

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Introduction to the Keras Tuner



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/keras/keras_tuner)

(https://www.tensorflow.org/tutorials/keras/keras_tuner) (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/keras/keras_tuner.ipynb) (https://github.com/tensorflow/docs/blob/master/site/en/tutorials/keras/keras_tuner.ipynb)



[Run in Google Colab](#)

Overview

The Keras Tuner is a library that helps you pick the optimal set of hyperparameters for your TensorFlow program. The process of selecting the right set of hyperparameters for your machine learning (ML) application is called *hyperparameter tuning* or *hypertuning*.

Hyperparameters are the variables that govern the training process and the topology of an ML model. These variables remain constant over the training process and directly impact the performance of your ML program. Hyperparameters are of two types:

1. **Model hyperparameters** which influence model selection such as the number and width of hidden layers
2. **Algorithm hyperparameters** which influence the speed and quality of the learning algorithm such as the learning rate for Stochastic Gradient Descent (SGD) and the number of nearest neighbors for a k Nearest Neighbors (KNN) classifier

In this tutorial, you will use the Keras Tuner to perform hypertuning for an image classification application.

Setup

In []:

```
import tensorflow as tf
from tensorflow import keras
```

Install and import the Keras Tuner.

In []:

```
!pip install -q -U keras-tuner
```

In []:

```
import keras_tuner as kt
```

Download and prepare the dataset

In this tutorial, you will use the Keras Tuner to find the best hyperparameters for a machine learning model that classifies images of clothing from the [Fashion MNIST dataset](https://github.com/zalandoresearch/fashion-mnist) (<https://github.com/zalandoresearch/fashion-mnist>).

Load the data.

In []:

```
(img_train, label_train), (img_test, label_test) = keras.datasets.fashion_mnist.load_data()
```

In []:

```
# Normalize pixel values between 0 and 1
img_train = img_train.astype('float32') / 255.0
img_test = img_test.astype('float32') / 255.0
```

Define the model

When you build a model for hypertuning, you also define the hyperparameter search space in addition to the model architecture. The model you set up for hypertuning is called a *hypermodel*.

You can define a hypermodel through two approaches:

- By using a model builder function
- By subclassing the `HyperModel` class of the Keras Tuner API

You can also use two pre-defined `HyperModel` classes - [HyperXception](https://keras-team.github.io/keras-tuner/documentation/hypermodels/#hyperxception-class) (<https://keras-team.github.io/keras-tuner/documentation/hypermodels/#hyperxception-class>) and [HyperResNet](https://keras-team.github.io/keras-tuner/documentation/hypermodels/#hyperresnet-class) (<https://keras-team.github.io/keras-tuner/documentation/hypermodels/#hyperresnet-class>) for computer vision applications.

In this tutorial, you use a model builder function to define the image classification model. The model builder function returns a compiled model and uses hyperparameters you define inline to hypertune the model.

In []:

```
def model_builder(hp):
    model = keras.Sequential()
    model.add(keras.layers.Flatten(input_shape=(28, 28)))

    # Tune the number of units in the first Dense layer
    # Choose an optimal value between 32-512
    hp_units = hp.Int('units', min_value=32, max_value=512, step=32)
    model.add(keras.layers.Dense(units=hp_units, activation='relu'))
    model.add(keras.layers.Dense(10))

    # Tune the learning rate for the optimizer
    # Choose an optimal value from 0.01, 0.001, or 0.0001
    hp_learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])

    model.compile(optimizer=keras.optimizers.Adam(learning_rate=hp_learning_rate),
                  loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                  metrics=['accuracy'])

    return model
```

Instantiate the tuner and perform hypertuning

Instantiate the tuner to perform the hypertuning. The Keras Tuner has four tuners available - `RandomSearch` , `Hyperband` , `BayesianOptimization` , and `Sklearn` . In this tutorial, you use the [Hyperband](https://arxiv.org/pdf/1603.06560.pdf) (<https://arxiv.org/pdf/1603.06560.pdf>) tuner.

To instantiate the Hyperband tuner, you must specify the hypermodel, the `objective` to optimize and the maximum number of epochs to train (`max_epochs`).

In []:

```
tuner = kt.Hyperband(model_builder,
                      objective='val_accuracy',
                      max_epochs=10,
                      factor=3,
                      directory='my_dir',
                      project_name='intro_to_kt')
```

The Hyperband tuning algorithm uses adaptive resource allocation and early-stopping to quickly converge on a high-performing model. This is done using a sports championship style bracket. The algorithm trains a large number of models for a few epochs and carries forward only the top-performing half of models to the next round. Hyperband determines the number of models to train in a bracket by computing $1 + \log_2(\text{factor} \cdot \text{max_epochs})$ and rounding it up to the nearest integer.

Create a callback to stop training early after reaching a certain value for the validation loss.

In []:

```
stop_early = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)
```

Run the hyperparameter search. The arguments for the search method are the same as those used for `tf.keras.model.fit` in addition to the callback above.

In []:

```
tuner.search(img_train, label_train, epochs=50, validation_split=0.2, callbacks=[stop_early])

# Get the optimal hyperparameters
best_hps=tuner.get_best_hyperparameters(num_trials=1)[0]

print(f"""
The hyperparameter search is complete. The optimal number of units in the first densely-connected
layer is {best_hps.get('units')} and the optimal learning rate for the optimizer
is {best_hps.get('learning_rate')}.
""")
```

Train the model

Find the optimal number of epochs to train the model with the hyperparameters obtained from the search.

In []:

```
# Build the model with the optimal hyperparameters and train it on the data for 50 epochs
model = tuner.hypermodel.build(best_hps)
history = model.fit(img_train, label_train, epochs=50, validation_split=0.2)

val_acc_per_epoch = history.history['val_accuracy']
best_epoch = val_acc_per_epoch.index(max(val_acc_per_epoch)) + 1
print('Best epoch: %d' % (best_epoch,))
```

Re-instantiate the hypermodel and train it with the optimal number of epochs from above.

In []:

```
hypermodel = tuner.hypermodel.build(best_hps)

# Retrain the model
hypermodel.fit(img_train, label_train, epochs=best_epoch, validation_split=0.2)
```

To finish this tutorial, evaluate the hypermodel on the test data.

In []:

```
eval_result = hypermodel.evaluate(img_test, label_test)
print("[test loss, test accuracy]:", eval_result)
```

The `my_dir/intro_to_kt` directory contains detailed logs and checkpoints for every trial (model configuration) run during the hyperparameter search. If you re-run the hyperparameter search, the Keras Tuner uses the existing state from these logs to resume the search. To disable this behavior, pass an additional `overwrite=True` argument while instantiating the tuner.

Summary

In this tutorial, you learned how to use the Keras Tuner to tune hyperparameters for a model. To learn more about the Keras Tuner, check out these additional resources:

- [Keras Tuner on the TensorFlow blog](https://blog.tensorflow.org/2020/01/hyperparameter-tuning-with-keras-tuner.html) (<https://blog.tensorflow.org/2020/01/hyperparameter-tuning-with-keras-tuner.html>)
- [Keras Tuner website](https://keras-team.github.io/keras-tuner/) (<https://keras-team.github.io/keras-tuner/>)

Also check out the [HParams Dashboard](https://www.tensorflow.org/tensorboard/hyperparameter_tuning_with_hparams) (https://www.tensorflow.org/tensorboard/hyperparameter_tuning_with_hparams) in TensorBoard to interactively tune your model hyperparameters.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

In []:

```
#@title MIT License
#
# Copyright (c) 2017 François Chollet
#
# Permission is hereby granted, free of charge, to any person obtaining a
# copy of this software and associated documentation files (the "Software"),
# to deal in the Software without restriction, including without limitation
# the rights to use, copy, modify, merge, publish, distribute, sublicense,
# and/or sell copies of the Software, and to permit persons to whom the
# Software is furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
# THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
# FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
# DEALINGS IN THE SOFTWARE.
```

Text classification with TensorFlow Hub: Movie reviews



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/keras/text_classification_with_hub)

(https://www.tensorflow.org/tutorials/keras/text_classification_with_hub)



[View on Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/keras/text_classifica)

(https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/keras/text_classifica)

This notebook classifies movie reviews as *positive* or *negative* using the text of the review. This is an example of *binary*—or two-class—classification, an important and widely applicable kind of machine learning problem.

The tutorial demonstrates the basic application of transfer learning with [TensorFlow Hub](https://tfhub.dev) (<https://tfhub.dev>) and Keras.

It uses the [IMDB dataset](https://www.tensorflow.org/api_docs/python/tf/keras/datasets/imdb) (https://www.tensorflow.org/api_docs/python/tf/keras/datasets/imdb) that contains the text of 50,000 movie reviews from the [Internet Movie Database](https://www.imdb.com/) (<https://www.imdb.com/>). These are split into 25,000 reviews for training and 25,000 reviews for testing. The training and testing sets are *balanced*, meaning they contain an equal number of positive and negative reviews.

This notebook uses `tf.keras` (<https://www.tensorflow.org/guide/keras>), a high-level API to build and train models in TensorFlow, and `tensorflow_hub` (<https://www.tensorflow.org/hub>), a library for loading trained models from [TFHub](https://tfhub.dev) (<https://tfhub.dev>) in a single line of code. For a more advanced text classification tutorial using `tf.keras`, see the [MLCC Text Classification Guide](https://developers.google.com/machine-learning/guides/text-classification/) (<https://developers.google.com/machine-learning/guides/text-classification/>).

In []:

```
!pip install tensorflow-hub
!pip install tensorflow-datasets
```

```
In [ ]:
```

```
import os
import numpy as np

import tensorflow as tf
import tensorflow_hub as hub
import tensorflow_datasets as tfds

print("Version: ", tf.__version__)
print("Eager mode: ", tf.executing_eagerly())
print("Hub version: ", hub.__version__)
print("GPU is", "available" if tf.config.list_physical_devices("GPU") else "NOT AVAILABLE")
```

Download the IMDB dataset

The IMDB dataset is available on [imdb reviews](https://www.tensorflow.org/datasets/catalog/imdb_reviews) (https://www.tensorflow.org/datasets/catalog/imdb_reviews) or on [TensorFlow datasets](https://www.tensorflow.org/datasets) (<https://www.tensorflow.org/datasets>). The following code downloads the IMDB dataset to your machine (or the colab runtime):

```
In [ ]:
```

```
# Split the training set into 60% and 40% to end up with 15,000 examples
# for training, 10,000 examples for validation and 25,000 examples for testing.
train_data, validation_data, test_data = tfds.load(
    name="imdb_reviews",
    split=('train[:60%]', 'train[60%:]', 'test'),
    as_supervised=True)
```

Explore the data

Let's take a moment to understand the format of the data. Each example is a sentence representing the movie review and a corresponding label. The sentence is not preprocessed in any way. The label is an integer value of either 0 or 1, where 0 is a negative review, and 1 is a positive review.

Let's print first 10 examples.

```
In [ ]:
```

```
train_examples_batch, train_labels_batch = next(iter(train_data.batch(10)))
train_examples_batch
```

Let's also print the first 10 labels.

```
In [ ]:
```

```
train_labels_batch
```

Build the model

The neural network is created by stacking layers—this requires three main architectural decisions:

- How to represent the text?
- How many layers to use in the model?
- How many *hidden units* to use for each layer?

In this example, the input data consists of sentences. The labels to predict are either 0 or 1.

One way to represent the text is to convert sentences into embeddings vectors. Use a pre-trained text embedding as the first layer, which will have three advantages:

- You don't have to worry about text preprocessing.
- Benefit from transfer learning,
- the embedding has a fixed size, so it's simpler to process.

For this example you use a **pre-trained text embedding model** from [TensorFlow Hub](https://tfhub.dev) (<https://tfhub.dev>) called [google/nnlm-en-dim50/2](https://tfhub.dev/google/nnlm-en-dim50/2) (<https://tfhub.dev/google/nnlm-en-dim50/2>).

There are many other pre-trained text embeddings from TFHub that can be used in this tutorial:

- [google/nnlm-en-dim128/2](https://tfhub.dev/google/nnlm-en-dim128/2) (<https://tfhub.dev/google/nnlm-en-dim128/2>) - trained with the same NNLM architecture on the same data as [google/nnlm-en-dim50/2](https://tfhub.dev/google/nnlm-en-dim50/2) (<https://tfhub.dev/google/nnlm-en-dim50/2>), but with a larger embedding dimension. Larger dimensional embeddings can improve on your task but it may take longer to train your model.
- [google/nnlm-en-dim128-with-normalization/2](https://tfhub.dev/google/nnlm-en-dim128-with-normalization/2) (<https://tfhub.dev/google/nnlm-en-dim128-with-normalization/2>) - the same as [google/nnlm-en-dim128/2](https://tfhub.dev/google/nnlm-en-dim128/2) (<https://tfhub.dev/google/nnlm-en-dim128/2>), but with additional text normalization such as removing punctuation. This can help if the text in your task contains additional characters or punctuation.
- [google/universal-sentence-encoder/4](https://tfhub.dev/google/universal-sentence-encoder/4) (<https://tfhub.dev/google/universal-sentence-encoder/4>) - a much larger model yielding 512 dimensional embeddings trained with a deep averaging network (DAN) encoder.

And many more! Find more [text embedding models](https://tfhub.dev/s?module-type=text-embedding) (<https://tfhub.dev/s?module-type=text-embedding>) on TFHub.

Let's first create a Keras layer that uses a TensorFlow Hub model to embed the sentences, and try it out on a couple of input examples. Note that no matter the length of the input text, the output shape of the embeddings is: (num_examples, embedding_dimension) .

In []:

```
embedding = "https://tfhub.dev/google/nnlm-en-dim50/2"
hub_layer = hub.KerasLayer(embedding, input_shape=[],
                           dtype=tf.string, trainable=True)
hub_layer(train_examples_batch[:3])
```

Let's now build the full model:

In []:

```
model = tf.keras.Sequential()
model.add(hub_layer)
model.add(tf.keras.layers.Dense(16, activation='relu'))
model.add(tf.keras.layers.Dense(1))

model.summary()
```

The layers are stacked sequentially to build the classifier:

1. The first layer is a TensorFlow Hub layer. This layer uses a pre-trained Saved Model to map a sentence into its embedding vector. The pre-trained text embedding model that you are using ([google/nnlm-en-dim50/2](https://tfhub.dev/google/nnlm-en-dim50/2) (<https://tfhub.dev/google/nnlm-en-dim50/2>)) splits the sentence into tokens, embeds each token and then combines the embedding. The resulting dimensions are: (num_examples, embedding_dimension) . For this NNLM model, the embedding_dimension is 50.
2. This fixed-length output vector is piped through a fully-connected (Dense) layer with 16 hidden units.
3. The last layer is densely connected with a single output node.

Let's compile the model.

Loss function and optimizer

A model needs a loss function and an optimizer for training. Since this is a binary classification problem and the model outputs logits (a single-unit layer with a linear activation), you'll use the `binary_crossentropy` loss function.

This isn't the only choice for a loss function, you could, for instance, choose `mean_squared_error`. But, generally, `binary_crossentropy` is better for dealing with probabilities—it measures the "distance" between probability distributions, or in our case, between the ground-truth distribution and the predictions.

Later, when you are exploring regression problems (say, to predict the price of a house), you'll see how to use another loss function called mean squared error.

Now, configure the model to use an optimizer and a loss function:

In []:

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

Train the model

Train the model for 10 epochs in mini-batches of 512 samples. This is 10 iterations over all samples in the `x_train` and `y_train` tensors. While training, monitor the model's loss and accuracy on the 10,000 samples from the validation set:

In []:

```
history = model.fit(train_data.shuffle(10000).batch(512),
                     epochs=10,
                     validation_data=validation_data.batch(512),
                     verbose=1)
```

Evaluate the model

And let's see how the model performs. Two values will be returned. Loss (a number which represents our error, lower values are better), and accuracy.

In []:

```
results = model.evaluate(test_data.batch(512), verbose=2)

for name, value in zip(model.metrics_names, results):
    print("%s: %.3f" % (name, value))
```

This fairly naive approach achieves an accuracy of about 87%. With more advanced approaches, the model should get closer to 95%.

Further reading

- For a more general way to work with string inputs and for a more detailed analysis of the progress of accuracy and loss during training, see the [Text classification with preprocessed text \(./text_classification.ipynb\)](#) tutorial.
- Try out more [text-related tutorials](#) (<https://www.tensorflow.org/hub/tutorials#text-related-tutorials>) using trained models from TFHub.

Copyright 2018 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

In []:

```
#@title MIT License
#
# Copyright (c) 2017 François Chollet
#
# Permission is hereby granted, free of charge, to any person obtaining a
# copy of this software and associated documentation files (the "Software"),
# to deal in the Software without restriction, including without limitation
# the rights to use, copy, modify, merge, publish, distribute, sublicense,
# and/or sell copies of the Software, and to permit persons to whom the
# Software is furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
# THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
# FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
# DEALINGS IN THE SOFTWARE.
```

Basic regression: Predict fuel efficiency



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/keras/regression)

(<https://www.tensorflow.org/tutorials/keras/regression>)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/keras/regression.ipynb)

(<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/keras/regression.ipynb>)

In a *regression* problem, the aim is to predict the output of a continuous value, like a price or a probability. Contrast this with a *classification* problem, where the aim is to select a class from a list of classes (for example, where a picture contains an apple or an orange, recognizing which fruit is in the picture).

This tutorial uses the classic [Auto MPG](https://archive.ics.uci.edu/ml/datasets/auto+mpg) (<https://archive.ics.uci.edu/ml/datasets/auto+mpg>) dataset and demonstrates how to build models to predict the fuel efficiency of the late-1970s and early 1980s automobiles. To do this, you will provide the models with a description of many automobiles from that time period. This description includes attributes like cylinders, displacement, horsepower, and weight.

This example uses the Keras API. (Visit the Keras [tutorials](https://www.tensorflow.org/tutorials/keras) (<https://www.tensorflow.org/tutorials/keras>) and [guides](https://www.tensorflow.org/guide/keras) (<https://www.tensorflow.org/guide/keras>) to learn more.)

In []:

```
# Use seaborn for pairplot.
!pip install -q seaborn
```

In []:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns

# Make NumPy printouts easier to read.
np.set_printoptions(precision=3, suppress=True)
```

In []:

```
import tensorflow as tf

from tensorflow import keras
from tensorflow.keras import layers

print(tf.__version__)
```

The Auto MPG dataset

The dataset is available from the [UCI Machine Learning Repository](https://archive.ics.uci.edu/ml/) (<https://archive.ics.uci.edu/ml/>).

Get the data

First download and import the dataset using pandas:

```
In [ ]:
```

```
url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-mpg.data'
column_names = ['MPG', 'Cylinders', 'Displacement', 'Horsepower', 'Weight',
               'Acceleration', 'Model Year', 'Origin']

raw_dataset = pd.read_csv(url, names=column_names,
                           na_values='?', comment='\t',
                           sep=' ', skipinitialspace=True)
```

```
In [ ]:
```

```
dataset = raw_dataset.copy()
dataset.tail()
```

Clean the data

The dataset contains a few unknown values:

```
In [ ]:
```

```
dataset.isna().sum()
```

Drop those rows to keep this initial tutorial simple:

```
In [ ]:
```

```
dataset = dataset.dropna()
```

The "Origin" column is categorical, not numeric. So the next step is to one-hot encode the values in the column with [pd.get_dummies](#) (https://pandas.pydata.org/docs/reference/api/pandas.get_dummies.html).

Note: You can set up the `tf.keras.Model` to do this kind of transformation for you but that's beyond the scope of this tutorial. Check out the [Classify structured data using Keras preprocessing layers](#) ([./structured_data/preprocessing_layers.ipynb](#)) or [Load CSV data](#) ([./load_data/csv.ipynb](#)) tutorials for examples.

```
In [ ]:
```

```
dataset['Origin'] = dataset['Origin'].map({1: 'USA', 2: 'Europe', 3: 'Japan'})
```

```
In [ ]:
```

```
dataset = pd.get_dummies(dataset, columns=['Origin'], prefix='', prefix_sep=' ')
dataset.tail()
```

Split the data into training and test sets

Now, split the dataset into a training set and a test set. You will use the test set in the final evaluation of your models.

```
In [ ]:
```

```
train_dataset = dataset.sample(frac=0.8, random_state=0)
test_dataset = dataset.drop(train_dataset.index)
```

Inspect the data

Review the joint distribution of a few pairs of columns from the training set.

The top row suggests that the fuel efficiency (MPG) is a function of all the other parameters. The other rows indicate they are functions of each other.

```
In [ ]:
```

```
sns.pairplot(train_dataset[['MPG', 'Cylinders', 'Displacement', 'Weight']], diag_kind='kde')
```

Let's also check the overall statistics. Note how each feature covers a very different range:

```
In [ ]:
```

```
train_dataset.describe().transpose()
```

Split features from labels

Separate the target value—the "label"—from the features. This label is the value that you will train the model to predict.

In []:

```
train_features = train_dataset.copy()
test_features = test_dataset.copy()

train_labels = train_features.pop('MPG')
test_labels = test_features.pop('MPG')
```

Normalization

In the table of statistics it's easy to see how different the ranges of each feature are:

In []:

```
train_dataset.describe().transpose()[['mean', 'std']]
```

It is good practice to normalize features that use different scales and ranges.

One reason this is important is because the features are multiplied by the model weights. So, the scale of the outputs and the scale of the gradients are affected by the scale of the inputs.

Although a model *might* converge without feature normalization, normalization makes training much more stable.

Note: There is no advantage to normalizing the one-hot features—it is done here for simplicity. For more details on how to use the preprocessing layers, refer to the [Working with preprocessing layers \(\[https://www.tensorflow.org/guide/keras/preprocessing_layers\]\(https://www.tensorflow.org/guide/keras/preprocessing_layers\)\)](https://www.tensorflow.org/guide/keras/preprocessing_layers) guide and the [Classify structured data using Keras preprocessing layers \(\[./structured_data/preprocessing_layers.ipynb\]\(#\)\)](#) tutorial.

The Normalization layer

The `tf.keras.layers.Normalization` is a clean and simple way to add feature normalization into your model.

The first step is to create the layer:

In []:

```
normalizer = tf.keras.layers.Normalization(axis=-1)
```

Then, fit the state of the preprocessing layer to the data by calling `Normalization.adapt`:

In []:

```
normalizer.adapt(np.array(train_features))
```

Calculate the mean and variance, and store them in the layer:

In []:

```
print(normalizer.mean.numpy())
```

When the layer is called, it returns the input data, with each feature independently normalized:

In []:

```
first = np.array(train_features[:1])

with np.printoptions(precision=2, suppress=True):
    print('First example:', first)
    print()
    print('Normalized:', normalizer(first).numpy())
```

Linear regression

Before building a deep neural network model, start with linear regression using one and several variables.

Linear regression with one variable

Begin with a single-variable linear regression to predict 'MPG' from 'Horsepower' .

Training a model with `tf.keras` typically starts by defining the model architecture. Use a `tf.keras.Sequential` model, which [represents a sequence of steps](https://www.tensorflow.org/guide/keras/sequential_model) (https://www.tensorflow.org/guide/keras/sequential_model).

There are two steps in your single-variable linear regression model:

- Normalize the 'Horsepower' input features using the `tf.keras.layers.Normalization` preprocessing layer.
- Apply a linear transformation ($y = mx + b$) to produce 1 output using a linear layer (`tf.keras.layers.Dense`).

The number of *inputs* can either be set by the `input_shape` argument, or automatically when the model is run for the first time.

First, create a NumPy array made of the 'Horsepower' features. Then, instantiate the `tf.keras.layers.Normalization` and fit its state to the `horsepower` data:

In []:

```
horsepower = np.array(train_features['Horsepower'])

horsepower_normalizer = layers.Normalization(input_shape=[1,], axis=None)
horsepower_normalizer.adapt(horsepower)
```

Build the Keras Sequential model:

In []:

```
horsepower_model = tf.keras.Sequential([
    horsepower_normalizer,
    layers.Dense(units=1)
])

horsepower_model.summary()
```

This model will predict 'MPG' from 'Horsepower' .

Run the untrained model on the first 10 'Horsepower' values. The output won't be good, but notice that it has the expected shape of (10, 1) :

In []:

```
horsepower_model.predict(horsepower[:10])
```

Once the model is built, configure the training procedure using the Keras `Model.compile` method. The most important arguments to compile are the `loss` and the `optimizer` , since these define what will be optimized (`mean_absolute_error`) and how (using the `tf.keras.optimizers.Adam`).

In []:

```
horsepower_model.compile(
    optimizer=tf.optimizers.Adam(learning_rate=0.1),
    loss='mean_absolute_error')
```

Use Keras `Model.fit` to execute the training for 100 epochs:

In []:

```
%%time
history = horsepower_model.fit(
    train_features['Horsepower'],
    train_labels,
    epochs=100,
    # Suppress logging.
    verbose=0,
    # Calculate validation results on 20% of the training data.
    validation_split = 0.2)
```

Visualize the model's training progress using the stats stored in the `history` object:

In []:

```
hist = pd.DataFrame(history.history)
hist['epoch'] = history.epoch
hist.tail()
```

```
In [ ]:
```

```
def plot_loss(history):
    plt.plot(history.history['loss'], label='loss')
    plt.plot(history.history['val_loss'], label='val_loss')
    plt.ylim([0, 10])
    plt.xlabel('Epoch')
    plt.ylabel('Error [MPG]')
    plt.legend()
    plt.grid(True)
```

```
In [ ]:
```

```
plot_loss(history)
```

Collect the results on the test set for later:

```
In [ ]:
```

```
test_results = {}

test_results['horsepower_model'] = horsepower_model.evaluate(
    test_features['Horsepower'],
    test_labels, verbose=0)
```

Since this is a single variable regression, it's easy to view the model's predictions as a function of the input:

```
In [ ]:
```

```
x = tf.linspace(0.0, 250, 251)
y = horsepower_model.predict(x)
```

```
In [ ]:
```

```
def plot_horsepower(x, y):
    plt.scatter(train_features['Horsepower'], train_labels, label='Data')
    plt.plot(x, y, color='k', label='Predictions')
    plt.xlabel('Horsepower')
    plt.ylabel('MPG')
    plt.legend()
```

```
In [ ]:
```

```
plot_horsepower(x, y)
```

Linear regression with multiple inputs

You can use an almost identical setup to make predictions based on multiple inputs. This model still does the same $y = mx + b$ except that m is a matrix and b is a vector.

Create a two-step Keras Sequential model again with the first layer being `normalizer` (`tf.keras.layers.Normalization(axis=-1)`) you defined earlier and adapted to the whole dataset:

```
In [ ]:
```

```
linear_model = tf.keras.Sequential([
    normalizer,
    layers.Dense(units=1)
])
```

When you call `Model.predict` on a batch of inputs, it produces `units=1` outputs for each example:

```
In [ ]:
```

```
linear_model.predict(train_features[:10])
```

When you call the model, its weight matrices will be built—check that the `kernel` weights (the m in $y = mx + b$) have a shape of `(9, 1)`:

```
In [ ]:
```

```
linear_model.layers[1].kernel
```

Configure the model with Keras `Model.compile` and train with `Model.fit` for 100 epochs:

```
In [ ]:
```

```
linear_model.compile(  
    optimizer=tf.optimizers.Adam(learning_rate=0.1),  
    loss='mean_absolute_error')
```

```
In [ ]:
```

```
%%time  
history = linear_model.fit(  
    train_features,  
    train_labels,  
    epochs=100,  
    # Suppress logging.  
    verbose=0,  
    # Calculate validation results on 20% of the training data.  
    validation_split = 0.2)
```

Using all the inputs in this regression model achieves a much lower training and validation error than the `horsepower_model`, which had one input:

```
In [ ]:
```

```
plot_loss(history)
```

Collect the results on the test set for later:

```
In [ ]:
```

```
test_results['linear_model'] = linear_model.evaluate(  
    test_features, test_labels, verbose=0)
```

Regression with a deep neural network (DNN)

In the previous section, you implemented two linear models for single and multiple inputs.

Here, you will implement single-input and multiple-input DNN models.

The code is basically the same except the model is expanded to include some "hidden" non-linear layers. The name "hidden" here just means not directly connected to the inputs or outputs.

These models will contain a few more layers than the linear model:

- The normalization layer, as before (with `horsepower_normalizer` for a single-input model and `normalizer` for a multiple-input model).
- Two hidden, non-linear, `Dense` layers with the `ReLU` (`relu`) activation function nonlinearity.
- A linear `Dense` single-output layer.

Both models will use the same training procedure so the `compile` method is included in the `build_and_compile_model` function below.

```
In [ ]:
```

```
def build_and_compile_model(norm):  
    model = keras.Sequential([  
        norm,  
        layers.Dense(64, activation='relu'),  
        layers.Dense(64, activation='relu'),  
        layers.Dense(1)  
    ])  
  
    model.compile(loss='mean_absolute_error',  
                  optimizer=tf.keras.optimizers.Adam(0.001))  
    return model
```

Regression using a DNN and a single input

Create a DNN model with only 'Horsepower' as input and `horsepower_normalizer` (defined earlier) as the normalization layer:

```
In [ ]:
```

```
dnn_horsepower_model = build_and_compile_model(horsepower_normalizer)
```

This model has quite a few more trainable parameters than the linear models:

In []:

```
dnn_horsepower_model.summary()
```

Train the model with Keras Model.fit :

In []:

```
%%time
history = dnn_horsepower_model.fit(
    train_features['Horsepower'],
    train_labels,
    validation_split=0.2,
    verbose=0, epochs=100)
```

This model does slightly better than the linear single-input horsepower_model :

In []:

```
plot_loss(history)
```

If you plot the predictions as a function of 'Horsepower' , you should notice how this model takes advantage of the nonlinearity provided by the hidden layers:

In []:

```
x = tf.linspace(0.0, 250, 251)
y = dnn_horsepower_model.predict(x)
```

In []:

```
plot_horsepower(x, y)
```

Collect the results on the test set for later:

In []:

```
test_results['dnn_horsepower_model'] = dnn_horsepower_model.evaluate(
    test_features['Horsepower'], test_labels,
    verbose=0)
```

Regression using a DNN and multiple inputs

Repeat the previous process using all the inputs. The model's performance slightly improves on the validation dataset.

In []:

```
dnn_model = build_and_compile_model(normalizer)
dnn_model.summary()
```

In []:

```
%%time
history = dnn_model.fit(
    train_features,
    train_labels,
    validation_split=0.2,
    verbose=0, epochs=100)
```

In []:

```
plot_loss(history)
```

Collect the results on the test set:

In []:

```
test_results['dnn_model'] = dnn_model.evaluate(test_features, test_labels, verbose=0)
```

Performance

Since all models have been trained, you can review their test set performance:

In []:

```
pd.DataFrame(test_results, index=[ 'Mean absolute error [MPG]' ]).T
```

These results match the validation error observed during training.

Make predictions

You can now make predictions with the `dnn_model` on the test set using Keras `Model.predict` and review the loss:

In []:

```
test_predictions = dnn_model.predict(test_features).flatten()

a = plt.axes(aspect='equal')
plt.scatter(test_labels, test_predictions)
plt.xlabel('True Values [MPG]')
plt.ylabel('Predictions [MPG]')
lims = [0, 50]
plt.xlim(lims)
plt.ylim(lims)
_ = plt.plot(lims, lims)
```

It appears that the model predicts reasonably well.

Now, check the error distribution:

In []:

```
error = test_predictions - test_labels
plt.hist(error, bins=25)
plt.xlabel('Prediction Error [MPG]')
_ = plt.ylabel('Count')
```

If you're happy with the model, save it for later use with `Model.save`:

In []:

```
dnn_model.save('dnn_model')
```

If you reload the model, it gives identical output:

In []:

```
reloaded = tf.keras.models.load_model('dnn_model')

test_results['reloaded'] = reloaded.evaluate(
    test_features, test_labels, verbose=0)
```

In []:

```
pd.DataFrame(test_results, index=[ 'Mean absolute error [MPG]' ]).T
```

Conclusion

This notebook introduced a few techniques to handle a regression problem. Here are a few more tips that may help:

- Mean squared error (MSE) (`tf.keras.losses.MeanSquaredError`) and mean absolute error (MAE) (`tf.keras.losses.MeanAbsoluteError`) are common loss functions used for regression problems. MAE is less sensitive to outliers. Different loss functions are used for classification problems.
- Similarly, evaluation metrics used for regression differ from classification.
- When numeric input data features have values with different ranges, each feature should be scaled independently to the same range.
- Overfitting is a common problem for DNN models, though it wasn't a problem for this tutorial. Visit the [Overfit and underfit \(overfit_and_underfit.ipynb\)](#) tutorial for more help with this.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

In []:

```
#@title MIT License
#
# Copyright (c) 2017 François Chollet
#
# Permission is hereby granted, free of charge, to any person obtaining a
# copy of this software and associated documentation files (the "Software"),
# to deal in the Software without restriction, including without limitation
# the rights to use, copy, modify, merge, publish, distribute, sublicense,
# and/or sell copies of the Software, and to permit persons to whom the
# Software is furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
# THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
# FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
# DEALINGS IN THE SOFTWARE.
```

Overfit and underfit



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/keras/overfit_and_underfit)

(https://www.tensorflow.org/tutorials/keras/overfit_and_underfit)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/keras/overfit_and_underfit.ipynb)

(https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/keras/overfit_and_underfit.ipynb)

As always, the code in this example will use the `tf.keras` API, which you can learn more about in the TensorFlow [Keras guide](https://www.tensorflow.org/guide/keras) (<https://www.tensorflow.org/guide/keras>).

In both of the previous examples—[classifying text \(text_classification_with_hub.ipynb\)](#) and [predicting fuel efficiency \(regression.ipynb\)](#)—the accuracy of models on the validation data would peak after training for a number of epochs and then stagnate or start decreasing.

In other words, your model would *overfit* to the training data. Learning how to deal with overfitting is important. Although it's often possible to achieve high accuracy on the *training set*, what you really want is to develop models that generalize well to a *testing set* (or data they haven't seen before).

The opposite of overfitting is *underfitting*. Underfitting occurs when there is still room for improvement on the train data. This can happen for a number of reasons: If the model is not powerful enough, is over-regularized, or has simply not been trained long enough. This means the network has not learned the relevant patterns in the training data.

If you train for too long though, the model will start to overfit and learn patterns from the training data that don't generalize to the test data. You need to strike a balance. Understanding how to train for an appropriate number of epochs as you'll explore below is a useful skill.

To prevent overfitting, the best solution is to use more complete training data. The dataset should cover the full range of inputs that the model is expected to handle. Additional data may only be useful if it covers new and interesting cases.

A model trained on more complete data will naturally generalize better. When that is no longer possible, the next best solution is to use techniques like regularization. These place constraints on the quantity and type of information your model can store. If a network can only afford to memorize a small number of patterns, the optimization process will force it to focus on the most prominent patterns, which have a better chance of generalizing well.

In this notebook, you'll explore several common regularization techniques, and use them to improve on a classification model.

Setup

Before getting started, import the necessary packages:

```
In [ ]:
```

```
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras import regularizers
print(tf.__version__)
```

```
In [ ]:
```

```
!pip install git+https://github.com/tensorflow/docs
```

```
import tensorflow_docs as tfdocs
import tensorflow_docs.modeling
import tensorflow_docs.plots
```

```
In [ ]:
```

```
from IPython import display
from matplotlib import pyplot as plt

import numpy as np

import pathlib
import shutil
import tempfile
```

```
In [ ]:
```

```
logdir = pathlib.Path(tempfile.mkdtemp())/"tensorboard_logs"
shutil.rmtree(logdir, ignore_errors=True)
```

The Higgs dataset

The goal of this tutorial is not to do particle physics, so don't dwell on the details of the dataset. It contains 11,000,000 examples, each with 28 features, and a binary class label.

```
In [ ]:
```

```
gz = tf.keras.utils.get_file('HIGGS.csv.gz', 'http://mlphysics.ics.uci.edu/data/higgs/HIGGS.csv.gz')
```

```
In [ ]:
```

```
FEATURES = 28
```

The `tf.data.experimental.CsvDataset` class can be used to read csv records directly from a gzip file with no intermediate decompression step.

```
In [ ]:
```

```
ds = tf.data.experimental.CsvDataset(gz,[float(),]*(FEATURES+1), compression_type="GZIP")
```

That csv reader class returns a list of scalars for each record. The following function repacks that list of scalars into a (feature_vector, label) pair.

```
In [ ]:
```

```
def pack_row(*row):
    label = row[0]
    features = tf.stack(row[1:],1)
    return features, label
```

TensorFlow is most efficient when operating on large batches of data.

So, instead of repacking each row individually make a new `tf.data.Dataset` that takes batches of 10,000 examples, applies the `pack_row` function to each batch, and then splits the batches back up into individual records:

```
In [ ]:
```

```
packed_ds = ds.batch(10000).map(pack_row).unbatch()
```

Inspect some of the records from this new `packed_ds`.

The features are not perfectly normalized, but this is sufficient for this tutorial.

In []:

```
for features,label in packed_ds.batch(1000).take(1):
    print(features[0])
    plt.hist(features.numpy().flatten(), bins = 101)
```

To keep this tutorial relatively short, use just the first 1,000 samples for validation, and the next 10,000 for training:

In []:

```
N_VALIDATION = int(1e3)
N_TRAIN = int(1e4)
BUFFER_SIZE = int(1e4)
BATCH_SIZE = 500
STEPS_PER_EPOCH = N_TRAIN//BATCH_SIZE
```

The `Dataset.skip` and `Dataset.take` methods make this easy.

At the same time, use the `Dataset.cache` method to ensure that the loader doesn't need to re-read the data from the file on each epoch:

In []:

```
validate_ds = packed_ds.take(N_VALIDATION).cache()
train_ds = packed_ds.skip(N_VALIDATION).take(N_TRAIN).cache()
```

In []:

```
train_ds
```

These datasets return individual examples. Use the `Dataset.batch` method to create batches of an appropriate size for training. Before batching, also remember to use `Dataset.shuffle` and `Dataset.repeat` on the training set.

In []:

```
validate_ds = validate_ds.batch(BATCH_SIZE)
train_ds = train_ds.shuffle(BUFFER_SIZE).repeat().batch(BATCH_SIZE)
```

Demonstrate overfitting

The simplest way to prevent overfitting is to start with a small model: A model with a small number of learnable parameters (which is determined by the number of layers and the number of units per layer). In deep learning, the number of learnable parameters in a model is often referred to as the model's "capacity".

Intuitively, a model with more parameters will have more "memorization capacity" and therefore will be able to easily learn a perfect dictionary-like mapping between training samples and their targets, a mapping without any generalization power, but this would be useless when making predictions on previously unseen data.

Always keep this in mind: deep learning models tend to be good at fitting to the training data, but the real challenge is generalization, not fitting.

On the other hand, if the network has limited memorization resources, it will not be able to learn the mapping as easily. To minimize its loss, it will have to learn compressed representations that have more predictive power. At the same time, if you make your model too small, it will have difficulty fitting to the training data. There is a balance between "too much capacity" and "not enough capacity".

Unfortunately, there is no magical formula to determine the right size or architecture of your model (in terms of the number of layers, or the right size for each layer). You will have to experiment using a series of different architectures.

To find an appropriate model size, it's best to start with relatively few layers and parameters, then begin increasing the size of the layers or adding new layers until you see diminishing returns on the validation loss.

Start with a simple model using only densely-connected layers (`tf.keras.layers.Dense`) as a baseline, then create larger models, and compare them.

Training procedure

Many models train better if you gradually reduce the learning rate during training. Use `tf.keras.optimizers.schedules` to reduce the learning rate over time:

In []:

```
lr_schedule = tf.keras.optimizers.schedules.InverseTimeDecay(  
    0.001,  
    decay_steps=STEPS_PER_EPOCH*1000,  
    decay_rate=1,  
    staircase=False)  
  
def get_optimizer():  
    return tf.keras.optimizers.Adam(lr_schedule)
```

The code above sets a `tf.keras.optimizers.schedules.InverseTimeDecay` to hyperbolically decrease the learning rate to 1/2 of the base rate at 1,000 epochs, 1/3 at 2,000 epochs, and so on.

In []:

```
step = np.linspace(0,100000)  
lr = lr_schedule(step)  
plt.figure(figsize = (8,6))  
plt.plot(step/STEPS_PER_EPOCH, lr)  
plt.ylim([0,max(plt.ylim())])  
plt.xlabel('Epoch')  
_ = plt.ylabel('Learning Rate')
```

Each model in this tutorial will use the same training configuration. So set these up in a reusable way, starting with the list of callbacks.

The training for this tutorial runs for many short epochs. To reduce the logging noise use the `tfdocs.EpochDots` which simply prints a `.` for each epoch, and a full set of metrics every 100 epochs.

Next include `tf.keras.callbacks.EarlyStopping` to avoid long and unnecessary training times. Note that this callback is set to monitor the `val_binary_crossentropy`, not the `val_loss`. This difference will be important later.

Use `callbacks.TensorBoard` to generate TensorBoard logs for the training.

In []:

```
def get_callbacks(name):  
    return [  
        tfdocs.modeling.EpochDots(),  
        tf.keras.callbacks.EarlyStopping(monitor='val_binary_crossentropy', patience=200),  
        tf.keras.callbacks.TensorBoard(logdir=name),  
    ]
```

Similarly each model will use the same `Model.compile` and `Model.fit` settings:

In []:

```
def compile_and_fit(model, name, optimizer=None, max_epochs=10000):  
    if optimizer is None:  
        optimizer = get_optimizer()  
    model.compile(optimizer=optimizer,  
                  loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),  
                  metrics=[  
                      tf.keras.losses.BinaryCrossentropy(  
                          from_logits=True, name='binary_crossentropy'),  
                      'accuracy'])  
  
    model.summary()  
  
    history = model.fit(  
        train_ds,  
        steps_per_epoch = STEPS_PER_EPOCH,  
        epochs=max_epochs,  
        validation_data=validate_ds,  
        callbacks=get_callbacks(name),  
        verbose=0)  
    return history
```

Tiny model

Start by training a model:

```
In [ ]:
```

```
tiny_model = tf.keras.Sequential([
    layers.Dense(16, activation='elu', input_shape=(FEATURES,)),
    layers.Dense(1)
])
```

```
In [ ]:
```

```
size_histories = {}
```

```
In [ ]:
```

```
size_histories['Tiny'] = compile_and_fit(tiny_model, 'sizes/Tiny')
```

Now check how the model did:

```
In [ ]:
```

```
plotter = tfdocs.plots.HistoryPlotter(metric = 'binary_crossentropy', smoothing_std=10)
plotter.plot(size_histories)
plt.ylim([0.5, 0.7])
```

Small model

To check if you can beat the performance of the small model, progressively train some larger models.

Try two hidden layers with 16 units each:

```
In [ ]:
```

```
small_model = tf.keras.Sequential([
    #`input_shape` is only required here so that `summary` works.
    layers.Dense(16, activation='elu', input_shape=(FEATURES,)),
    layers.Dense(16, activation='elu'),
    layers.Dense(1)
])
```

```
In [ ]:
```

```
size_histories['Small'] = compile_and_fit(small_model, 'sizes/Small')
```

Medium model

Now try three hidden layers with 64 units each:

```
In [ ]:
```

```
medium_model = tf.keras.Sequential([
    layers.Dense(64, activation='elu', input_shape=(FEATURES,)),
    layers.Dense(64, activation='elu'),
    layers.Dense(64, activation='elu'),
    layers.Dense(1)
])
```

And train the model using the same data:

```
In [ ]:
```

```
size_histories['Medium'] = compile_and_fit(medium_model, "sizes/Medium")
```

Large model

As an exercise, you can create an even larger model and check how quickly it begins overfitting. Next, add to this benchmark a network that has much more capacity, far more than the problem would warrant:

```
In [ ]:
```

```
large_model = tf.keras.Sequential([
    layers.Dense(512, activation='elu', input_shape=(FEATURES,)),
    layers.Dense(512, activation='elu'),
    layers.Dense(512, activation='elu'),
    layers.Dense(512, activation='elu'),
    layers.Dense(1)
])
```

And, again, train the model using the same data:

```
In [ ]:
```

```
size_histories['large'] = compile_and_fit(large_model, "sizes/large")
```

Plot the training and validation losses

The solid lines show the training loss, and the dashed lines show the validation loss (remember: a lower validation loss indicates a better model).

While building a larger model gives it more power, if this power is not constrained somehow it can easily overfit to the training set.

In this example, typically, only the "Tiny" model manages to avoid overfitting altogether, and each of the larger models overfit the data more quickly. This becomes so severe for the "large" model that you need to switch the plot to a log-scale to really figure out what's happening.

This is apparent if you plot and compare the validation metrics to the training metrics.

- It's normal for there to be a small difference.
- If both metrics are moving in the same direction, everything is fine.
- If the validation metric begins to stagnate while the training metric continues to improve, you are probably close to overfitting.
- If the validation metric is going in the wrong direction, the model is clearly overfitting.

```
In [ ]:
```

```
plotter.plot(size_histories)
a = plt.xscale('log')
plt.xlim([5, max(plt.xlim())])
plt.ylim([0.5, 0.7])
plt.xlabel("Epochs [Log Scale]")
```

Note: All the above training runs used the `callbacks.EarlyStopping` to end the training once it was clear the model was not making progress.

View in TensorBoard

These models all wrote TensorBoard logs during training.

Open an embedded TensorBoard viewer inside a notebook:

```
In [ ]:
```

```
#docs_infra: no_execute

# Load the TensorBoard notebook extension
%load_ext tensorboard

# Open an embedded TensorBoard viewer
%tensorboard --logdir {logdir}/sizes
```

You can view the [results of a previous run \(\[https://tensorboard.dev/experiment/vW7jmmF9TmKmy3rbheMQpw/#scalars&_smoothingWeight=0.97\]\(https://tensorboard.dev/experiment/vW7jmmF9TmKmy3rbheMQpw/#scalars&_smoothingWeight=0.97\)\)](https://tensorboard.dev/experiment/vW7jmmF9TmKmy3rbheMQpw/#scalars&_smoothingWeight=0.97) of this notebook on [TensorBoard.dev \(<https://TensorBoard.dev>\)](https://TensorBoard.dev).

TensorBoard.dev is a managed experience for hosting, tracking, and sharing ML experiments with everyone.

It's also included in an `<iframe>` for convenience:

```
In [ ]:
```

```
display.IFrame(
    src="https://tensorboard.dev/experiment/vW7jmmF9TmKmy3rbheMQpw/#scalars&_smoothingWeight=0.97",
    width="100%", height="800px")
```

If you want to share TensorBoard results you can upload the logs to [TensorBoard.dev \(https://tensorboard.dev/\)](https://tensorboard.dev/) by copying the following into a code-cell.

Note: This step requires a Google account.

```
!tensorboard dev upload --logdir {logdir}/sizes
```

Caution: This command does not terminate. It's designed to continuously upload the results of long-running experiments. Once your data is uploaded you need to stop it using the "interrupt execution" option in your notebook tool.

Strategies to prevent overfitting

Before getting into the content of this section copy the training logs from the "Tiny" model above, to use as a baseline for comparison.

In []:

```
shutil.rmtree(logdir/'regularizers/Tiny', ignore_errors=True)
shutil.copytree(logdir/'sizes/Tiny', logdir/'regularizers/Tiny')
```

In []:

```
regularizer_histories = {}
regularizer_histories['Tiny'] = size_histories['Tiny']
```

Add weight regularization

You may be familiar with Occam's Razor principle: given two explanations for something, the explanation most likely to be correct is the "simplest" one, the one that makes the least amount of assumptions. This also applies to the models learned by neural networks: given some training data and a network architecture, there are multiple sets of weights values (multiple models) that could explain the data, and simpler models are less likely to overfit than complex ones.

A "simple model" in this context is a model where the distribution of parameter values has less entropy (or a model with fewer parameters altogether, as demonstrated in the section above). Thus a common way to mitigate overfitting is to put constraints on the complexity of a network by forcing its weights only to take small values, which makes the distribution of weight values more "regular". This is called "weight regularization", and it is done by adding to the loss function of the network a cost associated with having large weights. This cost comes in two flavors:

- [L1 regularization \(https://developers.google.com/machine-learning/glossary/#L1_regularization\)](https://developers.google.com/machine-learning/glossary/#L1_regularization), where the cost added is proportional to the absolute value of the weights coefficients (i.e. to what is called the "L1 norm" of the weights).
- [L2 regularization \(https://developers.google.com/machine-learning/glossary/#L2_regularization\)](https://developers.google.com/machine-learning/glossary/#L2_regularization), where the cost added is proportional to the square of the value of the weights coefficients (i.e. to what is called the squared "L2 norm" of the weights). L2 regularization is also called weight decay in the context of neural networks. Don't let the different name confuse you: weight decay is mathematically the exact same as L2 regularization.

L1 regularization pushes weights towards exactly zero, encouraging a sparse model. L2 regularization will penalize the weights parameters without making them sparse since the penalty goes to zero for small weights—one reason why L2 is more common.

In `tf.keras`, weight regularization is added by passing weight regularizer instances to layers as keyword arguments. Add L2 weight regularization:

In []:

```
l2_model = tf.keras.Sequential([
    layers.Dense(512, activation='elu',
                 kernel_regularizer=regularizers.l2(0.001),
                 input_shape=(FEATURES,)),
    layers.Dense(512, activation='elu',
                 kernel_regularizer=regularizers.l2(0.001)),
    layers.Dense(512, activation='elu',
                 kernel_regularizer=regularizers.l2(0.001)),
    layers.Dense(512, activation='elu',
                 kernel_regularizer=regularizers.l2(0.001)),
    layers.Dense(1)
])

regularizer_histories['l2'] = compile_and_fit(l2_model, "regularizers/l2")
```

`l2(0.001)` means that every coefficient in the weight matrix of the layer will add `0.001 * weight_coefficient_value**2` to the total `loss` of the network.

That is why we're monitoring the `binary_crossentropy` directly. Because it doesn't have this regularization component mixed in.

So, that same "Large" model with an L2 regularization penalty performs much better:

In []:

```
plotter.plot(regularizer_histories)
plt.ylim([0.5, 0.7])
```

As demonstrated in the diagram above, the "L2" regularized model is now much more competitive with the "Tiny" model. This "L2" model is also much more resistant to overfitting than the "Large" model it was based on despite having the same number of parameters.

More info

There are two important things to note about this sort of regularization:

1. If you are writing your own training loop, then you need to be sure to ask the model for its regularization losses.

In []:

```
result = l2_model(features)
regularization_loss=tf.add_n(l2_model.losses)
```

1. This implementation works by adding the weight penalties to the model's loss, and then applying a standard optimization procedure after that.

There is a second approach that instead only runs the optimizer on the raw loss, and then while applying the calculated step the optimizer also applies some weight decay. This "decoupled weight decay" is used in optimizers like `tf.keras.optimizers.Ftrl` and `tfa.optimizers.AdamW`.

Add dropout

Dropout is one of the most effective and most commonly used regularization techniques for neural networks, developed by Hinton and his students at the University of Toronto.

The intuitive explanation for dropout is that because individual nodes in the network cannot rely on the output of the others, each node must output features that are useful on their own.

Dropout, applied to a layer, consists of randomly "dropping out" (i.e. set to zero) a number of output features of the layer during training. For example, a given layer would normally have returned a vector `[0.2, 0.5, 1.3, 0.8, 1.1]` for a given input sample during training; after applying dropout, this vector will have a few zero entries distributed at random, e.g. `[0, 0.5, 1.3, 0, 1.1]`.

The "dropout rate" is the fraction of the features that are being zeroed-out; it is usually set between 0.2 and 0.5. At test time, no units are dropped out, and instead the layer's output values are scaled down by a factor equal to the dropout rate, so as to balance for the fact that more units are active than at training time.

In Keras, you can introduce dropout in a network via the `tf.keras.layers.Dropout` layer, which gets applied to the output of layer right before.

Add two dropout layers to your network to check how well they do at reducing overfitting:

In []:

```
dropout_model = tf.keras.Sequential([
    layers.Dense(512, activation='elu', input_shape=(FEATURES,)),
    layers.Dropout(0.5),
    layers.Dense(512, activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(512, activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(512, activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(1)
])
regularizer_histories['dropout'] = compile_and_fit(dropout_model, "regularizers/dropout")
```

In []:

```
plotter.plot(regularizer_histories)
plt.ylim([0.5, 0.7])
```

It's clear from this plot that both of these regularization approaches improve the behavior of the "Large" model. But this still doesn't beat even the "Tiny" baseline.

Next try them both, together, and see if that does better.

Combined L2 + dropout

In []:

```
combined_model = tf.keras.Sequential([
    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001),
                 activation='elu', input_shape=(FEATURES,)),
    layers.Dropout(0.5),
    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001),
                 activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001),
                 activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001),
                 activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(1)
])

regularizer_histories['combined'] = compile_and_fit(combined_model, "regularizers/combined")
```

In []:

```
plotter.plot(regularizer_histories)
plt.ylim([0.5, 0.7])
```

This model with the "Combined" regularization is obviously the best one so far.

View in TensorBoard

These models also recorded TensorBoard logs.

To open an embedded tensorboard viewer inside a notebook, copy the following into a code-cell:

```
%tensorboard --logdir {logdir}/regularizers
```

You can view the [results of a previous run](https://tensorboard.dev/experiment/fGInKDo8TXes1z7HQku9mw/#scalars&_smoothingWeight=0.97) (https://tensorboard.dev/experiment/fGInKDo8TXes1z7HQku9mw/#scalars&_smoothingWeight=0.97) of this notebook on TensorBoard.dev (<https://TensorBoard.dev>).

It's also included in an <iframe> for convenience:

In []:

```
display.IFrame(
    src="https://tensorboard.dev/experiment/fGInKDo8TXes1z7HQku9mw/#scalars&_smoothingWeight=0.97",
    width = "100%",
    height="800px")
```

This was uploaded with:

```
!tensorboard dev upload --logdir {logdir}/regularizers
```

Conclusions

To recap, here are the most common ways to prevent overfitting in neural networks:

- Get more training data.
- Reduce the capacity of the network.
- Add weight regularization.
- Add dropout.

Two important approaches not covered in this guide are:

- [Data augmentation](#) ([./images/data_augmentation.ipynb](#))
- Batch normalization (`tf.keras.layers.BatchNormalization`)

Remember that each method can help on its own, but often combining them can be even more effective.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

In []:

```
#@title MIT License
#
# Copyright (c) 2017 François Chollet
#
# Permission is hereby granted, free of charge, to any person obtaining a
# copy of this software and associated documentation files (the "Software"),
# to deal in the Software without restriction, including without limitation
# the rights to use, copy, modify, merge, publish, distribute, sublicense,
# and/or sell copies of the Software, and to permit persons to whom the
# Software is furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
# THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
# FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
# DEALINGS IN THE SOFTWARE.
```

Basic classification: Classify images of clothing



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/keras/classification)

(<https://www.tensorflow.org/tutorials/keras/classification>) (<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/keras/classification.ipynb>) (<https://>)



[Run in Google Colab](#)

This guide trains a neural network model to classify images of clothing, like sneakers and shirts. It's okay if you don't understand all the details; this is a fast-paced overview of a complete TensorFlow program with the details explained as you go.

This guide uses [tf.keras](https://www.tensorflow.org/guide/keras) (<https://www.tensorflow.org/guide/keras>), a high-level API to build and train models in TensorFlow.

In []:

```
# TensorFlow and tf.keras
import tensorflow as tf

# Helper libraries
import numpy as np
import matplotlib.pyplot as plt

print(tf.__version__)
```

Import the Fashion MNIST dataset

This guide uses the [Fashion MNIST](https://github.com/zalandoresearch/fashion-mnist) (<https://github.com/zalandoresearch/fashion-mnist>) dataset which contains 70,000 grayscale images in 10 categories. The images show individual articles of clothing at low resolution (28 by 28 pixels), as seen here:

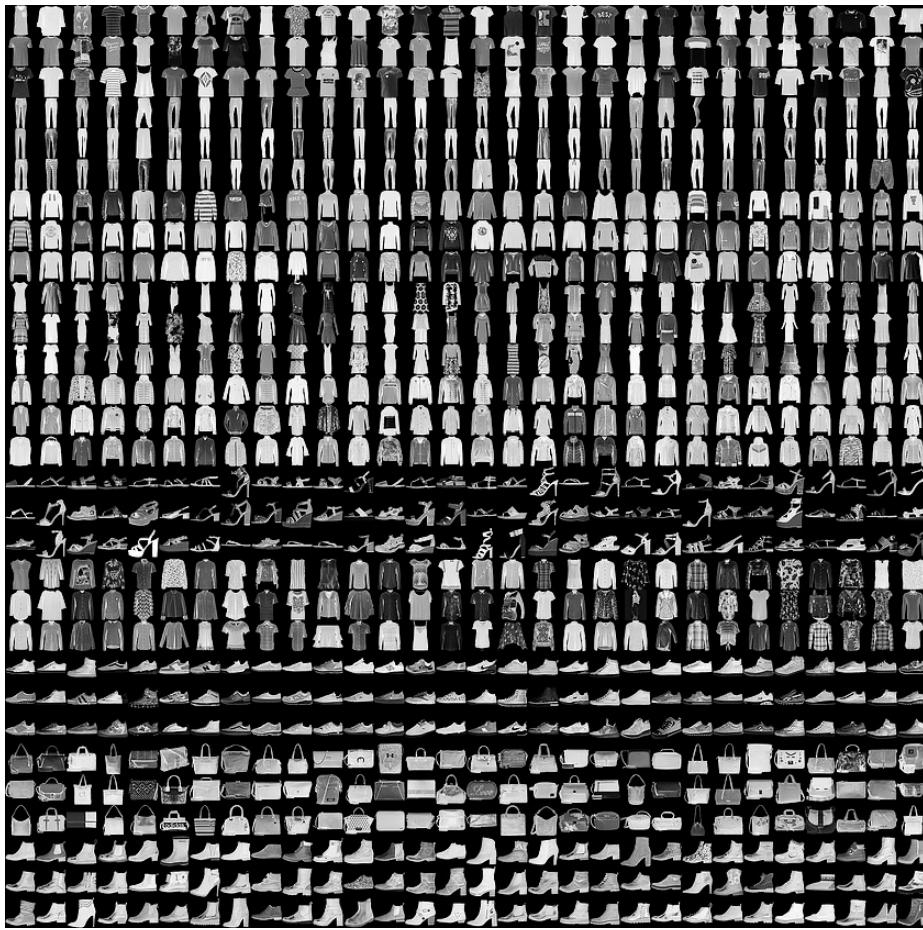


Figure 1. [Fashion-MNIST samples](https://github.com/zalandoresearch/fashion-mnist) (<https://github.com/zalandoresearch/fashion-mnist>) (by Zalando, MIT License).

Fashion MNIST is intended as a drop-in replacement for the classic [MNIST](http://yann.lecun.com/exdb/mnist/) (<http://yann.lecun.com/exdb/mnist/>) dataset—often used as the "Hello, World" of machine learning programs for computer vision. The MNIST dataset contains images of handwritten digits (0, 1, 2, etc.) in a format identical to that of the articles of clothing you'll use here.

This guide uses Fashion MNIST for variety, and because it's a slightly more challenging problem than regular MNIST. Both datasets are relatively small and are used to verify that an algorithm works as expected. They're good starting points to test and debug code.

Here, 60,000 images are used to train the network and 10,000 images to evaluate how accurately the network learned to classify images. You can access the Fashion MNIST directly from TensorFlow. Import and [load the Fashion MNIST data](#) (https://www.tensorflow.org/api_docs/python/tf/keras/datasets/fashion_mnist/load_data) directly from TensorFlow:

In []:

```
fashion_mnist = tf.keras.datasets.fashion_mnist  
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
```

Loading the dataset returns four NumPy arrays:

- The `train_images` and `train_labels` arrays are the *training set*—the data the model uses to learn.
- The model is tested against the *test set*, the `test_images`, and `test_labels` arrays.

The images are 28x28 NumPy arrays, with pixel values ranging from 0 to 255. The *labels* are an array of integers, ranging from 0 to 9. These correspond to the *class* of clothing the image represents:

Label	Class
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Each image is mapped to a single label. Since the *class names* are not included with the dataset, store them here to use later when plotting the images:

In []:

```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

Explore the data

Let's explore the format of the dataset before training the model. The following shows there are 60,000 images in the training set, with each image represented as 28 x 28 pixels:

In []:

```
train_images.shape
```

Likewise, there are 60,000 labels in the training set:

In []:

```
len(train_labels)
```

Each label is an integer between 0 and 9:

In []:

```
train_labels
```

There are 10,000 images in the test set. Again, each image is represented as 28 x 28 pixels:

In []:

```
test_images.shape
```

And the test set contains 10,000 images labels:

In []:

```
len(test_labels)
```

Preprocess the data

The data must be preprocessed before training the network. If you inspect the first image in the training set, you will see that the pixel values fall in the range of 0 to 255:

In []:

```
plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.show()
```

Scale these values to a range of 0 to 1 before feeding them to the neural network model. To do so, divide the values by 255. It's important that the *training set* and the *testing set* be preprocessed in the same way:

In []:

```
train_images = train_images / 255.0
test_images = test_images / 255.0
```

To verify that the data is in the correct format and that you're ready to build and train the network, let's display the first 25 images from the *training set* and display the class name below each image.

In []:

```
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```

Build the model

Building the neural network requires configuring the layers of the model, then compiling the model.

Set up the layers

The basic building block of a neural network is the [layer](https://www.tensorflow.org/api_docs/python/tf/keras/layers) (https://www.tensorflow.org/api_docs/python/tf/keras/layers). Layers extract representations from the data fed into them. Hopefully, these representations are meaningful for the problem at hand.

Most of deep learning consists of chaining together simple layers. Most layers, such as `tf.keras.layers.Dense`, have parameters that are learned during training.

In []:

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
])
```

The first layer in this network, `tf.keras.layers.Flatten`, transforms the format of the images from a two-dimensional array (of 28 by 28 pixels) to a one-dimensional array (of $28 \times 28 = 784$ pixels). Think of this layer as unstacking rows of pixels in the image and lining them up. This layer has no parameters to learn; it only reformats the data.

After the pixels are flattened, the network consists of a sequence of two `tf.keras.layers.Dense` layers. These are densely connected, or fully connected, neural layers. The first `Dense` layer has 128 nodes (or neurons). The second (and last) layer returns a logits array with length of 10. Each node contains a score that indicates the current image belongs to one of the 10 classes.

Compile the model

Before the model is ready for training, it needs a few more settings. These are added during the model's [compile](https://www.tensorflow.org/api_docs/python/tf/keras/Model#compile) (https://www.tensorflow.org/api_docs/python/tf/keras/Model#compile) step:

- [Loss function](https://www.tensorflow.org/api_docs/python/tf/keras/losses) (https://www.tensorflow.org/api_docs/python/tf/keras/losses) —This measures how accurate the model is during training. You want to minimize this function to "steer" the model in the right direction.
- [Optimizer](https://www.tensorflow.org/api_docs/python/tf/keras/optimizers) (https://www.tensorflow.org/api_docs/python/tf/keras/optimizers) —This is how the model is updated based on the data it sees and its loss function.
- [Metrics](https://www.tensorflow.org/api_docs/python/tf/keras/metrics) (https://www.tensorflow.org/api_docs/python/tf/keras/metrics) —Used to monitor the training and testing steps. The following example uses `accuracy`, the fraction of the images that are correctly classified.

In []:

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

Train the model

Training the neural network model requires the following steps:

1. Feed the training data to the model. In this example, the training data is in the `train_images` and `train_labels` arrays.
2. The model learns to associate images and labels.
3. You ask the model to make predictions about a test set—in this example, the `test_images` array.
4. Verify that the predictions match the labels from the `test_labels` array.

Feed the model

To start training, call the `model.fit` (https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit) method—so called because it "fits" the model to the training data:

In []:

```
model.fit(train_images, train_labels, epochs=10)
```

As the model trains, the loss and accuracy metrics are displayed. This model reaches an accuracy of about 0.91 (or 91%) on the training data.

Evaluate accuracy

Next, compare how the model performs on the test dataset:

In []:

```
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
print('\nTest accuracy:', test_acc)
```

It turns out that the accuracy on the test dataset is a little less than the accuracy on the training dataset. This gap between training accuracy and test accuracy represents *overfitting*. Overfitting happens when a machine learning model performs worse on new, previously unseen inputs than it does on the training data. An overfitted model "memorizes" the noise and details in the training dataset to a point where it negatively impacts the performance of the model on the new data. For more information, see the following:

- [Demonstrate overfitting \(https://www.tensorflow.org/tutorials/keras/overfit_and_underfit#demonstrate_overfitting\)](https://www.tensorflow.org/tutorials/keras/overfit_and_underfit#demonstrate_overfitting)
- [Strategies to prevent overfitting \(https://www.tensorflow.org/tutorials/keras/overfit_and_underfit#strategies_to_prevent_overfitting\)](https://www.tensorflow.org/tutorials/keras/overfit_and_underfit#strategies_to_prevent_overfitting)

Make predictions

With the model trained, you can use it to make predictions about some images. Attach a softmax layer to convert the model's linear outputs—`logits` (<https://developers.google.com/machine-learning/glossary/#logits>)—to probabilities, which should be easier to interpret.

In []:

```
probability_model = tf.keras.Sequential([model,
                                         tf.keras.layers.Softmax()])
```

In []:

```
predictions = probability_model.predict(test_images)
```

Here, the model has predicted the label for each image in the testing set. Let's take a look at the first prediction:

In []:

```
predictions[0]
```

A prediction is an array of 10 numbers. They represent the model's "confidence" that the image corresponds to each of the 10 different articles of clothing. You can see which label has the highest confidence value:

In []:

```
np.argmax(predictions[0])
```

So, the model is most confident that this image is an ankle boot, or `class_names[9]`. Examining the test label shows that this classification is correct:

In []:

```
test_labels[0]
```

Graph this to look at the full set of 10 class predictions.

In []:

```
def plot_image(i, predictions_array, true_label, img):
    true_label, img = true_label[i], img[i]
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])

    plt.imshow(img, cmap=plt.cm.binary)

    predicted_label = np.argmax(predictions_array)
    if predicted_label == true_label:
        color = 'blue'
    else:
        color = 'red'

    plt.xlabel("{} {:2.0f}% ({})".format(class_names[predicted_label],
                                           100*np.max(predictions_array),
                                           class_names[true_label]),
               color=color)

def plot_value_array(i, predictions_array, true_label):
    true_label = true_label[i]
    plt.grid(False)
    plt.xticks(range(10))
    plt.yticks([])

    thisplot = plt.bar(range(10), predictions_array, color="#777777")
    plt.ylim([0, 1])
    predicted_label = np.argmax(predictions_array)

    thisplot[predicted_label].set_color('red')
    thisplot[true_label].set_color('blue')
```

Verify predictions

With the model trained, you can use it to make predictions about some images.

Let's look at the 0th image, predictions, and prediction array. Correct prediction labels are blue and incorrect prediction labels are red. The number gives the percentage (out of 100) for the predicted label.

In []:

```
i = 0
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i], test_labels)
plt.show()
```

In []:

```
i = 12
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i], test_labels)
plt.show()
```

Let's plot several images with their predictions. Note that the model can be wrong even when very confident.

In []:

```
# Plot the first X test images, their predicted labels, and the true labels.
# Color correct predictions in blue and incorrect predictions in red.
num_rows = 5
num_cols = 3
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
    plt.subplot(num_rows, 2*num_cols, 2*i+1)
    plot_image(i, predictions[i], test_labels, test_images)
    plt.subplot(num_rows, 2*num_cols, 2*i+2)
    plot_value_array(i, predictions[i], test_labels)
plt.tight_layout()
plt.show()
```

Use the trained model

Finally, use the trained model to make a prediction about a single image.

In []:

```
# Grab an image from the test dataset.
img = test_images[1]

print(img.shape)
```

`tf.keras` models are optimized to make predictions on a *batch*, or collection, of examples at once. Accordingly, even though you're using a single image, you need to add it to a list:

In []:

```
# Add the image to a batch where it's the only member.
img = (np.expand_dims(img,0))

print(img.shape)
```

Now predict the correct label for this image:

In []:

```
predictions_single = probability_model.predict(img)

print(predictions_single)
```

In []:

```
plot_value_array(1, predictions_single[0], test_labels)
_ = plt.xticks(range(10), class_names, rotation=45)
plt.show()
```

`tf.keras.Model.predict` returns a list of lists—one list for each image in the batch of data. Grab the predictions for our (only) image in the batch:

In []:

```
np.argmax(predictions_single[0])
```

And the model predicts a label as expected.

Copyright 2019 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

In []:

```
#@title MIT License
#
# Copyright (c) 2017 François Chollet
#
# Permission is hereby granted, free of charge, to any person obtaining a
# copy of this software and associated documentation files (the "Software"),
# to deal in the Software without restriction, including without limitation
# the rights to use, copy, modify, merge, publish, distribute, sublicense,
# and/or sell copies of the Software, and to permit persons to whom the
# Software is furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
# THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
# FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
# DEALINGS IN THE SOFTWARE.
```

Basic text classification



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/keras/text_classification)

[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/keras/text_classification.ipynb)

This tutorial demonstrates text classification starting from plain text files stored on disk. You'll train a binary classifier to perform sentiment analysis on an IMDB dataset. At the end of the notebook, there is an exercise for you to try, in which you'll train a multi-class classifier to predict the tag for a programming question on Stack Overflow.

In []:

```
import matplotlib.pyplot as plt
import os
import re
import shutil
import string
import tensorflow as tf

from tensorflow.keras import layers
from tensorflow.keras import losses
```

In []:

```
print(tf.__version__)
```

Sentiment analysis

This notebook trains a sentiment analysis model to classify movie reviews as *positive* or *negative*, based on the text of the review. This is an example of *binary*—or two-class—classification, an important and widely applicable kind of machine learning problem.

You'll use the [Large Movie Review Dataset](https://ai.stanford.edu/~amaas/data/sentiment/) (<https://ai.stanford.edu/~amaas/data/sentiment/>) that contains the text of 50,000 movie reviews from the [Internet Movie Database](https://www.imdb.com/) (<https://www.imdb.com/>). These are split into 25,000 reviews for training and 25,000 reviews for testing. The training and testing sets are *balanced*, meaning they contain an equal number of positive and negative reviews.

Download and explore the IMDB dataset

Let's download and extract the dataset, then explore the directory structure.

In []:

```
url = "https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz"

dataset = tf.keras.utils.get_file("aclImdb_v1", url,
                                  untar=True, cache_dir='.',
                                  cache_subdir='')

dataset_dir = os.path.join(os.path.dirname(dataset), 'aclImdb')
```

```
In [ ]:
```

```
os.listdir(dataset_dir)
```

```
In [ ]:
```

```
train_dir = os.path.join(dataset_dir, 'train')
os.listdir(train_dir)
```

The aclImdb/train/pos and aclImdb/train/neg directories contain many text files, each of which is a single movie review. Let's take a look at one of them.

```
In [ ]:
```

```
sample_file = os.path.join(train_dir, 'pos/1181_9.txt')
with open(sample_file) as f:
    print(f.read())
```

Load the dataset

Next, you will load the data off disk and prepare it into a format suitable for training. To do so, you will use the helpful [text_dataset_from_directory](#) (https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/text_dataset_from_directory) utility, which expects a directory structure as follows.

```
main_directory/
...class_a/
.....a_text_1.txt
.....a_text_2.txt
...class_b/
.....b_text_1.txt
.....b_text_2.txt
```

To prepare a dataset for binary classification, you will need two folders on disk, corresponding to `class_a` and `class_b`. These will be the positive and negative movie reviews, which can be found in `aclImdb/train/pos` and `aclImdb/train/neg`. As the IMDB dataset contains additional folders, you will remove them before using this utility.

```
In [ ]:
```

```
remove_dir = os.path.join(train_dir, 'unsup')
shutil.rmtree(remove_dir)
```

Next, you will use the `text_dataset_from_directory` utility to create a labeled `tf.data.Dataset`. [tf.data](#) (<https://www.tensorflow.org/guide/data>) is a powerful collection of tools for working with data.

When running a machine learning experiment, it is a best practice to divide your dataset into three splits: [train](#) (https://developers.google.com/machine-learning/glossary#training_set), [validation](#) (https://developers.google.com/machine-learning/glossary#validation_set), and [test](#) (<https://developers.google.com/machine-learning/glossary#test-set>).

The IMDB dataset has already been divided into train and test, but it lacks a validation set. Let's create a validation set using an 80:20 split of the training data by using the `validation_split` argument below.

```
In [ ]:
```

```
batch_size = 32
seed = 42

raw_train_ds = tf.keras.utils.text_dataset_from_directory(
    'aclImdb/train',
    batch_size=batch_size,
    validation_split=0.2,
    subset='training',
    seed=seed)
```

As you can see above, there are 25,000 examples in the training folder, of which you will use 80% (or 20,000) for training. As you will see in a moment, you can train a model by passing a dataset directly to `model.fit`. If you're new to `tf.data`, you can also iterate over the dataset and print out a few examples as follows.

```
In [ ]:
```

```
for text_batch, label_batch in raw_train_ds.take(1):
    for i in range(3):
        print("Review", text_batch.numpy()[i])
        print("Label", label_batch.numpy()[i])
```

Notice the reviews contain raw text (with punctuation and occasional HTML tags like `
`). You will show how to handle these in the following section.

The labels are 0 or 1. To see which of these correspond to positive and negative movie reviews, you can check the `class_names` property on the dataset.

In []:

```
print("Label 0 corresponds to", raw_train_ds.class_names[0])
print("Label 1 corresponds to", raw_train_ds.class_names[1])
```

Next, you will create a validation and test dataset. You will use the remaining 5,000 reviews from the training set for validation.

Note: When using the `validation_split` and `subset` arguments, make sure to either specify a random seed, or to pass `shuffle=False`, so that the validation and training splits have no overlap.

In []:

```
raw_val_ds = tf.keras.utils.text_dataset_from_directory(
    'aclImdb/train',
    batch_size=batch_size,
    validation_split=0.2,
    subset='validation',
    seed=seed)
```

In []:

```
raw_test_ds = tf.keras.utils.text_dataset_from_directory(
    'aclImdb/test',
    batch_size=batch_size)
```

Prepare the dataset for training

Next, you will standardize, tokenize, and vectorize the data using the helpful `tf.keras.layers.TextVectorization` layer.

Standardization refers to preprocessing the text, typically to remove punctuation or HTML elements to simplify the dataset. Tokenization refers to splitting strings into tokens (for example, splitting a sentence into individual words, by splitting on whitespace). Vectorization refers to converting tokens into numbers so they can be fed into a neural network. All of these tasks can be accomplished with this layer.

As you saw above, the reviews contain various HTML tags like `
`. These tags will not be removed by the default standardizer in the `TextVectorization` layer (which converts text to lowercase and strips punctuation by default, but doesn't strip HTML). You will write a custom standardization function to remove the HTML.

Note: To prevent [training-testing skew](https://developers.google.com/machine-learning/guides/rules-of-ml#training-serving_skew) (https://developers.google.com/machine-learning/guides/rules-of-ml#training-serving_skew) (also known as training-serving skew), it is important to preprocess the data identically at train and test time. To facilitate this, the `TextVectorization` layer can be included directly inside your model, as shown later in this tutorial.

In []:

```
def custom_standardization(input_data):
    lowercase = tf.strings.lower(input_data)
    stripped_html = tf.strings.regex_replace(lowercase, '<br />', ' ')
    return tf.strings.regex_replace(stripped_html,
                                    '[%s]' % re.escape(string.punctuation),
                                    '')
```

Next, you will create a `TextVectorization` layer. You will use this layer to standardize, tokenize, and vectorize our data. You set the `output_mode` to `int` to create unique integer indices for each token.

Note that you're using the default split function, and the custom standardization function you defined above. You'll also define some constants for the model, like an explicit maximum `sequence_length`, which will cause the layer to pad or truncate sequences to exactly `sequence_length` values.

In []:

```
max_features = 10000
sequence_length = 250

vectorize_layer = layers.TextVectorization(
    standardize=custom_standardization,
    max_tokens=max_features,
    output_mode='int',
    output_sequence_length=sequence_length)
```

Next, you will call `adapt` to fit the state of the preprocessing layer to the dataset. This will cause the model to build an index of strings to integers.

Note: It's important to only use your training data when calling adapt (using the test set would leak information).

In []:

```
# Make a text-only dataset (without labels), then call adapt
train_text = raw_train_ds.map(lambda x, y: x)
vectorize_layer.adapt(train_text)
```

Let's create a function to see the result of using this layer to preprocess some data.

In []:

```
def vectorize_text(text, label):
    text = tf.expand_dims(text, -1)
    return vectorize_layer(text), label
```

In []:

```
# retrieve a batch (of 32 reviews and labels) from the dataset
text_batch, label_batch = next(iter(raw_train_ds))
first_review, first_label = text_batch[0], label_batch[0]
print("Review", first_review)
print("Label", raw_train_ds.class_names[first_label])
print("Vectorized review", vectorize_text(first_review, first_label))
```

As you can see above, each token has been replaced by an integer. You can lookup the token (string) that each integer corresponds to by calling .get_vocabulary() on the layer.

In []:

```
print("1287 ---> ",vectorize_layer.get_vocabulary()[1287])
print(" 313 ---> ",vectorize_layer.get_vocabulary()[313])
print('Vocabulary size: {}'.format(len(vectorize_layer.get_vocabulary())))
```

You are nearly ready to train your model. As a final preprocessing step, you will apply the TextVectorization layer you created earlier to the train, validation, and test dataset.

In []:

```
train_ds = raw_train_ds.map(vectorize_text)
val_ds = raw_val_ds.map(vectorize_text)
test_ds = raw_test_ds.map(vectorize_text)
```

Configure the dataset for performance

These are two important methods you should use when loading data to make sure that I/O does not become blocking.

.cache() keeps data in memory after it's loaded off disk. This will ensure the dataset does not become a bottleneck while training your model. If your dataset is too large to fit into memory, you can also use this method to create a performant on-disk cache, which is more efficient to read than many small files.

.prefetch() overlaps data preprocessing and model execution while training.

You can learn more about both methods, as well as how to cache data to disk in the [data performance guide](#) (https://www.tensorflow.org/guide/data_performance).

In []:

```
AUTOTUNE = tf.data.AUTOTUNE

train_ds = train_ds.cache().prefetch(buffer_size=AUTOTUNE)
val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)
test_ds = test_ds.cache().prefetch(buffer_size=AUTOTUNE)
```

Create the model

It's time to create your neural network:

In []:

```
embedding_dim = 16
```

In []:

```
model = tf.keras.Sequential([
    layers.Embedding(max_features + 1, embedding_dim),
    layers.Dropout(0.2),
    layers.GlobalAveragePooling1D(),
    layers.Dropout(0.2),
    layers.Dense(1)])
model.summary()
```

The layers are stacked sequentially to build the classifier:

1. The first layer is an `Embedding` layer. This layer takes the integer-encoded reviews and looks up an embedding vector for each word-index. These vectors are learned as the model trains. The vectors add a dimension to the output array. The resulting dimensions are: (`batch, sequence, embedding`) . To learn more about embeddings, check out the [Word embeddings \(https://www.tensorflow.org/text/guide/word_embeddings\)](https://www.tensorflow.org/text/guide/word_embeddings) tutorial.
2. Next, a `GlobalAveragePooling1D` layer returns a fixed-length output vector for each example by averaging over the sequence dimension. This allows the model to handle input of variable length, in the simplest way possible.
3. This fixed-length output vector is piped through a fully-connected (`Dense`) layer with 16 hidden units.
4. The last layer is densely connected with a single output node.

