X_train,image))
    Y_train = label if Y_train is None else np.concatenate((Y_train,label))

for filename in filenames_test:
    image,label = read_file(filename)
    X_test = image if X_test is None else np.concatenate((X_test,image))
    Y_test = label if Y_test is None else np.concatenate((Y_test,label))

# Construct training Dataset
dataset = tf.data.Dataset.from_tensor_slices((X_train,Y_train)).map(map_fun).shuffle(10

# # Construct testing Dataset
dataset_test = tf.data.Dataset.from_tensor_slices((X_test,Y_test)).map(map_fun_test).ba
```

In [47]:

```
model_cifar = models.Sequential()

model_cifar.add(layers.Conv2D(64, (5, 5), padding='same', activation='relu', input_shap
model_cifar.add(layers.MaxPool2D(pool_size=3,strides=2,padding='same'))
model_cifar.add(layers.BatchNormalization())

model_cifar.add(layers.Conv2D(64, (5, 5), padding='same', activation='relu'))
model_cifar.add(layers.MaxPool2D(pool_size=3,strides=2,padding='same'))
model_cifar.add(layers.BatchNormalization())

model_cifar.add(layers.Flatten())
model_cifar.add(layers.Dense(384, activation='relu'))
model_cifar.add(layers.Dropout(0.5))
model_cifar.add(layers.Dense(192, activation='relu'))
model_cifar.add(layers.Dropout(0.5))
model_cifar.add(layers.Dense(10, activation='softmax'))
```

In [48]:

```
model_cifar.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model_cifar.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_5 (Conv2D)           (None, 24, 24, 64)        4864

 max_pooling2d_4 (MaxPooling  (None, 12, 12, 64)       0
 2D)

 batch_normalization_2 (Batc  (None, 12, 12, 64)       256
 hNormalization)

 conv2d_6 (Conv2D)           (None, 12, 12, 64)        102464

 max_pooling2d_5 (MaxPooling  (None, 6, 6, 64)         0
 2D)
```

```
 batch_normalization_3 (Batc  (None, 6, 6, 64)         256
 hNormalization)

 flatten_2 (Flatten)         (None, 2304)              0

 dense_6 (Dense)             (None, 384)               885120

 dropout_2 (Dropout)         (None, 384)               0

 dense_7 (Dense)             (None, 192)               73920

 dropout_3 (Dropout)         (None, 192)               0

 dense_8 (Dense)             (None, 10)                1930

=================================================================
Total params: 1,068,810
Trainable params: 1,068,554
Non-trainable params: 256
_________________________________________________________________
```

In [49]:

```python
model_cifar.fit(dataset, epochs=5, validation_data=dataset_test, verbose=1)
_, test_acc = model_cifar.evaluate(dataset_test, verbose=0)
print('test accuracy:',test_acc)
```

```
Epoch 1/5
782/782 [==============================] - 15s 17ms/step - loss: 1.7770 - accuracy: 0.37
14 - val_loss: 1.3185 - val_accuracy: 0.5377
Epoch 2/5
782/782 [==============================] - 15s 17ms/step - loss: 1.3668 - accuracy: 0.52
30 - val_loss: 1.1346 - val_accuracy: 0.6129
Epoch 3/5
782/782 [==============================] - 14s 16ms/step - loss: 1.1993 - accuracy: 0.58
68 - val_loss: 0.9907 - val_accuracy: 0.6583
Epoch 4/5
782/782 [==============================] - 15s 17ms/step - loss: 1.0905 - accuracy: 0.62
59 - val_loss: 0.9453 - val_accuracy: 0.6792
Epoch 5/5
782/782 [==============================] - 14s 16ms/step - loss: 1.0106 - accuracy: 0.65
53 - val_loss: 0.8773 - val_accuracy: 0.7092
test accuracy: 0.7092000246047974
```

Optimization for input pipeline

We all know that GPUs can radically reduce the time required to execute a single training step; however, all other affairs (including data loading, data transformations, memory copy from CPU to GPUs) are done by CPU, which **sometimes** becomes the bottleneck instead. We have learned above that there are lots transformations that make datasets more complex and reusable. Now, we are going to accelerate the input pipeline for better training performance, following this guide.

The code below briefly do the same thing in CNN Model for CIFAR 10. However, we change the dataset structure to show the time comsuming during the training.

Dataset with time

In [50]:

```python
# construct a new dataset with time informantion
class TimeMeasuredDataset(tf.data.Dataset):
    # OUTPUT: (steps, timings, counters, img, label)
    OUTPUT_SIGNATURE=(
        tf.TensorSpec(shape=(2, 1), dtype=tf.string), # steps: [("Open",), ("Read",)]
        tf.TensorSpec(shape=(2, 2), dtype=tf.float32), # timings: [(open_enter, open_el
```

```
        tf.TensorSpec(shape=(2, 3), dtype=tf.int32), # counters: [(instance_idx, epoch_
        tf.TensorSpec(shape=(3072), dtype=tf.float32), # img: 32*32*3
        tf.TensorSpec(shape=(), dtype=tf.int32) # label
    )

    _INSTANCES_COUNTER = itertools.count()  # Number of datasets generated
    _EPOCHS_COUNTER = defaultdict(itertools.count)  # Number of epochs done for each da

    def _generator(instance_idx, filename, open_file, read_file):
        epoch_idx = next(TimeMeasuredDataset._EPOCHS_COUNTER[instance_idx])

        # Opening the file
        open_enter = time.perf_counter()
        filenames = open_file(filename)
        open_elapsed = time.perf_counter() - open_enter
        # ----------------

        # Reading the file
        read_enter = time.perf_counter()
        imgs, label = [], []
        for filename in filenames:
            tmp_imgs, tmp_label = read_file(filename)
            imgs.append(tmp_imgs)
            label.append(tmp_label)
        imgs = tf.concat(imgs, axis=0)
        label = tf.concat(label, axis=0)
        read_elapsed = (time.perf_counter() - read_enter) / imgs.shape[0]

        for sample_idx in range(imgs.shape[0]):
            read_enter = read_enter if sample_idx == 0 else time.perf_counter()

            yield (
                [("Open",), ("Read",)],
                [(open_enter, open_elapsed), (read_enter, read_elapsed)],
                [(instance_idx, epoch_idx, -1), (instance_idx, epoch_idx, sample_idx)],
                imgs[sample_idx],
                label[sample_idx]
            )
            open_enter, open_elapsed = -1., -1.  # Negative values will be filtered


    def __new__(cls, filename, open_file, read_file):
        def generator_func(instance_idx, filename):
            return cls._generator(instance_idx, filename, open_file, read_file)

        return tf.data.Dataset.from_generator(
            generator_func,
            output_signature=cls.OUTPUT_SIGNATURE,
            args=(next(cls._INSTANCES_COUNTER), filename)
        )
```

The block above defines our dataset, not only **image** and **label**, but also **steps**, **timings** and **counters**. Therefore, if we take two examples:

In [51]:

```
def open_file(filename):
    rows = pd.read_csv(filename.decode("utf-8"))
    filenames = rows['filenames']
    return filenames

def read_file(filename):
    with open(filename, 'rb') as fo:
        raw_data = pickle.load(fo, encoding='bytes')
    return raw_data[b'data'], raw_data[b'labels']
```

```python
def dataset_generator_fun_train(*args):
    return TimeMeasuredDataset('cifar10_train.csv', open_file, read_file)

def dataset_generator_fun_test(*args):
    return TimeMeasuredDataset('cifar10_test.csv', open_file, read_file)

for i in tf.data.Dataset.range(1).flat_map(dataset_generator_fun_train).take(2):
    print(i)
    print("now time", time.perf_counter())
    print("-------------------------------------------------------------------------------
```

```
(<tf.Tensor: shape=(2, 1), dtype=string, numpy=
array([[b'Open'],
       [b'Read']], dtype=object)>, <tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[4.9066791e+05, 4.5651230e-03],
       [4.9066794e+05, 2.5628499e-06]], dtype=float32)>, <tf.Tensor: shape=(2, 3), dtype
=int32, numpy=
array([[ 0,  0, -1],
       [ 0,  0,  0]], dtype=int32)>, <tf.Tensor: shape=(3072,), dtype=float32, numpy=arr
ay([ 59.,  43.,  50., ..., 140.,  84.,  72.], dtype=float32)>, <tf.Tensor: shape=(), dty
pe=int32, numpy=6>)
now time 490668.059314616
--------------------------------------------------------------------------------
(<tf.Tensor: shape=(2, 1), dtype=string, numpy=
array([[b'Open'],
       [b'Read']], dtype=object)>, <tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[-1.0000000e+00, -1.0000000e+00],
       [ 4.9066806e+05,  2.5628499e-06]], dtype=float32)>, <tf.Tensor: shape=(2, 3), dty
pe=int32, numpy=
array([[ 0,  0, -1],
       [ 0,  0,  1]], dtype=int32)>, <tf.Tensor: shape=(3072,), dtype=float32, numpy=arr
ay([154., 126., 105., ..., 139., 142., 144.], dtype=float32)>, <tf.Tensor: shape=(), dty
pe=int32, numpy=9>)
now time 490668.063010501
--------------------------------------------------------------------------------
```

In above, the first block shows:

- 0th instance, 0th epoch, -1th example is **Open** at 4.9066791e+05 and spend 4.5651230e-03 seconds
- 0th instance, 0th epoch, 0th example **Read** at 4.9066794e+05 and spend 2.5628499e-06 seconds
- 0th exmaple's image: [59., 43., 50., ..., 140., 84., 72.] with shape=(3072,)
- 0th example's lable: 6

The second block shows:

- 0th instance, 0th epoch, -1th example is **Open** at -1 and spend -1 seconds
- 0th instance, 0th epoch, 1th example **Read** at 4.9066806e+05 and spend 2.5628499e-06 seconds
- 1th example's image: [154., 126., 105., ..., 139., 142., 144.] with shape=(3072,)
- 1th example's lable: 9

Note that since 'cifar10_train.csv' is only opened once, only the first example is recorded **Open** time and the after examples are assigned with -1 (negative values would be filtered out). Also, the example_idx is assigned with -1, meaning that all examples are opened at the same time.

Besides, the duration of **Read** in all example are same because we calculte in average.

Map function with time

Now, Image shape is 3072
$(= 32$
$\cdot 32 \cdot 3)$
and label is 0 to 1, so we have to apply map funciton to each example, meanwhile recording the time cost of map function:

In [52]:
```
IMAGE_SIZE_CROPPED = 24
IMAGE_HEIGHT = 32
IMAGE_WIDTH = 32
IMAGE_DEPTH = 3

def map_decorator(func):
    def wrapper(steps, times, values, image, label):
        # Use a tf.py_function to prevent auto-graph from compiling the method
        return tf.py_function(
            func,
            inp=(steps, times, values, image, label),
            Tout=(steps.dtype, times.dtype, values.dtype, image.dtype, tf.float32)
        )
    return wrapper

@map_decorator
def map_fun_with_time(steps, times, values, image, label):
    # sleep to avoid concurrency issue
    time.sleep(0.05)

    # record the enter time into map_fun()
    map_enter = time.perf_counter()

    image = tf.reshape(image,[IMAGE_DEPTH, IMAGE_HEIGHT, IMAGE_WIDTH])
    image = tf.divide(tf.cast(tf.transpose(image,[1, 2, 0]),tf.float32),255.0)
    label = tf.one_hot(label, 10)
    distorted_image = tf.image.random_crop(image, [IMAGE_SIZE_CROPPED,IMAGE_SIZE_CROPPE
    # distorted_image = tf.image.resize(image, [IMAGE_SIZE_CROPPED,IMAGE_SIZE_CROPPED])
    distorted_image = tf.image.random_flip_left_right(distorted_image)
    distorted_image = tf.image.random_brightness(distorted_image, max_delta=63)
    distorted_image = tf.image.random_contrast(distorted_image, lower=0.2, upper=1.8)
    distorted_image = tf.image.per_image_standardization(distorted_image)

    map_elapsed = time.perf_counter() - map_enter
    # ----------------

    return tf.concat((steps, [["Map"]]), axis=0),\
           tf.concat((times, [[map_enter, map_elapsed]]), axis=0),\
           tf.concat((values, [values[-1]]), axis=0),\
           distorted_image,\
           label

@map_decorator
def map_fun_test_with_time(steps, times, values, image, label):
    # sleep to avoid concurrency issue
    time.sleep(0.05)

    # record the enter time into map_fun_test()
    map_enter = time.perf_counter()

    image = tf.reshape(image,[IMAGE_DEPTH,IMAGE_HEIGHT,IMAGE_WIDTH])
    image = tf.divide(tf.cast(tf.transpose(image,[1,2,0]),tf.float32),255.0)
    label = tf.one_hot(label,10)
```

```
    distorted_image = tf.image.resize(image, [IMAGE_SIZE_CROPPED,IMAGE_SIZE_CROPPED])
    distorted_image = tf.image.per_image_standardization(distorted_image)

    map_elapsed = time.perf_counter() - map_enter
    # ----------------

    return tf.concat((steps, [["Map"]]), axis=0),\
           tf.concat((times, [[map_enter, map_elapsed]]), axis=0),\
           tf.concat((values, [values[-1]]), axis=0),\
           distorted_image,\
           label
```

Note that the `@map_decorator` in map function is necessary for record correct time. Therefore, if we take two examples again with map functions:

In [53]:
```
for i in tf.data.Dataset.range(1).flat_map(dataset_generator_fun_train).map(map_fun_wit
    print(i)
    print("now time", time.perf_counter())
    print("-----------------------------------------------------------------------------
```

```
(<tf.Tensor: shape=(3, 1), dtype=string, numpy=
array([[b'Open'],
       [b'Read'],
       [b'Map']], dtype=object)>, <tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[4.9066819e+05, 2.9742192e-03],
       [4.9066819e+05, 2.3807518e-06],
       [4.9066838e+05, 4.5068976e-02]], dtype=float32)>, <tf.Tensor: shape=(3, 3), dtype
=int32, numpy=
array([[ 1,  0, -1],
       [ 1,  0,  0],
       [ 1,  0,  0]], dtype=int32)>, <tf.Tensor: shape=(24, 24, 3), dtype=float32, numpy
=
array([[[ 0.794585  ,  0.1525154 , -0.43435538],
        [ 0.794585  ,  0.20373973, -0.36603695],
        [ 0.48714247, -0.08662798, -0.6393299 ],
        ...,
        [ 0.94831586,  0.40871426, -0.1098188 ],
        [ 0.8458286 ,  0.2891522 , -0.21230607],
        [ 0.77751017,  0.27207744, -0.19521199]],

       [[ 0.40173   , -0.35992092, -1.1005033 ],
        [ 0.23092432, -0.49657702, -1.2029905 ],
        [ 0.06011866, -0.65028864, -1.3225526 ],
        ...,
        [ 0.35048637, -0.34284613, -1.0151101 ],
        [ 0.28216797, -0.42823932, -1.1005033 ],
        [ 0.33341157, -0.35992092, -0.98094124]],

       [[ 0.17968069, -0.53072655, -1.2200654 ],
        [ 0.16260591, -0.5136518 , -1.2029905 ],
        [ 0.16260591, -0.49657702, -1.1688217 ],
        ...,
        [ 0.26509318, -0.41116452, -1.1005033 ],
        [ 0.29924273, -0.41116452, -1.1005033 ],
        [ 0.11136229, -0.599045  , -1.23714   ]],

       ...,

       [[ 1.4607521 ,  0.989469  ,  0.5563291 ],
        [ 1.3582649 ,  0.818644  ,  0.30011094],
        [ 1.1703844 ,  0.4428831 , -0.2635497 ],
        ...,
        [ 0.45297363,  0.13542132, -1.2883837 ],
        [ 0.60670453,  0.13542132, -1.1859157 ],
        [ 1.3070213 ,  0.92115057, -0.53684264]],
```

```
        [[ 1.2387029 ,  0.47703266, -0.17813721],
         [ 1.204534  ,  0.47703266, -0.12689358],
         [ 1.477827  ,  0.7332508 ,  0.06098687],
         ...,
         [ 0.17968069,  0.23790859, -1.7837453 ],
         [ 0.77751017,  0.47703266, -1.2029905 ],
         [ 1.6486326 ,  1.2969115 , -0.21230607]],

        [[ 1.1362156 ,  0.22083381, -0.5710115 ],
         [ 1.1020467 ,  0.25498337, -0.46852422],
         [ 1.3070213 ,  0.56244516, -0.16106243],
         ...,
         [-0.00819975,  0.0841777 , -1.80082   ],
         [ 0.9653906 ,  0.69908196, -0.87845397],
         [ 1.4949018 ,  1.1261058 , -0.2635497 ]]], dtype=float32)>, <tf.Tensor: shape=(1
0,), dtype=float32, numpy=array([0., 0., 0., 0., 0., 0., 1., 0., 0., 0.], dtype=float32)
>)
now time 490668.42789291
-------------------------------------------------------------------------------
(<tf.Tensor: shape=(3, 1), dtype=string, numpy=
array([[b'Open'],
       [b'Read'],
       [b'Map']], dtype=object)>, <tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[-1.0000000e+00, -1.0000000e+00],
       [ 4.9066844e+05,  2.3807518e-06],
       [ 4.9066847e+05,  1.1549212e-02]], dtype=float32)>, <tf.Tensor: shape=(3, 3), dty
pe=int32, numpy=
array([[ 1,  0, -1],
       [ 1,  0,  1],
       [ 1,  0,  1]], dtype=int32)>, <tf.Tensor: shape=(24, 24, 3), dtype=float32, numpy
=
array([[[ 0.47468224,  0.75477153,  0.97282755],
        [ 0.60898346,  0.90586287,  1.0399684 ],
        [-0.0289228 ,  0.1840257 ,  0.06633849],
        ...,
        [-1.1200638 , -1.1253278 , -1.3437556 ],
        [-0.83467126, -0.82316476, -1.142294  ],
        [-0.8850416 , -0.83995485, -1.3605261 ]],