Loss function and optimizer

A model needs a loss function and an optimizer for training. Since this is a binary classification problem and the model outputs a probability (a single-unit layer with a sigmoid activation), you'll use `losses.BinaryCrossentropy` loss function.

Now, configure the model to use an optimizer and a loss function:

In []:

```
model.compile(loss=losses.BinaryCrossentropy(from_logits=True),
              optimizer='adam',
              metrics=tf.metrics.BinaryAccuracy(threshold=0.0))
```

Train the model

You will train the model by passing the `dataset` object to the `fit` method.

In []:

```
epochs = 10
history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=epochs)
```

Evaluate the model

Let's see how the model performs. Two values will be returned. Loss (a number which represents our error, lower values are better), and accuracy.

In []:

```
loss, accuracy = model.evaluate(test_ds)

print("Loss: ", loss)
print("Accuracy: ", accuracy)
```

This fairly naive approach achieves an accuracy of about 86%.

Create a plot of accuracy and loss over time

`model.fit()` returns a `History` object that contains a dictionary with everything that happened during training:

In []:

```
history_dict = history.history
history_dict.keys()
```

There are four entries: one for each monitored metric during training and validation. You can use these to plot the training and validation loss for comparison, as well as the training and validation accuracy:

In []:

```
acc = history_dict['binary_accuracy']
val_acc = history_dict['val_binary_accuracy']
loss = history_dict['loss']
val_loss = history_dict['val_loss']

epochs = range(1, len(acc) + 1)

# "bo" is for "blue dot"
plt.plot(epochs, loss, 'bo', label='Training loss')
# b is for "solid blue line"
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

In []:

```
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')

plt.show()
```

In this plot, the dots represent the training loss and accuracy, and the solid lines are the validation loss and accuracy.

Notice the training loss *decreases* with each epoch and the training accuracy *increases* with each epoch. This is expected when using a gradient descent optimization—it should minimize the desired quantity on every iteration.

This isn't the case for the validation loss and accuracy—they seem to peak before the training accuracy. This is an example of overfitting: the model performs better on the training data than it does on data it has never seen before. After this point, the model over-optimizes and learns representations *specific* to the training data that do not *generalize* to test data.

For this particular case, you could prevent overfitting by simply stopping the training when the validation accuracy is no longer increasing. One way to do so is to use the `tf.keras.callbacks.EarlyStopping` callback.

Export the model

In the code above, you applied the `TextVectorization` layer to the dataset before feeding text to the model. If you want to make your model capable of processing raw strings (for example, to simplify deploying it), you can include the `TextVectorization` layer inside your model. To do so, you can create a new model using the weights you just trained.

In []:

```
export_model = tf.keras.Sequential([
    vectorize_layer,
    model,
    layers.Activation('sigmoid')
])

export_model.compile(
    loss=losses.BinaryCrossentropy(from_logits=False), optimizer="adam", metrics=['accuracy']
)

# Test it with `raw_test_ds`, which yields raw strings
loss, accuracy = export_model.evaluate(raw_test_ds)
print(accuracy)
```

Inference on new data

To get predictions for new examples, you can simply call `model.predict()`.

In []:

```
examples = [
    "The movie was great!",
    "The movie was okay.",
    "The movie was terrible..."
]
export_model.predict(examples)
```

Including the text preprocessing logic inside your model enables you to export a model for production that simplifies deployment, and reduces the potential for [train/test skew](https://developers.google.com/machine-learning/guides/rules-of-ml#training-serving_skew) (https://developers.google.com/machine-learning/guides/rules-of-ml#training-serving_skew).

There is a performance difference to keep in mind when choosing where to apply your TextVectorization layer. Using it outside of your model enables you to do asynchronous CPU processing and buffering of your data when training on GPU. So, if you're training your model on the GPU, you probably want to go with this option to get the best performance while developing your model, then switch to including the TextVectorization layer inside your model when you're ready to prepare for deployment.

Visit this [tutorial](https://www.tensorflow.org/tutorials/keras/save_and_load) (https://www.tensorflow.org/tutorials/keras/save_and_load) to learn more about saving models.

Exercise: multi-class classification on Stack Overflow questions

This tutorial showed how to train a binary classifier from scratch on the IMDB dataset. As an exercise, you can modify this notebook to train a multi-class classifier to predict the tag of a programming question on [Stack Overflow](http://stackoverflow.com/) (<http://stackoverflow.com/>).

A [dataset](https://storage.googleapis.com/download.tensorflow.org/data/stack_overflow_16k.tar.gz) (https://storage.googleapis.com/download.tensorflow.org/data/stack_overflow_16k.tar.gz) has been prepared for you to use containing the body of several thousand programming questions (for example, "How can I sort a dictionary by value in Python?") posted to Stack Overflow. Each of these is labeled with exactly one tag (either Python, CSharp, JavaScript, or Java). Your task is to take a question as input, and predict the appropriate tag, in this case, Python.

The dataset you will work with contains several thousand questions extracted from the much larger public Stack Overflow dataset on [BigQuery](https://console.cloud.google.com/marketplace/details/stack-exchange/stack-overflow) (<https://console.cloud.google.com/marketplace/details/stack-exchange/stack-overflow>), which contains more than 17 million posts.

After downloading the dataset, you will find it has a similar directory structure to the IMDB dataset you worked with previously:

```
train/
...python/
.....0.txt
.....1.txt
...javascript/
.....0.txt
.....1.txt
...csharp/
.....0.txt
.....1.txt
...java/
.....0.txt
.....1.txt
```

Note: To increase the difficulty of the classification problem, occurrences of the words Python, CSharp, JavaScript, or Java in the programming questions have been replaced with the word *blank* (as many questions contain the language they're about).

To complete this exercise, you should modify this notebook to work with the Stack Overflow dataset by making the following modifications:

1. At the top of your notebook, update the code that downloads the IMDB dataset with code to download the [Stack Overflow dataset](https://storage.googleapis.com/download.tensorflow.org/data/stack_overflow_16k.tar.gz) (https://storage.googleapis.com/download.tensorflow.org/data/stack_overflow_16k.tar.gz) that has already been prepared. As the Stack Overflow dataset has a similar directory structure, you will not need to make many modifications.
2. Modify the last layer of your model to `Dense(4)`, as there are now four output classes.
3. When compiling the model, change the loss to `tf.keras.losses.SparseCategoricalCrossentropy`. This is the correct loss function to use for a multi-class classification problem, when the labels for each class are integers (in this case, they can be 0, 1, 2, or 3). In addition, change the metrics to `metrics=['accuracy']`, since this is a multi-class classification problem (`tf.metrics.BinaryAccuracy` is only used for binary classifiers).
4. When plotting accuracy over time, change `binary_accuracy` and `val_binary_accuracy` to `accuracy` and `val_accuracy`, respectively.
5. Once these changes are complete, you will be able to train a multi-class classifier.

Learning more

This tutorial introduced text classification from scratch. To learn more about the text classification workflow in general, check out the [Text classification guide](https://developers.google.com/machine-learning/guides/text-classification/) (<https://developers.google.com/machine-learning/guides/text-classification/>) from Google Developers.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

In []:

```
#@title MIT License
#
# Copyright (c) 2017 François Chollet
#
# Permission is hereby granted, free of charge, to any person obtaining a
# copy of this software and associated documentation files (the "Software"),
# to deal in the Software without restriction, including without limitation
# the rights to use, copy, modify, merge, publish, distribute, sublicense,
# and/or sell copies of the Software, and to permit persons to whom the
# Software is furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
# THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
# FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
# DEALINGS IN THE SOFTWARE.
```

Save and load models



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/keras/save_and_load)

(https://www.tensorflow.org/tutorials/keras/save_and_load)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/keras/save_and_load.ipynb)

(https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/keras/save_and_load.ipynb) (h)

Model progress can be saved during and after training. This means a model can resume where it left off and avoid long training times. Saving also means you can share your model and others can recreate your work. When publishing research models and techniques, most machine learning practitioners share:

- code to create the model, and
- the trained weights, or parameters, for the model

Sharing this data helps others understand how the model works and try it themselves with new data.

Caution: TensorFlow models are code and it is important to be careful with untrusted code. See [Using TensorFlow Securely](https://github.com/tensorflow/tensorflow/blob/master/SECURITY.md) (<https://github.com/tensorflow/tensorflow/blob/master/SECURITY.md>) for details.

Options

There are different ways to save TensorFlow models depending on the API you're using. This guide uses [tf.keras](https://www.tensorflow.org/guide/keras) (<https://www.tensorflow.org/guide/keras>), a high-level API to build and train models in TensorFlow. For other approaches see the TensorFlow [Save and Restore](#) (https://www.tensorflow.org/guide/saved_model) guide or [Saving in eager](#) (https://www.tensorflow.org/guide/eager#object-based_saving).

Setup

Installs and imports

Install and import TensorFlow and dependencies:

```
In [ ]:
```

```
!pip install pyyaml h5py # Required to save models in HDF5 format
```

```
In [ ]:
```

```
import os  
  
import tensorflow as tf  
from tensorflow import keras  
  
print(tf.version.VERSION)
```

Get an example dataset

To demonstrate how to save and load weights, you'll use the [MNIST dataset](http://yann.lecun.com/exdb/mnist/) (<http://yann.lecun.com/exdb/mnist/>). To speed up these runs, use the first 1000 examples:

```
In [ ]:
```

```
(train_images, train_labels), (test_images, test_labels) = tf.keras.datasets.mnist.load_data()  
  
train_labels = train_labels[:1000]  
test_labels = test_labels[:1000]  
  
train_images = train_images[:1000].reshape(-1, 28 * 28) / 255.0  
test_images = test_images[:1000].reshape(-1, 28 * 28) / 255.0
```

Define a model

Start by building a simple sequential model:

```
In [ ]:
```

```
# Define a simple sequential model  
def create_model():  
    model = tf.keras.models.Sequential([  
        keras.layers.Dense(512, activation='relu', input_shape=(784,)),  
        keras.layers.Dropout(0.2),  
        keras.layers.Dense(10)  
    ])  
  
    model.compile(optimizer='adam',  
                  loss=tf.losses.SparseCategoricalCrossentropy(from_logits=True),  
                  metrics=[tf.metrics.SparseCategoricalAccuracy()])  
  
    return model  
  
# Create a basic model instance  
model = create_model()  
  
# Display the model's architecture  
model.summary()
```

Save checkpoints during training

You can use a trained model without having to retrain it, or pick-up training where you left off in case the training process was interrupted. The `tf.keras.callbacks.ModelCheckpoint` callback allows you to continually save the model both *during* and at *the end* of training.

Checkpoint callback usage

Create a `tf.keras.callbacks.ModelCheckpoint` callback that saves weights only during training:

In []:

```
checkpoint_path = "training_1/cp.ckpt"
checkpoint_dir = os.path.dirname(checkpoint_path)

# Create a callback that saves the model's weights
cp_callback = tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_path,
                                                save_weights_only=True,
                                                verbose=1)

# Train the model with the new callback
model.fit(train_images,
          train_labels,
          epochs=10,
          validation_data=(test_images, test_labels),
          callbacks=[cp_callback]) # Pass callback to training

# This may generate warnings related to saving the state of the optimizer.
# These warnings (and similar warnings throughout this notebook)
# are in place to discourage outdated usage, and can be ignored.
```

This creates a single collection of TensorFlow checkpoint files that are updated at the end of each epoch:

In []:

```
os.listdir(checkpoint_dir)
```

As long as two models share the same architecture you can share weights between them. So, when restoring a model from weights-only, create a model with the same architecture as the original model and then set its weights.

Now rebuild a fresh, untrained model and evaluate it on the test set. An untrained model will perform at chance levels (~10% accuracy):

In []:

```
# Create a basic model instance
model = create_model()

# Evaluate the model
loss, acc = model.evaluate(test_images, test_labels, verbose=2)
print("Untrained model, accuracy: {:.2f}%".format(100 * acc))
```

Then load the weights from the checkpoint and re-evaluate:

In []:

```
# Loads the weights
model.load_weights(checkpoint_path)

# Re-evaluate the model
loss, acc = model.evaluate(test_images, test_labels, verbose=2)
print("Restored model, accuracy: {:.2f}%".format(100 * acc))
```

Checkpoint callback options

The callback provides several options to provide unique names for checkpoints and adjust the checkpointing frequency.

Train a new model, and save uniquely named checkpoints once every five epochs:

In []:

```
# Include the epoch in the file name (uses `str.format`)
checkpoint_path = "training_2/cp-{epoch:04d}.ckpt"
checkpoint_dir = os.path.dirname(checkpoint_path)

batch_size = 32

# Create a callback that saves the model's weights every 5 epochs
cp_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_path,
    verbose=1,
    save_weights_only=True,
    save_freq=5*batch_size)

# Create a new model instance
model = create_model()

# Save the weights using the `checkpoint_path` format
model.save_weights(checkpoint_path.format(epoch=0))

# Train the model with the new callback
model.fit(train_images,
          train_labels,
          epochs=50,
          batch_size=batch_size,
          callbacks=[cp_callback],
          validation_data=(test_images, test_labels),
          verbose=0)
```

Now, look at the resulting checkpoints and choose the latest one:

In []:

```
os.listdir(checkpoint_dir)
```

In []:

```
latest = tf.train.latest_checkpoint(checkpoint_dir)
latest
```

Note: the default TensorFlow format only saves the 5 most recent checkpoints.

To test, reset the model and load the latest checkpoint:

In []:

```
# Create a new model instance
model = create_model()

# Load the previously saved weights
model.load_weights(latest)

# Re-evaluate the model
loss, acc = model.evaluate(test_images, test_labels, verbose=2)
print("Restored model, accuracy: {:.2f}%".format(100 * acc))
```

What are these files?

The above code stores the weights to a collection of [checkpoint \(..../guide/checkpoint.ipynb\)](#)-formatted files that contain only the trained weights in a binary format. Checkpoints contain:

- One or more shards that contain your model's weights.
- An index file that indicates which weights are stored in which shard.

If you are training a model on a single machine, you'll have one shard with the suffix: .data-00000-of-00001

Manually save weights

To save weights manually, use `tf.keras.Model.save_weights`. By default, `tf.keras` —and the `Model.save_weights` method in particular— uses the TensorFlow [Checkpoint \(..../guide/checkpoint.ipynb\)](#) format with a `.ckpt` extension. To save in the HDF5 format with a `.h5` extension, refer to the [Save and load models \(\[https://www.tensorflow.org/guide/keras/save_and_serialize\]\(https://www.tensorflow.org/guide/keras/save_and_serialize\)\)](#) guide.

In []:

```
# Save the weights
model.save_weights('./checkpoints/my_checkpoint')

# Create a new model instance
model = create_model()

# Restore the weights
model.load_weights('./checkpoints/my_checkpoint')

# Evaluate the model
loss, acc = model.evaluate(test_images, test_labels, verbose=2)
print("Restored model, accuracy: {:.2f}%".format(100 * acc))
```

Save the entire model

Call `model.save` (https://www.tensorflow.org/api_docs/python/tf/keras/Model#save) to save a model's architecture, weights, and training configuration in a single file/folder. This allows you to export a model so it can be used without access to the original Python code*. Since the optimizer-state is recovered, you can resume training from exactly where you left off.

An entire model can be saved in two different file formats (`SavedModel` and `HDF5`). The TensorFlow `SavedModel` format is the default file format in TF2.x. However, models can be saved in `HDF5` format. More details on saving entire models in the two file formats is described below.

Saving a fully-functional model is very useful—you can load them in TensorFlow.js ([Saved Model](#) (https://www.tensorflow.org/js/tutorials/conversion/import_saved_model), [HDF5](#) (https://www.tensorflow.org/js/tutorials/conversion/import_keras)) and then train and run them in web browsers, or convert them to run on mobile devices using TensorFlow Lite ([Saved Model](#) (https://www.tensorflow.org/lite/convert/python_api#converting_a_savedmodel_), [HDF5](#) (https://www.tensorflow.org/lite/convert/python_api#converting_a_keras_model_))

*Custom objects (e.g. subclassed models or layers) require special attention when saving and loading. See the **Saving custom objects** section below

SavedModel format

The `SavedModel` format is another way to serialize models. Models saved in this format can be restored using `tf.keras.models.load_model` and are compatible with TensorFlow Serving. The [SavedModel guide](#) (https://www.tensorflow.org/guide/saved_model) goes into detail about how to serve/inspect the `SavedModel`. The section below illustrates the steps to save and restore the model.

In []:

```
# Create and train a new model instance.
model = create_model()
model.fit(train_images, train_labels, epochs=5)

# Save the entire model as a SavedModel.
!mkdir -p saved_model
model.save('saved_model/my_model')
```

The `SavedModel` format is a directory containing a protobuf binary and a TensorFlow checkpoint. Inspect the saved model directory:

In []:

```
# my_model directory
!ls saved_model

# Contains an assets folder, saved_model.pb, and variables folder.
!ls saved_model/my_model
```

Reload a fresh Keras model from the saved model:

In []:

```
new_model = tf.keras.models.load_model('saved_model/my_model')

# Check its architecture
new_model.summary()
```

The restored model is compiled with the same arguments as the original model. Try running `evaluate` and `predict` with the loaded model:

In []:

```
# Evaluate the restored model
loss, acc = new_model.evaluate(test_images, test_labels, verbose=2)
print('Restored model, accuracy: {:.2f}%'.format(100 * acc))

print(new_model.predict(test_images).shape)
```

HDF5 format

Keras provides a basic save format using the [HDF5](https://en.wikipedia.org/wiki/Hierarchical_Data_Format) (https://en.wikipedia.org/wiki/Hierarchical_Data_Format) standard.

In []:

```
# Create and train a new model instance.
model = create_model()
model.fit(train_images, train_labels, epochs=5)

# Save the entire model to a HDF5 file.
# The '.h5' extension indicates that the model should be saved to HDF5.
model.save('my_model.h5')
```

Now, recreate the model from that file:

In []:

```
# Recreate the exact same model, including its weights and the optimizer
new_model = tf.keras.models.load_model('my_model.h5')

# Show the model architecture
new_model.summary()
```

Check its accuracy:

In []:

```
loss, acc = new_model.evaluate(test_images, test_labels, verbose=2)
print('Restored model, accuracy: {:.2f}%'.format(100 * acc))
```

Keras saves models by inspecting their architectures. This technique saves everything:

- The weight values
- The model's architecture
- The model's training configuration (what you pass to the `.compile()` method)
- The optimizer and its state, if any (this enables you to restart training where you left off)

Keras is not able to save the `v1.x` optimizers (from `tf.compat.v1.train`) since they aren't compatible with checkpoints. For `v1.x` optimizers, you need to re-compile the model after loading—losing the state of the optimizer.

Saving custom objects

If you are using the SavedModel format, you can skip this section. The key difference between HDF5 and SavedModel is that HDF5 uses object configs to save the model architecture, while SavedModel saves the execution graph. Thus, SavedModels are able to save custom objects like subclassed models and custom layers without requiring the original code.

To save custom objects to HDF5, you must do the following:

1. Define a `get_config` method in your object, and optionally a `from_config` classmethod.
 - `get_config(self)` returns a JSON-serializable dictionary of parameters needed to recreate the object.
 - `from_config(cls, config)` uses the returned config from `get_config` to create a new object. By default, this function will use the config as initialization kwargs (`return cls(**config)`).
2. Pass the object to the `custom_objects` argument when loading the model. The argument must be a dictionary mapping the string class name to the Python class. E.g. `tf.keras.models.load_model(path, custom_objects={'CustomLayer': CustomLayer})`

See the [Writing layers and models from scratch](https://www.tensorflow.org/guide/keras/custom_layers_and_models) (https://www.tensorflow.org/guide/keras/custom_layers_and_models) tutorial for examples of custom objects and `get_config`.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Customization basics: tensors and operations



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/customization/basics)

(<https://www.tensorflow.org/tutorials/customization/basics>)



[Run in Google Colab](#)

(<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/customization/basics.ipynb>) ([http](#)

This is an introductory TensorFlow tutorial that shows how to:

- Import the required package
- Create and use tensors
- Use GPU acceleration
- Demonstrate `tf.data.Dataset`

Import TensorFlow

To get started, import the `tensorflow` module. As of TensorFlow 2, eager execution is turned on by default. Eager execution enables a more interactive frontend to TensorFlow, which you will later explore in more detail.

In []:

```
import tensorflow as tf
```

Tensors

A Tensor is a multi-dimensional array. Similar to NumPy `ndarray` objects, `tf.Tensor` objects have a data type and a shape. Additionally, `tf.Tensor`s can reside in accelerator memory (like a GPU). TensorFlow offers a rich library of operations ([tf.add](#) (https://www.tensorflow.org/api_docs/python/tf/add), [tf.matmul](#) (https://www.tensorflow.org/api_docs/python/tf/matmul), [tf.linalg.inv](#) (https://www.tensorflow.org/api_docs/python/tf/linalg/inv) etc.) that consume and produce `tf.Tensor`s. These operations automatically convert built-in Python types, for example:

In []:

```
print(tf.add(1, 2))
print(tf.add([1, 2], [3, 4]))
print(tf.square(5))
print(tf.reduce_sum([1, 2, 3]))

# Operator overloading is also supported
print(tf.square(2) + tf.square(3))
```

Each `tf.Tensor` has a shape and a datatype:

In []:

```
x = tf.matmul([[1]], [[2, 3]])
print(x)
print(x.shape)
print(x.dtype)
```

The most obvious differences between NumPy arrays and `tf.Tensor`s are:

1. Tensors can be backed by accelerator memory (like GPU, TPU).
2. Tensors are immutable.

NumPy Compatibility

Converting between a TensorFlow `tf.Tensor`s and a NumPy `ndarray` is easy:

- TensorFlow operations automatically convert NumPy ndarrays to Tensors.
- NumPy operations automatically convert Tensors to NumPy ndarrays.

Tensors are explicitly converted to NumPy ndarrays using their `.numpy()` method. These conversions are typically cheap since the array and `tf.Tensor` share the underlying memory representation, if possible. However, sharing the underlying representation isn't always possible since the `tf.Tensor` may be hosted in GPU memory while NumPy arrays are always backed by host memory, and the conversion involves a copy from GPU to host memory.

In []:

```
import numpy as np

ndarray = np.ones([3, 3])

print("TensorFlow operations convert numpy arrays to Tensors automatically")
tensor = tf.multiply(ndarray, 42)
print(tensor)

print("And NumPy operations convert Tensors to numpy arrays automatically")
print(np.add(tensor, 1))

print("The .numpy() method explicitly converts a Tensor to a numpy array")
print(tensor.numpy())
```

GPU acceleration

Many TensorFlow operations are accelerated using the GPU for computation. Without any annotations, TensorFlow automatically decides whether to use the GPU or CPU for an operation—copying the tensor between CPU and GPU memory, if necessary. Tensors produced by an operation are typically backed by the memory of the device on which the operation executed, for example:

In []:

```
x = tf.random.uniform([3, 3])

print("Is there a GPU available: "),
print(tf.config.list_physical_devices("GPU"))

print("Is the Tensor on GPU #0:  "),
print(x.device.endswith('GPU:0'))
```

Device Names

The `Tensor.device` property provides a fully qualified string name of the device hosting the contents of the tensor. This name encodes many details, such as an identifier of the network address of the host on which this program is executing and the device within that host. This is required for distributed execution of a TensorFlow program. The string ends with `GPU:<N>` if the tensor is placed on the `N`-th GPU on the host.

Explicit Device Placement

In TensorFlow, *placement* refers to how individual operations are assigned (placed on) a device for execution. As mentioned, when there is no explicit guidance provided, TensorFlow automatically decides which device to execute an operation and copies tensors to that device, if needed. However, TensorFlow operations can be explicitly placed on specific devices using the `tf.device` context manager, for example:

```
In [ ]:
```

```
import time

def time_matmul(x):
    start = time.time()
    for loop in range(10):
        tf.matmul(x, x)

    result = time.time() - start

    print("10 loops: {:.2f}ms".format(1000 * result))

# Force execution on CPU
print("On CPU:")
with tf.device("CPU:0"):
    x = tf.random.uniform([1000, 1000])
    assert x.device.endswith("CPU:0")
    time_matmul(x)

# Force execution on GPU #0 if available
if tf.config.list_physical_devices("GPU"):
    print("On GPU:")
    with tf.device("GPU:0"): # Or GPU:1 for the 2nd GPU, GPU:2 for the 3rd etc.
        x = tf.random.uniform([1000, 1000])
        assert x.device.endswith("GPU:0")
        time_matmul(x)
```

Datasets

This section uses the `tf.data.Dataset` API (<https://www.tensorflow.org/guide/datasets>) to build a pipeline for feeding data to your model. The `tf.data.Dataset` API is used to build performant, complex input pipelines from simple, re-usable pieces that will feed your model's training or evaluation loops.

Create a source Dataset

Create a source dataset using one of the factory functions like `Dataset.from_tensors` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#from_tensors), `Dataset.from_tensor_slices` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#from_tensor_slices), or using objects that read from files like `TextLineDataset` (https://www.tensorflow.org/api_docs/python/tf/data/TextLineDataset) or `TFRecordDataset` (https://www.tensorflow.org/api_docs/python/tf/data/TFRecordDataset). See the `TensorFlow Dataset guide` (https://www.tensorflow.org/guide/datasets#reading_input_data) for more information.

```
In [ ]:
```

```
ds_tensors = tf.data.Dataset.from_tensor_slices([1, 2, 3, 4, 5, 6])

# Create a CSV file
import tempfile
_, filename = tempfile.mkstemp()

with open(filename, 'w') as f:
    f.write("""Line 1
Line 2
Line 3
""")

ds_file = tf.data.TextLineDataset(filename)
```

Apply transformations

Use the transformations functions like `map` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#map), `batch` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#batch), and `shuffle` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#shuffle) to apply transformations to dataset records.

```
In [ ]:
```

```
ds_tensors = ds_tensors.map(tf.square).shuffle(2).batch(2)
ds_file = ds_file.batch(2)
```

Iterate

`tf.data.Dataset` objects support iteration to loop over records:

In []:

```
print('Elements of ds_tensors:')
for x in ds_tensors:
    print(x)

print('\nElements in ds_file:')
for x in ds_file:
    print(x)
```

Copyright 2018 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Custom training: walkthrough



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/customization/custom_training_walkthrough)

(https://www.tensorflow.org/tutorials/customization/custom_training_walkthrough) (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/customization/custom_training_walkthrough.ipynb)



This tutorial shows you how to train a machine learning model with a custom training loop to categorize penguins by species. In this notebook, you use TensorFlow to accomplish the following:

1. Import a dataset
2. Build a simple linear model
3. Train the model
4. Evaluate the model's effectiveness
5. Use the trained model to make predictions

TensorFlow programming

This tutorial demonstrates the following TensorFlow programming tasks:

- Importing data with the [TensorFlow Datasets API](https://www.tensorflow.org/datasets/overview#load_a_dataset) (https://www.tensorflow.org/datasets/overview#load_a_dataset)
- Building models and layers with the [Keras API](https://www.tensorflow.org/guide/keras/) (<https://www.tensorflow.org/guide/keras/>)

Penguin classification problem

Imagine you are an ornithologist seeking an automated way to categorize each penguin you find. Machine learning provides many algorithms to classify penguins statistically. For instance, a sophisticated machine learning program could classify penguins based on photographs. The model you build in this tutorial is a little simpler. It classifies penguins based on their body weight, flipper length, and beaks, specifically the length and width measurements of their [culmen](https://en.wikipedia.org/wiki/Beak#Culmen) (<https://en.wikipedia.org/wiki/Beak#Culmen>).

There are 18 species of penguins, but in this tutorial you will only attempt to classify the following three:

- Chinstrap penguins
- Gentoo penguins
- Adélie penguins

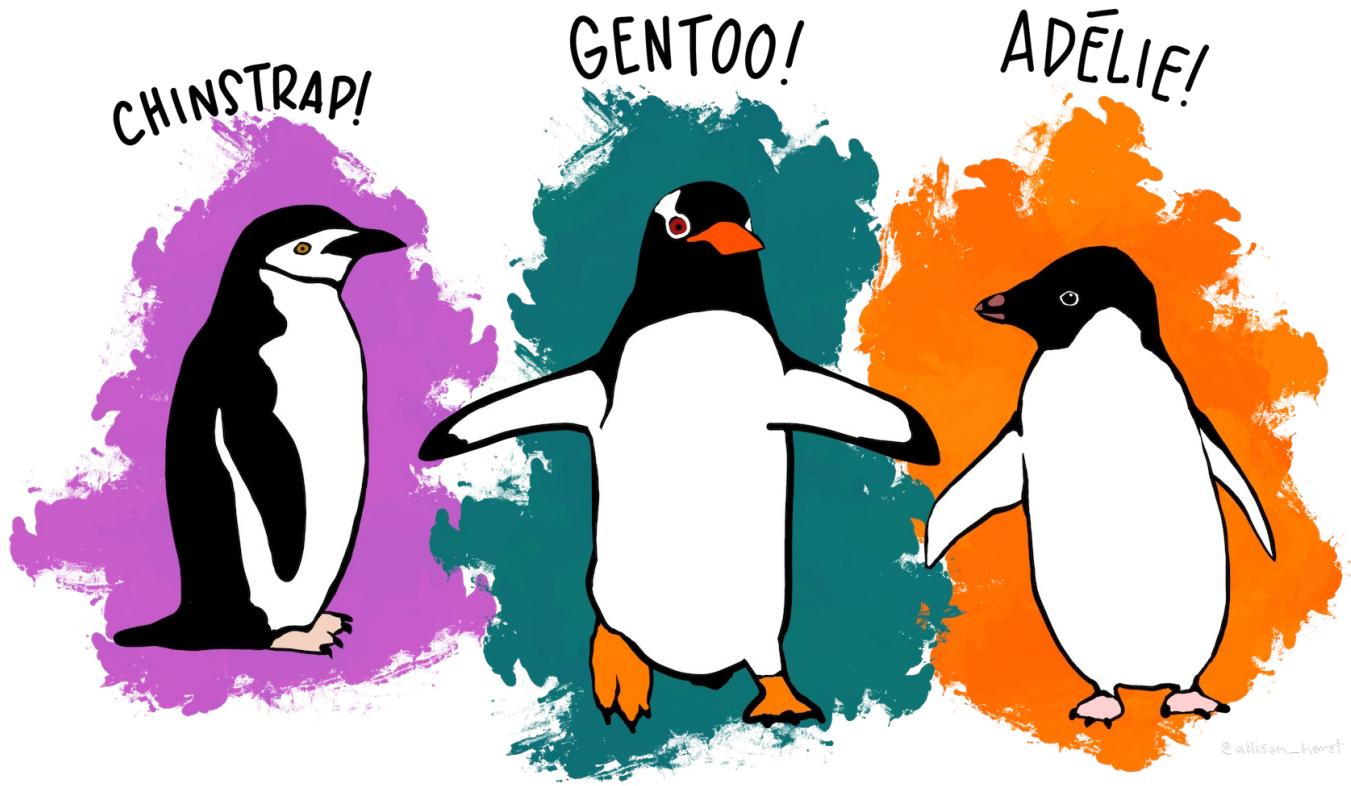


Figure 1. [Chinstrap](https://en.wikipedia.org/wiki/Chinstrap_penguin) (https://en.wikipedia.org/wiki/Chinstrap_penguin), [Gentoo](https://en.wikipedia.org/wiki/Gentoo_penguin) (https://en.wikipedia.org/wiki/Gentoo_penguin), and [Adélie](https://en.wikipedia.org/wiki/Ad%C3%A9lie_penguin) (https://en.wikipedia.org/wiki/Ad%C3%A9lie_penguin) penguins (Artwork by @allison_horst, CC BY-SA 2.0).

Fortunately, a research team has already created and shared a [dataset of 334 penguins](https://allisonhorst.github.io/palmerpenguins/) (<https://allisonhorst.github.io/palmerpenguins/>) with body weight, flipper length, beak measurements, and other data. This dataset is also conveniently available as the [penguins](https://www.tensorflow.org/datasets/catalog/penguins) (<https://www.tensorflow.org/datasets/catalog/penguins>) TensorFlow Dataset.

Setup

Install the `tfds-nightly` package for the penguins dataset. The `tfds-nightly` package is the nightly released version of the TensorFlow Datasets (TFDS). For more information on TFDS, see [TensorFlow Datasets overview](https://www.tensorflow.org/datasets/overview) (<https://www.tensorflow.org/datasets/overview>).

In []:

```
!pip install -q tfds-nightly
```

Then select **Runtime > Restart Runtime** from the Colab menu to restart the Colab runtime.

Do not proceed with the rest of this tutorial without first restarting the runtime.

Import TensorFlow and the other required Python modules.

In []:

```
import os
import tensorflow as tf
import tensorflow_datasets as tfds
import matplotlib.pyplot as plt

print("TensorFlow version: {}".format(tf.__version__))
print("TensorFlow Datasets version: ",tfds.__version__)
```

Import the dataset

The default [penguins/processed](https://www.tensorflow.org/datasets/catalog/penguins) (<https://www.tensorflow.org/datasets/catalog/penguins>) TensorFlow Dataset is already cleaned, normalized, and ready for building a model. Before you download the processed data, preview a simplified version to get familiar with the original penguin survey data.

Preview the data

Download the simplified version of the penguins dataset (`penguins/simple`) using the TensorFlow Datasets `tfds.load` (https://www.tensorflow.org/datasets/api_docs/python/tfds/load) method. There are 344 data records in this dataset. Extract the first five records into a `DataFrame` (https://www.tensorflow.org/datasets/api_docs/python/tfds/as_dataframe) object to inspect a sample of the values in this dataset:

In []:

```
ds_preview, info = tfds.load('penguins/simple', split='train', with_info=True)
df = tfds.as_dataframe(ds_preview.take(5), info)
print(df)
print(info.features)
```

The numbered rows are data records, one [example](https://developers.google.com/machine-learning/glossary/#example) (<https://developers.google.com/machine-learning/glossary/#example>) per line, where:

- The first six fields are [features](https://developers.google.com/machine-learning/glossary/#feature) (<https://developers.google.com/machine-learning/glossary/#feature>): these are the characteristics of an example. Here, the fields hold numbers representing penguin measurements.
- The last column is the [label](https://developers.google.com/machine-learning/glossary/#label) (<https://developers.google.com/machine-learning/glossary/#label>): this is the value you want to predict. For this dataset, it's an integer value of 0, 1, or 2 that corresponds to a penguin species name.

In the dataset, the label for the penguin species is represented as a number to make it easier to work with in the model you are building. These numbers correspond to the following penguin species:

- 0 : Adélie penguin
- 1 : Chinstrap penguin
- 2 : Gentoo penguin

Create a list containing the penguin species names in this order. You will use this list to interpret the output of the classification model:

In []:

```
class_names = ['Adélie', 'Chinstrap', 'Gentoo']
```

For more information about features and labels, refer to the [ML Terminology section of the Machine Learning Crash Course](https://developers.google.com/machine-learning/crash-course/ml-terminology) (<https://developers.google.com/machine-learning/crash-course/ml-terminology>).

Download the preprocessed dataset

Now, download the preprocessed penguins dataset (`penguins/processed`) with the `tfds.load` method, which returns a list of `tf.data.Dataset` objects. Note that the `penguins/processed` dataset doesn't come with its own test set, so use an 80:20 split to [slice the full dataset](https://www.tensorflow.org/datasets/splits) (<https://www.tensorflow.org/datasets/splits>) into the training and test sets. You will use the test dataset later to verify your model.

In []:

```
ds_split, info = tfds.load("penguins/processed", split=['train[:20%]', 'train[20%:]'], as_supervised=True, with_info=True)

ds_test = ds_split[0]
ds_train = ds_split[1]
assert isinstance(ds_test, tf.data.Dataset)

print(info.features)
df_test = tfds.as_dataframe(ds_test.take(5), info)
print("Test dataset sample: ")
print(df_test)

df_train = tfds.as_dataframe(ds_train.take(5), info)
print("Train dataset sample: ")
print(df_train)

ds_train_batch = ds_train.batch(32)
```

Notice that this version of the dataset has been processed by reducing the data down to four normalized features and a species label. In this format, the data can be quickly used to train a model without further processing.

In []:

```
features, labels = next(iter(ds_train_batch))

print(features)
print(labels)
```

You can visualize some clusters by plotting a few features from the batch:

In []:

```
plt.scatter(features[:,0],
            features[:,2],
            c=labels,
            cmap='viridis')

plt.xlabel("Body Mass")
plt.ylabel("Culmen Length")
plt.show()
```

Build a simple linear model

Why model?

A [model](https://developers.google.com/machine-learning/crash-course/glossary#model) (<https://developers.google.com/machine-learning/crash-course/glossary#model>) is a relationship between features and the label. For the penguin classification problem, the model defines the relationship between the body mass, flipper and culmen measurements and the predicted penguin species. Some simple models can be described with a few lines of algebra, but complex machine learning models have a large number of parameters that are difficult to summarize.

Could you determine the relationship between the four features and the penguin species *without* using machine learning? That is, could you use traditional programming techniques (for example, a lot of conditional statements) to create a model? Perhaps—if you analyzed the dataset long enough to determine the relationships between body mass and culmen measurements to a particular species. And this becomes difficult—maybe impossible—on more complicated datasets. A good machine learning approach *determines the model for you*. If you feed enough representative examples into the right machine learning model type, the program figures out the relationships for you.

Select the model

Next you need to select the kind of model to train. There are many types of models and picking a good one takes experience. This tutorial uses a neural network to solve the penguin classification problem. [Neural networks](https://developers.google.com/machine-learning/glossary#neural_network) (https://developers.google.com/machine-learning/glossary#neural_network) can find complex relationships between features and the label. It is a highly-structured graph, organized into one or more [hidden layers](https://developers.google.com/machine-learning/glossary#hidden_layer) (https://developers.google.com/machine-learning/glossary#hidden_layer). Each hidden layer consists of one or more [neurons](https://developers.google.com/machine-learning/glossary#neuron) (<https://developers.google.com/machine-learning/glossary#neuron>). There are several categories of neural networks and this program uses a dense, or [fully-connected neural network](https://developers.google.com/machine-learning/glossary#fully_connected_neural_network) (https://developers.google.com/machine-learning/glossary#fully_connected_layer): the neurons in one layer receive input connections from every neuron in the previous layer. For example, Figure 2 illustrates a dense neural network consisting of an input layer, two hidden layers, and an output layer:

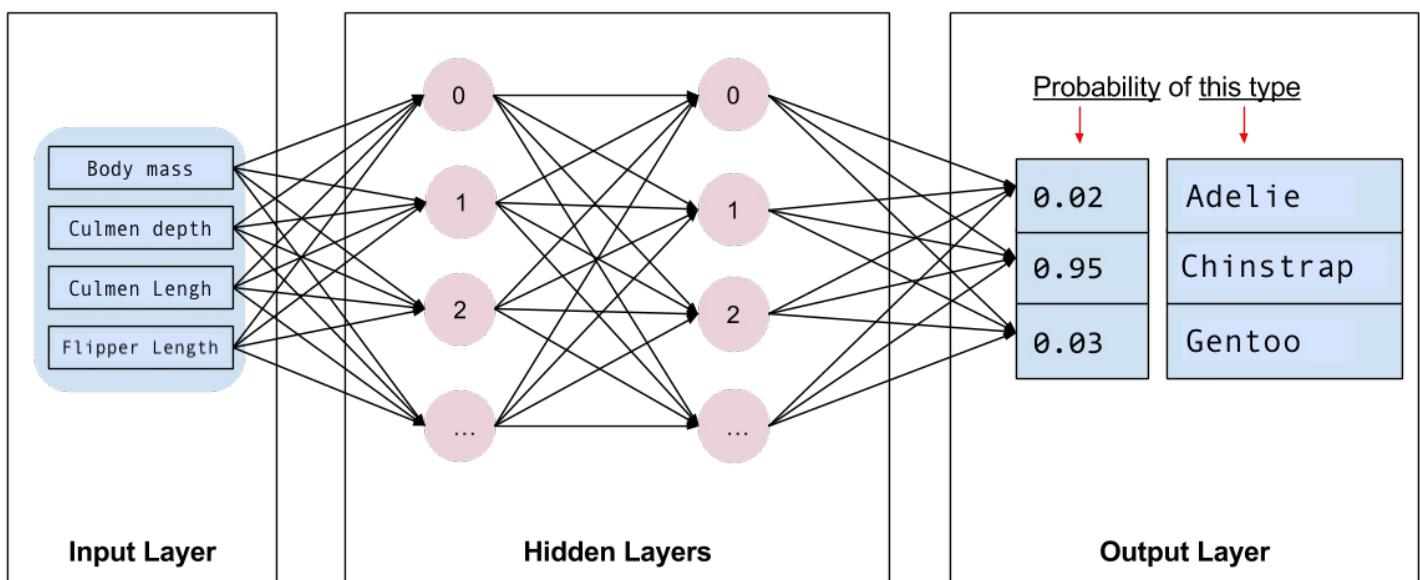


Figure 2. A neural network with features, hidden layers, and predictions.

When you train the model from Figure 2 and feed it an unlabeled example, it yields three predictions: the likelihood that this penguin is the given penguin species. This prediction is called [inference](https://developers.google.com/machine-learning/crash-course/glossary#inference) (<https://developers.google.com/machine-learning/crash-course/glossary#inference>). For this example, the sum of the output predictions is 1.0. In Figure 2, this prediction breaks down as: 0.02 for Adelie, 0.95 for Chinstrap, and 0.03 for Gentoo species. This means that the model predicts—with 95% probability—that an unlabeled example penguin is a Chinstrap penguin.

Create a model using Keras

The TensorFlow `tf.keras` API is the preferred way to create models and layers. This makes it easy to build models and experiment while Keras handles the complexity of connecting everything together.

The `tf.keras.Sequential` model is a linear stack of layers. Its constructor takes a list of layer instances, in this case, two `tf.keras.layers.Dense` layers with 10 nodes each, and an output layer with 3 nodes representing your label predictions. The first layer's `input_shape` parameter corresponds to the number of features from the dataset, and is required:

In []:

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(10, activation=tf.nn.relu, input_shape=(4,)), # input shape required
    tf.keras.layers.Dense(10, activation=tf.nn.relu),
    tf.keras.layers.Dense(3)
])
```

The [activation function](https://developers.google.com/machine-learning/crash-course/glossary#activation_function) (https://developers.google.com/machine-learning/crash-course/glossary#activation_function) determines the output shape of each node in the layer. These non-linearities are important—without them the model would be equivalent to a single layer. There are many `tf.keras.activations`, but [ReLU](https://developers.google.com/machine-learning/crash-course/glossary#ReLU) (<https://developers.google.com/machine-learning/crash-course/glossary#ReLU>) is common for hidden layers.

The ideal number of hidden layers and neurons depends on the problem and the dataset. Like many aspects of machine learning, picking the best shape of the neural network requires a mixture of knowledge and experimentation. As a rule of thumb, increasing the number of hidden layers and neurons typically creates a more powerful model, which requires more data to train effectively.

Use the model

Let's have a quick look at what this model does to a batch of features:

In []:

```
predictions = model(features)  
predictions[:5]
```

Here, each example returns a [logit](https://developers.google.com/machine-learning/crash-course/glossary#logits) (<https://developers.google.com/machine-learning/crash-course/glossary#logits>) for each class.

To convert these logits to a probability for each class, use the [softmax](https://developers.google.com/machine-learning/crash-course/glossary#softmax) (<https://developers.google.com/machine-learning/crash-course/glossary#softmax>) function:

In []:

```
tf.nn.softmax(predictions[:5])
```

Taking the `tf.math.argmax` across classes gives us the predicted class index. But, the model hasn't been trained yet, so these aren't good predictions:

In []:

```
print("Prediction: {}".format(tf.argmax(predictions, axis=1)))  
print("Labels: {}".format(labels))
```

Train the model

[Training](https://developers.google.com/machine-learning/crash-course/glossary#training) (<https://developers.google.com/machine-learning/crash-course/glossary#training>) is the stage of machine learning when the model is gradually optimized, or the model *learns* the dataset. The goal is to learn enough about the structure of the training dataset to make predictions about unseen data. If you learn *too much* about the training dataset, then the predictions only work for the data it has seen and will not be generalizable. This problem is called [overfitting](https://developers.google.com/machine-learning/crash-course/glossary#overfitting) (<https://developers.google.com/machine-learning/crash-course/glossary#overfitting>)—it's like memorizing the answers instead of understanding how to solve a problem.

The penguin classification problem is an example of [supervised machine learning](https://developers.google.com/machine-learning/glossary/#supervised_machine_learning) (https://developers.google.com/machine-learning/glossary/#supervised_machine_learning): the model is trained from examples that contain labels. In [unsupervised machine learning](https://developers.google.com/machine-learning/glossary/#unsupervised_machine_learning) (https://developers.google.com/machine-learning/glossary/#unsupervised_machine_learning), the examples don't contain labels. Instead, the model typically finds patterns among the features.

Define the loss and gradients function

Both training and evaluation stages need to calculate the model's [loss](https://developers.google.com/machine-learning/crash-course/glossary#loss) (<https://developers.google.com/machine-learning/crash-course/glossary#loss>). This measures how off a model's predictions are from the desired label, in other words, how bad the model is performing. You want to minimize, or optimize, this value.

Your model will calculate its loss using the `tf.keras.losses.SparseCategoricalCrossentropy` function which takes the model's class probability predictions and the desired label, and returns the average loss across the examples.

In []:

```
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
```

In []:

```
def loss(model, x, y, training):  
    # training=training is needed only if there are layers with different  
    # behavior during training versus inference (e.g. Dropout).  
    y_ = model(x, training=training)  
  
    return loss_object(y_true=y, y_pred=y_)  
  
l = loss(model, features, labels, training=False)  
print("Loss test: {}".format(l))
```

Use the `tf.GradientTape` context to calculate the [gradients](https://developers.google.com/machine-learning/crash-course/glossary#gradient) (<https://developers.google.com/machine-learning/crash-course/glossary#gradient>) used to optimize your model:

In []:

```
def grad(model, inputs, targets):
    with tf.GradientTape() as tape:
        loss_value = loss(model, inputs, targets, training=True)
    return loss_value, tape.gradient(loss_value, model.trainable_variables)
```

Create an optimizer

An [optimizer](https://developers.google.com/machine-learning/crash-course/glossary#optimizer) (<https://developers.google.com/machine-learning/crash-course/glossary#optimizer>) applies the computed gradients to the model's parameters to minimize the `loss` function. You can think of the loss function as a curved surface (refer to Figure 3) and you want to find its lowest point by walking around. The gradients point in the direction of steepest ascent—so you'll travel the opposite way and move down the hill. By iteratively calculating the loss and gradient for each batch, you'll adjust the model during training. Gradually, the model will find the best combination of weights and bias to minimize the loss. And the lower the loss, the better the model's predictions.

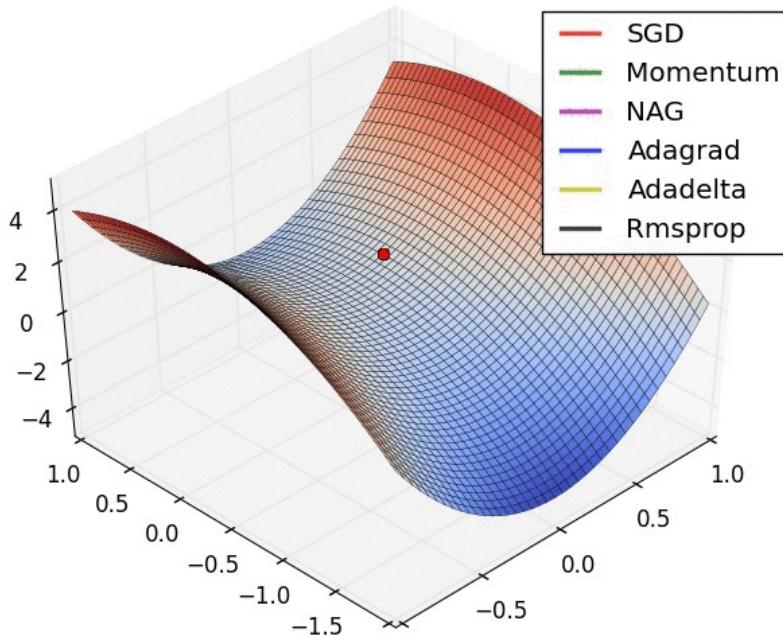


Figure 3. Optimization algorithms visualized over time in 3D space.

(Source: [Stanford class CS231n](http://cs231n.github.io/neural-networks-3/) (<http://cs231n.github.io/neural-networks-3/>), MIT License, Image credit: [@Alec Radford](https://twitter.com/alecrad) (<https://twitter.com/alecrad>))

TensorFlow has many optimization algorithms available for training. In this tutorial, you will use the `tf.keras.optimizers.SGD` that implements the [stochastic gradient descent](https://developers.google.com/machine-learning/crash-course/glossary#gradient_descent) (https://developers.google.com/machine-learning/crash-course/glossary#gradient_descent) (SGD) algorithm. The `learning_rate` parameter sets the step size to take for each iteration down the hill. This rate is a [hyperparameter](https://developers.google.com/machine-learning/glossary/#hyperparameter) (<https://developers.google.com/machine-learning/glossary/#hyperparameter>) that you'll commonly adjust to achieve better results.

Instantiate the optimizer with a [learning rate](https://developers.google.com/machine-learning/glossary#learning-rate) (<https://developers.google.com/machine-learning/glossary#learning-rate>) of `0.01`, a scalar value that is multiplied by the gradient at each iteration of the training:

In []:

```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
```

Then use this object to calculate a single optimization step:

In []:

```
loss_value, grads = grad(model, features, labels)

print("Step: {}, Initial Loss: {}".format(optimizer.iterations.numpy(),
                                             loss_value.numpy()))

optimizer.apply_gradients(zip(grads, model.trainable_variables))

print("Step: {},           Loss: {}".format(optimizer.iterations.numpy(),
                                             loss(model, features, labels, training=True).numpy()))
```

Training loop

With all the pieces in place, the model is ready for training! A training loop feeds the dataset examples into the model to help it make better predictions. The following code block sets up these training steps:

1. Iterate each *epoch*. An epoch is one pass through the dataset.
2. Within an epoch, iterate over each example in the training `Dataset` grabbing its *features* (`x`) and *label* (`y`).
3. Using the example's features, make a prediction and compare it with the label. Measure the inaccuracy of the prediction and use that to calculate the model's loss and gradients.
4. Use an `optimizer` to update the model's parameters.
5. Keep track of some stats for visualization.
6. Repeat for each epoch.

The `num_epochs` variable is the number of times to loop over the dataset collection. In the code below, `num_epochs` is set to 201 which means this training loop will run 201 times. Counter-intuitively, training a model longer does not guarantee a better model. `num_epochs` is a [hyperparameter](#) (<https://developers.google.com/machine-learning/glossary/#hyperparameter>) that you can tune. Choosing the right number usually requires both experience and experimentation:

In []:

```
## Note: Rerunning this cell uses the same model parameters

# Keep results for plotting
train_loss_results = []
train_accuracy_results = []

num_epochs = 201

for epoch in range(num_epochs):
    epoch_loss_avg = tf.keras.metrics.Mean()
    epoch_accuracy = tf.keras.metrics.SparseCategoricalAccuracy()

    # Training loop - using batches of 32
    for x, y in ds_train_batch:
        # Optimize the model
        loss_value, grads = grad(model, x, y)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))

        # Track progress
        epoch_loss_avg.update_state(loss_value) # Add current batch loss
        # Compare predicted label to actual label
        # training=True is needed only if there are layers with different
        # behavior during training versus inference (e.g. Dropout).
        epoch_accuracy.update_state(y, model(x, training=True))

    # End epoch
    train_loss_results.append(epoch_loss_avg.result())
    train_accuracy_results.append(epoch_accuracy.result())

    if epoch % 50 == 0:
        print("Epoch {}: Loss: {:.3f}, Accuracy: {:.3%}".format(epoch,
                                                               epoch_loss_avg.result(),
                                                               epoch_accuracy.result()))
```

Alternatively, you could use the built-in Keras `Model.fit(ds_train_batch)` (https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit) method to train your model.

Visualize the loss function over time

While it's helpful to print out the model's training progress, you can visualize the progress with [TensorBoard](#) (<https://www.tensorflow.org/tensorboard>) - a visualization and metrics tool that is packaged with TensorFlow. For this simple example, you will create basic charts using the `matplotlib` module.

Interpreting these charts takes some experience, but in general you want to see the *loss* decrease and the *accuracy* increase:

In []:

```
fig, axes = plt.subplots(2, sharex=True, figsize=(12, 8))
fig.suptitle('Training Metrics')

axes[0].set_ylabel("Loss", fontsize=14)
axes[0].plot(train_loss_results)

axes[1].set_ylabel("Accuracy", fontsize=14)
axes[1].set_xlabel("Epoch", fontsize=14)
axes[1].plot(train_accuracy_results)
plt.show()
```

Evaluate the model's effectiveness

Now that the model is trained, you can get some statistics on its performance.

Evaluating means determining how effectively the model makes predictions. To determine the model's effectiveness at penguin classification, pass some measurements to the model and ask the model to predict what penguin species they represent. Then compare the model's predictions against the actual label. For example, a model that picked the correct species on half the input examples has an [accuracy](https://developers.google.com/machine-learning/glossary/#accuracy) (<https://developers.google.com/machine-learning/glossary/#accuracy>) of 0.5. Figure 4 shows a slightly more effective model, getting 4 out of 5 predictions correct at 80% accuracy:

Example features	Label	Model prediction
5.9 3.0 4.3 1.5	1	1
6.9 3.1 5.4 2.1	2	2
5.1 3.3 1.7 0.5	0	0
6.0 3.4 4.5 1.6	1	2
5.5 2.5 4.0 1.3	1	1

Figure 4. A penguin classifier that is 80% accurate.

Set up the test set

Evaluating the model is similar to training the model. The biggest difference is the examples come from a separate [test set](https://developers.google.com/machine-learning/crash-course/glossary#test_set) (https://developers.google.com/machine-learning/crash-course/glossary#test_set) rather than the training set. To fairly assess a model's effectiveness, the examples used to evaluate a model must be different from the examples used to train the model.

The penguin dataset doesn't have a separate test dataset so in the previous Download the dataset section, you split the original dataset into test and train datasets. Use the `ds_test_batch` dataset for the evaluation.

Evaluate the model on the test dataset

Unlike the training stage, the model only evaluates a single [epoch](https://developers.google.com/machine-learning/glossary/#epoch) (<https://developers.google.com/machine-learning/glossary/#epoch>) of the test data. The following code iterates over each example in the test set and compare the model's prediction against the actual label. This comparison is used to measure the model's accuracy across the entire test set:

In []:

```
test_accuracy = tf.keras.metrics.Accuracy()
ds_test_batch = ds_test.batch(10)

for (x, y) in ds_test_batch:
    # training=False is needed only if there are layers with different
    # behavior during training versus inference (e.g. Dropout).
    logits = model(x, training=False)
    prediction = tf.argmax(logits, axis=1, output_type=tf.int64)
    test_accuracy(prediction, y)

print("Test set accuracy: {:.3%}".format(test_accuracy.result()))
```

You can also use the `model.evaluate(ds_test, return_dict=True)` keras function to get accuracy information on your test dataset.

By inspecting the last batch, for example, you can observe that the model predictions are usually correct.

In []:

```
tf.stack([y,prediction],axis=1)
```

Use the trained model to make predictions

You've trained a model and "proven" that it's good—but not perfect—at classifying penguin species. Now let's use the trained model to make some predictions on [unlabeled examples](https://developers.google.com/machine-learning/glossary/#unlabeled_example) (https://developers.google.com/machine-learning/glossary/#unlabeled_example); that is, on examples that contain features but not labels.

In real-life, the unlabeled examples could come from lots of different sources including apps, CSV files, and data feeds. For this tutorial, manually provide three unlabeled examples to predict their labels. Recall, the label numbers are mapped to a named representation as:

- 0 : Adélie penguin
- 1 : Chinstrap penguin
- 2 : Gentoo penguin

In []:

```
predict_dataset = tf.convert_to_tensor([
    [0.3, 0.8, 0.4, 0.5],
    [0.4, 0.1, 0.8, 0.5],
    [0.7, 0.9, 0.8, 0.4]
])

# training=False is needed only if there are layers with different
# behavior during training versus inference (e.g. Dropout).
predictions = model(predict_dataset, training=False)

for i, logits in enumerate(predictions):
    class_idx = tf.argmax(logits).numpy()
    p = tf.nn.softmax(logits)[class_idx]
    name = class_names[class_idx]
    print("Example {} prediction: {} {:.1f}%".format(i, name, 100*p))
```

Copyright 2018 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Custom layers



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/customization/custom_layers)

(https://www.tensorflow.org/tutorials/customization/custom_layers)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/customization/custom_lay)

We recommend using `tf.keras` as a high-level API for building neural networks. That said, most TensorFlow APIs are usable with eager execution.

In []:

```
import tensorflow as tf
```

In []:

```
print(tf.config.list_physical_devices('GPU'))
```

Layers: common sets of useful operations

Most of the time when writing code for machine learning models you want to operate at a higher level of abstraction than individual operations and manipulation of individual variables.

Many machine learning models are expressible as the composition and stacking of relatively simple layers, and TensorFlow provides both a set of many common layers as well as easy ways for you to write your own application-specific layers either from scratch or as the composition of existing layers.

TensorFlow includes the full [Keras \(https://keras.io\)](https://keras.io) API in the tf.keras package, and the Keras layers are very useful when building your own models.

In []:

```
# In the tf.keras.layers package, layers are objects. To construct a layer,
# simply construct the object. Most layers take as a first argument the number
# of output dimensions / channels.
layer = tf.keras.layers.Dense(100)
# The number of input dimensions is often unnecessary, as it can be inferred
# the first time the layer is used, but it can be provided if you want to
# specify it manually, which is useful in some complex models.
layer = tf.keras.layers.Dense(10, input_shape=(None, 5))
```

The full list of pre-existing layers can be seen in [the documentation \(https://www.tensorflow.org/api_docs/python/tf/keras/layers\)](https://www.tensorflow.org/api_docs/python/tf/keras/layers). It includes Dense (a fully-connected layer), Conv2D, LSTM, BatchNormalization, Dropout, and many others.

In []:

```
# To use a layer, simply call it.
layer(tf.zeros([10, 5]))
```

In []:

```
# Layers have many useful methods. For example, you can inspect all variables
# in a layer using `layer.variables` and trainable variables using
# `layer.trainable_variables`. In this case a fully-connected layer
# will have variables for weights and biases.
layer.variables
```

In []:

```
# The variables are also accessible through nice accessors
layer.kernel, layer.bias
```

Implementing custom layers

The best way to implement your own layer is extending the tf.keras.Layer class and implementing:

1. `__init__`, where you can do all input-independent initialization
2. `build`, where you know the shapes of the input tensors and can do the rest of the initialization
3. `call`, where you do the forward computation

Note that you don't have to wait until `build` is called to create your variables, you can also create them in `__init__`. However, the advantage of creating them in `build` is that it enables late variable creation based on the shape of the inputs the layer will operate on. On the other hand, creating variables in `__init__` would mean that shapes required to create the variables will need to be explicitly specified.

In []:

```
class MyDenseLayer(tf.keras.layers.Layer):
    def __init__(self, num_outputs):
        super(MyDenseLayer, self).__init__()
        self.num_outputs = num_outputs

    def build(self, input_shape):
        self.kernel = self.add_weight("kernel",
                                     shape=[int(input_shape[-1]),
                                             self.num_outputs])

    def call(self, inputs):
        return tf.matmul(inputs, self.kernel)

layer = MyDenseLayer(10)
```

In []:

```
_ = layer(tf.zeros([10, 5])) # Calling the layer `builds` it.
```

```
In [ ]:
```

```
print([var.name for var in layer.trainable_variables])
```

Overall code is easier to read and maintain if it uses standard layers whenever possible, as other readers will be familiar with the behavior of standard layers. If you want to use a layer which is not present in `tf.keras.layers`, consider filing a [github issue](#) (<http://github.com/tensorflow/tensorflow/issues/new>) or, even better, sending us a pull request!

Models: Composing layers

Many interesting layer-like things in machine learning models are implemented by composing existing layers. For example, each residual block in a resnet is a composition of convolutions, batch normalizations, and a shortcut. Layers can be nested inside other layers.

Typically you inherit from `keras.Model` when you need the model methods like: `Model.fit`, `Model.evaluate`, and `Model.save` (see [Custom Keras layers and models](#) (https://www.tensorflow.org/guide/keras/custom_layers_and_models) for details).

One other feature provided by `keras.Model` (instead of `keras.layers.Layer`) is that in addition to tracking variables, a `keras.Model` also tracks its internal layers, making them easier to inspect.

For example here is a ResNet block:

```
In [ ]:
```

```
class ResnetIdentityBlock(tf.keras.Model):
    def __init__(self, kernel_size, filters):
        super(ResnetIdentityBlock, self).__init__(name='')
        filters1, filters2, filters3 = filters

        self.conv2a = tf.keras.layers.Conv2D(filters1, (1, 1))
        self.bn2a = tf.keras.layers.BatchNormalization()

        self.conv2b = tf.keras.layers.Conv2D(filters2, kernel_size, padding='same')
        self.bn2b = tf.keras.layers.BatchNormalization()

        self.conv2c = tf.keras.layers.Conv2D(filters3, (1, 1))
        self.bn2c = tf.keras.layers.BatchNormalization()

    def call(self, input_tensor, training=False):
        x = self.conv2a(input_tensor)
        x = self.bn2a(x, training=training)
        x = tf.nn.relu(x)

        x = self.conv2b(x)
        x = self.bn2b(x, training=training)
        x = tf.nn.relu(x)

        x = self.conv2c(x)
        x = self.bn2c(x, training=training)

        x += input_tensor
        return tf.nn.relu(x)

block = ResnetIdentityBlock(1, [1, 2, 3])
```

```
In [ ]:
```

```
_ = block(tf.zeros([1, 2, 3, 3]))
```

```
In [ ]:
```

```
block.layers
```

```
In [ ]:
```

```
len(block.variables)
```

```
In [ ]:
```

```
block.summary()
```

Much of the time, however, models which compose many layers simply call one layer after the other. This can be done in very little code using `tf.keras.Sequential`:

In []:

```
my_seq = tf.keras.Sequential([
    tf.keras.layers.Conv2D(1, (1, 1),
        input_shape=(None, None, 3)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(2, 1,
        padding='same'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(3, (1, 1)),
    tf.keras.layers.BatchNormalization()])
my_seq(tf.zeros([1, 2, 3, 3]))
```

In []:

```
my_seq.summary()
```

Next steps

Now you can go back to the previous notebook and adapt the linear regression example to use layers and models to be better structured.

Copyright 2019 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

In []:

```
#@title MIT License
#
# Copyright (c) 2017 François Chollet
# IGNORE_COPYRIGHT: cleared by OSS licensing
#
# Permission is hereby granted, free of charge, to any person obtaining a
# copy of this software and associated documentation files (the "Software"),
# to deal in the Software without restriction, including without limitation
# the rights to use, copy, modify, merge, publish, distribute, sublicense,
# and/or sell copies of the Software, and to permit persons to whom the
# Software is furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
# THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
# FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
# DEALINGS IN THE SOFTWARE.
```

Transfer learning and fine-tuning



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/images/transfer_learning)

(https://www.tensorflow.org/tutorials/images/transfer_learning)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/images/transfer_learning.ipynb)

(https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/images/transfer_learning.ipynb)

In this tutorial, you will learn how to classify images of cats and dogs by using transfer learning from a pre-trained network.

A pre-trained model is a saved network that was previously trained on a large dataset, typically on a large-scale image-classification task. You either use the pretrained model as is or use transfer learning to customize this model to a given task.

The intuition behind transfer learning for image classification is that if a model is trained on a large and general enough dataset, this model will effectively serve as a generic model of the visual world. You can then take advantage of these learned feature maps without having to start from scratch by training a large model on a large dataset.

In this notebook, you will try two ways to customize a pretrained model:

1. Feature Extraction: Use the representations learned by a previous network to extract meaningful features from new samples. You simply add a new classifier, which will be trained from scratch, on top of the pretrained model so that you can repurpose the feature maps learned previously for the dataset.

You do not need to (re)train the entire model. The base convolutional network already contains features that are generically useful for classifying pictures. However, the final, classification part of the pretrained model is specific to the original classification task, and subsequently specific to the set of classes on which the model was trained.

2. Fine-Tuning: Unfreeze a few of the top layers of a frozen model base and jointly train both the newly-added classifier layers and the last layers of the base model. This allows us to "fine-tune" the higher-order feature representations in the base model in order to make them more relevant for the specific task.

You will follow the general machine learning workflow.

1. Examine and understand the data
2. Build an input pipeline, in this case using Keras ImageDataGenerator
3. Compose the model
 - Load in the pretrained base model (and pretrained weights)
 - Stack the classification layers on top
4. Train the model
5. Evaluate model

In []:

```
import matplotlib.pyplot as plt
import numpy as np
import os
import tensorflow as tf
```

Data preprocessing

Data download

In this tutorial, you will use a dataset containing several thousand images of cats and dogs. Download and extract a zip file containing the images, then create a `tf.data.Dataset` for training and validation using the `tf.keras.utils.image_dataset_from_directory` utility. You can learn more about loading images in this [tutorial](https://www.tensorflow.org/tutorials/load_data/images) (https://www.tensorflow.org/tutorials/load_data/images).

In []:

```
_URL = 'https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip'
path_to_zip = tf.keras.utils.get_file('cats_and_dogs.zip', origin=_URL, extract=True)
PATH = os.path.join(os.path.dirname(path_to_zip), 'cats_and_dogs_filtered')

train_dir = os.path.join(PATH, 'train')
validation_dir = os.path.join(PATH, 'validation')

BATCH_SIZE = 32
IMG_SIZE = (160, 160)

train_dataset = tf.keras.utils.image_dataset_from_directory(train_dir,
                                                          shuffle=True,
                                                          batch_size=BATCH_SIZE,
                                                          image_size=IMG_SIZE)
```

In []:

```
validation_dataset = tf.keras.utils.image_dataset_from_directory(validation_dir,
                                                               shuffle=True,
                                                               batch_size=BATCH_SIZE,
                                                               image_size=IMG_SIZE)
```

Show the first nine images and labels from the training set:

In []:

```
class_names = train_dataset.class_names

plt.figure(figsize=(10, 10))
for images, labels in train_dataset.take(1):
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(images[i].numpy().astype("uint8"))
        plt.title(class_names[labels[i]])
        plt.axis("off")
```

As the original dataset doesn't contain a test set, you will create one. To do so, determine how many batches of data are available in the validation set using `tf.data.experimental.cardinality`, then move 20% of them to a test set.

In []:

```
val_batches = tf.data.experimental.cardinality(validation_dataset)
test_dataset = validation_dataset.take(val_batches // 5)
validation_dataset = validation_dataset.skip(val_batches // 5)
```

In []:

```
print('Number of validation batches: %d' % tf.data.experimental.cardinality(validation_dataset))
print('Number of test batches: %d' % tf.data.experimental.cardinality(test_dataset))
```

Configure the dataset for performance

Use buffered prefetching to load images from disk without having I/O become blocking. To learn more about this method see the [data performance](#) (https://www.tensorflow.org/guide/data_performance) guide.

In []:

```
AUTOTUNE = tf.data.AUTOTUNE

train_dataset = train_dataset.prefetch(buffer_size=AUTOTUNE)
validation_dataset = validation_dataset.prefetch(buffer_size=AUTOTUNE)
test_dataset = test_dataset.prefetch(buffer_size=AUTOTUNE)
```

Use data augmentation

When you don't have a large image dataset, it's a good practice to artificially introduce sample diversity by applying random, yet realistic, transformations to the training images, such as rotation and horizontal flipping. This helps expose the model to different aspects of the training data and reduce [overfitting](#) (https://www.tensorflow.org/tutorials/keras/overfit_and_underfit). You can learn more about data augmentation in this [tutorial](#) (https://www.tensorflow.org/tutorials/images/data_augmentation).

In []:

```
data_augmentation = tf.keras.Sequential([
    tf.keras.layers.RandomFlip('horizontal'),
    tf.keras.layers.RandomRotation(0.2),
])
```

Note: These layers are active only during training, when you call `Model.fit`. They are inactive when the model is used in inference mode in `Model.evaluate` or `Model.fit`.

Let's repeatedly apply these layers to the same image and see the result.

In []:

```
for image, _ in train_dataset.take(1):
    plt.figure(figsize=(10, 10))
    first_image = image[0]
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        augmented_image = data_augmentation(tf.expand_dims(first_image, 0))
        plt.imshow(augmented_image[0] / 255)
        plt.axis('off')
```

Rescale pixel values

In a moment, you will download `tf.keras.applications.MobileNetV2` for use as your base model. This model expects pixel values in `[-1, 1]`, but at this point, the pixel values in your images are in `[0, 255]`. To rescale them, use the preprocessing method included with the model.

In []:

```
preprocess_input = tf.keras.applications.mobilenet_v2.preprocess_input
```

Note: Alternatively, you could rescale pixel values from `[0, 255]` to `[-1, 1]` using `tf.keras.layers.Rescaling`.

In []:

```
rescale = tf.keras.layers.Rescaling(1./127.5, offset=-1)
```

Note: If using other `tf.keras.applications`, be sure to check the API doc to determine if they expect pixels in `[-1, 1]` or `[0, 1]`, or use the included `preprocess_input` function.

Create the base model from the pre-trained convnets

You will create the base model from the **MobileNet V2** model developed at Google. This is pre-trained on the ImageNet dataset, a large dataset consisting of 1.4M images and 1000 classes. ImageNet is a research training dataset with a wide variety of categories like `jackfruit` and `syringe`. This base of knowledge will help us classify cats and dogs from our specific dataset.

First, you need to pick which layer of MobileNet V2 you will use for feature extraction. The very last classification layer (on "top", as most diagrams of machine learning models go from bottom to top) is not very useful. Instead, you will follow the common practice to depend on the very last layer before the flatten operation. This layer is called the "bottleneck layer". The bottleneck layer features retain more generality as compared to the final/top layer.

First, instantiate a MobileNet V2 model pre-loaded with weights trained on ImageNet. By specifying the `include_top=False` argument, you load a network that doesn't include the classification layers at the top, which is ideal for feature extraction.

In []:

```
# Create the base model from the pre-trained model MobileNet V2
IMG_SHAPE = IMG_SIZE + (3,)
base_model = tf.keras.applications.MobileNetV2(input_shape=IMG_SHAPE,
                                                include_top=False,
                                                weights='imagenet')
```

This feature extractor converts each `160x160x3` image into a `5x5x1280` block of features. Let's see what it does to an example batch of images:

In []:

```
image_batch, label_batch = next(iter(train_dataset))
feature_batch = base_model(image_batch)
print(feature_batch.shape)
```

Feature extraction

In this step, you will freeze the convolutional base created from the previous step and to use as a feature extractor. Additionally, you add a classifier on top of it and train the top-level classifier.

Freeze the convolutional base

It is important to freeze the convolutional base before you compile and train the model. Freezing (by setting `layer.trainable = False`) prevents the weights in a given layer from being updated during training. MobileNet V2 has many layers, so setting the entire model's `trainable` flag to `False` will freeze all of them.

In []:

```
base_model.trainable = False
```

Important note about BatchNormalization layers

Many models contain `tf.keras.layers.BatchNormalization` layers. This layer is a special case and precautions should be taken in the context of fine-tuning, as shown later in this tutorial.

When you set `layer.trainable = False`, the `BatchNormalization` layer will run in inference mode, and will not update its mean and variance statistics.

When you unfreeze a model that contains `BatchNormalization` layers in order to do fine-tuning, you should keep the `BatchNormalization` layers in inference mode by passing `training = False` when calling the base model. Otherwise, the updates applied to the non-trainable weights will destroy what the model has learned.

For more details, see the [Transfer learning guide \(`https://www.tensorflow.org/guide/keras/transfer_learning`\)](https://www.tensorflow.org/guide/keras/transfer_learning).

In []:

```
# Let's take a look at the base model architecture  
base_model.summary()
```

Add a classification head

To generate predictions from the block of features, average over the spatial `5x5` spatial locations, using a `tf.keras.layers.GlobalAveragePooling2D` layer to convert the features to a single 1280-element vector per image.

In []:

```
global_average_layer = tf.keras.layers.GlobalAveragePooling2D()  
feature_batch_average = global_average_layer(feature_batch)  
print(feature_batch_average.shape)
```

Apply a `tf.keras.layers.Dense` layer to convert these features into a single prediction per image. You don't need an activation function here because this prediction will be treated as a `logit`, or a raw prediction value. Positive numbers predict class 1, negative numbers predict class 0.

In []:

```
prediction_layer = tf.keras.layers.Dense(1)  
prediction_batch = prediction_layer(feature_batch_average)  
print(prediction_batch.shape)
```

Build a model by chaining together the data augmentation, rescaling, `base_model` and feature extractor layers using the [Keras Functional API \(`https://www.tensorflow.org/guide/keras/functional`\)](https://www.tensorflow.org/guide/keras/functional). As previously mentioned, use `training=False` as our model contains a `BatchNormalization` layer.

In []:

```
inputs = tf.keras.Input(shape=(160, 160, 3))  
x = data_augmentation(inputs)  
x = preprocess_input(x)  
x = base_model(x, training=False)  
x = global_average_layer(x)  
x = tf.keras.layers.Dropout(0.2)(x)  
outputs = prediction_layer(x)  
model = tf.keras.Model(inputs, outputs)
```

Compile the model

Compile the model before training it. Since there are two classes, use the `tf.keras.losses.BinaryCrossentropy` loss with `from_logits=True` since the model provides a linear output.

In []:

```
base_learning_rate = 0.0001  
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=base_learning_rate),  
              loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),  
              metrics=['accuracy'])
```

In []:

```
model.summary()
```

The 2.5 million parameters in MobileNet are frozen, but there are 1.2 thousand *trainable* parameters in the Dense layer. These are divided between two `tf.Variable` objects, the weights and biases.

In []:

```
len(model.trainable_variables)
```

Train the model

After training for 10 epochs, you should see ~94% accuracy on the validation set.

In []:

```
initial_epochs = 10  
loss0, accuracy0 = model.evaluate(validation_dataset)
```

In []:

```
print("initial loss: {:.2f}".format(loss0))  
print("initial accuracy: {:.2f}".format(accuracy0))
```

In []:

```
history = model.fit(train_dataset,  
                     epochs=initial_epochs,  
                     validation_data=validation_dataset)
```

Learning curves

Let's take a look at the learning curves of the training and validation accuracy/loss when using the MobileNetV2 base model as a fixed feature extractor.

In []:

```
acc = history.history['accuracy']  
val_acc = history.history['val_accuracy']  
  
loss = history.history['loss']  
val_loss = history.history['val_loss']  
  
plt.figure(figsize=(8, 8))  
plt.subplot(2, 1, 1)  
plt.plot(acc, label='Training Accuracy')  
plt.plot(val_acc, label='Validation Accuracy')  
plt.legend(loc='lower right')  
plt.ylabel('Accuracy')  
plt.ylim([min(plt.ylim()),1])  
plt.title('Training and Validation Accuracy')  
  
plt.subplot(2, 1, 2)  
plt.plot(loss, label='Training Loss')  
plt.plot(val_loss, label='Validation Loss')  
plt.legend(loc='upper right')  
plt.ylabel('Cross Entropy')  
plt.ylim([0,1.0])  
plt.title('Training and Validation Loss')  
plt.xlabel('epoch')  
plt.show()
```

Note: If you are wondering why the validation metrics are clearly better than the training metrics, the main factor is because layers like `tf.keras.layers.BatchNormalization` and `tf.keras.layers.Dropout` affect accuracy during training. They are turned off when calculating validation loss.

To a lesser extent, it is also because training metrics report the average for an epoch, while validation metrics are evaluated after the epoch, so validation metrics see a model that has trained slightly longer.

Fine tuning

In the feature extraction experiment, you were only training a few layers on top of an MobileNetV2 base model. The weights of the pre-trained network were **not** updated during training.

One way to increase performance even further is to train (or "fine-tune") the weights of the top layers of the pre-trained model alongside the training of the classifier you added. The training process will force the weights to be tuned from generic feature maps to features associated specifically with the dataset.

Note: This should only be attempted after you have trained the top-level classifier with the pre-trained model set to non-trainable. If you add a randomly initialized classifier on top of a pre-trained model and attempt to train all layers jointly, the magnitude of the gradient updates will be too large (due to the random weights from the classifier) and your pre-trained model will forget what it has learned.

Also, you should try to fine-tune a small number of top layers rather than the whole MobileNet model. In most convolutional networks, the higher up a layer is, the more specialized it is. The first few layers learn very simple and generic features that generalize to almost all types of images. As you go higher up, the features are increasingly more specific to the dataset on which the model was trained. The goal of fine-tuning is to adapt these specialized features to work with the new dataset, rather than overwrite the generic learning.

Un-freeze the top layers of the model

All you need to do is unfreeze the `base_model` and set the bottom layers to be un-trainable. Then, you should recompile the model (necessary for these changes to take effect), and resume training.

In []:

```
base_model.trainable = True
```

In []:

```
# Let's take a look to see how many layers are in the base model
print("Number of layers in the base model: ", len(base_model.layers))

# Fine-tune from this layer onwards
fine_tune_at = 100

# Freeze all the layers before the `fine_tune_at` layer
for layer in base_model.layers[:fine_tune_at]:
    layer.trainable = False
```

Compile the model

As you are training a much larger model and want to readapt the pretrained weights, it is important to use a lower learning rate at this stage. Otherwise, your model could overfit very quickly.

In []:

```
model.compile(loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),  
              optimizer = tf.keras.optimizers.RMSprop(learning_rate=base_learning_rate/10),  
              metrics=[ 'accuracy' ])
```

In []:

```
model.summary()
```

In []:

```
len(model.trainable_variables)
```

Continue training the model

If you trained to convergence earlier, this step will improve your accuracy by a few percentage points.

In []:

Let's take a look at the learning curves of the training and validation accuracy/loss when fine-tuning the last few layers of the MobileNetV2 base model and training the classifier on top of it. The validation loss is much higher than the training loss, so you may get some overfitting.

You may also get some overfitting as the new training set is relatively small and similar to the original MobileNetV2 datasets.

After fine tuning the model nearly reaches 98% accuracy on the validation set.

In []:

```
acc += history_fine.history['accuracy']
val_acc += history_fine.history['val_accuracy']

loss += history_fine.history['loss']
val_loss += history_fine.history['val_loss']
```

In []:

```
plt.figure(figsize=(8, 8))
plt.subplot(2, 1, 1)
plt.plot(acc, label='Training Accuracy')
plt.plot(val_acc, label='Validation Accuracy')
plt.ylim([0.8, 1])
plt.plot([initial_epochs-1, initial_epochs-1],
         plt.ylim(), label='Start Fine Tuning')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(2, 1, 2)
plt.plot(loss, label='Training Loss')
plt.plot(val_loss, label='Validation Loss')
plt.ylim([0, 1.0])
plt.plot([initial_epochs-1, initial_epochs-1],
         plt.ylim(), label='Start Fine Tuning')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.xlabel('epoch')
plt.show()
```

Evaluation and prediction

Finally you can verify the performance of the model on new data using test set.

In []:

```
loss, accuracy = model.evaluate(test_dataset)
print('Test accuracy :', accuracy)
```

And now you are all set to use this model to predict if your pet is a cat or dog.

In []:

```
# Retrieve a batch of images from the test set
image_batch, label_batch = test_dataset.as_numpy_iterator().next()
predictions = model.predict_on_batch(image_batch).flatten()

# Apply a sigmoid since our model returns logits
predictions = tf.nn.sigmoid(predictions)
predictions = tf.where(predictions < 0.5, 0, 1)

print('Predictions:\n', predictions.numpy())
print('Labels:\n', label_batch)

plt.figure(figsize=(10, 10))
for i in range(9):
    ax = plt.subplot(3, 3, i + 1)
    plt.imshow(image_batch[i].astype("uint8"))
    plt.title(class_names[predictions[i]])
    plt.axis("off")
```

Summary

- **Using a pre-trained model for feature extraction:** When working with a small dataset, it is a common practice to take advantage of features learned by a model trained on a larger dataset in the same domain. This is done by instantiating the pre-trained model and adding a fully-connected classifier on top. The pre-trained model is "frozen" and only the weights of the classifier get updated during training. In this case, the convolutional base extracted all the features associated with each image and you just trained a classifier that determines the image class given that set of extracted features.
- **Fine-tuning a pre-trained model:** To further improve performance, one might want to repurpose the top-level layers of the pre-trained models to the new dataset via fine-tuning. In this case, you tuned your weights such that your model learned high-level features specific to the dataset. This technique is usually recommended when the training dataset is large and very similar to the original dataset that the pre-trained model was trained on.

To learn more, visit the [Transfer learning guide \(https://www.tensorflow.org/guide/keras/transfer_learning\)](https://www.tensorflow.org/guide/keras/transfer_learning).

Copyright 2019 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Convolutional Neural Network (CNN)



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/images/cnn)



[Run in Google Colab](#)

This tutorial demonstrates training a simple [Convolutional Neural Network \(https://developers.google.com/machine-learning/glossary/#convolutional_neural_network\)](https://developers.google.com/machine-learning/glossary/#convolutional_neural_network) (CNN) to classify [CIFAR images \(https://www.cs.toronto.edu/~kriz/cifar.html\)](https://www.cs.toronto.edu/~kriz/cifar.html). Because this tutorial uses the [Keras Sequential API \(https://www.tensorflow.org/guide/keras/overview\)](https://www.tensorflow.org/guide/keras/overview), creating and training your model will take just a few lines of code.

Import TensorFlow

In []:

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
```

Download and prepare the CIFAR10 dataset

The CIFAR10 dataset contains 60,000 color images in 10 classes, with 6,000 images in each class. The dataset is divided into 50,000 training images and 10,000 testing images. The classes are mutually exclusive and there is no overlap between them.

In []:

```
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()
# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0
```

Verify the data

To verify that the dataset looks correct, let's plot the first 25 images from the training set and display the class name below each image:

In []:

```
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']

plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i])
    # The CIFAR labels happen to be arrays,
    # which is why you need the extra index
    plt.xlabel(class_names[train_labels[i][0]])
plt.show()
```

Create the convolutional base

The 6 lines of code below define the convolutional base using a common pattern: a stack of [Conv2D](#) (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D) and [MaxPooling2D](#) (https://www.tensorflow.org/api_docs/python/tf/keras/layers/MaxPool2D) layers.

As input, a CNN takes tensors of shape (image_height, image_width, color_channels), ignoring the batch size. If you are new to these dimensions, color_channels refers to (R,G,B). In this example, you will configure your CNN to process inputs of shape (32, 32, 3), which is the format of CIFAR images. You can do this by passing the argument `input_shape` to your first layer.

In []:

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

Let's display the architecture of your model so far:

In []:

```
model.summary()
```

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 you 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, you can afford (computationally) to add more output channels in each Conv2D layer.

Add Dense layers on top

To complete the model, you will feed the last output tensor from the convolutional base (of shape (4, 4, 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, you will flatten (or unroll) the 3D output to 1D, then add one or more Dense layers on top. CIFAR has 10 output classes, so you use a final Dense layer with 10 outputs.

In []:

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10))
```

Here's the complete architecture of your model:

In []:

```
model.summary()
```

The network summary shows that (4, 4, 64) outputs were flattened into vectors of shape (1024) before going through two Dense layers.

Compile and train the model

In []:

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

history = model.fit(train_images, train_labels, epochs=10,
                     validation_data=(test_images, test_labels))
```

Evaluate the model

In []:

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')

test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
```

In []:

```
print(test_acc)
```

Your simple CNN has achieved a test accuracy of over 70%. Not bad for a few lines of code! For another CNN style, check out the [TensorFlow 2 quickstart for experts](#) (<https://www.tensorflow.org/tutorials/quickstart/advanced>) example that uses the Keras subclassing API and `tf.GradientTape`.

Copyright 2019 The TensorFlow Authors.

Licensed under the Apache License, Version 2.0 (the "License");

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Image segmentation



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/images/segmentation)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/images/segmentation.ipynb)

This tutorial focuses on the task of image segmentation, using a modified [U-Net](https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/) (<https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/>).

What is image segmentation?

In an image classification task the network assigns a label (or class) to each input image. However, suppose you want to know the shape of that object, which pixel belongs to which object, etc. In this case you will want to assign a class to each pixel of the image. This task is known as segmentation. A segmentation model returns much more detailed information about the image. Image segmentation has many applications in medical imaging, self-driving cars and satellite imaging to name a few.

This tutorial uses the [Oxford-IIIT Pet Dataset](https://www.roberts.ox.ac.uk/~vgg/data/pets/) (<https://www.roberts.ox.ac.uk/~vgg/publications/2012/parkhi12a/parkhi12a.pdf>). The dataset consists of images of 37 pet breeds, with 200 images per breed (~100 each in the training and test splits). Each image includes the corresponding labels, and pixel-wise masks. The masks are class-labels for each pixel. Each pixel is given one of three categories:

- Class 1: Pixel belonging to the pet.
- Class 2: Pixel bordering the pet.
- Class 3: None of the above/a surrounding pixel.

```
In [ ]:
```

```
!pip install git+https://github.com/tensorflow/examples.git
```

```
In [ ]:
```

```
import tensorflow as tf
```

```
import tensorflow_datasets as tfds
```

```
In [ ]:
```

```
from tensorflow_examples.models.pix2pix import pix2pix
```

```
from IPython.display import clear_output  
import matplotlib.pyplot as plt
```

Download the Oxford-IIIT Pets dataset

The dataset is [available from TensorFlow Datasets \(https://www.tensorflow.org/datasets/catalog/oxford_iiit_pet\)](https://www.tensorflow.org/datasets/catalog/oxford_iiit_pet). The segmentation masks are included in version 3+.

```
In [ ]:
```

```
dataset, info = tfds.load('oxford_iiit_pet:3.*.*', with_info=True)
```

In addition, the image color values are normalized to the `[0,1]` range. Finally, as mentioned above the pixels in the segmentation mask are labeled either {1, 2, 3}. For the sake of convenience, subtract 1 from the segmentation mask, resulting in labels that are : {0, 1, 2}.

```
In [ ]:
```

```
def normalize(input_image, input_mask):  
    input_image = tf.cast(input_image, tf.float32) / 255.0  
    input_mask -= 1  
    return input_image, input_mask
```

```
In [ ]:
```

```
def load_image(datapoint):  
    input_image = tf.image.resize(datapoint['image'], (128, 128))  
    input_mask = tf.image.resize(datapoint['segmentation_mask'], (128, 128))  
  
    input_image, input_mask = normalize(input_image, input_mask)  
  
    return input_image, input_mask
```

The dataset already contains the required training and test splits, so continue to use the same splits.

```
In [ ]:
```

```
TRAIN_LENGTH = info.splits['train'].num_examples  
BATCH_SIZE = 64  
BUFFER_SIZE = 1000  
STEPS_PER_EPOCH = TRAIN_LENGTH // BATCH_SIZE
```

```
In [ ]:
```

```
train_images = dataset['train'].map(load_image, num_parallel_calls=tf.data.AUTOTUNE)  
test_images = dataset['test'].map(load_image, num_parallel_calls=tf.data.AUTOTUNE)
```

The following class performs a simple augmentation by randomly-flipping an image. Go to the [Image augmentation \(data_augmentation.ipynb\)](#) tutorial to learn more.

```
In [ ]:
```

```
class Augment(tf.keras.layers.Layer):  
    def __init__(self, seed=42):  
        super().__init__()  
        # both use the same seed, so they'll make the same random changes.  
        self.augment_inputs = tf.keras.layers.RandomFlip(mode="horizontal", seed=seed)  
        self.augment_labels = tf.keras.layers.RandomFlip(mode="horizontal", seed=seed)  
  
    def call(self, inputs, labels):  
        inputs = self.augment_inputs(inputs)  
        labels = self.augment_labels(labels)  
        return inputs, labels
```

Build the input pipeline, applying the Augmentation after batching the inputs.

In []:

```
train_batches = (
    train_images
    .cache()
    .shuffle(BUFFER_SIZE)
    .batch(BATCH_SIZE)
    .repeat()
    .map(Augment())
    .prefetch(buffer_size=tf.data.AUTOTUNE))

test_batches = test_images.batch(BATCH_SIZE)
```

Visualize an image example and its corresponding mask from the dataset.

In []:

```
def display(display_list):
    plt.figure(figsize=(15, 15))

    title = ['Input Image', 'True Mask', 'Predicted Mask']

    for i in range(len(display_list)):
        plt.subplot(1, len(display_list), i+1)
        plt.title(title[i])
        plt.imshow(tf.keras.utils.array_to_img(display_list[i]))
        plt.axis('off')
    plt.show()
```

In []:

```
for images, masks in train_batches.take(2):
    sample_image, sample_mask = images[0], masks[0]
    display([sample_image, sample_mask])
```

Define the model

The model being used here is a modified [U-Net](https://arxiv.org/abs/1505.04597) (<https://arxiv.org/abs/1505.04597>). A U-Net consists of an encoder (downsampler) and decoder (upsampler). In-order to learn robust features and reduce the number of trainable parameters, you will use a pretrained model - MobileNetV2 - as the encoder. For the decoder, you will use the upsample block, which is already implemented in the [pix2pix](https://github.com/tensorflow/examples/blob/master/tensorflow_examples/models/pix2pix/pix2pix.py) (https://github.com/tensorflow/examples/blob/master/tensorflow_examples/models/pix2pix/pix2pix.py) example in the TensorFlow Examples repo. (Check out the [pix2pix: Image-to-image translation with a conditional GAN \(./generative/pix2pix.ipynb\)](#) tutorial in a notebook.)

As mentioned, the encoder will be a pretrained MobileNetV2 model which is prepared and ready to use in `tf.keras.applications`. The encoder consists of specific outputs from intermediate layers in the model. Note that the encoder will not be trained during the training process.

In []:

```
base_model = tf.keras.applications.MobileNetV2(input_shape=[128, 128, 3], include_top=False)

# Use the activations of these layers
layer_names = [
    'block_1_expand_relu',    # 64x64
    'block_3_expand_relu',    # 32x32
    'block_6_expand_relu',    # 16x16
    'block_13_expand_relu',   # 8x8
    'block_16_project',      # 4x4
]
base_model_outputs = [base_model.get_layer(name).output for name in layer_names]

# Create the feature extraction model
down_stack = tf.keras.Model(inputs=base_model.input, outputs=base_model_outputs)

down_stack.trainable = False
```

The decoder/upsampler is simply a series of upsample blocks implemented in TensorFlow examples.

In []:

```
up_stack = [
    pix2pix.upsample(512, 3), # 4x4 -> 8x8
    pix2pix.upsample(256, 3), # 8x8 -> 16x16
    pix2pix.upsample(128, 3), # 16x16 -> 32x32
    pix2pix.upsample(64, 3), # 32x32 -> 64x64
]
```

In []:

```
def unet_model(output_channels:int):
    inputs = tf.keras.layers.Input(shape=[128, 128, 3])

    # Downsampling through the model
    skips = down_stack(inputs)
    x = skips[-1]
    skips = reversed(skips[:-1])

    # Upsampling and establishing the skip connections
    for up, skip in zip(up_stack, skips):
        x = up(x)
        concat = tf.keras.layers.concatenate([x, skip])
        x = concat([x, skip])

    # This is the last layer of the model
    last = tf.keras.layers.Conv2DTranspose(
        filters=output_channels, kernel_size=3, strides=2,
        padding='same') #64x64 -> 128x128

    x = last(x)

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

Note that the number of filters on the last layer is set to the number of `output_channels`. This will be one output channel per class.

Train the model

Now, all that is left to do is to compile and train the model.

Since this is a multiclass classification problem, use the `tf.keras.losses.CategoricalCrossentropy` loss function with the `from_logits` argument set to `True`, since the labels are scalar integers instead of vectors of scores for each pixel of every class.

When running inference, the label assigned to the pixel is the channel with the highest value. This is what the `create_mask` function is doing.

In []:

```
OUTPUT_CLASSES = 3

model = unet_model(output_channels=OUTPUT_CLASSES)
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

Have a quick look at the resulting model architecture:

In []:

```
tf.keras.utils.plot_model(model, show_shapes=True)
```

Try out the model to check what it predicts before training.

In []:

```
def create_mask(pred_mask):
    pred_mask = tf.argmax(pred_mask, axis=-1)
    pred_mask = pred_mask[..., tf.newaxis]
    return pred_mask[0]
```

```
In [ ]:
```

```
def show_predictions(dataset=None, num=1):
    if dataset:
        for image, mask in dataset.take(num):
            pred_mask = model.predict(image)
            display([image[0], mask[0], create_mask(pred_mask)])
    else:
        display([sample_image, sample_mask,
                 create_mask(model.predict(sample_image[tf.newaxis, ...]))])
```

```
In [ ]:
```

```
show_predictions()
```

The callback defined below is used to observe how the model improves while it is training.

```
In [ ]:
```

```
class DisplayCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs=None):
        clear_output(wait=True)
        show_predictions()
        print ('\nSample Prediction after epoch {}:\n'.format(epoch+1))
```

```
In [ ]:
```

```
EPOCHS = 20
VAL_SUBSPLITS = 5
VALIDATION_STEPS = info.splits['test'].num_examples//BATCH_SIZE//VAL_SUBSPLITS

model_history = model.fit(train_batches, epochs=EPOCHS,
                           steps_per_epoch=STEPS_PER_EPOCH,
                           validation_steps=VALIDATION_STEPS,
                           validation_data=test_batches,
                           callbacks=[DisplayCallback()])
```

```
In [ ]:
```

```
loss = model_history.history['loss']
val_loss = model_history.history['val_loss']

plt.figure()
plt.plot(model_history.epoch, loss, 'r', label='Training loss')
plt.plot(model_history.epoch, val_loss, 'bo', label='Validation loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss Value')
plt.ylim([0, 1])
plt.legend()
plt.show()
```

Make predictions

Now, make some predictions. In the interest of saving time, the number of epochs was kept small, but you may set this higher to achieve more accurate results.

```
In [ ]:
```

```
show_predictions(test_batches, 3)
```

Optional: Imbalanced classes and class weights

Semantic segmentation datasets can be highly imbalanced meaning that particular class pixels can be present more inside images than that of other classes. Since segmentation problems can be treated as per-pixel classification problems, you can deal with the imbalance problem by weighing the loss function to account for this. It's a simple and elegant way to deal with this problem. Refer to the [Classification on imbalanced data](#) ([./structured_data/imbalanced_data.ipynb](#)) tutorial to learn more.

To [avoid ambiguity](https://github.com/keras-team/keras/issues/3653#issuecomment-243939748) (<https://github.com/keras-team/keras/issues/3653#issuecomment-243939748>), `Model.fit` does not support the `class_weight` argument for inputs with 3+ dimensions.

```
In [ ]:
```

```
try:  
    model_history = model.fit(train_batches, epochs=EPOCHS,  
                               steps_per_epoch=STEPS_PER_EPOCH,  
                               class_weight = {0:2.0, 1:2.0, 2:1.0})  
    assert False  
except Exception as e:  
    print(f"Expected {type(e).__name__}: {e}")
```

So, in this case you need to implement the weighting yourself. You'll do this using sample weights: In addition to `(data, label)` pairs, `Model.fit` also accepts `(data, label, sample_weight)` triples.

`Model.fit` propagates the `sample_weight` to the losses and metrics, which also accept a `sample_weight` argument. The sample weight is multiplied by the sample's value before the reduction step. For example:

```
In [ ]:
```

```
label = [0,0]  
prediction = [[-3., 0], [-3, 0]]  
sample_weight = [1, 10]  
  
loss = tf.losses.SparseCategoricalCrossentropy(from_logits=True,  
                                              reduction=tf.losses.Reduction.NONE)  
loss(label, prediction, sample_weight).numpy()
```

So to make sample weights for this tutorial you need a function that takes a `(data, label)` pair and returns a `(data, label, sample_weight)` triple. Where the `sample_weight` is a 1-channel image containing the class weight for each pixel.

The simplest possible implementation is to use the label as an index into a `class_weight` list:

```
In [ ]:
```

```
def add_sample_weights(image, label):  
    # The weights for each class, with the constraint that:  
    # sum(class_weights) == 1.0  
    class_weights = tf.constant([2.0, 2.0, 1.0])  
    class_weights = class_weights/tf.reduce_sum(class_weights)  
  
    # Create an image of `sample_weights` by using the label at each pixel as an  
    # index into the `class weights` .  
    sample_weights = tf.gather(class_weights, indices=tf.cast(label, tf.int32))  
  
    return image, label, sample_weights
```

The resulting dataset elements contain 3 images each:

```
In [ ]:
```

```
train_batches.map(add_sample_weights).element_spec
```

Now you can train a model on this weighted dataset:

```
In [ ]:
```

```
weighted_model = unet_model(OUTPUT_CLASSES)  
weighted_model.compile(  
    optimizer='adam',  
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),  
    metrics=['accuracy'])
```

```
In [ ]:
```

```
weighted_model.fit(  
    train_batches.map(add_sample_weights),  
    epochs=1,  
    steps_per_epoch=10)
```

Next steps

Now that you have an understanding of what image segmentation is and how it works, you can try this tutorial out with different intermediate layer outputs, or even different pretrained models. You may also challenge yourself by trying out the [Carvana \(<https://www.kaggle.com/c/carvana-image-masking-challenge/overview>\)](https://www.kaggle.com/c/carvana-image-masking-challenge/overview) image masking challenge hosted on Kaggle.

You may also want to see the [Tensorflow Object Detection API \(\[https://github.com/tensorflow/models/blob/master/research/object_detection/README.md\]\(https://github.com/tensorflow/models/blob/master/research/object_detection/README.md\)\)](https://github.com/tensorflow/models/blob/master/research/object_detection/README.md) for another model you can retrain on your own data. Pretrained models are available on [TensorFlow Hub \(\[https://www.tensorflow.org/hub/tutorials/tf2_object_detection#optional\]\(https://www.tensorflow.org/hub/tutorials/tf2_object_detection#optional\)\)](https://www.tensorflow.org/hub/tutorials/tf2_object_detection#optional)

Copyright 2018 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Image classification



[View on TensorFlow.org
\(<https://www.tensorflow.org/tutorials/images/classification>\)](https://www.tensorflow.org/tutorials/images/classification)



[Run in Google Colab
\(<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/images/classification.ipynb>\)](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/images/classification.ipynb)

This tutorial shows how to classify images of flowers. It creates an image classifier using a `tf.keras.Sequential` model, and loads data using `tf.keras.utils.image_dataset_from_directory`. You will gain practical experience with the following concepts:

- Efficiently loading a dataset off disk.
- Identifying overfitting and applying techniques to mitigate it, including data augmentation and dropout.

This tutorial follows a basic machine learning workflow:

1. Examine and understand data
2. Build an input pipeline
3. Build the model
4. Train the model
5. Test the model
6. Improve the model and repeat the process

Import TensorFlow and other libraries

In []:

```
import matplotlib.pyplot as plt
import numpy as np
import os
import PIL
import tensorflow as tf

from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
```

Download and explore the dataset

This tutorial uses a dataset of about 3,700 photos of flowers. The dataset contains five sub-directories, one per class:

```
flower_photo/
daisy/
dandelion/
roses/
sunflowers/
tulips/
```

In []:

```
import pathlib
dataset_url = "https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz"
data_dir = tf.keras.utils.get_file('flower_photos', origin=dataset_url, untar=True)
data_dir = pathlib.Path(data_dir)
```

After downloading, you should now have a copy of the dataset available. There are 3,670 total images:

In []:

```
image_count = len(list(data_dir.glob('*/*.jpg')))
print(image_count)
```

Here are some roses:

In []:

```
roses = list(data_dir.glob('roses/*'))
PIL.Image.open(str(roses[0]))
```

In []:

```
PIL.Image.open(str(roses[1]))
```

And some tulips:

In []:

```
tulips = list(data_dir.glob('tulips/*'))
PIL.Image.open(str(tulips[0]))
```

In []:

```
PIL.Image.open(str(tulips[1]))
```

Load data using a Keras utility

Let's load these images off disk using the helpful `tf.keras.utils.image_dataset_from_directory` utility. This will take you from a directory of images on disk to a `tf.data.Dataset` in just a couple lines of code. If you like, you can also write your own data loading code from scratch by visiting the [Load and preprocess images \(./load_data/images.ipynb\)](#) tutorial.

Create a dataset

Define some parameters for the loader:

In []:

```
batch_size = 32
img_height = 180
img_width = 180
```

It's good practice to use a validation split when developing your model. Let's use 80% of the images for training, and 20% for validation.

In []:

```
train_ds = tf.keras.utils.image_dataset_from_directory(  
    data_dir,  
    validation_split=0.2,  
    subset="training",  
    seed=123,  
    image_size=(img_height, img_width),  
    batch_size=batch_size)
```

In []:

```
val_ds = tf.keras.utils.image_dataset_from_directory(  
    data_dir,  
    validation_split=0.2,  
    subset="validation",  
    seed=123,  
    image_size=(img_height, img_width),  
    batch_size=batch_size)
```

You can find the class names in the `class_names` attribute on these datasets. These correspond to the directory names in alphabetical order.

In []:

```
class_names = train_ds.class_names  
print(class_names)
```

Visualize the data

Here are the first nine images from the training dataset:

In []:

```
import matplotlib.pyplot as plt  
  
plt.figure(figsize=(10, 10))  
for images, labels in train_ds.take(1):  
    for i in range(9):  
        ax = plt.subplot(3, 3, i + 1)  
        plt.imshow(images[i].numpy().astype("uint8"))  
        plt.title(class_names[labels[i]])  
        plt.axis("off")
```

You will train a model using these datasets by passing them to `Model.fit` in a moment. If you like, you can also manually iterate over the dataset and retrieve batches of images:

In []:

```
for image_batch, labels_batch in train_ds:  
    print(image_batch.shape)  
    print(labels_batch.shape)  
    break
```

The `image_batch` is a tensor of the shape `(32, 180, 180, 3)`. This is a batch of 32 images of shape `180x180x3` (the last dimension refers to color channels RGB). The `label_batch` is a tensor of the shape `(32,)`, these are corresponding labels to the 32 images.

You can call `.numpy()` on the `image_batch` and `labels_batch` tensors to convert them to a `numpy.ndarray`.

Configure the dataset for performance

Let's make sure to use buffered prefetching so you can yield data from disk without having I/O become blocking. These are two important methods you should use when loading data:

- `Dataset.cache` keeps the images in memory after they're loaded off disk during the first epoch. This will ensure the dataset does not become a bottleneck while training your model. If your dataset is too large to fit into memory, you can also use this method to create a performant on-disk cache.
- `Dataset.prefetch` overlaps data preprocessing and model execution while training.

Interested readers can learn more about both methods, as well as how to cache data to disk in the *Prefetching* section of the [Better performance with the `tf.data` API](#) ([.../guide/data_performance.ipynb](#)) guide.

In []:

```
AUTOTUNE = tf.data.AUTOTUNE

train_ds = train_ds.cache().shuffle(1000).prefetch(buffer_size=AUTOTUNE)
val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)
```

Standardize the data

The RGB channel values are in the `[0, 255]` range. This is not ideal for a neural network; in general you should seek to make your input values small.

Here, you will standardize values to be in the `[0, 1]` range by using `tf.keras.layers.Rescaling`:

In []:

```
normalization_layer = layers.Rescaling(1./255)
```

There are two ways to use this layer. You can apply it to the dataset by calling `Dataset.map`:

In []:

```
normalized_ds = train_ds.map(lambda x, y: (normalization_layer(x), y))
image_batch, labels_batch = next(iter(normalized_ds))
first_image = image_batch[0]
# Notice the pixel values are now in `[0,1]`.
print(np.min(first_image), np.max(first_image))
```

Or, you can include the layer inside your model definition, which can simplify deployment. Let's use the second approach here.

Note: You previously resized images using the `image_size` argument of `tf.keras.utils.image_dataset_from_directory`. If you want to include the resizing logic in your model as well, you can use the `tf.keras.layers.Resizing` layer.

Create the model

The [Sequential](https://www.tensorflow.org/guide/keras/sequential_model) (https://www.tensorflow.org/guide/keras/sequential_model) model consists of three convolution blocks (`tf.keras.layers.Conv2D`) with a max pooling layer (`tf.keras.layers.MaxPooling2D`) in each of them. There's a fully-connected layer (`tf.keras.layers.Dense`) with 128 units on top of it that is activated by a ReLU activation function (`'relu'`). This model has not been tuned for high accuracy—the goal of this tutorial is to show a standard approach.

In []:

```
num_classes = len(class_names)

model = Sequential([
    layers.Rescaling(1./255, input_shape=(img_height, img_width, 3)),
    layers.Conv2D(16, 3, padding='same', activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(32, 3, padding='same', activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(64, 3, padding='same', activation='relu'),
    layers.MaxPooling2D(),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(num_classes)
])
```

Compile the model

For this tutorial, choose the `tf.keras.optimizers.Adam` optimizer and `tf.keras.losses.SparseCategoricalCrossentropy` loss function. To view training and validation accuracy for each training epoch, pass the `metrics` argument to `Model.compile`.

In []:

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

Model summary

View all the layers of the network using the model's `Model.summary` method:

In []:

```
model.summary()
```

Train the model

In []:

```
epochs=10
history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=epochs
)
```

Visualize training results

Create plots of loss and accuracy on the training and validation sets:

In []:

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```

The plots show that training accuracy and validation accuracy are off by large margins, and the model has achieved only around 60% accuracy on the validation set.

Let's inspect what went wrong and try to increase the overall performance of the model.

Overfitting

In the plots above, the training accuracy is increasing linearly over time, whereas validation accuracy stalls around 60% in the training process. Also, the difference in accuracy between training and validation accuracy is noticeable—a sign of [overfitting](#) (https://www.tensorflow.org/tutorials/keras/overfit_and_underfit).

When there are a small number of training examples, the model sometimes learns from noises or unwanted details from training examples—to an extent that it negatively impacts the performance of the model on new examples. This phenomenon is known as overfitting. It means that the model will have a difficult time generalizing on a new dataset.

There are multiple ways to fight overfitting in the training process. In this tutorial, you'll use *data augmentation* and add *Dropout* to your model.

Data augmentation

Overfitting generally occurs when there are a small number of training examples. [Data augmentation \(./data_augmentation.ipynb\)](#) takes the approach of generating additional training data from your existing examples by augmenting them using random transformations that yield believable-looking images. This helps expose the model to more aspects of the data and generalize better.

You will implement data augmentation using the following Keras preprocessing layers: `tf.keras.layers.RandomFlip`, `tf.keras.layers.RandomRotation`, and `tf.keras.layers.RandomZoom`. These can be included inside your model like other layers, and run on the GPU.

In []:

```
data_augmentation = keras.Sequential(  
[  
    layers.RandomFlip("horizontal",  
                      input_shape=(img_height,  
                                  img_width,  
                                  3)),  
    layers.RandomRotation(0.1),  
    layers.RandomZoom(0.1),  
]  
)
```

Let's visualize what a few augmented examples look like by applying data augmentation to the same image several times:

In []:

```
plt.figure(figsize=(10, 10))  
for images, _ in train_ds.take(1):  
    for i in range(9):  
        augmented_images = data_augmentation(images)  
        ax = plt.subplot(3, 3, i + 1)  
        plt.imshow(augmented_images[0].numpy().astype("uint8"))  
        plt.axis("off")
```

You will use data augmentation to train a model in a moment.

Dropout

Another technique to reduce overfitting is to introduce [dropout \(\[https://developers.google.com/machine-learning/glossary#dropout_regularization\]\(https://developers.google.com/machine-learning/glossary#dropout_regularization\)\)](https://developers.google.com/machine-learning/glossary#dropout_regularization) regularization to the network.

When you apply dropout to a layer, it randomly drops out (by setting the activation to zero) a number of output units from the layer during the training process. Dropout takes a fractional number as its input value, in the form such as 0.1, 0.2, 0.4, etc. This means dropping out 10%, 20% or 40% of the output units randomly from the applied layer.

Let's create a new neural network with `tf.keras.layers.Dropout` before training it using the augmented images:

In []:

```
model = Sequential([  
    data_augmentation,  
    layers.Rescaling(1./255),  
    layers.Conv2D(16, 3, padding='same', activation='relu'),  
    layers.MaxPooling2D(),  
    layers.Conv2D(32, 3, padding='same', activation='relu'),  
    layers.MaxPooling2D(),  
    layers.Conv2D(64, 3, padding='same', activation='relu'),  
    layers.MaxPooling2D(),  
    layers.Dropout(0.2),  
    layers.Flatten(),  
    layers.Dense(128, activation='relu'),  
    layers.Dense(num_classes)  
)
```

Compile and train the model

In []:

```
model.compile(optimizer='adam',  
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),  
              metrics=['accuracy'])
```

In []:

```
model.summary()
```

In []:

```
epochs = 15  
history = model.fit(  
    train_ds,  
    validation_data=val_ds,  
    epochs=epochs  
)
```

Visualize training results

After applying data augmentation and `tf.keras.layers.Dropout`, there is less overfitting than before, and training and validation accuracy are closer aligned:

In []:

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```

Predict on new data

Finally, let's use our model to classify an image that wasn't included in the training or validation sets.

Note: Data augmentation and dropout layers are inactive at inference time.

In []:

```
sunflower_url = "https://storage.googleapis.com/download.tensorflow.org/example_images/592px-Red_sunflower.jpg"
sunflower_path = tf.keras.utils.get_file('Red_sunflower', origin=sunflower_url)

img = tf.keras.utils.load_img(
    sunflower_path, target_size=(img_height, img_width)
)
img_array = tf.keras.utils.img_to_array(img)
img_array = tf.expand_dims(img_array, 0) # Create a batch

predictions = model.predict(img_array)
score = tf.nn.softmax(predictions[0])

print(
    "This image most likely belongs to {} with a {:.2f} percent confidence."
    .format(class_names[np.argmax(score)], 100 * np.max(score))
)
```

Copyright 2020 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Data augmentation



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/images/data_augmentation)

(https://www.tensorflow.org/tutorials/images/data_augmentation)



[Run in Google Colab](#)

Overview

This tutorial demonstrates data augmentation: a technique to increase the diversity of your training set by applying random (but realistic) transformations, such as image rotation.

You will learn how to apply data augmentation in two ways:

- Use the Keras preprocessing layers, such as `tf.keras.layers.Resizing`, `tf.keras.layers.Rescaling`, `tf.keras.layers.RandomFlip`, and `tf.keras.layers.RandomRotation`.
- Use the `tf.image` methods, such as `tf.image.flip_left_right`, `tf.image.rgb_to_grayscale`, `tf.image.adjust_brightness`, `tf.image.central_crop`, and `tf.image.stateless_random*`.

Setup

In []:

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
import tensorflow_datasets as tfds

from tensorflow.keras import layers
```

Download a dataset

This tutorial uses the `tf_flowers` (https://www.tensorflow.org/datasets/catalog/tf_flowers) dataset. For convenience, download the dataset using [TensorFlow Datasets](#) (<https://www.tensorflow.org/datasets>). If you would like to learn about other ways of importing data, check out the [load images](#) (https://www.tensorflow.org/tutorials/load_data/images) tutorial.

In []:

```
(train_ds, val_ds, test_ds), metadata = tfds.load(
    'tf_flowers',
    split=['train[:80%]', 'train[80%:90%]', 'train[90%:]'],
    with_info=True,
    as_supervised=True,
)
```

The flowers dataset has five classes.

In []:

```
num_classes = metadata.features['label'].num_classes
print(num_classes)
```

Let's retrieve an image from the dataset and use it to demonstrate data augmentation.

In []:

```
get_label_name = metadata.features['label'].int2str

image, label = next(iter(train_ds))
_ = plt.imshow(image)
_ = plt.title(get_label_name(label))
```

Use Keras preprocessing layers

Resizing and rescaling

You can use the Keras preprocessing layers to resize your images to a consistent shape (with `tf.keras.layers.Resizing`), and to rescale pixel values (with `tf.keras.layers.Rescaling`).

In []:

```
IMG_SIZE = 180  
resize_and_rescale = tf.keras.Sequential([  
    layers.Resizing(IMG_SIZE, IMG_SIZE),  
    layers.Rescaling(1./255)  
])
```

Note: The rescaling layer above standardizes pixel values to the `[0, 1]` range. If instead you wanted it to be `[-1, 1]`, you would write `tf.keras.layers.Rescaling(1./127.5, offset=-1)`.

You can visualize the result of applying these layers to an image.

In []:

```
result = resize_and_rescale(image)  
_ = plt.imshow(result)
```

Verify that the pixels are in the `[0, 1]` range:

In []:

```
print("Min and max pixel values:", result.numpy().min(), result.numpy().max())
```

Data augmentation

You can use the Keras preprocessing layers for data augmentation as well, such as `tf.keras.layers.RandomFlip` and `tf.keras.layers.RandomRotation`.

Let's create a few preprocessing layers and apply them repeatedly to the same image.

In []:

```
data_augmentation = tf.keras.Sequential([  
    layers.RandomFlip("horizontal_and_vertical"),  
    layers.RandomRotation(0.2),  
])
```

In []:

```
# Add the image to a batch.  
image = tf.cast(tf.expand_dims(image, 0), tf.float32)
```

In []:

```
plt.figure(figsize=(10, 10))  
for i in range(9):  
    augmented_image = data_augmentation(image)  
    ax = plt.subplot(3, 3, i + 1)  
    plt.imshow(augmented_image[0])  
    plt.axis("off")
```

There are a variety of preprocessing layers you can use for data augmentation including `tf.keras.layers.RandomContrast`, `tf.keras.layers.RandomCrop`, `tf.keras.layers.RandomZoom`, and others.

Two options to use the Keras preprocessing layers

There are two ways you can use these preprocessing layers, with important trade-offs.

Option 1: Make the preprocessing layers part of your model

In []:

```
model = tf.keras.Sequential([  
    # Add the preprocessing layers you created earlier.  
    resize_and_rescale,  
    data_augmentation,  
    layers.Conv2D(16, 3, padding='same', activation='relu'),  
    layers.MaxPooling2D(),  
    # Rest of your model.  
])
```

There are two important points to be aware of in this case:

- Data augmentation will run on-device, synchronously with the rest of your layers, and benefit from GPU acceleration.
- When you export your model using `model.save`, the preprocessing layers will be saved along with the rest of your model. If you later deploy this model, it will automatically standardize images (according to the configuration of your layers). This can save you from the effort of having to reimplement that logic server-side.

Note: Data augmentation is inactive at test time so input images will only be augmented during calls to `Model.fit` (not `Model.evaluate` or `Model.predict`).

Option 2: Apply the preprocessing layers to your dataset

In []:

```
aug_ds = train_ds.map(  
    lambda x, y: (resize_and_rescale(x, training=True), y))
```

With this approach, you use `Dataset.map` to create a dataset that yields batches of augmented images. In this case:

- Data augmentation will happen asynchronously on the CPU, and is non-blocking. You can overlap the training of your model on the GPU with data preprocessing, using `Dataset.prefetch`, shown below.
- In this case the preprocessing layers will not be exported with the model when you call `Model.save`. You will need to attach them to your model before saving it or reimplement them server-side. After training, you can attach the preprocessing layers before export.

You can find an example of the first option in the [Image classification \(classification.ipynb\)](#) tutorial. Let's demonstrate the second option here.

Apply the preprocessing layers to the datasets

Configure the training, validation, and test datasets with the Keras preprocessing layers you created earlier. You will also configure the datasets for performance, using parallel reads and buffered prefetching to yield batches from disk without I/O become blocking. (Learn more dataset performance in the [Better performance with the tf.data API \(https://www.tensorflow.org/guide/data_performance\)](#) guide.)

Note: Data augmentation should only be applied to the training set.

In []:

```
batch_size = 32  
AUTOTUNE = tf.data.AUTOTUNE  
  
def prepare(ds, shuffle=False, augment=False):  
    # Resize and rescale all datasets.  
    ds = ds.map(lambda x, y: (resize_and_rescale(x), y),  
               num_parallel_calls=AUTOTUNE)  
  
    if shuffle:  
        ds = ds.shuffle(1000)  
  
    # Batch all datasets.  
    ds = ds.batch(batch_size)  
  
    # Use data augmentation only on the training set.  
    if augment:  
        ds = ds.map(lambda x, y: (data_augmentation(x, training=True), y),  
                   num_parallel_calls=AUTOTUNE)  
  
    # Use buffered prefetching on all datasets.  
    return ds.prefetch(buffer_size=AUTOTUNE)
```

In []:

```
train_ds = prepare(train_ds, shuffle=True, augment=True)  
val_ds = prepare(val_ds)  
test_ds = prepare(test_ds)
```

Train a model

For completeness, you will now train a model using the datasets you have just prepared.

The [Sequential](https://www.tensorflow.org/guide/keras/sequential_model) (https://www.tensorflow.org/guide/keras/sequential_model) model consists of three convolution blocks (`tf.keras.layers.Conv2D`) with a max pooling layer (`tf.keras.layers.MaxPooling2D`) in each of them. There's a fully-connected layer (`tf.keras.layers.Dense`) with 128 units on top of it that is activated by a ReLU activation function ('`relu`'). This model has not been tuned for accuracy (the goal is to show you the mechanics).

In []:

```
model = tf.keras.Sequential([
    layers.Conv2D(16, 3, padding='same', activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(32, 3, padding='same', activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(64, 3, padding='same', activation='relu'),
    layers.MaxPooling2D(),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(num_classes)
])
```

Choose the `tf.keras.optimizers.Adam` optimizer and `tf.keras.losses.SparseCategoricalCrossentropy` loss function. To view training and validation accuracy for each training epoch, pass the `metrics` argument to `Model.compile`.

In []:

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

Train for a few epochs:

In []:

```
epochs=5
history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=epochs
)
```

In []:

```
loss, acc = model.evaluate(test_ds)
print("Accuracy", acc)
```

Custom data augmentation

You can also create custom data augmentation layers.

This section of the tutorial shows two ways of doing so:

- First, you will create a `tf.keras.layers.Lambda` layer. This is a good way to write concise code.
- Next, you will write a new layer via [subclassing](https://www.tensorflow.org/guide/keras/custom_layers_and_models) (https://www.tensorflow.org/guide/keras/custom_layers_and_models), which gives you more control.

Both layers will randomly invert the colors in an image, according to some probability.

In []:

```
def random_invert_img(x, p=0.5):
    if tf.random.uniform([]) < p:
        x = (255-x)
    else:
        x
    return x
```

In []:

```
def random_invert(factor=0.5):
    return layers.Lambda(lambda x: random_invert_img(x, factor))

random_invert = random_invert()
```

In []:

```
plt.figure(figsize=(10, 10))
for i in range(9):
    augmented_image = random_invert(image)
    ax = plt.subplot(3, 3, i + 1)
    plt.imshow(augmented_image[0].numpy().astype("uint8"))
    plt.axis("off")
```

Next, implement a custom layer by [subclassing](https://www.tensorflow.org/guide/keras/custom_layers_and_models) (https://www.tensorflow.org/guide/keras/custom_layers_and_models):

In []:

```
class RandomInvert(layers.Layer):
    def __init__(self, factor=0.5, **kwargs):
        super().__init__(**kwargs)
        self.factor = factor

    def call(self, x):
        return random_invert_img(x)
```

In []:

```
_ = plt.imshow(RandomInvert()(image)[0])
```

Both of these layers can be used as described in options 1 and 2 above.

Using tf.image

The above Keras preprocessing utilities are convenient. But, for finer control, you can write your own data augmentation pipelines or layers using `tf.data` and `tf.image`. (You may also want to check out [TensorFlow Addons Image: Operations](https://www.tensorflow.org/addons/tutorials/image_ops) (https://www.tensorflow.org/addons/tutorials/image_ops) and [TensorFlow I/O: Color Space Conversions](https://www.tensorflow.org/io/tutorials/colorspace) (<https://www.tensorflow.org/io/tutorials/colorspace>).

Since the flowers dataset was previously configured with data augmentation, let's reimport it to start fresh:

In []:

```
(train_ds, val_ds, test_ds), metadata = tfds.load(
    'tf_flowers',
    split=['train[:80%]', 'train[80%:90%]', 'train[90%:]'],
    with_info=True,
    as_supervised=True,
)
```

Retrieve an image to work with:

In []:

```
image, label = next(iter(train_ds))
_ = plt.imshow(image)
_ = plt.title(get_label_name(label))
```

Let's use the following function to visualize and compare the original and augmented images side-by-side:

In []:

```
def visualize(original, augmented):
    fig = plt.figure()
    plt.subplot(1,2,1)
    plt.title('Original image')
    plt.imshow(original)

    plt.subplot(1,2,2)
    plt.title('Augmented image')
    plt.imshow(augmented)
```

Data augmentation

Flip an image

Flip an image either vertically or horizontally with `tf.image.flip_left_right`:

In []:

```
flipped = tf.image.flip_left_right(image)
visualize(image, flipped)
```

Grayscale an image

You can grayscale an image with `tf.image.rgb_to_grayscale`:

In []:

```
grayscaled = tf.image.rgb_to_grayscale(image)
visualize(image, tf.squeeze(grayscaled))
_ = plt.colorbar()
```

Saturate an image

Saturate an image with `tf.image.adjust_saturation` by providing a saturation factor:

In []:

```
saturated = tf.image.adjust_saturation(image, 3)
visualize(image, saturated)
```

Change image brightness

Change the brightness of image with `tf.image.adjust_brightness` by providing a brightness factor:

In []:

```
bright = tf.image.adjust_brightness(image, 0.4)
visualize(image, bright)
```

Center crop an image

Crop the image from center up to the image part you desire using `tf.image.central_crop`:

In []:

```
cropped = tf.image.central_crop(image, central_fraction=0.5)
visualize(image, cropped)
```

Rotate an image

Rotate an image by 90 degrees with `tf.image.rot90`:

In []:

```
rotated = tf.image.rot90(image)
visualize(image, rotated)
```

Random transformations

Warning: There are two sets of random image operations: `tf.image.random*` and `tf.image.stateless_random*`. Using `tf.image.random*` operations is strongly discouraged as they use the old RNGs from TF 1.x. Instead, please use the random image operations introduced in this tutorial. For more information, refer to [Random number generation \(../../guide/random_numbers.ipynb\)](#).

Applying random transformations to the images can further help generalize and expand the dataset. The current `tf.image` API provides eight such random image operations (ops):

- `tf.image.stateless_random_brightness` (https://www.tensorflow.org/api_docs/python/tf/image/stateless_random_brightness)
- `tf.image.stateless_random_contrast` (https://www.tensorflow.org/api_docs/python/tf/image/stateless_random_contrast)
- `tf.image.stateless_random_crop` (https://www.tensorflow.org/api_docs/python/tf/image/stateless_random_crop)
- `tf.image.stateless_random_flip_left_right` (https://www.tensorflow.org/api_docs/python/tf/image/stateless_random_flip_left_right)
- `tf.image.stateless_random_flip_up_down` (https://www.tensorflow.org/api_docs/python/tf/image/stateless_random_flip_up_down)
- `tf.image.stateless_random_hue` (https://www.tensorflow.org/api_docs/python/tf/image/stateless_random_hue)
- `tf.image.stateless_random_jpeg_quality` (https://www.tensorflow.org/api_docs/python/tf/image/stateless_random_jpeg_quality)
- `tf.image.stateless_random_saturation` (https://www.tensorflow.org/api_docs/python/tf/image/stateless_random_saturation)

These random image ops are purely functional: the output only depends on the input. This makes them simple to use in high performance, deterministic input pipelines. They require a `seed` value be input each step. Given the same `seed`, they return the same results independent of how many times they are called.

Note: `seed` is a `Tensor` of shape `(2,)` whose values are any integers.

In the following sections, you will:

1. Go over examples of using random image operations to transform an image.
2. Demonstrate how to apply random transformations to a training dataset.

Randomly change image brightness

Randomly change the brightness of `image` using `tf.image.stateless_random_brightness` by providing a brightness factor and `seed`. The brightness factor is chosen randomly in the range `[-max_delta, max_delta]` and is associated with the given `seed`.

In []:

```
for i in range(3):
    seed = (i, 0) # tuple of size (2,)
    stateless_random_brightness = tf.image.stateless_random_brightness(
        image, max_delta=0.95, seed=seed)
    visualize(image, stateless_random_brightness)
```

Randomly change image contrast

Randomly change the contrast of `image` using `tf.image.stateless_random_contrast` by providing a contrast range and `seed`. The contrast range is chosen randomly in the interval `[lower, upper]` and is associated with the given `seed`.

In []:

```
for i in range(3):
    seed = (i, 0) # tuple of size (2,)
    stateless_random_contrast = tf.image.stateless_random_contrast(
        image, lower=0.1, upper=0.9, seed=seed)
    visualize(image, stateless_random_contrast)
```

Randomly crop an image

Randomly crop `image` using `tf.image.stateless_random_crop` by providing target `size` and `seed`. The portion that gets cropped out of `image` is at a randomly chosen offset and is associated with the given `seed`.

In []:

```
for i in range(3):
    seed = (i, 0) # tuple of size (2,)
    stateless_random_crop = tf.image.stateless_random_crop(
        image, size=[210, 300, 3], seed=seed)
    visualize(image, stateless_random_crop)
```

Apply augmentation to a dataset

Let's first download the image dataset again in case they are modified in the previous sections.

In []:

```
(train_datasets, val_ds, test_ds), metadata = tfds.load(  
    'tf_flowers',  
    split=['train[:80%]', 'train[80%:90%]', 'train[90%:]'],  
    with_info=True,  
    as_supervised=True,  
)
```

Next, define a utility function for resizing and rescaling the images. This function will be used in unifying the size and scale of images in the dataset:

In []:

```
def resize_and_rescale(image, label):  
    image = tf.cast(image, tf.float32)  
    image = tf.image.resize(image, [IMG_SIZE, IMG_SIZE])  
    image = (image / 255.0)  
    return image, label
```

Let's also define the `augment` function that can apply the random transformations to the images. This function will be used on the dataset in the next step.

In []:

```
def augment(image_label):  
    image, label = image_label  
    image, label = resize_and_rescale(image, label)  
    image = tf.image.resize_with_crop_or_pad(image, IMG_SIZE + 6, IMG_SIZE + 6)  
    # Make a new seed.  
    new_seed = tf.random.experimental.stateless_split(seed, num=1)[0, :]  
    # Random crop back to the original size.  
    image = tf.image.stateless_random_crop(  
        image, size=[IMG_SIZE, IMG_SIZE, 3], seed=new_seed)  
    # Random brightness.  
    image = tf.image.stateless_random_brightness(  
        image, max_delta=0.5, seed=new_seed)  
    image = tf.clip_by_value(image, 0, 1)  
    return image, label
```

Option 1: Using `tf.data.experimental.Counter`

Create a `tf.data.experimental.Counter` object (let's call it `counter`) and `Dataset.zip` the dataset with `(counter, counter)`. This will ensure that each image in the dataset gets associated with a unique value (of shape `(2,)`) based on `counter` which later can get passed into the `augment` function as the `seed` value for random transformations.

In []:

```
# Create a `Counter` object and `Dataset.zip` it together with the training set.  
counter = tf.data.experimental.Counter()  
train_ds = tf.data.Dataset.zip((train_datasets, (counter, counter)))
```

Map the `augment` function to the training dataset:

In []:

```
train_ds = (  
    train_ds  
    .shuffle(1000)  
    .map(augment, num_parallel_calls=AUTOTUNE)  
    .batch(batch_size)  
    .prefetch(AUTOTUNE)  
)
```

In []:

```
val_ds = (  
    val_ds  
    .map(resize_and_rescale, num_parallel_calls=AUTOTUNE)  
    .batch(batch_size)  
    .prefetch(AUTOTUNE)  
)
```

In []:

```
test_ds = (
    test_ds
    .map(resize_and_rescale, num_parallel_calls=AUTOTUNE)
    .batch(batch_size)
    .prefetch(AUTOTUNE)
)
```

Option 2: Using `tf.random.Generator`

- Create a `tf.random.Generator` object with an initial `seed` value. Calling the `make_seeds` function on the same generator object always returns a new, unique `seed` value.
- Define a wrapper function that: 1) calls the `make_seeds` function; and 2) passes the newly generated `seed` value into the `augment` function for random transformations.

Note: `tf.random.Generator` objects store RNG state in a `tf.Variable`, which means it can be saved as a [checkpoint](#) ([./guide/checkpoint.ipynb](#)) or in a [SavedModel](#) ([./guide/saved_model.ipynb](#)). For more details, please refer to [Random number generation](#) ([./guide/random_numbers.ipynb](#)).

In []:

```
# Create a generator.
rng = tf.random.Generator.from_seed(123, alg='philox')
```

In []:

```
# Create a wrapper function for updating seeds.
def f(x, y):
    seed = rng.make_seeds(2)[0]
    image, label = augment((x, y), seed)
    return image, label
```

Map the wrapper function `f` to the training dataset, and the `resize_and_rescale` function—to the validation and test sets:

In []:

```
train_ds = (
    train_datasets
    .shuffle(1000)
    .map(f, num_parallel_calls=AUTOTUNE)
    .batch(batch_size)
    .prefetch(AUTOTUNE)
)
```

In []:

```
val_ds = (
    val_ds
    .map(resize_and_rescale, num_parallel_calls=AUTOTUNE)
    .batch(batch_size)
    .prefetch(AUTOTUNE)
)
```

In []:

```
test_ds = (
    test_ds
    .map(resize_and_rescale, num_parallel_calls=AUTOTUNE)
    .batch(batch_size)
    .prefetch(AUTOTUNE)
)
```

These datasets can now be used to train a model as shown previously.

Next steps

This tutorial demonstrated data augmentation using Keras preprocessing layers and `tf.image`.

- To learn how to include preprocessing layers inside your model, refer to the [Image classification \(classification.ipynb\)](#) tutorial.
- You may also be interested in learning how preprocessing layers can help you classify text, as shown in the [Basic text classification](#) ([./keras/text_classification.ipynb](#)) tutorial.
- You can learn more about `tf.data` in this [guide](#) ([./../guide/data.ipynb](#)), and you can learn how to configure your input pipelines for performance [here](#) ([./../guide/data_performance.ipynb](#)).

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Transfer learning with TensorFlow Hub



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/images/transfer_learning_with_hub)



(https://www.tensorflow.org/tutorials/images/transfer_learning_with_hub) (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/images/transfer_le

TensorFlow Hub (<https://tfhub.dev/>) is a repository of pre-trained TensorFlow models.

This tutorial demonstrates how to:

1. Use models from TensorFlow Hub with `tf.keras`.
2. Use an image classification model from TensorFlow Hub.
3. Do simple transfer learning to fine-tune a model for your own image classes.

Setup

In []:

```
import numpy as np
import time

import PIL.Image as Image
import matplotlib.pyplot as plt

import tensorflow as tf
import tensorflow_hub as hub

import datetime

%load_ext tensorboard
```

An ImageNet classifier

You'll start by using a classifier model pre-trained on the [ImageNet](https://en.wikipedia.org/wiki/ImageNet) (<https://en.wikipedia.org/wiki/ImageNet>) benchmark dataset—no initial training required!

Download the classifier

Select a [MobileNetV2](https://arxiv.org/abs/1801.04381) (<https://arxiv.org/abs/1801.04381>) pre-trained model [from TensorFlow Hub](https://tfhub.dev/google/tf2-preview/mobilenet_v2/classification/2) (https://tfhub.dev/google/tf2-preview/mobilenet_v2/classification/2) and wrap it as a Keras layer with `hub.KerasLayer` (https://www.tensorflow.org/hub/api_docs/python/hub/KerasLayer). Any [compatible image classifier model](https://tfhub.dev/s?q=tf2&module-type=image-classification) (<https://tfhub.dev/s?q=tf2&module-type=image-classification>) from TensorFlow Hub will work here, including the examples provided in the drop-down below.

In []:

```
mobilenet_v2 ="https://tfhub.dev/google/tf2-preview/mobilenet_v2/classification/4"
inception_v3 = "https://tfhub.dev/google/imagenet/inception_v3/classification/5"

classifier_model = mobilenet_v2 #@param ["mobilenet_v2", "inception_v3"] {type:"raw"}
```

In []:

```
IMAGE_SHAPE = (224, 224)

classifier = tf.keras.Sequential([
    hub.KerasLayer(classifier_model, input_shape=IMAGE_SHAPE+(3, ))
])
```

Run it on a single image

Download a single image to try the model on:

In []:

```
grace_hopper = tf.keras.utils.get_file('image.jpg', 'https://storage.googleapis.com/download.tensorflow.org/example_images/grace_hopper.jpg')
grace_hopper = Image.open(grace_hopper).resize(IMAGE_SHAPE)
grace_hopper
```

In []:

```
grace_hopper = np.array(grace_hopper)/255.0
grace_hopper.shape
```

Add a batch dimension (with `np.newaxis`) and pass the image to the model:

In []:

```
result = classifier.predict(grace_hopper[np.newaxis, ...])
result.shape
```

The result is a 1001-element vector of logits, rating the probability of each class for the image.

The top class ID can be found with `tf.math.argmax`:

In []:

```
predicted_class = tf.math.argmax(result[0], axis=-1)
predicted_class
```

Decode the predictions

Take the `predicted_class` ID (such as 653) and fetch the ImageNet dataset labels to decode the predictions:

In []:

```
labels_path = tf.keras.utils.get_file('ImageNetLabels.txt', 'https://storage.googleapis.com/download.tensorflow.org/data/ImageNetLabels.txt')
imagenet_labels = np.array(open(labels_path).read().splitlines())
```

In []:

```
plt.imshow(grace_hopper)
plt.axis('off')
predicted_class_name = imagenet_labels[predicted_class]
_ = plt.title("Prediction: " + predicted_class_name.title())
```

Simple transfer learning

But what if you want to create a custom classifier using your own dataset that has classes that aren't included in the original ImageNet dataset (that the pre-trained model was trained on)?

To do that, you can:

1. Select a pre-trained model from TensorFlow Hub; and
2. Retrain the top (last) layer to recognize the classes from your custom dataset.

Dataset

In this example, you will use the TensorFlow flowers dataset:

In []:

```
data_root = tf.keras.utils.get_file(  
    'flower_photos',  
    'https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz',  
    untar=True)
```

First, load this data into the model using the image data off disk with `tf.keras.utils.image_dataset_from_directory`, which will generate a `tf.data.Dataset`:

In []:

```
batch_size = 32  
img_height = 224  
img_width = 224  
  
train_ds = tf.keras.utils.image_dataset_from_directory(  
    str(data_root),  
    validation_split=0.2,  
    subset="training",  
    seed=123,  
    image_size=(img_height, img_width),  
    batch_size=batch_size  
)  
  
val_ds = tf.keras.utils.image_dataset_from_directory(  
    str(data_root),  
    validation_split=0.2,  
    subset="validation",  
    seed=123,  
    image_size=(img_height, img_width),  
    batch_size=batch_size  
)
```

The flowers dataset has five classes:

In []:

```
class_names = np.array(train_ds.class_names)  
print(class_names)
```

Second, because TensorFlow Hub's convention for image models is to expect float inputs in the `[0, 1]` range, use the `tf.keras.layers.Rescaling` preprocessing layer to achieve this.

Note: You could also include the `tf.keras.layers.Rescaling` layer inside the model. Refer to the [Working with preprocessing layers](#) (https://www.tensorflow.org/guide/keras/preprocessing_layers) guide for a discussion of the tradeoffs.

In []:

```
normalization_layer = tf.keras.layers.Rescaling(1./255)  
train_ds = train_ds.map(lambda x, y: (normalization_layer(x), y)) # Where x=images, y=labels.  
val_ds = val_ds.map(lambda x, y: (normalization_layer(x), y)) # Where x=images, y=labels.
```

Third, finish the input pipeline by using buffered prefetching with `Dataset.prefetch`, so you can yield the data from disk without I/O blocking issues.

These are some of the most important `tf.data` methods you should use when loading data. Interested readers can learn more about them, as well as how to cache data to disk and other techniques, in the [Better performance with the tf.data API](#) (https://www.tensorflow.org/guide/data_performance#prefetching) guide.

In []:

```
AUTOTUNE = tf.data.AUTOTUNE  
train_ds = train_ds.cache().prefetch(buffer_size=AUTOTUNE)  
val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)
```

In []:

```
for image_batch, labels_batch in train_ds:  
    print(image_batch.shape)  
    print(labels_batch.shape)  
    break
```

Run the classifier on a batch of images

Now, run the classifier on an image batch:

In []:

```
result_batch = classifier.predict(train_ds)
```

In []:

```
predicted_class_names = imagenet_labels[tf.math.argmax(result_batch, axis=-1)]  
predicted_class_names
```

Check how these predictions line up with the images:

In []:

```
plt.figure(figsize=(10,9))  
plt.subplots_adjust(hspace=0.5)  
for n in range(30):  
    plt.subplot(6,5,n+1)  
    plt.imshow(image_batch[n])  
    plt.title(predicted_class_names[n])  
    plt.axis('off')  
_ = plt.suptitle("ImageNet predictions")
```

Note: all images are licensed CC-BY, creators are listed in the LICENSE.txt file.

The results are far from perfect, but reasonable considering that these are not the classes the model was trained for (except for "daisy").

Download the headless model

TensorFlow Hub also distributes models without the top classification layer. These can be used to easily perform transfer learning.

Select a [MobileNetV2](https://arxiv.org/abs/1801.04381) (<https://arxiv.org/abs/1801.04381>) pre-trained model [from TensorFlow Hub](https://tfhub.dev/google/tf2-preview/mobilenet_v2/feature_vector/4) (https://tfhub.dev/google/tf2-preview/mobilenet_v2/feature_vector/4). Any [compatible image feature vector model](https://tfhub.dev/s?module-type=image-feature-vector&q=tf2) (<https://tfhub.dev/s?module-type=image-feature-vector&q=tf2>) from TensorFlow Hub will work here, including the examples from the drop-down menu.

In []:

```
mobilenet_v2 = "https://tfhub.dev/google/tf2-preview/mobilenet_v2/feature_vector/4"  
inception_v3 = "https://tfhub.dev/google/tf2-preview/inception_v3/feature_vector/4"  
  
feature_extractor_model = mobilenet_v2 #@param ["mobilenet_v2", "inception_v3"] {type: "raw"}
```

Create the feature extractor by wrapping the pre-trained model as a Keras layer with [hub.KerasLayer](https://www.tensorflow.org/hub/api_docs/python/hub/KerasLayer) (https://www.tensorflow.org/hub/api_docs/python/hub/KerasLayer). Use the `trainable=False` argument to freeze the variables, so that the training only modifies the new classifier layer:

In []:

```
feature_extractor_layer = hub.KerasLayer(  
    feature_extractor_model,  
    input_shape=(224, 224, 3),  
    trainable=False)
```

The feature extractor returns a 1280-long vector for each image (the image batch size remains at 32 in this example):

In []:

```
feature_batch = feature_extractor_layer(image_batch)  
print(feature_batch.shape)
```

Attach a classification head

To complete the model, wrap the feature extractor layer in a `tf.keras.Sequential` model and add a fully-connected layer for classification:

In []:

```
num_classes = len(class_names)

model = tf.keras.Sequential([
    feature_extractor_layer,
    tf.keras.layers.Dense(num_classes)
])

model.summary()
```

In []:

```
predictions = model(image_batch)
```

In []:

```
predictions.shape
```

Train the model

Use `Model.compile` to configure the training process and add a `tf.keras.callbacks.TensorBoard` callback to create and store logs:

In []:

```
model.compile(
    optimizer=tf.keras.optimizers.Adam(),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=['acc'])

log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(
    log_dir=log_dir,
    histogram_freq=1) # Enable histogram computation for every epoch.
```

Now use the `Model.fit` method to train the model.

To keep this example short, you'll be training for just 10 epochs. To visualize the training progress in TensorBoard later, create and store logs an a [TensorBoard callback](https://www.tensorflow.org/tensorboard/get_started#using_tensorboard_with_keras_modelfit) (https://www.tensorflow.org/tensorboard/get_started#using_tensorboard_with_keras_modelfit).

In []:

```
NUM_EPOCHS = 10

history = model.fit(train_ds,
                     validation_data=val_ds,
                     epochs=NUM_EPOCHS,
                     callbacks=tensorboard_callback)
```

Start the TensorBoard to view how the metrics change with each epoch and to track other scalar values:

In []:

```
%tensorboard --logdir logs/fit
```

Check the predictions

Obtain the ordered list of class names from the model predictions:

In []:

```
predicted_batch = model.predict(image_batch)
predicted_id = tf.math.argmax(predicted_batch, axis=-1)
predicted_label_batch = class_names[predicted_id]
print(predicted_label_batch)
```

Plot the model predictions:

In []:

```
plt.figure(figsize=(10,9))
plt.subplots_adjust(hspace=0.5)

for n in range(30):
    plt.subplot(6,5,n+1)
    plt.imshow(image_batch[n])
    plt.title(predicted_label_batch[n].title())
    plt.axis('off')
_ = plt.suptitle("Model predictions")
```

Export and reload your model

Now that you've trained the model, export it as a SavedModel for reusing it later.

In []:

```
t = time.time()

export_path = "/tmp/saved_models/{}".format(int(t))
model.save(export_path)

export_path
```

Confirm that you can reload the SavedModel and that the model is able to output the same results:

In []:

```
reloaded = tf.keras.models.load_model(export_path)
```

In []:

```
result_batch = model.predict(image_batch)
reloaded_result_batch = reloaded.predict(image_batch)
```

In []:

```
abs(reloaded_result_batch - result_batch).max()
```

In []:

```
reloaded_predicted_id = tf.math.argmax(reloaded_result_batch, axis=-1)
reloaded_predicted_label_batch = class_names[reloaded_predicted_id]
print(reloaded_predicted_label_batch)
```

In []:

```
plt.figure(figsize=(10,9))
plt.subplots_adjust(hspace=0.5)
for n in range(30):
    plt.subplot(6,5,n+1)
    plt.imshow(image_batch[n])
    plt.title(reloaded_predicted_label_batch[n].title())
    plt.axis('off')
_ = plt.suptitle("Model predictions")
```

Next steps

You can use the SavedModel to load for inference or convert it to a [TensorFlow Lite](https://www.tensorflow.org/lite/convert/) (<https://www.tensorflow.org/lite/convert/>) model (for on-device machine learning) or a [TensorFlow.js](https://www.tensorflow.org/js/tutorials#convert_pretrained_models_to_tensorflowjs) (https://www.tensorflow.org/js/tutorials#convert_pretrained_models_to_tensorflowjs) model (for machine learning in JavaScript).

Discover [more tutorials](https://www.tensorflow.org/hub/tutorials) (<https://www.tensorflow.org/hub/tutorials>) to learn how to use pre-trained models from TensorFlow Hub on image, text, audio, and video tasks.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Create an Estimator from a Keras model



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/estimator/keras_model_to_estimator)



(https://www.tensorflow.org/tutorials/estimator/keras_model_to_estimator) (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/estimator/keras_n

Warning: Estimators are not recommended for new code. Estimators run v1.Session -style code which is more difficult to write correctly, and can behave unexpectedly, especially when combined with TF 2 code. Estimators do fall under our [compatibility guarantees](https://tensorflow.org/guide/compatibility) (<https://tensorflow.org/guide/compatibility>), but will receive no fixes other than security vulnerabilities. See the [migration guide](https://tensorflow.org/guide/migrate) (<https://tensorflow.org/guide/migrate>) for details.

Overview

TensorFlow Estimators are supported in TensorFlow, and can be created from new and existing `tf.keras` models. This tutorial contains a complete, minimal example of that process.

Note: If you have a Keras model, you can use it directly with `tf.distribute` [strategies](https://tensorflow.org/guide/migrate/guide/distributed_training) (https://tensorflow.org/guide/migrate/guide/distributed_training) without converting it to an estimator. As such, `model_to_estimator` is no longer recommended.

Setup

In []:

```
import tensorflow as tf

import numpy as np
import tensorflow_datasets as tfds
```

Create a simple Keras model.

In Keras, you assemble *layers* to build *models*. A model is (usually) a graph of layers. The most common type of model is a stack of layers: the `tf.keras.Sequential` model.

To build a simple, fully-connected network (i.e. multi-layer perceptron):

In []:

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(16, activation='relu', input_shape=(4,)),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(3)
])
```

Compile the model and get a summary.

In []:

```
model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              optimizer='adam')
model.summary()
```

Create an input function

Use the [Datasets API \(../../guide/data.md\)](#) to scale to large datasets or multi-device training.

Estimators need control of when and how their input pipeline is built. To allow this, they require an "Input function" or `input_fn`. The Estimator will call this function with no arguments. The `input_fn` must return a `tf.data.Dataset`.

In []:

```
def input_fn():
    split = tfds.Split.TRAIN
    dataset = tfds.load('iris', split=split, as_supervised=True)
    dataset = dataset.map(lambda features, labels: ({'dense_input':features}, labels))
    dataset = dataset.batch(32).repeat()
    return dataset
```

Test out your `input_fn`

In []:

```
for features_batch, labels_batch in input_fn().take(1):
    print(features_batch)
    print(labels_batch)
```

Create an Estimator from the `tf.keras` model.

A `tf.keras.Model` can be trained with the `tf.estimator` API by converting the model to an `tf.estimator.Estimator` object with `tf.keras.estimator.model_to_estimator`.

In []:

```
import tempfile
model_dir = tempfile.mkdtemp()
keras_estimator = tf.keras.estimator.model_to_estimator(
    keras_model=model, model_dir=model_dir)
```

Train and evaluate the estimator.

In []:

```
keras_estimator.train(input_fn=input_fn, steps=500)
eval_result = keras_estimator.evaluate(input_fn=input_fn, steps=10)
print('Eval result: {}'.format(eval_result))
```

Copyright 2019 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Build a linear model with Estimators



[View on TensorFlow.org](#)

(<https://www.tensorflow.org/tutorials/estimator/linear>) (<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/estimator/linear.ipynb>) (<https://github.com/tensorflow/tensorflow/blob/master/tensorflow/tutorials/estimator/linear.ipynb>)



[Run in Google Colab](#)

Warning: Estimators are not recommended for new code. Estimators run `v1.Session`-style code which is more difficult to write correctly, and can behave unexpectedly, especially when combined with TF 2 code. Estimators do fall under our [compatibility guarantees](https://tensorflow.org/guide/versions) (<https://tensorflow.org/guide/versions>), but will receive no fixes other than security vulnerabilities. See the [migration guide](https://tensorflow.org/guide/migrate) (<https://tensorflow.org/guide/migrate>) for details.

Overview

This end-to-end walkthrough trains a logistic regression model using the `tf.estimator` API. The model is often used as a baseline for other, more complex, algorithms.

Note: A Keras logistic regression example is [available](https://tensorflow.org/guide/migrate/tutorials/keras/regression) (<https://tensorflow.org/guide/migrate/tutorials/keras/regression>) and is recommended over this tutorial.

Setup

In []:

```
!pip install sklearn
```

In []:

```
import os
import sys

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from IPython.display import clear_output
from six.moves import urllib
```

Load the titanic dataset

You will use the Titanic dataset with the (rather morbid) goal of predicting passenger survival, given characteristics such as gender, age, class, etc.

In []:

```
import tensorflow.compat.v2.feature_column as fc
import tensorflow as tf
```

In []:

```
# Load dataset.
dftrain = pd.read_csv('https://storage.googleapis.com/tf-datasets/titanic/train.csv')
dfeval = pd.read_csv('https://storage.googleapis.com/tf-datasets/titanic/eval.csv')
y_train = dftrain.pop('survived')
y_eval = dfeval.pop('survived')
```

Explore the data

The dataset contains the following features

In []:

```
dftrain.head()
```

In []:

```
dftrain.describe()
```

There are 627 and 264 examples in the training and evaluation sets, respectively.

In []:

```
dftrain.shape[0], dfeval.shape[0]
```

The majority of passengers are in their 20's and 30's.

```
In [ ]:
```

```
dftrain.age.hist(bins=20)
```

There are approximately twice as many male passengers as female passengers aboard.

```
In [ ]:
```

```
dftrain.sex.value_counts().plot(kind='barh')
```

The majority of passengers were in the "third" class.

```
In [ ]:
```

```
dftrain['class'].value_counts().plot(kind='barh')
```

Females have a much higher chance of surviving versus males. This is clearly a predictive feature for the model.

```
In [ ]:
```

```
pd.concat([dftrain, y_train], axis=1).groupby('sex').survived.mean().plot(kind='barh').set_xlabel('% survive')
```

Feature Engineering for the Model

Estimators use a system called [feature columns](https://www.tensorflow.org/guide/feature_columns) (https://www.tensorflow.org/guide/feature_columns) to describe how the model should interpret each of the raw input features. An Estimator expects a vector of numeric inputs, and *feature columns* describe how the model should convert each feature.

Selecting and crafting the right set of feature columns is key to learning an effective model. A feature column can be either one of the raw inputs in the original features `dict` (a *base feature column*), or any new columns created using transformations defined over one or multiple base columns (a *derived feature columns*).

The linear estimator uses both numeric and categorical features. Feature columns work with all TensorFlow estimators and their purpose is to define the features used for modeling. Additionally, they provide some feature engineering capabilities like one-hot-encoding, normalization, and bucketization.

Base Feature Columns

```
In [ ]:
```

```
CATEGORICAL_COLUMNS = ['sex', 'n_siblings_spouses', 'parch', 'class', 'deck',
                       'embark_town', 'alone']
NUMERIC_COLUMNS = ['age', 'fare']

feature_columns = []
for feature_name in CATEGORICAL_COLUMNS:
    vocabulary = dftrain[feature_name].unique()
    feature_columns.append(tf.feature_column.categorical_column_with_vocabulary_list(feature_name, vocabulary))

for feature_name in NUMERIC_COLUMNS:
    feature_columns.append(tf.feature_column.numeric_column(feature_name, dtype=tf.float32))
```

The `input_function` specifies how data is converted to a `tf.data.Dataset` that feeds the input pipeline in a streaming fashion. `tf.data.Dataset` can take in multiple sources such as a dataframe, a csv-formatted file, and more.

```
In [ ]:
```

```
def make_input_fn(data_df, label_df, num_epochs=10, shuffle=True, batch_size=32):
    def input_function():
        ds = tf.data.Dataset.from_tensor_slices((dict(data_df), label_df))
        if shuffle:
            ds = ds.shuffle(1000)
        ds = ds.batch(batch_size).repeat(num_epochs)
        return ds
    return input_function

train_input_fn = make_input_fn(dftrain, y_train)
eval_input_fn = make_input_fn(dfeval, y_eval, num_epochs=1, shuffle=False)
```

You can inspect the dataset:

In []:

```
ds = make_input_fn(dftrain, y_train, batch_size=10)()
for feature_batch, label_batch in ds.take(1):
    print('Some feature keys:', list(feature_batch.keys()))
    print()
    print('A batch of class:', feature_batch['class'].numpy())
    print()
    print('A batch of Labels:', label_batch.numpy())
```

You can also inspect the result of a specific feature column using the `tf.keras.layers.DenseFeatures` layer:

In []:

```
age_column = feature_columns[7]
tf.keras.layers.DenseFeatures([age_column])(feature_batch).numpy()
```

`DenseFeatures` only accepts dense tensors, to inspect a categorical column you need to transform that to a indicator column first:

In []:

```
gender_column = feature_columns[0]
tf.keras.layers.DenseFeatures([tf.feature_column.indicator_column(gender_column)])(feature_batch).numpy()
```

After adding all the base features to the model, let's train the model. Training a model is just a single command using the `tf.estimator` API:

In []:

```
linear_est = tf.estimator.LinearClassifier(feature_columns=feature_columns)
linear_est.train(train_input_fn)
result = linear_est.evaluate(eval_input_fn)

clear_output()
print(result)
```

Derived Feature Columns

Now you reached an accuracy of 75%. Using each base feature column separately may not be enough to explain the data. For example, the correlation between age and the label may be different for different gender. Therefore, if you only learn a single model weight for `gender="Male"` and `gender="Female"`, you won't capture every age-gender combination (e.g. distinguishing between `gender="Male"` AND `age="30"` AND `gender="Male"` AND `age="40"`).

To learn the differences between different feature combinations, you can add *crossed feature columns* to the model (you can also bucketize age column before the cross column):

In []:

```
age_x_gender = tf.feature_column.crossed_column(['age', 'sex'], hash_bucket_size=100)
```

After adding the combination feature to the model, let's train the model again:

In []:

```
derived_feature_columns = [age_x_gender]
linear_est = tf.estimator.LinearClassifier(feature_columns=feature_columns+derived_feature_columns)
linear_est.train(train_input_fn)
result = linear_est.evaluate(eval_input_fn)

clear_output()
print(result)
```

It now achieves an accuracy of 77.6%, which is slightly better than only trained in base features. You can try using more features and transformations to see if you can do better!

Now you can use the train model to make predictions on a passenger from the evaluation set. TensorFlow models are optimized to make predictions on a batch, or collection, of examples at once. Earlier, the `eval_input_fn` was defined using the entire evaluation set.

In []:

```
pred_dicts = list(linear_est.predict(eval_input_fn))
probs = pd.Series([pred['probabilities'][1] for pred in pred_dicts])

probs.plot(kind='hist', bins=20, title='predicted probabilities')
```

Finally, look at the receiver operating characteristic (ROC) of the results, which will give us a better idea of the tradeoff between the true positive rate and false positive rate.

In []:

```
from sklearn.metrics import roc_curve
from matplotlib import pyplot as plt

fpr, tpr, _ = roc_curve(y_eval, probs)
plt.plot(fpr, tpr)
plt.title('ROC curve')
plt.xlabel('false positive rate')
plt.ylabel('true positive rate')
plt.xlim(0,)
plt.ylim(0,)
```

Copyright 2019 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Premade Estimators



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/estimator/premade)

(<https://www.tensorflow.org/tutorials/estimator/premade>)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/estimator/premade.ipynb)

(<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/estimator/premade.ipynb>) (<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/estimator/premade.ipynb>)

Warning: Estimators are not recommended for new code. Estimators run v1.Session -style code which is more difficult to write correctly, and can behave unexpectedly, especially when combined with TF 2 code. Estimators do fall under [compatibility guarantees](https://tensorflow.org/guide/compatibility) (<https://tensorflow.org/guide/compatibility>), but will receive no fixes other than security vulnerabilities. See the [migration guide](https://tensorflow.org/guide/migrate) (<https://tensorflow.org/guide/migrate>) for details.

This tutorial shows you how to solve the Iris classification problem in TensorFlow using Estimators. An Estimator is a legacy TensorFlow high-level representation of a complete model. For more details see [Estimators](https://www.tensorflow.org/guide/estimator) (<https://www.tensorflow.org/guide/estimator>).

Note: In TensorFlow 2.0, the [Keras API](https://www.tensorflow.org/guide/keras) (<https://www.tensorflow.org/guide/keras>) can accomplish these same tasks, and is believed to be an easier API to learn. If you are starting fresh, it is recommended you start with Keras.

First things first

In order to get started, you will first import TensorFlow and a number of libraries you will need.

In []:

```
import tensorflow as tf
import pandas as pd
```

The data set

The sample program in this document builds and tests a model that classifies Iris flowers into three different species based on the size of their [sepals](https://en.wikipedia.org/wiki/Sepal) (<https://en.wikipedia.org/wiki/Sepal>) and [petals](https://en.wikipedia.org/wiki/Petal) (<https://en.wikipedia.org/wiki/Petal>).

You will train a model using the Iris data set. The Iris data set contains four features and one [label](https://developers.google.com/machine-learning/glossary/#label) (<https://developers.google.com/machine-learning/glossary/#label>). The four features identify the following botanical characteristics of individual Iris flowers:

- sepal length
- sepal width
- petal length
- petal width

Based on this information, you can define a few helpful constants for parsing the data:

In []:

```
CSV_COLUMN_NAMES = ['SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth', 'Species']
SPECIES = ['Setosa', 'Versicolor', 'Virginica']
```

Next, download and parse the Iris data set using Keras and Pandas. Note that you keep distinct datasets for training and testing.

In []:

```
train_path = tf.keras.utils.get_file(
    "iris_training.csv", "https://storage.googleapis.com/download.tensorflow.org/data/iris_training.csv")
test_path = tf.keras.utils.get_file(
    "iris_test.csv", "https://storage.googleapis.com/download.tensorflow.org/data/iris_test.csv")

train = pd.read_csv(train_path, names=CSV_COLUMN_NAMES, header=0)
test = pd.read_csv(test_path, names=CSV_COLUMN_NAMES, header=0)
```

You can inspect your data to see that you have four float feature columns and one int32 label.

In []:

```
train.head()
```

For each of the datasets, split out the labels, which the model will be trained to predict.

In []:

```
train_y = train.pop('Species')
test_y = test.pop('Species')

# The label column has now been removed from the features.
train.head()
```

Overview of programming with Estimators

Now that you have the data set up, you can define a model using a TensorFlow Estimator. An Estimator is any class derived from `tf.estimator.Estimator`. TensorFlow provides a collection of `tf.estimator` (for example, `LinearRegressor`) to implement common ML algorithms. Beyond those, you may write your own [custom Estimators](https://www.tensorflow.org/guide/estimator#custom_estimators) (https://www.tensorflow.org/guide/estimator#custom_estimators). It is recommended using pre-made Estimators when just getting started.

To write a TensorFlow program based on pre-made Estimators, you must perform the following tasks:

- Create one or more input functions.
- Define the model's feature columns.
- Instantiate an Estimator, specifying the feature columns and various hyperparameters.
- Call one or more methods on the Estimator object, passing the appropriate input function as the source of the data.

Let's see how those tasks are implemented for Iris classification.

Create input functions

You must create input functions to supply data for training, evaluating, and prediction.