       [[ 0.5082624 ,  0.72121096,  0.8049462 ],
        [ 0.5754032 ,  0.73800105,  0.73780537],
        [-0.16320443, -0.10134721, -0.25261462],
        ...,
        [-1.0025526 , -0.9910462 , -1.2765952 ],
        [-0.86825144, -0.83995485, -1.2094544 ],
        [-0.8011107 , -0.77281404, -1.2430346 ]],

       [[ 0.44110203,  0.6540505 ,  0.63708436],
        [ 0.5754032 ,  0.73800105,  0.67066455],
        [ 0.45789215,  0.5197689 ,  0.3852916 ],
        ...,
        [-1.0193232 , -0.9910462 , -1.3773162 ],
        [-0.9018317 , -0.8735351 , -1.2933853 ],
        [-1.0864836 , -1.058187  , -1.4780372 ]],

       ...,

       [[-1.3886466 , -1.461071  , -1.3101754 ],
        [-1.4893676 , -1.4946316 , -1.4612471 ],
        [-0.33108583, -0.11813731, -0.2694047 ],
        ...,
        [ 1.1629392 ,  0.8722827 , -0.05117261],
        [ 1.1293786 ,  0.82193196, -0.10152333],
        [ 1.146149  ,  0.7883518 , -0.16868371]],

       [[-1.3718565 , -1.4442809 , -1.2933853 ],
        [-1.3550664 , -1.37714   , -1.3773162 ],
```

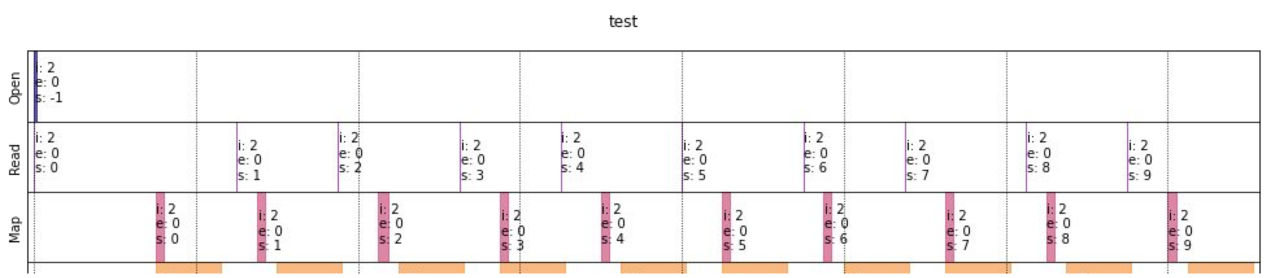
```
         [-0.41501674, -0.28599912, -0.48761725],
         ...,
         [ 0.35717112,  0.1000948 , -0.58833826],
         [ 0.5082624 ,  0.2679566 , -0.45405665],
         [ 0.7600551 ,  0.45260853, -0.3701257 ]],

        [[-1.7915306 , -1.847165  , -1.74662   ],
         [-1.7915306 , -1.8303748 , -1.7802002 ],
         [-1.5732985 , -1.561792  , -1.5955483 ],
         ...,
         [-1.5229282 , -1.7967947 , -2.0823634 ],
         [-1.4725775 , -1.6793032 , -1.9648522 ],
         [-1.2543454 , -1.4946316 , -1.8809212 ]]], dtype=float32)>, <tf.Tensor: shape=(1
0,), dtype=float32, numpy=array([0., 0., 0., 0., 0., 0., 0., 0., 0., 1.], dtype=float32)
>)
now time 490668.499416673
-------------------------------------------------------------------------------
```

After map function (and since we do not apply `shuffle()`, in first block above, the image is mapped from original [59., 43., 50., ..., 140., 84., 72.] shape=(3072,) to [[[0.794585 , 0.1525154 , -0.43435538], ...]] shape=(24,24,3), the original label 6 is now mapped to [0., 0., 0., 0., 0., 0., 1., 0., 0., 0.].

In [54]:
```python
steps_acc = tf.zeros([0, 1], dtype=tf.dtypes.string)
times_acc = tf.zeros([0, 2], dtype=tf.dtypes.float32)
values_acc = tf.zeros([0, 3], dtype=tf.dtypes.int32)

start_time = time.perf_counter()
print("start time: ", start_time)

for steps, times, values, image, label in tqdm(tf.data.Dataset.range(1).flat_map(datase
    steps_acc = tf.concat((steps_acc, steps), axis=0)
    times_acc = tf.concat((times_acc, times), axis=0)
    values_acc = tf.concat((values_acc, values), axis=0)

    # simulate training time
    train_enter = time.perf_counter()
    time.sleep(0.1)
    train_elapsed = time.perf_counter() - train_enter

    steps_acc = tf.concat((steps_acc, [["Train"]]), axis=0)
    times_acc = tf.concat((times_acc, [(train_enter, train_elapsed)]), axis=0)
    values_acc = tf.concat((values_acc, [values[-1]]), axis=0)

tf.print("Execution time:", time.perf_counter() - start_time)
timeline = {"steps": steps_acc, "times": times_acc, "values": values_acc}
```

```
start time:  490668.527553598
10it [00:01,  5.19it/s]
Execution time: 1.975233745004516
```

In [55]:
```python
from lab11_1_lib import draw_timeline

draw_timeline(timeline=timeline, title="test", min_width=1, annotate=True)
```

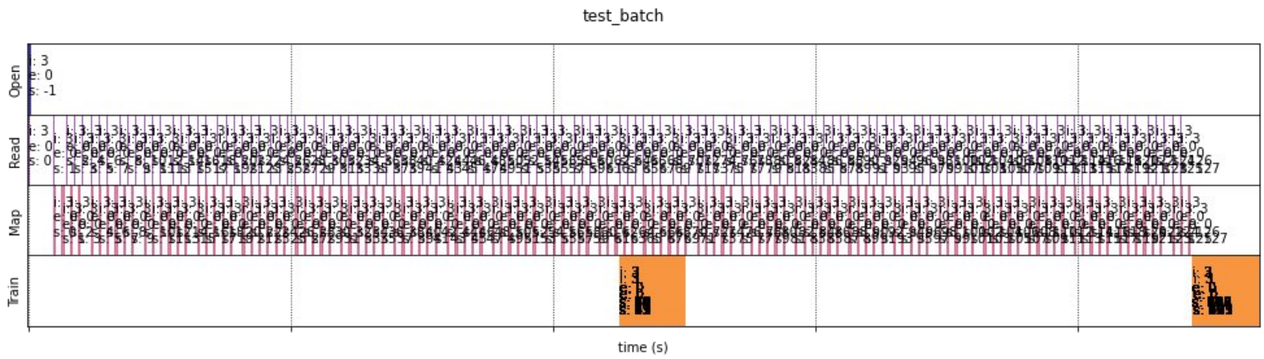


or in batch:

In [56]:
```
steps_acc = tf.zeros([0, 1], dtype=tf.dtypes.string)
times_acc = tf.zeros([0, 2], dtype=tf.dtypes.float32)
values_acc = tf.zeros([0, 3], dtype=tf.dtypes.int32)

start_time = time.perf_counter()
print("start time: ", start_time)

for steps, times, values, image, label in tqdm(tf.data.Dataset.range(1).flat_map(datase
    '''
        steps: shape=(64, 3, 1)
        times: shape=(64, 3, 2)
        vaules: shape=(64, 3, 3)
        image: shape=(64, 24, 24, 3)
        label: shape=(64, 10)
    '''

    steps_acc = tf.concat([steps_acc, tf.reshape(steps, (steps.shape[0]*steps.shape[1],
    times_acc = tf.concat([times_acc, tf.reshape(times, (times.shape[0]*times.shape[1],
    values_acc = tf.concat([values_acc, tf.reshape(values, (values.shape[0]*values.shap

    train_enter = time.perf_counter()
    time.sleep(0.5) # simulate training time
    train_elapsed = time.perf_counter() - train_enter

    train_time = tf.concat([tf.fill([times.shape[0], 1], train_enter), tf.fill([times.s

    steps_acc = tf.concat([steps_acc, tf.fill([steps.shape[0], 1], "Train")], axis=0)
    times_acc = tf.concat([times_acc, train_time], axis=0)
    values_acc = tf.concat([values_acc, values[:,-1,:]], axis=0)

tf.print("Execution time:", time.perf_counter() - start_time)
timeline = {"steps": steps_acc, "times": times_acc, "values": values_acc}
```

```
start time:  490670.882653565
2it [00:09,  4.73s/it]
Execution time: 9.501270917011425
```

In [57]:
```
draw_timeline(timeline=timeline, title="test_batch", min_width=1, annotate=True)
```



The annotatation is quite unreadable though, we still can roughly find that example_idx in **Read**, **Map**, **Train** all run through from 0 to 127.

BTW, `min_width` in `draw_timeline()` indicate the minimum time duration of the graph. Since `draw_timeline()` will apply `max()` to `min_width` and total execution time to decide the final time duration of graph, if you set a small value of `min_width`, the final time duration of graph will be the total execution time.

Re-train CNN with time

In [58]:

```
# re-construct a same model
model_cifar_op = models.Sequential()

model_cifar_op.add(layers.Conv2D(64, (5, 5), padding='same', activation='relu', input_s
model_cifar_op.add(layers.MaxPool2D(pool_size=3,strides=2,padding='same'))
model_cifar_op.add(layers.BatchNormalization())

model_cifar_op.add(layers.Conv2D(64, (5, 5), padding='same', activation='relu'))
model_cifar_op.add(layers.MaxPool2D(pool_size=3,strides=2,padding='same'))
model_cifar_op.add(layers.BatchNormalization())

model_cifar_op.add(layers.Flatten())
model_cifar_op.add(layers.Dense(384, activation='relu'))
model_cifar_op.add(layers.Dropout(0.5))
model_cifar_op.add(layers.Dense(192, activation='relu'))
model_cifar_op.add(layers.Dropout(0.5))
model_cifar_op.add(layers.Dense(10, activation='softmax'))
model_cifar_op.build()
model_cifar_op.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 conv2d_7 (Conv2D)           (None, 24, 24, 64)        4864      
                                                                 
 max_pooling2d_6 (MaxPooling  (None, 12, 12, 64)       0         
 2D)                                                             
                                                                 
 batch_normalization_4 (Batc  (None, 12, 12, 64)       256       
 hNormalization)                                                 
                                                                 
 conv2d_8 (Conv2D)           (None, 12, 12, 64)        102464    
                                                                 
 max_pooling2d_7 (MaxPooling  (None, 6, 6, 64)         0         
 2D)                                                             
                                                                 
 batch_normalization_5 (Batc  (None, 6, 6, 64)         256       
 hNormalization)                                                 
                                                                 
 flatten_3 (Flatten)         (None, 2304)              0         
                                                                 
 dense_9 (Dense)             (None, 384)               885120    
                                                                 
 dropout_4 (Dropout)         (None, 384)               0         
                                                                 
 dense_10 (Dense)            (None, 192)               73920     
                                                                 
 dropout_5 (Dropout)         (None, 192)               0         
                                                                 
 dense_11 (Dense)            (None, 10)                1930      
                                                                 
=================================================================
Total params: 1,068,810
Trainable params: 1,068,554
Non-trainable params: 256
_________________________________________________________________
```

In [59]:

```
# save the initialization of weights
```

```
model_cifar_op.save_weights('model_cifar_op.h5')
```

In [60]:
```
# define loss and optimizer
loss_object = tf.keras.losses.CategoricalCrossentropy(from_logits=True)
optimizer = tf.keras.optimizers.Adam()

train_loss = tf.keras.metrics.Mean(name='train_loss')
train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='train_accuracy')

test_loss = tf.keras.metrics.Mean(name='test_loss')
test_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='test_accuracy')
```

In [61]:
```
@tf.function
def train_step(image, label):
    with tf.GradientTape() as tape:
        predictions = model_cifar_op(image, training=True)
        loss = loss_object(label, predictions)
    gradients = tape.gradient(loss, model_cifar_op.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model_cifar_op.trainable_variables))

    train_loss(loss)
    train_accuracy(tf.argmax(label,axis=1), predictions)

@tf.function
def test_step(image, label):
    predictions = model_cifar_op(image, training=False)
    t_loss = loss_object(label, predictions)

    test_loss(t_loss)
    test_accuracy(tf.argmax(label,axis=1), predictions)
```

In [62]:
```
def timelined_benchmark(dataset_train, dataset_test, EPOCHS):
    steps_acc = tf.zeros([0, 1], dtype=tf.dtypes.string)
    times_acc = tf.zeros([0, 2], dtype=tf.dtypes.float32)
    values_acc = tf.zeros([0, 3], dtype=tf.dtypes.int32)

    start_time = time.perf_counter()
    print("start time: ", start_time)
    for epoch in range(EPOCHS):
        epoch_enter = time.perf_counter()

        # Reset the metrics at the start of the next epoch
        train_loss.reset_states()
        train_accuracy.reset_states()
        test_loss.reset_states()
        test_accuracy.reset_states()

        tf.print("training:")
        for steps, times, values, image, label in tqdm(dataset_train, total=math.floor(
            # sleep to avoid concurrency issue
            time.sleep(0.05)

            steps_acc = tf.concat([steps_acc, tf.reshape(steps, (steps.shape[0]*steps.s
            times_acc = tf.concat([times_acc, tf.reshape(times, (times.shape[0]*times.s
            values_acc = tf.concat([values_acc, tf.reshape(values, (values.shape[0]*val

            # record training time
            train_enter = time.perf_counter()
            train_step(image, label)
            train_elapsed = time.perf_counter() - train_enter

            # sleep to avoid concurrency issue
```

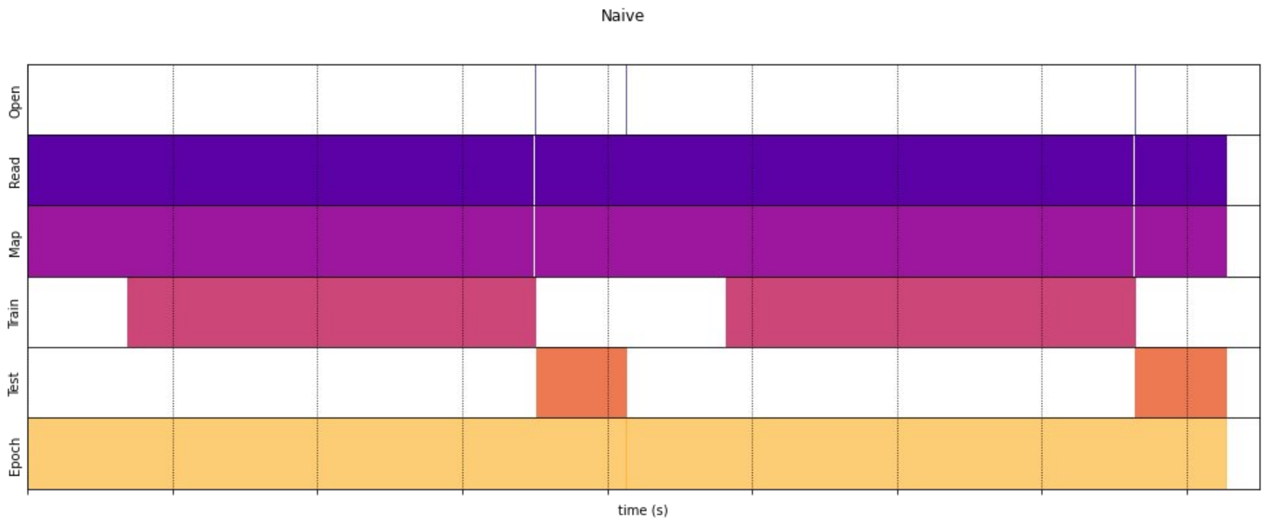
```
            time.sleep(0.05)

            train_time = tf.concat([tf.fill([times.shape[0], 1], train_enter), tf.fill(
            steps_acc = tf.concat([steps_acc, tf.fill([steps.shape[0], 1], "Train")], a
            times_acc = tf.concat([times_acc, train_time], axis=0)
            values_acc = tf.concat([values_acc, values[:,-1,:]], axis=0)

        tf.print("testing:")
        for steps, times, values, image, label in tqdm(dataset_test, total=math.floor(1
            # sleep to avoid concurrency issue
            time.sleep(0.05)

            steps_acc = tf.concat([steps_acc, tf.reshape(steps, (steps.shape[0]*steps.s
            times_acc = tf.concat([times_acc, tf.reshape(times, (times.shape[0]*times.s
            values_acc = tf.concat([values_acc, tf.reshape(values, (values.shape[0]*val

            test_enter = time.perf_counter()
            test_step(image, label)
            test_elapsed = time.perf_counter() - test_enter

            # sleep to avoid concurrency issue
            time.sleep(0.05)

            test_time = tf.concat([tf.fill([times.shape[0], 1], test_enter), tf.fill([t
            steps_acc = tf.concat([steps_acc, tf.fill([steps.shape[0], 1], "Test")], ax
            times_acc = tf.concat([times_acc, test_time], axis=0)
            values_acc = tf.concat([values_acc, values[:,-1,:]], axis=0)

        template = 'Epoch {:0}, Loss: {:.4f}, Accuracy: {:.4f}, test Loss: {:.4f}, test
        tf.print (template.format(epoch+1,
                               train_loss.result(),
                               train_accuracy.result()*100,
                               test_loss.result(),
                               test_accuracy.result()*100))

        epoch_elapsed = time.perf_counter() - epoch_enter
        steps_acc = tf.concat([steps_acc, [["Epoch"]]], axis=0)
        times_acc = tf.concat([times_acc, [(epoch_enter, epoch_elapsed)]], axis=0)
        values_acc = tf.concat([values_acc, [[-1, epoch, -1]]], axis=0)

    tf.print("Execution time:", time.perf_counter() - start_time)
    return {"steps": steps_acc, "times": times_acc, "values": values_acc}
```

Here we only train 2 epoch since we are not pursuing performace but running experiments about better data pipeline (shorter time cost).

In [64]:
```
# feel free to modify these two Settings.
BUFFER_SIZE = 10000
BATCH_SIZE = 64

# Construct training Dataset with similar steps
dataset_train = tf.data.Dataset.range(1).flat_map(dataset_generator_fun_train)\
                                        .map(map_fun_with_time)\
                                        .shuffle(10000)\
                                        .batch(BATCH_SIZE, drop_remainder=True)

# Construct testing Dataset with similar steps
dataset_test = tf.data.Dataset.range(1).flat_map(dataset_generator_fun_test)\
                                       .map(map_fun_test_with_time)\
                                       .batch(BATCH_SIZE, drop_remainder=True)

timeline_Naive = timelined_benchmark(dataset_train, dataset_test, EPOCHS=2)
```

```
start time:  492394.487955699
training:
100%|██████████████████████████████████████████████████| 781/781 [58:23<00:00,  4.49s/it]
testing:
100%|██████████████████████████████████████████████████| 156/156 [10:30<00:00,  4.04s/it]
Epoch 1, Loss: 1.7187, Accuracy: 37.2259, test Loss: 1.4126, test Accuracy: 49.8698
training:
100%|██████████████████████████████████████████████████| 781/781 [58:25<00:00,  4.49s/it]
testing:
100%|██████████████████████████████████████████████████| 156/156 [10:28<00:00,  4.03s/it]
Epoch 2, Loss: 1.4753, Accuracy: 47.3712, test Loss: 1.2083, test Accuracy: 57.8125
Execution time: 8268.120533703011
```

In [65]:
```
draw_timeline(timeline=timeline_Naive, title="Naive", min_width=8500)
```



optimization dataset pipeline

The dataset pipeline of `(dataset_train, dataset_test)` is same to the CNN Model for CIFAR 10 part. However, if we optimize the pipeline as below, the performance would be better. The optimization is including:

1. prefetching: overlaps the preprocessing and model execution of a training step.
2. Interleave (Parallelizing data extraction): parallelize the data loading step, interleaving the contents of other datasets (such as data file readers).
3. Parallel mapping: parallelized mapping across multiple CPU cores.
4. Caching: cache a dataset, save some operations (like file opening and data reading) from being executed during each epoch.
5. Vectorizing mapping: batch before map, so that mapping can be vectorized.

We won't explain each of them in detail. It's recommended to study the terms above in the official documentation. Here we only demonstrate the improvement.

Note that since we are vectorizing map function, there's one more dimension for batch in each inputs when mapping. Therefore, we have to modify map function first:

In [66]:
```
@map_decorator
def map_fun_with_time_batchwise(steps, times, values, image, label):
    # sleep to avoid concurrency issue
    time.sleep(0.05)

    map_enter = time.perf_counter()
```

```
    image = tf.reshape(image,[tf.shape(image)[0], IMAGE_DEPTH, IMAGE_HEIGHT, IMAGE_WIDT
    image = tf.divide(tf.cast(tf.transpose(image,[0, 2, 3, 1]),tf.float32),255.0)
    label = tf.one_hot(label, 10)
    distorted_image = tf.image.random_crop(image, [tf.shape(image)[0], IMAGE_SIZE_CROPP
    distorted_image = tf.image.random_flip_left_right(distorted_image)
    distorted_image = tf.image.random_brightness(distorted_image, max_delta=63)
    distorted_image = tf.image.random_contrast(distorted_image, lower=0.2, upper=1.8)
    distorted_image = tf.image.per_image_standardization(distorted_image)

    map_elapsed = time.perf_counter() - map_enter

    return tf.concat((steps, tf.tile([[["Map"]]], [BATCH_SIZE, 1, 1])), axis=1),\
           tf.concat((times, tf.tile([[[map_enter, map_elapsed]]], [BATCH_SIZE, 1, 1]))
           tf.concat((values, tf.tile([[values[:][-1][0]]], [BATCH_SIZE, 1, 1])), axis=
           distorted_image,\
           label

@map_decorator
def map_fun_test_with_time_batchwise(steps, times, values, image, label):
    # sleep to avoid concurrency issue
    time.sleep(0.05)

    map_enter = time.perf_counter()

    image = tf.reshape(image,[tf.shape(image)[0],IMAGE_DEPTH,IMAGE_HEIGHT,IMAGE_WIDTH])
    image = tf.divide(tf.cast(tf.transpose(image,[0, 2, 3, 1]),tf.float32),255.0)
    label = tf.one_hot(label,10)
    distorted_image = tf.image.resize(image, [IMAGE_SIZE_CROPPED,IMAGE_SIZE_CROPPED])
    distorted_image = tf.image.per_image_standardization(distorted_image)

    map_elapsed = time.perf_counter() - map_enter

    return tf.concat((steps, tf.tile([[["Map"]]], [BATCH_SIZE, 1, 1])), axis=1),\
           tf.concat((times, tf.tile([[[map_enter, map_elapsed]]], [BATCH_SIZE, 1, 1]))
           tf.concat((values, tf.tile([[values[:][-1][0]]], [BATCH_SIZE, 1, 1])), axis=
           distorted_image,\
           label
```

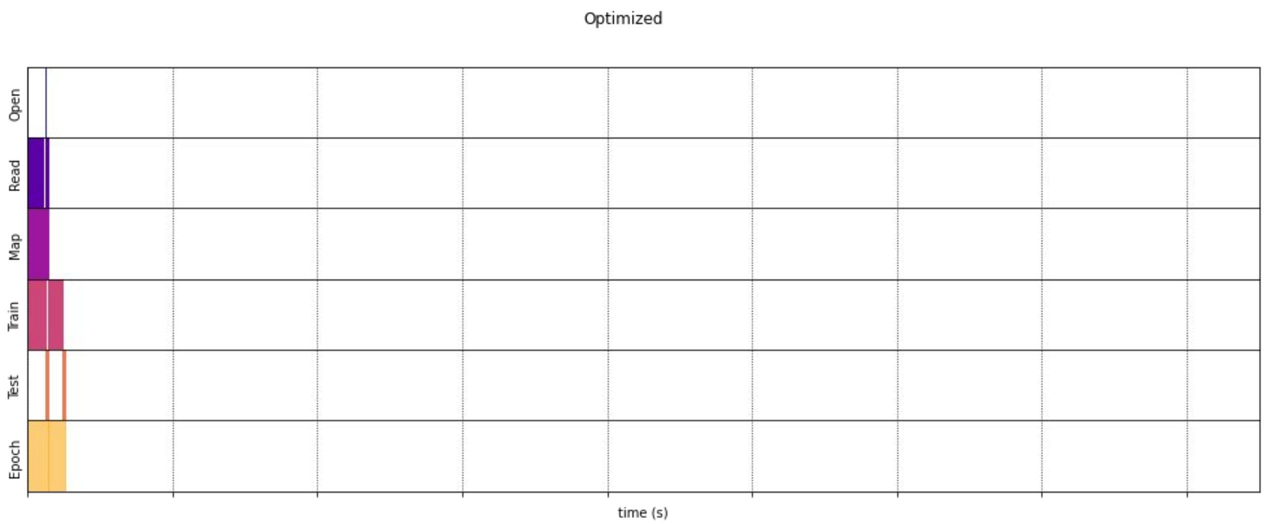
In [69]:

```
dataset_train_optimized = tf.data.Dataset.range(1).interleave(dataset_generator_fun_tra
                                                   .shuffle(BUFFER_SIZE)\
                                                   .batch(BATCH_SIZE, drop_remainder=Tru
                                                   .map(map_fun_with_time_batchwise, num
                                                   .cache()\
                                                   .prefetch(tf.data.AUTOTUNE)
dataset_test_optimized = tf.data.Dataset.range(1).interleave(dataset_generator_fun_test
                                                  .batch(BATCH_SIZE, drop_remainder=True
                                                  .map(map_fun_test_with_time_batchwise,
                                                  .cache()\
                                                  .prefetch(tf.data.AUTOTUNE)

# Load the same initialization of weights and re-train with optimized input pipeline
model_cifar_op.load_weights('model_cifar_op.h5')
timeline_Optimized = timelined_benchmark(dataset_train_optimized, dataset_test_optimize
```

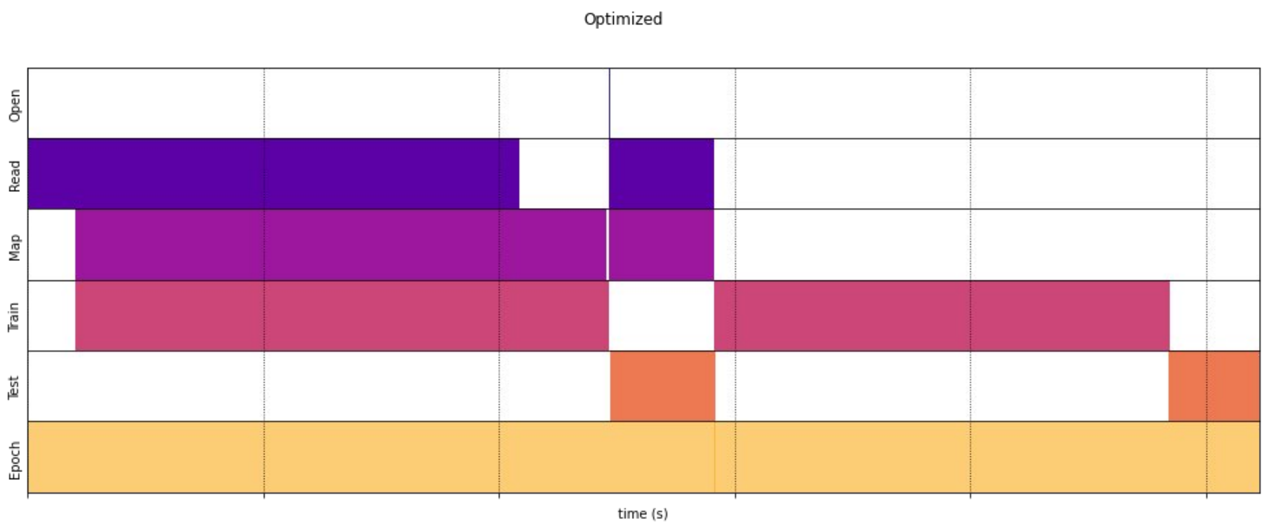
```
start time:  505283.709143188
training:
100%|██████████████████████████████████████████| 781/781 [02:03<00:00,  6.34it/s]
testing:
100%|██████████████████████████████████████████| 156/156 [00:22<00:00,  6.95it/s]
Epoch 1, Loss: 1.9210, Accuracy: 30.6198, test Loss: 1.5673, test Accuracy: 42.1174
training:
100%|██████████████████████████████████████████| 781/781 [01:36<00:00,  8.10it/s]
```