An **input function** is a function that returns a `tf.data.Dataset` object which outputs the following two-element tuple:

- **features** (<https://developers.google.com/machine-learning/glossary/#feature>) - A Python dictionary in which:
 - Each key is the name of a feature.
 - Each value is an array containing all of that feature's values.
- **label** - An array containing the values of the [label](https://developers.google.com/machine-learning/glossary/#label) (<https://developers.google.com/machine-learning/glossary/#label>) for every example.

Just to demonstrate the format of the input function, here's a simple implementation:

In []:

```
def input_evaluation_set():
    features = {'SepalLength': np.array([6.4, 5.0]),
                'SepalWidth': np.array([2.8, 2.3]),
                'PetalLength': np.array([5.6, 3.3]),
                'PetalWidth': np.array([2.2, 1.0])}
    labels = np.array([2, 1])
    return features, labels
```

Your input function may generate the `features` dictionary and `label` list any way you like. However, It is recommended using TensorFlow's [Dataset API](#) (<https://www.tensorflow.org/guide/datasets>), which can parse all sorts of data.

The Dataset API can handle a lot of common cases for you. For example, using the Dataset API, you can easily read in records from a large collection of files in parallel and join them into a single stream.

To keep things simple in this example you are going to load the data with [pandas](#) (<https://pandas.pydata.org/>), and build an input pipeline from this in-memory data:

In []:

```
def input_fn(features, labels, training=True, batch_size=256):
    """An input function for training or evaluating"""
    # Convert the inputs to a Dataset.
    dataset = tf.data.Dataset.from_tensor_slices((dict(features), labels))

    # Shuffle and repeat if you are in training mode.
    if training:
        dataset = dataset.shuffle(1000).repeat()

    return dataset.batch(batch_size)
```

Define the feature columns

A [feature column](#) (https://developers.google.com/machine-learning/glossary/#feature_columns) is an object describing how the model should use raw input data from the features dictionary. When you build an Estimator model, you pass it a list of feature columns that describes each of the features you want the model to use. The `tf.feature_column` module provides many options for representing data to the model.

For Iris, the 4 raw features are numeric values, so you'll build a list of feature columns to tell the Estimator model to represent each of the four features as 32-bit floating-point values. Therefore, the code to create the feature column is:

In []:

```
# Feature columns describe how to use the input.
my_feature_columns = []
for key in train.keys():
    my_feature_columns.append(tf.feature_column.numeric_column(key=key))
```

Feature columns can be far more sophisticated than those shown here. You can read more about Feature Columns in [this guide](#) (https://www.tensorflow.org/guide/feature_columns).

Now that you have the description of how you want the model to represent the raw features, you can build the estimator.

Instantiate an estimator

The Iris problem is a classic classification problem. Fortunately, TensorFlow provides several pre-made classifier Estimators, including:

- `tf.estimator.DNNClassifier` for deep models that perform multi-class classification.
- `tf.estimator.DNNLinearCombinedClassifier` for wide & deep models.
- `tf.estimator.LinearClassifier` for classifiers based on linear models.

For the Iris problem, `tf.estimator.DNNClassifier` seems like the best choice. Here's how you instantiated this Estimator:

In []:

```
# Build a DNN with 2 hidden layers with 30 and 10 hidden nodes each.
classifier = tf.estimator.DNNClassifier(
    feature_columns=my_feature_columns,
    # Two hidden layers of 30 and 10 nodes respectively.
    hidden_units=[30, 10],
    # The model must choose between 3 classes.
    n_classes=3)
```

Train, Evaluate, and Predict

Now that you have an Estimator object, you can call methods to do the following:

- Train the model.
- Evaluate the trained model.
- Use the trained model to make predictions.

Train the model

Train the model by calling the Estimator's `train` method as follows:

In []:

```
# Train the Model.
classifier.train(
    input_fn=lambda: input_fn(train, train_y, training=True),
    steps=5000)
```

Note that you wrap up your `input_fn` call in a `lambda` (<https://docs.python.org/3/tutorial/controlflow.html>) to capture the arguments while providing an input function that takes no arguments, as expected by the Estimator. The `steps` argument tells the method to stop training after a number of training steps.

Evaluate the trained model

Now that the model has been trained, you can get some statistics on its performance. The following code block evaluates the accuracy of the trained model on the test data:

In []:

```
eval_result = classifier.evaluate(
    input_fn=lambda: input_fn(test, test_y, training=False))

print('\nTest set accuracy: {accuracy:0.3f}\n'.format(**eval_result))
```

Unlike the call to the `train` method, you did not pass the `steps` argument to evaluate. The `input_fn` for eval only yields a single `epoch` (<https://developers.google.com/machine-learning/glossary/#epoch>) of data.

The `eval_result` dictionary also contains the `average_loss` (mean loss per sample), the `loss` (mean loss per mini-batch) and the value of the estimator's `global_step` (the number of training iterations it underwent).

Making predictions (inferring) from the trained model

You now have a trained model that produces good evaluation results. You can now use the trained model to predict the species of an Iris flower based on some unlabeled measurements. As with training and evaluation, you make predictions using a single function call:

In []:

```
# Generate predictions from the model
expected = ['Setosa', 'Versicolor', 'Virginica']
predict_x = {
    'SepalLength': [5.1, 5.9, 6.9],
    'SepalWidth': [3.3, 3.0, 3.1],
    'PetalLength': [1.7, 4.2, 5.4],
    'PetalWidth': [0.5, 1.5, 2.1],
}

def input_fn(features, batch_size=256):
    """An input function for prediction."""
    # Convert the inputs to a Dataset without labels.
    return tf.data.Dataset.from_tensor_slices(dict(features)).batch(batch_size)

predictions = classifier.predict(
    input_fn=lambda: input_fn(predict_x))
```

The `predict` method returns a Python iterable, yielding a dictionary of prediction results for each example. The following code prints a few predictions and their probabilities:

In []:

```
for pred_dict, expec in zip(predictions, expected):
    class_id = pred_dict['class_ids'][0]
    probability = pred_dict['probabilities'][class_id]

    print('Prediction is "{}" {:.1f}%, expected "{}"'.format(
        SPECIES[class_id], 100 * probability, expec))
```

Copyright 2021 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Uncertainty-aware Deep Learning with SNGP



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/understanding/sngp)

(<https://www.tensorflow.org/tutorials/understanding/sngp>)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/understanding/sngp.ipynb)

(<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/understanding/sngp.ipynb>) (<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/understanding/sngp.ipynb>)

In AI applications that are safety-critical (e.g., medical decision making and autonomous driving) or where the data is inherently noisy (e.g., natural language understanding), it is important for a deep classifier to reliably quantify its uncertainty. The deep classifier should be able to be aware of its own limitations and when it should hand control over to the human experts. This tutorial shows how to improve a deep classifier's ability in quantifying uncertainty using a technique called **Spectral-normalized Neural Gaussian Process (SNGP (<https://arxiv.org/abs/2006.10108>)**.

The core idea of SNGP is to improve a deep classifier's ***distance awareness*** by applying simple modifications to the network. A model's *distance awareness* is a measure of how its predictive probability reflects the distance between the test example and the training data. This is a desirable property that is common for gold-standard probabilistic models (e.g., the [Gaussian process \(\[https://en.wikipedia.org/wiki/Gaussian_process\]\(https://en.wikipedia.org/wiki/Gaussian_process\)\)](https://en.wikipedia.org/wiki/Gaussian_process) with RBF kernels) but is lacking in models with deep neural networks. SNGP provides a simple way to inject this Gaussian-process behavior into a deep classifier while maintaining its predictive accuracy.

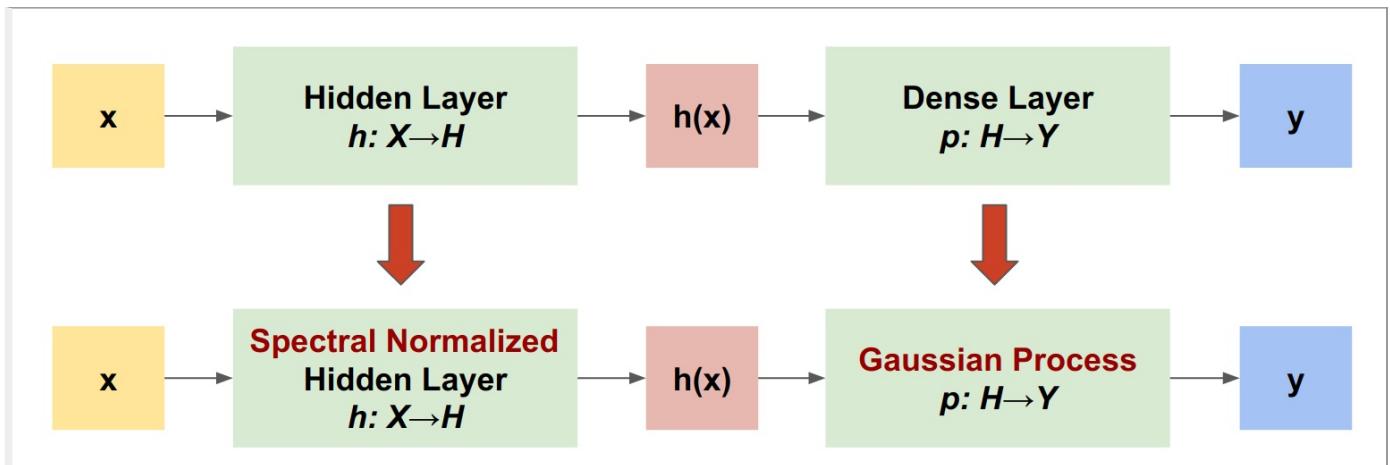
This tutorial implements a deep residual network (ResNet)-based SNGP model on the [two moons \(\[https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_moons.html\]\(https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_moons.html\)\)](https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_moons.html) dataset, and compares its uncertainty surface with that of two other popular uncertainty approaches - [Monte Carlo dropout \(<https://arxiv.org/abs/1506.02142>\)](https://arxiv.org/abs/1506.02142) and [Deep ensemble \(<https://arxiv.org/abs/1612.01474>\)](https://arxiv.org/abs/1612.01474).

This tutorial illustrates the SNGP model on a toy 2D dataset. For an example of applying SNGP to a real-world natural language understanding task using BERT-base, check out the [SNGP-BERT tutorial \(\[https://www.tensorflow.org/text/tutorials/uncertainty_quantification_with_sngp_bert\]\(https://www.tensorflow.org/text/tutorials/uncertainty_quantification_with_sngp_bert\)\)](https://www.tensorflow.org/text/tutorials/uncertainty_quantification_with_sngp_bert). For high-quality implementations of an SNGP model (and many other uncertainty methods) on a wide variety of benchmark datasets (such as [CIFAR-100 \(<https://www.tensorflow.org/datasets/catalog/cifar100>\)](https://www.tensorflow.org/datasets/catalog/cifar100), [ImageNet \(<https://www.tensorflow.org/datasets/catalog/imagenet2012>\)](https://www.tensorflow.org/datasets/catalog/imagenet2012), [Jigsaw toxicity detection \(\[https://www.tensorflow.org/datasets/catalog/wikipedia_toxicity_subtypes\]\(https://www.tensorflow.org/datasets/catalog/wikipedia_toxicity_subtypes\)\)](https://www.tensorflow.org/datasets/catalog/wikipedia_toxicity_subtypes), etc), refer to the [Uncertainty Baselines \(<https://github.com/google/uncertainty-baselines>\)](https://github.com/google/uncertainty-baselines) benchmark.

About SNGP

[Spectral-normalized Neural Gaussian Process \(<https://arxiv.org/abs/2006.10108>\)](https://arxiv.org/abs/2006.10108) (SNGP) is a simple approach to improve a deep classifier's uncertainty quality while maintaining a similar level of accuracy and latency. Given a deep residual network, SNGP makes two simple changes to the model:

- It applies spectral normalization to the hidden residual layers.
- It replaces the Dense output layer with a Gaussian process layer.



Compared to other uncertainty approaches (e.g., Monte Carlo dropout or Deep ensemble), SNGP has several advantages:

- It works for a wide range of state-of-the-art residual-based architectures (e.g., (Wide) ResNet, DenseNet, BERT, etc).
- It is a single-model method (i.e., does not rely on ensemble averaging). Therefore SNGP has a similar level of latency as a single deterministic network, and can be scaled easily to large datasets like [ImageNet \(<https://github.com/google/uncertainty-baselines/tree/main/baselines/imagenet>\)](https://github.com/google/uncertainty-baselines/tree/main/baselines/imagenet) and [Jigsaw Toxic Comments classification \(\[https://github.com/google/uncertainty-baselines/tree/main/baselines/toxic_comments\]\(https://github.com/google/uncertainty-baselines/tree/main/baselines/toxic_comments\)\)](https://github.com/google/uncertainty-baselines/tree/main/baselines/toxic_comments).
- It has strong out-of-domain detection performance due to the *distance-awareness* property.

The downsides of this method are:

- The predictive uncertainty of a SNGP is computed using the [Laplace approximation \(<http://www.gaussianprocess.org/gpml/chapters/RW3.pdf>\)](http://www.gaussianprocess.org/gpml/chapters/RW3.pdf). Therefore theoretically, the posterior uncertainty of SNGP is different from that of an exact Gaussian process.
- SNGP training needs a covariance reset step at the begining of a new epoch. This can add a tiny amount of extra complexity to a training pipeline. This tutorial shows a simple way to implement this using Keras callbacks.

Setup

In []:

```
!pip install --use-deprecated=legacy-resolver tf-models-official
```

In []:

```
# refresh pkg_resources so it takes the changes into account.
import pkg_resources
import importlib
importlib.reload(pkg_resources)
```

In []:

```
import matplotlib.pyplot as plt
import matplotlib.colors as colors

import sklearn.datasets

import numpy as np
import tensorflow as tf

import official.nlp.modeling.layers as nlp_layers
```

Define visualization macros

In []:

```
plt.rcParams['figure.dpi'] = 140

DEFAULT_X_RANGE = (-3.5, 3.5)
DEFAULT_Y_RANGE = (-2.5, 2.5)
DEFAULT_CMAP = colors.ListedColormap(['#377eb8', "#ff7f00"])
DEFAULT_NORM = colors.Normalize(vmin=0, vmax=1, )
DEFAULT_N_GRID = 100
```

The two moon dataset

Create the training and evaluation datasets from the [two moon dataset \(\[https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_moons.html\]\(https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_moons.html\)\)](https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_moons.html).

In []:

```
def make_training_data(sample_size=500):
    """Create two moon training dataset."""
    train_examples, train_labels = sklearn.datasets.make_moons(
        n_samples=2 * sample_size, noise=0.1)

    # Adjust data position slightly.
    train_examples[train_labels == 0] += [-0.1, 0.2]
    train_examples[train_labels == 1] += [0.1, -0.2]

    return train_examples, train_labels
```

Evaluate the model's predictive behavior over the entire 2D input space.

In []:

```
def make_testing_data(x_range=DEFAULT_X_RANGE, y_range=DEFAULT_Y_RANGE, n_grid=DEFAULT_N_GRID):
    """Create a mesh grid in 2D space."""
    # testing data (mesh grid over data space)
    x = np.linspace(x_range[0], x_range[1], n_grid)
    y = np.linspace(y_range[0], y_range[1], n_grid)
    xv, yv = np.meshgrid(x, y)
    return np.stack([xv.flatten(), yv.flatten()], axis=-1)
```

To evaluate model uncertainty, add an out-of-domain (OOD) dataset that belongs to a third class. The model never sees these OOD examples during training.

In []:

```
def make_ood_data(sample_size=500, means=(2.5, -1.75), vars=(0.01, 0.01)):
    return np.random.multivariate_normal(
        means, cov=np.diag(vars), size=sample_size)
```

In []:

```
# Load the train, test and OOD datasets.
train_examples, train_labels = make_training_data(
    sample_size=500)
test_examples = make_testing_data()
ood_examples = make_ood_data(sample_size=500)

# Visualize
pos_examples = train_examples[train_labels == 0]
neg_examples = train_examples[train_labels == 1]

plt.figure(figsize=(7, 5.5))

plt.scatter(pos_examples[:, 0], pos_examples[:, 1], c="#377eb8", alpha=0.5)
plt.scatter(neg_examples[:, 0], neg_examples[:, 1], c="#ff7f00", alpha=0.5)
plt.scatter(ood_examples[:, 0], ood_examples[:, 1], c="red", alpha=0.1)

plt.legend(["Positive", "Negative", "Out-of-Domain"])

plt.ylim(DEFAULT_Y_RANGE)
plt.xlim(DEFAULT_X_RANGE)

plt.show()
```

Here the blue and orange represent the positive and negative classes, and the red represents the OOD data. A model that quantifies the uncertainty well is expected to be confident when close to training data (i.e., $p(x_{test})$ close to 0 or 1), and be uncertain when far away from the training data regions (i.e., $p(x_{test})$ close to 0.5).

The deterministic model

Define model

Start from the (baseline) deterministic model: a multi-layer residual network (ResNet) with dropout regularization.

In []:

```
#@title
class DeepResNet(tf.keras.Model):
    """Defines a multi-layer residual network."""
    def __init__(self, num_classes, num_layers=3, num_hidden=128,
                 dropout_rate=0.1, **classifier_kwargs):
        super().__init__()
        # Defines class meta data.
        self.num_hidden = num_hidden
        self.num_layers = num_layers
        self.dropout_rate = dropout_rate
        self.classifier_kwargs = classifier_kwargs

        # Defines the hidden layers.
        self.input_layer = tf.keras.layers.Dense(self.num_hidden, trainable=False)
        self.dense_layers = [self.make_dense_layer() for _ in range(num_layers)]

        # Defines the output layer.
        self.classifier = self.make_output_layer(num_classes)

    def call(self, inputs):
        # Projects the 2d input data to high dimension.
        hidden = self.input_layer(inputs)

        # Computes the resnet hidden representations.
        for i in range(self.num_layers):
            resid = self.dense_layers[i](hidden)
            resid = tf.keras.layers.Dropout(self.dropout_rate)(resid)
            hidden += resid

        return self.classifier(hidden)

    def make_dense_layer(self):
        """Uses the Dense layer as the hidden layer."""
        return tf.keras.layers.Dense(self.num_hidden, activation="relu")

    def make_output_layer(self, num_classes):
        """Uses the Dense layer as the output layer."""
        return tf.keras.layers.Dense(
            num_classes, **self.classifier_kwargs)
```

This tutorial uses a 6-layer ResNet with 128 hidden units.

In []:

```
resnet_config = dict(num_classes=2, num_layers=6, num_hidden=128)
```

In []:

```
resnet_model = DeepResNet(**resnet_config)
```

In []:

```
resnet_model.build((None, 2))  
resnet_model.summary()
```

Train model

Configure the training parameters to use `SparseCategoricalCrossentropy` as the loss function and the Adam optimizer.

In []:

```
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)  
metrics = tf.keras.metrics.SparseCategoricalAccuracy(),  
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-4)  
  
train_config = dict(loss=loss, metrics=metrics, optimizer=optimizer)
```

Train the model for 100 epochs with batch size 128.

In []:

```
fit_config = dict(batch_size=128, epochs=100)
```

In []:

```
resnet_model.compile(**train_config)  
resnet_model.fit(train_examples, train_labels, **fit_config)
```

Visualize uncertainty

In []:

```
#@title
def plot_uncertainty_surface(test_uncertainty, ax, cmap=None):
    """Visualizes the 2D uncertainty surface.

For simplicity, assume these objects already exist in the memory:

test_examples: Array of test examples, shape (num_test, 2).
train_labels: Array of train labels, shape (num_train,).
train_examples: Array of train examples, shape (num_train, 2).

Arguments:
test_uncertainty: Array of uncertainty scores, shape (num_test,).
ax: A matplotlib Axes object that specifies a matplotlib figure.
cmap: A matplotlib colormap object specifying the palette of the
predictive surface.

Returns:
pcm: A matplotlib PathCollection object that contains the palette
information of the uncertainty plot.

# Normalize uncertainty for better visualization.
test_uncertainty = test_uncertainty / np.max(test_uncertainty)

# Set view limits.
ax.set_xlim(DEFAULT_X_RANGE)
ax.set_ylim(DEFAULT_Y_RANGE)

# Plot normalized uncertainty surface.
pcm = ax.imshow(
    np.reshape(test_uncertainty, [DEFAULT_N_GRID, DEFAULT_N_GRID]),
    cmap=cmap,
    origin="lower",
    extent=DEFAULT_X_RANGE + DEFAULT_Y_RANGE,
    vmin=DEFAULT_NORM.vmin,
    vmax=DEFAULT_NORM.vmax,
    interpolation='bicubic',
    aspect='auto')

# Plot training data.
ax.scatter(train_examples[:, 0], train_examples[:, 1],
           c=train_labels, cmap=DEFAULT_CMAP, alpha=0.5)
ax.scatter(ood_examples[:, 0], ood_examples[:, 1], c="red", alpha=0.1)

return pcm
```

Now visualize the predictions of the deterministic model. First plot the class probability:

$$p(x) = \text{softmax}(\text{logit}(x))$$

In []:

```
resnet_logits = resnet_model(test_examples)
resnet_probs = tf.nn.softmax(resnet_logits, axis=-1)[:, 0] # Take the probability for class 0.
```

In []:

```
_, ax = plt.subplots(figsize=(7, 5.5))
pcm = plot_uncertainty_surface(resnet_probs, ax=ax)
plt.colorbar(pcm, ax=ax)
plt.title("Class Probability, Deterministic Model")
plt.show()
```

In this plot, the yellow and purple are the predictive probabilities for the two classes. The deterministic model did a good job in classifying the two known classes (blue and orange) with a nonlinear decision boundary. However, it is not **distance-aware**, and classified the never-seen red out-of-domain (OOD) examples confidently as the orange class.

Visualize the model uncertainty by computing the [predictive variance \(\[https://en.wikipedia.org/wiki/Bernoulli_distribution#Variance\]\(https://en.wikipedia.org/wiki/Bernoulli_distribution#Variance\)\)](https://en.wikipedia.org/wiki/Bernoulli_distribution#Variance):

$$\text{var}(x) = p(x) * (1 - p(x))$$

In []:

```
resnet_uncertainty = resnet_probs * (1 - resnet_probs)
```

In []:

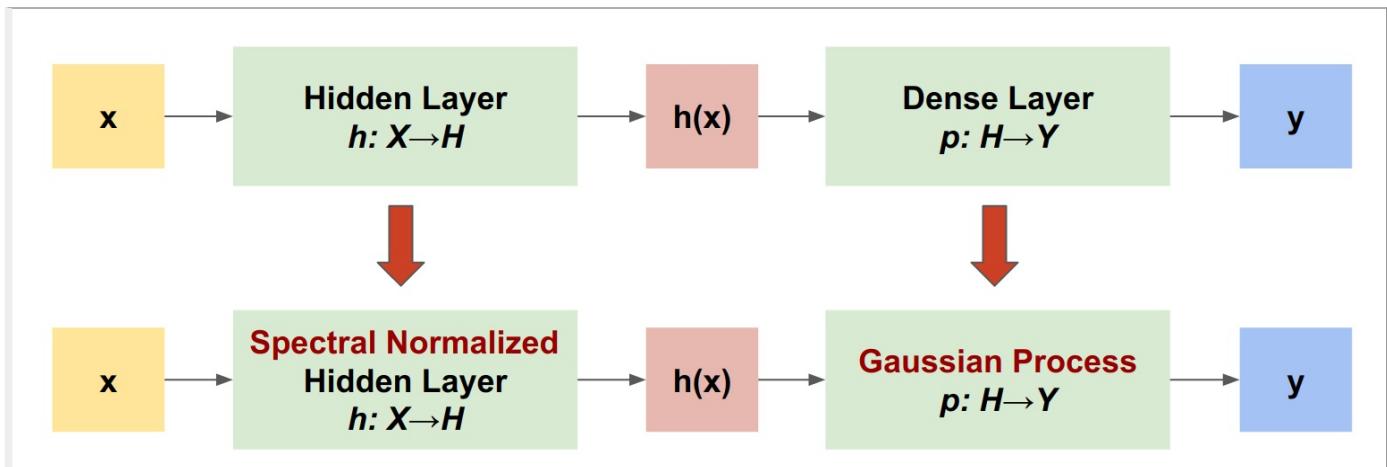
```
_, ax = plt.subplots(figsize=(7, 5.5))
pcm = plot_uncertainty_surface(resnet_uncertainty, ax=ax)
plt.colorbar(pcm, ax=ax)
plt.title("Predictive Uncertainty, Deterministic Model")
plt.show()
```

In this plot, the yellow indicates high uncertainty, and the purple indicates low uncertainty. A deterministic ResNet's uncertainty depends only on the test examples' distance from the decision boundary. This leads the model to be over-confident when out of the training domain. The next section shows how SNGP behaves differently on this dataset.

The SNGP model

Define SNGP model

Let's now implement the SNGP model. Both the SNGP components, `SpectralNormalization` and `RandomFeatureGaussianProcess`, are available at the tensorflow_model's [built-in layers](https://github.com/tensorflow/models/tree/master/official/nlp/modeling/layers) (<https://github.com/tensorflow/models/tree/master/official/nlp/modeling/layers>).



Let's look at these two components in more detail. (You can also jump to the [The SNGP model](#) section to see how the full model is implemented.)

Spectral Normalization wrapper

`SpectralNormalization` (https://github.com/tensorflow/models/blob/master/official/nlp/modeling/layers/spectral_normalization.py) is a Keras layer wrapper. It can be applied to an existing Dense layer like this:

In []:

```
dense = tf.keras.layers.Dense(units=10)
dense = nlp_layers.SpectralNormalization(dense, norm_multiplier=0.9)
```

Spectral normalization regularizes the hidden weight W by gradually guiding its spectral norm (i.e., the largest eigenvalue of W) toward the target value `norm_multiplier`.

Note: Usually it is preferable to set `norm_multiplier` to a value smaller than 1. However in practice, it can be also relaxed to a larger value to ensure the deep network has enough expressive power.

The Gaussian Process (GP) layer

`RandomFeatureGaussianProcess` (https://github.com/tensorflow/models/blob/master/official/nlp/modeling/layers/gaussian_process.py) implements a [random-feature based approximation](https://people.eecs.berkeley.edu/~brecht/papers/07.rah.rec.nips.pdf) (<https://people.eecs.berkeley.edu/~brecht/papers/07.rah.rec.nips.pdf>) to a Gaussian process model that is end-to-end trainable with a deep neural network. Under the hood, the Gaussian process layer implements a two-layer network:

$$\text{logits}(x) = \Phi(x)\beta, \quad \Phi(x) = \sqrt{\frac{2}{M}} * \cos(Wx + b)$$

Here x is the input, and W and b are frozen weights initialized randomly from Gaussian and uniform distributions, respectively. (Therefore $\Phi(x)$ are called "random features".) β is the learnable kernel weight similar to that of a Dense layer.

In []:

```
batch_size = 32
input_dim = 1024
num_classes = 10
```

In []:

```
gp_layer = nlp_layers.RandomFeatureGaussianProcess(units=num_classes,
                                                    num_inducing=1024,
                                                    normalize_input=False,
                                                    scale_random_features=True,
                                                    gp_cov_momentum=-1)
```

The main parameters of the GP layers are:

- `units` : The dimension of the output logits.
- `num_inducing` : The dimension M of the hidden weight W . Default to 1024.
- `normalize_input` : Whether to apply layer normalization to the input x .
- `scale_random_features` : Whether to apply the scale $\sqrt{2/M}$ to the hidden output.

Note: For a deep neural network that is sensitive to the learning rate (e.g., ResNet-50 and ResNet-110), it is generally recommended to set `normalize_input=True` to stabilize training, and set `scale_random_features=False` to avoid the learning rate from being modified in unexpected ways when passing through the GP layer.

- `gp_cov_momentum` controls how the model covariance is computed. If set to a positive value (e.g., 0.999), the covariance matrix is computed using the momentum-based moving average update (similar to batch normalization). If set to -1, the covariance matrix is updated without momentum.

Note: The momentum-based update method can be sensitive to batch size. Therefore it is generally recommended to set `gp_cov_momentum=-1` to compute the covariance exactly. For this to work properly, the covariance matrix estimator needs to be reset at the beginning of a new epoch in order to avoid counting the same data twice. For `RandomFeatureGaussianProcess`, this can be done by calling its `reset_covariance_matrix()`. The next section shows an easy implementation of this using Keras' built-in API.

Given a batch input with shape `(batch_size, input_dim)`, the GP layer returns a `logits` tensor (shape `(batch_size, num_classes)`) for prediction, and also `covmat` tensor (shape `(batch_size, batch_size)`) which is the posterior covariance matrix of the batch logits.

In []:

```
embedding = tf.random.normal(shape=(batch_size, input_dim))
logits, covmat = gp_layer(embedding)
```

Note: Notice that under this implementation of the SNGP model, the predictive logits $\text{logit}(x_{test})$ for all classes share the same covariance matrix $\text{var}(x_{test})$, which describes the distance between x_{test} from the training data.

Theoretically, it is possible to extend the algorithm to compute different variance values for different classes (as introduced in the [original SNGP paper](https://arxiv.org/abs/2006.10108) (<https://arxiv.org/abs/2006.10108>)). However, this is difficult to scale to problems with large output spaces (e.g., ImageNet or language modeling).

The full SNGP model

Given the base class `DeepResNet`, the SNGP model can be implemented easily by modifying the residual network's hidden and output layers. For compatibility with Keras `model.fit()` API, also modify the model's `call()` method so it only outputs `logits` during training.

In []:

```
class DeepResNetSNGP(DeepResNet):
    def __init__(self, spec_norm_bound=0.9, **kwargs):
        self.spec_norm_bound = spec_norm_bound
        super().__init__(**kwargs)

    def make_dense_layer(self):
        """Applies spectral normalization to the hidden layer."""
        dense_layer = super().make_dense_layer()
        return nlp_layers.SpectralNormalization(
            dense_layer, norm_multiplier=self.spec_norm_bound)

    def make_output_layer(self, num_classes):
        """Uses Gaussian process as the output layer."""
        return nlp_layers.RandomFeatureGaussianProcess(
            num_classes,
            gp_cov_momentum=-1,
            **self.classifier_kwargs)

    def call(self, inputs, training=False, return_covmat=False):
        # Gets logits and covariance matrix from GP layer.
        logits, covmat = super().call(inputs)

        # Returns only logits during training.
        if not training and return_covmat:
            return logits, covmat

        return logits
```

Use the same architecture as the deterministic model.

In []:

```
resnet_config
```

In []:

```
sngp_model = DeepResNetSNGP(**resnet_config)
```

In []:

```
sngp_model.build((None, 2))
sngp_model.summary()
```

Implement a Keras callback to reset the covariance matrix at the beginning of a new epoch.

In []:

```
class ResetCovarianceCallback(tf.keras.callbacks.Callback):

    def on_epoch_begin(self, epoch, logs=None):
        """Resets covariance matrix at the begining of the epoch."""
        if epoch > 0:
            self.model.classifier.reset_covariance_matrix()
```

Add this callback to the DeepResNetSNGP model class.

In []:

```
class DeepResNetSNGPWithCovReset(DeepResNetSNGP):
    def fit(self, *args, **kwargs):
        """Adds ResetCovarianceCallback to model callbacks."""
        kwargs["callbacks"] = list(kwargs.get("callbacks", []))
        kwargs["callbacks"].append(ResetCovarianceCallback())

    return super().fit(*args, **kwargs)
```

Train model

Use `tf.keras.model.fit` to train the model.

In []:

```
sngp_model = DeepResNetSNGPWithCovReset(**resnet_config)
sngp_model.compile(**train_config)
sngp_model.fit(train_examples, train_labels, **fit_config)
```

Visualize uncertainty

First compute the predictive logits and variances.

In []:

```
sngp_logits, sngp_covmat = sngp_model(test_examples, return_covmat=True)
```

In []:

```
sngp_variance = tf.linalg.diag_part(sngp_covmat)[:, None]
```

Now compute the posterior predictive probability. The classic method for computing the predictive probability of a probabilistic model is to use Monte Carlo sampling, i.e.,

$$E(p(x)) = \frac{1}{M} \sum_{m=1}^M \text{logit}_m(x),$$

where M is the sample size, and $\text{logit}_m(x)$ are random samples from the SNGP posterior $\text{MultivariateNormal}(\text{sngp_logits}, \text{sngp_covmat})$. However, this approach can be slow for latency-sensitive applications such as autonomous driving or real-time bidding. Instead, can approximate $E(p(x))$ using the [mean-field method](https://arxiv.org/abs/2006.07584) (<https://arxiv.org/abs/2006.07584>):

$$E(p(x)) \approx \text{softmax}\left(\frac{\text{logit}(x)}{\sqrt{1 + \lambda * \sigma^2(x)}}\right)$$

where $\sigma^2(x)$ is the SNGP variance, and λ is often chosen as $\pi/8$ or $3/\pi^2$.

In []:

```
sngp_logits_adjusted = sngp_logits / tf.sqrt(1. + (np.pi / 8.) * sngp_variance)
sngp_probs = tf.nn.softmax(sngp_logits_adjusted, axis=-1)[:, 0]
```

Note: Instead of fixing λ to a fixed value, you can also treat it as a hyperparameter, and tune it to optimize the model's calibration performance. This is known as [temperature scaling](http://proceedings.mlr.press/v70/guo17a.html) (<http://proceedings.mlr.press/v70/guo17a.html>) in the deep learning uncertainty literature.

This mean-field method is implemented as a built-in function `layers.gaussian_process.mean_field_logits`:

In []:

```
def compute_posterior_mean_probability(logits, covmat, lambda_param=np.pi / 8.):
    # Computes uncertainty-adjusted logits using the built-in method.
    logits_adjusted = nlp_layers.gaussian_process.mean_field_logits(
        logits, covmat, mean_field_factor=lambda_param)

    return tf.nn.softmax(logits_adjusted, axis=-1)[:, 0]
```

In []:

```
sngp_logits, sngp_covmat = sngp_model(test_examples, return_covmat=True)
sngp_probs = compute_posterior_mean_probability(sngp_logits, sngp_covmat)
```

SNGP Summary

```
In [ ]:
```

```
#@title

def plot_predictions(pred_probs, model_name=""):
    """Plot normalized class probabilities and predictive uncertainties."""
    # Compute predictive uncertainty.
    uncertainty = pred_probs * (1. - pred_probs)

    # Initialize the plot axes.
    fig, axs = plt.subplots(1, 2, figsize=(14, 5))

    # Plots the class probability.
    pcm_0 = plot_uncertainty_surface(pred_probs, ax=axs[0])
    # Plots the predictive uncertainty.
    pcm_1 = plot_uncertainty_surface(uncertainty, ax=axs[1])

    # Adds color bars and titles.
    fig.colorbar(pcm_0, ax=axs[0])
    fig.colorbar(pcm_1, ax=axs[1])

    axs[0].set_title(f"Class Probability, {model_name}")
    axs[1].set_title(f"(Normalized) Predictive Uncertainty, {model_name}")

    plt.show()
```

Put everything together. The entire procedure (training, evaluation and uncertainty computation) can be done in just five lines:

```
In [ ]:
```

```
def train_and_test_sngp(train_examples, test_examples):
    sngp_model = DeepResNetSNGPWithCovReset(**resnet_config)

    sngp_model.compile(**train_config)
    sngp_model.fit(train_examples, train_labels, verbose=0, **fit_config)

    sngp_logits, sngp_covmat = sngp_model(test_examples, return_covmat=True)
    sngp_probs = compute_posterior_mean_probability(sngp_logits, sngp_covmat)

    return sngp_probs
```

```
In [ ]:
```

```
sngp_probs = train_and_test_sngp(train_examples, test_examples)
```

Visualize the class probability (left) and the predictive uncertainty (right) of the SNGP model.

```
In [ ]:
```

```
plot_predictions(sngp_probs, model_name="SNGP")
```

Remember that in the class probability plot (left), the yellow and purple are class probabilities. When close to the training data domain, SNGP correctly classifies the examples with high confidence (i.e., assigning near 0 or 1 probability). When far away from the training data, SNGP gradually becomes less confident, and its predictive probability becomes close to 0.5 while the (normalized) model uncertainty rises to 1.

Compare this to the uncertainty surface of the deterministic model:

```
In [ ]:
```

```
plot_predictions(resnet_probs, model_name="Deterministic")
```

Like mentioned earlier, a deterministic model is not *distance-aware*. Its uncertainty is defined by the distance of the test example from the decision boundary. This leads the model to produce overconfident predictions for the out-of-domain examples (red).

Comparison with other uncertainty approaches

This section compares the uncertainty of SNGP with [Monte Carlo dropout](https://arxiv.org/abs/1506.02142) (<https://arxiv.org/abs/1506.02142>) and [Deep ensemble](https://arxiv.org/abs/1612.01474) (<https://arxiv.org/abs/1612.01474>).

Both of these methods are based on Monte Carlo averaging of multiple forward passes of deterministic models. First set the ensemble size M .

```
In [ ]:
```

```
num_ensemble = 10
```

Monte Carlo dropout

Given a trained neural network with Dropout layers, [Monte Carlo dropout \(https://arxiv.org/abs/1506.02142\)](https://arxiv.org/abs/1506.02142) computes the mean predictive probability

$$E(p(x)) = \frac{1}{M} \sum_{m=1}^M \text{softmax}(\logit_m(x))$$

by averaging over multiple Dropout-enabled forward passes $\{\logit_m(x)\}_{m=1}^M$.

In []:

```
def mc_dropout_sampling(test_examples):
    # Enable dropout during inference.
    return resnet_model(test_examples, training=True)
```

In []:

```
# Monte Carlo dropout inference.
dropout_logit_samples = [mc_dropout_sampling(test_examples) for _ in range(num_ensemble)]
dropout_prob_samples = [tf.nn.softmax(dropout_logits, axis=-1)[:, 0] for dropout_logits in dropout_logit_samples]
dropout_probs = tf.reduce_mean(dropout_prob_samples, axis=0)
```

In []:

```
dropout_probs = tf.reduce_mean(dropout_prob_samples, axis=0)
```

In []:

```
plot_predictions(dropout_probs, model_name="MC Dropout")
```

Deep ensemble

[Deep ensemble \(https://arxiv.org/abs/1612.01474\)](https://arxiv.org/abs/1612.01474) is a state-of-the-art (but expensive) method for deep learning uncertainty. To train a Deep ensemble, first train M ensemble members.

In []:

```
# Deep ensemble training
resnet_ensemble = []
for _ in range(num_ensemble):
    resnet_model = DeepResNet(**resnet_config)
    resnet_model.compile(optimizer=optimizer, loss=loss, metrics=metrics)
    resnet_model.fit(train_examples, train_labels, verbose=0, **fit_config)

    resnet_ensemble.append(resnet_model)
```

Collect logits and compute the mean predictive probability $E(p(x)) = \frac{1}{M} \sum_{m=1}^M \text{softmax}(\logit_m(x))$.

In []:

```
# Deep ensemble inference
ensemble_logit_samples = [model(test_examples) for model in resnet_ensemble]
ensemble_prob_samples = [tf.nn.softmax(logits, axis=-1)[:, 0] for logits in ensemble_logit_samples]
ensemble_probs = tf.reduce_mean(ensemble_prob_samples, axis=0)
```

In []:

```
plot_predictions(ensemble_probs, model_name="Deep ensemble")
```

Both MC Dropout and Deep ensemble improve a model's uncertainty ability by making the decision boundary less certain. However, they both inherit the deterministic deep network's limitation in lacking distance awareness.

Summary

In this tutorial, you have:

- Implemented a SNGP model on a deep classifier to improve its distance awareness.
- Trained the SNGP model end-to-end using Keras `model.fit()` API.
- Visualized the uncertainty behavior of SNGP.
- Compared the uncertainty behavior between SNGP, Monte Carlo dropout and deep ensemble models.

Resources and further reading

- See the [SNGP-BERT tutorial](https://www.tensorflow.org/text/tutorials/uncertainty_quantification_with_sngp_bert) (https://www.tensorflow.org/text/tutorials/uncertainty_quantification_with_sngp_bert) for an example of applying SNGP on a BERT model for uncertainty-aware natural language understanding.
- See [Uncertainty Baselines](https://github.com/google/uncertainty-baselines) (<https://github.com/google/uncertainty-baselines>) for the implementation of SNGP model (and many other uncertainty methods) on a wide variety of benchmark datasets (e.g., [CIFAR](https://www.tensorflow.org/datasets/catalog/cifar100) (<https://www.tensorflow.org/datasets/catalog/cifar100>), [ImageNet](https://www.tensorflow.org/datasets/catalog/imagenet2012) (<https://www.tensorflow.org/datasets/catalog/imagenet2012>), [Jigsaw toxicity detection](https://www.tensorflow.org/datasets/catalog/wikipedia_toxicity_subtypes) (https://www.tensorflow.org/datasets/catalog/wikipedia_toxicity_subtypes), etc).
- For a deeper understanding of the SNGP method, check out the paper [Simple and Principled Uncertainty Estimation with Deterministic Deep Learning via Distance Awareness](https://arxiv.org/abs/2006.10108) (<https://arxiv.org/abs/2006.10108>).

Copyright 2020 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Playing CartPole with the Actor-Critic Method



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/reinforcement_learning/actor_critic)



https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/reinforcement_lear

This tutorial demonstrates how to implement the [Actor-Critic](https://papers.nips.cc/paper/1786-actor-critic-algorithms.pdf) (<https://papers.nips.cc/paper/1786-actor-critic-algorithms.pdf>) method using TensorFlow to train an agent on the [Open AI Gym](https://gym.openai.com/) (<https://gym.openai.com/>) CartPole-V0 environment. The reader is assumed to have some familiarity with [policy gradient methods](https://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.pdf) (<https://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.pdf>) of reinforcement learning.

Actor-Critic methods

Actor-Critic methods are [temporal difference \(TD\) learning](https://en.wikipedia.org/wiki/Temporal_difference_learning) (https://en.wikipedia.org/wiki/Temporal_difference_learning) methods that represent the policy function independent of the value function.

A policy function (or policy) returns a probability distribution over actions that the agent can take based on the given state. A value function determines the expected return for an agent starting at a given state and acting according to a particular policy forever after.

In the Actor-Critic method, the policy is referred to as the *actor* that proposes a set of possible actions given a state, and the estimated value function is referred to as the *critic*, which evaluates actions taken by the *actor* based on the given policy.

In this tutorial, both the *Actor* and *Critic* will be represented using one neural network with two outputs.

CartPole-v0

In the [CartPole-v0 environment](https://gym.openai.com/envs/CartPole-v0) (<https://gym.openai.com/envs/CartPole-v0>), a pole is attached to a cart moving along a frictionless track. The pole starts upright and the goal of the agent is to prevent it from falling over by applying a force of -1 or +1 to the cart. A reward of +1 is given for every time step the pole remains upright. An episode ends when (1) the pole is more than 15 degrees from vertical or (2) the cart moves more than 2.4 units from the center.



Trained actor-critic model in Cartpole-v0 environment

The problem is considered "solved" when the average total reward for the episode reaches 195 over 100 consecutive trials.

Setup

Import necessary packages and configure global settings.

```
In [ ]:
```

```
!pip install gym  
!pip install pyglet
```

```
In [ ]:
```

```
%%bash  
# Install additional packages for visualization  
sudo apt-get install -y xvfb python-opengl > /dev/null 2>&1  
pip install pyvirtualdisplay > /dev/null 2>&1  
pip install git+https://github.com/tensorflow/docs > /dev/null 2>&1
```

```
In [ ]:
```

```
import collections  
import gym  
import numpy as np  
import statistics  
import tensorflow as tf  
import tqdm  
  
from matplotlib import pyplot as plt  
from tensorflow.keras import layers  
from typing import Any, List, Sequence, Tuple  
  
# Create the environment  
env = gym.make("CartPole-v0")  
  
# Set seed for experiment reproducibility  
seed = 42  
env.seed(seed)  
tf.random.set_seed(seed)  
np.random.seed(seed)  
  
# Small epsilon value for stabilizing division operations  
eps = np.finfo(np.float32).eps.item()
```

Model

The *Actor* and *Critic* will be modeled using one neural network that generates the action probabilities and critic value respectively. This tutorial uses model subclassing to define the model.

During the forward pass, the model will take in the state as the input and will output both action probabilities and critic value V , which models the state-dependent [value function](https://spinningup.openai.com/en/latest/spinningup/rl_intro.html#value-functions) (https://spinningup.openai.com/en/latest/spinningup/rl_intro.html#value-functions). The goal is to train a model that chooses actions based on a policy π that maximizes expected [return](https://spinningup.openai.com/en/latest/spinningup/rl_intro.html#reward-and-return) (https://spinningup.openai.com/en/latest/spinningup/rl_intro.html#reward-and-return).

For Cartpole-v0, there are four values representing the state: cart position, cart-velocity, pole angle and pole velocity respectively. The agent can take two actions to push the cart left (0) and right (1) respectively.

Refer to [OpenAI Gym's CartPole-v0 wiki page](http://www.derongliu.org/adp/adp-cdrom/Barto1983.pdf) (<http://www.derongliu.org/adp/adp-cdrom/Barto1983.pdf>) for more information.

```
In [ ]:
```

```
class ActorCritic(tf.keras.Model):  
    """Combined actor-critic network."""  
  
    def __init__(  
        self,  
        num_actions: int,  
        num_hidden_units: int):  
        """Initialize."""  
        super().__init__()  
  
        self.common = layers.Dense(num_hidden_units, activation="relu")  
        self.actor = layers.Dense(num_actions)  
        self.critic = layers.Dense(1)  
  
    def call(self, inputs: tf.Tensor) -> Tuple[tf.Tensor, tf.Tensor]:  
        x = self.common(inputs)  
        return self.actor(x), self.critic(x)
```

```
In [ ]:
```

```
num_actions = env.action_space.n # 2  
num_hidden_units = 128  
  
model = ActorCritic(num_actions, num_hidden_units)
```

Training

To train the agent, you will follow these steps:

1. Run the agent on the environment to collect training data per episode.
 2. Compute expected return at each time step.
 3. Compute the loss for the combined actor-critic model.
 4. Compute gradients and update network parameters.
 5. Repeat 1-4 until either success criterion or max episodes has been reached.

1. Collecting training data

As in supervised learning, in order to train the actor-critic model, you need to have training data. However, in order to collect such data, the model would need to be "run" in the environment.

Training data is collected for each episode. Then at each time step, the model's forward pass will be run on the environment's state in order to generate action probabilities and the critic value based on the current policy parameterized by the model's weights.

The next action will be sampled from the action probabilities generated by the model, which would then be applied to the environment, causing the next state and reward to be generated.

This process is implemented in the `run_episode` function, which uses TensorFlow operations so that it can later be compiled into a TensorFlow graph for faster training. Note that `tf.TensorArray`s were used to support Tensor iteration on variable length arrays.

In []:

In []:

```
def run_episode(
    initial_state: tf.Tensor,
    model: tf.keras.Model,
    max_steps: int) -> Tuple[tf.Tensor, tf.Tensor, tf.Tensor]:
    """Runs a single episode to collect training data."""

action_probs = tf.TensorArray(dtype=tf.float32, size=0, dynamic_size=True)
values = tf.TensorArray(dtype=tf.float32, size=0, dynamic_size=True)
rewards = tf.TensorArray(dtype=tf.int32, size=0, dynamic_size=True)

initial_state_shape = initial_state.shape
state = initial_state

for t in tf.range(max_steps):
    # Convert state into a batched tensor (batch size = 1)
    state = tf.expand_dims(state, 0)

    # Run the model and to get action probabilities and critic value
    action_logits_t, value = model(state)

    # Sample next action from the action probability distribution
    action = tf.random.categorical(action_logits_t, 1)[0, 0]
    action_probs_t = tf.nn.softmax(action_logits_t)

    # Store critic values
    values = values.write(t, tf.squeeze(value))

    # Store log probability of the action chosen
    action_probs = action_probs.write(t, action_probs_t[0, action])

    # Apply action to the environment to get next state and reward
    state, reward, done = tf_env_step(action)
    state.set_shape(initial_state_shape)

    # Store reward
    rewards = rewards.write(t, reward)

    if tf.cast(done, tf.bool):
        break

action_probs = action_probs.stack()
values = values.stack()
rewards = rewards.stack()

return action_probs, values, rewards
```

2. Computing expected returns

The sequence of rewards for each timestep t , $\{r_t\}_{t=1}^T$ collected during one episode is converted into a sequence of expected returns $\{G_t\}_{t=1}^T$ in which the sum of rewards is taken from the current timestep t to T and each reward is multiplied with an exponentially decaying discount factor γ :

$$G_t = \sum_{t'=t}^T \gamma^{t'-t} r_{t'}$$

Since $\gamma \in (0, 1)$, rewards further out from the current timestep are given less weight.

Intuitively, expected return simply implies that rewards now are better than rewards later. In a mathematical sense, it is to ensure that the sum of the rewards converges.

To stabilize training, the resulting sequence of returns is also standardized (i.e. to have zero mean and unit standard deviation).

In []:

```
def get_expected_return(
    rewards: tf.Tensor,
    gamma: float,
    standardize: bool = True) -> tf.Tensor:
    """Compute expected returns per timestep."""

n = tf.shape(rewards)[0]
returns = tf.TensorArray(dtype=tf.float32, size=n)

# Start from the end of `rewards` and accumulate reward sums
# into the `returns` array
rewards = tf.cast(rewards[::-1], dtype=tf.float32)
discounted_sum = tf.constant(0.0)
discounted_sum_shape = discounted_sum.shape
for i in tf.range(n):
    reward = rewards[i]
    discounted_sum = reward + gamma * discounted_sum
    discounted_sum.set_shape(discounted_sum_shape)
    returns = returns.write(i, discounted_sum)
returns = returns.stack()[::-1]

if standardize:
    returns = ((returns - tf.math.reduce_mean(returns)) /
               (tf.math.reduce_std(returns) + eps))

return returns
```

3. The actor-critic loss

Since a hybrid actor-critic model is used, the chosen loss function is a combination of actor and critic losses for training, as shown below:

$$L = L_{actor} + L_{critic}$$

Actor loss

The actor loss is based on [policy gradients with the critic as a state dependent baseline](https://www.youtube.com/watch?v=EKqxumCuAY&t=62m23s) (<https://www.youtube.com/watch?v=EKqxumCuAY&t=62m23s>) and computed with single-sample (per-episode) estimates.

$$L_{actor} = - \sum_{t=1}^T \log \pi_\theta(a_t | s_t) [G(s_t, a_t) - V_\theta^\pi(s_t)]$$

where:

- T : the number of timesteps per episode, which can vary per episode
- s_t : the state at timestep t
- a_t : chosen action at timestep t given state s
- π_θ : is the policy (actor) parameterized by θ
- V_θ^π : is the value function (critic) also parameterized by θ
- $G = G_t$: the expected return for a given state, action pair at timestep t

A negative term is added to the sum since the idea is to maximize the probabilities of actions yielding higher rewards by minimizing the combined loss.

Advantage

The $G - V$ term in our L_{actor} formulation is called the [advantage](https://spinningup.openai.com/en/latest/spinningup/rl_intro.html#advantage-functions) (https://spinningup.openai.com/en/latest/spinningup/rl_intro.html#advantage-functions), which indicates how much better an action is given a particular state over a random action selected according to the policy π for that state.

While it's possible to exclude a baseline, this may result in high variance during training. And the nice thing about choosing the critic V as a baseline is that it trained to be as close as possible to G , leading to a lower variance.

In addition, without the critic, the algorithm would try to increase probabilities for actions taken on a particular state based on expected return, which may not make much of a difference if the relative probabilities between actions remain the same.

For instance, suppose that two actions for a given state would yield the same expected return. Without the critic, the algorithm would try to raise the probability of these actions based on the objective J . With the critic, it may turn out that there's no advantage ($G - V = 0$) and thus no benefit gained in increasing the actions' probabilities and the algorithm would set the gradients to zero.

Critic loss

Training V to be as close possible to G can be set up as a regression problem with the following loss function:

$$L_{critic} = L_\delta(G, V_\theta^\pi)$$

where L_δ is the [Huber loss](https://en.wikipedia.org/wiki/Huber_loss) (https://en.wikipedia.org/wiki/Huber_loss), which is less sensitive to outliers in data than squared-error loss.

In []:

```
huber_loss = tf.keras.losses.Huber(reduction=tf.keras.losses.Reduction.SUM)

def compute_loss(
    action_probs: tf.Tensor,
    values: tf.Tensor,
    returns: tf.Tensor) -> tf.Tensor:
    """Computes the combined actor-critic loss."""
    advantage = returns - values

    action_log_probs = tf.math.log(action_probs)
    actor_loss = -tf.math.reduce_sum(action_log_probs * advantage)

    critic_loss = huber_loss(values, returns)

    return actor_loss + critic_loss
```

4. Defining the training step to update parameters

All of the steps above are combined into a training step that is run every episode. All steps leading up to the loss function are executed with the `tf.GradientTape` context to enable automatic differentiation.

This tutorial uses the Adam optimizer to apply the gradients to the model parameters.

The sum of the undiscounted rewards, `episode_reward`, is also computed in this step. This value will be used later on to evaluate if the success criterion is met.

The `tf.function` context is applied to the `train_step` function so that it can be compiled into a callable TensorFlow graph, which can lead to 10x speedup in training.

In []:

```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.01)

@tf.function
def train_step(
    initial_state: tf.Tensor,
    model: tf.keras.Model,
    optimizer: tf.keras.optimizers.Optimizer,
    gamma: float,
    max_steps_per_episode: int) -> tf.Tensor:
    """Runs a model training step."""
    with tf.GradientTape() as tape:
        # Run the model for one episode to collect training data
        action_probs, values, rewards = run_episode(
            initial_state, model, max_steps_per_episode)

        # Calculate expected returns
        returns = get_expected_return(rewards, gamma)

        # Convert training data to appropriate TF tensor shapes
        action_probs, values, returns = [
            tf.expand_dims(x, 1) for x in [action_probs, values, returns]]

        # Calculating loss values to update our network
        loss = compute_loss(action_probs, values, returns)

        # Compute the gradients from the loss
        grads = tape.gradient(loss, model.trainable_variables)

        # Apply the gradients to the model's parameters
        optimizer.apply_gradients(zip(grads, model.trainable_variables))

        episode_reward = tf.math.reduce_sum(rewards)

    return episode_reward
```

5. Run the training loop

Training is executed by running the training step until either the success criterion or maximum number of episodes is reached.

A running record of episode rewards is kept in a queue. Once 100 trials are reached, the oldest reward is removed at the left (tail) end of the queue and the newest one is added at the head (right). A running sum of the rewards is also maintained for computational efficiency.

Depending on your runtime, training can finish in less than a minute.

In []:

```
%%time

min_episodes_criterion = 100
max_episodes = 10000
max_steps_per_episode = 1000

# Cartpole-v0 is considered solved if average reward is >= 195 over 100
# consecutive trials
reward_threshold = 195
running_reward = 0

# Discount factor for future rewards
gamma = 0.99

# Keep last episodes reward
episodes_reward: collections.deque = collections.deque(maxlen=min_episodes_criterion)

with tqdm.trange(max_episodes) as t:
    for i in t:
        initial_state = tf.constant(env.reset(), dtype=tf.float32)
        episode_reward = int(train_step(
            initial_state, model, optimizer, gamma, max_steps_per_episode))

        episodes_reward.append(episode_reward)
        running_reward = statistics.mean(episodes_reward)

        t.set_description(f'Episode {i}')
        t.set_postfix(
            episode_reward=episode_reward, running_reward=running_reward)

    # Show average episode reward every 10 episodes
    if i % 10 == 0:
        pass # print(f'Episode {i}: average reward: {avg_reward}')

    if running_reward > reward_threshold and i >= min_episodes_criterion:
        break

print(f'\nSolved at episode {i}: average reward: {running_reward:.2f}!')
```

Visualization

After training, it would be good to visualize how the model performs in the environment. You can run the cells below to generate a GIF animation of one episode run of the model. Note that additional packages need to be installed for OpenAI Gym to render the environment's images correctly in Colab.

In []:

```
# Render an episode and save as a GIF file

from IPython import display as ipythondisplay
from PIL import Image
from pyvirtualdisplay import Display

display = Display(visible=0, size=(400, 300))
display.start()

def render_episode(env: gym.Env, model: tf.keras.Model, max_steps: int):
    screen = env.render(mode='rgb_array')
    im = Image.fromarray(screen)

    images = [im]

    state = tf.constant(env.reset(), dtype=tf.float32)
    for i in range(1, max_steps + 1):
        state = tf.expand_dims(state, 0)
        action_probs, _ = model(state)
        action = np.argmax(np.squeeze(action_probs))

        state, _, done, _ = env.step(action)
        state = tf.constant(state, dtype=tf.float32)

        # Render screen every 10 steps
        if i % 10 == 0:
            screen = env.render(mode='rgb_array')
            images.append(Image.fromarray(screen))

        if done:
            break

    return images

# Save GIF image
images = render_episode(env, model, max_steps_per_episode)
image_file = 'cartpole-v0.gif'
# loop=0: loop forever, duration=1: play each frame for 1ms
images[0].save(
    image_file, save_all=True, append_images=images[1:], loop=0, duration=1)
```

In []:

```
import tensorflow_docs.vis.embed as embed
embed.embed_file(image_file)
```

Next steps

This tutorial demonstrated how to implement the actor-critic method using Tensorflow.

As a next step, you could try training a model on a different environment in OpenAI Gym.

For additional information regarding actor-critic methods and the Cartpole-v0 problem, you may refer to the following resources:

- [Actor Critic Method \(https://hal.inria.fr/hal-00840470/document\)](https://hal.inria.fr/hal-00840470/document)
- [Actor Critic Lecture \(CAL\) \(https://www.youtube.com/watch?v=EKqxumCuAY&list=PLkFD6_40KJlwhWJpGazJ9VSj9CFMkb79A&index=7&t=0s\)](https://www.youtube.com/watch?v=EKqxumCuAY&list=PLkFD6_40KJlwhWJpGazJ9VSj9CFMkb79A&index=7&t=0s)
- [Cartpole learning control problem \[Barto, et al. 1983\] \(http://www.derongliu.org/adp/adp-cdrom/Barto1983.pdf\)](http://www.derongliu.org/adp/adp-cdrom/Barto1983.pdf)

For more reinforcement learning examples in TensorFlow, you can check the following resources:

- [Reinforcement learning code examples \(keras.io\) \(https://keras.io/examples/rl/\)](https://keras.io/examples/rl/)
- [TF-Agents reinforcement learning library \(https://www.tensorflow.org/agents\)](https://www.tensorflow.org/agents)

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

TensorFlow 2 quickstart for beginners



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/quickstart/beginner)

(<https://www.tensorflow.org/tutorials/quickstart/beginner>)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/quickstart/beginner.ipynb)

(<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/quickstart/beginner.ipynb>)

This short introduction uses [Keras](https://www.tensorflow.org/guide/keras/overview) (<https://www.tensorflow.org/guide/keras/overview>) to:

1. Load a prebuilt dataset.
2. Build a neural network machine learning model that classifies images.
3. Train this neural network.
4. Evaluate the accuracy of the model.

This tutorial is a [Google Colaboratory](https://colab.research.google.com/notebooks/welcome.ipynb) (<https://colab.research.google.com/notebooks/welcome.ipynb>) notebook. Python programs are run directly in the browser—a great way to learn and use TensorFlow. To follow this tutorial, run the notebook in Google Colab by clicking the button at the top of this page.

1. In Colab, connect to a Python runtime: At the top-right of the menu bar, select CONNECT.
2. Run all the notebook code cells: Select Runtime > Run all.

Set up TensorFlow

Import TensorFlow into your program to get started:

In []:

```
import tensorflow as tf
print("TensorFlow version:", tf.__version__)
```

If you are following along in your own development environment, rather than [Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/quickstart/beginner.ipynb) (<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/quickstart/beginner.ipynb>), see the [install guide](https://www.tensorflow.org/install) (<https://www.tensorflow.org/install>) for setting up TensorFlow for development.

Note: Make sure you have upgraded to the latest pip to install the TensorFlow 2 package if you are using your own development environment. See the [install guide](https://www.tensorflow.org/install) (<https://www.tensorflow.org/install>) for details.

Load a dataset

Load and prepare the [MNIST dataset](http://yann.lecun.com/exdb/mnist/) (<http://yann.lecun.com/exdb/mnist/>). Convert the sample data from integers to floating-point numbers:

In []:

```
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
```

Build a machine learning model

Build a `tf.keras.Sequential` model by stacking layers.

In []:

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10)
])
```

For each example, the model returns a vector of [logits](https://developers.google.com/machine-learning/glossary#logits) (<https://developers.google.com/machine-learning/glossary#logits>) or [log-odds](https://developers.google.com/machine-learning/glossary#log-odds) (<https://developers.google.com/machine-learning/glossary#log-odds>) scores, one for each class.

In []:

```
predictions = model(x_train[:1]).numpy()
predictions
```

The `tf.nn.softmax` function converts these logits to *probabilities* for each class:

In []:

```
tf.nn.softmax(predictions).numpy()
```

Note: It is possible to bake the `tf.nn.softmax` function into the activation function for the last layer of the network. While this can make the model output more directly interpretable, this approach is discouraged as it's impossible to provide an exact and numerically stable loss calculation for all models when using a softmax output.

Define a loss function for training using `losses.SparseCategoricalCrossentropy`, which takes a vector of logits and a `True` index and returns a scalar loss for each example.

In []:

```
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
```

This loss is equal to the negative log probability of the true class: The loss is zero if the model is sure of the correct class.

This untrained model gives probabilities close to random (1/10 for each class), so the initial loss should be close to `-tf.math.log(1/10) ~= 2.3`.

In []:

```
loss_fn(y_train[:1], predictions).numpy()
```

Before you start training, configure and compile the model using Keras `Model.compile`. Set the [optimizer](https://www.tensorflow.org/api_docs/python/tf/keras/optimizers) (https://www.tensorflow.org/api_docs/python/tf/keras/optimizers) class to `adam`, set the `loss` to the `loss_fn` function you defined earlier, and specify a metric to be evaluated for the model by setting the `metrics` parameter to `accuracy`.

In []:

```
model.compile(optimizer='adam',
              loss=loss_fn,
              metrics=['accuracy'])
```

Train and evaluate your model

Use the `Model.fit` method to adjust your model parameters and minimize the loss:

In []:

```
model.fit(x_train, y_train, epochs=5)
```

The `Model.evaluate` method checks the models performance, usually on a "[Validation-set](https://developers.google.com/machine-learning/glossary#validation-set) (<https://developers.google.com/machine-learning/glossary#validation-set>)" or "[Test-set](https://developers.google.com/machine-learning/glossary#test-set) (<https://developers.google.com/machine-learning/glossary#test-set>)".

In []:

```
model.evaluate(x_test, y_test, verbose=2)
```

The image classifier is now trained to ~98% accuracy on this dataset. To learn more, read the [TensorFlow tutorials](https://www.tensorflow.org/tutorials/) (<https://www.tensorflow.org/tutorials/>).

If you want your model to return a probability, you can wrap the trained model, and attach the softmax to it:

In []:

```
probability_model = tf.keras.Sequential([
    model,
    tf.keras.layers.Softmax()
])
```

In []:

```
probability_model(x_test[:5])
```

Conclusion

Congratulations! You have trained a machine learning model using a prebuilt dataset using the [Keras](https://www.tensorflow.org/guide/keras/overview) (<https://www.tensorflow.org/guide/keras/overview>) API.

For more examples of using Keras, check out the [tutorials](https://www.tensorflow.org/tutorials/keras/) (<https://www.tensorflow.org/tutorials/keras/>). To learn more about building models with Keras, read the [guides](https://www.tensorflow.org/guide/keras) (<https://www.tensorflow.org/guide/keras>). If you want learn more about loading and preparing data, see the tutorials on [image data loading](https://www.tensorflow.org/tutorials/load_data/images) (https://www.tensorflow.org/tutorials/load_data/images) or [CSV data loading](https://www.tensorflow.org/tutorials/load_data/csv) (https://www.tensorflow.org/tutorials/load_data/csv).

Copyright 2019 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

TensorFlow 2 quickstart for experts



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/quickstart/advanced)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/quickstart/advanced.ipynb)

This is a [Google Colaboratory](https://colab.research.google.com/notebooks/welcome.ipynb) (<https://colab.research.google.com/notebooks/welcome.ipynb>) notebook file. Python programs are run directly in the browser—a great way to learn and use TensorFlow. To follow this tutorial, run the notebook in Google Colab by clicking the button at the top of this page.

1. In Colab, connect to a Python runtime: At the top-right of the menu bar, select CONNECT.
2. Run all the notebook code cells: Select Runtime > Run all.

Download and install TensorFlow 2. Import TensorFlow into your program:

Note: Upgrade pip to install the TensorFlow 2 package. See the [install guide](https://www.tensorflow.org/install) (<https://www.tensorflow.org/install>) for details.

Import TensorFlow into your program:

In []:

```
import tensorflow as tf
print("TensorFlow version:", tf.__version__)

from tensorflow.keras.layers import Dense, Flatten, Conv2D
from tensorflow.keras import Model
```

Load and prepare the [MNIST dataset](http://yann.lecun.com/exdb/mnist/) (<http://yann.lecun.com/exdb/mnist/>).

In []:

```
mnist = tf.keras.datasets.mnist  
(x_train, y_train), (x_test, y_test) = mnist.load_data()  
x_train, x_test = x_train / 255.0, x_test / 255.0  
  
# Add a channels dimension  
x_train = x_train[..., tf.newaxis].astype("float32")  
x_test = x_test[..., tf.newaxis].astype("float32")
```

Use `tf.data` to batch and shuffle the dataset:

In []:

```
train_ds = tf.data.Dataset.from_tensor_slices(  
    (x_train, y_train)).shuffle(10000).batch(32)  
  
test_ds = tf.data.Dataset.from_tensor_slices((x_test, y_test)).batch(32)
```

Build the `tf.keras` model using the Keras [model subclassing API](https://www.tensorflow.org/guide/keras#model_subclassing) (https://www.tensorflow.org/guide/keras#model_subclassing):

In []:

```
class MyModel(Model):  
    def __init__(self):  
        super(MyModel, self).__init__()  
        self.conv1 = Conv2D(32, 3, activation='relu')  
        self.flatten = Flatten()  
        self.d1 = Dense(128, activation='relu')  
        self.d2 = Dense(10)  
  
    def call(self, x):  
        x = self.conv1(x)  
        x = self.flatten(x)  
        x = self.d1(x)  
        return self.d2(x)  
  
# Create an instance of the model  
model = MyModel()
```

Choose an optimizer and loss function for training:

In []:

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

Select metrics to measure the loss and the accuracy of the model. These metrics accumulate the values over epochs and then print the overall result.

In []:

```
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')
```

Use `tf.GradientTape` to train the model:

In []:

```
@tf.function  
def train_step(images, labels):  
    with tf.GradientTape() as tape:  
        # training=True is only needed if there are layers with different  
        # behavior during training versus inference (e.g. Dropout).  
        predictions = model(images, training=True)  
        loss = loss_object(labels, predictions)  
        gradients = tape.gradient(loss, model.trainable_variables)  
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))  
  
        train_loss(loss)  
        train_accuracy(labels, predictions)
```

Test the model:

In []:

```
@tf.function
def test_step(images, labels):
    # training=False is only needed if there are layers with different
    # behavior during training versus inference (e.g. Dropout).
    predictions = model(images, training=False)
    t_loss = loss_object(labels, predictions)

    test_loss(t_loss)
    test_accuracy(labels, predictions)
```

In []:

```
EPOCHS = 5

for epoch in range(EPOCHS):
    # 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()

    for images, labels in train_ds:
        train_step(images, labels)

    for test_images, test_labels in test_ds:
        test_step(test_images, test_labels)

    print(
        f'Epoch {epoch + 1}, '
        f'Loss: {train_loss.result()}, '
        f'Accuracy: {train_accuracy.result() * 100}, '
        f'Test Loss: {test_loss.result()}, '
        f'Test Accuracy: {test_accuracy.result() * 100}'
    )
```

The image classifier is now trained to ~98% accuracy on this dataset. To learn more, read the [TensorFlow tutorials](https://www.tensorflow.org/tutorials) (<https://www.tensorflow.org/tutorials>).

Copyright 2020 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Parameter server training with ParameterServerStrategy



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/distribute/parameter_server_training)



(https://www.tensorflow.org/tutorials/distribute/parameter_server_training) (<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/distribute/paramet>)

Overview

[Parameter server training](https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-li_mu.pdf) (https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-li_mu.pdf) is a common data-parallel method to scale up model training on multiple machines.

A parameter server training cluster consists of *workers* and *parameter servers*. Variables are created on parameter servers and they are read and updated by workers in each step. By default, workers read and update these variables independently without synchronizing with each other. This is why sometimes parameter server-style training is called *asynchronous training*.

In TensorFlow 2, parameter server training is powered by the `tf.distribute.ParameterServerStrategy` class, which distributes the training steps to a cluster that scales up to thousands of workers (accompanied by parameter servers).

Supported training methods

There are two main supported training methods:

- The Keras `Model.fit` API: if you prefer a high-level abstraction and handling of training. This is generally recommended if you are training a `tf.keras.Model`.
- A custom training loop: if you prefer to define the details of your training loop (you can refer to guides on [Custom training \(..customization/custom_training_walkthrough.ipynb\)](#), [Writing a training loop from scratch \(https://www.tensorflow.org/guide/keras/writing_a_training_loop_from_scratch\)](#) and [Custom training loop with Keras and MultiWorkerMirroredStrategy \(multi_worker_with_ctl.ipynb\)](#) for more details).

A cluster with jobs and tasks

Regardless of the API of choice (`Model.fit` or a custom training loop), distributed training in TensorFlow 2 involves a 'cluster' with several 'jobs', and each of the jobs may have one or more 'tasks' .

When using parameter server training, it is recommended to have:

- One *coordinator* job (which has the job name `chief`)
- Multiple *worker* jobs (job name `worker`)
- Multiple *parameter server* jobs (job name `ps`)

The *coordinator* creates resources, dispatches training tasks, writes checkpoints, and deals with task failures. The *workers* and *parameter servers* run `tf.distribute.Server` instances that listen for requests from the coordinator.

Parameter server training with the `Model.fit` API

Parameter server training with the `Model.fit` API requires the coordinator to use a `tf.distribute.ParameterServerStrategy` object. Similar to `Model.fit` usage with no strategy, or with other strategies, the workflow involves creating and compiling the model, preparing the callbacks, and calling `Model.fit` .

Parameter server training with a custom training loop

With custom training loops, the `tf.distribute.coordinator.ClusterCoordinator` class is the key component used for the coordinator.

- The `ClusterCoordinator` class needs to work in conjunction with a `tf.distribute.Strategy` object.
- This `tf.distribute.Strategy` object is needed to provide the information of the cluster and is used to define a training step, as demonstrated in [Custom training with tf.distribute.Strategy \(custom_training.ipynb\)](#).
- The `ClusterCoordinator` object then dispatches the execution of these training steps to remote workers.
- For parameter server training, the `ClusterCoordinator` needs to work with a `tf.distribute.ParameterServerStrategy` .

The most important API provided by the `ClusterCoordinator` object is `schedule` :

- The `schedule` API enqueues a `tf.function` and returns a future-like `RemoteValue` immediately.
- The queued functions will be dispatched to remote workers in background threads and their `RemoteValue`s will be filled asynchronously.
- Since `schedule` doesn't require worker assignment, the `tf.function` passed in can be executed on any available worker.
- If the worker it is executed on becomes unavailable before its completion, the function will be retried on another available worker.
- Because of this fact and the fact that function execution is not atomic, a single function call may be executed more than once.

In addition to dispatching remote functions, the `ClusterCoordinator` also helps to create datasets on all the workers and rebuild these datasets when a worker recovers from failure.

Tutorial setup

The tutorial will branch into `Model.fit` and custom training loop paths, and you can choose the one that fits your needs. Sections other than "Training with X" are applicable to both paths.

In []:

```
!pip install portpicker
```

In []:

```
#@title
import multiprocessing
import os
import random
import portpicker
import tensorflow as tf
```

Cluster setup

As mentioned above, a parameter server training cluster requires a coordinator task that runs your training program, one or several workers and parameter server tasks that run TensorFlow servers—`tf.distribute.Server`—and possibly an additional evaluation task that runs sidebar evaluation (refer to the [sidebar evaluation section](#) below). The requirements to set them up are:

- The coordinator task needs to know the addresses and ports of all other TensorFlow servers, except the evaluator.
- The workers and parameter servers need to know which port they need to listen to. For the sake of simplicity, you can usually pass in the complete cluster information when creating TensorFlow servers on these tasks.
- The evaluator task doesn't have to know the setup of the training cluster. If it does, it should not attempt to connect to the training cluster.
- Workers and parameter servers should have task types as "worker" and "ps", respectively. The coordinator should use "chief" as the task type for legacy reasons.

In this tutorial, you will create an in-process cluster so that the whole parameter server training can be run in Colab. You will learn how to set up [real clusters](#) in a later section.

In-process cluster

You will start by creating several TensorFlow servers in advance and you will connect to them later. Note that this is only for the purpose of this tutorial's demonstration, and in real training the servers will be started on "worker" and "ps" machines.

In []:

```
def create_in_process_cluster(num_workers, num_ps):
    """Creates and starts local servers and returns the cluster_resolver."""
    worker_ports = [portpicker.pick_unused_port() for _ in range(num_workers)]
    ps_ports = [portpicker.pick_unused_port() for _ in range(num_ps)]

    cluster_dict = {}
    cluster_dict["worker"] = ["localhost:%s" % port for port in worker_ports]
    if num_ps > 0:
        cluster_dict["ps"] = ["localhost:%s" % port for port in ps_ports]

    cluster_spec = tf.train.ClusterSpec(cluster_dict)

    # Workers need some inter_ops threads to work properly.
    worker_config = tf.compat.v1.ConfigProto()
    if multiprocessing.cpu_count() < num_workers + 1:
        worker_config.inter_op_parallelism_threads = num_workers + 1

    for i in range(num_workers):
        tf.distribute.Server(
            cluster_spec,
            job_name="worker",
            task_index=i,
            config=worker_config,
            protocol="grpc")

    for i in range(num_ps):
        tf.distribute.Server(
            cluster_spec,
            job_name="ps",
            task_index=i,
            protocol="grpc")

    cluster_resolver = tf.distribute.cluster_resolver.SimpleClusterResolver(
        cluster_spec, rpc_layer="grpc")
    return cluster_resolver

# Set the environment variable to allow reporting worker and ps failure to the
# coordinator. This is a workaround and won't be necessary in the future.
os.environ["GRPC_FAIL_FAST"] = "use_caller"

NUM_WORKERS = 3
NUM_PS = 2
cluster_resolver = create_in_process_cluster(NUM_WORKERS, NUM_PS)
```

The in-process cluster setup is frequently used in unit testing, such as [here](#)

(https://github.com/tensorflow/tensorflow/blob/eb4c40fc91da260199fa2aed6fe67d36ad49fafd/tensorflow/python/distribute/coordinator/cluster_coordinator_te)

Another option for local testing is to launch processes on the local machine—check out [Multi-worker training with Keras \(multi_worker_with_keras.ipynb\)](#) for an example of this approach

Instantiate a ParameterServerStrategy

Before you dive into the training code, let's instantiate a `tf.distribute.ParameterServerStrategy` object. Note that this is needed regardless of whether you are proceeding with `Model.fit` or a custom training loop. The `variable_partitioner` argument will be explained in the [Variable sharding section](#).

In []:

```
variable_partitioner = (
    tf.distribute.experimental.partitioners.MinSizePartitioner(
        min_shard_bytes=(256 << 10),
        max_shards=NUM_PS))

strategy = tf.distribute.ParameterServerStrategy(
    cluster_resolver,
    variable_partitioner=variable_partitioner)
```

In order to use GPUs for training, allocate GPUs visible to each worker. `ParameterServerStrategy` will use all the available GPUs on each worker, with the restriction that all workers should have the same number of GPUs available.

Variable sharding

Variable sharding refers to splitting a variable into multiple smaller variables, which are called *shards*. Variable sharding may be useful to distribute the network load when accessing these shards. It is also useful to distribute computation and storage of a normal variable across multiple parameter servers, for example, when using very large embeddings that may not fit in a single machine's memory.

To enable variable sharding, you can pass in a `variable_partitioner` when constructing a `ParameterServerStrategy` object. The `variable_partitioner` will be invoked every time when a variable is created and it is expected to return the number of shards along each dimension of the variable. Some out-of-box `variable_partitioner`s are provided such as `tf.distribute.experimental.partitioners.MinSizePartitioner`. It is recommended to use size-based partitioners like `tf.distribute.experimental.partitioners.MinSizePartitioner` to avoid partitioning small variables, which could have a negative impact on model training speed.

When a `variable_partitioner` is passed in, and you create a variable directly under `Strategy.scope`, the variable will become a container type with a `variables` property, which provides access to the list of shards. In most cases, this container will be automatically converted to a Tensor by concatenating all the shards. As a result, it can be used as a normal variable. On the other hand, some TensorFlow methods such as `tf.nn.embedding_lookup` provide efficient implementation for this container type and in these methods automatic concatenation will be avoided.

Refer to the API docs of `tf.distribute.ParameterServerStrategy` for more details.

Training with Model.fit

Keras provides an easy-to-use training API via `Model.fit` that handles the training loop under the hood, with the flexibility of an overridable `train_step`, and callbacks which provide functionalities such as checkpoint saving or summary saving for TensorBoard. With `Model.fit`, the same training code can be used with other strategies with a simple swap of the strategy object.

Input data

Keras `Model.fit` with `tf.distribute.ParameterServerStrategy` can take input data in the form of a `tf.data.Dataset`, `tf.distribute.DistributedDataset`, or a `tf.keras.utils.experimental.DatasetCreator`, with `Dataset` being the recommended option for ease of use. If you encounter memory issues using `Dataset`, however, you may need to use `DatasetCreator` with a callable `dataset_fn` argument (refer to the `tf.keras.utils.experimental.DatasetCreator` API documentation for details).

If you transform your dataset into a `tf.data.Dataset`, you should use `Dataset.shuffle` and `Dataset.repeat`, as demonstrated in the code example below.

- Keras `Model.fit` with parameter server training assumes that each worker receives the same dataset, except when it is shuffled differently. Therefore, by calling `Dataset.shuffle`, you ensure more even iterations over the data.
- Because workers do not synchronize, they may finish processing their datasets at different times. Therefore, the easiest way to define epochs with parameter server training is to use `Dataset.repeat` —which repeats a dataset indefinitely when called without an argument—and specify the `steps_per_epoch` argument in the `Model.fit` call.

Refer to the "Training workflows" section of the [tf.data guide \(..../guide/data.ipynb\)](#) for more details on `shuffle` and `repeat`.

In []:

```
global_batch_size = 64  
  
x = tf.random.uniform((10, 10))  
y = tf.random.uniform((10,))  
  
dataset = tf.data.Dataset.from_tensor_slices((x, y)).shuffle(10).repeat()  
dataset = dataset.batch(global_batch_size)  
dataset = dataset.prefetch(2)
```

If you instead create your dataset with `tf.keras.utils.experimental.DatasetCreator`, the code in `dataset_fn` will be invoked on the input device, which is usually the CPU, on each of the worker machines.

Model construction and compiling

Now, you will create a `tf.keras.Model`—a trivial `tf.keras.models.Sequential` model for demonstration purposes—followed by a `Model.compile` call to incorporate components, such as an optimizer, metrics, and other parameters such as `steps_per_execution`:

In []:

```
with strategy.scope():  
    model = tf.keras.models.Sequential([tf.keras.layers.Dense(10)])  
  
    model.compile(tf.keras.optimizers.SGD(), loss="mse", steps_per_execution=10)
```

Callbacks and training

Before you call Keras `Model.fit` for the actual training, prepare any needed [callbacks](https://www.tensorflow.org/guide/keras/train_and_evaluate) (https://www.tensorflow.org/guide/keras/train_and_evaluate) for common tasks, such as:

- `tf.keras.callbacks.ModelCheckpoint`: saves the model at a certain frequency, such as after every epoch.
- `tf.keras.callbacks.BackupAndRestore`: provides fault tolerance by backing up the model and current epoch number, if the cluster experiences unavailability (such as abort or preemption). You can then restore the training state upon a restart from a job failure, and continue training from the beginning of the interrupted epoch.
- `tf.keras.callbacks.TensorBoard`: periodically writes model logs in summary files that can be visualized in the TensorBoard tool.

Note: Due to performance considerations, custom callbacks cannot have batch level callbacks overridden when used with `ParameterServerStrategy`. Please modify your custom callbacks to make them epoch level calls, and adjust `steps_per_epoch` to a suitable value. In addition, `steps_per_epoch` is a required argument for `Model.fit` when used with `ParameterServerStrategy`.

In []:

```
working_dir = "/tmp/my_working_dir"  
log_dir = os.path.join(working_dir, "log")  
ckpt_filepath = os.path.join(working_dir, "ckpt")  
backup_dir = os.path.join(working_dir, "backup")  
  
callbacks = [  
    tf.keras.callbacks.TensorBoard(log_dir=log_dir),  
    tf.keras.callbacks.ModelCheckpoint(filepath=ckpt_filepath),  
    tf.keras.callbacks.BackupAndRestore(backup_dir=backup_dir),  
]  
  
model.fit(dataset, epochs=5, steps_per_epoch=20, callbacks=callbacks)
```

Direct usage with ClusterCoordinator (optional)

Even if you choose the `Model.fit` training path, you can optionally instantiate a `tf.distribute.coordinator.ClusterCoordinator` object to schedule other functions you would like to be executed on the workers. Refer to the [Training with a custom training loop](#) section for more details and examples.

Training with a custom training loop

Using custom training loops with `tf.distribute.Strategy` provides great flexibility to define training loops. With the `ParameterServerStrategy` defined above (as `strategy`), you will use a `tf.distribute.coordinator.ClusterCoordinator` to dispatch the execution of training steps to remote workers.

Then, you will create a model, define a dataset, and define a step function, as you have done in the training loop with other `tf.distribute.Strategy`s. You can find more details in the [Custom training with tf.distribute.Strategy \(custom_training.ipynb\)](#) tutorial.

To ensure efficient dataset prefetching, use the recommended distributed dataset creation APIs mentioned in the [Dispatch training steps to remote workers](#) section below. Also, make sure to call `Strategy.run` inside `worker_fn` to take full advantage of GPUs allocated to workers. The rest of the steps are the same for training with or without GPUs.

Let's create these components in the following steps:

Set up the data

First, write a function that creates a dataset.

If you would like to preprocess the data with [Keras preprocessing layers](#) (https://www.tensorflow.org/guide/keras/preprocessing_layers) or [Tensorflow Transform layers](#) (<https://www.tensorflow.org/tfx/tutorials/transform/simple>), create these layers **outside the `dataset_fn`** and **under `Strategy.scope`**, like you would do for any other Keras layers. This is because the `dataset_fn` will be wrapped into a `tf.function` and then executed on each worker to generate the data pipeline.

If you don't follow the above procedure, creating the layers might create Tensorflow states which will be lifted out of the `tf.function` to the coordinator. Thus, accessing them on workers would incur repetitive RPC calls between coordinator and workers, and cause significant slowdown.

Placing the layers under `Strategy.scope` will instead create them on all workers. Then, you will apply the transformation inside the `dataset_fn` via `tf.data.Dataset.map`. Refer to *Data preprocessing* in the [Distributed input \(input.ipynb\)](#) tutorial for more information on data preprocessing with distributed input.

In []:

```
feature_vocab = [
    "avenger", "ironman", "batman", "hulk", "spiderman", "kingkong", "wonder_woman"
]
label_vocab = ["yes", "no"]

with strategy.scope():
    feature_lookup_layer = tf.keras.layers.StringLookup(
        vocabulary=feature_vocab,
        mask_token=None)
    label_lookup_layer = tf.keras.layers.StringLookup(
        vocabulary=label_vocab,
        num_oov_indices=0,
        mask_token=None)

    raw_feature_input = tf.keras.layers.Input(
        shape=(3,),
        dtype=tf.string,
        name="feature")
    feature_id_input = feature_lookup_layer(raw_feature_input)
    feature_preprocess_stage = tf.keras.Model(
        {"features": raw_feature_input},
        feature_id_input)

    raw_label_input = tf.keras.layers.Input(
        shape=(1,),
        dtype=tf.string,
        name="label")
    label_id_input = label_lookup_layer(raw_label_input)

    label_preprocess_stage = tf.keras.Model(
        {"label": raw_label_input},
        label_id_input)
```

Generate toy examples in a dataset:

```
In [ ]:
```

```
def feature_and_label_gen(num_examples=200):
    examples = {"features": [], "label": []}
    for _ in range(num_examples):
        features = random.sample(feature_vocab, 3)
        label = ["yes"] if "avenger" in features else ["no"]
        examples["features"].append(features)
        examples["label"].append(label)
    return examples

examples = feature_and_label_gen()
```

Then, create the training dataset wrapped in a `dataset_fn`:

```
In [ ]:
```

```
def dataset_fn(_):
    raw_dataset = tf.data.Dataset.from_tensor_slices(examples)

    train_dataset = raw_dataset.map(
        lambda x: {
            "features": feature_preprocess_stage(x["features"]),
            "label": label_preprocess_stage(x["label"])
        }).shuffle(200).batch(32).repeat()
    return train_dataset
```

Build the model

Next, create the model and other objects. Make sure to create all variables under `Strategy.scope`.