```
testing:
100%|██████████████████████████████| 156/156 [00:19<00:00,  8.11it/s]
Epoch 2, Loss: 1.5617, Accuracy: 43.8200, test Loss: 1.2972, test Accuracy: 55.3185
Execution time: 261.2892151809647
```

In [70]:
```python
draw_timeline(timeline_Optimized, "Optimized", min_width=8500)
```



In [71]:
```python
draw_timeline(timeline_Optimized, "Optimized", min_width=1)
```



From the results above, we can find that the time comsuming reduces from 8268 to 261 (sec) but get close accuracy. There's exactly no **Open**, **Read** and **Map** time consuming in 2nd epoch (which is because of the Caching). Besides, the training and testing time in 2nd epoch also decrease.

In this lab, we study how to optimize the data pipeling (I/O). The result is great though, the result is highly **depended on device**. If you re-run the exactly same code above on your device, you may get totally different result (if the bottleneck on your device is the training speed, not I/O). Besides, the data type may also affect the result. Here we read image from `.pkl` files, which is an binary file with faster I/O speed. If we switch the situation like reading image from `.jpg` / `.png` files (what you would do in the assignment below), the imporvement would be even evident.

in practical use (a simple demo)

The code above is complicate because we have to combine time into dataset. In practical, the usage may look like:

```python
# if files have been opened and read into memory
dataset_train_example = tf.data.Dataset.from_tensor_slices((image, label))\
                                       .shuffle(BUFFER_SIZE)\
                                       .batch(BATCH_SIZE, drop_remainder=True)\
                                       .map(map_fun_batchwise, num_parallel_calls=tf.da
                                       .cache()\
                                       .prefetch(tf.data.AUTOTUNE)

# or giving file path with `ImageDataGenerator()`, e.g.
flowers_file_path = tf.keras.utils.get_file('flower_photos', 'https://storage.googleapi
img_gen = tf.keras.preprocessing.image.ImageDataGenerator(rescale=1./255, rotation_rang
dataset_train_example = tf.data.Dataset.from_generator(
                            lambda: img_gen.flow_from_directory(flowers_file_path),
                            output_types=(tf.float32, tf.float32),
                            output_shapes=([32,256,256,3], [32,5])
                        )
dataset_train_example = dataset_train_example.shuffle(BUFFER_SIZE)\
                                             .batch(BATCH_SIZE, drop_remainder=True)\
                                             .map(map_fun_batchwise, num_parallel_calls
                                             .cache()\
                                             .prefetch(tf.data.AUTOTUNE)

# or tf.data.Dataset.list_files()
dataset_train_example = tf.data.Dataset.list_files(flowers_file_path+"/*/*.jpg")\
                                       .shuffle(BUFFER_SIZE)\
                                       .batch(BATCH_SIZE, drop_remainder=True)\
                                       .map(map_fun_batchwise, num_parallel_calls=tf.da
                                       .cache()\
                                       .prefetch(tf.data.AUTOTUNE)
```

`interleave()` is rarely used in my experience. Also remember that `map_fun_batchwise()` should include `@tf.function` decorator for AutoGraph speed up.

Assignment

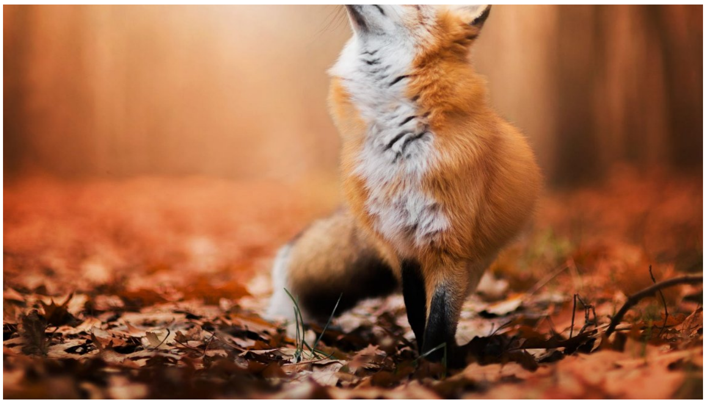
In this assignment, you have to implement the input pipeline of the CNN model and try to write/read tfrecord with the **Oregon Wildlife** dataset.

We provide you with the complete code for the image classification task of the CNN model, but remove the part of the input pipeline. What you need to do is completing this part and training the model for at least 5 epochs.

Description of Dataset:

1. The raw data is from kaggle, which consists of 20 class image of wildlife.
2. We have filtered the raw data. You need to download the filtered image from here and use them to complete the image classification task.
3. In the dataset we prepared for you, there are nearly 7,200 image, which contain **10 kinds** of wildlife.

The sample image is shown below:



red_fox

Requirement:

- Try some the input transfromation mentioned above (e.g. shuffle, batch, repeat, map(random_crop, random_flip_left_right, ...)) but without optimization terms (e.g. prefetch, cache, num_parallel_calls)
  - Compare the performance and time consumption to the Naive one.
  - Note that it's important to take some examples and plot the image like block [84] to make sure theat the map function does as you desired
- Retrain your model with optimized terms, comparing the performance and time consumption.
- Training both models above for at least 3 epochs.
- Briefly summarize what you did and explain the performance results (accuracy and time consuming).
  - It's fine if you get worse performace after applying input transfromation, or get longer time consumption after applying data pipeline optimization. Just try to analyze the result and propose your assumption, e.g.
    - Is it possible that the more difficult training data after input transfromation may need more epochs to train to get better performance?
    - Which step is the bottleneck on your device? Why? Any other steps cost more time than your expect?

Note:

The `time.sleep(0.05)` in the example is to avoid concurrency issues that TAs are unable to solve at short notice. However, the duration **depends on devices** (the throughput between CPU and GPU maybe). For example, in our lab servers, 0.05 is enough for one newer computer while another computer still sometimes meet the error even we increase to 0.1. Therefore, if you meet strange errors like below and **not always meets the error when re-run the same code**, setting higher sleep time may help though it's slower. Errors that TA meets:

- `Expected size[1] in [0, 101], but got 224 [Op:Slice]` when `tf.image.random_crop(image, [tf.shape(image)[0],IMAGE_SIZE_CROPPED,IMAGE_SIZE_CROPPED,IMAGE_DEPTH])`
- `ConcatOp : Dimension 0 in both shapes must be equal: shape[0] = [196,3] vs. shape[1] = [3,3] [Op:ConcatV2] name: concat` when `tf.concat([values_acc, tf.reshape(values, (values.shape[0]*values.shape[1], 3))], axis=0)`

- `Expected multiples argument to be a vector of length 1 but got length 3 [Op:Tile]` when `values_acc = tf.concat([values_acc, values[:,-1,:]], axis=0)`

Notification:

- Submit to **eeclass** with your ipynb (Lab11-1_{student_id}.ipynb)
- Deadline: 2023-11-9 (Thr) 23:59

In [72]:

```
import os
import warnings
warnings.filterwarnings("ignore")
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

import tensorflow as tf
from tensorflow.keras import utils, datasets, layers, models
from tensorflow.keras.applications.vgg16 import VGG16
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split
import IPython.display as display
import matplotlib.pyplot as plt
import pathlib
import random
import numpy as np
import matplotlib as mpl
import time

import csv
import pandas as pd
import math
from tqdm import tqdm

import itertools
from collections import defaultdict
```

In [73]:

```
# You need to download the prepared data and unzip the file in current path('./')
data_root = pathlib.Path('./oregon_wildlife')

# print the subfolders.
print('classes:')
for item in data_root.iterdir():
    print(item)

all_image_paths = list(data_root.glob('*/*'))
all_image_paths = [str(path) for path in all_image_paths]
all_image_paths = shuffle(all_image_paths, random_state=1)
all_image_paths = [path for path in all_image_paths if path[-3:] not in ('gif','bmp')]
image_count = len(all_image_paths)
print('\ntotal img num:', image_count)
```