```
In [ ]:
```

```
# These variables created under the `Strategy.scope` will be placed on parameter
# servers in a round-robin fashion.
with strategy.scope():
    # Create the model. The input needs to be compatible with Keras processing layers.
    model_input = tf.keras.layers.Input(
        shape=(3,), dtype=tf.int64, name="model_input")

    emb_layer = tf.keras.layers.Embedding(
        input_dim=len(feature_lookup_layer.get_vocabulary()), output_dim=16384)
    emb_output = tf.reduce_mean(emb_layer(model_input), axis=1)
    dense_output = tf.keras.layers.Dense(units=1, activation="sigmoid")(emb_output)
    model = tf.keras.Model({"features": model_input}, dense_output)

    optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.1)
    accuracy = tf.keras.metrics.Accuracy()
```

Let's confirm that the use of `FixedShardsPartitioner` split all variables into two shards and that each shard was assigned to a different parameter server:

```
In [ ]:
```

```
assert len(emb_layer.weights) == 2
assert emb_layer.weights[0].shape == (4, 16384)
assert emb_layer.weights[1].shape == (4, 16384)
assert emb_layer.weights[0].device == "/job:ps/replica:0/task:0/device:CPU:0"
assert emb_layer.weights[1].device == "/job:ps/replica:0/task:1/device:CPU:0"
```

Define the training step

Third, create the training step wrapped into a `tf.function`:

In []:

```
@tf.function
def step_fn(iterator):

    def replica_fn(batch_data, labels):
        with tf.GradientTape() as tape:
            pred = model(batch_data, training=True)
            per_example_loss = tf.keras.losses.BinaryCrossentropy(
                reduction=tf.keras.losses.Reduction.NONE)(labels, pred)
            loss = tf.nn.compute_average_loss(per_example_loss)
            gradients = tape.gradient(loss, model.trainable_variables)

        optimizer.apply_gradients(zip(gradients, model.trainable_variables))

        actual_pred = tf.cast(tf.greater(pred, 0.5), tf.int64)
        accuracy.update_state(labels, actual_pred)
        return loss

    batch_data, labels = next(iterator)
    losses = strategy.run(replica_fn, args=(batch_data, labels))
    return strategy.reduce(tf.distribute.ReduceOp.SUM, losses, axis=None)
```

In the above training step function, calling `Strategy.run` and `Strategy.reduce` in the `step_fn` can support multiple GPUs per worker. If the workers have GPUs allocated, `Strategy.run` will distribute the datasets on multiple replicas.

Dispatch training steps to remote workers

After all the computations are defined by `ParameterServerStrategy`, you will use the `tf.distribute.coordinator.ClusterCoordinator` class to create resources and distribute the training steps to remote workers.

Let's first create a `ClusterCoordinator` object and pass in the strategy object:

In []:

```
coordinator = tf.distribute.coordinator.ClusterCoordinator(strategy)
```

Then, create a per-worker dataset and an iterator using the `ClusterCoordinator.create_per_worker_dataset` API, which replicates the dataset to all workers. In the `per_worker_dataset_fn` below, wrapping the `dataset_fn` into `strategy.distribute_datasets_from_function` is recommended to allow efficient prefetching to GPUs seamlessly.

In []:

```
@tf.function
def per_worker_dataset_fn():
    return strategy.distribute_datasets_from_function(dataset_fn)

per_worker_dataset = coordinator.create_per_worker_dataset(per_worker_dataset_fn)
per_worker_iterator = iter(per_worker_dataset)
```

The final step is to distribute the computation to remote workers using `ClusterCoordinator.schedule`:

- The `schedule` method enqueues a `tf.function` and returns a future-like `RemoteValue` immediately. The queued functions will be dispatched to remote workers in background threads and the `RemoteValue` will be filled asynchronously.
- The `join` method (`ClusterCoordinator.join`) can be used to wait until all scheduled functions are executed.

In []:

```
num_epochs = 4
steps_per_epoch = 5
for i in range(num_epochs):
    accuracy.reset_states()
    for _ in range(steps_per_epoch):
        coordinator.schedule(step_fn, args=(per_worker_iterator,))
    # Wait at epoch boundaries.
    coordinator.join()
    print("Finished epoch %d, accuracy is %f." % (i, accuracy.result().numpy()))
```

Here is how you can fetch the result of a `RemoteValue`:

In []:

```
loss = coordinator.schedule(step_fn, args=(per_worker_iterator,))
print("Final loss is %f" % loss.fetch())
```

Alternatively, you can launch all steps and do something while waiting for completion:

```
for _ in range(total_steps):
    coordinator.schedule(step_fn, args=(per_worker_iterator,))
while not coordinator.done():
    time.sleep(10)
    # Do something like logging metrics or writing checkpoints.
```

For the complete training and serving workflow for this particular example, please check out this [test](https://github.com/keras-team/keras/blob/master/keras/integration_test/parameter_server_keras_preprocessing_test.py) (https://github.com/keras-team/keras/blob/master/keras/integration_test/parameter_server_keras_preprocessing_test.py).

More about dataset creation

The dataset in the above code is created using the `ClusterCoordinator.create_per_worker_dataset` API. It creates one dataset per worker and returns a container object. You can call the `iter` method on it to create a per-worker iterator. The per-worker iterator contains one iterator per worker and the corresponding slice of a worker will be substituted in the input argument of the function passed to the `ClusterCoordinator.schedule` method before the function is executed on a particular worker.

The `ClusterCoordinator.schedule` method assumes workers are equivalent and thus assumes the datasets on different workers are the same (except that they may be shuffled differently). Because of this, it is also recommended to repeat datasets, and schedule a finite number of steps instead of relying on receiving an `OutOfRangeError` from a dataset.

Another important note is that `tf.data` datasets don't support implicit serialization and deserialization across task boundaries. So it is important to create the whole dataset inside the function passed to `ClusterCoordinator.create_per_worker_dataset`. The `create_per_worker_dataset` API can also directly take a `tf.data.Dataset` or `tf.distribute.DistributedDataset` as input.

Evaluation

The two main approaches to performing evaluation with `tf.distribute.ParameterServerStrategy` training are inline evaluation and sidecar evaluation. Each has its own pros and cons as described below. The inline evaluation method is recommended if you don't have a preference.

Inline evaluation

In this method, the coordinator alternates between training and evaluation, and thus it is called *inline evaluation*.

There are several benefits of inline evaluation. For example:

- It can support large evaluation models and evaluation datasets that a single task cannot hold.
- The evaluation results can be used to make decisions for training the next epoch, for example, whether to stop training early.

There are two ways to implement inline evaluation: direct evaluation and distributed evaluation.

- **Direct evaluation:** For small models and evaluation datasets, the coordinator can run evaluation directly on the distributed model with the evaluation dataset on the coordinator:

In []:

```
eval_dataset = tf.data.Dataset.from_tensor_slices(
    feature_and_label_gen(num_examples=16)).map(
        lambda x: (
            {"features": feature_preprocess_stage(x["features"])},
            label_preprocess_stage(x["label"]))
    ).batch(8)

eval_accuracy = tf.keras.metrics.Accuracy()

for batch_data, labels in eval_dataset:
    pred = model(batch_data, training=False)
    actual_pred = tf.cast(tf.greater(pred, 0.5), tf.int64)
    eval_accuracy.update_state(labels, actual_pred)

print("Evaluation accuracy: %f" % eval_accuracy.result())
```

- **Distributed evaluation:** For large models or datasets that are infeasible to run directly on the coordinator, the coordinator task can distribute evaluation tasks to the workers via the `ClusterCoordinator.schedule` / `ClusterCoordinator.join` methods:

In []:

```
with strategy.scope():
    # Define the eval metric on parameter servers.
    eval_accuracy = tf.keras.metrics.Accuracy()

@tf.function
def eval_step(iterator):
    def replica_fn(batch_data, labels):
        pred = model(batch_data, training=False)
        actual_pred = tf.cast(tf.greater(pred, 0.5), tf.int64)
        eval_accuracy.update_state(labels, actual_pred)
    batch_data, labels = next(iterator)
    strategy.run(replica_fn, args=(batch_data, labels))

def eval_dataset_fn():
    return tf.data.Dataset.from_tensor_slices(
        feature_and_label_gen(num_examples=16)).map(
            lambda x: {
                "features": feature_preprocess_stage(x["features"]),
                "label": label_preprocess_stage(x["label"])
            }).shuffle(16).repeat().batch(8)

per_worker_eval_dataset = coordinator.create_per_worker_dataset(eval_dataset_fn)
per_worker_eval_iterator = iter(per_worker_eval_dataset)

eval_steps_per_epoch = 2
for _ in range(eval_steps_per_epoch):
    coordinator.schedule(eval_step, args=(per_worker_eval_iterator,))
coordinator.join()
print("Evaluation accuracy: %f" % eval_accuracy.result())
```

Note: The `schedule` and `join` methods of `tf.distribute.coordinator.ClusterCoordinator` don't support visitation guarantees or exactly-once semantics. In other words, there is no guarantee that all evaluation examples in a dataset will be evaluated exactly once; some may not be visited and some may be evaluated multiple times. The `tf.data` service API can be used to provide exactly-once visitation for evaluation when using `ParameterServerStrategy` (refer to the *Dynamic Sharding* section of the `tf.data.experimental.service` API documentation).

Sidecar evaluation

Another method for defining and running an evaluation loop in `tf.distribute.ParameterServerStrategy` training is called *sidecar evaluation*, in which you create a dedicated evaluator task that repeatedly reads checkpoints and runs evaluation on the latest checkpoint (refer to [this guide](#) ([./guide/checkpoint.ipynb](#)) for more details on checkpointing). The chief and worker tasks do not spend any time on evaluation, so for a fixed number of iterations the overall training time should be shorter than using other evaluation methods. However, it requires an additional evaluator task and periodic checkpointing to trigger evaluation.

To write an evaluation loop for sidecar evaluation, you have two options:

1. Use the `tf.keras.utils.SidecarEvaluator` API.
2. Create a custom evaluation loop.

Refer to the `tf.keras.utils.SidecarEvaluator` API documentation for more details on option 1.

Sidecar evaluation is supported only with a single task. This means:

- It is guaranteed that each example is evaluated once. In the event the evaluator is preempted or restarted, it simply restarts the evaluation loop from the latest checkpoint, and the partial evaluation progress made before the restart is discarded.
- However, running evaluation on a single task implies that a full evaluation can possibly take a long time.
- If the size of the model is too large to fit into an evaluator's memory, single sidecar evaluation is not applicable.

Another caveat is that the `tf.keras.utils.SidecarEvaluator` implementation, and the custom evaluation loop below, may skip some checkpoints because it always picks up the latest checkpoint available, and during an evaluation epoch, multiple checkpoints can be produced from the training cluster. You can write a custom evaluation loop that evaluates every checkpoint, but it is not covered in this tutorial. On the other hand, it may sit idle if checkpoints are produced less frequently than how long it takes to run evaluation.

A custom evaluation loop provides more control over the details, such as choosing which checkpoint to evaluate, or providing any additional logic to run along with evaluation. The following is a possible custom sidecar evaluation loop:

```
checkpoint_dir = ...
eval_model = ...
eval_data = ...
checkpoint = tf.train.Checkpoint(model=eval_model)

for latest_checkpoint in tf.train.checkpoints_iterator(
    checkpoint_dir):
    try:
        checkpoint.restore(latest_checkpoint).expect_partial()
    except (tf.errors.OpError,) as e:
        # checkpoint may be deleted by training when it is about to read it.
        continue

    # Optionally add callbacks to write summaries.
    eval_model.evaluate(eval_data)

    # Evaluation finishes when it has evaluated the last epoch.
    if latest_checkpoint.endswith('-{}'.format(train_epochs)):
        break
```

Clusters in the real world

Note: this section is not necessary for running the tutorial code in this page.

In a real production environment, you will run all tasks in different processes on different machines. The simplest way to configure cluster information on each task is to set "TF_CONFIG" environment variables and use a `tf.distribute.cluster_resolver.TFConfigClusterResolver` to parse "TF_CONFIG".

For a general description of "TF_CONFIG" environment variables, refer to "Setting up the TF_CONFIG environment variable" in the [Distributed training \(../guide/distributed_training.ipynb\)](#) guide.

If you start your training tasks using Kubernetes or other configuration templates, likely, these templates have already set "TF_CONFIG" for you.

Set the "TF_CONFIG" environment variable

Suppose you have 3 workers and 2 parameter servers. Then the "TF_CONFIG" of worker 1 can be:

```
os.environ["TF_CONFIG"] = json.dumps({
    "cluster": {
        "worker": ["host1:port", "host2:port", "host3:port"],
        "ps": ["host4:port", "host5:port"],
        "chief": ["host6:port"]
    },
    "task": {"type": "worker", "index": 1}
})
```

The "TF_CONFIG" of the evaluator can be:

```
os.environ["TF_CONFIG"] = json.dumps({
    "cluster": {
        "evaluator": ["host7:port"]
    },
    "task": {"type": "evaluator", "index": 0}
})
```

The "cluster" part in the above "TF_CONFIG" string for the evaluator is optional.

If you use the same binary for all tasks

If you prefer to run all these tasks using a single binary, you will need to let your program branch into different roles at the very beginning:

```
cluster_resolver = tf.distribute.cluster_resolver.TFConfigClusterResolver()
if cluster_resolver.task_type in ("worker", "ps"):
    # Start a TensorFlow server and wait.
elif cluster_resolver.task_type == "evaluator":
    # Run sidecar evaluation
else:
    # Run the coordinator.
```

The following code starts a TensorFlow server and waits, useful for the "worker" and "ps" roles:

```
# Set the environment variable to allow reporting worker and ps failure to the
# coordinator. This is a workaround and won't be necessary in the future.
os.environ["GRPC_FAIL_FAST"] = "use_caller"

server = tf.distribute.Server(
    cluster_resolver.cluster_spec(),
    job_name=cluster_resolver.task_type,
    task_index=cluster_resolver.task_id,
    protocol=cluster_resolver.rpc_layer or "grpc",
    start=True)
server.join()
```

Handling task failure

Worker failure

Both the `tf.distribute.coordinator.ClusterCoordinator` custom training loop and `Model.fit` approaches provide built-in fault tolerance for worker failure. Upon worker recovery, the `ClusterCoordinator` invokes dataset re-creation on the workers.

Parameter server or coordinator failure

However, when the coordinator sees a parameter server error, it will raise an `UnavailableError` or `AbortedError` immediately. You can restart the coordinator in this case. The coordinator itself can also become unavailable. Therefore, certain tooling is recommended in order to not lose the training progress:

- For `Model.fit`, you should use a `BackupAndRestore` callback, which handles the progress saving and restoration automatically. See [Callbacks and training](#) section above for an example.
- For a custom training loop, you should checkpoint the model variables periodically and load model variables from a checkpoint, if any, before training starts. The training progress can be inferred approximately from `optimizer.iterations` if an optimizer is checkpointed:

```
checkpoint_manager = tf.train.CheckpointManager(
    tf.train.Checkpoint(model=model, optimizer=optimizer),
    checkpoint_dir,
    max_to_keep=3)
if checkpoint_manager.latest_checkpoint:
    checkpoint = checkpoint_manager.checkpoint
    checkpoint.restore(
        checkpoint_manager.latest_checkpoint).assert_existing_objects_matched()

global_steps = int(optimizer.iterations.numpy())
starting_epoch = global_steps // steps_per_epoch

for _ in range(starting_epoch, num_epochs):
    for _ in range(steps_per_epoch):
        coordinator.schedule(step_fn, args=(per_worker_iterator,))
    coordinator.join()
    checkpoint_manager.save()
```

Fetching a RemoteValue

Fetching a `RemoteValue` is guaranteed to succeed if a function is executed successfully. This is because currently the return value is immediately copied to the coordinator after a function is executed. If there is any worker failure during the copy, the function will be retried on another available worker. Therefore, if you want to optimize for performance, you can schedule functions without a return value.

Error reporting

Once the coordinator sees an error such as `UnavailableError` from parameter servers or other application errors such as an `InvalidArgumentException` from `tf.debugging.check_numerics`, it will cancel all pending and queued functions before raising the error. Fetching their corresponding `RemoteValue`s will raise a `CancelledError`.

After an error is raised, the coordinator will not raise the same error or any error from cancelled functions.

Performance improvement

There are several possible reasons you may experience performance issues when you train with `tf.distribute.ParameterServerStrategy` and `tf.distribute.coordinator.ClusterCoordinator`.

One common reason is that the parameter servers have unbalanced load and some heavily-loaded parameter servers have reached capacity. There can also be multiple root causes. Some simple methods to mitigate this issue are to:

1. Shard your large model variables via specifying a `variable_partitioner` when constructing a `ParameterServerStrategy`.
2. Avoid creating a hotspot variable that is required by all parameter servers in a single step if possible. For example, use a constant learning rate or subclass `tf.keras.optimizers.schedules.LearningRateSchedule` in optimizers since the default behavior is that the learning rate will become a variable placed on a particular parameter server and requested by all other parameter servers in each step.
3. Shuffle your large vocabularies before passing them to Keras preprocessing layers.

Another possible reason for performance issues is the coordinator. The implementation of `schedule / join` is Python-based and thus may have threading overhead. Also, the latency between the coordinator and the workers can be large. If this is the case:

- For `Model.fit`, you can set the `steps_per_execution` argument provided at `Model.compile` to a value larger than 1.
- For a custom training loop, you can pack multiple steps into a single `tf.function`:

```
steps_per_invocation = 10

@tf.function
def step_fn(iterator):
    for _ in range(steps_per_invocation):
        features, labels = next(iterator)
        def replica_fn(features, labels):
            ...
            strategy.run(replica_fn, args=(features, labels))
```

As the library is optimized further, hopefully most users won't have to manually pack steps in the future.

In addition, a small trick for performance improvement is to schedule functions without a return value as explained in the [handling task failure section](#) above.

Known limitations

Most of the known limitations are already covered in the above sections. This section provides a summary.

ParameterServerStrategy general

- `os.environment["grpc_fail_fast"]="use_caller"` is needed on every task including the coordinator, to make fault tolerance work properly.
- Synchronous parameter server training is not supported.
- It is usually necessary to pack multiple steps into a single function to achieve optimal performance.
- It is not supported to load a `saved_model` via `tf.saved_model.load` containing sharded variables. Note loading such a `saved_model` using TensorFlow Serving is expected to work (refer to the [serving tutorial](#) (https://www.tensorflow.org/tfx/tutorials/serving/rest_simple) for details).
- It is not supported to recover from parameter server failure without restarting the coordinator task.
- Creation of `tf.lookup.StaticHashTable`, commonly employed by some Keras preprocessing layers, such as `tf.keras.layers.IntegerLookup`, `tf.keras.layers.StringLookup`, and `tf.keras.layers.TextVectorization`, should be placed under `Strategy.scope`. Otherwise, resources will be placed on the coordinator, and lookup RPCs from workers to the coordinator incur performance implications.

Model.fit specifics

- `steps_per_epoch` argument is required in `Model.fit`. You can select a value that provides appropriate intervals in an epoch.
- `ParameterServerStrategy` does not have support for custom callbacks that have batch-level calls for performance reasons. You should convert those calls into epoch-level calls with suitably picked `steps_per_epoch`, so that they are called every `steps_per_epoch` number of steps. Built-in callbacks are not affected: their batch-level calls have been modified to be performant. Supporting batch-level calls for `ParameterServerStrategy` is being planned.
- For the same reason, unlike other strategies, progress bars and metrics are logged only at epoch boundaries.
- `run_eagerly` is not supported.

Custom training loop specifics

- `ClusterCoordinator.schedule` doesn't support visitation guarantees for a dataset.
- When `ClusterCoordinator.create_per_worker_dataset` is used with a callable as input, the whole dataset must be created inside the function passed to it.
- `tf.data.Options` is ignored in a dataset created by `ClusterCoordinator.create_per_worker_dataset`.

Copyright 2019 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Custom training loop with Keras and MultiWorkerMirroredStrategy



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/distribute/multi_worker_with_tf)

(https://www.tensorflow.org/tutorials/distribute/multi_worker_with_tf) (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/distribute/multi_worker_with_tf.ipynb)

Run in

Overview

This tutorial demonstrates multi-worker training with custom training loop API, distributed via `MultiWorkerMirroredStrategy`, so a Keras model designed to run on [single-worker](#) (https://www.tensorflow.org/tutorials/distribute/custom_training) can seamlessly work on multiple workers with minimal code change.

We are using custom training loops to train our model because they give us flexibility and a greater control on training. Moreover, it is easier to debug the model and the training loop. More detailed information is available in [Writing a training loop from scratch](#) (https://www.tensorflow.org/guide/keras/writing_a_training_loop_from_scratch).

If you are looking for how to use `MultiWorkerMirroredStrategy` with keras `model.fit`, refer to this [tutorial](#) (https://www.tensorflow.org/tutorials/distribute/multi_worker_with_keras) instead.

[Distributed Training in TensorFlow](#) ([https://www.tensorflow.org/.../distributed_training.ipynb](#)) guide is available for an overview of the distribution strategies TensorFlow supports for those interested in a deeper understanding of `tf.distribute.Strategy` APIs.

Setup

First, some necessary imports.

In []:

```
import json
import os
import sys
```

Before importing TensorFlow, make a few changes to the environment.

Disable all GPUs. This prevents errors caused by the workers all trying to use the same GPU. For a real application each worker would be on a different machine.

In []:

```
os.environ["CUDA_VISIBLE_DEVICES"] = "-1"
```

Reset the `TF_CONFIG` environment variable, you'll see more about this later.

In []:

```
os.environ.pop('TF_CONFIG', None)
```

Be sure that the current directory is on python's path. This allows the notebook to import the files written by `%%writefile` later.

In []:

```
if '.' not in sys.path:  
    sys.path.insert(0, '.')
```

Now import TensorFlow.

In []:

```
import tensorflow as tf
```

Dataset and model definition

Next create an `mnist.py` file with a simple model and dataset setup. This python file will be used by the worker-processes in this tutorial:

In []:

```
%%writefile mnist.py  
  
import os  
import tensorflow as tf  
import numpy as np  
  
def mnist_dataset(batch_size):  
    (x_train, y_train), _ = tf.keras.datasets.mnist.load_data()  
    # The `x` arrays are in uint8 and have values in the range [0, 255].  
    # You need to convert them to float32 with values in the range [0, 1]  
    x_train = x_train / np.float32(255)  
    y_train = y_train.astype(np.int64)  
    train_dataset = tf.data.Dataset.from_tensor_slices(  
        (x_train, y_train)).shuffle(60000)  
    return train_dataset  
  
def dataset_fn(global_batch_size, input_context):  
    batch_size = input_context.get_per_replica_batch_size(global_batch_size)  
    dataset = mnist_dataset(batch_size)  
    dataset = dataset.shard(input_context.num_input_pipelines,  
                           input_context.input_pipeline_id)  
    dataset = dataset.batch(batch_size)  
    return dataset  
  
def build_cnn_model():  
    return tf.keras.Sequential([  
        tf.keras.Input(shape=(28, 28)),  
        tf.keras.layers.Reshape(target_shape=(28, 28, 1)),  
        tf.keras.layers.Conv2D(32, 3, activation='relu'),  
        tf.keras.layers.Flatten(),  
        tf.keras.layers.Dense(128, activation='relu'),  
        tf.keras.layers.Dense(10)  
    ])
```

Multi-worker Configuration

Now let's enter the world of multi-worker training. In TensorFlow, the `TF_CONFIG` environment variable is required for training on multiple machines, each of which possibly has a different role. `TF_CONFIG` used below, is a JSON string used to specify the cluster configuration on each worker that is part of the cluster. This is the default method for specifying a cluster, using `cluster_resolver.TFConfigClusterResolver`, but there are other options available in the `distribute.cluster_resolver` module.

Describe your cluster

Here is an example configuration:

In []:

```
tf_config = {
    'cluster': {
        'worker': ['localhost:12345', 'localhost:23456']
    },
    'task': {'type': 'worker', 'index': 0}
}
```

Here is the same `TF_CONFIG` serialized as a JSON string:

In []:

```
json.dumps(tf_config)
```

There are two components of `TF_CONFIG`: `cluster` and `task`.

- `cluster` is the same for all workers and provides information about the training cluster, which is a dict consisting of different types of jobs such as `worker`. In multi-worker training with `MultiWorkerMirroredStrategy`, there is usually one `worker` that takes on a little more responsibility like saving checkpoint and writing summary file for TensorBoard in addition to what a regular `worker` does. Such a worker is referred to as the `chief worker`, and it is customary that the `worker` with `index 0` is appointed as the `chief worker` (in fact this is how `tf.distribute.Strategy` is implemented).
- `task` provides information of the current task and is different on each worker. It specifies the `type` and `index` of that worker.

In this example, you set the task `type` to "worker" and the task `index` to 0. This machine is the first worker and will be appointed as the chief worker and do more work than the others. Note that other machines will need to have the `TF_CONFIG` environment variable set as well, and it should have the same `cluster` dict, but different task `type` or task `index` depending on what the roles of those machines are.

For illustration purposes, this tutorial shows how one may set a `TF_CONFIG` with 2 workers on `localhost`. In practice, users would create multiple workers on external IP addresses/ports, and set `TF_CONFIG` on each worker appropriately.

In this example you will use 2 workers, the first worker's `TF_CONFIG` is shown above. For the second worker you would set `tf_config['task']['index']=1`

Above, `tf_config` is just a local variable in python. To actually use it to configure training, this dictionary needs to be serialized as JSON, and placed in the `TF_CONFIG` environment variable.

Environment variables and subprocesses in notebooks

Subprocesses inherit environment variables from their parent. So if you set an environment variable in this `jupyter` notebook process:

In []:

```
os.environ['GREETINGS'] = 'Hello TensorFlow!'
```

You can access the environment variable from a subprocesses:

In []:

```
%%bash
echo ${GREETINGS}
```

In the next section, you'll use this to pass the `TF_CONFIG` to the worker subprocesses. You would never really launch your jobs this way, but it's sufficient for the purposes of this tutorial: To demonstrate a minimal multi-worker example.

MultiWorkerMirroredStrategy

To train the model, use an instance of `tf.distribute.MultiWorkerMirroredStrategy`, which creates copies of all variables in the model's layers on each device across all workers. The [tf.distribute.Strategy guide \(../../guide/distributed_training.ipynb\)](#) has more details about this strategy.

In []:

```
strategy = tf.distribute.MultiWorkerMirroredStrategy()
```

Note: `TF_CONFIG` is parsed and TensorFlow's GRPC servers are started at the time `MultiWorkerMirroredStrategy()` is called, so the `TF_CONFIG` environment variable must be set before a `tf.distribute.Strategy` instance is created. To save time in this illustrative example we have not done this so that servers do not need to start. A full example is found in the last section of this tutorial.

Use `tf.distribute.Strategy.scope` to specify that a strategy should be used when building your model. This puts you in the "[cross-replica context \(https://www.tensorflow.org/guide/distributed_training?hl=en#mirroredstrategy\)](https://www.tensorflow.org/guide/distributed_training?hl=en#mirroredstrategy)" for this strategy, which means the strategy is put in control of things like variable placement.

In []:

```
import mnist
with strategy.scope():
    # Model building needs to be within `strategy.scope()`.
    multi_worker_model = mnist.build_cnn_model()
```

Auto-shard your data across workers

In multi-worker training, dataset sharding is not necessarily needed, however it gives you exactly-once semantics which makes more training more reproducible, i.e. training on multiple workers should be the same as training on one worker. Note: performance can be affected in some cases.

See: [distribute_datasets_from_function \(https://www.tensorflow.org/api_docs/python/tf/distribute/Strategy?version=nightly#distribute_datasets_from_function\)](https://www.tensorflow.org/api_docs/python/tf/distribute/Strategy?version=nightly#distribute_datasets_from_function)

In []:

```
per_worker_batch_size = 64
num_workers = len(tf_config['cluster']['worker'])
global_batch_size = per_worker_batch_size * num_workers

with strategy.scope():
    multi_worker_dataset = strategy.distribute_datasets_from_function(
        lambda input_context: mnist.dataset_fn(global_batch_size, input_context))
```

Define Custom Training Loop and Train the model

Specify an optimizer

In []:

```
with strategy.scope():
    # The creation of optimizer and train_accuracy will need to be in
    # `strategy.scope()` as well, since they create variables.
    optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.001)
    train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(
        name='train_accuracy')
```

Define a training step with `tf.function`

In []:

```
@tf.function
def train_step(iterator):
    """Training step function."""

    def step_fn(inputs):
        """Per-Replica step function."""
        x, y = inputs
        with tf.GradientTape() as tape:
            predictions = multi_worker_model(x, training=True)
            per_batch_loss = tf.keras.losses.SparseCategoricalCrossentropy(
                from_logits=True,
                reduction=tf.keras.losses.Reduction.NONE)(y, predictions)
            loss = tf.nn.compute_average_loss(
                per_batch_loss, global_batch_size=global_batch_size)

        grads = tape.gradient(loss, multi_worker_model.trainable_variables)
        optimizer.apply_gradients(
            zip(grads, multi_worker_model.trainable_variables))
        train_accuracy.update_state(y, predictions)
        return loss

    per_replica_losses = strategy.run(step_fn, args=(next(iterator),))
    return strategy.reduce(
        tf.distribute.ReduceOp.SUM, per_replica_losses, axis=None)
```

Checkpoint saving and restoring

Checkpointing implementation in a Custom Training Loop requires the user to handle it instead of using a keras callback. It allows you to save model's weights and restore them without having to save the whole model.

In []:

```
from multiprocessing import util
checkpoint_dir = os.path.join(util.get_temp_dir(), 'ckpt')

def _is_chief(task_type, task_id, cluster_spec):
    return (task_type is None
            or task_type == 'chief'
            or (task_type == 'worker'
                and task_id == 0
                and "chief" not in cluster_spec.as_dict()))

def _get_temp_dir(dirpath, task_id):
    base_dirpath = 'workertemp_' + str(task_id)
    temp_dir = os.path.join(dirpath, base_dirpath)
    tf.io.gfile.makedirs(temp_dir)
    return temp_dir

def write_filepath(filepath, task_type, task_id, cluster_spec):
    dirpath = os.path.dirname(filepath)
    base = os.path.basename(filepath)
    if not _is_chief(task_type, task_id, cluster_spec):
        dirpath = _get_temp_dir(dirpath, task_id)
    return os.path.join(dirpath, base)
```

Note: Checkpointing and Saving need to happen on each worker and they need to write to different paths as they would override each others. If you chose to only checkpoint/save on the chief, this can lead to deadlock and is not recommended.

Here, you'll create one `tf.train.Checkpoint` that tracks the model, which is managed by a `tf.train.CheckpointManager` so that only the latest checkpoint is preserved.

In []:

```
epoch = tf.Variable(
    initial_value=tf.constant(0, dtype=tf.dtypes.int64), name='epoch')
step_in_epoch = tf.Variable(
    initial_value=tf.constant(0, dtype=tf.dtypes.int64),
    name='step_in_epoch')
task_type, task_id = (strategy.cluster_resolver.task_type,
                      strategy.cluster_resolver.task_id)
# We normally don't need to manually instantiate a ClusterSpec, but in this
# illustrative example we did not set TF_CONFIG before initializing the
# strategy. See the next section for "real-world" usage.
cluster_spec = tf.train.ClusterSpec(tf_config['cluster'])

checkpoint = tf.train.Checkpoint(
    model=multi_worker_model, epoch=epoch, step_in_epoch=step_in_epoch)

write_checkpoint_dir = write_filepath(checkpoint_dir, task_type, task_id,
                                      cluster_spec)
checkpoint_manager = tf.train.CheckpointManager(
    checkpoint, directory=write_checkpoint_dir, max_to_keep=1)
```

Now, when you need to restore, you can find the latest checkpoint saved using the convenient `tf.train.latest_checkpoint` function.

In []:

```
latest_checkpoint = tf.train.latest_checkpoint(checkpoint_dir)
if latest_checkpoint:
    checkpoint.restore(latest_checkpoint)
```

After restoring the checkpoint, you can continue with training your custom training loop.

In []:

```
num_epochs = 3
num_steps_per_epoch = 70

while epoch.numpy() < num_epochs:
    iterator = iter(multi_worker_dataset)
    total_loss = 0.0
    num_batches = 0

    while step_in_epoch.numpy() < num_steps_per_epoch:
        total_loss += train_step(iterator)
        num_batches += 1
        step_in_epoch.assign_add(1)

    train_loss = total_loss / num_batches
    print('Epoch: %d, accuracy: %f, train_loss: %f.'
          %(epoch.numpy(), train_accuracy.result(), train_loss))

    train_accuracy.reset_states()

# Once the `CheckpointManager` is set up, you're now ready to save, and remove
# the checkpoints non-chief workers saved.
checkpoint_manager.save()
if not _is_chief(task_type, task_id, cluster_spec):
    tf.io.gfile.rmtree(write_checkpoint_dir)

epoch.assign_add(1)
step_in_epoch.assign(0)
```

Full code setup on workers

To actually run with `MultiWorkerMirroredStrategy` you'll need to run worker processes and pass a `TF_CONFIG` to them.

Like the `mnist.py` file written earlier, here is the `main.py` that contain the same code we walked through step by step previously in this colab, we're just writing it to a file so each of the workers will run it:

In []:

```
%%writefile main.py
#@title File: `main.py`
import os
import json
import tensorflow as tf
import mnist
from multiprocessing import util

per_worker_batch_size = 64
tf_config = json.loads(os.environ['TF_CONFIG'])
num_workers = len(tf_config['cluster']['worker'])
global_batch_size = per_worker_batch_size * num_workers

num_epochs = 3
num_steps_per_epoch=70

# Checkpoint saving and restoring
def _is_chief(task_type, task_id, cluster_spec):
    return (task_type is None
            or task_type == 'chief'
            or (task_type == 'worker'
                and task_id == 0
                and 'chief' not in cluster_spec.as_dict()))

def _get_temp_dir(dirpath, task_id):
    base_dirpath = 'workertemp_' + str(task_id)
    temp_dir = os.path.join(dirpath, base_dirpath)
    tf.io.gfile.makedirs(temp_dir)
    return temp_dir

def write_filepath(filepath, task_type, task_id, cluster_spec):
    dirpath = os.path.dirname(filepath)
    base = os.path.basename(filepath)
    if not _is_chief(task_type, task_id, cluster_spec):
        dirpath = _get_temp_dir(dirpath, task_id)
    return os.path.join(dirpath, base)

checkpoint_dir = os.path.join(util.get_temp_dir(), 'ckpt')

# Define Strategy
strategy = tf.distribute.MultiWorkerMirroredStrategy()
```

```

with strategy.scope():
    # Model building/compiling need to be within `strategy.scope()`.
    multi_worker_model = mnist.build_cnn_model()

    multi_worker_dataset = strategy.distribute_datasets_from_function(
        lambda input_context: mnist.dataset_fn(global_batch_size, input_context))
    optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.001)
    train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(
        name='train_accuracy')

@tf.function
def train_step(iterator):
    """Training step function."""

    def step_fn(inputs):
        """Per-Replica step function."""
        x, y = inputs
        with tf.GradientTape() as tape:
            predictions = multi_worker_model(x, training=True)
            per_batch_loss = tf.keras.losses.SparseCategoricalCrossentropy(
                from_logits=True,
                reduction=tf.keras.losses.Reduction.NONE)(y, predictions)
            loss = tf.nn.compute_average_loss(
                per_batch_loss, global_batch_size=global_batch_size)

        grads = tape.gradient(loss, multi_worker_model.trainable_variables)
        optimizer.apply_gradients(
            zip(grads, multi_worker_model.trainable_variables))
        train_accuracy.update_state(y, predictions)

        return loss

    per_replica_losses = strategy.run(step_fn, args=(next(iterator),))
    return strategy.reduce(
        tf.distribute.ReduceOp.SUM, per_replica_losses, axis=None)

epoch = tf.Variable(
    initial_value=tf.constant(0, dtype=tf.dtypes.int64), name='epoch')
step_in_epoch = tf.Variable(
    initial_value=tf.constant(0, dtype=tf.dtypes.int64),
    name='step_in_epoch')

task_type, task_id, cluster_spec = (strategy.cluster_resolver.task_type,
                                      strategy.cluster_resolver.task_id,
                                      strategy.cluster_resolver.cluster_spec())

checkpoint = tf.train.Checkpoint(
    model=multi_worker_model, epoch=epoch, step_in_epoch=step_in_epoch)

write_checkpoint_dir = write_filepath(checkpoint_dir, task_type, task_id,
                                      cluster_spec)
checkpoint_manager = tf.train.CheckpointManager(
    checkpoint, directory=write_checkpoint_dir, max_to_keep=1)

# Restoring the checkpoint
latest_checkpoint = tf.train.latest_checkpoint(checkpoint_dir)
if latest_checkpoint:
    checkpoint.restore(latest_checkpoint)

# Resume our CTL training
while epoch.numpy() < num_epochs:
    iterator = iter(multi_worker_dataset)
    total_loss = 0.0
    num_batches = 0

    while step_in_epoch.numpy() < num_steps_per_epoch:
        total_loss += train_step(iterator)
        num_batches += 1
        step_in_epoch.assign_add(1)

    train_loss = total_loss / num_batches
    print('Epoch: %d, accuracy: %f, train_loss: %f.'
          %(epoch.numpy(), train_accuracy.result(), train_loss))

    train_accuracy.reset_states()

    checkpoint_manager.save()
    if not _is_chief(task_type, task_id, cluster_spec):
        tf.io.gfile.rmtree(write_checkpoint_dir)

    epoch.assign_add(1)
    step_in_epoch.assign(0)

```

Train and Evaluate

The current directory now contains both Python files:

In []:

```
%%bash  
ls *.py
```

So json-serialize the `TF_CONFIG` and add it to the environment variables:

In []:

```
os.environ['TF_CONFIG'] = json.dumps(tf_config)
```

Now, you can launch a worker process that will run the `main.py` and use the `TF_CONFIG`:

In []:

```
# first kill any previous runs  
%killbgscripts
```

In []:

```
%%bash --bg  
python main.py &> job_0.log
```

There are a few things to note about the above command:

1. It uses the `%%bash` which is a [notebook "magic"](https://ipython.readthedocs.io/en/stable/interactive/magics.html) (<https://ipython.readthedocs.io/en/stable/interactive/magics.html>) to run some bash commands.
2. It uses the `--bg` flag to run the `bash` process in the background, because this worker will not terminate. It waits for all the workers before it starts.

The backgrounded worker process won't print output to this notebook, so the `&>` redirects its output to a file, so you can see what happened.

So, wait a few seconds for the process to start up:

In []:

```
import time  
time.sleep(20)
```

Now look what's been output to the worker's logfile so far:

In []:

```
%%bash  
cat job_0.log
```

The last line of the log file should say: `Started server with target: grpc://localhost:12345`. The first worker is now ready, and is waiting for all the other worker(s) to be ready to proceed.

So update the `tf_config` for the second worker's process to pick up:

In []:

```
tf_config['task']['index'] = 1  
os.environ['TF_CONFIG'] = json.dumps(tf_config)
```

Now launch the second worker. This will start the training since all the workers are active (so there's no need to background this process):

In []:

```
%%bash  
python main.py > /dev/null 2>&1
```

Now if you recheck the logs written by the first worker you'll see that it participated in training that model:

In []:

```
%%bash  
cat job_0.log
```

In []:

```
# Delete the `TF_CONFIG`, and kill any background tasks so they don't affect the next section.  
os.environ.pop('TF_CONFIG', None)  
%killbgscripts
```

Multi worker training in depth

This tutorial has demonstrated a `Custom Training Loop` workflow of the multi-worker setup. A detailed description of other topics is available in the `model.fit`'s guide (https://colab.sandbox.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/distribute/multi_worker_with_keras.ipynb) of the multi-worker setup and applicable to CTLS.

See also

1. [Distributed Training in TensorFlow](https://www.tensorflow.org/guide/distributed_training) (https://www.tensorflow.org/guide/distributed_training) guide provides an overview of the available distribution strategies.
2. [Official models](https://github.com/tensorflow/models/tree/master/official) (<https://github.com/tensorflow/models/tree/master/official>), many of which can be configured to run multiple distribution strategies.
3. The [Performance section](#) ([./guide/function.ipynb](#)) in the guide provides information about other strategies and [tools](#) ([./guide/profiler.md](#)) you can use to optimize the performance of your TensorFlow models.

Copyright 2019 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");  
# you may not use this file except in compliance with the License.  
# You may obtain a copy of the License at  
#  
# https://www.apache.org/licenses/LICENSE-2.0  
#  
# Unless required by applicable law or agreed to in writing, software  
# distributed under the License is distributed on an "AS IS" BASIS,  
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
# See the License for the specific language governing permissions and  
# limitations under the License.
```

Custom training with `tf.distribute.Strategy`



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/distribute/custom_training)

(https://www.tensorflow.org/tutorials/distribute/custom_training)



[Run in Google Colab](#)

(https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/distribute/custom_training.ipynb)

This tutorial demonstrates how to use `tf.distribute.Strategy` — a TensorFlow API that provides an abstraction for [distributing your training](#) ([./guide/distributed_training.ipynb](#)) across multiple processing units (GPUs, multiple machines, or TPUs) — with custom training loops. In this example, you will train a simple convolutional neural network on the [Fashion MNIST dataset](#) (<https://github.com/zalandoresearch/fashion-mnist>) containing 70,000 images of size 28 x 28.

[Custom training loops](#) ([./customization/custom_training_walkthrough.ipynb](#)) provide flexibility and a greater control on training. They also make it easier to debug the model and the training loop.

In []:

```
# Import TensorFlow  
import tensorflow as tf  
  
# Helper libraries  
import numpy as np  
import os  
  
print(tf.__version__)
```

Download the Fashion MNIST dataset

In []:

```
fashion_mnist = tf.keras.datasets.fashion_mnist  
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()  
  
# Add a dimension to the array -> new shape == (28, 28, 1)  
# This is done because the first layer in our model is a convolutional  
# layer and it requires a 4D input (batch_size, height, width, channels).  
# batch_size dimension will be added later on.  
train_images = train_images[..., None]  
test_images = test_images[..., None]  
  
# Scale the images to the [0, 1] range.  
train_images = train_images / np.float32(255)  
test_images = test_images / np.float32(255)
```

Create a strategy to distribute the variables and the graph

How does `tf.distribute.MirroredStrategy` strategy work?

- All the variables and the model graph are replicated across the replicas.
- Input is evenly distributed across the replicas.
- Each replica calculates the loss and gradients for the input it received.
- The gradients are synced across all the replicas by summing them.
- After the sync, the same update is made to the copies of the variables on each replica.

Note: You can put all the code below inside a single scope. This example divides it into several code cells for illustration purposes.

In []:

```
# If the list of devices is not specified in  
# `tf.distribute.MirroredStrategy` constructor, they will be auto-detected.  
strategy = tf.distribute.MirroredStrategy()
```

In []:

```
print('Number of devices: {}'.format(strategy.num_replicas_in_sync))
```

Setup input pipeline

In []:

```
BUFFER_SIZE = len(train_images)  
  
BATCH_SIZE_PER_REPLICA = 64  
GLOBAL_BATCH_SIZE = BATCH_SIZE_PER_REPLICA * strategy.num_replicas_in_sync  
  
EPOCHS = 10
```

Create the datasets and distribute them:

In []:

```
train_dataset = tf.data.Dataset.from_tensor_slices((train_images, train_labels)).shuffle(BUFFER_SIZE).batch(GLOBAL_BATCH_SIZE)  
test_dataset = tf.data.Dataset.from_tensor_slices((test_images, test_labels)).batch(GLOBAL_BATCH_SIZE)  
  
train_dist_dataset = strategy.experimental_distribute_dataset(train_dataset)  
test_dist_dataset = strategy.experimental_distribute_dataset(test_dataset)
```

Create the model

Create a model using `tf.keras.Sequential`. You can also use the [Model Subclassing API](#) (https://www.tensorflow.org/guide/keras/custom_layers_and_models) or the functional API (<https://www.tensorflow.org/guide/keras/functional>) to do this.

In []:

```
def create_model():
    model = tf.keras.Sequential([
        tf.keras.layers.Conv2D(32, 3, activation='relu'),
        tf.keras.layers.MaxPooling2D(),
        tf.keras.layers.Conv2D(64, 3, activation='relu'),
        tf.keras.layers.MaxPooling2D(),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(10)
    ])
    return model
```

In []:

```
# Create a checkpoint directory to store the checkpoints.
checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
```

Define the loss function

Normally, on a single machine with single GPU/CPU, loss is divided by the number of examples in the batch of input.

So, how should the loss be calculated when using a `tf.distribute.Strategy` ?

- For an example, let's say you have 4 GPU's and a batch size of 64. One batch of input is distributed across the replicas (4 GPUs), each replica getting an input of size 16.
- The model on each replica does a forward pass with its respective input and calculates the loss. Now, instead of dividing the loss by the number of examples in its respective input (`BATCH_SIZE_PER_REPLICA = 16`), the loss should be divided by the `GLOBAL_BATCH_SIZE` (64).

Why do this?

- This needs to be done because after the gradients are calculated on each replica, they are synced across the replicas by **summing** them.

How to do this in TensorFlow?

- If you're writing a custom training loop, as in this tutorial, you should sum the per example losses and divide the sum by the `GLOBAL_BATCH_SIZE` :
`scale_loss = tf.reduce_sum(loss) * (1. / GLOBAL_BATCH_SIZE)` or you can use `tf.nn.compute_average_loss` which takes the per example loss, optional sample weights, and `GLOBAL_BATCH_SIZE` as arguments and returns the scaled loss.
- If you are using regularization losses in your model then you need to scale the loss value by number of replicas. You can do this by using the `tf.nn.scale_regularization_loss` function.
- Using `tf.reduce_mean` is not recommended. Doing so divides the loss by actual per replica batch size which may vary step to step.
- This reduction and scaling is done automatically in keras `model.compile` and `model.fit`
- If using `tf.keras.losses` classes (as in the example below), the loss reduction needs to be explicitly specified to be one of `NONE` or `SUM` . `AUTO` and `SUM_OVER_BATCH_SIZE` are disallowed when used with `tf.distribute.Strategy` . `AUTO` is disallowed because the user should explicitly think about what reduction they want to make sure it is correct in the distributed case. `SUM_OVER_BATCH_SIZE` is disallowed because currently it would only divide by per replica batch size, and leave the dividing by number of replicas to the user, which might be easy to miss. So instead we ask the user do the reduction themselves explicitly.
- If `labels` is multi-dimensional, then average the `per_example_loss` across the number of elements in each sample. For example, if the shape of `predictions` is `(batch_size, H, W, n_classes)` and `labels` is `(batch_size, H, W)` , you will need to update `per_example_loss` like: `per_example_loss /= tf.cast(tf.reduce_prod(tf.shape(labels)[1:]), tf.float32)`

Caution: **Verify the shape of your loss**. Loss functions in `tf.losses` / `tf.keras.losses` typically return the average over the last dimension of the input. The loss classes wrap these functions. Passing `reduction=Reduction.NONE` when creating an instance of a loss class means "no additional reduction". For categorical losses with an example input shape of `[batch, W, H, n_classes]` the `n_classes` dimension is reduced. For pointwise losses like `losses.mean_squared_error` or `losses.binary_crossentropy` include a dummy axis so that `[batch, W, H, 1]` is reduced to `[batch, W, H]` . Without the dummy axis `[batch, W, H]` will be incorrectly reduced to `[batch, W]` .

In []:

```
with strategy.scope():
    # Set reduction to `NONE` so you can do the reduction afterwards and divide by
    # global batch size.
    loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
        from_logits=True,
        reduction=tf.keras.losses.Reduction.NONE)
    def compute_loss(labels, predictions):
        per_example_loss = loss_object(labels, predictions)
        return tf.nn.compute_average_loss(per_example_loss, global_batch_size=GLOBAL_BATCH_SIZE)
```

Define the metrics to track loss and accuracy

These metrics track the test loss and training and test accuracy. You can use `.result()` to get the accumulated statistics at any time.

In []:

```
with strategy.scope():
    test_loss = tf.keras.metrics.Mean(name='test_loss')

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

Training loop

In []:

```
# model, optimizer, and checkpoint must be created under `strategy.scope`.
with strategy.scope():
    model = create_model()

    optimizer = tf.keras.optimizers.Adam()

    checkpoint = tf.train.Checkpoint(optimizer=optimizer, model=model)
```

In []:

```
def train_step(inputs):
    images, labels = inputs

    with tf.GradientTape() as tape:
        predictions = model(images, training=True)
        loss = compute_loss(labels, predictions)

    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))

    train_accuracy.update_state(labels, predictions)
    return loss

def test_step(inputs):
    images, labels = inputs

    predictions = model(images, training=False)
    t_loss = loss_object(labels, predictions)

    test_loss.update_state(t_loss)
    test_accuracy.update_state(labels, predictions)
```

In []:

```
# `run` replicates the provided computation and runs it
# with the distributed input.
@tf.function
def distributed_train_step(dataset_inputs):
    per_replica_losses = strategy.run(train_step, args=(dataset_inputs,))
    return strategy.reduce(tf.distribute.ReduceOp.SUM, per_replica_losses,
                           axis=None)

@tf.function
def distributed_test_step(dataset_inputs):
    return strategy.run(test_step, args=(dataset_inputs,))

for epoch in range(EPOCHS):
    # TRAIN LOOP
    total_loss = 0.0
    num_batches = 0
    for x in train_dist_dataset:
        total_loss += distributed_train_step(x)
        num_batches += 1
    train_loss = total_loss / num_batches

    # TEST LOOP
    for x in test_dist_dataset:
        distributed_test_step(x)

    if epoch % 2 == 0:
        checkpoint.save(checkpoint_prefix)

    template = ("Epoch {}, Loss: {}, Accuracy: {}, Test Loss: {}, "
               "Test Accuracy: {}")
    print(template.format(epoch + 1, train_loss,
                          train_accuracy.result() * 100, test_loss.result(),
                          test_accuracy.result() * 100))

    test_loss.reset_states()
    train_accuracy.reset_states()
    test_accuracy.reset_states()
```

Things to note in the example above:

- Iterate over the `train_dist_dataset` and `test_dist_dataset` using a `for x in ...` construct.
- The scaled loss is the return value of the `distributed_train_step`. This value is aggregated across replicas using the `tf.distribute.Strategy.reduce` call and then across batches by summing the return value of the `tf.distribute.Strategy.reduce` calls.
- `tf.keras.Metrics` should be updated inside `train_step` and `test_step` that gets executed by `tf.distribute.Strategy.run`. * `tf.distribute.Strategy.run` returns results from each local replica in the strategy, and there are multiple ways to consume this result. You can do `tf.distribute.Strategy.reduce` to get an aggregated value. You can also do `tf.distribute.Strategy.experimental_local_results` to get the list of values contained in the result, one per local replica.

Restore the latest checkpoint and test

A model checkpointed with a `tf.distribute.Strategy` can be restored with or without a strategy.

In []:

```
eval_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(
    name='eval_accuracy')

new_model = create_model()
new_optimizer = tf.keras.optimizers.Adam()

test_dataset = tf.data.Dataset.from_tensor_slices((test_images, test_labels)).batch(GLOBAL_BATCH_SIZE)
```

In []:

```
@tf.function
def eval_step(images, labels):
    predictions = new_model(images, training=False)
    eval_accuracy(labels, predictions)
```

In []:

```
checkpoint = tf.train.Checkpoint(optimizer=new_optimizer, model=new_model)
checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))

for images, labels in test_dataset:
    eval_step(images, labels)

print('Accuracy after restoring the saved model without strategy: {}'.format(
    eval_accuracy.result() * 100))
```

Alternate ways of iterating over a dataset

Using iterators

If you want to iterate over a given number of steps and not through the entire dataset you can create an iterator using the `iter` call and explicitly call `next` on the iterator. You can choose to iterate over the dataset both inside and outside the `tf.function`. Here is a small snippet demonstrating iteration of the dataset outside the `tf.function` using an iterator.

In []:

```
for _ in range(EPOCHS):
    total_loss = 0.0
    num_batches = 0
    train_iter = iter(train_dist_dataset)

    for _ in range(10):
        total_loss += distributed_train_step(next(train_iter))
        num_batches += 1
    average_train_loss = total_loss / num_batches

    template = ("Epoch {}, Loss: {}, Accuracy: {}")
    print(template.format(epoch + 1, average_train_loss, train_accuracy.result() * 100))
    train_accuracy.reset_states()
```

Iterating inside a `tf.function`

You can also iterate over the entire input `train_dist_dataset` inside a `tf.function` using the `for x in ...` construct or by creating iterators like you did above. The example below demonstrates wrapping one epoch of training with a `@tf.function` decorator and iterating over `train_dist_dataset` inside the function.

In []:

```
@tf.function
def distributed_train_epoch(dataset):
    total_loss = 0.0
    num_batches = 0
    for x in dataset:
        per_replica_losses = strategy.run(train_step, args=(x,))
        total_loss += strategy.reduce(
            tf.distribute.ReduceOp.SUM, per_replica_losses, axis=None)
        num_batches += 1
    return total_loss / tf.cast(num_batches, dtype=tf.float32)

for epoch in range(EPOCHS):
    train_loss = distributed_train_epoch(train_dist_dataset)

    template = ("Epoch {}, Loss: {}, Accuracy: {}")
    print(template.format(epoch + 1, train_loss, train_accuracy.result() * 100))

    train_accuracy.reset_states()
```

Tracking training loss across replicas

Note: As a general rule, you should use `tf.keras.Metrics` to track per-sample values and avoid values that have been aggregated within a replica.

Because of the loss scaling computation that is carried out, it's not recommended to use `tf.metrics.Mean` to track the training loss across different replicas.

For example, if you run a training job with the following characteristics:

- Two replicas
- Two samples are processed on each replica
- Resulting loss values: [2, 3] and [4, 5] on each replica
- Global batch size = 4

With loss scaling, you calculate the per-sample value of loss on each replica by adding the loss values, and then dividing by the global batch size. In this case: $(2 + 3) / 4 = 1.25$ and $(4 + 5) / 4 = 2.25$.

If you use `tf.metrics.Mean` to track loss across the two replicas, the result is different. In this example, you end up with a `total` of 3.50 and `count` of 2, which results in `total / count = 1.75` when `result()` is called on the metric. Loss calculated with `tf.keras.Metrics` is scaled by an additional factor that is equal to the number of replicas in sync.

Guide and examples

Here are some examples for using distribution strategy with custom training loops:

1. [Distributed training guide \(../../guide/distributed_training\)](#)
2. [DenseNet \(\[https://github.com/tensorflow/examples/blob/master/tensorflow_examples/models/densenet/distributed_train.py\]\(https://github.com/tensorflow/examples/blob/master/tensorflow_examples/models/densenet/distributed_train.py\)\)](#) example using `MirroredStrategy`.
3. [BERT \(\[https://github.com/tensorflow/models/blob/master/official/nlp/bert/run_classifier.py\]\(https://github.com/tensorflow/models/blob/master/official/nlp/bert/run_classifier.py\)\)](#) example trained using `MirroredStrategy` and `TPUStrategy`. This example is particularly helpful for understanding how to load from a checkpoint and generate periodic checkpoints during distributed training etc.
4. [NCF \(\[https://github.com/tensorflow/models/blob/master/official/recommendation/ncf_keras_main.py\]\(https://github.com/tensorflow/models/blob/master/official/recommendation/ncf_keras_main.py\)\)](#) example trained using `MirroredStrategy` that can be enabled using the `keras_use_ctl` flag.
5. [NMT \(\[https://github.com/tensorflow/examples/blob/master/tensorflow_examples/models/nmt_with_attention/distributed_train.py\]\(https://github.com/tensorflow/examples/blob/master/tensorflow_examples/models/nmt_with_attention/distributed_train.py\)\)](#) example trained using `MirroredStrategy`.

More examples listed in the [Distribution strategy guide \(../../guide/distributed_training.ipynb#examples_and_tutorials\)](#).

Next steps

- Try out the new `tf.distribute.Strategy` API on your models.
- Visit the [Better performance with `tf.function` \(../../guide/function.ipynb\)](#) and [TensorFlow Profiler \(../../guide/profiler.md\)](#) guide to learn more about tools to optimize the performance of your TensorFlow models.
- The [Distributed training in TensorFlow \(../../guide/distributed_training.ipynb\)](#) guide provides an overview of the available distribution strategies.

Copyright 2020 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Distributed Input



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/distribute/input)

(<https://www.tensorflow.org/tutorials/distribute/input>) (<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/distribute/input.ipynb>) (<https://github.com>)



[Run in Google Colab](#)

(<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/distribute/input.ipynb>) (<https://github.com>)

The [tf.distribute](https://www.tensorflow.org/guide/distributed_training) (https://www.tensorflow.org/guide/distributed_training) APIs provide an easy way for users to scale their training from a single machine to multiple machines. When scaling their model, users also have to distribute their input across multiple devices. `tf.distribute` provides APIs using which you can automatically distribute your input across devices.

This guide will show you the different ways in which you can create distributed dataset and iterators using `tf.distribute` APIs. Additionally, the following topics will be covered:

- Usage, sharding and batching options when using `tf.distribute.Strategy.experimental_distribute_dataset` and `tf.distribute.Strategy.distribute_datasets_from_function`.
- Different ways in which you can iterate over the distributed dataset.
- Differences between `tf.distribute.Strategy.experimental_distribute_dataset`/`tf.distribute.Strategy.distribute_datasets_from_function` APIs and `tf.data` APIs as well any limitations that users may come across in their usage.

This guide does not cover usage of distributed input with Keras APIs.

Distributed datasets

To use `tf.distribute` APIs to scale, use `tf.data.Dataset` to represent their input. `tf.distribute` works efficiently with `tf.data.Dataset` —for example, via automatic prefetching onto each accelerator device and regular performance updates. If you have a use case for using something other than `tf.data.Dataset`, please refer to the [Tensor inputs section](#) in this guide. In a non-distributed training loop, first create a `tf.data.Dataset` instance and then iterate over the elements. For example:

In []:

```
import tensorflow as tf

# Helper libraries
import numpy as np
import os

print(tf.__version__)
```

In []:

```
# Simulate multiple CPUs with virtual devices
N_VIRTUAL_DEVICES = 2
physical_devices = tf.config.list_physical_devices("CPU")
tf.config.set_logical_device_configuration(
    physical_devices[0], [tf.config.LogicalDeviceConfiguration() for _ in range(N_VIRTUAL_DEVICES)])
```

In []:

```
print("Available devices:")
for i, device in enumerate(tf.config.list_logical_devices()):
    print("%d %s" % (i, device))
```

In []:

```
global_batch_size = 16
# Create a tf.data.Dataset object.
dataset = tf.data.Dataset.from_tensors(([1.], [1.])).repeat(100).batch(global_batch_size)

@tf.function
def train_step(inputs):
    features, labels = inputs
    return labels - 0.3 * features

# Iterate over the dataset using the for..in construct.
for inputs in dataset:
    print(train_step(inputs))
```

To allow users to use `tf.distribute` strategy with minimal changes to a user's existing code, two APIs were introduced which would distribute a `tf.data.Dataset` instance and return a distributed dataset object. A user could then iterate over this distributed dataset instance and train their model as before. Let us now look at the two APIs - `tf.distribute.Strategy.experimental_distribute_dataset` and `tf.distribute.Strategy.distribute_datasets_from_function` in more detail:

`tf.distribute.Strategy.experimental_distribute_dataset`

Usage

This API takes a `tf.data.Dataset` instance as input and returns a `tf.distribute.DistributedDataset` instance. You should batch the input dataset with a value that is equal to the global batch size. This global batch size is the number of samples that you want to process across all devices in 1 step. You can iterate over this distributed dataset in a Pythonic fashion or create an iterator using `iter`. The returned object is not a `tf.data.Dataset` instance and does not support any other APIs that transform or inspect the dataset in any way. This is the recommended API if you don't have specific ways in which you want to shard your input over different replicas.

In []:

```
global_batch_size = 16
mirrored_strategy = tf.distribute.MirroredStrategy()

dataset = tf.data.Dataset.from_tensors(([1., 1.])).repeat(100).batch(global_batch_size)
# Distribute input using the `experimental_distribute_dataset`.
dist_dataset = mirrored_strategy.experimental_distribute_dataset(dataset)
# 1 global batch of data fed to the model in 1 step.
print(next(iter(dist_dataset)))
```

Properties

Batching

`tf.distribute` rebatches the input `tf.data.Dataset` instance with a new batch size that is equal to the global batch size divided by the number of replicas in sync. The number of replicas in sync is equal to the number of devices that are taking part in the gradient allreduce during training. When a user calls `next` on the distributed iterator, a per replica batch size of data is returned on each replica. The rebatched dataset cardinality will always be a multiple of the number of replicas. Here are a couple of examples:

- `tf.data.Dataset.range(6).batch(4, drop_remainder=False)`
 - Without distribution:
 - Batch 1: [0, 1, 2, 3]
 - Batch 2: [4, 5]
 - With distribution over 2 replicas. The last batch ([4, 5]) is split between 2 replicas.
 - Batch 1:
 - Replica 1:[0, 1]
 - Replica 2:[2, 3]
 - Batch 2:
 - Replica 1: [4]
 - Replica 2: [5]
- `tf.data.Dataset.range(4).batch(4)`
 - Without distribution:
 - Batch 1: [0, 1, 2, 3]
 - With distribution over 5 replicas:
 - Batch 1:
 - Replica 1: [0]
 - Replica 2: [1]
 - Replica 3: [2]
 - Replica 4: [3]
 - Replica 5: []
- `tf.data.Dataset.range(8).batch(4)`
 - Without distribution:
 - Batch 1: [0, 1, 2, 3]
 - Batch 2: [4, 5, 6, 7]
 - With distribution over 3 replicas:
 - Batch 1:
 - Replica 1: [0, 1]
 - Replica 2: [2, 3]
 - Replica 3: []
 - Batch 2:
 - Replica 1: [4, 5]
 - Replica 2: [6, 7]
 - Replica 3: []

Note: The above examples only illustrate how a global batch is split on different replicas. It is not advisable to depend on the actual values that might end up on each replica as it can change depending on the implementation.

Rebatching the dataset has a space complexity that increases linearly with the number of replicas. This means that for the multi-worker training use case the input pipeline can run into OOM errors.

Sharding

`tf.distribute` also autoshards the input dataset in multi-worker training with `MultiWorkerMirroredStrategy` and `TPUStrategy`. Each dataset is created on the CPU device of the worker. Autosharding a dataset over a set of workers means that each worker is assigned a subset of the entire dataset (if the right `tf.data.experimental.AutoShardPolicy` is set). This is to ensure that at each step, a global batch size of non-overlapping dataset elements will be processed by each worker. Autosharding has a couple of different options that can be specified using `tf.data.experimental.DistributeOptions`. Note that there is no autosharding in multi-worker training with `ParameterServerStrategy`, and more information on dataset creation with this strategy can be found in the [ParameterServerStrategy tutorial \(parameter_server_training.ipynb\)](#).

In []:

```
dataset = tf.data.Dataset.from_tensors(([1.], [1.])).repeat(64).batch(16)
options = tf.data.Options()
options.experimental_distribute.auto_shard_policy = tf.data.experimental.AutoShardPolicy.DATA
dataset = dataset.with_options(options)
```

There are three different options that you can set for the `tf.data.experimental.AutoShardPolicy`:

- AUTO: This is the default option which means an attempt will be made to shard by FILE. The attempt to shard by FILE fails if a file-based dataset is not detected. `tf.distribute` will then fall back to sharding by DATA. Note that if the input dataset is file-based but the number of files is less than the number of workers, an `InvalidArgumentError` will be raised. If this happens, explicitly set the policy to `AutoShardPolicy.DATA`, or split your input source into smaller files such that number of files is greater than number of workers.
- FILE: This is the option if you want to shard the input files over all the workers. You should use this option if the number of input files is much larger than the number of workers and the data in the files is evenly distributed. The downside of this option is having idle workers if the data in the files is not evenly distributed. If the number of files is less than the number of workers, an `InvalidArgumentError` will be raised. If this happens, explicitly set the policy to `AutoShardPolicy.DATA`. For example, let us distribute 2 files over 2 workers with 1 replica each. File 1 contains [0, 1, 2, 3, 4, 5] and File 2 contains [6, 7, 8, 9, 10, 11]. Let the total number of replicas in sync be 2 and global batch size be 4.
 - Worker 0:
 - Batch 1 = Replica 1: [0, 1]
 - Batch 2 = Replica 1: [2, 3]
 - Batch 3 = Replica 1: [4]
 - Batch 4 = Replica 1: [5]
 - Worker 1:
 - Batch 1 = Replica 2: [6, 7]
 - Batch 2 = Replica 2: [8, 9]
 - Batch 3 = Replica 2: [10]
 - Batch 4 = Replica 2: [11]
- DATA: This will autoshard the elements across all the workers. Each of the workers will read the entire dataset and only process the shard assigned to it. All other shards will be discarded. This is generally used if the number of input files is less than the number of workers and you want better sharding of data across all workers. The downside is that the entire dataset will be read on each worker. For example, let us distribute 1 files over 2 workers. File 1 contains [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]. Let the total number of replicas in sync be 2.
 - Worker 0:
 - Batch 1 = Replica 1: [0, 1]
 - Batch 2 = Replica 1: [4, 5]
 - Batch 3 = Replica 1: [8, 9]
 - Worker 1:
 - Batch 1 = Replica 2: [2, 3]
 - Batch 2 = Replica 2: [6, 7]
 - Batch 3 = Replica 2: [10, 11]
- OFF: If you turn off autosharding, each worker will process all the data. For example, let us distribute 1 files over 2 workers. File 1 contains [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]. Let the total number of replicas in sync be 2. Then each worker will see the following distribution:
 - Worker 0:
 - Batch 1 = Replica 1: [0, 1]
 - Batch 2 = Replica 1: [2, 3]
 - Batch 3 = Replica 1: [4, 5]
 - Batch 4 = Replica 1: [6, 7]
 - Batch 5 = Replica 1: [8, 9]
 - Batch 6 = Replica 1: [10, 11]
 - Worker 1:
 - Batch 1 = Replica 2: [0, 1]
 - Batch 2 = Replica 2: [2, 3]
 - Batch 3 = Replica 2: [4, 5]
 - Batch 4 = Replica 2: [6, 7]
 - Batch 5 = Replica 2: [8, 9]
 - Batch 6 = Replica 2: [10, 11]

Prefetching

By default, `tf.distribute` adds a prefetch transformation at the end of the user provided `tf.data.Dataset` instance. The argument to the prefetch transformation which is `buffer_size` is equal to the number of replicas in sync.

`tf.distribute.Strategy.distribute_datasets_from_function`

Usage

This API takes an input function and returns a `tf.distribute.DistributedDataset` instance. The input function that users pass in has a `tf.distribute.InputContext` argument and should return a `tf.data.Dataset` instance. With this API, `tf.distribute` does not make any further changes to the user's `tf.data.Dataset` instance returned from the input function. It is the responsibility of the user to batch and shard the dataset. `tf.distribute` calls the input function on the CPU device of each of the workers. Apart from allowing users to specify their own batching and sharding logic, this API also demonstrates better scalability and performance compared to `tf.distribute.Strategy.experimental_distribute_dataset` when used for multi-worker training.

In []:

```
mirrored_strategy = tf.distribute.MirroredStrategy()

def dataset_fn(input_context):
    batch_size = input_context.get_per_replica_batch_size(global_batch_size)
    dataset = tf.data.Dataset.from_tensors(([1.], [1.])).repeat(64).batch(16)
    dataset = dataset.shard(
        input_context.num_input_pipelines, input_context.input_pipeline_id)
    dataset = dataset.batch(batch_size)
    dataset = dataset.prefetch(2) # This prefetches 2 batches per device.
    return dataset

dist_dataset = mirrored_strategy.distribute_datasets_from_function(dataset_fn)
```

Properties

Batching

The `tf.data.Dataset` instance that is the return value of the input function should be batched using the per replica batch size. The per replica batch size is the global batch size divided by the number of replicas that are taking part in sync training. This is because `tf.distribute` calls the input function on the CPU device of each of the workers. The dataset that is created on a given worker should be ready to use by all the replicas on that worker.

Sharding

The `tf.distribute.InputContext` object that is implicitly passed as an argument to the user's input function is created by `tf.distribute` under the hood. It has information about the number of workers, current worker ID etc. This input function can handle sharding as per policies set by the user using these properties that are part of the `tf.distribute.InputContext` object.

Prefetching

`tf.distribute` does not add a prefetch transformation at the end of the `tf.data.Dataset` returned by the user-provided input function, so you explicitly call `Dataset.prefetch` in the example above.

Note: Both `tf.distribute.Strategy.experimental_distribute_dataset` and `tf.distribute.Strategy.distribute_datasets_from_function` return **`tf.distribute.DistributedDataset` instances that are not of type `tf.data.Dataset`**. You can iterate over these instances (as shown in the Distributed Iterators section) and use the `element_spec` property.

Distributed iterators

Similar to non-distributed `tf.data.Dataset` instances, you will need to create an iterator on the `tf.distribute.DistributedDataset` instances to iterate over it and access the elements in the `tf.distribute.DistributedDataset`. The following are the ways in which you can create a `tf.distribute.DistributedIterator` and use it to train your model:

Usages

Use a Pythonic for loop construct

You can use a user friendly Pythonic loop to iterate over the `tf.distribute.DistributedDataset`. The elements returned from the `tf.distribute.DistributedIterator` can be a single `tf.Tensor` or a `tf.distribute.DistributedValues` which contains a value per replica. Placing the loop inside a `tf.function` will give a performance boost. However, `break` and `return` are currently not supported for a loop over a `tf.distribute.DistributedDataset` that is placed inside of a `tf.function`.

In []:

```
global_batch_size = 16
mirrored_strategy = tf.distribute.MirroredStrategy()

dataset = tf.data.Dataset.from_tensors(([1.], [1.])).repeat(100).batch(global_batch_size)
dist_dataset = mirrored_strategy.experimental_distribute_dataset(dataset)

@tf.function
def train_step(inputs):
    features, labels = inputs
    return labels - 0.3 * features

for x in dist_dataset:
    # train_step trains the model using the dataset elements
    loss = mirrored_strategy.run(train_step, args=(x,))
    print("Loss is ", loss)
```

Use `iter` to create an explicit iterator

To iterate over the elements in a `tf.distribute.DistributedDataset` instance, you can create a `tf.distribute.DistributedIterator` using the `iter` API on it. With an explicit iterator, you can iterate for a fixed number of steps. In order to get the next element from an `tf.distribute.DistributedIterator` instance `dist_iterator`, you can call `next(dist_iterator)`, `dist_iterator.get_next()`, or `dist_iterator.get_next_as_optional()`. The former two are essentially the same:

In []:

```
num_epochs = 10
steps_per_epoch = 5
for epoch in range(num_epochs):
    dist_iterator = iter(dist_dataset)
    for step in range(steps_per_epoch):
        # train_step trains the model using the dataset elements
        loss = mirrored_strategy.run(train_step, args=(next(dist_iterator),))
        # which is the same as
        # loss = mirrored_strategy.run(train_step, args=(dist_iterator.get_next(),))
        print("Loss is ", loss)
```

With `next` or `tf.distribute.DistributedIterator.get_next`, if the `tf.distribute.DistributedIterator` has reached its end, an `OutOfRange` error will be thrown. The client can catch the error on python side and continue doing other work such as checkpointing and evaluation. However, this will not work if you are using a host training loop (i.e., run multiple steps per `tf.function`), which looks like:

```
@tf.function
def train_fn(iterator):
    for _ in tf.range(steps_per_loop):
        strategy.run(step_fn, args=(next(iterator),))
```

This example `train_fn` contains multiple steps by wrapping the step body inside a `tf.range`. In this case, different iterations in the loop with no dependency could start in parallel, so an `OutOfRange` error can be triggered in later iterations before the computation of previous iterations finishes. Once an `OutOfRange` error is thrown, all the ops in the function will be terminated right away. If this is some case that you would like to avoid, an alternative that does not throw an `OutOfRange` error is `tf.distribute.DistributedIterator.get_next_as_optional`. `get_next_as_optional` returns a `tf.experimental.Optional` which contains the next element or no value if the `tf.distribute.DistributedIterator` has reached an end.

In []:

```
# You can break the loop with `get_next_as_optional` by checking if the `Optional` contains a value
global_batch_size = 4
steps_per_loop = 5
strategy = tf.distribute.MirroredStrategy()

dataset = tf.data.Dataset.range(9).batch(global_batch_size)
distributed_iterator = iter(strategy.experimental_distribute_dataset(dataset))

@tf.function
def train_fn(distributed_iterator):
    for _ in tf.range(steps_per_loop):
        optional_data = distributed_iterator.get_next_as_optional()
        if not optional_data.has_value():
            break
        per_replica_results = strategy.run(lambda x: x, args=(optional_data.get_value(),))
        tf.print(strategy.experimental_local_results(per_replica_results))
train_fn(distributed_iterator)
```

Using the element_spec property

If you pass the elements of a distributed dataset to a `tf.function` and want a `tf.TypeSpec` guarantee, you can specify the `input_signature` argument of the `tf.function`. The output of a distributed dataset is `tf.distribute.DistributedValues` which can represent the input to a single device or multiple devices. To get the `tf.TypeSpec` corresponding to this distributed value, you can use `tf.distribute.DistributedDataset.element_spec` or `tf.distribute.DistributedIterator.element_spec`.

In []:

```
global_batch_size = 16
epochs = 5
steps_per_epoch = 5
mirrored_strategy = tf.distribute.MirroredStrategy()

dataset = tf.data.Dataset.from_tensors(([1.], [1.])).repeat(100).batch(global_batch_size)
dist_dataset = mirrored_strategy.experimental_distribute_dataset(dataset)

@tf.function(input_signature=[dist_dataset.element_spec])
def train_step(per_replica_inputs):
    def step_fn(inputs):
        return 2 * inputs

    return mirrored_strategy.run(step_fn, args=(per_replica_inputs,))

for _ in range(epochs):
    iterator = iter(dist_dataset)
    for _ in range(steps_per_epoch):
        output = train_step(next(iterator))
        tf.print(output)
```

Data preprocessing

So far, you have learned how to distribute a `tf.data.Dataset`. Yet before the data is ready for the model, it needs to be preprocessed, for example by cleansing, transforming, and augmenting it. Two sets of those handy tools are:

- [Keras preprocessing layers \(\[https://www.tensorflow.org/guide/keras/preprocessing_layers\]\(https://www.tensorflow.org/guide/keras/preprocessing_layers\)\)](https://www.tensorflow.org/guide/keras/preprocessing_layers): a set of Keras layers that allow developers to build Keras-native input processing pipelines. Some Keras preprocessing layers contain non-trainable states, which can be set on initialization or `adapt`ed (refer to the `adapt` section of the [Keras preprocessing layers guide \(\[https://www.tensorflow.org/guide/keras/preprocessing_layers\]\(https://www.tensorflow.org/guide/keras/preprocessing_layers\)\)](https://www.tensorflow.org/guide/keras/preprocessing_layers)). When distributing stateful preprocessing layers, the states should be replicated to all workers. To use these layers, you can either make them part of the model or apply them to the datasets.
- [TensorFlow Transform \(tf.Transform\) \(\[https://www.tensorflow.org/tfx/transform/get_started\]\(https://www.tensorflow.org/tfx/transform/get_started\)\)](https://www.tensorflow.org/tfx/transform/get_started): a library for TensorFlow that allows you to define both instance-level and full-pass data transformation through data preprocessing pipelines. Tensorflow Transform has two phases. The first is the Analyze phase, where the raw training data is analyzed in a full-pass process to compute the statistics needed for the transformations, and the transformation logic is generated as instance-level operations. The second is the Transform phase, where the raw training data is transformed in an instance-level process.

Keras preprocessing layers vs. Tensorflow Transform

Both Tensorflow Transform and Keras preprocessing layers provide a way to split out preprocessing during training and bundle preprocessing with a model during inference, reducing train/serve skew.

Tensorflow Transform, deeply integrated with [TFX](https://www.tensorflow.org/tfx) (<https://www.tensorflow.org/tfx>), provides a scalable map-reduce solution to analyzing and transforming datasets of any size in a job separate from the training pipeline. If you need to run an analysis on a dataset that cannot fit on a single machine, Tensorflow Transform should be your first choice.

Keras preprocessing layers are more geared towards preprocessing applied during training, after reading data from disk. They fit seamlessly with model development in the Keras library. They support analysis of a smaller dataset via `adapt` (https://www.tensorflow.org/guide/keras/preprocessing_layers#the_adapt_method) and supports use cases like image data augmentation, where each pass over the input dataset will yield different examples for training.

The two libraries can also be mixed, where Tensorflow Transform is used for analysis and static transformations of input data, and Keras preprocessing layers are used for train-time transformations (e.g., one-hot encoding or data augmentation).

Best Practice with `tf.distribute`

Working with both tools involves initializing the transformation logic to apply to data, which might create Tensorflow resources. These resources or states should be replicated to all workers to save inter-workers or worker-coordinator communication. To do so, you are recommended to create Keras preprocessing layers, `tft.TFTransformOutput.transform_features_layer`, or `tft.TransformFeaturesLayer` under `tf.distribute.Strategy.scope`, just like you would for any other Keras layers.

The following examples demonstrate usage of the `tf.distribute.Strategy` API with the high-level Keras `Model.fit` API and with a custom training loop separately.

Extra notes for Keras preprocessing layers users:

Preprocessing layers and large vocabularies

When dealing with large vocabularies (over one gigabyte) in a multi-worker setting (for example, `tf.distribute.MultiWorkerMirroredStrategy`, `tf.distribute.experimental.ParameterServerStrategy`, `tf.distribute.TPUStrategy`), it is recommended to save the vocabulary to a static file accessible from all workers (for example, with Cloud Storage). This will reduce the time spent replicating the vocabulary to all workers during training.

Preprocessing in the `tf.data` pipeline versus in the model

While Keras preprocessing layers can be applied either as part of the model or directly to a `tf.data.Dataset`, each of the options come with their edge:

- Applying the preprocessing layers within the model makes your model portable, and it helps reduce the training/serving skew. (For more details, refer to the *Benefits of doing preprocessing inside the model at inference time* section in the [Working with preprocessing layers guide](https://www.tensorflow.org/guide/keras/preprocessing_layers#benefits_of_doing_preprocessing_inside_the_model_at_inference_time) (https://www.tensorflow.org/guide/keras/preprocessing_layers#benefits_of_doing_preprocessing_inside_the_model_at_inference_time))
- Applying within the `tf.data` pipeline allows prefetching or offloading to the CPU, which generally gives better performance when using accelerators.

When running on one or more TPUs, users should almost always place Keras preprocessing layers in the `tf.data` pipeline, as not all layers support TPUs, and string ops do not execute on TPUs. (The two exceptions are `tf.keras.layers.Normalization` and `tf.keras.layers.Rescaling`, which run fine on TPUs and are commonly used as the first layer in an image model.)

Model.fit

When using Keras `Model.fit`, you do not need to distribute data with `tf.distribute.Strategy.experimental_distribute_dataset` nor `tf.distribute.Strategy.distribute_datasets_from_function` themselves. Check out the [Working with preprocessing layers](https://www.tensorflow.org/guide/keras/preprocessing_layers) (https://www.tensorflow.org/guide/keras/preprocessing_layers) guide and the [Distributed training with Keras](https://www.tensorflow.org/tutorials/distribute/keras) (<https://www.tensorflow.org/tutorials/distribute/keras>) guide for details. A shortened example may look as below:

```
strategy = tf.distribute.MirroredStrategy()
with strategy.scope():
    # Create the layer(s) under scope.
    integer_preprocessing_layer = tf.keras.layers.IntegerLookup(vocabulary=FILE_PATH)
    model = ...
    model.compile(...)
dataset = dataset.map(lambda x, y: (integer_preprocessing_layer(x), y))
model.fit(dataset)
```

Users of `tf.distribute.experimental.ParameterServerStrategy` with the `Model.fit` API need to use a `tf.keras.utils.experimental.DatasetCreator` as the input. (See the [Parameter Server Training](https://www.tensorflow.org/tutorials/distribute/parameter_server_training#parameter_server_training_with_modelfit_api) (https://www.tensorflow.org/tutorials/distribute/parameter_server_training#parameter_server_training_with_modelfit_api) guide for more)

```
strategy = tf.distribute.experimental.ParameterServerStrategy(  
    cluster_resolver,  
    variable_partitioner=variable_partitioner)  
  
with strategy.scope():  
    preprocessing_layer = tf.keras.layers.StringLookup(vocabulary=FILE_PATH)  
    model = ...  
    model.compile(...)  
  
def dataset_fn(input_context):  
    ...  
    dataset = dataset.map(preprocessing_layer)  
    ...  
    return dataset  
  
dataset_creator = tf.keras.utils.experimental.DatasetCreator(dataset_fn)  
model.fit(dataset_creator, epochs=5, steps_per_epoch=20, callbacks=callbacks)
```

Custom Training Loop

When writing a [custom training loop](https://www.tensorflow.org/tutorials/distribute/custom_training) (https://www.tensorflow.org/tutorials/distribute/custom_training), you will distribute your data with either the `tf.distribute.Strategy.experimental_distribute_dataset` API or the `tf.distribute.Strategy.distribute_datasets_from_function` API. If you distribute your dataset through `tf.distribute.Strategy.experimental_distribute_dataset`, applying these preprocessing APIs in your data pipeline will lead the resources automatically co-located with the data pipeline to avoid remote resource access. Thus the examples here will all use `tf.distribute.Strategy.distribute_datasets_from_function`, in which case it is crucial to place initialization of these APIs under `strategy.scope()` for efficiency:

In []:

```
strategy = tf.distribute.MirroredStrategy()  
vocab = ["a", "b", "c", "d", "f"]  
  
with strategy.scope():  
    # Create the layer(s) under scope.  
    layer = tf.keras.layers.StringLookup(vocabulary=vocab)  
  
def dataset_fn(input_context):  
    # a tf.data.Dataset  
    dataset = tf.data.Dataset.from_tensor_slices(["a", "c", "e"]).repeat()  
  
    # Custom your batching, sharding, prefetching, etc.  
    global_batch_size = 4  
    batch_size = input_context.get_per_replica_batch_size(global_batch_size)  
    dataset = dataset.batch(batch_size)  
    dataset = dataset.shard(  
        input_context.num_input_pipelines,  
        input_context.input_pipeline_id)  
  
    # Apply the preprocessing layer(s) to the tf.data.Dataset  
    def preprocess_with_kpl(input):  
        return layer(input)  
  
    processed_ds = dataset.map(preprocess_with_kpl)  
    return processed_ds  
  
distributed_dataset = strategy.distribute_datasets_from_function(dataset_fn)  
  
# Print out a few example batches.  
distributed_dataset_iterator = iter(distributed_dataset)  
for _ in range(3):  
    print(next(distributed_dataset_iterator))
```

Note that if you are training with `tf.distribute.experimental.ParameterServerStrategy`, you'll also call `tf.distribute.experimental.coordinator.ClusterCoordinator.create_per_worker_dataset`

```
@tf.function
def per_worker_dataset_fn():
    return strategy.distribute_datasets_from_function(dataset_fn)

per_worker_dataset = coordinator.create_per_worker_dataset(per_worker_dataset_fn)
per_worker_iterator = iter(per_worker_dataset)
```

For Tensorflow Transform, as mentioned above, the Analyze stage is done separately from training and thus omitted here. See the [tutorial](https://www.tensorflow.org/tfx/tutorials/transform/census) (<https://www.tensorflow.org/tfx/tutorials/transform/census>) for a detailed how-to. Usually, this stage includes creating a `tf.Transform` preprocessing function and transforming the data in an [Apache Beam](https://beam.apache.org/) (<https://beam.apache.org/>) pipeline with this preprocessing function. At the end of the Analyze stage, the output can be exported as a TensorFlow graph which you can use for both training and serving. Our example covers only the training pipeline part:

```
with strategy.scope():
    # working_dir contains the tf.Transform output.
    tf_transform_output = tft.TFTransformOutput(working_dir)
    # Loading from working_dir to create a Keras layer for applying the tf.Transform output to data
    tft_layer = tf_transform_output.transform_features_layer()
    ...

def dataset_fn(input_context):
    ...
    dataset.map(tft_layer, num_parallel_calls=tf.data.AUTOTUNE)
    ...
    return dataset

distributed_dataset = strategy.distribute_datasets_from_function(dataset_fn)
```

Partial batches

Partial batches are encountered when: 1) `tf.data.Dataset` instances that users create may contain batch sizes that are not evenly divisible by the number of replicas; or 2) when the cardinality of the dataset instance is not divisible by the batch size. This means that when the dataset is distributed over multiple replicas, the `next` call on some iterators will result in an `tf.errors.OutOfRangeError`. To handle this use case, `tf.distribute` returns dummy batches of batch size `0` on replicas that do not have any more data to process.

For the single-worker case, if the data is not returned by the `next` call on the iterator, dummy batches of 0 batch size are created and used along with the real data in the dataset. In the case of partial batches, the last global batch of data will contain real data alongside dummy batches of data. The stopping condition for processing data now checks if any of the replicas have data. If there is no data on any of the replicas, you will get a `tf.errors.OutOfRangeError`.

For the multi-worker case, the boolean value representing presence of data on each of the workers is aggregated using cross replica communication and this is used to identify if all the workers have finished processing the distributed dataset. Since this involves cross worker communication there is some performance penalty involved.

Caveats

- When using `tf.distribute.Strategy.experimental_distribute_dataset` APIs with a multi-worker setup, you pass a `tf.data.Dataset` that reads from files. If the `tf.data.experimental.AutoShardPolicy` is set to `AUTO` or `FILE`, the actual per-step batch size may be smaller than the one you defined for the global batch size. This can happen when the remaining elements in the file are less than the global batch size. You can either exhaust the dataset without depending on the number of steps to run, or set `tf.data.experimental.AutoShardPolicy` to `DATA` to work around it.
- Stateful dataset transformations are currently not supported with `tf.distribute` and any stateful ops that the dataset may have are currently ignored. For example, if your dataset has a `map_fn` that uses `tf.random.uniform` to rotate an image, then you have a dataset graph that depends on state (i.e the random seed) on the local machine where the python process is being executed.
- Experimental `tf.data.experimental.OptimizationOptions` that are disabled by default can in certain contexts—such as when used together with `tf.distribute`—cause a performance degradation. You should only enable them after you validate that they benefit the performance of your workload in a distribute setting.
- Please refer to [this guide](https://www.tensorflow.org/guide/data_performance) (https://www.tensorflow.org/guide/data_performance) for how to optimize your input pipeline with `tf.data` in general. A few additional tips:
 - If you have multiple workers and are using `tf.data.Dataset.list_files` to create a dataset from all files matching one or more glob patterns, remember to set the `seed` argument or set `shuffle=False` so that each worker shard the file consistently.
 - If your input pipeline includes both shuffling the data on record level and parsing the data, unless the unparsed data is significantly larger than the parsed data (which is usually not the case), shuffle first and then parse, as shown in the following example. This may benefit memory usage and performance.

```
d = tf.data.Dataset.list_files(pattern, shuffle=False)
d = d.shard(num_workers, worker_index)
d = d.repeat(num_epochs)
d = d.shuffle(shuffle_buffer_size)
d = d.interleave(tf.data.TFRecordDataset,
                 cycle_length=num_readers, block_length=1)
d = d.map(parser_fn, num_parallel_calls=num_map_threads)
```

- `tf.data.Dataset.shuffle(buffer_size, seed=None, reshuffle_each_iteration=None)` maintain an internal buffer of `buffer_size` elements, and thus reducing `buffer_size` could alleviate OOM issue.
- The order in which the data is processed by the workers when using `tf.distribute.experimental_distribute_dataset` or `tf.distribute.distribute_datasets_from_function` is not guaranteed. This is typically required if you are using `tf.distribute` to scale prediction. You can however insert an index for each element in the batch and order outputs accordingly. The following snippet is an example of how to order outputs.

Note: `tf.distribute.MirroredStrategy` is used here for the sake of convenience. You only need to reorder inputs when you are using multiple workers, but `tf.distribute.MirroredStrategy` is used to distribute training on a single worker.

In []:

```
mirrored_strategy = tf.distribute.MirroredStrategy()
dataset_size = 24
batch_size = 6
dataset = tf.data.Dataset.range(dataset_size).enumerate().batch(batch_size)
dist_dataset = mirrored_strategy.experimental_distribute_dataset(dataset)

def predict(index, inputs):
    outputs = 2 * inputs
    return index, outputs

result = {}
for index, inputs in dist_dataset:
    output_index, outputs = mirrored_strategy.run(predict, args=(index, inputs))
    indices = list(mirrored_strategy.experimental_local_results(output_index))
    rindices = []
    for a in indices:
        rindices.extend(a.numpy())
    outputs = list(mirrored_strategy.experimental_local_results(outputs))
    routputs = []
    for a in outputs:
        routputs.extend(a.numpy())
    for i, value in zip(rindices, routputs):
        result[i] = value

print(result)
```

Tensor inputs instead of `tf.data`

Sometimes users cannot use a `tf.data.Dataset` to represent their input and subsequently the above mentioned APIs to distribute the dataset to multiple devices. In such cases you can use raw tensors or inputs from a generator.

Use `experimental_distribute_values_from_function` for arbitrary tensor inputs

`strategy.run` accepts `tf.distribute.DistributedValues` which is the output of `next(iterator)`. To pass the tensor values, use `tf.distribute.Strategy.experimental_distribute_values_from_function` to construct `tf.distribute.DistributedValues` from raw tensors. The user will have to specify their own batching and sharding logic in the input function with this option, which can be done using the `tf.distribute.experimental.ValueContext` input object.

In []:

```
mirrored_strategy = tf.distribute.MirroredStrategy()

def value_fn(ctx):
    return tf.constant(ctx.replica_id_in_sync_group)

distributed_values = mirrored_strategy.experimental_distribute_values_from_function(value_fn)
for _ in range(4):
    result = mirrored_strategy.run(lambda x: x, args=(distributed_values,))
    print(result)
```

Use `tf.data.Dataset.from_generator` if your input is from a generator

If you have a generator function that you want to use, you can create a `tf.data.Dataset` instance using the `from_generator` API.

Note: This is currently not supported for `tf.distribute.TPUStrategy`.

In []:

```
mirrored_strategy = tf.distribute.MirroredStrategy()
def input_gen():
    while True:
        yield np.random.rand(4)

# use Dataset.from_generator
dataset = tf.data.Dataset.from_generator(
    input_gen, output_types=tf.float32, output_shapes=tf.TensorShape([4]))
dist_dataset = mirrored_strategy.experimental_distribute_dataset(dataset)
iterator = iter(dist_dataset)
for _ in range(4):
    result = mirrored_strategy.run(lambda x: x, args=(next(iterator),))
    print(result)
```

Copyright 2019 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Save and load a model using a distribution strategy



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/distribute/save_and_load)

(https://www.tensorflow.org/tutorials/distribute/save_and_load)



[Run in Google Colab](#)

(https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/distribute/save_and_load.ipynb)

Overview

This tutorial demonstrates how you can save and load models in a SavedModel format with `tf.distribute.Strategy` during or after training. There are two kinds of APIs for saving and loading a Keras model: high-level (`tf.keras.Model.save` and `tf.keras.models.load_model`) and low-level (`tf.saved_model.save` and `tf.saved_model.load`).

To learn about SavedModel and serialization in general, please read the [saved model guide \(..../guide/saved_model.ipynb\)](#), and the [Keras model serialization guide \(https://www.tensorflow.org/guide/keras/save_and_serialize\)](#). Let's start with a simple example:

Import dependencies:

In []:

```
import tensorflow_datasets as tfds  
import tensorflow as tf
```

Load and prepare the data with TensorFlow Datasets and `tf.data`, and create the model using `tf.distribute.MirroredStrategy`:

In []:

```
mirrored_strategy = tf.distribute.MirroredStrategy()  
  
def get_data():  
    datasets = tfds.load(name='mnist', as_supervised=True)  
    mnist_train, mnist_test = datasets['train'], datasets['test']  
  
    BUFFER_SIZE = 10000  
  
    BATCH_SIZE_PER_REPLICA = 64  
    BATCH_SIZE = BATCH_SIZE_PER_REPLICA * mirrored_strategy.num_replicas_in_sync  
  
    def scale(image, label):  
        image = tf.cast(image, tf.float32)  
        image /= 255  
  
        return image, label  
  
    train_dataset = mnist_train.map(scale).cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE)  
    eval_dataset = mnist_test.map(scale).batch(BATCH_SIZE)  
  
    return train_dataset, eval_dataset  
  
def get_model():  
    with mirrored_strategy.scope():  
        model = tf.keras.Sequential([  
            tf.keras.layers.Conv2D(32, 3, activation='relu', input_shape=(28, 28, 1)),  
            tf.keras.layers.MaxPooling2D(),  
            tf.keras.layers.Flatten(),  
            tf.keras.layers.Dense(64, activation='relu'),  
            tf.keras.layers.Dense(10)  
        ])  
  
        model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),  
                      optimizer=tf.keras.optimizers.Adam(),  
                      metrics=[tf.metrics.SparseCategoricalAccuracy()])  
    return model
```

Train the model with `tf.keras.Model.fit`:

In []:

```
model = get_model()  
train_dataset, eval_dataset = get_data()  
model.fit(train_dataset, epochs=2)
```

Save and load the model

Now that you have a simple model to work with, let's explore the saving/loading APIs. There are two kinds of APIs available:

- High-level (Keras): `Model.save` and `tf.keras.models.load_model`
- Low-level: `tf.saved_model.save` and `tf.saved_model.load`

The Keras API

Here is an example of saving and loading a model with the Keras API:

In []:

```
keras_model_path = '/tmp/keras_save'  
model.save(keras_model_path)
```

Restore the model without `tf.distribute.Strategy`:

In []:

```
restored_keras_model = tf.keras.models.load_model(keras_model_path)  
restored_keras_model.fit(train_dataset, epochs=2)
```

After restoring the model, you can continue training on it, even without needing to call `Model.compile` again, since it was already compiled before saving. The model is saved in TensorFlow's standard `SavedModel` proto format. For more information, please refer to [the guide to SavedModel format](#) ([../guide/saved_model.ipynb](#)).

Now, restore the model and train it using a `tf.distribute.Strategy`:

In []:

```
another_strategy = tf.distribute.OneDeviceStrategy('/cpu:0')  
with another_strategy.scope():  
    restored_keras_model_ds = tf.keras.models.load_model(keras_model_path)  
    restored_keras_model_ds.fit(train_dataset, epochs=2)
```

As the `Model.fit` output shows, loading works as expected with `tf.distribute.Strategy`. The strategy used here does not have to be the same strategy used before saving.

The `tf.saved_model` API

Saving the model with lower-level API is similar to the Keras API:

In []:

```
model = get_model() # get a fresh model  
saved_model_path = '/tmp/tf_save'  
tf.saved_model.save(model, saved_model_path)
```

Loading can be done with `tf.saved_model.load`. However, since it is a lower-level API (and hence has a wider range of use cases), it does not return a Keras model. Instead, it returns an object that contain functions that can be used to do inference. For example:

In []:

```
DEFAULT_FUNCTION_KEY = 'serving_default'  
loaded = tf.saved_model.load(saved_model_path)  
inference_func = loaded.signatures[DEFAULT_FUNCTION_KEY]
```

The loaded object may contain multiple functions, each associated with a key. The "serving_default" key is the default key for the inference function with a saved Keras model. To do inference with this function:

In []:

```
predict_dataset = eval_dataset.map(lambda image, label: image)  
for batch in predict_dataset.take(1):  
    print(inference_func(batch))
```

You can also load and do inference in a distributed manner:

In []:

```
another_strategy = tf.distribute.MirroredStrategy()
with another_strategy.scope():
    loaded = tf.saved_model.load(saved_model_path)
    inference_func = loaded.signatures[DEFAULT_FUNCTION_KEY]

    dist_predict_dataset = another_strategy.experimental_distribute_dataset(
        predict_dataset)

    # Calling the function in a distributed manner
    for batch in dist_predict_dataset:
        result = another_strategy.run(inference_func, args=(batch,))
        print(result)
        break
```

Calling the restored function is just a forward pass on the saved model (`tf.keras.Model.predict`). What if you want to continue training the loaded function? Or what if you need to embed the loaded function into a bigger model? A common practice is to wrap this loaded object into a Keras layer to achieve this. Luckily, [TF Hub \(https://www.tensorflow.org/hub\)](https://www.tensorflow.org/hub) has `hub.KerasLayer`

(https://github.com/tensorflow/hub/blob/master/tensorflow_hub/keras_layer.py) for this purpose, shown here:

In []:

```
import tensorflow_hub as hub

def build_model.loaded():
    x = tf.keras.layers.Input(shape=(28, 28, 1), name='input_x')
    # Wrap what's loaded to a KerasLayer
    keras_layer = hub.KerasLayer(loaded, trainable=True)(x)
    model = tf.keras.Model(x, keras_layer)
    return model

another_strategy = tf.distribute.MirroredStrategy()
with another_strategy.scope():
    loaded = tf.saved_model.load(saved_model_path)
    model = build_model.loaded()

    model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                  optimizer=tf.keras.optimizers.Adam(),
                  metrics=[tf.metrics.SparseCategoricalAccuracy()])
    model.fit(train_dataset, epochs=2)
```

In the above example, Tensorflow Hub's `hub.KerasLayer` wraps the result loaded back from `tf.saved_model.load` into a Keras layer that is used to build another model. This is very useful for transfer learning.

Which API should I use?

For saving, if you are working with a Keras model, use the Keras `Model.save` API unless you need the additional control allowed by the low-level API. If what you are saving is not a Keras model, then the lower-level API, `tf.saved_model.save`, is your only choice.

For loading, your API choice depends on what you want to get from the model loading API. If you cannot (or do not want to) get a Keras model, then use `tf.saved_model.load`. Otherwise, use `tf.keras.models.load_model`. Note that you can get a Keras model back only if you saved a Keras model.

It is possible to mix and match the APIs. You can save a Keras model with `Model.save`, and load a non-Keras model with the low-level API, `tf.saved_model.load`.

In []:

```
model = get_model()

# Saving the model using Keras `Model.save`
model.save(keras_model_path)

another_strategy = tf.distribute.MirroredStrategy()
# Loading the model using the lower-level API
with another_strategy.scope():
    loaded = tf.saved_model.load(keras_model_path)
```

Saving/Loading from a local device

When saving and loading from a local I/O device while training on remote devices—for example, when using a Cloud TPU—you must use the option `experimental_io_device` in `tf.saved_model.SaveOptions` and `tf.saved_model.LoadOptions` to set the I/O device to `localhost`. For example:

In []:

```
model = get_model()

# Saving the model to a path on localhost.
saved_model_path = '/tmp/tf_save'
save_options = tf.saved_model.SaveOptions(experimental_io_device='/job:localhost')
model.save(saved_model_path, options=save_options)

# Loading the model from a path on localhost.
another_strategy = tf.distribute.MirroredStrategy()
with another_strategy.scope():
    load_options = tf.saved_model.LoadOptions(experimental_io_device='/job:localhost')
    loaded = tf.keras.models.load_model(saved_model_path, options=load_options)
```

Caveats

One special case is when you create Keras models in certain ways, and then save them before training. For example:

In []:

```
class SubclassedModel(tf.keras.Model):
    """Example model defined by subclassing `tf.keras.Model`."""

    output_name = 'output_layer'

    def __init__(self):
        super(SubclassedModel, self).__init__()
        self._dense_layer = tf.keras.layers.Dense(
            5, dtype=tf.dtypes.float32, name=self.output_name)

    def call(self, inputs):
        return self._dense_layer(inputs)

my_model = SubclassedModel()
try:
    my_model.save(keras_model_path)
except ValueError as e:
    print(f'{type(e).__name__}: ', *e.args)
```

A SavedModel saves the `tf.types.experimental.ConcreteFunction` objects generated when you trace a `tf.function` (check *When is a Function tracing?* in the [Introduction to graphs and tf.function \(..../guide/intro_to_graphs.ipynb\)](#) guide to learn more). If you get a `ValueError` like this it's because `Model.save` was not able to find or create a traced `ConcreteFunction`.

Caution: You shouldn't save a model without at least one `ConcreteFunction`, since the low-level API will otherwise generate a SavedModel with no `ConcreteFunction` signatures ([learn more \(..../guide/saved_model.ipynb\)](#) about the SavedModel format). For example:

In []:

```
tf.saved_model.save(my_model, saved_model_path)
x = tf.saved_model.load(saved_model_path)
x.signatures
```

Usually the model's forward pass—the `call` method—will be traced automatically when the model is called for the first time, often via the Keras `Model.fit` method. A `ConcreteFunction` can also be generated by the Keras [Sequential](#) (https://www.tensorflow.org/guide/keras/sequential_model) and [Functional](#) (<https://www.tensorflow.org/guide/keras/functional>) APIs, if you set the input shape, for example, by making the first layer either a `tf.keras.layers.InputLayer` or another layer type, and passing it the `input_shape` keyword argument.

To verify if your model has any traced `ConcreteFunction`s, check if `Model.save_spec` is `None`:

In []:

```
print(my_model.save_spec() is None)
```

Let's use `tf.keras.Model.fit` to train the model, and notice that the `save_spec` gets defined and model saving will work:

In []:

```
BATCH_SIZE_PER_REPLICA = 4
BATCH_SIZE = BATCH_SIZE_PER_REPLICA * mirrored_strategy.num_replicas_in_sync

dataset_size = 100
dataset = tf.data.Dataset.from_tensors(
    (tf.range(5, dtype=tf.float32), tf.range(5, dtype=tf.float32)))
    .repeat(dataset_size).batch(BATCH_SIZE)

my_model.compile(optimizer='adam', loss='mean_squared_error')
my_model.fit(dataset, epochs=2)

print(my_model.save_spec() is None)
my_model.save(keras_model_path)
```

Copyright 2019 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Multi-worker training with Estimator



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/distribute/multi_worker_with_estimator)

(https://www.tensorflow.org/tutorials/distribute/multi_worker_with_estimator) (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/distribute/multi_

Warning: Estimators are not recommended for new code. Estimators run v1.Session -style code which is more difficult to write correctly, and can behave unexpectedly, especially when combined with TF 2 code. Estimators do fall under [compatibility guarantees](https://tensorflow.org/guide/compatibility) (<https://tensorflow.org/guide/versions>), but will receive no fixes other than security vulnerabilities. See the [migration guide](https://tensorflow.org/guide/migrate) (<https://tensorflow.org/guide/migrate>) for details.

Overview

Note: While you can use Estimators with `tf.distribute` API, it's recommended to use Keras with `tf.distribute`, see [multi-worker training with Keras \(multi_worker_with_keras.ipynb\)](#). Estimator training with `tf.distribute.Strategy` has limited support.

This tutorial demonstrates how `tf.distribute.Strategy` can be used for distributed multi-worker training with `tf.estimator`. If you write your code using `tf.estimator`, and you're interested in scaling beyond a single machine with high performance, this tutorial is for you.

Before getting started, please read the [distribution strategy \(./guide/distributed_training.ipynb\)](#) guide. The [multi-GPU training tutorial \(./keras.ipynb\)](#) is also relevant, because this tutorial uses the same model.

Setup

First, setup TensorFlow and the necessary imports.

In []:

```
import tensorflow_datasets as tfds
import tensorflow as tf

import os, json
```

Note: Starting from TF2.4 multi worker mirrored strategy fails with estimators if run with eager enabled (the default). The error in TF2.4 is `TypeError: cannot pickle '_thread.lock' object`, See [issue #46556](https://github.com/tensorflow/tensorflow/issues/46556) (<https://github.com/tensorflow/tensorflow/issues/46556>) for details. The workaround is to disable eager execution.

In []:

```
tf.compat.v1.disable_eager_execution()
```

Input function

This tutorial uses the MNIST dataset from [TensorFlow Datasets](https://www.tensorflow.org/datasets) (<https://www.tensorflow.org/datasets>). The code here is similar to the [multi-GPU training tutorial](#) ([./keras.ipynb](#)) with one key difference: when using Estimator for multi-worker training, it is necessary to shard the dataset by the number of workers to ensure model convergence. The input data is sharded by worker index, so that each worker processes `1/num_workers` distinct portions of the dataset.

In []:

```
BUFFER_SIZE = 10000
BATCH_SIZE = 64

def input_fn(mode, input_context=None):
    datasets, info = tfds.load(name='mnist',
                               with_info=True,
                               as_supervised=True)
    mnist_dataset = (datasets['train'] if mode == tf.estimator.ModeKeys.TRAIN else
                     datasets['test'])

    def scale(image, label):
        image = tf.cast(image, tf.float32)
        image /= 255
        return image, label

    if input_context:
        mnist_dataset = mnist_dataset.shard(input_context.num_input_pipelines,
                                             input_context.input_pipeline_id)
    return mnist_dataset.map(scale).cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
```

Another reasonable approach to achieve convergence would be to shuffle the dataset with distinct seeds at each worker.

Multi-worker configuration

One of the key differences in this tutorial (compared to the [multi-GPU training tutorial](#) ([./keras.ipynb](#))) is the multi-worker setup. The `TF_CONFIG` environment variable is the standard way to specify the cluster configuration to each worker that is part of the cluster.

There are two components of `TF_CONFIG`: `cluster` and `task`. `cluster` provides information about the entire cluster, namely the workers and parameter servers in the cluster. `task` provides information about the current task. The first component `cluster` is the same for all workers and parameter servers in the cluster, and the second component `task` is different on each worker and parameter server and specifies its own `type` and `index`. In this example, the task `type` is `worker` and the task `index` is `0`.

For illustration purposes, this tutorial shows how to set a `TF_CONFIG` with 2 workers on `localhost`. In practice, you would create multiple workers on an external IP address and port, and set `TF_CONFIG` on each worker appropriately, i.e. modify the task `index`.

Warning: *Do not execute the following code in Colab*. TensorFlow's runtime will attempt to create a gRPC server at the specified IP address and port, which will likely fail. See the [keras version](#) ([multi_worker_with_keras.ipynb](#)) of this tutorial for an example of how you can test run multiple workers on a single machine.

```
os.environ['TF_CONFIG'] = json.dumps({
    'cluster': {
        'worker': ["localhost:12345", "localhost:23456"]
    },
    'task': {'type': 'worker', 'index': 0}
})
```

Define the model

Write the layers, the optimizer, and the loss function for training. This tutorial defines the model with Keras layers, similar to the [multi-GPU training tutorial](#) ([./keras.ipynb](#)).

In []:

```
LEARNING_RATE = 1e-4
def model_fn(features, labels, mode):
    model = tf.keras.Sequential([
        tf.keras.layers.Conv2D(32, 3, activation='relu', input_shape=(28, 28, 1)),
        tf.keras.layers.MaxPooling2D(),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(10)
    ])
    logits = model(features, training=False)

    if mode == tf.estimator.ModeKeys.PREDICT:
        predictions = {'logits': logits}
        return tf.estimator.EstimatorSpec(labels=labels, predictions=predictions)

    optimizer = tf.compat.v1.train.GradientDescentOptimizer(
        learning_rate=LEARNING_RATE)
    loss = tf.keras.losses.SparseCategoricalCrossentropy(
        from_logits=True, reduction=tf.keras.losses.Reduction.NONE)(labels, logits)
    loss = tf.reduce_mean(loss) * (1. / BATCH_SIZE)
    if mode == tf.estimator.ModeKeys.EVAL:
        return tf.estimator.EstimatorSpec(mode, loss=loss)

    return tf.estimator.EstimatorSpec(
        mode=mode,
        loss=loss,
        train_op=optimizer.minimize(
            loss, tf.compat.v1.train.get_or_create_global_step()))
```

Note: Although the learning rate is fixed in this example, in general it may be necessary to adjust the learning rate based on the global batch size.

MultiWorkerMirroredStrategy

To train the model, use an instance of `tf.distribute.experimental.MultiWorkerMirroredStrategy`. `MultiWorkerMirroredStrategy` creates copies of all variables in the model's layers on each device across all workers. It uses `CollectiveOps`, a TensorFlow op for collective communication, to aggregate gradients and keep the variables in sync. The [tf.distribute.Strategy guide \(../guide/distributed_training.ipynb\)](#) has more details about this strategy.

In []:

```
strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy()
```

Train and evaluate the model

Next, specify the distribution strategy in the `RunConfig` for the estimator, and train and evaluate by invoking `tf.estimator.train_and_evaluate`. This tutorial distributes only the training by specifying the strategy via `train_distribute`. It is also possible to distribute the evaluation via `eval_distribute`.

In []:

```
config = tf.estimator.RunConfig(train_distribute=strategy)

classifier = tf.estimator.Estimator(
    model_fn=model_fn, model_dir='/tmp/multiworker', config=config)
tf.estimator.train_and_evaluate(
    classifier,
    train_spec=tf.estimator.TrainSpec(input_fn=input_fn),
    eval_spec=tf.estimator.EvalSpec(input_fn=input_fn))
)
```

Optimize training performance

You now have a model and a multi-worker capable Estimator powered by `tf.distribute.Strategy`. You can try the following techniques to optimize performance of multi-worker training:

- *Increase the batch size*: The batch size specified here is per-GPU. In general, the largest batch size that fits the GPU memory is advisable.
- *Cast variables*: Cast the variables to `tf.float` if possible. The official ResNet model includes [an example](https://github.com/tensorflow/models/blob/8367cf6dabe11adf7628541706b660821f397dce/official/resnet/resnet_model.py#L466) (https://github.com/tensorflow/models/blob/8367cf6dabe11adf7628541706b660821f397dce/official/resnet/resnet_model.py#L466) of how this can be done.
- *Use collective communication*: `MultiWorkerMirroredStrategy` provides multiple [collective communication implementations](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/distribute/cross_device_ops.py) (https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/distribute/cross_device_ops.py).
 - RING implements ring-based collectives using gRPC as the cross-host communication layer.
 - NCCL uses [Nvidia's NCCL](https://developer.nvidia.com/nccl) (<https://developer.nvidia.com/nccl>) to implement collectives.
 - AUTO defers the choice to the runtime.

The best choice of collective implementation depends upon the number and kind of GPUs, and the network interconnect in the cluster. To override the automatic choice, specify a valid value to the `communication` parameter of `MultiWorkerMirroredStrategy`'s constructor, e.g.
`communication=tf.distribute.experimental.CollectiveCommunication.NCCL`.

Visit the [Performance section](#) ([../guide/function.ipynb](#)) in the guide to learn more about other strategies and [tools](#) ([../guide/profiler.md](#)) you can use to optimize the performance of your TensorFlow models.

Other code examples

1. [End to end example](#) (https://github.com/tensorflow/ecosystem/tree/master/distribution_strategy) for multi worker training in tensorflow/ecosystem using Kubernetes templates. This example starts with a Keras model and converts it to an Estimator using the `tf.keras.estimator.model_to_estimator` API.
2. [Official models](#) (<https://github.com/tensorflow/models/tree/master/official>), many of which can be configured to run multiple distribution strategies.

Copyright 2019 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Multi-worker training with Keras



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/distribute/multi_worker_with_keras)



(https://www.tensorflow.org/tutorials/distribute/multi_worker_with_keras) (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/distribute/multi_worker_with_keras.ipynb)

Overview

This tutorial demonstrates how to perform multi-worker distributed training with a Keras model and the `Model.fit` API using the `tf.distribute.Strategy` API—specifically the `tf.distribute.MultiWorkerMirroredStrategy` class. With the help of this strategy, a Keras model that was designed to run on a single-worker can seamlessly work on multiple workers with minimal code changes.

For those interested in a deeper understanding of `tf.distribute.Strategy` APIs, the [Distributed training in TensorFlow](#) ([../guide/distributed_training.ipynb](#)) guide is available for an overview of the distribution strategies TensorFlow supports.

To learn how to use the `MultiWorkerMirroredStrategy` with Keras and a custom training loop, refer to [Custom training loop with Keras and MultiWorkerMirroredStrategy \(multi_worker_with_ctl.ipynb\)](#).

Note that the purpose of this tutorial is to demonstrate a minimal multi-worker example with two workers.

Setup

Start with some necessary imports:

```
In [ ]:
```

```
import json
import os
import sys
```

Before importing TensorFlow, make a few changes to the environment:

1. Disable all GPUs. This prevents errors caused by the workers all trying to use the same GPU. In a real-world application, each worker would be on a different machine.

```
In [ ]:
```

```
os.environ["CUDA_VISIBLE_DEVICES"] = "-1"
```

1. Reset the `TF_CONFIG` environment variable (you'll learn more about this later):

```
In [ ]:
```

```
os.environ.pop('TF_CONFIG', None)
```

1. Make sure that the current directory is on Python's path—this allows the notebook to import the files written by `%%writefile` later:

```
In [ ]:
```

```
if '.' not in sys.path:
    sys.path.insert(0, '.')
```

Now import TensorFlow:

```
In [ ]:
```

```
import tensorflow as tf
```

Dataset and model definition

Next, create an `mnist_setup.py` file with a simple model and dataset setup. This Python file will be used by the worker processes in this tutorial:

In []:

```
%%writefile mnist_setup.py

import os
import tensorflow as tf
import numpy as np

def mnist_dataset(batch_size):
    (x_train, y_train), _ = tf.keras.datasets.mnist.load_data()
    # The `x` arrays are in uint8 and have values in the [0, 255] range.
    # You need to convert them to float32 with values in the [0, 1] range.
    x_train = x_train / np.float32(255)
    y_train = y_train.astype(np.int64)
    train_dataset = tf.data.Dataset.from_tensor_slices(
        (x_train, y_train)).shuffle(60000).repeat().batch(batch_size)
    return train_dataset

def build_and_compile_cnn_model():
    model = tf.keras.Sequential([
        tf.keras.layers.InputLayer(input_shape=(28, 28)),
        tf.keras.layers.Reshape(target_shape=(28, 28, 1)),
        tf.keras.layers.Conv2D(32, 3, activation='relu'),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(128, activation='relu'),
        tf.keras.layers.Dense(10)
    ])
    model.compile(
        loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
        optimizer=tf.keras.optimizers.SGD(learning_rate=0.001),
        metrics=['accuracy'])
    return model
```

Model training on a single worker

Try training the model for a small number of epochs and observe the results of a *single worker* to make sure everything works correctly. As training progresses, the loss should drop and the accuracy should increase.

In []:

```
import mnist_setup

batch_size = 64
single_worker_dataset = mnist_setup.mnist_dataset(batch_size)
single_worker_model = mnist_setup.build_and_compile_cnn_model()
single_worker_model.fit(single_worker_dataset, epochs=3, steps_per_epoch=70)
```

Multi-worker configuration

Now let's enter the world of multi-worker training.

A cluster with jobs and tasks

In TensorFlow, distributed training involves: a 'cluster' with several jobs, and each of the jobs may have one or more 'task' s.

You will need the `TF_CONFIG` configuration environment variable for training on multiple machines, each of which possibly has a different role. `TF_CONFIG` is a JSON string used to specify the cluster configuration for each worker that is part of the cluster.

There are two components of a `TF_CONFIG` variable: 'cluster' and 'task' .

- A 'cluster' is the same for all workers and provides information about the training cluster, which is a dict consisting of different types of jobs, such as 'worker' or 'chief' .
 - In multi-worker training with `tf.distribute.MultiWorkerMirroredStrategy` , there is usually one 'worker' that takes on responsibilities, such as saving a checkpoint and writing a summary file for TensorBoard, in addition to what a regular 'worker' does. Such 'worker' is referred to as the chief worker (with a job name 'chief').
 - It is customary for the 'chief' to have 'index' 0 be appointed to (in fact, this is how `tf.distribute.Strategy` is implemented).
- A 'task' provides information of the current task and is different for each worker. It specifies the 'type' and 'index' of that worker.

Below is an example configuration:

```
In [ ]:
```

```
tf_config = {
    'cluster': {
        'worker': ['localhost:12345', 'localhost:23456']
    },
    'task': {'type': 'worker', 'index': 0}
}
```

Here is the same `TF_CONFIG` serialized as a JSON string:

```
In [ ]:
```

```
json.dumps(tf_config)
```

Note that `tf_config` is just a local variable in Python. To be able to use it for a training configuration, this dict needs to be serialized as a JSON and placed in a `TF_CONFIG` environment variable.

In the example configuration above, you set the task 'type' to 'worker' and the task 'index' to 0. Therefore, this machine is the *first* worker. It will be appointed as the 'chief' worker and do more work than the others.

Note: Other machines will need to have the `TF_CONFIG` environment variable set as well, and it should have the same 'cluster' dict, but different task 'type's or task 'index' es, depending on the roles of those machines.

For illustration purposes, this tutorial shows how you may set up a `TF_CONFIG` variable with two workers on a `localhost`.

In practice, you would create multiple workers on external IP addresses/ports and set a `TF_CONFIG` variable on each worker accordingly.

In this tutorial, you will use two workers:

- The first ('chief') worker's `TF_CONFIG` is shown above.
- For the second worker, you will set `tf_config['task']['index']=1`

Environment variables and subprocesses in notebooks

Subprocesses inherit environment variables from their parent.

For example, you can set an environment variable in this Jupyter Notebook process as follows:

```
In [ ]:
```

```
os.environ['GREETINGS'] = 'Hello TensorFlow!'
```

Then, you can access the environment variable from a subprocesses:

```
In [ ]:
```

```
%%bash
echo ${GREETINGS}
```

In the next section, you'll use a similar method to pass the `TF_CONFIG` to the worker subprocesses. In a real-world scenario, you wouldn't launch your jobs this way, but it's sufficient in this example.

Choose the right strategy

In TensorFlow, there are two main forms of distributed training:

- *Synchronous training*, where the steps of training are synced across the workers and replicas, and
- *Asynchronous training*, where the training steps are not strictly synced (for example, [parameter server training \(parameter_server_training.ipynb\)](#)).

This tutorial demonstrates how to perform synchronous multi-worker training using an instance of `tf.distribute.MultiWorkerMirroredStrategy`.

`MultiWorkerMirroredStrategy` creates copies of all variables in the model's layers on each device across all workers. It uses `CollectiveOps`, a TensorFlow op for collective communication, to aggregate gradients and keep the variables in sync. The `tf.distribute.Strategy` [guide](#) ([../guide/distributed_training.ipynb](#)) has more details about this strategy.

```
In [ ]:
```

```
strategy = tf.distribute.MultiWorkerMirroredStrategy()
```

Note: `TF_CONFIG` is parsed and TensorFlow's GRPC servers are started at the time `MultiWorkerMirroredStrategy` is called, so the `TF_CONFIG` environment variable must be set before a `tf.distribute.Strategy` instance is created. Since `TF_CONFIG` is not set yet, the above strategy is effectively single-worker training.

`MultiWorkerMirroredStrategy` provides multiple implementations via the `tf.distribute.experimental.CommunicationOptions` parameter: 1) RING implements ring-based collectives using gRPC as the cross-host communication layer; 2) NCCL uses the [NVIDIA Collective Communication Library](https://developer.nvidia.com/nccl) (<https://developer.nvidia.com/nccl>) to implement collectives; and 3) AUTO defers the choice to the runtime. The best choice of collective implementation depends upon the number and kind of GPUs, and the network interconnect in the cluster.

Train the model

With the integration of `tf.distribute.Strategy` API into `tf.keras`, the only change you will make to distribute the training to multiple-workers is enclosing the model building and `model.compile()` call inside `strategy.scope()`. The distribution strategy's scope dictates how and where the variables are created, and in the case of `MultiWorkerMirroredStrategy`, the variables created are `MirroredVariable`s, and they are replicated on each of the workers.

In []:

```
with strategy.scope():
    # Model building/compiling need to be within `strategy.scope()`.
    multi_worker_model = mnist_setup.build_and_compile_cnn_model()
```

Note: Currently there is a limitation in `MultiWorkerMirroredStrategy` where TensorFlow ops need to be created after the instance of strategy is created. If you encounter `RuntimeError: Collective ops must be configured at program startup`, try creating the instance of `MultiWorkerMirroredStrategy` at the beginning of the program and put the code that may create ops after the strategy is instantiated.

To actually run with `MultiWorkerMirroredStrategy` you'll need to run worker processes and pass a `TF_CONFIG` to them.

Like the `mnist_setup.py` file written earlier, here is the `main.py` that each of the workers will run:

In []:

```
%%writefile main.py

import os
import json

import tensorflow as tf
import mnist_setup

per_worker_batch_size = 64
tf_config = json.loads(os.environ['TF_CONFIG'])
num_workers = len(tf_config['cluster']['worker'])

strategy = tf.distribute.MultiWorkerMirroredStrategy()

global_batch_size = per_worker_batch_size * num_workers
multi_worker_dataset = mnist_setup.mnist_dataset(global_batch_size)

with strategy.scope():
    # Model building/compiling need to be within `strategy.scope()`.
    multi_worker_model = mnist_setup.build_and_compile_cnn_model()

multi_worker_model.fit(multi_worker_dataset, epochs=3, steps_per_epoch=70)
```

In the code snippet above note that the `global_batch_size`, which gets passed to `Dataset.batch`, is set to `per_worker_batch_size * num_workers`. This ensures that each worker processes batches of `per_worker_batch_size` examples regardless of the number of workers.

The current directory now contains both Python files:

In []:

```
%%bash
ls *.py
```

So json-serialize the `TF_CONFIG` and add it to the environment variables:

In []:

```
os.environ['TF_CONFIG'] = json.dumps(tf_config)
```

Now, you can launch a worker process that will run the `main.py` and use the `TF_CONFIG`:

In []:

```
# first kill any previous runs  
%killbgscripts
```

In []:

```
%%bash --bg  
python main.py &> job_0.log
```

There are a few things to note about the above command:

1. It uses the `%%bash` which is a [notebook "magic"](https://ipython.readthedocs.io/en/stable/interactive/magics.html) (<https://ipython.readthedocs.io/en/stable/interactive/magics.html>) to run some bash commands.
2. It uses the `--bg` flag to run the `bash` process in the background, because this worker will not terminate. It waits for all the workers before it starts.

The backgrounded worker process won't print output to this notebook, so the `&>` redirects its output to a file so that you can inspect what happened in a log file later.

So, wait a few seconds for the process to start up:

In []:

```
import time  
time.sleep(10)
```

Now, inspect what's been output to the worker's log file so far:

In []:

```
%%bash  
cat job_0.log
```

The last line of the log file should say: `Started server with target: grpc://localhost:12345`. The first worker is now ready, and is waiting for all the other worker(s) to be ready to proceed.

So update the `tf_config` for the second worker's process to pick up:

In []:

```
tf_config['task']['index'] = 1  
os.environ['TF_CONFIG'] = json.dumps(tf_config)
```

Launch the second worker. This will start the training since all the workers are active (so there's no need to background this process):

In []:

```
%%bash  
python main.py
```

If you recheck the logs written by the first worker, you'll learn that it participated in training that model:

In []:

```
%%bash  
cat job_0.log
```

Unsurprisingly, this ran *slower* than the test run at the beginning of this tutorial.

Running multiple workers on a single machine only adds overhead.

The goal here was not to improve the training time, but only to give an example of multi-worker training.

In []:

```
# Delete the `TF_CONFIG`, and kill any background tasks so they don't affect the next section.  
os.environ.pop('TF_CONFIG', None)  
%killbgscripts
```

Multi-worker training in depth

So far, you have learned how to perform a basic multi-worker setup.

During the rest of the tutorial, you will learn about other factors, which may be useful or important for real use cases, in detail.

Dataset sharding

In multi-worker training, *dataset sharding* is needed to ensure convergence and performance.

The example in the previous section relies on the default autosharding provided by the `tf.distribute.Strategy` API. You can control the sharding by setting the `tf.data.experimental.AutoShardPolicy` of the `tf.data.experimental.DistributeOptions`.

To learn more about *auto-sharding*, refer to the [Distributed input guide \(https://www.tensorflow.org/tutorials/distribute/input#sharding\)](https://www.tensorflow.org/tutorials/distribute/input#sharding).

Here is a quick example of how to turn the auto sharding off, so that each replica processes every example (*not recommended*):

In []:

```
options = tf.data.Options()
options.experimental_distribute.auto_shard_policy = tf.data.experimental.AutoShardPolicy.OFF

global_batch_size = 64
multi_worker_dataset = mnist_setup.mnist_dataset(batch_size=64)
dataset_no_auto_shard = multi_worker_dataset.with_options(options)
```

Evaluation

If you pass the `validation_data` into `Model.fit`, it will alternate between training and evaluation for each epoch. The evaluation taking the `validation_data` is distributed across the same set of workers and the evaluation results are aggregated and available for all workers.

Similar to training, the validation dataset is automatically sharded at the file level. You need to set a global batch size in the validation dataset and set the `validation_steps`.

A repeated dataset is also recommended for evaluation.

Alternatively, you can also create another task that periodically reads checkpoints and runs the evaluation. This is what Estimator does. But this is not a recommended way to perform evaluation and thus its details are omitted.

Performance

You now have a Keras model that is all set up to run in multiple workers with the `MultiWorkerMirroredStrategy`.

To tweak performance of multi-worker training, you can try the following:

- `tf.distribute.MultiWorkerMirroredStrategy` provides multiple [collective communication implementations \(https://www.tensorflow.org/api_docs/python/tf/distribute/experimental/CommunicationImplementation\)](https://www.tensorflow.org/api_docs/python/tf/distribute/experimental/CommunicationImplementation):
 - RING implements ring-based collectives using gRPC as the cross-host communication layer.
 - NCCL uses the [NVIDIA Collective Communication Library \(https://developer.nvidia.com/nccl\)](https://developer.nvidia.com/nccl) to implement collectives.
 - AUTO defers the choice to the runtime.

The best choice of collective implementation depends upon the number of GPUs, the type of GPUs, and the network interconnect in the cluster.

To override the automatic choice, specify the `communication_options` parameter of `MultiWorkerMirroredStrategy`'s constructor.

For example:

```
communication_options=tf.distribute.experimental.CommunicationOptions(implementation=tf.distribute.experimental.CollectiveCommunication.NCCL)
```

- Cast the variables to `tf.float` if possible:
 - The official ResNet model includes [an example \(https://github.com/tensorflow/models/blob/8367cf6dabe11adf7628541706b660821f397dce/official/resnet/resnet_model.py#L466\)](https://github.com/tensorflow/models/blob/8367cf6dabe11adf7628541706b660821f397dce/official/resnet/resnet_model.py#L466) of how this can be done.

Fault tolerance

In synchronous training, the cluster would fail if one of the workers fails and no failure-recovery mechanism exists.

Using Keras with `tf.distribute.Strategy` comes with the advantage of fault tolerance in cases where workers die or are otherwise unstable. You can do this by preserving the training state in the distributed file system of your choice, such that upon a restart of the instance that previously failed or preempted, the training state is recovered.

When a worker becomes unavailable, other workers will fail (possibly after a timeout). In such cases, the unavailable worker needs to be restarted, as well as other workers that have failed.

Note: Previously, the `ModelCheckpoint` callback provided a mechanism to restore the training state upon a restart from a job failure for multi-worker training. The TensorFlow team are introducing a new `BackupAndRestore` callback, which also adds the support to single-worker training for a consistent experience, and removed the fault tolerance functionality from existing `ModelCheckpoint` callback. From now on, applications that rely on this behavior should migrate to the new `BackupAndRestore` callback.

ModelCheckpoint callback

`ModelCheckpoint` callback no longer provides fault tolerance functionality, please use `BackupAndRestore` callback instead.

The `ModelCheckpoint` callback can still be used to save checkpoints. But with this, if training was interrupted or successfully finished, in order to continue training from the checkpoint, the user is responsible to load the model manually.

Optionally the user can choose to save and restore model/weights outside `ModelCheckpoint` callback.

Model saving and loading

To save your model using `model.save` or `tf.saved_model.save`, the saving destination needs to be different for each worker.

- For non-chief workers, you will need to save the model to a temporary directory.
- For the chief, you will need to save to the provided model directory.

The temporary directories on the worker need to be unique to prevent errors resulting from multiple workers trying to write to the same location.

The model saved in all the directories is identical, and typically only the model saved by the chief should be referenced for restoring or serving.

You should have some cleanup logic that deletes the temporary directories created by the workers once your training has completed.

The reason for saving on the chief and workers at the same time is because you might be aggregating variables during checkpointing which requires both the chief and workers to participate in the allreduce communication protocol. On the other hand, letting chief and workers save to the same model directory will result in errors due to contention.

Using the `MultiWorkerMirroredStrategy`, the program is run on every worker, and in order to know whether the current worker is chief, it takes advantage of the cluster resolver object that has attributes `task_type` and `task_id`:

- `task_type` tells you what the current job is (e.g. 'worker').
- `task_id` tells you the identifier of the worker.
- The worker with `task_id == 0` is designated as the chief worker.

In the code snippet below, the `write_filepath` function provides the file path to write, which depends on the the worker's `task_id`:

- For the chief worker (with `task_id == 0`), it writes to the original file path.
- For other workers, it creates a temporary directory—`temp_dir`—with the `task_id` in the directory path to write in:

```
In [ ]:
```

```
model_path = '/tmp/keras-model'

def _is_chief(task_type, task_id):
    # Note: there are two possible `TF_CONFIG` configuration.
    #   1) In addition to `worker` tasks, a `chief` task type is used;
    #       in this case, this function should be modified to
    #       `return task_type == 'chief'`.
    #   2) Only `worker` task type is used; in this case, worker 0 is
    #       regarded as the chief. The implementation demonstrated here
    #       is for this case.
    # For the purpose of this Colab section, the `task_type is None` case
    # is added because it is effectively run with only a single worker.
    return (task_type == 'worker' and task_id == 0) or task_type is None

def _get_temp_dir(dirpath, task_id):
    base_dirpath = 'workertemp_' + str(task_id)
    temp_dir = os.path.join(dirpath, base_dirpath)
    tf.io.gfile.makedirs(temp_dir)
    return temp_dir

def write_filepath(filepath, task_type, task_id):
    dirpath = os.path.dirname(filepath)
    base = os.path.basename(filepath)
    if not _is_chief(task_type, task_id):
        dirpath = _get_temp_dir(dirpath, task_id)
    return os.path.join(dirpath, base)

task_type, task_id = (strategy.cluster_resolver.task_type,
                      strategy.cluster_resolver.task_id)
write_model_path = write_filepath(model_path, task_type, task_id)
```

With that, you're now ready to save:

```
In [ ]:
```

```
multi_worker_model.save(write_model_path)
```

As described above, later on the model should only be loaded from the path chief saved to, so let's remove the temporary ones the non-chief workers saved:

```
In [ ]:
```

```
if not _is_chief(task_type, task_id):
    tf.io.gfile.rmtree(os.path.dirname(write_model_path))
```

Now, when it's time to load, let's use convenient `tf.keras.models.load_model` API, and continue with further work.

Here, assume only using single worker to load and continue training, in which case you do not call `tf.keras.models.load_model` within another `strategy.scope()` (note that `strategy = tf.distribute.MultiWorkerMirroredStrategy()`, as defined earlier):

```
In [ ]:
```

```
loaded_model = tf.keras.models.load_model(model_path)

# Now that the model is restored, and can continue with the training.
loaded_model.fit(single_worker_dataset, epochs=2, steps_per_epoch=20)
```

Checkpoint saving and restoring

On the other hand, checkpointing allows you to save your model's weights and restore them without having to save the whole model.

Here, you'll create one `tf.train.Checkpoint` that tracks the model, which is managed by the `tf.train.CheckpointManager`, so that only the latest checkpoint is preserved:

```
In [ ]:
```

```
checkpoint_dir = '/tmp/ckpt'

checkpoint = tf.train.Checkpoint(model=multi_worker_model)
write_checkpoint_dir = write_filepath(checkpoint_dir, task_type, task_id)
checkpoint_manager = tf.train.CheckpointManager(
    checkpoint, directory=write_checkpoint_dir, max_to_keep=1)
```

Once the `CheckpointManager` is set up, you're now ready to save and remove the checkpoints the non-chief workers had saved:

In []:

```
checkpoint_manager.save()
if not _is_chief(task_type, task_id):
    tf.io.gfile.rmtree(write_checkpoint_dir)
```

Now, when you need to restore the model, you can find the latest checkpoint saved using the convenient `tf.train.latest_checkpoint` function. After restoring the checkpoint, you can continue with training.

In []:

```
latest_checkpoint = tf.train.latest_checkpoint(checkpoint_dir)
checkpoint.restore(latest_checkpoint)
multi_worker_model.fit(multi_worker_dataset, epochs=2, steps_per_epoch=20)
```

BackupAndRestore callback

The `tf.keras.callbacks.BackupAndRestore` callback provides the fault tolerance functionality by backing up the model and current epoch number in a temporary checkpoint file under `backup_dir` argument to `BackupAndRestore`. This is done at the end of each epoch.

Once jobs get interrupted and restart, the callback restores the last checkpoint, and training continues from the beginning of the interrupted epoch. Any partial training already done in the unfinished epoch before interruption will be thrown away, so that it doesn't affect the final model state.

To use it, provide an instance of `tf.keras.callbacks.BackupAndRestore` at the `Model.fit` call.

With `MultiWorkerMirroredStrategy`, if a worker gets interrupted, the whole cluster pauses until the interrupted worker is restarted. Other workers will also restart, and the interrupted worker rejoins the cluster. Then, every worker reads the checkpoint file that was previously saved and picks up its former state, thereby allowing the cluster to get back in sync. Then, the training continues.

The `BackupAndRestore` callback uses the `CheckpointManager` to save and restore the training state, which generates a file called `checkpoint` that tracks existing checkpoints together with the latest one. For this reason, `backup_dir` should not be re-used to store other checkpoints in order to avoid name collision.

Currently, the `BackupAndRestore` callback supports single-worker training with no strategy—`MirroredStrategy`—and multi-worker training with `MultiWorkerMirroredStrategy`.

Below are two examples for both multi-worker training and single-worker training:

In []:

```
# Multi-worker training with `MultiWorkerMirroredStrategy`
# and the `BackupAndRestore` callback.

callbacks = [tf.keras.callbacks.BackupAndRestore(backup_dir='/tmp/backup')]
with strategy.scope():
    multi_worker_model = mnist_setup.build_and_compile_cnn_model()
    multi_worker_model.fit(multi_worker_dataset,
                           epochs=3,
                           steps_per_epoch=70,
                           callbacks=callbacks)
```

If you inspect the directory of `backup_dir` you specified in `BackupAndRestore`, you may notice some temporarily generated checkpoint files. Those files are needed for recovering the previously lost instances, and they will be removed by the library at the end of `Model.fit` upon successful exiting of your training.

Note: Currently the `BackupAndRestore` callback only supports eager mode. In graph mode, consider using [Save/Restore Model](#) mentioned above, and by providing `initial_epoch` in `Model.fit`.

Additional resources

1. The [Distributed training in TensorFlow](https://www.tensorflow.org/guide/distributed_training) (https://www.tensorflow.org/guide/distributed_training) guide provides an overview of the available distribution strategies.
2. The [Custom training loop with Keras and MultiWorkerMirroredStrategy](#) ([multi_worker_with_ctl.ipynb](#)) tutorial shows how to use the `MultiWorkerMirroredStrategy` with Keras and a custom training loop.
3. Check out the [official models](https://github.com/tensorflow/models/tree/master/official) (<https://github.com/tensorflow/models/tree/master/official>), many of which can be configured to run multiple distribution strategies.
4. The [Better performance with tf.function](#) ([../guide/function.ipynb](#)) guide provides information about other strategies and tools, such as the [TensorFlow Profiler](#) ([../guide/profiler.md](#)) you can use to optimize the performance of your TensorFlow models.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Distributed training with Keras



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/distribute/keras)

(<https://www.tensorflow.org/tutorials/distribute/keras>)



[Run in Google Colab](#)

(<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/distribute/keras.ipynb>)

Overview

The `tf.distribute.Strategy` API provides an abstraction for distributing your training across multiple processing units. It allows you to carry out distributed training using existing models and training code with minimal changes.

This tutorial demonstrates how to use the `tf.distribute.MirroredStrategy` to perform in-graph replication with *synchronous training on many GPUs on one machine*. The strategy essentially copies all of the model's variables to each processor. Then, it uses [all-reduce](http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/) (<http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/>) to combine the gradients from all processors, and applies the combined value to all copies of the model.

You will use the `tf.keras` APIs to build the model and `Model.fit` for training it. (To learn about distributed training with a custom training loop and the `MirroredStrategy`, check out [this tutorial \(custom_training.ipynb\)](#).)

`MirroredStrategy` trains your model on multiple GPUs on a single machine. For *synchronous training on many GPUs on multiple workers*, use the `tf.distribute.MultiWorkerMirroredStrategy` with the [Keras Model.fit \(multi_worker_with_keras.ipynb\)](#) or [a custom training loop \(multi_worker_with_ctl.ipynb\)](#). For other options, refer to the [Distributed training guide](#) ([..../guide/distributed_training.ipynb](#)).

To learn about various other strategies, there is the [Distributed training with TensorFlow](#) ([..../guide/distributed_training.ipynb](#)) guide.

Setup

In []:

```
import tensorflow_datasets as tfds
import tensorflow as tf

import os

# Load the TensorBoard notebook extension.
%load_ext tensorboard
```

In []:

```
print(tf.__version__)
```

Download the dataset

Load the MNIST dataset from [TensorFlow Datasets](#) (<https://www.tensorflow.org/datasets>). This returns a dataset in the `tf.data` format.

Setting the `with_info` argument to `True` includes the metadata for the entire dataset, which is being saved here to `info`. Among other things, this metadata object includes the number of train and test examples.

In []:

```
datasets, info = tfds.load(name='mnist', with_info=True, as_supervised=True)

mnist_train, mnist_test = datasets['train'], datasets['test']
```

Define the distribution strategy

Create a `MirroredStrategy` object. This will handle distribution and provide a context manager (`MirroredStrategy.scope`) to build your model inside.

In []:

```
strategy = tf.distribute.MirroredStrategy()
```

In []:

```
print('Number of devices: {}'.format(strategy.num_replicas_in_sync))
```

Set up the input pipeline

When training a model with multiple GPUs, you can use the extra computing power effectively by increasing the batch size. In general, use the largest batch size that fits the GPU memory and tune the learning rate accordingly.

In []:

```
# You can also do info.splits.total_num_examples to get the total
# number of examples in the dataset.

num_train_examples = info.splits['train'].num_examples
num_test_examples = info.splits['test'].num_examples

BUFFER_SIZE = 10000

BATCH_SIZE_PER_REPLICA = 64
BATCH_SIZE = BATCH_SIZE_PER_REPLICA * strategy.num_replicas_in_sync
```

Define a function that normalizes the image pixel values from the `[0, 255]` range to the `[0, 1]` range ([feature scaling](#) (https://en.wikipedia.org/wiki/Feature_scaling)):

In []:

```
def scale(image, label):
    image = tf.cast(image, tf.float32)
    image /= 255

    return image, label
```

Apply this `scale` function to the training and test data, and then use the `tf.data.Dataset` APIs to shuffle the training data (`Dataset.shuffle`), and batch it (`Dataset.batch`). Notice that you are also keeping an in-memory cache of the training data to improve performance (`Dataset.cache`).

In []:

```
train_dataset = mnist_train.map(scale).cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
eval_dataset = mnist_test.map(scale).batch(BATCH_SIZE)
```

Create the model

Within the context of `Strategy.scope`, create and compile the model using the Keras API:

In []:

```
with strategy.scope():
    model = tf.keras.Sequential([
        tf.keras.layers.Conv2D(32, 3, activation='relu', input_shape=(28, 28, 1)),
        tf.keras.layers.MaxPooling2D(),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(10)
    ])

    model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                  optimizer=tf.keras.optimizers.Adam(),
                  metrics=['accuracy'])
```

Define the callbacks

Define the following [Keras Callbacks](https://www.tensorflow.org/guide/keras/train_and_evaluate) (https://www.tensorflow.org/guide/keras/train_and_evaluate):

- `tf.keras.callbacks.TensorBoard` : writes a log for TensorBoard, which allows you to visualize the graphs.
- `tf.keras.callbacks.ModelCheckpoint` : saves the model at a certain frequency, such as after every epoch.
- `tf.keras.callbacks.BackupAndRestore` : provides the fault tolerance functionality by backing up the model and current epoch number. Learn more in the *Fault tolerance* section of the [Multi-worker training with Keras \(multi_worker_with_keras.ipynb\)](#) tutorial.
- `tf.keras.callbacks.LearningRateScheduler` : schedules the learning rate to change after, for example, every epoch/batch.

For illustrative purposes, add a [custom callback](https://www.tensorflow.org/guide/keras/custom_callback) (https://www.tensorflow.org/guide/keras/custom_callback) called `PrintLR` to display the *learning rate* in the notebook.

Note: Use the `BackupAndRestore` callback instead of `ModelCheckpoint` as the main mechanism to restore the training state upon a restart from a job failure. Since `BackupAndRestore` only supports eager mode, in graph mode consider using `ModelCheckpoint`.

In []:

```
# Define the checkpoint directory to store the checkpoints.  
checkpoint_dir = './training_checkpoints'  
# Define the name of the checkpoint files.  
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt_{epoch}")
```

In []:

```
# Define a function for decaying the learning rate.  
# You can define any decay function you need.  
def decay(epoch):  
    if epoch < 3:  
        return 1e-3  
    elif epoch >= 3 and epoch < 7:  
        return 1e-4  
    else:  
        return 1e-5
```

In []:

```
# Define a callback for printing the learning rate at the end of each epoch.  
class PrintLR(tf.keras.callbacks.Callback):  
    def on_epoch_end(self, epoch, logs=None):  
        print('\nLearning rate for epoch {} is {}'.format(epoch + 1, model.optimizer.lr.numpy()))
```

In []:

```
# Put all the callbacks together.  
callbacks = [  
    tf.keras.callbacks.TensorBoard(log_dir='./logs'),  
    tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_prefix,  
                                      save_weights_only=True),  
    tf.keras.callbacks.LearningRateScheduler(decay),  
    PrintLR()  
]
```

Train and evaluate

Now, train the model in the usual way by calling Keras `Model.fit` on the model and passing in the dataset created at the beginning of the tutorial. This step is the same whether you are distributing the training or not.

In []:

```
EPOCHS = 12  
  
model.fit(train_dataset, epochs=EPOCHS, callbacks=callbacks)
```

Check for saved checkpoints:

In []:

```
# Check the checkpoint directory.  
!ls {checkpoint_dir}
```

To check how well the model performs, load the latest checkpoint and call `Model.evaluate` on the test data:

```
In [ ]:
```

```
model.load_weights(tf.train.latest_checkpoint(checkpoint_dir))
eval_loss, eval_acc = model.evaluate(eval_dataset)
print('Eval loss: {}, Eval accuracy: {}'.format(eval_loss, eval_acc))
```

To visualize the output, launch TensorBoard and view the logs:

```
In [ ]:
```

```
%tensorboard --logdir=logs
```

```
In [ ]:
```

```
!ls -sh ./logs
```

Export to SavedModel

Export the graph and the variables to the platform-agnostic SavedModel format using Keras `Model.save`. After your model is saved, you can load it with or without the `Strategy.scope`.

```
In [ ]:
```

```
path = 'saved_model/'
```

```
In [ ]:
```

```
model.save(path, save_format='tf')
```

Now, load the model without `Strategy.scope`:

```
In [ ]:
```

```
unreplicated_model = tf.keras.models.load_model(path)

unreplicated_model.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    optimizer=tf.keras.optimizers.Adam(),
    metrics=['accuracy'])

eval_loss, eval_acc = unreplicated_model.evaluate(eval_dataset)
print('Eval loss: {}, Eval Accuracy: {}'.format(eval_loss, eval_acc))
```

Load the model with `Strategy.scope`:

```
In [ ]:
```

```
with strategy.scope():
    replicated_model = tf.keras.models.load_model(path)
    replicated_model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                             optimizer=tf.keras.optimizers.Adam(),
                             metrics=['accuracy'])

    eval_loss, eval_acc = replicated_model.evaluate(eval_dataset)
    print ('Eval loss: {}, Eval Accuracy: {}'.format(eval_loss, eval_acc))
```

Additional resources

More examples that use different distribution strategies with the Keras `Model.fit` API:

1. The [Solve GLUE tasks using BERT on TPU](https://www.tensorflow.org/text/tutorials/bert_glue) (https://www.tensorflow.org/text/tutorials/bert_glue) tutorial uses `tf.distribute.MirroredStrategy` for training on GPUs and `tf.distribute.TPUStrategy` on TPUs.
2. The [Save and load a model using a distribution strategy](https://www.tensorflow.org/tutorials/distribution_strategy/save_and_load.ipynb) ([save_and_load.ipynb](https://www.tensorflow.org/tutorials/distribution_strategy/save_and_load.ipynb)) tutorial demonstrates how to use the `SavedModel` APIs with `tf.distribute.Strategy`.
3. The [official TensorFlow models](https://github.com/tensorflow/models/tree/master/official) (<https://github.com/tensorflow/models/tree/master/official>) can be configured to run multiple distribution strategies.

To learn more about TensorFlow distribution strategies:

1. The [Custom training with `tf.distribute.Strategy`](https://www.tensorflow.org/tutorials/distribution_strategy/custom_training.ipynb) ([custom_training.ipynb](https://www.tensorflow.org/tutorials/distribution_strategy/custom_training.ipynb)) tutorial shows how to use the `tf.distribute.MirroredStrategy` for single-worker training with a custom training loop.
2. The [Multi-worker training with Keras](https://www.tensorflow.org/tutorials/distribution_strategy/multi_worker_with_keras.ipynb) ([multi_worker_with_keras.ipynb](https://www.tensorflow.org/tutorials/distribution_strategy/multi_worker_with_keras.ipynb)) tutorial shows how to use the `MultiWorkerMirroredStrategy` with `Model.fit`.
3. The [Custom training loop with Keras and `MultiWorkerMirroredStrategy`](https://www.tensorflow.org/tutorials/distribution_strategy/multi_worker_with_ctl.ipynb) ([multi_worker_with_ctl.ipynb](https://www.tensorflow.org/tutorials/distribution_strategy/multi_worker_with_ctl.ipynb)) tutorial shows how to use the `MultiWorkerMirroredStrategy` with Keras and a custom training loop.
4. The [Distributed training in TensorFlow](https://www.tensorflow.org/guide/distributed_training) (https://www.tensorflow.org/guide/distributed_training) guide provides an overview of the available distribution strategies.
5. The [Better performance with `tf.function`](https://www.tensorflow.org/guide/function.ipynb) ([./guide/function.ipynb](https://www.tensorflow.org/guide/function.ipynb)) guide provides information about other strategies and tools, such as the [TensorFlow Profiler](https://www.tensorflow.org/guide/profiler.md) ([./guide/profiler.md](https://www.tensorflow.org/guide/profiler.md)) you can use to optimize the performance of your TensorFlow models.

Note: `tf.distribute.Strategy` is actively under development and TensorFlow will be adding more examples and tutorials in the near future. Please give it a try. Your feedback is welcome—feel free to submit it via [issues on GitHub](https://github.com/tensorflow/tensorflow/issues/new) (<https://github.com/tensorflow/tensorflow/issues/new>).

Copyright 2020 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Simple audio recognition: Recognizing keywords



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/audio/simple_audio)

(https://www.tensorflow.org/tutorials/audio/simple_audio)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/audio/simple_audio.ipynb)

(https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/audio/simple_audio.ipynb)

This tutorial demonstrates how to preprocess audio files in the WAV format and build and train a basic [automatic speech recognition](#) (https://en.wikipedia.org/wiki/Speech_recognition) (ASR) model for recognizing ten different words. You will use a portion of the [Speech Commands dataset](#) (https://www.tensorflow.org/datasets/catalog/speech_commands) ([Warden, 2018 \(https://arxiv.org/abs/1804.03209\)](https://arxiv.org/abs/1804.03209)), which contains short (one-second or less) audio clips of commands, such as "down", "go", "left", "no", "right", "stop", "up" and "yes".

Real-world speech and audio recognition [systems](#) (<https://ai.googleblog.com/search/label/Speech%20Recognition>) are complex. But, like [image classification with the MNIST dataset](#) ([./quickstart/beginner.ipynb](https://quickstart/beginner.ipynb)), this tutorial should give you a basic understanding of the techniques involved.

Setup

Import necessary modules and dependencies. Note that you'll be using [seaborn](#) (<https://seaborn.pydata.org/>) for visualization in this tutorial.