```
classes:
oregon_wildlife/bald_eagle
oregon_wildlife/black_bear
oregon_wildlife/cougar
oregon_wildlife/deer
oregon_wildlife/nutria
oregon_wildlife/raccoon
oregon_wildlife/raven
oregon_wildlife/red_fox
oregon_wildlife/sea_lions
oregon_wildlife/virginia_opossum

total img num: 7168
```

In [74]:
```
# random showing 3 iamges for you
for n in range(3):
    image_path = random.choice(all_image_paths)
    display.display(display.Image(image_path, width=200, height=200))
    print(image_path.split('/')[-2])
```

```
raven
```

```
nutria
```

```
raven
```

In [75]:
```
# get the label
label_names = sorted(item.name for item in data_root.glob('*/') if item.is_dir())
# total label
n_classes = len(label_names)
print(label_names)
```

```
['bald_eagle', 'black_bear', 'cougar', 'deer', 'nutria', 'raccoon', 'raven', 'red_fox',
'sea_lions', 'virginia_opossum']
```

In [76]:
```
# get the mapping dict
label_to_index = dict((name, index) for index,name in enumerate(label_names))
index_to_label = dict((index, name) for index,name in enumerate(label_names))
print(label_to_index)
```

```
{'bald_eagle': 0, 'black_bear': 1, 'cougar': 2, 'deer': 3, 'nutria': 4, 'raccoon': 5, 'r
aven': 6, 'red_fox': 7, 'sea_lions': 8, 'virginia_opossum': 9}
```

In [77]:
```
# get the label data
all_image_label = [label_to_index[pathlib.Path(path).parent.name] for path in all_image
```

```python
print("First 10 label indices: ", all_image_label[:10])
```

```
First 10 label indices:  [8, 5, 2, 5, 0, 2, 1, 0, 1, 0]
```

In [78]:
```python
# Create training and testing sets using an 80-20 split
img_path_train, img_path_test, label_train, label_test = train_test_split(all_image_pat
                                all_image_label,test_size=0.2,random_state=0)
print('training data: %d'%(len(img_path_train)))
print('testing data: %d'%(len(img_path_test)))
```

```
training data: 5734
testing data: 1434
```

In [79]:
```python
# save (img_path, label) pairs
with open('train.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(['img_path', 'label'])
    for img_path, label in zip(img_path_train, label_train):
        writer.writerow([img_path, label])

with open('test.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(['img_path', 'label'])
    for img_path, label in zip(img_path_test, label_test):
        writer.writerow([img_path, label])
```

In [80]:
```python
# Feel free to change IMAGE_SIZE_CROPPED if using random_crop in your data augmentation
IMAGE_SIZE_CROPPED = 224
IMAGE_HEIGHT = 300
IMAGE_WIDTH = 300
IMAGE_DEPTH = 3
```

In [81]:
```python
# construct a new dataset with time informantion
class TimeMeasuredDataset(tf.data.Dataset):
    # OUTPUT: (steps, timings, counters, img, label)
    OUTPUT_SIGNATURE=(
        tf.TensorSpec(shape=(2, 1), dtype=tf.string), # steps: [("Open",), ("Read",)]
        tf.TensorSpec(shape=(2, 2), dtype=tf.float32), # timings: [(open_enter, open_el
        tf.TensorSpec(shape=(2, 3), dtype=tf.int32), # counters: [(instance_idx, epoch_
        tf.TensorSpec(shape=(300,300,3), dtype=tf.float32),
        tf.TensorSpec(shape=(), dtype=tf.int32) # label
    )

    _INSTANCES_COUNTER = itertools.count()  # Number of datasets generated
    _EPOCHS_COUNTER = defaultdict(itertools.count)  # Number of epochs done for each da

    def _generator(instance_idx, filename, open_file, read_file):
        epoch_idx = next(TimeMeasuredDataset._EPOCHS_COUNTER[instance_idx])

        # Opening the file
        open_enter = time.perf_counter()
        img_paths, label = open_file(filename)
        open_elapsed = time.perf_counter() - open_enter
        # ----------------

        # Reading the file
        for sample_idx in range(len(img_paths)):
            # Reading data (line, record) from the file
            read_enter = time.perf_counter()
            img = read_file(img_paths[sample_idx])
            read_elapsed = time.perf_counter() - read_enter

            yield (
```

```
                [("Open",), ("Read",)],
                [(open_enter, open_elapsed), (read_enter, read_elapsed)],
                [(instance_idx, epoch_idx, -1), (instance_idx, epoch_idx, sample_idx)],
                img,
                label[sample_idx]
            )
            open_enter, open_elapsed = -1., -1.  # Negative values will be filtered


    def __new__(cls, filename, open_file, read_file):
        def generator_func(instance_idx, filename):
            return cls._generator(instance_idx, filename, open_file, read_file)

        return tf.data.Dataset.from_generator(
            generator_func,
            output_signature=cls.OUTPUT_SIGNATURE,
            args=(next(cls._INSTANCES_COUNTER), filename)
        )
```

In [82]:

```
def open_file(filename):
    rows = pd.read_csv(filename.decode("utf-8"))
    img_paths = rows['img_path'].tolist()
    label = rows['label'].tolist()
    return img_paths, label

def read_file(image_path):
    img = tf.io.read_file(image_path)
    img = tf.image.decode_jpeg(img, channels=IMAGE_DEPTH)
    img = tf.image.resize(img, (IMAGE_HEIGHT, IMAGE_WIDTH))
    img = tf.cast(img, tf.float32)
    img = tf.divide(img,255.0)
    return img

def dataset_generator_fun_train(*args):
    return TimeMeasuredDataset('train.csv', open_file, read_file)

def dataset_generator_fun_test(*args):
    return TimeMeasuredDataset('test.csv', open_file, read_file)
```

In [83]:

```
# feel free to modify these two Settings.
BUFFER_SIZE = 10000
BATCH_SIZE = 1

dataset_train = tf.data.Dataset.range(1).flat_map(dataset_generator_fun_train).batch(BA
dataset_test = tf.data.Dataset.range(1).flat_map(dataset_generator_fun_test).batch(BATC
```

In [84]:

```
for steps, timings, counters, img, label in dataset_train.take(1):
    print(steps[0], timings[0], counters[0])
    print(img[0].shape)
    plt.imshow(img[0])
    plt.axis('off')
    plt.show()
    print(index_to_label[label[0].numpy()])
```

```
tf.Tensor(
[[b'Open']
 [b'Read']], shape=(2, 1), dtype=string) tf.Tensor(
[[5.0564150e+05 1.3395692e-02]
 [5.0564150e+05 3.0839540e-02]], shape=(2, 2), dtype=float32) tf.Tensor(
[[ 0  0 -1]
 [ 0  0  0]], shape=(2, 3), dtype=int32)
(300, 300, 3)
```

bald_eagle

In [85]:
```
base_model = VGG16(
    include_top=False,
    weights='imagenet',
    input_shape=(300, 300, 3),
    pooling=None,
)
for layer in base_model.layers:
    layer.trainable = False

top_model = models.Sequential()
top_model.add(layers.Flatten())
top_model.add(layers.Dense(4096, activation='relu'))
top_model.add(layers.Dropout(0.5))
top_model.add(layers.Dense(1024, activation='relu'))
top_model.add(layers.Dropout(0.5))
top_model.add(layers.Dense(n_classes, activation='softmax'))

wild_model = tf.keras.Model(inputs=base_model.input, outputs=top_model(base_model.outpu
```

In [86]:
```
wild_model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_1 (InputLayer)        [(None, 300, 300, 3)]     0         
                                                                 
 block1_conv1 (Conv2D)       (None, 300, 300, 64)      1792      
                                                                 
 block1_conv2 (Conv2D)       (None, 300, 300, 64)      36928     
                                                                 
 block1_pool (MaxPooling2D)  (None, 150, 150, 64)      0         
                                                                 
 block2_conv1 (Conv2D)       (None, 150, 150, 128)     73856     
                                                                 
 block2_conv2 (Conv2D)       (None, 150, 150, 128)     147584    
                                                                 
 block2_pool (MaxPooling2D)  (None, 75, 75, 128)       0         
                                                                 
 block3_conv1 (Conv2D)       (None, 75, 75, 256)       295168    
                                                                 
 block3_conv2 (Conv2D)       (None, 75, 75, 256)       590080    
                                                                 
 block3_conv3 (Conv2D)       (None, 75, 75, 256)       590080    
                                                                 
 block3_pool (MaxPooling2D)  (None, 37, 37, 256)       0         
                                                                 
 block4_conv1 (Conv2D)       (None, 37, 37, 512)       1180160   
```

```
 block4_conv2 (Conv2D)       (None, 37, 37, 512)       2359808

 block4_conv3 (Conv2D)       (None, 37, 37, 512)       2359808

 block4_pool (MaxPooling2D)  (None, 18, 18, 512)       0

 block5_conv1 (Conv2D)       (None, 18, 18, 512)       2359808

 block5_conv2 (Conv2D)       (None, 18, 18, 512)       2359808

 block5_conv3 (Conv2D)       (None, 18, 18, 512)       2359808

 block5_pool (MaxPooling2D)  (None, 9, 9, 512)         0

 sequential_5 (Sequential)   (None, 10)                174078986

=================================================================
Total params: 188,793,674
Trainable params: 174,078,986
Non-trainable params: 14,714,688
_________________________________________________________________
```

In [87]:

```python
# save the initialization of weights
wild_model.save_weights('wild_model.h5')
```

In [88]:

```python
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
optimizer = tf.keras.optimizers.Adam()

train_loss = tf.keras.metrics.Mean(name='train_loss')
train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='train_accuracy')

test_loss = tf.keras.metrics.Mean(name='test_loss')
test_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='test_accuracy')
```

In [89]:

```python
@tf.function
def train_step(image, label):
    with tf.GradientTape() as tape:
        predictions = wild_model(image, training=True)
        loss = loss_object(label, predictions)
    gradients = tape.gradient(loss, wild_model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, wild_model.trainable_variables))

    train_loss(loss)
    train_accuracy(label, predictions)

@tf.function
def test_step(image, label):
    predictions = wild_model(image, training=False)
    loss = loss_object(label, predictions)

    test_loss(loss)
    test_accuracy(label, predictions)
```

In [90]:

```python
def timelined_benchmark(dataset_train, dataset_test, EPOCHS):
    steps_acc = tf.zeros([0, 1], dtype=tf.dtypes.string)
    times_acc = tf.zeros([0, 2], dtype=tf.dtypes.float32)
    values_acc = tf.zeros([0, 3], dtype=tf.dtypes.int32)

    start_time = time.perf_counter()
    print("start time: ", start_time)
    for epoch in range(EPOCHS):
        epoch_enter = time.perf_counter()
```

```
        # Reset the metrics at the start of the next epoch
        train_loss.reset_states()
        train_accuracy.reset_states()
        test_loss.reset_states()
        test_accuracy.reset_states()

        tf.print("training:")
        for steps, times, values, image, label in tqdm(dataset_train, total=math.floor(
            time.sleep(0.05)

            steps_acc = tf.concat([steps_acc, tf.reshape(steps, (steps.shape[0]*steps.s
            times_acc = tf.concat([times_acc, tf.reshape(times, (times.shape[0]*times.s
            values_acc = tf.concat([values_acc, tf.reshape(values, (values.shape[0]*val

            # record training time
            train_enter = time.perf_counter()
            train_step(image, label)
            train_elapsed = time.perf_counter() - train_enter

            time.sleep(0.05)

            train_time = tf.concat([tf.fill([times.shape[0], 1], train_enter), tf.fill(
            steps_acc = tf.concat([steps_acc, tf.fill([steps.shape[0], 1], "Train")], a
            times_acc = tf.concat([times_acc, train_time], axis=0)
            values_acc = tf.concat([values_acc, values[:,-1,:]], axis=0)

        tf.print("testing:")
        for steps, times, values, image, label in tqdm(dataset_test, total=math.floor(l
            time.sleep(0.05)

            steps_acc = tf.concat([steps_acc, tf.reshape(steps, (steps.shape[0]*steps.s
            times_acc = tf.concat([times_acc, tf.reshape(times, (times.shape[0]*times.s
            values_acc = tf.concat([values_acc, tf.reshape(values, (values.shape[0]*val

            test_enter = time.perf_counter()
            test_step(image, label)
            test_elapsed = time.perf_counter() - test_enter

            time.sleep(0.05)

            test_time = tf.concat([tf.fill([times.shape[0], 1], test_enter), tf.fill([t
            steps_acc = tf.concat([steps_acc, tf.fill([steps.shape[0], 1], "Test")], ax
            times_acc = tf.concat([times_acc, test_time], axis=0)
            values_acc = tf.concat([values_acc, values[:,-1,:]], axis=0)

        template = 'Epoch {:0}, Loss: {:.4f}, Accuracy: {:.4f}, test Loss: {:.4f}, test
        tf.print (template.format(epoch+1,
                               train_loss.result(),
                               train_accuracy.result()*100,
                               test_loss.result(),
                               test_accuracy.result()*100))

        epoch_elapsed = time.perf_counter() - epoch_enter
        steps_acc = tf.concat([steps_acc, [["Epoch"]]], axis=0)
        times_acc = tf.concat([times_acc, [(epoch_enter, epoch_elapsed)]], axis=0)
        values_acc = tf.concat([values_acc, [[-1, epoch, -1]]], axis=0)

    tf.print("Execution time:", time.perf_counter() - start_time)
    return {"steps": steps_acc, "times": times_acc, "values": values_acc}
```

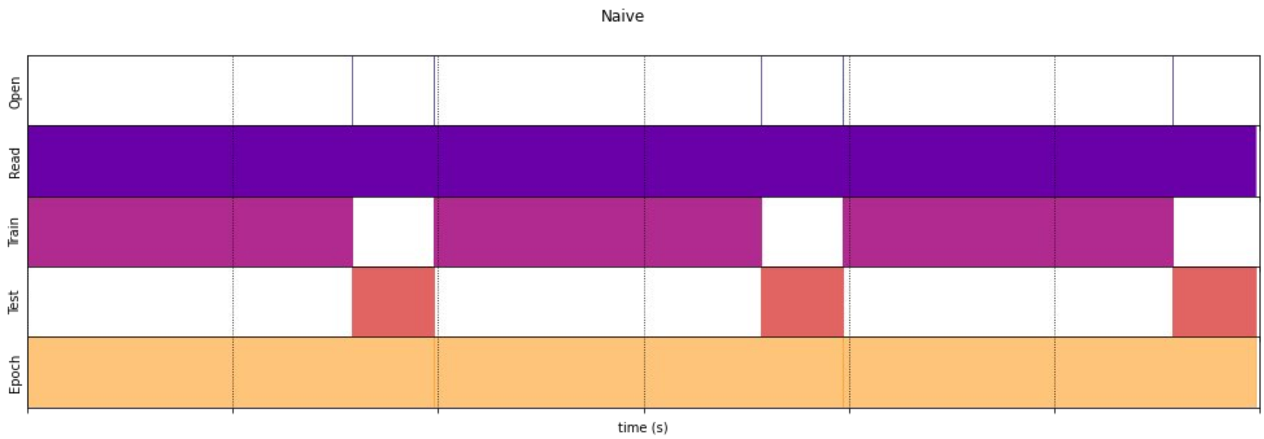
In [91]:

```
timeline_Naive = timelined_benchmark(dataset_train, dataset_test, EPOCHS=3)
```

```
start time:  505650.195122698
```

```
training:
100%|██████████████████████████████| 5734/5734 [13:09<00:00,  7.26it/s]
testing:
100%|██████████████████████████████| 1434/1434 [03:18<00:00,  7.21it/s]
Epoch 1, Loss: 4.5322, Accuracy: 10.4290, test Loss: 2.3034, test Accuracy: 10.0418
training:
100%|██████████████████████████████| 5734/5734 [13:16<00:00,  7.20it/s]
testing:
100%|██████████████████████████████| 1434/1434 [03:20<00:00,  7.15it/s]
Epoch 2, Loss: 3.0177, Accuracy: 10.3767, test Loss: 2.3295, test Accuracy: 10.0418
training:
100%|██████████████████████████████| 5734/5734 [13:22<00:00,  7.15it/s]
testing:
100%|██████████████████████████████| 1434/1434 [03:21<00:00,  7.12it/s]
Epoch 3, Loss: 2.5098, Accuracy: 10.6209, test Loss: 2.3145, test Accuracy: 9.6932
Execution time: 2989.7814084590063
```

In [92]:
```python
draw_timeline(timeline=timeline_Naive, title="Naive", min_width=3000)
```



The accuracy now is 9.69% in testing set, costing with 2989 sec. Now try some data augmentation (transformation) to observe whether the accuracy and execution time are increased or decreased.

In [ ]:
```python
## TODO: build `dataset_train_augmentation` and `dataset_test_augmentation` with transf
## Remember to define your own map functions with map_decorator before calling map

# dataset_train_augmentation = tf.data.Dataset.range(1). ...
# dataset_test_augmentation = tf.data.Dataset.range(1). ...
```

In [ ]:
```python
# load the same initialization of weights and re-train with optimized input pipeline
wild_model.load_weights('wild_model.h5')
timeline_Augmentation = timelined_benchmark(dataset_train_augmentation, dataset_test_au
```

In [ ]:
```python
draw_timeline(timeline=timeline_Augmentation, title="Augmentation", min_width=3000)
```

After trying data augmentation (transformation), it's time to optimize what you did above for better efficiency.

In [ ]:
```python
## TODO: build `dataset_train_optimized` and `dataset_test_optimized` with transformati
## Remember to re-define your own map functions again to make mapping time re-calculate

# dataset_train_optimized = tf.data.Dataset.range(1). ...
# dataset_test_optimized = tf.data.Dataset.range(1). ...
```

In [ ]:
```python
# load the same initialization of weights and re-train with optimized input pipeline
wild_model.load_weights('wild_model.h5')
```

```
timeline_Optimized = timelined_benchmark(dataset_train_optimized, dataset_test_optimize
```

In [ ]:
```
draw_timeline(timeline=timeline_Optimized, title="Optimized", min_width=3000)
```