```
In [ ]:
```

```
import os
import pathlib

import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import tensorflow as tf

from tensorflow.keras import layers
from tensorflow.keras import models
from IPython import display

# Set the seed value for experiment reproducibility.
seed = 42
tf.random.set_seed(seed)
np.random.seed(seed)
```

Import the mini Speech Commands dataset

To save time with data loading, you will be working with a smaller version of the Speech Commands dataset. The [original dataset](https://www.tensorflow.org/datasets/catalog/speech_commands) (https://www.tensorflow.org/datasets/catalog/speech_commands) consists of over 105,000 audio files in the [WAV \(Waveform\) audio file format](#) (<https://www.aelius.com/njh/wavemetatools/doc/riffmci.pdf>) of people saying 35 different words. This data was collected by Google and released under a CC BY license.

Download and extract the `mini_speech_commands.zip` file containing the smaller Speech Commands datasets with `tf.keras.utils.get_file`:

```
In [ ]:
```

```
DATASET_PATH = 'data/mini_speech_commands'

data_dir = pathlib.Path(DATASET_PATH)
if not data_dir.exists():
    tf.keras.utils.get_file(
        'mini_speech_commands.zip',
        origin="http://storage.googleapis.com/download.tensorflow.org/data/mini_speech_commands.zip",
        extract=True,
        cache_dir='.', cache_subdir='data')
```

The dataset's audio clips are stored in eight folders corresponding to each speech command: `no` , `yes` , `down` , `go` , `left` , `up` , `right` , and `stop` :

```
In [ ]:
```

```
commands = np.array(tf.io.gfile.listdir(str(data_dir)))
commands = commands[commands != 'README.md']
print('Commands:', commands)
```

Extract the audio clips into a list called `filenames` , and shuffle it:

```
In [ ]:
```

```
filenames = tf.io.gfile.glob(str(data_dir) + '/*/*')
filenames = tf.random.shuffle(filenames)
num_samples = len(filenames)
print('Number of total examples:', num_samples)
print('Number of examples per label:',
      len(tf.io.gfile.listdir(str(data_dir/commands[0]))))
print('Example file tensor:', filenames[0])
```

Split `filenames` into training, validation and test sets using a 80:10:10 ratio, respectively:

```
In [ ]:
```

```
train_files = filenames[:6400]
val_files = filenames[6400: 6400 + 800]
test_files = filenames[-800:]

print('Training set size', len(train_files))
print('Validation set size', len(val_files))
print('Test set size', len(test_files))
```

Read the audio files and their labels

In this section you will preprocess the dataset, creating decoded tensors for the waveforms and the corresponding labels. Note that:

- Each WAV file contains time-series data with a set number of samples per second.
- Each sample represents the [amplitude](https://en.wikipedia.org/wiki/Ampitude) (<https://en.wikipedia.org/wiki/Ampitude>) of the audio signal at that specific time.
- In a [16-bit](https://en.wikipedia.org/wiki/16-bit) (<https://en.wikipedia.org/wiki/16-bit>) system, like the WAV files in the mini Speech Commands dataset, the amplitude values range from -32,768 to 32,767.
- The [sample rate](https://en.wikipedia.org/wiki/Sampling_(signal_processing)#Audio_sampling) ([https://en.wikipedia.org/wiki/Sampling_\(signal_processing\)#Audio_sampling](https://en.wikipedia.org/wiki/Sampling_(signal_processing)#Audio_sampling)) for this dataset is 16kHz.

The shape of the tensor returned by `tf.audio.decode_wav` is `[samples, channels]`, where `channels` is 1 for mono or 2 for stereo. The mini Speech Commands dataset only contains mono recordings.

In []:

```
test_file = tf.io.read_file(DATASET_PATH+'down/0a9f9af7_nohash_0.wav')
test_audio, _ = tf.audio.decode_wav(contents=test_file)
test_audio.shape
```

Now, let's define a function that preprocesses the dataset's raw WAV audio files into audio tensors:

In []:

```
def decode_audio(audio_binary):
    # Decode WAV-encoded audio files to `float32` tensors, normalized
    # to the [-1.0, 1.0] range. Return `float32` audio and a sample rate.
    audio, _ = tf.audio.decode_wav(contents=audio_binary)
    # Since all the data is single channel (mono), drop the `channels`
    # axis from the array.
    return tf.squeeze(audio, axis=-1)
```

Define a function that creates labels using the parent directories for each file:

- Split the file paths into `tf.RaggedTensor`s (tensors with ragged dimensions—with slices that may have different lengths).

In []:

```
def get_label(file_path):
    parts = tf.strings.split(
        input=file_path,
        sep=os.path.sep)
    # Note: You'll use indexing here instead of tuple unpacking to enable this
    # to work in a TensorFlow graph.
    return parts[-2]
```

Define another helper function—`get_waveform_and_label`—that puts it all together:

- The input is the WAV audio filename.
- The output is a tuple containing the audio and label tensors ready for supervised learning.

In []:

```
def get_waveform_and_label(file_path):
    label = get_label(file_path)
    audio_binary = tf.io.read_file(file_path)
    waveform = decode_audio(audio_binary)
    return waveform, label
```

Build the training set to extract the audio-label pairs:

- Create a `tf.data.Dataset` with `Dataset.from_tensor_slices` and `Dataset.map`, using `get_waveform_and_label` defined earlier.

You'll build the validation and test sets using a similar procedure later on.

In []:

```
AUTOTUNE = tf.data.AUTOTUNE

files_ds = tf.data.Dataset.from_tensor_slices(train_files)

waveform_ds = files_ds.map(
    map_func=get_waveform_and_label,
    num_parallel_calls=AUTOTUNE)
```

Let's plot a few audio waveforms:

In []:

```
rows = 3
cols = 3
n = rows * cols
fig, axes = plt.subplots(rows, cols, figsize=(10, 12))

for i, (audio, label) in enumerate(waveform_ds.take(n)):
    r = i // cols
    c = i % cols
    ax = axes[r][c]
    ax.plot(audio.numpy())
    ax.set_yticks(np.arange(-1.2, 1.2, 0.2))
    label = label.numpy().decode('utf-8')
    ax.set_title(label)

plt.show()
```

Convert waveforms to spectrograms

The waveforms in the dataset are represented in the time domain. Next, you'll transform the waveforms from the time-domain signals into the time-frequency-domain signals by computing the [short-time Fourier transform \(STFT\)](https://en.wikipedia.org/wiki/Short-time_Fourier_transform) (https://en.wikipedia.org/wiki/Short-time_Fourier_transform) to convert the waveforms to as [spectrograms](https://en.wikipedia.org/wiki/Spectrogram) (<https://en.wikipedia.org/wiki/Spectrogram>), which show frequency changes over time and can be represented as 2D images. You will feed the spectrogram images into your neural network to train the model.

A Fourier transform (`tf.signal.fft`) converts a signal to its component frequencies, but loses all time information. In comparison, STFT (`tf.signal.stft`) splits the signal into windows of time and runs a Fourier transform on each window, preserving some time information, and returning a 2D tensor that you can run standard convolutions on.

Create a utility function for converting waveforms to spectrograms:

- The waveforms need to be of the same length, so that when you convert them to spectrograms, the results have similar dimensions. This can be done by simply zero-padding the audio clips that are shorter than one second (using `tf.zeros`).
- When calling `tf.signal.stft`, choose the `frame_length` and `frame_step` parameters such that the generated spectrogram "image" is almost square. For more information on the STFT parameters choice, refer to [this Coursera video](https://www.coursera.org/lecture/audio-signal-processing/stft-2-tjEQe) (<https://www.coursera.org/lecture/audio-signal-processing/stft-2-tjEQe>) on audio signal processing and STFT.
- The STFT produces an array of complex numbers representing magnitude and phase. However, in this tutorial you'll only use the magnitude, which you can derive by applying `tf.abs` on the output of `tf.signal.stft` .

In []:

```
def get_spectrogram(waveform):
    # Zero-padding for an audio waveform with less than 16,000 samples.
    input_len = 16000
    waveform = waveform[:input_len]
    zero_padding = tf.zeros(
        [16000] - tf.shape(waveform),
        dtype=tf.float32)
    # Cast the waveform tensors' dtype to float32.
    waveform = tf.cast(waveform, dtype=tf.float32)
    # Concatenate the waveform with `zero_padding`, which ensures all audio
    # clips are of the same length.
    equal_length = tf.concat([waveform, zero_padding], 0)
    # Convert the waveform to a spectrogram via a STFT.
    spectrogram = tf.signal.stft(
        equal_length, frame_length=255, frame_step=128)
    # Obtain the magnitude of the STFT.
    spectrogram = tf.abs(spectrogram)
    # Add a `channels` dimension, so that the spectrogram can be used
    # as image-like input data with convolution layers (which expect
    # shape (`batch_size`, `height`, `width`, `channels`).
    spectrogram = spectrogram[..., tf.newaxis]
    return spectrogram
```

Next, start exploring the data. Print the shapes of one example's tensorized waveform and the corresponding spectrogram, and play the original audio:

In []:

```
for waveform, label in waveform_ds.take(1):
    label = label.numpy().decode('utf-8')
    spectrogram = get_spectrogram(waveform)

print('Label:', label)
print('Waveform shape:', waveform.shape)
print('Spectrogram shape:', spectrogram.shape)
print('Audio playback')
display.display(display.Audio(waveform, rate=16000))
```

Now, define a function for displaying a spectrogram:

In []:

```
def plot_spectrogram(spectrogram, ax):
    if len(spectrogram.shape) > 2:
        assert len(spectrogram.shape) == 3
        spectrogram = np.squeeze(spectrogram, axis=-1)
    # Convert the frequencies to log scale and transpose, so that the time is
    # represented on the x-axis (columns).
    # Add an epsilon to avoid taking a log of zero.
    log_spec = np.log(spectrogram.T + np.finfo(float).eps)
    height = log_spec.shape[0]
    width = log_spec.shape[1]
    X = np.linspace(0, np.size(spectrogram), num=width, dtype=int)
    Y = range(height)
    ax.pcolormesh(X, Y, log_spec)
```

Plot the example's waveform over time and the corresponding spectrogram (frequencies over time):

In []:

```
fig, axes = plt.subplots(2, figsize=(12, 8))
timescale = np.arange(waveform.shape[0])
axes[0].plot(timescale, waveform.numpy())
axes[0].set_title('Waveform')
axes[0].set_xlim([0, 16000])

plot_spectrogram(spectrogram.numpy(), axes[1])
axes[1].set_title('Spectrogram')
plt.show()
```

Now, define a function that transforms the waveform dataset into spectrograms and their corresponding labels as integer IDs:

In []:

```
def get_spectrogram_and_label_id(audio, label):
    spectrogram = get_spectrogram(audio)
    label_id = tf.argmax(label == commands)
    return spectrogram, label_id
```

Map `get_spectrogram_and_label_id` across the dataset's elements with `Dataset.map`:

In []:

```
spectrogram_ds = waveform_ds.map(
    map_func=get_spectrogram_and_label_id,
    num_parallel_calls=AUTOTUNE)
```

Examine the spectrograms for different examples of the dataset:

In []:

```
rows = 3
cols = 3
n = rows*cols
fig, axes = plt.subplots(rows, cols, figsize=(10, 10))

for i, (spectrogram, label_id) in enumerate(spectrogram_ds.take(n)):
    r = i // cols
    c = i % cols
    ax = axes[r][c]
    plot_spectrogram(spectrogram.numpy(), ax)
    ax.set_title(commands[label_id.numpy()])
    ax.axis('off')

plt.show()
```

Build and train the model

Repeat the training set preprocessing on the validation and test sets:

In []:

```
def preprocess_dataset(files):
    files_ds = tf.data.Dataset.from_tensor_slices(files)
    output_ds = files_ds.map(
        map_func=get_waveform_and_label,
        num_parallel_calls=AUTOTUNE)
    output_ds = output_ds.map(
        map_func=get_spectrogram_and_label_id,
        num_parallel_calls=AUTOTUNE)
    return output_ds
```

In []:

```
train_ds = spectrogram_ds
val_ds = preprocess_dataset(val_files)
test_ds = preprocess_dataset(test_files)
```

Batch the training and validation sets for model training:

In []:

```
batch_size = 64
train_ds = train_ds.batch(batch_size)
val_ds = val_ds.batch(batch_size)
```

Add `Dataset.cache` and `Dataset.prefetch` operations to reduce read latency while training the model:

In []:

```
train_ds = train_ds.cache().prefetch(AUTOTUNE)
val_ds = val_ds.cache().prefetch(AUTOTUNE)
```

For the model, you'll use a simple convolutional neural network (CNN), since you have transformed the audio files into spectrogram images.

Your `tf.keras.Sequential` model will use the following Keras preprocessing layers:

- `tf.keras.layers.Resizing` : to downsample the input to enable the model to train faster.
- `tf.keras.layers.Normalization` : to normalize each pixel in the image based on its mean and standard deviation.

For the `Normalization` layer, its `adapt` method would first need to be called on the training data in order to compute aggregate statistics (that is, the mean and the standard deviation).

In []:

```
for spectrogram, _ in spectrogram_ds.take(1):
    input_shape = spectrogram.shape
print('Input shape:', input_shape)
num_labels = len(commands)

# Instantiate the `tf.keras.layers.Normalization` layer.
norm_layer = layers.Normalization()
# Fit the state of the layer to the spectrograms
# with `Normalization.adapt`.
norm_layer.adapt(data=spectrogram_ds.map(map_func=lambda spec, label: spec))

model = models.Sequential([
    layers.Input(shape=input_shape),
    # Downsample the input.
    layers.Resizing(32, 32),
    # Normalize.
    norm_layer,
    layers.Conv2D(32, 3, activation='relu'),
    layers.Conv2D(64, 3, activation='relu'),
    layers.MaxPooling2D(),
    layers.Dropout(0.25),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(num_labels),
])

model.summary()
```

Configure the Keras model with the Adam optimizer and the cross-entropy loss:

In []:

```
model.compile(  
    optimizer=tf.keras.optimizers.Adam(),  
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),  
    metrics=['accuracy'],  
)
```

Train the model over 10 epochs for demonstration purposes:

In []:

```
EPOCHS = 10  
history = model.fit(  
    train_ds,  
    validation_data=val_ds,  
    epochs=EPOCHS,  
    callbacks=tf.keras.callbacks.EarlyStopping(verbose=1, patience=2),  
)
```

Let's plot the training and validation loss curves to check how your model has improved during training:

In []:

```
metrics = history.history  
plt.plot(history.epoch, metrics['loss'], metrics['val_loss'])  
plt.legend(['loss', 'val_loss'])  
plt.show()
```

Evaluate the model performance

Run the model on the test set and check the model's performance:

In []:

```
test_audio = []  
test_labels = []  
  
for audio, label in test_ds:  
    test_audio.append(audio.numpy())  
    test_labels.append(label.numpy())  
  
test_audio = np.array(test_audio)  
test_labels = np.array(test_labels)
```

In []:

```
y_pred = np.argmax(model.predict(test_audio), axis=1)  
y_true = test_labels  
  
test_acc = sum(y_pred == y_true) / len(y_true)  
print(f'Test set accuracy: {test_acc:.0%}')
```

Display a confusion matrix

Use a [confusion matrix \(<https://developers.google.com/machine-learning/glossary#confusion-matrix>\)](https://developers.google.com/machine-learning/glossary#confusion-matrix) to check how well the model did classifying each of the commands in the test set:

In []:

```
confusion_mtx = tf.math.confusion_matrix(y_true, y_pred)  
plt.figure(figsize=(10, 8))  
sns.heatmap(confusion_mtx,  
            xticklabels=commands,  
            yticklabels=commands,  
            annot=True, fmt='g')  
plt.xlabel('Prediction')  
plt.ylabel('Label')  
plt.show()
```

Run inference on an audio file

Finally, verify the model's prediction output using an input audio file of someone saying "no". How well does your model perform?

In []:

```
sample_file = data_dir/'no/01bb6a2a_nohash_0.wav'
sample_ds = preprocess_dataset([str(sample_file)])

for spectrogram, label in sample_ds.batch(1):
    prediction = model(spectrogram)
    plt.bar(commands, tf.nn.softmax(prediction[0]))
    plt.title(f'Predictions for "{commands[label[0]]}"')
    plt.show()
```

As the output suggests, your model should have recognized the audio command as "no".

Next steps

This tutorial demonstrated how to carry out simple audio classification/automatic speech recognition using a convolutional neural network with TensorFlow and Python. To learn more, consider the following resources:

- The [Sound classification with YAMNet](https://www.tensorflow.org/hub/tutorials/yamnet) (<https://www.tensorflow.org/hub/tutorials/yamnet>) tutorial shows how to use transfer learning for audio classification.
- The notebooks from [Kaggle's TensorFlow speech recognition challenge](https://www.kaggle.com/c/tensorflow-speech-recognition-challenge/overview) (<https://www.kaggle.com/c/tensorflow-speech-recognition-challenge/overview>).
- The [TensorFlow.js - Audio recognition using transfer learning codelab](https://codelabs.developers.google.com/codelabs/tensorflowjs-audio-codelab/index.html#0) (<https://codelabs.developers.google.com/codelabs/tensorflowjs-audio-codelab/index.html#0>) teaches how to build your own interactive web app for audio classification.
- [A tutorial on deep learning for music information retrieval](https://arxiv.org/abs/1709.04396) (<https://arxiv.org/abs/1709.04396>) (Choi et al., 2017) on arXiv.
- TensorFlow also has additional support for [audio data preparation and augmentation](https://www.tensorflow.org/io/tutorials/audio) (<https://www.tensorflow.org/io/tutorials/audio>) to help with your own audio-based projects.
- Consider using the [librosa](https://librosa.org/) (<https://librosa.org/>) library—a Python package for music and audio analysis.

Copyright 2021 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/audio/transfer_learning_audio)



[Run in G](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/audio/transfer_learning.ipynb)

Transfer learning with YAMNet for environmental sound classification

YAMNet (<https://tfhub.dev/google/yamnet/1>) is a pre-trained deep neural network that can predict audio events from [521 classes](#) (https://github.com/tensorflow/models/blob/master/research/audioset/yamnet/yamnet_class_map.csv), such as laughter, barking, or a siren.

In this tutorial you will learn how to:

- Load and use the YAMNet model for inference.
- Build a new model using the YAMNet embeddings to classify cat and dog sounds.
- Evaluate and export your model.

Import TensorFlow and other libraries

Start by installing [TensorFlow I/O](https://www.tensorflow.org/io) (<https://www.tensorflow.org/io>), which will make it easier for you to load audio files off disk.

In []:

```
!pip install tensorflow_io
```

```
In [ ]:
```

```
import os  
  
from IPython import display  
import matplotlib.pyplot as plt  
import numpy as np  
import pandas as pd  
  
import tensorflow as tf  
import tensorflow_hub as hub  
import tensorflow_io as tfio
```

About YAMNet

YAMNet (<https://github.com/tensorflow/models/tree/master/research/audioset/yamnet>) is a pre-trained neural network that employs the [MobileNetV1](#) (<https://arxiv.org/abs/1704.04861>) depthwise-separable convolution architecture. It can use an audio waveform as input and make independent predictions for each of the 521 audio events from the [AudioSet](#) (<http://g.co/audioset>) corpus.

Internally, the model extracts "frames" from the audio signal and processes batches of these frames. This version of the model uses frames that are 0.96 second long and extracts one frame every 0.48 seconds .

The model accepts a 1-D float32 Tensor or NumPy array containing a waveform of arbitrary length, represented as single-channel (mono) 16 kHz samples in the range [-1.0, +1.0] . This tutorial contains code to help you convert WAV files into the supported format.

The model returns 3 outputs, including the class scores, embeddings (which you will use for transfer learning), and the log mel [spectrogram](#) (https://www.tensorflow.org/tutorials/audio/simple_audio#spectrogram). You can find more details [here](#) (<https://tfhub.dev/google/yamnet/1>).

One specific use of YAMNet is as a high-level feature extractor - the 1,024-dimensional embedding output. You will use the base (YAMNet) model's input features and feed them into your shallower model consisting of one hidden `tf.keras.layers.Dense` layer. Then, you will train the network on a small amount of data for audio classification *without* requiring a lot of labeled data and training end-to-end. (This is similar to [transfer learning for image classification with TensorFlow Hub](#) (https://www.tensorflow.org/tutorials/images/transfer_learning_with_hub) for more information.)

First, you will test the model and see the results of classifying audio. You will then construct the data pre-processing pipeline.

Loading YAMNet from TensorFlow Hub

You are going to use a pre-trained YAMNet from [Tensorflow Hub](#) (<https://tfhub.dev/>) to extract the embeddings from the sound files.

Loading a model from TensorFlow Hub is straightforward: choose the model, copy its URL, and use the `load` function.

Note: to read the documentation of the model, use the model URL in your browser.

```
In [ ]:
```

```
yamnet_model_handle = 'https://tfhub.dev/google/yamnet/1'  
yamnet_model = hub.load(yamnet_model_handle)
```

With the model loaded, you can follow the [YAMNet basic usage tutorial](#) (<https://www.tensorflow.org/hub/tutorials/yamnet>) and download a sample WAV file to run the inference.

```
In [ ]:
```

```
testing_wav_file_name = tf.keras.utils.get_file('miaow_16k.wav',  
                                              'https://storage.googleapis.com/audioset/miaow_16k.wav',  
                                              cache_dir='./',  
                                              cache_subdir='test_data')  
  
print(testing_wav_file_name)
```

You will need a function to load audio files, which will also be used later when working with the training data. (Learn more about reading audio files and their labels in [Simple audio recognition](#) (https://www.tensorflow.org/tutorials/audio/simple_audio#reading_audio_files_and_their_labels)).

Note: The returned `wav_data` from `load_wav_16k_mono` is already normalized to values in the [-1.0, 1.0] range (for more information, go to [YAMNet's documentation on TF Hub](#) (<https://tfhub.dev/google/yamnet/1>)).

In []:

```
# Utility functions for loading audio files and making sure the sample rate is correct.

@tf.function
def load_wav_16k_mono(filename):
    """ Load a WAV file, convert it to a float tensor, resample to 16 kHz single-channel audio. """
    file_contents = tf.io.read_file(filename)
    wav, sample_rate = tf.audio.decode_wav(
        file_contents,
        desired_channels=1)
    wav = tf.squeeze(wav, axis=-1)
    sample_rate = tf.cast(sample_rate, dtype=tf.int64)
    wav = tfio.audio.resample(wav, rate_in=sample_rate, rate_out=16000)
    return wav
```

In []:

```
testing_wav_data = load_wav_16k_mono(testing_wav_file_name)
_ = plt.plot(testing_wav_data)

# Play the audio file.
display.Audio(testing_wav_data,rate=16000)
```

Load the class mapping

It's important to load the class names that YAMNet is able to recognize. The mapping file is present at `yamnet_model.class_map_path()` in the CSV format.

In []:

```
class_map_path = yamnet_model.class_map_path().numpy().decode('utf-8')
class_names = list(pd.read_csv(class_map_path)['display_name'])

for name in class_names[:20]:
    print(name)
print('...')
```

Run inference

YAMNet provides frame-level class-scores (i.e., 521 scores for every frame). In order to determine clip-level predictions, the scores can be aggregated per-class across frames (e.g., using mean or max aggregation). This is done below by `scores_np.mean(axis=0)`. Finally, to find the top-scored class at the clip-level, you take the maximum of the 521 aggregated scores.

In []:

```
scores, embeddings, spectrogram = yamnet_model(testing_wav_data)
class_scores = tf.reduce_mean(scores, axis=0)
top_class = tf.argmax(class_scores)
inferred_class = class_names[top_class]

print(f'The main sound is: {inferred_class}')
print(f'The embeddings shape: {embeddings.shape}')
```

Note: The model correctly inferred an animal sound. Your goal in this tutorial is to increase the model's accuracy for specific classes. Also, notice that the model generated 13 embeddings, 1 per frame.

ESC-50 dataset

The [ESC-50 dataset \(<https://github.com/karolpiczak/ESC-50#repository-content>\)](https://github.com/karolpiczak/ESC-50#repository-content) (Piczak, 2015 (<https://www.karolpiczak.com/papers/Piczak2015-ESC-Dataset.pdf>)) is a labeled collection of 2,000 five-second long environmental audio recordings. The dataset consists of 50 classes, with 40 examples per class.

Download the dataset and extract it.

In []:

```
_ = tf.keras.utils.get_file('esc-50.zip',
    'https://github.com/karoldvl/ESC-50/archive/master.zip',
    cache_dir='./',
    cache_subdir='datasets',
    extract=True)
```

Explore the data

The metadata for each file is specified in the csv file at `./datasets/ESC-50-master/meta/esc50.csv`

and all the audio files are in `./datasets/ESC-50-master/audio/`

You will create a pandas `DataFrame` with the mapping and use that to have a clearer view of the data.

In []:

```
esc50_csv = './datasets/ESC-50-master/meta/esc50.csv'
base_data_path = './datasets/ESC-50-master/audio/'

pd_data = pd.read_csv(esc50_csv)
pd_data.head()
```

Filter the data

Now that the data is stored in the `DataFrame`, apply some transformations:

- Filter out rows and use only the selected classes - `dog` and `cat`. If you want to use any other classes, this is where you can choose them.
- Amend the filename to have the full path. This will make loading easier later.
- Change targets to be within a specific range. In this example, `dog` will remain at `0`, but `cat` will become `1` instead of its original value of `5`.

In []:

```
my_classes = ['dog', 'cat']
map_class_to_id = {'dog':0, 'cat':1}

filtered_pd = pd_data[pd_data.category.isin(my_classes)]

class_id = filtered_pd['category'].apply(lambda name: map_class_to_id[name])
filtered_pd = filtered_pd.assign(target=class_id)

full_path = filtered_pd['filename'].apply(lambda row: os.path.join(base_data_path, row))
filtered_pd = filtered_pd.assign(filename=full_path)

filtered_pd.head(10)
```

Load the audio files and retrieve embeddings

Here you'll apply the `load_wav_16k_mono` and prepare the WAV data for the model.

When extracting embeddings from the WAV data, you get an array of shape `(N, 1024)` where `N` is the number of frames that YAMNet found (one for every 0.48 seconds of audio).

Your model will use each frame as one input. Therefore, you need to create a new column that has one frame per row. You also need to expand the labels and the `fold` column to properly reflect these new rows.

The expanded `fold` column keeps the original values. You cannot mix frames because, when performing the splits, you might end up having parts of the same audio on different splits, which would make your validation and test steps less effective.

In []:

```
filenames = filtered_pd['filename']
targets = filtered_pd['target']
folds = filtered_pd['fold']

main_ds = tf.data.Dataset.from_tensor_slices((filenames, targets, folds))
main_ds.element_spec
```

In []:

```
def load_wav_for_map(filename, label, fold):
    return load_wav_16k_mono(filename), label, fold

main_ds = main_ds.map(load_wav_for_map)
main_ds.element_spec
```

In []:

```
# applies the embedding extraction model to a wav data
def extract_embedding(wav_data, label, fold):
    """ run YAMNet to extract embedding from the wav data """
    scores, embeddings, spectrogram = yamnet_model(wav_data)
    num_embeddings = tf.shape(embeddings)[0]
    return (embeddings,
            tf.repeat(label, num_embeddings),
            tf.repeat(fold, num_embeddings))

# extract embedding
main_ds = main_ds.map(extract_embedding).unbatch()
main_ds.element_spec
```

Split the data

You will use the `fold` column to split the dataset into train, validation and test sets.

ESC-50 is arranged into five uniformly-sized cross-validation `fold`s, such that clips from the same original source are always in the same `fold` - find out more in the [ESC: Dataset for Environmental Sound Classification \(<https://www.karolpiczak.com/papers/Piczak2015-ESC-Dataset.pdf>\)](https://www.karolpiczak.com/papers/Piczak2015-ESC-Dataset.pdf) paper.

The last step is to remove the `fold` column from the dataset since you're not going to use it during training.

In []:

```
cached_ds = main_ds.cache()
train_ds = cached_ds.filter(lambda embedding, label, fold: fold < 4)
val_ds = cached_ds.filter(lambda embedding, label, fold: fold == 4)
test_ds = cached_ds.filter(lambda embedding, label, fold: fold == 5)

# remove the folds column now that it's not needed anymore
remove_fold_column = lambda embedding, label, fold: (embedding, label)

train_ds = train_ds.map(remove_fold_column)
val_ds = val_ds.map(remove_fold_column)
test_ds = test_ds.map(remove_fold_column)

train_ds = train_ds.cache().shuffle(1000).batch(32).prefetch(tf.data.AUTOTUNE)
val_ds = val_ds.cache().batch(32).prefetch(tf.data.AUTOTUNE)
test_ds = test_ds.cache().batch(32).prefetch(tf.data.AUTOTUNE)
```

Create your model

You did most of the work! Next, define a very simple [Sequential \(\[https://www.tensorflow.org/guide/keras/sequential_model\]\(https://www.tensorflow.org/guide/keras/sequential_model\)\)](https://www.tensorflow.org/guide/keras/sequential_model) model with one hidden layer and two outputs to recognize cats and dogs from sounds.

In []:

```
my_model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(1024), dtype=tf.float32,
                          name='input_embedding'),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(len(my_classes))
], name='my_model')

my_model.summary()
```

In []:

```
my_model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                  optimizer="adam",
                  metrics=['accuracy'])

callback = tf.keras.callbacks.EarlyStopping(monitor='loss',
                                             patience=3,
                                             restore_best_weights=True)
```

In []:

```
history = my_model.fit(train_ds,
                       epochs=20,
                       validation_data=val_ds,
                       callbacks=callback)
```

Let's run the `evaluate` method on the test data just to be sure there's no overfitting.

In []:

```
loss, accuracy = my_model.evaluate(test_ds)  
print("Loss: ", loss)  
print("Accuracy: ", accuracy)
```

You did it!

Test your model

Next, try your model on the embedding from the previous test using YAMNet only.

In []:

```
scores, embeddings, spectrogram = yamnet_model(testing_wav_data)  
result = my_model(embeddings).numpy()  
  
inferred_class = my_classes[result.mean(axis=0).argmax()]  
print(f'The main sound is: {inferred_class}')
```

Save a model that can directly take a WAV file as input

Your model works when you give it the embeddings as input.

In a real-world scenario, you'll want to use audio data as a direct input.

To do that, you will combine YAMNet with your model into a single model that you can export for other applications.

To make it easier to use the model's result, the final layer will be a `reduce_mean` operation. When using this model for serving (which you will learn about later in the tutorial), you will need the name of the final layer. If you don't define one, TensorFlow will auto-define an incremental one that makes it hard to test, as it will keep changing every time you train the model. When using a raw TensorFlow operation, you can't assign a name to it. To address this issue, you'll create a custom layer that applies `reduce_mean` and call it '`classifier`'.

In []:

```
class ReduceMeanLayer(tf.keras.layers.Layer):  
    def __init__(self, axis=0, **kwargs):  
        super(ReduceMeanLayer, self).__init__(**kwargs)  
        self.axis = axis  
  
    def call(self, input):  
        return tf.math.reduce_mean(input, axis=self.axis)
```

In []:

```
saved_model_path = './dogs_and_cats_yamnet'  
  
input_segment = tf.keras.layers.Input(shape=(), dtype=tf.float32, name='audio')  
embedding_extraction_layer = hub.KerasLayer(yamnet_model_handle,  
                                             trainable=False, name='yamnet')  
_, embeddings_output, _ = embedding_extraction_layer(input_segment)  
serving_outputs = my_model(embeddings_output)  
serving_outputs = ReduceMeanLayer(axis=0, name='classifier')(serving_outputs)  
serving_model = tf.keras.Model(input_segment, serving_outputs)  
serving_model.save(saved_model_path, include_optimizer=False)
```

In []:

```
tf.keras.utils.plot_model(serving_model)
```

Load your saved model to verify that it works as expected.

In []:

```
reloaded_model = tf.saved_model.load(saved_model_path)
```

And for the final test: given some sound data, does your model return the correct result?

In []:

```
reloaded_results = reloaded_model(testing_wav_data)  
cat_or_dog = my_classes[tf.argmax(reloaded_results)]  
print(f'The main sound is: {cat_or_dog}')
```

If you want to try your new model on a serving setup, you can use the 'serving_default' signature.

In []:

```
serving_results = reloaded_model.signatures['serving_default'](testing_wav_data)
cat_or_dog = my_classes[tf.argmax(serving_results['classifier'])]
print(f'The main sound is: {cat_or_dog}')
```

(Optional) Some more testing

The model is ready.

Let's compare it to YAMNet on the test dataset.

In []:

```
test_pd = filtered_pd.loc[filtered_pd['fold'] == 5]
row = test_pd.sample(1)
filename = row['filename'].item()
print(filename)
waveform = load_wav_16k_mono(filename)
print(f'Waveform values: {waveform}')
_ = plt.plot(waveform)

display.Audio(waveform, rate=16000)
```

In []:

```
# Run the model, check the output.
scores, embeddings, spectrogram = yamnet_model(waveform)
class_scores = tf.reduce_mean(scores, axis=0)
top_class = tf.argmax(class_scores)
inferred_class = class_names[top_class]
top_score = class_scores[top_class]
print(f'[YAMNet] The main sound is: {inferred_class} ({top_score})')

reloaded_results = reloaded_model(waveform)
your_top_class = tf.argmax(reloaded_results)
your_inferred_class = my_classes[your_top_class]
class_probabilities = tf.nn.softmax(reloaded_results, axis=-1)
your_top_score = class_probabilities[your_top_class]
print(f'[Your model] The main sound is: {your_inferred_class} ({your_top_score})')
```

Next steps

You have created a model that can classify sounds from dogs or cats. With the same idea and a different dataset you can try, for example, building an [acoustic identifier of birds](https://www.kaggle.com/c/birdclef-2021/) (<https://www.kaggle.com/c/birdclef-2021/>) based on their singing.

Share your project with the TensorFlow team on social media!

Copyright 2021 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Generate music with an RNN



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/audio/music_generation)

(https://www.tensorflow.org/tutorials/audio/music_generation)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/audio/music_generation.ipynb)

(https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/audio/music_generation.ipynb)

This tutorial shows you how to generate musical notes using a simple RNN. You will train a model using a collection of piano MIDI files from the [MAESTRO dataset](https://magenta.tensorflow.org/datasets/maestro) (<https://magenta.tensorflow.org/datasets/maestro>). Given a sequence of notes, your model will learn to predict the next note in the sequence. You can generate a longer sequences of notes by calling the model repeatedly.

This tutorial contains complete code to parse and create MIDI files. You can learn more about how RNNs work by visiting [Text generation with an RNN](https://www.tensorflow.org/text/tutorials/text_generation) (https://www.tensorflow.org/text/tutorials/text_generation).

Setup

This tutorial uses the `pretty_midi` (<https://github.com/craffel/pretty-midi>) library to create and parse MIDI files, and `pyfluidsynth` (<https://github.com/nwhitehead/pyfluidsynth>) for generating audio playback in Colab.

In []:

```
!sudo apt install -y fluidsynth
```

In []:

```
!pip install --upgrade pyfluidsynth
```

In []:

```
!pip install pretty_midi
```

In []:

```
import collections
import datetime
import fluidsynth
import glob
import numpy as np
import pathlib
import pandas as pd
import pretty_midi
import seaborn as sns
import tensorflow as tf

from IPython import display
from matplotlib import pyplot as plt
from typing import Dict, List, Optional, Sequence, Tuple
```

In []:

```
seed = 42
tf.random.set_seed(seed)
np.random.seed(seed)

# Sampling rate for audio playback
_SAMPLING_RATE = 16000
```

Download the Maestro dataset

In []:

```
data_dir = pathlib.Path('data/maestro-v2.0.0')
if not data_dir.exists():
    tf.keras.utils.get_file(
        'maestro-v2.0.0-midi.zip',
        origin='https://storage.googleapis.com/magentadata/datasets/maestro/v2.0.0/maestro-v2.0.0-midi.zip',
        extract=True,
        cache_dir='.', cache_subdir='data',
    )
```

The dataset contains about 1,200 MIDI files.

In []:

```
filenames = glob.glob(str(data_dir/'**/*.mid'))
print('Number of files:', len(filenames))
```

Process a MIDI file

First, use `pretty_midi` to parse a single MIDI file and inspect the format of the notes. If you would like to download the MIDI file below to play on your computer, you can do so in colab by writing `files.download(sample_file)`.

In []:

```
sample_file = filenames[1]
print(sample_file)
```

Generate a `PrettyMIDI` object for the sample MIDI file.

In []:

```
pm = pretty_midi.PrettyMIDI(sample_file)
```

Play the sample file. The playback widget may take several seconds to load.

In []:

```
def display_audio(pm: pretty_midi.PrettyMIDI, seconds=30):
    waveform = pm.fluidsynth(fs=_SAMPLING_RATE)
    # Take a sample of the generated waveform to mitigate kernel resets
    waveform_short = waveform[:seconds*_SAMPLING_RATE]
    return display.Audio(waveform_short, rate=_SAMPLING_RATE)
```

In []:

```
display_audio(pm)
```

Do some inspection on the MIDI file. What kinds of instruments are used?

In []:

```
print('Number of instruments:', len(pm.instruments))
instrument = pm.instruments[0]
instrument_name = pretty_midi.program_to_instrument_name(instrument.program)
print('Instrument name:', instrument_name)
```

Extract notes

In []:

```
for i, note in enumerate(instrument.notes[:10]):
    note_name = pretty_midi.note_number_to_name(note.pitch)
    duration = note.end - note.start
    print(f'{i}: pitch={note.pitch}, note_name={note_name},'
          f' duration={duration:.4f}')
```

You will use three variables to represent a note when training the model: `pitch`, `step` and `duration`. The `pitch` is the perceptual quality of the sound as a MIDI note number. The `step` is the time elapsed from the previous note or start of the track. The `duration` is how long the note will be playing in seconds and is the difference between the note end and note start times.

Extract the notes from the sample MIDI file.

```
In [ ]:
```

```
def midi_to_notes(midi_file: str) -> pd.DataFrame:
    pm = pretty_midi.PrettyMIDI(midi_file)
    instrument = pm.instruments[0]
    notes = collections.defaultdict(list)

    # Sort the notes by start time
    sorted_notes = sorted(instrument.notes, key=lambda note: note.start)
    prev_start = sorted_notes[0].start

    for note in sorted_notes:
        start = note.start
        end = note.end
        notes['pitch'].append(note.pitch)
        notes['start'].append(start)
        notes['end'].append(end)
        notes['step'].append(start - prev_start)
        notes['duration'].append(end - start)
        prev_start = start

    return pd.DataFrame({name: np.array(value) for name, value in notes.items()})
```

```
In [ ]:
```

```
raw_notes = midi_to_notes(sample_file)
raw_notes.head()
```

It may be easier to interpret the note names rather than the pitches, so you can use the function below to convert from the numeric pitch values to note names. The note name shows the type of note, accidental and octave number (e.g. C#4).

```
In [ ]:
```

```
get_note_names = np.vectorize(pretty_midi.note_number_to_name)
sample_note_names = get_note_names(raw_notes['pitch'])
sample_note_names[:10]
```

To visualize the musical piece, plot the note pitch, start and end across the length of the track (i.e. piano roll). Start with the first 100 notes

```
In [ ]:
```

```
def plot_piano_roll(notes: pd.DataFrame, count: Optional[int] = None):
    if count:
        title = f'First {count} notes'
    else:
        title = f'Whole track'
    count = len(notes['pitch'])
    plt.figure(figsize=(20, 4))
    plot_pitch = np.stack([notes['pitch'], notes['pitch']], axis=0)
    plot_start_stop = np.stack([notes['start'], notes['end']], axis=0)
    plt.plot(
        plot_start_stop[:, :count], plot_pitch[:, :count], color="b", marker=".")
    plt.xlabel('Time [s]')
    plt.ylabel('Pitch')
    _ = plt.title(title)
```

```
In [ ]:
```

```
plot_piano_roll(raw_notes, count=100)
```

Plot the notes for the entire track.

```
In [ ]:
```

```
plot_piano_roll(raw_notes)
```

Check the distribution of each note variable.

```
In [ ]:
```

```
def plot_distributions(notes: pd.DataFrame, drop_percentile=2.5):
    plt.figure(figsize=[15, 5])
    plt.subplot(1, 3, 1)
    sns.histplot(notes, x="pitch", bins=20)

    plt.subplot(1, 3, 2)
    max_step = np.percentile(notes['step'], 100 - drop_percentile)
    sns.histplot(notes, x="step", bins=np.linspace(0, max_step, 21))

    plt.subplot(1, 3, 3)
    max_duration = np.percentile(notes['duration'], 100 - drop_percentile)
    sns.histplot(notes, x="duration", bins=np.linspace(0, max_duration, 21))
```

```
In [ ]:
```

```
plot_distributions(raw_notes)
```

Create a MIDI file

You can generate your own MIDI file from a list of notes using the function below.

```
In [ ]:
```

```
def notes_to_midi(
    notes: pd.DataFrame,
    out_file: str,
    instrument_name: str,
    velocity: int = 100, # note loudness
) -> pretty_midi.PrettyMIDI:

    pm = pretty_midi.PrettyMIDI()
    instrument = pretty_midi.Instrument(
        program=pretty_midi.instrument_name_to_program(
            instrument_name))
    prev_start = 0
    for i, note in notes.iterrows():
        start = float(prev_start + note['step'])
        end = float(start + note['duration'])
        note = pretty_midi.Note(
            velocity=velocity,
            pitch=int(note['pitch']),
            start=start,
            end=end,
        )
        instrument.notes.append(note)
        prev_start = start
    pm.instruments.append(instrument)
    pm.write(out_file)
    return pm
```

```
In [ ]:
```

```
example_file = 'example.midi'
example_pm = notes_to_midi(
    raw_notes, out_file=example_file, instrument_name=instrument_name)
```

Play the generated MIDI file and see if there is any difference.

```
In [ ]:
```

```
display_audio(example_pm)
```

As before, you can write `files.download(example_file)` to download and play this file.

Create the training dataset

Create the training dataset by extracting notes from the MIDI files. You can start by using a small number of files, and experiment later with more. This may take a couple minutes.

```
In [ ]:
```

```
num_files = 5
all_notes = []
for f in filenames[:num_files]:
    notes = midi_to_notes(f)
    all_notes.append(notes)

all_notes = pd.concat(all_notes)
```

```
In [ ]:
```

```
n_notes = len(all_notes)
print('Number of notes parsed:', n_notes)
```

Next, create a [tf.data.Dataset](#) (<https://www.tensorflow.org/datasets>) from the parsed notes.

```
In [ ]:
```

```
key_order = ['pitch', 'step', 'duration']
train_notes = np.stack([all_notes[key] for key in key_order], axis=1)
```

```
In [ ]:
```

```
notes_ds = tf.data.Dataset.from_tensor_slices(train_notes)
notes_ds.element_spec
```

You will train the model on batches of sequences of notes. Each example will consist of a sequence of notes as the input features, and next note as the label. In this way, the model will be trained to predict the next note in a sequence. You can find a diagram explaining this process (and more details) in [Text classification with an RNN](#) (https://www.tensorflow.org/text/tutorials/text_generation).

You can use the handy [window](#) (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#window) function with size `seq_length` to create the features and labels in this format.

```
In [ ]:
```

```
def create_sequences(
    dataset: tf.data.Dataset,
    seq_length: int,
    vocab_size = 128,
) -> tf.data.Dataset:
    """Returns TF Dataset of sequence and label examples."""
    seq_length = seq_length+1

    # Take 1 extra for the labels
    windows = dataset.window(seq_length, shift=1, stride=1,
                             drop_remainder=True)

    # `flat_map` flattens the "dataset of datasets" into a dataset of tensors
    flatten = lambda x: x.batch(seq_length, drop_remainder=True)
    sequences = windows.flat_map(flatten)

    # Normalize note pitch
    def scale_pitch(x):
        x = x/[vocab_size,1.0,1.0]
        return x

    # Split the labels
    def split_labels(sequences):
        inputs = sequences[:-1]
        labels_dense = sequences[-1]
        labels = {key:labels_dense[i] for i,key in enumerate(key_order)}

        return scale_pitch(inputs), labels

    return sequences.map(split_labels, num_parallel_calls=tf.data.AUTOTUNE)
```

Set the sequence length for each example. Experiment with different lengths (e.g. 50, 100, 150) to see which one works best for the data, or use [hyperparameter tuning](#) (https://www.tensorflow.org/tutorials/keras/keras_tuner). The size of the vocabulary (`vocab_size`) is set to 128 representing all the pitches supported by `pretty_midi`.

```
In [ ]:
```

```
seq_length = 25
vocab_size = 128
seq_ds = create_sequences(notes_ds, seq_length, vocab_size)
seq_ds.element_spec
```

The shape of the dataset is `(100, 1)` , meaning that the model will take 100 notes as input, and learn to predict the following note as output.

In []:

```
for seq, target in seq_ds.take(1):
    print('sequence shape:', seq.shape)
    print('sequence elements (first 10):', seq[0: 10])
    print()
    print('target:', target)
```

Batch the examples, and configure the dataset for performance.

In []:

```
batch_size = 64
buffer_size = n_notes - seq_length # the number of items in the dataset
train_ds = (seq_ds
            .shuffle(buffer_size)
            .batch(batch_size, drop_remainder=True)
            .cache()
            .prefetch(tf.data.experimental.AUTOTUNE))
```

In []:

```
train_ds.element_spec
```

Create and train the model

The model will have three outputs, one for each note variable. For `pitch` and `duration` , you will use a custom loss function based on mean squared error that encourages the model to output non-negative values.

In []:

```
def mse_with_positive_pressure(y_true: tf.Tensor, y_pred: tf.Tensor):
    mse = (y_true - y_pred) ** 2
    positive_pressure = 10 * tf.maximum(-y_pred, 0.0)
    return tf.reduce_mean(mse + positive_pressure)
```

In []:

```
input_shape = (seq_length, 3)
learning_rate = 0.005

inputs = tf.keras.Input(input_shape)
x = tf.keras.layers.LSTM(128)(inputs)

outputs = {
    'pitch': tf.keras.layers.Dense(128, name='pitch')(x),
    'step': tf.keras.layers.Dense(1, name='step')(x),
    'duration': tf.keras.layers.Dense(1, name='duration')(x),
}

model = tf.keras.Model(inputs, outputs)

loss = {
    'pitch': tf.keras.losses.SparseCategoricalCrossentropy(
        from_logits=True),
    'step': mse_with_positive_pressure,
    'duration': mse_with_positive_pressure,
}

optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
model.compile(loss=loss, optimizer=optimizer)
model.summary()
```

Testing the `model.evaluate` function, you can see that the `pitch` loss is significantly greater than the `step` and `duration` losses. Note that `loss` is the total loss computed by summing all the other losses and is currently dominated by the `pitch` loss.

In []:

```
losses = model.evaluate(train_ds, return_dict=True)
losses
```

One way to balance this is to use the `loss_weights` argument to compile:

In []:

```
model.compile(  
    loss=loss,  
    loss_weights={  
        'pitch': 0.05,  
        'step': 1.0,  
        'duration':1.0,  
    },  
    optimizer=optimizer,  
)
```

The `loss` then becomes the weighted sum of the individual losses.

In []:

```
model.evaluate(train_ds, return_dict=True)
```

Train the model.

In []:

```
callbacks = [  
    tf.keras.callbacks.ModelCheckpoint(  
        filepath='./training_checkpoints/ckpt_{epoch}',  
        save_weights_only=True),  
    tf.keras.callbacks.EarlyStopping(  
        monitor='loss',  
        patience=5,  
        verbose=1,  
        restore_best_weights=True),  
]
```

In []:

```
%%time  
epochs = 50  
  
history = model.fit(  
    train_ds,  
    epochs=epochs,  
    callbacks=callbacks,  
)
```

In []:

```
plt.plot(history.epoch, history.history['loss'], label='total loss')  
plt.show()
```

Generate notes

To use the model to generate notes, you will first need to provide a starting sequence of notes. The function below generates one note from a sequence of notes.

For note pitch, it draws a sample from softmax distribution of notes produced by the model, and does not simply pick the note with the highest probability. Always picking the note with the highest probability would lead to repetitive sequences of notes being generated.

The `temperature` parameter can be used to control the randomness of notes generated. You can find more details on temperature in [Text generation with an RNN](#) (https://www.tensorflow.org/text/tutorials/text_generation).

In []:

```
def predict_next_note(
    notes: np.ndarray,
    keras_model: tf.keras.Model,
    temperature: float = 1.0) -> int:
    """Generates a note IDs using a trained sequence model."""

    assert temperature > 0

    # Add batch dimension
    inputs = tf.expand_dims(notes, 0)

    predictions = model.predict(inputs)
    pitch_logits = predictions['pitch']
    step = predictions['step']
    duration = predictions['duration']

    pitch_logits /= temperature
    pitch = tf.random.categorical(pitch_logits, num_samples=1)
    pitch = tf.squeeze(pitch, axis=-1)
    duration = tf.squeeze(duration, axis=-1)
    step = tf.squeeze(step, axis=-1)

    # `step` and `duration` values should be non-negative
    step = tf.maximum(0, step)
    duration = tf.maximum(0, duration)

    return int(pitch), float(step), float(duration)
```

Now generate some notes. You can play around with temperature and the starting sequence in `next_notes` and see what happens.

In []:

```
temperature = 2.0
num_predictions = 120

sample_notes = np.stack([raw_notes[key] for key in key_order], axis=1)

# The initial sequence of notes; pitch is normalized similar to training
# sequences
input_notes = (
    sample_notes[:seq_length] / np.array([vocab_size, 1, 1]))

generated_notes = []
prev_start = 0
for _ in range(num_predictions):
    pitch, step, duration = predict_next_note(input_notes, model, temperature)
    start = prev_start + step
    end = start + duration
    input_note = (pitch, step, duration)
    generated_notes.append((*input_note, start, end))
    input_notes = np.delete(input_notes, 0, axis=0)
    input_notes = np.append(input_notes, np.expand_dims(input_note, 0), axis=0)
    prev_start = start

generated_notes = pd.DataFrame(
    generated_notes, columns=(*key_order, 'start', 'end'))
```

In []:

```
generated_notes.head(10)
```

In []:

```
out_file = 'output.mid'
out_pm = notes_to_midi(
    generated_notes, out_file=out_file, instrument_name=instrument_name)
display_audio(out_pm)
```

You can also download the audio file by adding the two lines below:

```
from google.colab import files
files.download(out_file)
```

Visualize the generated notes.

In []:

```
plot_piano_roll(generated_notes)
```

Check the distributions of pitch , step and duration .

In []:

```
plot_distributions(generated_notes)
```

In the above plots, you will notice the change in distribution of the note variables. Since there is a feedback loop between the model's outputs and inputs, the model tends to generate similar sequences of outputs to reduce the loss. This is particularly relevant for step and duration , which has uses MSE loss. For pitch , you can increase the randomness by increasing the temperature in predict_next_note .

Next steps

This tutorial demonstrated the mechanics of using an RNN to generate sequences of notes from a dataset of MIDI files. To learn more, you can visit the closely related [Text generation with an RNN](https://www.tensorflow.org/text/tutorials/text_generation) (https://www.tensorflow.org/text/tutorials/text_generation) tutorial, which contains additional diagrams and explanations.

An alternative to using RNNs for music generation is using GANs. Rather than generating audio, a GAN-based approach can generate a entire sequence in parallel. The Magenta team has done impressive work on this approach with [GANSynth](https://magenta.tensorflow.org/gansynth) (<https://magenta.tensorflow.org/gansynth>). You can also find many wonderful music and art projects and open-source code on [Magenta project website](https://magenta.tensorflow.org/) (<https://magenta.tensorflow.org/>).

Copyright 2019 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Classification on imbalanced data



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/structured_data/imbalanced_data)



(https://www.tensorflow.org/tutorials/structured_data/imbalanced_data) (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/structured_data/imba)

R

This tutorial demonstrates how to classify a highly imbalanced dataset in which the number of examples in one class greatly outnumbers the examples in another. You will work with the [Credit Card Fraud Detection](https://www.kaggle.com/mlg-ulb/creditcardfraud) (<https://www.kaggle.com/mlg-ulb/creditcardfraud>) dataset hosted on Kaggle. The aim is to detect a mere 492 fraudulent transactions from 284,807 transactions in total. You will use [Keras](https://www.tensorflow.org/guide/keras/overview) (<https://www.tensorflow.org/guide/keras/overview>) to define the model and [class weights](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/Model) (https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/Model) to help the model learn from the imbalanced data..

This tutorial contains complete code to:

- Load a CSV file using Pandas.
- Create train, validation, and test sets.
- Define and train a model using Keras (including setting class weights).
- Evaluate the model using various metrics (including precision and recall).
- Try common techniques for dealing with imbalanced data like:
 - Class weighting
 - Oversampling

Setup

```
In [ ]:
```

```
import tensorflow as tf
from tensorflow import keras

import os
import tempfile

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns

import sklearn
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

```
In [ ]:
```

```
mpl.rcParams['figure.figsize'] = (12, 10)
colors = plt.rcParams['axes.prop_cycle'].by_key()['color']
```

Data processing and exploration

Download the Kaggle Credit Card Fraud data set

Pandas is a Python library with many helpful utilities for loading and working with structured data. It can be used to download CSVs into a Pandas DataFrame (<https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html#pandas.DataFrame>).

Note: This dataset has been collected and analysed during a research collaboration of Worldline and the [Machine Learning Group \(http://mlg.ulb.ac.be\)](http://mlg.ulb.ac.be) of ULB (Université Libre de Bruxelles) on big data mining and fraud detection. More details on current and past projects on related topics are available [here](https://www.researchgate.net/project/Fraud-detection-5) (<https://www.researchgate.net/project/Fraud-detection-5>) and the page of the [DefeatFraud](https://mlg.ulb.ac.be/wordpress/portfolio_page/defeatfraud-assessment-and-validation-of-deep-feature-engineering-and-learning-solutions-for-fraud-detection/) (https://mlg.ulb.ac.be/wordpress/portfolio_page/defeatfraud-assessment-and-validation-of-deep-feature-engineering-and-learning-solutions-for-fraud-detection/) project

```
In [ ]:
```

```
file = tf.keras.utils
raw_df = pd.read_csv('https://storage.googleapis.com/download.tensorflow.org/data/creditcard.csv')
raw_df.head()
```

```
In [ ]:
```

```
raw_df[['Time', 'V1', 'V2', 'V3', 'V4', 'V5', 'V26', 'V27', 'V28', 'Amount', 'Class']].describe()
```

Examine the class label imbalance

Let's look at the dataset imbalance:

```
In [ ]:
```

```
neg, pos = np.bincount(raw_df['Class'])
total = neg + pos
print('Examples:\n    Total: {}\n    Positive: {} ({:.2f}% of total)\n'.format(
    total, pos, 100 * pos / total))
```

This shows the small fraction of positive samples.

Clean, split and normalize the data

The raw data has a few issues. First the `Time` and `Amount` columns are too variable to use directly. Drop the `Time` column (since it's not clear what it means) and take the log of the `Amount` column to reduce its range.

```
In [ ]:
```

```
cleaned_df = raw_df.copy()

# You don't want the `Time` column.
cleaned_df.pop('Time')

# The `Amount` column covers a huge range. Convert to log-space.
eps = 0.001 # 0 => 0.1¢
cleaned_df['Log Amount'] = np.log(cleaned_df.pop('Amount'))+eps
```

Split the dataset into train, validation, and test sets. The validation set is used during the model fitting to evaluate the loss and any metrics, however the model is not fit with this data. The test set is completely unused during the training phase and is only used at the end to evaluate how well the model generalizes to new data. This is especially important with imbalanced datasets where [overfitting](https://developers.google.com/machine-learning/crash-course/generalization/peril-of-overfitting) (<https://developers.google.com/machine-learning/crash-course/generalization/peril-of-overfitting>) is a significant concern from the lack of training data.

In []:

```
# Use a utility from sklearn to split and shuffle your dataset.
train_df, test_df = train_test_split(cleaned_df, test_size=0.2)
train_df, val_df = train_test_split(train_df, test_size=0.2)

# Form np arrays of labels and features.
train_labels = np.array(train_df.pop('Class'))
bool_train_labels = train_labels != 0
val_labels = np.array(val_df.pop('Class'))
test_labels = np.array(test_df.pop('Class'))

train_features = np.array(train_df)
val_features = np.array(val_df)
test_features = np.array(test_df)
```

Normalize the input features using the sklearn StandardScaler. This will set the mean to 0 and standard deviation to 1.

Note: The `StandardScaler` is only fit using the `train_features` to be sure the model is not peeking at the validation or test sets.

In []:

```
scaler = StandardScaler()
train_features = scaler.fit_transform(train_features)

val_features = scaler.transform(val_features)
test_features = scaler.transform(test_features)

train_features = np.clip(train_features, -5, 5)
val_features = np.clip(val_features, -5, 5)
test_features = np.clip(test_features, -5, 5)

print('Training labels shape:', train_labels.shape)
print('Validation labels shape:', val_labels.shape)
print('Test labels shape:', test_labels.shape)

print('Training features shape:', train_features.shape)
print('Validation features shape:', val_features.shape)
print('Test features shape:', test_features.shape)
```

Caution: If you want to deploy a model, it's critical that you preserve the preprocessing calculations. The easiest way to implement them as layers, and attach them to your model before export.

Look at the data distribution

Next compare the distributions of the positive and negative examples over a few features. Good questions to ask yourself at this point are:

- Do these distributions make sense?
 - Yes. You've normalized the input and these are mostly concentrated in the +/- 2 range.
- Can you see the difference between the distributions?
 - Yes the positive examples contain a much higher rate of extreme values.

In []:

```
pos_df = pd.DataFrame(train_features[ bool_train_labels], columns=train_df.columns)
neg_df = pd.DataFrame(train_features[~bool_train_labels], columns=train_df.columns)

sns.jointplot(x=pos_df['V5'], y=pos_df['V6'],
               kind='hex', xlim=(-5,5), ylim=(-5,5))
plt.suptitle("Positive distribution")

sns.jointplot(x=neg_df['V5'], y=neg_df['V6'],
               kind='hex', xlim=(-5,5), ylim=(-5,5))
_ = plt.suptitle("Negative distribution")
```

Define the model and metrics

Define a function that creates a simple neural network with a densely connected hidden layer, a [dropout](https://developers.google.com/machine-learning/glossary/#dropout_regularization) (https://developers.google.com/machine-learning/glossary/#dropout_regularization) layer to reduce overfitting, and an output sigmoid layer that returns the probability of a transaction being fraudulent:

In []:

```
METRICS = [
    keras.metrics.TruePositives(name='tp'),
    keras.metrics.FalsePositives(name='fp'),
    keras.metrics.TrueNegatives(name='tn'),
    keras.metrics.FalseNegatives(name='fn'),
    keras.metrics.BinaryAccuracy(name='accuracy'),
    keras.metrics.Precision(name='precision'),
    keras.metrics.Recall(name='recall'),
    keras.metrics.AUC(name='auc'),
    keras.metrics.AUC(name='prc', curve='PR'), # precision-recall curve
]

def make_model(metrics=METRICS, output_bias=None):
    if output_bias is not None:
        output_bias = tf.keras.initializers.Constant(output_bias)
    model = keras.Sequential([
        keras.layers.Dense(
            16, activation='relu',
            input_shape=(train_features.shape[-1],)),
        keras.layers.Dropout(0.5),
        keras.layers.Dense(1, activation='sigmoid',
                          bias_initializer=output_bias),
    ])
    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate=1e-3),
        loss=keras.losses.BinaryCrossentropy(),
        metrics=metrics)
    return model
```

Understanding useful metrics

Notice that there are a few metrics defined above that can be computed by the model that will be helpful when evaluating the performance.

- **False** negatives and **false** positives are samples that were **incorrectly** classified
- **True** negatives and **true** positives are samples that were **correctly** classified
- **Accuracy** is the percentage of examples correctly classified

$$\frac{\text{true samples}}{\text{total samples}}$$

- **Precision** is the percentage of **predicted** positives that were correctly classified

$$\frac{\text{true positives}}{\text{true positives} + \text{false positives}}$$

- **Recall** is the percentage of **actual** positives that were correctly classified

$$\frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$$

- **AUC** refers to the Area Under the Curve of a Receiver Operating Characteristic curve (ROC-AUC). This metric is equal to the probability that a classifier will rank a random positive sample higher than a random negative sample.
- **AUPRC** refers to Area Under the Curve of the Precision-Recall Curve. This metric computes precision-recall pairs for different probability thresholds.

Note: Accuracy is not a helpful metric for this task. You can have 99.8%+ accuracy on this task by predicting False all the time.

Read more:

- [True vs. False and Positive vs. Negative](https://developers.google.com/machine-learning/crash-course/classification/true-false-positive-negative) (<https://developers.google.com/machine-learning/crash-course/classification/true-false-positive-negative>)
- [Accuracy](https://developers.google.com/machine-learning/crash-course/classification/accuracy) (<https://developers.google.com/machine-learning/crash-course/classification/accuracy>)
- [Precision and Recall](https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall) (<https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall>)
- [ROC-AUC](https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc) (<https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc>)
- [Relationship between Precision-Recall and ROC Curves](https://www.biostat.wisc.edu/~page/rocpr.pdf) (<https://www.biostat.wisc.edu/~page/rocpr.pdf>)

Baseline model

Build the model

Now create and train your model using the function that was defined earlier. Notice that the model is fit using a larger than default batch size of 2048, this is important to ensure that each batch has a decent chance of containing a few positive samples. If the batch size was too small, they would likely have no fraudulent transactions to learn from.

Note: this model will not handle the class imbalance well. You will improve it later in this tutorial.

In []:

```
EPOCHS = 100
BATCH_SIZE = 2048

early_stopping = tf.keras.callbacks.EarlyStopping(
    monitor='val_prc',
    verbose=1,
    patience=10,
    mode='max',
    restore_best_weights=True)
```

In []:

```
model = make_model()
model.summary()
```

Test run the model:

In []:

```
model.predict(train_features[:10])
```

Optional: Set the correct initial bias.

These initial guesses are not great. You know the dataset is imbalanced. Set the output layer's bias to reflect that (See: [A Recipe for Training Neural Networks: "init well"](#) (<http://karpathy.github.io/2019/04/25/recipe/#2-set-up-the-end-to-end-training-evaluation-skeleton--get-dumb-baselines>)). This can help with initial convergence.

With the default bias initialization the loss should be about `math.log(2) = 0.69314`

In []:

```
results = model.evaluate(train_features, train_labels, batch_size=BATCH_SIZE, verbose=0)
print("Loss: {:.4f}".format(results[0]))
```

The correct bias to set can be derived from:

$$p_0 = pos/(pos + neg) = 1/(1 + e^{-b_0})$$

$$b_0 = -\log_e(1/p_0 - 1)$$

$$b_0 = \log_e(pos/neg)$$

In []:

```
initial_bias = np.log([pos/neg])
initial_bias
```

Set that as the initial bias, and the model will give much more reasonable initial guesses.

It should be near: `pos/total = 0.0018`

In []:

```
model = make_model(output_bias=initial_bias)
model.predict(train_features[:10])
```

With this initialization the initial loss should be approximately:

$$-p_0 \log(p_0) - (1 - p_0) \log(1 - p_0) = 0.01317$$

In []:

```
results = model.evaluate(train_features, train_labels, batch_size=BATCH_SIZE, verbose=0)
print("Loss: {:.4f}".format(results[0]))
```

This initial loss is about 50 times less than if would have been with naive initialization.

This way the model doesn't need to spend the first few epochs just learning that positive examples are unlikely. This also makes it easier to read plots of the loss during training.

Checkpoint the initial weights

To make the various training runs more comparable, keep this initial model's weights in a checkpoint file, and load them into each model before training:

In []:

```
initial_weights = os.path.join(tempfile.mkdtemp(), 'initial_weights')
model.save_weights(initial_weights)
```

Confirm that the bias fix helps

Before moving on, confirm quick that the careful bias initialization actually helped.

Train the model for 20 epochs, with and without this careful initialization, and compare the losses:

In []:

```
model = make_model()
model.load_weights(initial_weights)
model.layers[-1].bias.assign([0.0])
zero_bias_history = model.fit(
    train_features,
    train_labels,
    batch_size=BATCH_SIZE,
    epochs=20,
    validation_data=(val_features, val_labels),
    verbose=0)
```

In []:

```
model = make_model()
model.load_weights(initial_weights)
careful_bias_history = model.fit(
    train_features,
    train_labels,
    batch_size=BATCH_SIZE,
    epochs=20,
    validation_data=(val_features, val_labels),
    verbose=0)
```

In []:

```
def plot_loss(history, label, n):
    # Use a log scale on y-axis to show the wide range of values.
    plt.semilogy(history.epoch, history.history['loss'],
                 color=colors[n], label='Train ' + label)
    plt.semilogy(history.epoch, history.history['val_loss'],
                 color=colors[n], label='Val ' + label,
                 linestyle="--")
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
```

In []:

```
plot_loss(zero_bias_history, "Zero Bias", 0)
plot_loss(careful_bias_history, "Careful Bias", 1)
```

The above figure makes it clear: In terms of validation loss, on this problem, this careful initialization gives a clear advantage.

Train the model

In []:

```
model = make_model()
model.load_weights(initial_weights)
baseline_history = model.fit(
    train_features,
    train_labels,
    batch_size=BATCH_SIZE,
    epochs=EPOCHS,
    callbacks=[early_stopping],
    validation_data=(val_features, val_labels))
```

Check training history

In this section, you will produce plots of your model's accuracy and loss on the training and validation set. These are useful to check for overfitting, which you can learn more about in the [Overfit and underfit \(\[https://www.tensorflow.org/tutorials/keras/overfit_and_underfit\]\(https://www.tensorflow.org/tutorials/keras/overfit_and_underfit\)\)](https://www.tensorflow.org/tutorials/keras/overfit_and_underfit) tutorial.

Additionally, you can produce these plots for any of the metrics you created above. False negatives are included as an example.

In []:

```
def plot_metrics(history):
    metrics = ['loss', 'prc', 'precision', 'recall']
    for n, metric in enumerate(metrics):
        name = metric.replace("_", " ").capitalize()
        plt.subplot(2, 2, n+1)
        plt.plot(history.epoch, history.history[metric], color=colors[0], label='Train')
        plt.plot(history.epoch, history.history['val_'+metric],
                 color=colors[0], linestyle="--", label='Val')
        plt.xlabel('Epoch')
        plt.ylabel(name)
        if metric == 'loss':
            plt.ylim([0, plt.ylim()[1]])
        elif metric == 'auc':
            plt.ylim([0.8, 1])
        else:
            plt.ylim([0, 1])
    plt.legend();
```

In []:

```
plot_metrics(baseline_history)
```

Note: That the validation curve generally performs better than the training curve. This is mainly caused by the fact that the dropout layer is not active when evaluating the model.

Evaluate metrics

You can use a [confusion matrix \(\[https://developers.google.com/machine-learning/glossary/#confusion_matrix\]\(https://developers.google.com/machine-learning/glossary/#confusion_matrix\)\)](https://developers.google.com/machine-learning/glossary/#confusion_matrix) to summarize the actual vs. predicted labels, where the X axis is the predicted label and the Y axis is the actual label:

In []:

```
train_predictions_baseline = model.predict(train_features, batch_size=BATCH_SIZE)
test_predictions_baseline = model.predict(test_features, batch_size=BATCH_SIZE)
```

In []:

```
def plot_cm(labels, predictions, p=0.5):
    cm = confusion_matrix(labels, predictions > p)
    plt.figure(figsize=(5,5))
    sns.heatmap(cm, annot=True, fmt="d")
    plt.title('Confusion matrix @{:2f}'.format(p))
    plt.ylabel('Actual label')
    plt.xlabel('Predicted label')

    print('Legitimate Transactions Detected (True Negatives): ', cm[0][0])
    print('Legitimate Transactions Incorrectly Detected (False Positives): ', cm[0][1])
    print('Fraudulent Transactions Missed (False Negatives): ', cm[1][0])
    print('Fraudulent Transactions Detected (True Positives): ', cm[1][1])
    print('Total Fraudulent Transactions: ', np.sum(cm[1]))
```

Evaluate your model on the test dataset and display the results for the metrics you created above:

In []:

```
baseline_results = model.evaluate(test_features, test_labels,
                                  batch_size=BATCH_SIZE, verbose=0)
for name, value in zip(model.metrics_names, baseline_results):
    print(name, ': ', value)
print()

plot_cm(test_labels, test_predictions_baseline)
```

If the model had predicted everything perfectly, this would be a [diagonal matrix](https://en.wikipedia.org/wiki/Diagonal_matrix) (https://en.wikipedia.org/wiki/Diagonal_matrix) where values off the main diagonal, indicating incorrect predictions, would be zero. In this case the matrix shows that you have relatively few false positives, meaning that there were relatively few legitimate transactions that were incorrectly flagged. However, you would likely want to have even fewer false negatives despite the cost of increasing the number of false positives. This trade off may be preferable because false negatives would allow fraudulent transactions to go through, whereas false positives may cause an email to be sent to a customer to ask them to verify their card activity.

Plot the ROC

Now plot the [ROC](https://developers.google.com/machine-learning/glossary#ROC) (<https://developers.google.com/machine-learning/glossary#ROC>). This plot is useful because it shows, at a glance, the range of performance the model can reach just by tuning the output threshold.

In []:

```
def plot_roc(name, labels, predictions, **kwargs):
    fp, tp, _ = sklearn.metrics.roc_curve(labels, predictions)

    plt.plot(100*fp, 100*tp, label=name, linewidth=2, **kwargs)
    plt.xlabel('False positives [%]')
    plt.ylabel('True positives [%]')
    plt.xlim([-0.5,20])
    plt.ylim([80,100.5])
    plt.grid(True)
    ax = plt.gca()
    ax.set_aspect('equal')
```

In []:

```
plot_roc("Train Baseline", train_labels, train_predictions_baseline, color=colors[0])
plot_roc("Test Baseline", test_labels, test_predictions_baseline, color=colors[0], linestyle='--')
plt.legend(loc='lower right');
```

Plot the AUPRC

Now plot the [AUPRC](https://developers.google.com/machine-learning/glossary?hl=en#PR_AUC) (https://developers.google.com/machine-learning/glossary?hl=en#PR_AUC). Area under the interpolated precision-recall curve, obtained by plotting (recall, precision) points for different values of the classification threshold. Depending on how it's calculated, PR AUC may be equivalent to the average precision of the model.

In []:

```
def plot_prc(name, labels, predictions, **kwargs):
    precision, recall, _ = sklearn.metrics.precision_recall_curve(labels, predictions)

    plt.plot(precision, recall, label=name, linewidth=2, **kwargs)
    plt.xlabel('Precision')
    plt.ylabel('Recall')
    plt.grid(True)
    ax = plt.gca()
    ax.set_aspect('equal')
```

In []:

```
plot_prc("Train Baseline", train_labels, train_predictions_baseline, color=colors[0])
plot_prc("Test Baseline", test_labels, test_predictions_baseline, color=colors[0], linestyle='--')
plt.legend(loc='lower right');
```

It looks like the precision is relatively high, but the recall and the area under the ROC curve (AUC) aren't as high as you might like. Classifiers often face challenges when trying to maximize both precision and recall, which is especially true when working with imbalanced datasets. It is important to consider the costs of different types of errors in the context of the problem you care about. In this example, a false negative (a fraudulent transaction is missed) may have a financial cost, while a false positive (a transaction is incorrectly flagged as fraudulent) may decrease user happiness.

Class weights

Calculate class weights

The goal is to identify fraudulent transactions, but you don't have very many of those positive samples to work with, so you would want to have the classifier heavily weight the few examples that are available. You can do this by passing Keras weights for each class through a parameter. These will cause the model to "pay more attention" to examples from an under-represented class.

In []:

```
# Scaling by total/2 helps keep the loss to a similar magnitude.
# The sum of the weights of all examples stays the same.
weight_for_0 = (1 / neg) * (total / 2.0)
weight_for_1 = (1 / pos) * (total / 2.0)

class_weight = {0: weight_for_0, 1: weight_for_1}

print('Weight for class 0: {:.2f}'.format(weight_for_0))
print('Weight for class 1: {:.2f}'.format(weight_for_1))
```

Train a model with class weights

Now try re-training and evaluating the model with class weights to see how that affects the predictions.

Note: Using `class_weights` changes the range of the loss. This may affect the stability of the training depending on the optimizer. Optimizers whose step size is dependent on the magnitude of the gradient, like `tf.keras.optimizers.SGD`, may fail. The optimizer used here, `tf.keras.optimizers.Adam`, is unaffected by the scaling change. Also note that because of the weighting, the total losses are not comparable between the two models.

In []:

```
weighted_model = make_model()
weighted_model.load_weights(initial_weights)

weighted_history = weighted_model.fit(
    train_features,
    train_labels,
    batch_size=BATCH_SIZE,
    epochs=EPOCHS,
    callbacks=[early_stopping],
    validation_data=(val_features, val_labels),
    # The class weights go here
    class_weight=class_weight)
```

Check training history

In []:

```
plot_metrics(weighted_history)
```

Evaluate metrics

In []:

```
train_predictions_weighted = weighted_model.predict(train_features, batch_size=BATCH_SIZE)
test_predictions_weighted = weighted_model.predict(test_features, batch_size=BATCH_SIZE)
```

In []:

```
weighted_results = weighted_model.evaluate(test_features, test_labels,
                                            batch_size=BATCH_SIZE, verbose=0)
for name, value in zip(weighted_model.metrics_names, weighted_results):
    print(name, ': ', value)
print()

plot_cm(test_labels, test_predictions_weighted)
```

Here you can see that with class weights the accuracy and precision are lower because there are more false positives, but conversely the recall and AUC are higher because the model also found more true positives. Despite having lower accuracy, this model has higher recall (and identifies more fraudulent transactions). Of course, there is a cost to both types of error (you wouldn't want to bug users by flagging too many legitimate transactions as fraudulent, either). Carefully consider the trade-offs between these different types of errors for your application.

Plot the ROC

In []:

```
plot_roc("Train Baseline", train_labels, train_predictions_baseline, color=colors[0])
plot_roc("Test Baseline", test_labels, test_predictions_baseline, color=colors[0], linestyle='--')

plot_roc("Train Weighted", train_labels, train_predictions_weighted, color=colors[1])
plot_roc("Test Weighted", test_labels, test_predictions_weighted, color=colors[1], linestyle='--')

plt.legend(loc='lower right');
```

Plot the AUPRC

In []:

```
plot_prc("Train Baseline", train_labels, train_predictions_baseline, color=colors[0])
plot_prc("Test Baseline", test_labels, test_predictions_baseline, color=colors[0], linestyle='--')

plot_prc("Train Weighted", train_labels, train_predictions_weighted, color=colors[1])
plot_prc("Test Weighted", test_labels, test_predictions_weighted, color=colors[1], linestyle='--')

plt.legend(loc='lower right');
```

Oversampling

Oversample the minority class

A related approach would be to resample the dataset by oversampling the minority class.

In []:

```
pos_features = train_features[train_labels]
neg_features = train_features[~train_labels]

pos_labels = train_labels[train_labels]
neg_labels = train_labels[~train_labels]
```

Using NumPy

You can balance the dataset manually by choosing the right number of random indices from the positive examples:

In []:

```
ids = np.arange(len(pos_features))
choices = np.random.choice(ids, len(neg_features))

res_pos_features = pos_features[choices]
res_pos_labels = pos_labels[choices]

res_pos_features.shape
```

In []:

```
resampled_features = np.concatenate([res_pos_features, neg_features], axis=0)
resampled_labels = np.concatenate([res_pos_labels, neg_labels], axis=0)

order = np.arange(len(resampled_labels))
np.random.shuffle(order)
resampled_features = resampled_features[order]
resampled_labels = resampled_labels[order]

resampled_features.shape
```

Using `tf.data`

If you're using `tf.data` the easiest way to produce balanced examples is to start with a `positive` and a `negative` dataset, and merge them. See [the `tf.data` guide](#) ([.../guide/data.ipynb](#)) for more examples.

In []:

```
BUFFER_SIZE = 100000

def make_ds(features, labels):
    ds = tf.data.Dataset.from_tensor_slices((features, labels))#.cache()
    ds = ds.shuffle(BUFFER_SIZE).repeat()
    return ds

pos_ds = make_ds(pos_features, pos_labels)
neg_ds = make_ds(neg_features, neg_labels)
```

Each dataset provides (feature, label) pairs:

In []:

```
for features, label in pos_ds.take(1):
    print("Features:\n", features.numpy())
    print()
    print("Label: ", label.numpy())
```

Merge the two together using `tf.data.Dataset.sample_from_datasets`:

In []:

```
resampled_ds = tf.data.Dataset.sample_from_datasets([pos_ds, neg_ds], weights=[0.5, 0.5])
resampled_ds = resampled_ds.batch(BATCH_SIZE).prefetch(2)
```

In []:

```
for features, label in resampled_ds.take(1):
    print(label.numpy().mean())
```

To use this dataset, you'll need the number of steps per epoch.

The definition of "epoch" in this case is less clear. Say it's the number of batches required to see each negative example once:

In []:

```
resampled_steps_per_epoch = np.ceil(2.0*neg/BATCH_SIZE)
resampled_steps_per_epoch
```

Train on the oversampled data

Now try training the model with the resampled data set instead of using class weights to see how these methods compare.

Note: Because the data was balanced by replicating the positive examples, the total dataset size is larger, and each epoch runs for more training steps.

In []:

```
resampled_model = make_model()
resampled_model.load_weights(initial_weights)

# Reset the bias to zero, since this dataset is balanced.
output_layer = resampled_model.layers[-1]
output_layer.bias.assign([0])

val_ds = tf.data.Dataset.from_tensor_slices((val_features, val_labels)).cache()
val_ds = val_ds.batch(BATCH_SIZE).prefetch(2)

resampled_history = resampled_model.fit(
    resampled_ds,
    epochs=EPOCHS,
    steps_per_epoch=resampled_steps_per_epoch,
    callbacks=[early_stopping],
    validation_data=val_ds)
```

If the training process were considering the whole dataset on each gradient update, this oversampling would be basically identical to the class weighting.

But when training the model batch-wise, as you did here, the oversampled data provides a smoother gradient signal: Instead of each positive example being shown in one batch with a large weight, they're shown in many different batches each time with a small weight.

This smoother gradient signal makes it easier to train the model.

Check training history

Note that the distributions of metrics will be different here, because the training data has a totally different distribution from the validation and test data.

In []:

```
plot_metrics(resampled_history)
```

Re-train

Because training is easier on the balanced data, the above training procedure may overfit quickly.

So break up the epochs to give the `tf.keras.callbacks.EarlyStopping` finer control over when to stop training.

In []:

```
resampled_model = make_model()
resampled_model.load_weights(initial_weights)

# Reset the bias to zero, since this dataset is balanced.
output_layer = resampled_model.layers[-1]
output_layer.bias.assign([0])

resampled_history = resampled_model.fit(
    resampled_ds,
    # These are not real epochs
    steps_per_epoch=20,
    epochs=10*EPOCHS,
    callbacks=[early_stopping],
    validation_data=(val_ds))
```

Re-check training history

In []:

```
plot_metrics(resampled_history)
```

Evaluate metrics

In []:

```
train_predictions_resampled = resampled_model.predict(train_features, batch_size=BATCH_SIZE)
test_predictions_resampled = resampled_model.predict(test_features, batch_size=BATCH_SIZE)
```

In []:

```
resampled_results = resampled_model.evaluate(test_features, test_labels,
                                              batch_size=BATCH_SIZE, verbose=0)
for name, value in zip(resampled_model.metrics_names, resampled_results):
    print(name, ': ', value)
print()

plot_cm(test_labels, test_predictions_resampled)
```

Plot the ROC

In []:

```
plot_roc("Train Baseline", train_labels, train_predictions_baseline, color=colors[0])
plot_roc("Test Baseline", test_labels, test_predictions_baseline, color=colors[0], linestyle='--')

plot_roc("Train Weighted", train_labels, train_predictions_weighted, color=colors[1])
plot_roc("Test Weighted", test_labels, test_predictions_weighted, color=colors[1], linestyle='--')

plot_roc("Train Resampled", train_labels, train_predictions_resampled, color=colors[2])
plot_roc("Test Resampled", test_labels, test_predictions_resampled, color=colors[2], linestyle='--')
plt.legend(loc='lower right');
```

Plot the AUPRC

In []:

```
plot_prc("Train Baseline", train_labels, train_predictions_baseline, color=colors[0])
plot_prc("Test Baseline", test_labels, test_predictions_baseline, color=colors[0], linestyle='--')

plot_prc("Train Weighted", train_labels, train_predictions_weighted, color=colors[1])
plot_prc("Test Weighted", test_labels, test_predictions_weighted, color=colors[1], linestyle='--')

plot_prc("Train Resampled", train_labels, train_predictions_resampled, color=colors[2])
plot_prc("Test Resampled", test_labels, test_predictions_resampled, color=colors[2], linestyle='--')
plt.legend(loc='lower right');
```

Applying this tutorial to your problem

Imbalanced data classification is an inherently difficult task since there are so few samples to learn from. You should always start with the data first and do your best to collect as many samples as possible and give substantial thought to what features may be relevant so the model can get the most out of your minority class. At some point your model may struggle to improve and yield the results you want, so it is important to keep in mind the context of your problem and the trade offs between different types of errors.

Copyright 2019 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Classify structured data with feature columns



[View on TensorFlow.org](https://www.tensorflow.org)



Ru

(https://www.tensorflow.org/tutorials/structured_data/feature_columns) (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/structured_data/featu

Warning: The `tf.feature_columns` module described in this tutorial is not recommended for new code. [Keras preprocessing layers](https://www.tensorflow.org/tutorials/structured_data/preprocessing_layers) (https://www.tensorflow.org/tutorials/structured_data/preprocessing_layers) cover this functionality, for migration instructions see the [Migrating feature columns](#) ([https://www.tensorflow.org/tutorials/structured_data/preprocessing_layers](#)) guide. The `tf.feature_columns` module was designed for use with TF1 Estimators. It does fall under our [compatibility guarantees](#) ([https://www.tensorflow.org/guide/versions](#)), but will receive no fixes other than security vulnerabilities.

This tutorial demonstrates how to classify structured data (e.g. tabular data in a CSV). We will use [Keras](#) ([https://www.tensorflow.org/guide/keras](#)) to define the model, and `tf.feature_column` as a bridge to map from columns in a CSV to features used to train the model. This tutorial contains complete code to:

- Load a CSV file using [Pandas](#) ([https://pandas.pydata.org/](#)).
- Build an input pipeline to batch and shuffle the rows using [tf.data](#) ([https://www.tensorflow.org/guide/datasets](#)).
- Map from columns in the CSV to features used to train the model using feature columns.
- Build, train, and evaluate a model using Keras.

The Dataset

We will use a simplified version of the PetFinder dataset ([https://www.kaggle.com/c/petfinder-adoption-prediction](#)). There are several thousand rows in the CSV. Each row describes a pet, and each column describes an attribute. We will use this information to predict the speed at which the pet will be adopted.

Following is a description of this dataset. Notice there are both numeric and categorical columns. There is a free text column which we will not use in this tutorial.

Column	Description	Feature Type	Data Type
Type	Type of animal (Dog, Cat)	Categorical	string
Age	Age of the pet	Numerical	integer
Breed1	Primary breed of the pet	Categorical	string
Color1	Color 1 of pet	Categorical	string
Color2	Color 2 of pet	Categorical	string
MaturitySize	Size at maturity	Categorical	string
FurLength	Fur length	Categorical	string
Vaccinated	Pet has been vaccinated	Categorical	string
Sterilized	Pet has been sterilized	Categorical	string
Health	Health Condition	Categorical	string
Fee	Adoption Fee	Numerical	integer
Description	Profile write-up for this pet	Text	string
PhotoAmt	Total uploaded photos for this pet	Numerical	integer
AdoptionSpeed	Speed of adoption	Classification	integer

Import TensorFlow and other libraries

In []:

```
!pip install sklearn
```

In []:

```
import numpy as np
import pandas as pd

import tensorflow as tf

from tensorflow import feature_column
from tensorflow.keras import layers
from sklearn.model_selection import train_test_split
```

Use Pandas to create a dataframe

[Pandas](#) ([https://pandas.pydata.org/](#)) is a Python library with many helpful utilities for loading and working with structured data. We will use Pandas to download the dataset from a URL, and load it into a dataframe.

```
In [ ]:
```

```
import pathlib

dataset_url = 'http://storage.googleapis.com/download.tensorflow.org/data/petfinder-mini.zip'
csv_file = 'datasets/petfinder-mini/petfinder-mini.csv'

tf.keras.utils.get_file('petfinder_mini.zip', dataset_url,
                        extract=True, cache_dir='.')
dataframe = pd.read_csv(csv_file)
```

```
In [ ]:
```

```
dataframe.head()
```

Create target variable

The task in the original dataset is to predict the speed at which a pet will be adopted (e.g., in the first week, the first month, the first three months, and so on). Let's simplify this for our tutorial. Here, we will transform this into a binary classification problem, and simply predict whether the pet was adopted, or not.

After modifying the label column, 0 will indicate the pet was not adopted, and 1 will indicate it was.

```
In [ ]:
```

```
# In the original dataset "4" indicates the pet was not adopted.
dataframe['target'] = np.where(dataframe['AdoptionSpeed']==4, 0, 1)

# Drop un-used columns.
dataframe = dataframe.drop(columns=['AdoptionSpeed', 'Description'])
```

Split the dataframe into train, validation, and test

The dataset we downloaded was a single CSV file. We will split this into train, validation, and test sets.

```
In [ ]:
```

```
train, test = train_test_split(dataframe, test_size=0.2)
train, val = train_test_split(train, test_size=0.2)
print(len(train), 'train examples')
print(len(val), 'validation examples')
print(len(test), 'test examples')
```

Create an input pipeline using tf.data

Next, we will wrap the dataframes with [tf.data \(<https://www.tensorflow.org/guide/datasets>\)](https://www.tensorflow.org/guide/datasets). This will enable us to use feature columns as a bridge to map from the columns in the Pandas dataframe to features used to train the model. If we were working with a very large CSV file (so large that it does not fit into memory), we would use tf.data to read it from disk directly. That is not covered in this tutorial.

```
In [ ]:
```

```
# A utility method to create a tf.data dataset from a Pandas Dataframe
def df_to_dataset(dataframe, shuffle=True, batch_size=32):
    dataframe = dataframe.copy()
    labels = dataframe.pop('target')
    ds = tf.data.Dataset.from_tensor_slices((dict(dataframe), labels))
    if shuffle:
        ds = ds.shuffle(buffer_size=len(dataframe))
    ds = ds.batch(batch_size)
    return ds
```

```
In [ ]:
```

```
batch_size = 5 # A small batch sized is used for demonstration purposes
train_ds = df_to_dataset(train, batch_size=batch_size)
val_ds = df_to_dataset(val, shuffle=False, batch_size=batch_size)
test_ds = df_to_dataset(test, shuffle=False, batch_size=batch_size)
```

Understand the input pipeline

Now that we have created the input pipeline, let's call it to see the format of the data it returns. We have used a small batch size to keep the output readable.

```
In [ ]:
```

```
for feature_batch, label_batch in train_ds.take(1):
    print('Every feature:', list(feature_batch.keys()))
    print('A batch of ages:', feature_batch['Age'])
    print('A batch of targets:', label_batch)
```

We can see that the dataset returns a dictionary of column names (from the dataframe) that map to column values from rows in the dataframe.

Demonstrate several types of feature columns

TensorFlow provides many types of feature columns. In this section, we will create several types of feature columns, and demonstrate how they transform a column from the dataframe.

```
In [ ]:
```

```
# We will use this batch to demonstrate several types of feature columns
example_batch = next(iter(train_ds))[0]
```

```
In [ ]:
```

```
# A utility method to create a feature column
# and to transform a batch of data
def demo(feature_column):
    feature_layer = layers.DenseFeatures(feature_column)
    print(feature_layer(example_batch).numpy())
```

Numeric columns

The output of a feature column becomes the input to the model (using the demo function defined above, we will be able to see exactly how each column from the dataframe is transformed). A [numeric column](https://www.tensorflow.org/api_docs/python/tf/feature_column/numeric_column) (https://www.tensorflow.org/api_docs/python/tf/feature_column/numeric_column) is the simplest type of column. It is used to represent real valued features. When using this column, your model will receive the column value from the dataframe unchanged.

```
In [ ]:
```

```
photo_count = feature_column.numeric_column('PhotoAmt')
demo(photo_count)
```

In the PetFinder dataset, most columns from the dataframe are categorical.

Bucketized columns

Often, you don't want to feed a number directly into the model, but instead split its value into different categories based on numerical ranges. Consider raw data that represents a person's age. Instead of representing age as a numeric column, we could split the age into several buckets using a [bucketized column](https://www.tensorflow.org/api_docs/python/tf/feature_column/bucketized_column) (https://www.tensorflow.org/api_docs/python/tf/feature_column/bucketized_column). Notice the one-hot values below describe which age range each row matches.

```
In [ ]:
```

```
age = feature_column.numeric_column('Age')
age_buckets = feature_column.bucketized_column(age, boundaries=[1, 3, 5])
demo(age_buckets)
```

Categorical columns

In this dataset, Type is represented as a string (e.g. 'Dog', or 'Cat'). We cannot feed strings directly to a model. Instead, we must first map them to numeric values. The categorical vocabulary columns provide a way to represent strings as a one-hot vector (much like you have seen above with age buckets). The vocabulary can be passed as a list using [categorical_column_with_vocabulary_list](https://www.tensorflow.org/api_docs/python/tf/feature_column/categorical_column_with_vocabulary_list) (https://www.tensorflow.org/api_docs/python/tf/feature_column/categorical_column_with_vocabulary_list), or loaded from a file using [categorical_column_with_vocabulary_file](https://www.tensorflow.org/api_docs/python/tf/feature_column/categorical_column_with_vocabulary_file) (https://www.tensorflow.org/api_docs/python/tf/feature_column/categorical_column_with_vocabulary_file).

```
In [ ]:
```

```
animal_type = feature_column.categorical_column_with_vocabulary_list(
    'Type', ['Cat', 'Dog'])

animal_type_one_hot = feature_column.indicator_column(animal_type)
demo(animal_type_one_hot)
```

Embedding columns

Suppose instead of having just a few possible strings, we have thousands (or more) values per category. For a number of reasons, as the number of categories grow large, it becomes infeasible to train a neural network using one-hot encodings. We can use an embedding column to overcome this limitation. Instead of representing the data as a one-hot vector of many dimensions, an [embedding column](#) (https://www.tensorflow.org/api_docs/python/tf/feature_column/embedding_column) represents that data as a lower-dimensional, dense vector in which each cell can contain any number, not just 0 or 1. The size of the embedding (8, in the example below) is a parameter that must be tuned.

Key point: using an embedding column is best when a categorical column has many possible values. We are using one here for demonstration purposes, so you have a complete example you can modify for a different dataset in the future.

In []:

```
# Notice the input to the embedding column is the categorical column
# we previously created
breed1 = feature_column.categorical_column_with_vocabulary_list(
    'Breed1', dataframe.Breed1.unique())
breed1_embedding = feature_column.embedding_column(breed1, dimension=8)
demo(breed1_embedding)
```

Hashed feature columns

Another way to represent a categorical column with a large number of values is to use a [categorical_column_with_hash_bucket](#) (https://www.tensorflow.org/api_docs/python/tf/feature_column/categorical_column_with_hash_bucket). This feature column calculates a hash value of the input, then selects one of the `hash_bucket_size` buckets to encode a string. When using this column, you do not need to provide the vocabulary, and you can choose to make the number of `hash_buckets` significantly smaller than the number of actual categories to save space.

Key point: An important downside of this technique is that there may be collisions in which different strings are mapped to the same bucket. In practice, this can work well for some datasets regardless.

In []:

```
breed1_hashed = feature_column.categorical_column_with_hash_bucket(
    'Breed1', hash_bucket_size=10)
demo(feature_column.indicator_column(breed1_hashed))
```

Crossed feature columns

Combining features into a single feature, better known as [feature crosses](#) (https://developers.google.com/machine-learning/glossary/#feature_cross), enables a model to learn separate weights for each combination of features. Here, we will create a new feature that is the cross of Age and Type. Note that `crossed_column` does not build the full table of all possible combinations (which could be very large). Instead, it is backed by a `hashed_column`, so you can choose how large the table is.

In []:

```
crossed_feature = feature_column.crossed_column([age_buckets, animal_type], hash_bucket_size=10)
demo(feature_column.indicator_column(crossed_feature))
```

Choose which columns to use

We have seen how to use several types of feature columns. Now we will use them to train a model. The goal of this tutorial is to show you the complete code (e.g. mechanics) needed to work with feature columns. We have selected a few columns to train our model below arbitrarily.

Key point: If your aim is to build an accurate model, try a larger dataset of your own, and think carefully about which features are the most meaningful to include, and how they should be represented.

In []:

```
feature_columns = []

# numeric cols
for header in ['PhotoAmt', 'Fee', 'Age']:
    feature_columns.append(feature_column.numeric_column(header))
```

In []:

```
# bucketized cols
age = feature_column.numeric_column('Age')
age_buckets = feature_column.bucketized_column(age, boundaries=[1, 2, 3, 4, 5])
feature_columns.append(age_buckets)
```

In []:

```
# indicator columns
indicator_column_names = ['Type', 'Color1', 'Color2', 'Gender', 'MaturitySize',
                           'FurLength', 'Vaccinated', 'Sterilized', 'Health']
for col_name in indicator_column_names:
    categorical_column = feature_column.categorical_column_with_vocabulary_list(
        col_name, dataframe[col_name].unique())
    indicator_column = feature_column.indicator_column(categorical_column)
    feature_columns.append(indicator_column)
```

In []:

```
# embedding columns
breed1 = feature_column.categorical_column_with_vocabulary_list(
    'Breed1', dataframe.Breed1.unique())
breed1_embedding = feature_column.embedding_column(breed1, dimension=8)
feature_columns.append(breed1_embedding)
```

In []:

```
# crossed columns
age_type_feature = feature_column.crossed_column([age_buckets, animal_type], hash_bucket_size=100)
feature_columns.append(feature_column.indicator_column(age_type_feature))
```

Create a feature layer

Now that we have defined our feature columns, we will use a [DenseFeatures](#)

(https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/DenseFeatures) layer to input them to our Keras model.

In []:

```
feature_layer = tf.keras.layers.DenseFeatures(feature_columns)
```

Earlier, we used a small batch size to demonstrate how feature columns worked. We create a new input pipeline with a larger batch size.

In []:

```
batch_size = 32
train_ds = df_to_dataset(train, batch_size=batch_size)
val_ds = df_to_dataset(val, shuffle=False, batch_size=batch_size)
test_ds = df_to_dataset(test, shuffle=False, batch_size=batch_size)
```

Create, compile, and train the model

In []:

```
model = tf.keras.Sequential([
    feature_layer,
    layers.Dense(128, activation='relu'),
    layers.Dense(128, activation='relu'),
    layers.Dropout(.1),
    layers.Dense(1)
])

model.compile(optimizer='adam',
              loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              metrics=[ 'accuracy'])

model.fit(train_ds,
          validation_data=val_ds,
          epochs=10)
```

In []:

```
loss, accuracy = model.evaluate(test_ds)
print("Accuracy", accuracy)
```

Key point: You will typically see best results with deep learning with much larger and more complex datasets. When working with a small dataset like this one, we recommend using a decision tree or random forest as a strong baseline. The goal of this tutorial is not to train an accurate model, but to demonstrate the mechanics of working with structured data, so you have code to use as a starting point when working with your own datasets in the future.

Next steps

The best way to learn more about classifying structured data is to try it yourself. We suggest finding another dataset to work with, and training a model to classify it using code similar to the above. To improve accuracy, think carefully about which features to include in your model, and how they should be represented.

Copyright 2019 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Time series forecasting



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/structured_data/time_series)

(https://www.tensorflow.org/tutorials/structured_data/time_series)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/structured_data/time_series.ipynb)

(https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/structured_data/time_series.ipynb)

This tutorial is an introduction to time series forecasting using TensorFlow. It builds a few different styles of models including Convolutional and Recurrent Neural Networks (CNNs and RNNs).

This is covered in two main parts, with subsections:

- Forecast for a single time step:
 - A single feature.
 - All features.
- Forecast multiple steps:
 - Single-shot: Make the predictions all at once.
 - Autoregressive: Make one prediction at a time and feed the output back to the model.

Setup

In []:

```
import os
import datetime

import IPython
import IPython.display
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import tensorflow as tf

mpl.rcParams['figure.figsize'] = (8, 6)
mpl.rcParams['axes.grid'] = False
```

The weather dataset

This tutorial uses a [weather time series dataset \(<https://www.bgc-jena.mpg.de/wetter/>\)](https://www.bgc-jena.mpg.de/wetter/) recorded by the [Max Planck Institute for Biogeochemistry \(<https://www.bgc-jena.mpg.de/>\)](https://www.bgc-jena.mpg.de/).

This dataset contains 14 different features such as air temperature, atmospheric pressure, and humidity. These were collected every 10 minutes, beginning in 2003. For efficiency, you will use only the data collected between 2009 and 2016. This section of the dataset was prepared by François Chollet for his book [Deep Learning with Python \(<https://www.manning.com/books/deep-learning-with-python>\)](https://www.manning.com/books/deep-learning-with-python).

```
In [ ]:
```

```
zip_path = tf.keras.utils.get_file(  
    origin='https://storage.googleapis.com/tensorflow/tf-keras-datasets/jena_climate_2009_2016.csv.zip',  
    fname='jena_climate_2009_2016.csv',  
    extract=True)  
csv_path, _ = os.path.splitext(zip_path)
```

This tutorial will just deal with **hourly predictions**, so start by sub-sampling the data from 10-minute intervals to one-hour intervals:

```
In [ ]:
```

```
df = pd.read_csv(csv_path)  
# Slice [start:stop:step], starting from index 5 take every 6th record.  
df = df[5::6]  
  
date_time = pd.to_datetime(df.pop('Date Time'), format='%d.%m.%Y %H:%M:%S')
```

Let's take a glance at the data. Here are the first few rows:

```
In [ ]:
```

```
df.head()
```

Here is the evolution of a few features over time:

```
In [ ]:
```

```
plot_cols = ['T (degC)', 'p (mbar)', 'rho (g/m**3)']  
plot_features = df[plot_cols]  
plot_features.index = date_time  
_ = plot_features.plot(subplots=True)  
  
plot_features = df[plot_cols][:480]  
plot_features.index = date_time[:480]  
_ = plot_features.plot(subplots=True)
```

Inspect and cleanup

Next, look at the statistics of the dataset:

```
In [ ]:
```

```
df.describe().transpose()
```

Wind velocity

One thing that should stand out is the `min` value of the wind velocity (`wv (m/s)`) and the maximum value (`max. wv (m/s)`) columns. This `-9999` is likely erroneous.

There's a separate wind direction column, so the velocity should be greater than zero (`>=0`). Replace it with zeros:

```
In [ ]:
```

```
wv = df['wv (m/s)']  
bad_wv = wv == -9999.0  
wv[bad_wv] = 0.0  
  
max_wv = df['max. wv (m/s)']  
bad_max_wv = max_wv == -9999.0  
max_wv[bad_max_wv] = 0.0  
  
# The above inplace edits are reflected in the DataFrame.  
df['wv (m/s)'].min()
```

Feature engineering

Before diving in to build a model, it's important to understand your data and be sure that you're passing the model appropriately formatted data.

Wind

The last column of the data, `wd (deg)` —gives the wind direction in units of degrees. Angles do not make good model inputs: 360° and 0° should be close to each other and wrap around smoothly. Direction shouldn't matter if the wind is not blowing.

Right now the distribution of wind data looks like this:

In []:

```
plt.hist2d(df['wd (deg)'], df['wv (m/s)'], bins=(50, 50), vmax=400)
plt.colorbar()
plt.xlabel('Wind Direction [deg]')
plt.ylabel('Wind Velocity [m/s]')
```

But this will be easier for the model to interpret if you convert the wind direction and velocity columns to a wind **vector**:

In []:

```
wv = df.pop('wv (m/s)')
max_wv = df.pop('max. wv (m/s)')

# Convert to radians.
wd_rad = df.pop('wd (deg)')*np.pi / 180

# Calculate the wind x and y components.
df['Wx'] = wv*np.cos(wd_rad)
df['Wy'] = wv*np.sin(wd_rad)

# Calculate the max wind x and y components.
df['max Wx'] = max_wv*np.cos(wd_rad)
df['max Wy'] = max_wv*np.sin(wd_rad)
```

The distribution of wind vectors is much simpler for the model to correctly interpret:

In []:

```
plt.hist2d(df['Wx'], df['Wy'], bins=(50, 50), vmax=400)
plt.colorbar()
plt.xlabel('Wind X [m/s]')
plt.ylabel('Wind Y [m/s]')
ax = plt.gca()
ax.axis('tight')
```

Time

Similarly, the `Date Time` column is very useful, but not in this string form. Start by converting it to seconds:

In []:

```
timestamp_s = date_time.map(pd.Timestamp.timestamp)
```

Similar to the wind direction, the time in seconds is not a useful model input. Being weather data, it has clear daily and yearly periodicity. There are many ways you could deal with periodicity.

You can get usable signals by using sine and cosine transforms to clear "Time of day" and "Time of year" signals:

In []:

```
day = 24*60*60
year = (365.2425)*day

df['Day sin'] = np.sin(timestamp_s * (2 * np.pi / day))
df['Day cos'] = np.cos(timestamp_s * (2 * np.pi / day))
df['Year sin'] = np.sin(timestamp_s * (2 * np.pi / year))
df['Year cos'] = np.cos(timestamp_s * (2 * np.pi / year))
```

In []:

```
plt.plot(np.array(df['Day sin'])[:25])
plt.plot(np.array(df['Day cos'])[:25])
plt.xlabel('Time [h]')
plt.title('Time of day signal')
```

This gives the model access to the most important frequency features. In this case you knew ahead of time which frequencies were important.

If you don't have that information, you can determine which frequencies are important by extracting features with [Fast Fourier Transform](#) (https://en.wikipedia.org/wiki/Fast_Fourier_transform). To check the assumptions, here is the `tf.signal.rfft` of the temperature over time. Note the obvious peaks at frequencies near 1/year and 1/day :

In []:

```
fft = tf.signal.rfft(df['T (degC)'])
f_per_dataset = np.arange(0, len(fft))

n_samples_h = len(df['T (degC)'])
hours_per_year = 24*365.2524
years_per_dataset = n_samples_h/(hours_per_year)

f_per_year = f_per_dataset/years_per_dataset
plt.step(f_per_year, np.abs(fft))
plt.xscale('log')
plt.ylim(0, 400000)
plt.xlim([0.1, max(plt.xlim())])
plt.xticks([1, 365.2524], labels=['1/Year', '1/day'])
_ = plt.xlabel('Frequency (log scale)')
```

Split the data

You'll use a (70%, 20%, 10%) split for the training, validation, and test sets. Note the data is **not** being randomly shuffled before splitting. This is for two reasons:

1. It ensures that chopping the data into windows of consecutive samples is still possible.
2. It ensures that the validation/test results are more realistic, being evaluated on the data collected after the model was trained.

In []:

```
column_indices = {name: i for i, name in enumerate(df.columns)}

n = len(df)
train_df = df[0:int(n*0.7)]
val_df = df[int(n*0.7):int(n*0.9)]
test_df = df[int(n*0.9):]

num_features = df.shape[1]
```

Normalize the data

It is important to scale features before training a neural network. Normalization is a common way of doing this scaling: subtract the mean and divide by the standard deviation of each feature.

The mean and standard deviation should only be computed using the training data so that the models have no access to the values in the validation and test sets.

It's also arguable that the model shouldn't have access to future values in the training set when training, and that this normalization should be done using moving averages. That's not the focus of this tutorial, and the validation and test sets ensure that you get (somewhat) honest metrics. So, in the interest of simplicity this tutorial uses a simple average.

In []:

```
train_mean = train_df.mean()
train_std = train_df.std()

train_df = (train_df - train_mean) / train_std
val_df = (val_df - train_mean) / train_std
test_df = (test_df - train_mean) / train_std
```

Now, peek at the distribution of the features. Some features do have long tails, but there are no obvious errors like the -9999 wind velocity value.

In []:

```
df_std = (df - train_mean) / train_std
df_std = df_std.melt(var_name='Column', value_name='Normalized')
plt.figure(figsize=(12, 6))
ax = sns.violinplot(x='Column', y='Normalized', data=df_std)
_ = ax.set_xticklabels(df.keys(), rotation=90)
```

Data windowing

The models in this tutorial will make a set of predictions based on a window of consecutive samples from the data.

The main features of the input windows are:

- The width (number of time steps) of the input and label windows.
- The time offset between them.
- Which features are used as inputs, labels, or both.

This tutorial builds a variety of models (including Linear, DNN, CNN and RNN models), and uses them for both:

- *Single-output*, and *multi-output* predictions.
- *Single-time-step* and *multi-time-step* predictions.

This section focuses on implementing the data windowing so that it can be reused for all of those models.

Depending on the task and type of model you may want to generate a variety of data windows. Here are some examples:

1. For example, to make a single prediction 24 hours into the future, given 24 hours of history, you might define a window like this:



2. A model that makes a prediction one hour into the future, given six hours of history, would need a window like this:



The rest of this section defines a `WindowGenerator` class. This class can:

1. Handle the indexes and offsets as shown in the diagrams above.
2. Split windows of features into (`features`, `labels`) pairs.
3. Plot the content of the resulting windows.
4. Efficiently generate batches of these windows from the training, evaluation, and test data, using `tf.data.Dataset`s.

1. Indexes and offsets

Start by creating the `WindowGenerator` class. The `__init__` method includes all the necessary logic for the input and label indices.

It also takes the training, evaluation, and test `DataFrames` as input. These will be converted to `tf.data.Dataset`s of windows later.

In []:

```
class WindowGenerator():
    def __init__(self, input_width, label_width, shift,
                 train_df=train_df, val_df=val_df, test_df=test_df,
                 label_columns=None):
        # Store the raw data.
        self.train_df = train_df
        self.val_df = val_df
        self.test_df = test_df

        # Work out the label column indices.
        self.label_columns = label_columns
        if label_columns is not None:
            self.label_columns_indices = {name: i for i, name in
                                         enumerate(label_columns)}
        self.column_indices = {name: i for i, name in
                              enumerate(train_df.columns)}

        # Work out the window parameters.
        self.input_width = input_width
        self.label_width = label_width
        self.shift = shift

        self.total_window_size = input_width + shift

        self.input_slice = slice(0, input_width)
        self.input_indices = np.arange(self.total_window_size)[self.input_slice]

        self.label_start = self.total_window_size - self.label_width
        self.labels_slice = slice(self.label_start, None)
        self.label_indices = np.arange(self.total_window_size)[self.labels_slice]

    def __repr__(self):
        return '\n'.join([
            f'Total window size: {self.total_window_size}',
            f'Input indices: {self.input_indices}',
            f'Label indices: {self.label_indices}',
            f'Label column name(s): {self.label_columns}'])
```

Here is code to create the 2 windows shown in the diagrams at the start of this section:

In []:

```
w1 = WindowGenerator(input_width=24, label_width=1, shift=24,
                     label_columns=['T (degC)'])
w1
```

In []:

```
w2 = WindowGenerator(input_width=6, label_width=1, shift=1,
                     label_columns=['T (degC)'])
w2
```

2. Split

Given a list of consecutive inputs, the `split_window` method will convert them to a window of inputs and a window of labels.

The example `w2` you define earlier will be split like this:



This diagram doesn't show the `features` axis of the data, but this `split_window` function also handles the `label_columns` so it can be used for both the single output and multi-output examples.

In []:

```
def split_window(self, features):
    inputs = features[:, self.input_slice, :]
    labels = features[:, self.labels_slice, :]
    if self.label_columns is not None:
        labels = tf.stack(
            [labels[:, :, self.column_indices[name]] for name in self.label_columns],
            axis=-1)

    # Slicing doesn't preserve static shape information, so set the shapes
    # manually. This way the `tf.data.Datasets` are easier to inspect.
    inputs.set_shape([None, self.input_width, None])
    labels.set_shape([None, self.label_width, None])

    return inputs, labels

WindowGenerator.split_window = split_window
```

Try it out:

In []:

```
# Stack three slices, the length of the total window.
example_window = tf.stack([np.array(train_df[:w2.total_window_size]),
                           np.array(train_df[100:100+w2.total_window_size]),
                           np.array(train_df[200:200+w2.total_window_size])])

example_inputs, example_labels = w2.split_window(example_window)

print('All shapes are: (batch, time, features)')
print(f'Window shape: {example_window.shape}')
print(f'Inputs shape: {example_inputs.shape}')
print(f'Labels shape: {example_labels.shape}')
```

Typically, data in TensorFlow is packed into arrays where the outermost index is across examples (the "batch" dimension). The middle indices are the "time" or "space" (width, height) dimension(s). The innermost indices are the features.

The code above took a batch of three 7-time step windows with 19 features at each time step. It splits them into a batch of 6-time step 19-feature inputs, and a 1-time step 1-feature label. The label only has one feature because the `WindowGenerator` was initialized with `label_columns=['T (degC)']`. Initially, this tutorial will build models that predict single output labels.

3. Plot

Here is a plot method that allows a simple visualization of the split window:

In []:

```
w2.example = example_inputs, example_labels
```

In []:

```
def plot(self, model=None, plot_col='T (degC)', max_subplots=3):
    inputs, labels = self.example
    plt.figure(figsize=(12, 8))
    plot_col_index = self.column_indices[plot_col]
    max_n = min(max_subplots, len(inputs))
    for n in range(max_n):
        plt.subplot(max_n, 1, n+1)
        plt.ylabel(f'{plot_col} [normed]')
        plt.plot(self.input_indices, inputs[n, :, plot_col_index],
                 label='Inputs', marker='.', zorder=-10)

    if self.label_columns:
        label_col_index = self.label_columns_indices.get(plot_col, None)
    else:
        label_col_index = plot_col_index

    if label_col_index is None:
        continue

    plt.scatter(self.label_indices, labels[n, :, label_col_index],
                edgecolors='k', label='Labels', c='#2ca02c', s=64)
    if model is not None:
        predictions = model(inputs)
        plt.scatter(self.label_indices, predictions[n, :, label_col_index],
                    marker='X', edgecolors='k', label='Predictions',
                    c='#ff7f0e', s=64)

    if n == 0:
        plt.legend()

plt.xlabel('Time [h]')

WindowGenerator.plot = plot
```

This plot aligns inputs, labels, and (later) predictions based on the time that the item refers to:

In []:

```
w2.plot()
```

You can plot the other columns, but the example window `w2` configuration only has labels for the `T (degC)` column.

In []:

```
w2.plot(plot_col='p (mbar)')
```

4. Create `tf.data.Datasets`

Finally, this `make_dataset` method will take a time series DataFrame and convert it to a `tf.data.Dataset` of `(input_window, label_window)` pairs using the `tf.keras.utils.timeseries_dataset_from_array` function:

In []:

```
def make_dataset(self, data):
    data = np.array(data, dtype=np.float32)
    ds = tf.keras.utils.timeseries_dataset_from_array(
        data=data,
        targets=None,
        sequence_length=self.total_window_size,
        sequence_stride=1,
        shuffle=True,
        batch_size=32,)

    ds = ds.map(self.split_window)

    return ds

WindowGenerator.make_dataset = make_dataset
```

The `WindowGenerator` object holds training, validation, and test data.

Add properties for accessing them as `tf.data.Dataset`s using the `make_dataset` method you defined earlier. Also, add a standard example batch for easy access and plotting:

In []:

```
@property
def train(self):
    return self.make_dataset(self.train_df)

@property
def val(self):
    return self.make_dataset(self.val_df)

@property
def test(self):
    return self.make_dataset(self.test_df)

@property
def example(self):
    """Get and cache an example batch of `inputs, labels` for plotting."""
    result = getattr(self, '_example', None)
    if result is None:
        # No example batch was found, so get one from the `.train` dataset
        result = next(iter(self.train))
        # And cache it for next time
        self._example = result
    return result

WindowGenerator.train = train
WindowGenerator.val = val
WindowGenerator.test = test
WindowGenerator.example = example
```

Now, the `WindowGenerator` object gives you access to the `tf.data.Dataset` objects, so you can easily iterate over the data.

The `Dataset.element_spec` property tells you the structure, data types, and shapes of the dataset elements.

In []:

```
# Each element is an (inputs, label) pair.
w2.train.element_spec
```

Iterating over a `Dataset` yields concrete batches:

In []:

```
for example_inputs, example_labels in w2.train.take(1):
    print(f'Inputs shape (batch, time, features): {example_inputs.shape}')
    print(f'Labels shape (batch, time, features): {example_labels.shape}')
```

Single step models

The simplest model you can build on this sort of data is one that predicts a single feature's value—1 time step (one hour) into the future based only on the current conditions.

So, start by building models to predict the `T (degC)` value one hour into the future.



Configure a `WindowGenerator` object to produce these single-step (`input, label`) pairs:

In []:

```
single_step_window = WindowGenerator(
    input_width=1, label_width=1, shift=1,
    label_columns=['T (degC)'])
single_step_window
```

The `window` object creates `tf.data.Dataset`s from the training, validation, and test sets, allowing you to easily iterate over batches of data.

In []:

```
for example_inputs, example_labels in single_step_window.train.take(1):
    print(f'Inputs shape (batch, time, features): {example_inputs.shape}')
    print(f'Labels shape (batch, time, features): {example_labels.shape}')
```

Baseline

Before building a trainable model it would be good to have a performance baseline as a point for comparison with the later more complicated models.

This first task is to predict temperature one hour into the future, given the current value of all features. The current values include the current temperature.

So, start with a model that just returns the current temperature as the prediction, predicting "No change". This is a reasonable baseline since temperature changes slowly. Of course, this baseline will work less well if you make a prediction further in the future.

[2]

In []:

```
class Baseline(tf.keras.Model):
    def __init__(self, label_index=None):
        super().__init__()
        self.label_index = label_index

    def call(self, inputs):
        if self.label_index is None:
            return inputs
        result = inputs[:, :, self.label_index]
        return result[:, :, tf.newaxis]
```

Instantiate and evaluate this model:

In []:

```
baseline = Baseline(label_index=column_indices['T (degC)'])

baseline.compile(loss=tf.losses.MeanSquaredError(),
                  metrics=[tf.metrics.MeanAbsoluteError()])

val_performance = {}
performance = {}
val_performance['Baseline'] = baseline.evaluate(single_step_window.val)
performance['Baseline'] = baseline.evaluate(single_step_window.test, verbose=0)
```

That printed some performance metrics, but those don't give you a feeling for how well the model is doing.

The `WindowGenerator` has a plot method, but the plots won't be very interesting with only a single sample.

So, create a wider `WindowGenerator` that generates windows 24 hours of consecutive inputs and labels at a time. The new `wide_window` variable doesn't change the way the model operates. The model still makes predictions one hour into the future based on a single input time step. Here, the `time` axis acts like the `batch` axis: each prediction is made independently with no interaction between time steps:

In []:

```
wide_window = WindowGenerator(
    input_width=24, label_width=24, shift=1,
    label_columns=['T (degC)'])

wide_window
```

This expanded window can be passed directly to the same `baseline` model without any code changes. This is possible because the inputs and labels have the same number of time steps, and the baseline just forwards the input to the output:

[2]

In []:

```
print('Input shape:', wide_window.example[0].shape)
print('Output shape:', baseline(wide_window.example[0]).shape)
```

By plotting the baseline model's predictions, notice that it is simply the labels shifted right by one hour:

In []:

```
wide_window.plot(baseline)
```

In the above plots of three examples the single step model is run over the course of 24 hours. This deserves some explanation:

- The blue `Inputs` line shows the input temperature at each time step. The model receives all features, this plot only shows the temperature.
- The green `Labels` dots show the target prediction value. These dots are shown at the prediction time, not the input time. That is why the range of labels is shifted 1 step relative to the inputs.
- The orange `Predictions` crosses are the model's prediction's for each output time step. If the model were predicting perfectly the predictions would land directly on the `Labels`.

Linear model

The simplest **trainable** model you can apply to this task is to insert linear transformation between the input and output. In this case the output from a time step only depends on that step:



A `tf.keras.layers.Dense` layer with no `activation` set is a linear model. The layer only transforms the last axis of the data from `(batch, time, inputs)` to `(batch, time, units)`; it is applied independently to every item across the batch and time axes.

In []:

```
linear = tf.keras.Sequential([
    tf.keras.layers.Dense(units=1)
])
```

In []:

```
print('Input shape:', single_step_window.example[0].shape)
print('Output shape:', linear(single_step_window.example[0]).shape)
```

This tutorial trains many models, so package the training procedure into a function:

In []:

```
MAX_EPOCHS = 20

def compile_and_fit(model, window, patience=2):
    early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss',
                                                       patience=patience,
                                                       mode='min')

    model.compile(loss=tf.losses.MeanSquaredError(),
                  optimizer=tf.optimizers.Adam(),
                  metrics=[tf.metrics.MeanAbsoluteError()])

    history = model.fit(window.train, epochs=MAX_EPOCHS,
                        validation_data=window.val,
                        callbacks=[early_stopping])
    return history
```

Train the model and evaluate its performance:

In []:

```
history = compile_and_fit(linear, single_step_window)

val_performance['Linear'] = linear.evaluate(single_step_window.val)
performance['Linear'] = linear.evaluate(single_step_window.test, verbose=0)
```

Like the `baseline` model, the linear model can be called on batches of wide windows. Used this way the model makes a set of independent predictions on consecutive time steps. The `time` axis acts like another `batch` axis. There are no interactions between the predictions at each time step.



In []:

```
print('Input shape:', wide_window.example[0].shape)
print('Output shape:', baseline(wide_window.example[0]).shape)
```

Here is the plot of its example predictions on the `wide_window`, note how in many cases the prediction is clearly better than just returning the input temperature, but in a few cases it's worse:

In []:

```
wide_window.plot(linear)
```

One advantage to linear models is that they're relatively simple to interpret. You can pull out the layer's weights and visualize the weight assigned to each input:

In []:

```
plt.bar(x = range(len(train_df.columns)),
         height=linear.layers[0].kernel[:,0].numpy())
axis = plt.gca()
axis.set_xticks(range(len(train_df.columns)))
_ = axis.set_xticklabels(train_df.columns, rotation=90)
```

Sometimes the model doesn't even place the most weight on the input `T (degC)`. This is one of the risks of random initialization.

Dense

Before applying models that actually operate on multiple time-steps, it's worth checking the performance of deeper, more powerful, single input step models.

Here's a model similar to the `linear` model, except it stacks several a few `Dense` layers between the input and the output:

In []:

```
dense = tf.keras.Sequential([
    tf.keras.layers.Dense(units=64, activation='relu'),
    tf.keras.layers.Dense(units=64, activation='relu'),
    tf.keras.layers.Dense(units=1)
])

history = compile_and_fit(dense, single_step_window)

val_performance['Dense'] = dense.evaluate(single_step_window.val)
performance['Dense'] = dense.evaluate(single_step_window.test, verbose=0)
```

Multi-step dense

A single-time-step model has no context for the current values of its inputs. It can't see how the input features are changing over time. To address this issue the model needs access to multiple time steps when making predictions:



The `baseline`, `linear` and `dense` models handled each time step independently. Here the model will take multiple time steps as input to produce a single output.

Create a `WindowGenerator` that will produce batches of three-hour inputs and one-hour labels:

Note that the `Window`'s `shift` parameter is relative to the end of the two windows.

In []:

```
CONV_WIDTH = 3
conv_window = WindowGenerator(
    input_width=CONV_WIDTH,
    label_width=1,
    shift=1,
    label_columns=['T (degC)'])

conv_window
```

In []:

```
conv_window.plot()
plt.title("Given 3 hours of inputs, predict 1 hour into the future.")
```

You could train a `dense` model on a multiple-input-step window by adding a `tf.keras.layers.Flatten` as the first layer of the model:

In []:

```
multi_step_dense = tf.keras.Sequential([
    # Shape: (time, features) => (time*features)
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=32, activation='relu'),
    tf.keras.layers.Dense(units=32, activation='relu'),
    tf.keras.layers.Dense(units=1),
    # Add back the time dimension.
    # Shape: (outputs) => (1, outputs)
    tf.keras.layers.Reshape([1, -1]),
])
```

In []:

```
print('Input shape:', conv_window.example[0].shape)
print('Output shape:', multi_step_dense(conv_window.example[0]).shape)
```

In []:

```
history = compile_and_fit(multi_step_dense, conv_window)

IPython.display.clear_output()
val_performance['Multi step dense'] = multi_step_dense.evaluate(conv_window.val)
performance['Multi step dense'] = multi_step_dense.evaluate(conv_window.test, verbose=0)
```

In []:

```
conv_window.plot(multi_step_dense)
```

The main down-side of this approach is that the resulting model can only be executed on input windows of exactly this shape.

In []:

```
print('Input shape:', wide_window.example[0].shape)
try:
    print('Output shape:', multi_step_dense(wide_window.example[0]).shape)
except Exception as e:
    print(f'\n{type(e).__name__}:{e}' )
```

The convolutional models in the next section fix this problem.

Convolution neural network

A convolution layer (`tf.keras.layers.Conv1D`) also takes multiple time steps as input to each prediction.

Below is the **same** model as `multi_step_dense`, re-written with a convolution.

Note the changes:

- The `tf.keras.layers.Flatten` and the first `tf.keras.layers.Dense` are replaced by a `tf.keras.layers.Conv1D`.
- The `tf.keras.layers.Reshape` is no longer necessary since the convolution keeps the time axis in its output.

In []:

```
conv_model = tf.keras.Sequential([
    tf.keras.layers.Conv1D(filters=32,
                          kernel_size=(CONV_WIDTH, ),
                          activation='relu'),
    tf.keras.layers.Dense(units=32, activation='relu'),
    tf.keras.layers.Dense(units=1),
])
```

Run it on an example batch to check that the model produces outputs with the expected shape:

In []:

```
print("Conv model on `conv_window`")
print('Input shape:', conv_window.example[0].shape)
print('Output shape:', conv_model(conv_window.example[0]).shape)
```

Train and evaluate it on the `conv_window` and it should give performance similar to the `multi_step_dense` model.

In []:

```
history = compile_and_fit(conv_model, conv_window)

IPython.display.clear_output()
val_performance['Conv'] = conv_model.evaluate(conv_window.val)
performance['Conv'] = conv_model.evaluate(conv_window.test, verbose=0)
```

The difference between this `conv_model` and the `multi_step_dense` model is that the `conv_model` can be run on inputs of any length. The convolutional layer is applied to a sliding window of inputs:



If you run it on wider input, it produces wider output:

In []:

```
print("Wide window")
print('Input shape:', wide_window.example[0].shape)
print('Labels shape:', wide_window.example[1].shape)
print('Output shape:', conv_model(wide_window.example[0]).shape)
```

Note that the output is shorter than the input. To make training or plotting work, you need the labels, and prediction to have the same length. So build a `WindowGenerator` to produce wide windows with a few extra input time steps so the label and prediction lengths match:

In []:

```
LABEL_WIDTH = 24
INPUT_WIDTH = LABEL_WIDTH + (CONV_WIDTH - 1)
wide_conv_window = WindowGenerator(
    input_width=INPUT_WIDTH,
    label_width=LABEL_WIDTH,
    shift=1,
    label_columns=['T (degC)'])

wide_conv_window
```

In []:

```
print("Wide conv window")
print('Input shape:', wide_conv_window.example[0].shape)
print('Labels shape:', wide_conv_window.example[1].shape)
print('Output shape:', conv_model(wide_conv_window.example[0]).shape)
```

Now, you can plot the model's predictions on a wider window. Note the 3 input time steps before the first prediction. Every prediction here is based on the 3 preceding time steps:

In []:

```
wide_conv_window.plot(conv_model)
```

Recurrent neural network

A Recurrent Neural Network (RNN) is a type of neural network well-suited to time series data. RNNs process a time series step-by-step, maintaining an internal state from time-step to time-step.

You can learn more in the [Text generation with an RNN](https://www.tensorflow.org/text/tutorials/text_generation) (https://www.tensorflow.org/text/tutorials/text_generation) tutorial and the [Recurrent Neural Networks \(RNN\) with Keras](https://www.tensorflow.org/guide/keras/rnn) (<https://www.tensorflow.org/guide/keras/rnn>) guide.

In this tutorial, you will use an RNN layer called Long Short-Term Memory (`tf.keras.layers.LSTM`).

An important constructor argument for all Keras RNN layers, such as `tf.keras.layers.LSTM`, is the `return_sequences` argument. This setting can configure the layer in one of two ways:

1. If `False`, the default, the layer only returns the output of the final time step, giving the model time to warm up its internal state before making a single prediction:



1. If `True`, the layer returns an output for each input. This is useful for:

- Stacking RNN layers.
- Training a model on multiple time steps simultaneously.



In []:

```
lstm_model = tf.keras.models.Sequential([
    # Shape [batch, time, features] => [batch, time, lstm_units]
    tf.keras.layers.LSTM(32, return_sequences=True),
    # Shape => [batch, time, features]
    tf.keras.layers.Dense(units=1)
])
```

With `return_sequences=True`, the model can be trained on 24 hours of data at a time.

Note: This will give a pessimistic view of the model's performance. On the first time step, the model has no access to previous steps and, therefore, can't do any better than the simple `linear` and `dense` models shown earlier.

In []:

```
print('Input shape:', wide_window.example[0].shape)
print('Output shape:', lstm_model(wide_window.example[0]).shape)
```

In []:

```
history = compile_and_fit(lstm_model, wide_window)

IPython.display.clear_output()
val_performance['LSTM'] = lstm_model.evaluate(wide_window.val)
performance['LSTM'] = lstm_model.evaluate(wide_window.test, verbose=0)
```

In []:

```
wide_window.plot(lstm_model)
```

Performance

With this dataset typically each of the models does slightly better than the one before it:

In []:

```
x = np.arange(len(performance))
width = 0.3
metric_name = 'mean_absolute_error'
metric_index = lstm_model.metrics_names.index('mean_absolute_error')
val_mae = [v[metric_index] for v in val_performance.values()]
test_mae = [v[metric_index] for v in performance.values()]

plt.ylabel('mean_absolute_error [T (degC), normalized]')
plt.bar(x - 0.17, val_mae, width, label='Validation')
plt.bar(x + 0.17, test_mae, width, label='Test')
plt.xticks(ticks=x, labels=performance.keys(),
           rotation=45)
_ = plt.legend()
```

In []:

```
for name, value in performance.items():
    print(f'{name:12s}: {value[1]:0.4f}' )
```

Multi-output models

The models so far all predicted a single output feature, `T (degC)`, for a single time step.

All of these models can be converted to predict multiple features just by changing the number of units in the output layer and adjusting the training windows to include all features in the `labels` (`example_labels`):

In []:

```
single_step_window = WindowGenerator(
    # `WindowGenerator` returns all features as labels if you
    # don't set the `label_columns` argument.
    input_width=1, label_width=1, shift=1)

wide_window = WindowGenerator(
    input_width=24, label_width=24, shift=1)

for example_inputs, example_labels in wide_window.train.take(1):
    print(f'Inputs shape (batch, time, features): {example_inputs.shape}')
    print(f'Labels shape (batch, time, features): {example_labels.shape}')
```

Note above that the `features` axis of the labels now has the same depth as the inputs, instead of 1 .

Baseline

The same baseline model (`Baseline`) can be used here, but this time repeating all features instead of selecting a specific `label_index` :

In []:

```
baseline = Baseline()  
baseline.compile(loss=tf.losses.MeanSquaredError(),  
                 metrics=[tf.metrics.MeanAbsoluteError()])
```

In []:

```
val_performance = {}  
performance = {}  
val_performance['Baseline'] = baseline.evaluate(wide_window.val)  
performance['Baseline'] = baseline.evaluate(wide_window.test, verbose=0)
```

Dense

In []:

```
dense = tf.keras.Sequential([  
    tf.keras.layers.Dense(units=64, activation='relu'),  
    tf.keras.layers.Dense(units=64, activation='relu'),  
    tf.keras.layers.Dense(units=num_features)  
)
```

In []:

```
history = compile_and_fit(dense, single_step_window)  
  
IPython.display.clear_output()  
val_performance['Dense'] = dense.evaluate(single_step_window.val)  
performance['Dense'] = dense.evaluate(single_step_window.test, verbose=0)
```

RNN

In []:

```
%%time  
wide_window = WindowGenerator(  
    input_width=24, label_width=24, shift=1)  
  
lstm_model = tf.keras.models.Sequential([  
    # Shape [batch, time, features] => [batch, time, lstm_units]  
    tf.keras.layers.LSTM(32, return_sequences=True),  
    # Shape => [batch, time, features]  
    tf.keras.layers.Dense(units=num_features)  
)  
  
history = compile_and_fit(lstm_model, wide_window)  
  
IPython.display.clear_output()  
val_performance['LSTM'] = lstm_model.evaluate( wide_window.val)  
performance['LSTM'] = lstm_model.evaluate( wide_window.test, verbose=0)  
  
print()
```

Advanced: Residual connections

The Baseline model from earlier took advantage of the fact that the sequence doesn't change drastically from time step to time step. Every model trained in this tutorial so far was randomly initialized, and then had to learn that the output is a small change from the previous time step.

While you can get around this issue with careful initialization, it's simpler to build this into the model structure.

It's common in time series analysis to build models that instead of predicting the next value, predict how the value will change in the next time step. Similarly, [residual networks](https://arxiv.org/abs/1512.03385) (<https://arxiv.org/abs/1512.03385>)—or ResNets—in deep learning refer to architectures where each layer adds to the model's accumulating result.

That is how you take advantage of the knowledge that the change should be small.



Essentially, this initializes the model to match the Baseline. For this task it helps models converge faster, with slightly better performance.

This approach can be used in conjunction with any model discussed in this tutorial.

Here, it is being applied to the LSTM model, note the use of the `tf.initializers.zeros` to ensure that the initial predicted changes are small, and don't overpower the residual connection. There are no symmetry-breaking concerns for the gradients here, since the `zeros` are only used on the last layer.

In []:

```
class ResidualWrapper(tf.keras.Model):
    def __init__(self, model):
        super().__init__()
        self.model = model

    def call(self, inputs, *args, **kwargs):
        delta = self.model(inputs, *args, **kwargs)

        # The prediction for each time step is the input
        # from the previous time step plus the delta
        # calculated by the model.
        return inputs + delta
```

In []:

```
%%time
residual_lstm = ResidualWrapper(
    tf.keras.Sequential([
        tf.keras.layers.LSTM(32, return_sequences=True),
        tf.keras.layers.Dense(
            num_features,
            # The predicted deltas should start small.
            # Therefore, initialize the output layer with zeros.
            kernel_initializer=tf.initializers.zeros()
    ]))

history = compile_and_fit(residual_lstm, wide_window)

IPython.display.clear_output()
val_performance['Residual LSTM'] = residual_lstm.evaluate(wide_window.val)
performance['Residual LSTM'] = residual_lstm.evaluate(wide_window.test, verbose=0)
print()
```

Performance

Here is the overall performance for these multi-output models.

In []:

```
x = np.arange(len(performance))
width = 0.3

metric_name = 'mean_absolute_error'
metric_index = lstm_model.metrics_names.index('mean_absolute_error')
val_mae = [v[metric_index] for v in val_performance.values()]
test_mae = [v[metric_index] for v in performance.values()]

plt.bar(x - 0.17, val_mae, width, label='Validation')
plt.bar(x + 0.17, test_mae, width, label='Test')
plt.xticks(ticks=x, labels=performance.keys(),
           rotation=45)
plt.ylabel('MAE (average over all outputs)')
_ = plt.legend()
```

In []:

```
for name, value in performance.items():
    print(f'{name:15s}: {value[1]:0.4f}')
```

The above performances are averaged across all model outputs.

Multi-step models

Both the single-output and multiple-output models in the previous sections made **single time step predictions**, one hour into the future.

This section looks at how to expand these models to make **multiple time step predictions**.

In a multi-step prediction, the model needs to learn to predict a range of future values. Thus, unlike a single step model, where only a single future point is predicted, a multi-step model predicts a sequence of the future values.

There are two rough approaches to this:

1. Single shot predictions where the entire time series is predicted at once.
2. Autoregressive predictions where the model only makes single step predictions and its output is fed back as its input.

In this section all the models will predict **all the features across all output time steps**.

For the multi-step model, the training data again consists of hourly samples. However, here, the models will learn to predict 24 hours into the future, given 24 hours of the past.

Here is a `Window` object that generates these slices from the dataset:

In []:

```
OUT_STEPS = 24
multi_window = WindowGenerator(input_width=24,
                               label_width=OUT_STEPS,
                               shift=OUT_STEPS)

multi_window.plot()
multi_window
```

Baselines

A simple baseline for this task is to repeat the last input time step for the required number of output time steps:



In []:

```
class MultiStepLastBaseline(tf.keras.Model):
    def call(self, inputs):
        return tf.tile(inputs[:, -1:, :], [1, OUT_STEPS, 1])

last_baseline = MultiStepLastBaseline()
last_baseline.compile(loss=tf.losses.MeanSquaredError(),
                      metrics=[tf.metrics.MeanAbsoluteError()])

multi_val_performance = {}
multi_performance = {}

multi_val_performance['Last'] = last_baseline.evaluate(multi_window.val)
multi_performance['Last'] = last_baseline.evaluate(multi_window.test, verbose=0)
multi_window.plot(last_baseline)
```

Since this task is to predict 24 hours into the future, given 24 hours of the past, another simple approach is to repeat the previous day, assuming tomorrow will be similar:



In []:

```
class RepeatBaseline(tf.keras.Model):
    def call(self, inputs):
        return inputs

repeat_baseline = RepeatBaseline()
repeat_baseline.compile(loss=tf.losses.MeanSquaredError(),
                        metrics=[tf.metrics.MeanAbsoluteError()])

multi_val_performance['Repeat'] = repeat_baseline.evaluate(multi_window.val)
multi_performance['Repeat'] = repeat_baseline.evaluate(multi_window.test, verbose=0)
multi_window.plot(repeat_baseline)
```

Single-shot models

One high-level approach to this problem is to use a "single-shot" model, where the model makes the entire sequence prediction in a single step.

This can be implemented efficiently as a `tf.keras.layers.Dense` with `OUT_STEPS*features` output units. The model just needs to reshape that output to the required `(OUTPUT_STEPS, features)`.

Linear

A simple linear model based on the last input time step does better than either baseline, but is underpowered. The model needs to predict `OUTPUT_STEPS` time steps, from a single input time step with a linear projection. It can only capture a low-dimensional slice of the behavior, likely based mainly on the time of day and time of year.



In []:

```
multi_linear_model = tf.keras.Sequential([
    # Take the last time-step.
    # Shape [batch, time, features] => [batch, 1, features]
    tf.keras.layers.Lambda(lambda x: x[:, -1:, :]),
    # Shape => [batch, 1, out_steps*features]
    tf.keras.layers.Dense(OUT_STEPS*num_features,
                          kernel_initializer=tf.initializers.zeros()),
    # Shape => [batch, out_steps, features]
    tf.keras.layers.Reshape([OUT_STEPS, num_features])
])

history = compile_and_fit(multi_linear_model, multi_window)

IPython.display.clear_output()
multi_val_performance['Linear'] = multi_linear_model.evaluate(multi_window.val)
multi_performance['Linear'] = multi_linear_model.evaluate(multi_window.test, verbose=0)
multi_window.plot(multi_linear_model)
```

Dense

Adding a `tf.keras.layers.Dense` between the input and output gives the linear model more power, but is still only based on a single input time step.

In []:

```
multi_dense_model = tf.keras.Sequential([
    # Take the last time step.
    # Shape [batch, time, features] => [batch, 1, features]
    tf.keras.layers.Lambda(lambda x: x[:, -1:, :]),
    # Shape => [batch, 1, dense_units]
    tf.keras.layers.Dense(512, activation='relu'),
    # Shape => [batch, out_steps*features]
    tf.keras.layers.Dense(OUT_STEPS*num_features,
        kernel_initializer=tf.initializers.zeros()),
    # Shape => [batch, out_steps, features]
    tf.keras.layers.Reshape([OUT_STEPS, num_features])
])

history = compile_and_fit(multi_dense_model, multi_window)

IPython.display.clear_output()
multi_val_performance['Dense'] = multi_dense_model.evaluate(multi_window.val)
multi_performance['Dense'] = multi_dense_model.evaluate(multi_window.test, verbose=0)
multi_window.plot(multi_dense_model)
```

CNN

A convolutional model makes predictions based on a fixed-width history, which may lead to better performance than the dense model since it can see how things are changing over time:

In []:

```
CONV_WIDTH = 3
multi_conv_model = tf.keras.Sequential([
    # Shape [batch, time, features] => [batch, CONV_WIDTH, features]
    tf.keras.layers.Lambda(lambda x: x[:, -CONV_WIDTH:, :]),
    # Shape => [batch, 1, conv_units]
    tf.keras.layers.Conv1D(256, activation='relu', kernel_size=(CONV_WIDTH)),
    # Shape => [batch, 1, out_steps*features]
    tf.keras.layers.Dense(OUT_STEPS*num_features,
        kernel_initializer=tf.initializers.zeros()),
    # Shape => [batch, out_steps, features]
    tf.keras.layers.Reshape([OUT_STEPS, num_features])
])

history = compile_and_fit(multi_conv_model, multi_window)

IPython.display.clear_output()

multi_val_performance['Conv'] = multi_conv_model.evaluate(multi_window.val)
multi_performance['Conv'] = multi_conv_model.evaluate(multi_window.test, verbose=0)
multi_window.plot(multi_conv_model)
```

RNN

A recurrent model can learn to use a long history of inputs, if it's relevant to the predictions the model is making. Here the model will accumulate internal state for 24 hours, before making a single prediction for the next 24 hours.

In this single-shot format, the LSTM only needs to produce an output at the last time step, so set `return_sequences=False` in `tf.keras.layers.LSTM`.

[?]

In []:

```
multi_lstm_model = tf.keras.Sequential([
    # Shape [batch, time, features] => [batch, lstm_units].
    # Adding more `lstm_units` just overfits more quickly.
    tf.keras.layers.LSTM(32, return_sequences=False),
    # Shape => [batch, out_steps*features].
    tf.keras.layers.Dense(OUT_STEPS*num_features,
                          kernel_initializer=tf.initializers.zeros()),
    # Shape => [batch, out_steps, features].
    tf.keras.layers.Reshape([OUT_STEPS, num_features])
])

history = compile_and_fit(multi_lstm_model, multi_window)

IPython.display.clear_output()

multi_val_performance['LSTM'] = multi_lstm_model.evaluate(multi_window.val)
multi_performance['LSTM'] = multi_lstm_model.evaluate(multi_window.test, verbose=0)
multi_window.plot(multi_lstm_model)
```

Advanced: Autoregressive model

The above models all predict the entire output sequence in a single step.

In some cases it may be helpful for the model to decompose this prediction into individual time steps. Then, each model's output can be fed back into itself at each step and predictions can be made conditioned on the previous one, like in the classic [Generating Sequences With Recurrent Neural Networks \(https://arxiv.org/abs/1308.0850\)](#).

One clear advantage to this style of model is that it can be set up to produce output with a varying length.

You could take any of the single-step multi-output models trained in the first half of this tutorial and run in an autoregressive feedback loop, but here you'll focus on building a model that's been explicitly trained to do that.



RNN

This tutorial only builds an autoregressive RNN model, but this pattern could be applied to any model that was designed to output a single time step.

The model will have the same basic form as the single-step LSTM models from earlier: a `tf.keras.layers.LSTM` layer followed by a `tf.keras.layers.Dense` layer that converts the `LSTM` layer's outputs to model predictions.

A `tf.keras.layers.LSTM` is a `tf.keras.layers.LSTMCell` wrapped in the higher level `tf.keras.layers.RNN` that manages the state and sequence results for you (Check out the [Recurrent Neural Networks \(RNN\) with Keras \(https://www.tensorflow.org/guide/keras/rnn\)](#) guide for details).

In this case, the model has to manually manage the inputs for each step, so it uses `tf.keras.layers.LSTMCell` directly for the lower level, single time step interface.

In []:

```
class FeedBack(tf.keras.Model):
    def __init__(self, units, out_steps):
        super().__init__()
        self.out_steps = out_steps
        self.units = units
        self.lstm_cell = tf.keras.layers.LSTMCell(units)
        # Also wrap the LSTMCell in an RNN to simplify the `warmup` method.
        self.lstm_rnn = tf.keras.layers.RNN(self.lstm_cell, return_state=True)
        self.dense = tf.keras.layers.Dense(num_features)
```

In []:

```
feedback_model = FeedBack(units=32, out_steps=OUT_STEPS)
```

The first method this model needs is a `warmup` method to initialize its internal state based on the inputs. Once trained, this state will capture the relevant parts of the input history. This is equivalent to the single-step `LSTM` model from earlier:

```
In [ ]:
```

```
def warmup(self, inputs):
    # inputs.shape => (batch, time, features)
    # x.shape => (batch, lstm_units)
    x, *state = self.lstm_rnn(inputs)

    # predictions.shape => (batch, features)
    prediction = self.dense(x)
    return prediction, state

FeedBack.warmup = warmup
```

This method returns a single time-step prediction and the internal state of the LSTM :

```
In [ ]:
```

```
prediction, state = feedback_model.warmup(multi_window.example[0])
prediction.shape
```

With the RNN's state, and an initial prediction you can now continue iterating the model feeding the predictions at each step back as the input.

The simplest approach for collecting the output predictions is to use a Python list and a `tf.stack` after the loop.

Note: Stacking a Python list like this only works with eager-execution, using `Model.compile(..., run_eagerly=True)` for training, or with a fixed length output. For a dynamic output length, you would need to use a `tf.TensorArray` instead of a Python list, and `tf.range` instead of the Python `range`.

```
In [ ]:
```

```
def call(self, inputs, training=None):
    # Use a TensorArray to capture dynamically unrolled outputs.
    predictions = []
    # Initialize the LSTM state.
    prediction, state = self.warmup(inputs)

    # Insert the first prediction.
    predictions.append(prediction)

    # Run the rest of the prediction steps.
    for n in range(1, self.out_steps):
        # Use the last prediction as input.
        x = prediction
        # Execute one lstm step.
        x, state = self.lstm_cell(x, states=state,
                                  training=training)
        # Convert the lstm output to a prediction.
        prediction = self.dense(x)
        # Add the prediction to the output.
        predictions.append(prediction)

    # predictions.shape => (time, batch, features)
    predictions = tf.stack(predictions)
    # predictions.shape => (batch, time, features)
    predictions = tf.transpose(predictions, [1, 0, 2])
    return predictions

FeedBack.call = call
```

Test run this model on the example inputs:

```
In [ ]:
```

```
print('Output shape (batch, time, features): ', feedback_model(multi_window.example[0]).shape)
```

Now, train the model:

```
In [ ]:
```

```
history = compile_and_fit(feedback_model, multi_window)

IPython.display.clear_output()

multi_val_performance['AR LSTM'] = feedback_model.evaluate(multi_window.val)
multi_performance['AR LSTM'] = feedback_model.evaluate(multi_window.test, verbose=0)
multi_window.plot(feedback_model)
```

Performance

There are clearly diminishing returns as a function of model complexity on this problem:

In []:

```
x = np.arange(len(multi_performance))
width = 0.3

metric_name = 'mean_absolute_error'
metric_index = lstm_model.metrics_names.index('mean_absolute_error')
val_mae = [v[metric_index] for v in multi_val_performance.values()]
test_mae = [v[metric_index] for v in multi_performance.values()]

plt.bar(x - 0.17, val_mae, width, label='Validation')
plt.bar(x + 0.17, test_mae, width, label='Test')
plt.xticks(ticks=x, labels=multi_performance.keys(),
           rotation=45)
plt.ylabel(f'MAE (average over all times and outputs)')
_ = plt.legend()
```

The metrics for the multi-output models in the first half of this tutorial show the performance averaged across all output features. These performances are similar but also averaged across output time steps.

In []:

```
for name, value in multi_performance.items():
    print(f'{name:8s}: {value[1]:0.4f}')
```

The gains achieved going from a dense model to convolutional and recurrent models are only a few percent (if any), and the autoregressive model performed clearly worse. So these more complex approaches may not be worth while on **this** problem, but there was no way to know without trying, and these models could be helpful for **your** problem.

Next steps

This tutorial was a quick introduction to time series forecasting using TensorFlow.

To learn more, refer to:

- Chapter 15 of [Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow](https://www.oreilly.com/library/view/hands-on-machine-learning/9781492032632/) (<https://www.oreilly.com/library/view/hands-on-machine-learning/9781492032632/>), 2nd Edition.
- Chapter 6 of [Deep Learning with Python](https://www.manning.com/books/deep-learning-with-python) (<https://www.manning.com/books/deep-learning-with-python>).
- Lesson 8 of [Udacity's intro to TensorFlow for deep learning](https://www.udacity.com/course/intro-to-tensorflow-for-deep-learning--ud187) (<https://www.udacity.com/course/intro-to-tensorflow-for-deep-learning--ud187>), including the [exercise notebooks](https://github.com/tensorflow/examples/tree/master/courses/udacity_intro_to_tensorflow_for_deep_learning) (https://github.com/tensorflow/examples/tree/master/courses/udacity_intro_to_tensorflow_for_deep_learning).

Also, remember that you can implement any [classical time series model](https://otexts.com/fpp2/index.html) (<https://otexts.com/fpp2/index.html>) in TensorFlow—this tutorial just focuses on TensorFlow's built-in functionality.

Copyright 2019 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Classify structured data using Keras preprocessing layers



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/structured_data/preprocessing_layers)

(https://www.tensorflow.org/tutorials/structured_data/preprocessing_layers) (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/structured_data/)



This tutorial demonstrates how to classify structured data, such as tabular data, using a simplified version of the [PetFinder dataset from a Kaggle competition](https://www.kaggle.com/c/petfinder-adoption-prediction) (<https://www.kaggle.com/c/petfinder-adoption-prediction>) stored in a CSV file.

You will use [Keras](https://www.tensorflow.org/guide/keras) (<https://www.tensorflow.org/guide/keras>) to define the model, and [Keras preprocessing layers](https://www.tensorflow.org/guide/keras/preprocessing_layers) (https://www.tensorflow.org/guide/keras/preprocessing_layers) as a bridge to map from columns in a CSV file to features used to train the model. The goal is to predict if a pet will be adopted.

This tutorial contains complete code for:

- Loading a CSV file into a [DataFrame](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html) (<https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html>) using [pandas](https://pandas.pydata.org/) (<https://pandas.pydata.org/>).
- Building an input pipeline to batch and shuffle the rows using `tf.data`. (Visit [tf.data: Build TensorFlow input pipelines](#) ([..../guide/data.ipynb](#)) for more details.)
- Mapping from columns in the CSV file to features used to train the model with the Keras preprocessing layers.
- Building, training, and evaluating a model using the Keras built-in methods.

Note: This tutorial is similar to [Classify structured data with feature columns](#) ([./structured_data/feature_columns.ipynb](#)). This version uses the [Keras preprocessing layers](https://www.tensorflow.org/guide/keras/preprocessing_layers) (https://www.tensorflow.org/guide/keras/preprocessing_layers) instead of the `tf.feature_column` API, as the former are more intuitive and can be easily included inside your model to simplify deployment.

The PetFinder.my mini dataset

There are several thousand rows in the PetFinder.my mini's CSV dataset file, where each row describes a pet (a dog or a cat) and each column describes an attribute, such as age, breed, color, and so on.

In the dataset's summary below, notice there are mostly numerical and categorical columns. In this tutorial, you will only be dealing with those two feature types, dropping `Description` (a free text feature) and `AdoptionSpeed` (a classification feature) during data preprocessing.

Column	Pet description	Feature type	Data type
Type	Type of animal (Dog , Cat)	Categorical	String
Age	Age	Numerical	Integer
Breed1	Primary breed	Categorical	String
Color1	Color 1	Categorical	String
Color2	Color 2	Categorical	String
MaturitySize	Size at maturity	Categorical	String
FurLength	Fur length	Categorical	String
Vaccinated	Pet has been vaccinated	Categorical	String
Sterilized	Pet has been sterilized	Categorical	String
Health	Health condition	Categorical	String
Fee	Adoption fee	Numerical	Integer
Description	Profile write-up	Text	String
PhotoAmt	Total uploaded photos	Numerical	Integer
AdoptionSpeed	Categorical speed of adoption	Classification	Integer

Import TensorFlow and other libraries

In []:

```
import numpy as np
import pandas as pd
import tensorflow as tf

from tensorflow.keras import layers
```

In []:

```
tf.__version__
```

Load the dataset and read it into a pandas DataFrame

[pandas](https://pandas.pydata.org/) (<https://pandas.pydata.org/>) is a Python library with many helpful utilities for loading and working with structured data. Use `tf.keras.utils.get_file` to download and extract the CSV file with the PetFinder.my mini dataset, and load it into a [DataFrame](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html) (<https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html>) with `'pandas.read_csv'` (https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html):

```
In [ ]:
```

```
dataset_url = 'http://storage.googleapis.com/download.tensorflow.org/data/petfinder-mini.zip'
csv_file = 'datasets/petfinder-mini/petfinder-mini.csv'

tf.keras.utils.get_file('petfinder_mini.zip', dataset_url,
                        extract=True, cache_dir='.')
dataframe = pd.read_csv(csv_file)
```

Inspect the dataset by checking the first five rows of the DataFrame:

```
In [ ]:
```

```
dataframe.head()
```

Create a target variable

The original task in Kaggle's [PetFinder.my Adoption Prediction competition](https://www.kaggle.com/c/petfinder-adoption-prediction) (<https://www.kaggle.com/c/petfinder-adoption-prediction>) was to predict the speed at which a pet will be adopted (e.g. in the first week, the first month, the first three months, and so on).

In this tutorial, you will simplify the task by transforming it into a binary classification problem, where you simply have to predict whether a pet was adopted or not.

After modifying the `AdoptionSpeed` column, `0` will indicate the pet was not adopted, and `1` will indicate it was.

```
In [ ]:
```

```
# In the original dataset, `AdoptionSpeed` of `4` indicates
# a pet was not adopted.
dataframe['target'] = np.where(dataframe['AdoptionSpeed']==4, 0, 1)

# Drop unused features.
dataframe = dataframe.drop(columns=[ 'AdoptionSpeed', 'Description'])
```

Split the DataFrame into training, validation, and test sets

The dataset is in a single pandas DataFrame. Split it into training, validation, and test sets using a, for example, 80:10:10 ratio, respectively:

```
In [ ]:
```

```
train, val, test = np.split(dataframe.sample(frac=1), [int(0.8*len(dataframe)), int(0.9*len(dataframe))])
```

```
In [ ]:
```

```
print(len(train), 'training examples')
print(len(val), 'validation examples')
print(len(test), 'test examples')
```

Create an input pipeline using tf.data

Next, create a utility function that converts each training, validation, and test set DataFrame into a `tf.data.Dataset`, then shuffles and batches the data.

Note: If you were working with a very large CSV file (so large that it does not fit into memory), you would use the `tf.data` API to read it from disk directly. That is not covered in this tutorial.

```
In [ ]:
```

```
def df_to_dataset(dataframe, shuffle=True, batch_size=32):
    df = dataframe.copy()
    labels = df.pop('target')
    df = {key: value[:,tf.newaxis] for key, value in dataframe.items()}
    ds = tf.data.Dataset.from_tensor_slices((dict(df), labels))
    if shuffle:
        ds = ds.shuffle(buffer_size=len(dataframe))
    ds = ds.batch(batch_size)
    ds = ds.prefetch(batch_size)
    return ds
```

Now, use the newly created function (`df_to_dataset`) to check the format of the data the input pipeline helper function returns by calling it on the training data, and use a small batch size to keep the output readable:

```
In [ ]:
```

```
batch_size = 5
train_ds = df_to_dataset(train, batch_size=batch_size)
```

```
In [ ]:
```

```
[(train_features, label_batch)] = train_ds.take(1)
print('Every feature:', list(train_features.keys()))
print('A batch of ages:', train_features['Age'])
print('A batch of targets:', label_batch )
```

As the output demonstrates, the training set returns a dictionary of column names (from the DataFrame) that map to column values from rows.

Apply the Keras preprocessing layers

The Keras preprocessing layers allow you to build Keras-native input processing pipelines, which can be used as independent preprocessing code in non-Keras workflows, combined directly with Keras models, and exported as part of a Keras SavedModel.

In this tutorial, you will use the following four preprocessing layers to demonstrate how to perform preprocessing, structured data encoding, and feature engineering:

- `tf.keras.layers.Normalization` : Performs feature-wise normalization of input features.
- `tf.keras.layers.CategoryEncoding` : Turns integer categorical features into one-hot, multi-hot, or [tf-idf](https://en.wikipedia.org/wiki/Tf%2E2%80%93idf/) (<https://en.wikipedia.org/wiki/Tf%2E2%80%93idf/>) dense representations.
- `tf.keras.layers.StringLookup` : Turns string categorical values into integer indices.
- `tf.keras.layers.IntegerLookup` : Turns integer categorical values into integer indices.

You can learn more about the available layers in the [Working with preprocessing layers](https://www.tensorflow.org/guide/keras/preprocessing_layers) (https://www.tensorflow.org/guide/keras/preprocessing_layers) guide.

- For *numerical features* of the PetFinder.my mini dataset, you will use a `tf.keras.layers.Normalization` layer to standardize the distribution of the data.
- For *categorical features*, such as pet Type s (Dog and Cat strings), you will transform them to multi-hot encoded tensors with `tf.keras.layers.CategoryEncoding`.

Numerical columns

For each numeric feature in the PetFinder.my mini dataset, you will use a `tf.keras.layers.Normalization` layer to standardize the distribution of the data.

Define a new utility function that returns a layer which applies feature-wise normalization to numerical features using that Keras preprocessing layer:

```
In [ ]:
```

```
def get_normalization_layer(name, dataset):
    # Create a Normalization layer for the feature.
    normalizer = layers.Normalization(axis=None)

    # Prepare a Dataset that only yields the feature.
    feature_ds = dataset.map(lambda x, y: x[name])

    # Learn the statistics of the data.
    normalizer.adapt(feature_ds)

    return normalizer
```

Next, test the new function by calling it on the total uploaded pet photo features to normalize 'PhotoAmt' :

```
In [ ]:
```

```
photo_count_col = train_features['PhotoAmt']
layer = get_normalization_layer('PhotoAmt', train_ds)
layer(photo_count_col)
```

Note: If you have many numeric features (hundreds, or more), it is more efficient to concatenate them first and use a single `tf.keras.layers.Normalization` layer.

Categorical columns

Pet Type s in the dataset are represented as strings— Dog s and Cat s—which need to be multi-hot encoded before being fed into the model. The Age feature

Define another new utility function that returns a layer which maps values from a vocabulary to integer indices and multi-hot encodes the features using the `tf.keras.layers.StringLookup`, `tf.keras.layers.IntegerLookup`, and `tf.keras.CategoryEncoding` preprocessing layers:

In []:

```
def get_category_encoding_layer(name, dataset, dtype, max_tokens=None):
    # Create a layer that turns strings into integer indices.
    if dtype == 'string':
        index = layers.StringLookup(max_tokens=max_tokens)
    # Otherwise, create a layer that turns integer values into integer indices.
    else:
        index = layers.IntegerLookup(max_tokens=max_tokens)

    # Prepare a `tf.data.Dataset` that only yields the feature.
    feature_ds = dataset.map(lambda x, y: x[name])

    # Learn the set of possible values and assign them a fixed integer index.
    index.adapt(feature_ds)

    # Encode the integer indices.
    encoder = layers.CategoryEncoding(num_tokens=index.vocabulary_size())

    # Apply multi-hot encoding to the indices. The lambda function captures the
    # layer, so you can use them, or include them in the Keras Functional model later.
    return lambda feature: encoder(index(feature))
```

Test the `get_category_encoding_layer` function by calling it on pet 'Type' features to turn them into multi-hot encoded tensors:

In []:

```
test_type_col = train_features['Type']
test_type_layer = get_category_encoding_layer(name='Type',
                                              dataset=train_ds,
                                              dtype='string')
test_type_layer(test_type_col)
```

Repeat the process on the pet 'Age' features:

In []:

```
test_age_col = train_features['Age']
test_age_layer = get_category_encoding_layer(name='Age',
                                              dataset=train_ds,
                                              dtype='int64',
                                              max_tokens=5)
test_age_layer(test_age_col)
```

Preprocess selected features to train the model on

You have learned how to use several types of Keras preprocessing layers. Next, you will:

- Apply the preprocessing utility functions defined earlier on 13 numerical and categorical features from the PetFinder.my mini dataset.
- Add all the feature inputs to a list.

As mentioned in the beginning, to train the model, you will use the PetFinder.my mini dataset's numerical ('PhotoAmt' , 'Fee') and categorical ('Age' , 'Type' , 'Color1' , 'Color2' , 'Gender' , 'MaturitySize' , 'FurLength' , 'Vaccinated' , 'Sterilized' , 'Health' , 'Breed1') features.

Note: If your aim is to build an accurate model, try a larger dataset of your own, and think carefully about which features are the most meaningful to include, and how they should be represented.

Earlier, you used a small batch size to demonstrate the input pipeline. Let's now create a new input pipeline with a larger batch size of 256:

In []:

```
batch_size = 256
train_ds = df_to_dataset(train, batch_size=batch_size)
val_ds = df_to_dataset(val, shuffle=False, batch_size=batch_size)
test_ds = df_to_dataset(test, shuffle=False, batch_size=batch_size)
```

Normalize the numerical features (the number of pet photos and the adoption fee), and add them to one list of inputs called `encoded_features`:

In []:

```
all_inputs = []
encoded_features = []

# Numerical features.
for header in ['PhotoAmt', 'Fee']:
    numeric_col = tf.keras.Input(shape=(1,), name=header)
    normalization_layer = get_normalization_layer(header, train_ds)
    encoded_numeric_col = normalization_layer(numeric_col)
    all_inputs.append(numeric_col)
    encoded_features.append(encoded_numeric_col)
```

Turn the integer categorical values from the dataset (the pet age) into integer indices, perform multi-hot encoding, and add the resulting feature inputs to `encoded_features`:

In []:

```
age_col = tf.keras.Input(shape=(1,), name='Age', dtype='int64')

encoding_layer = get_category_encoding_layer(name='Age',
                                             dataset=train_ds,
                                             dtype='int64',
                                             max_tokens=5)

encoded_age_col = encoding_layer(age_col)
all_inputs.append(age_col)
encoded_features.append(encoded_age_col)
```

Repeat the same step for the string categorical values:

In []:

```
categorical_cols = ['Type', 'Color1', 'Color2', 'Gender', 'MaturitySize',
                     'FurLength', 'Vaccinated', 'Sterilized', 'Health', 'Breed1']

for header in categorical_cols:
    categorical_col = tf.keras.Input(shape=(1,), name=header, dtype='string')
    encoding_layer = get_category_encoding_layer(name=header,
                                                 dataset=train_ds,
                                                 dtype='string',
                                                 max_tokens=5)

    encoded_categorical_col = encoding_layer(categorical_col)
    all_inputs.append(categorical_col)
    encoded_features.append(encoded_categorical_col)
```

Create, compile, and train the model

The next step is to create a model using the [Keras Functional API](https://www.tensorflow.org/guide/keras/functional) (<https://www.tensorflow.org/guide/keras/functional>). For the first layer in your model, merge the list of feature inputs—`encoded_features`—into one vector via concatenation with `tf.keras.layers.concatenate`.

In []:

```
all_features = tf.keras.layers.concatenate(encoded_features)
x = tf.keras.layers.Dense(32, activation="relu")(all_features)
x = tf.keras.layers.Dropout(0.5)(x)
output = tf.keras.layers.Dense(1)(x)

model = tf.keras.Model(all_inputs, output)
```

Configure the model with Keras `Model.compile`:

In []:

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

Let's visualize the connectivity graph:

In []:

```
# Use `rankdir='LR'` to make the graph horizontal.
tf.keras.utils.plot_model(model, show_shapes=True, rankdir="LR")
```

Next, train and test the model:

In []:

```
model.fit(train_ds, epochs=10, validation_data=val_ds)
```

In []:

```
loss, accuracy = model.evaluate(test_ds)
print("Accuracy", accuracy)
```

Perform inference

The model you have developed can now classify a row from a CSV file directly after you've included the preprocessing layers inside the model itself.

You can now [save and reload the Keras model \(./keras/save_and_load.ipynb\)](#) with `Model.save` and `Model.load_model` before performing inference on new data:

In []:

```
model.save('my_pet_classifier')
reloaded_model = tf.keras.models.load_model('my_pet_classifier')
```

To get a prediction for a new sample, you can simply call the Keras `Model.predict` method. There are just two things you need to do:

1. Wrap scalars into a list so as to have a batch dimension (Models only process batches of data, not single samples).
2. Call `tf.convert_to_tensor` on each feature.

In []:

```
sample = {
    'Type': 'Cat',
    'Age': 3,
    'Breed1': 'Tabby',
    'Gender': 'Male',
    'Color1': 'Black',
    'Color2': 'White',
    'MaturitySize': 'Small',
    'FurLength': 'Short',
    'Vaccinated': 'No',
    'Sterilized': 'No',
    'Health': 'Healthy',
    'Fee': 100,
    'PhotoAmt': 2,
}

input_dict = {name: tf.convert_to_tensor([value]) for name, value in sample.items()}
predictions = reloaded_model.predict(input_dict)
prob = tf.nn.sigmoid(predictions[0])

print(
    "This particular pet had a %.1f percent probability "
    "of getting adopted." % (100 * prob)
)
```

Note: You will typically have better results with deep learning with larger and more complex datasets. When working with a small dataset, such as the simplified PetFinder.my one, you can use a [decision tree](#) (<https://developers.google.com/machine-learning/glossary#decision-tree>) or a [random forest](#) (<https://developers.google.com/machine-learning/glossary#random-forest>) as a strong baseline. The goal of this tutorial is to demonstrate the mechanics of working with structured data, so you have a starting point when working with your own datasets in the future.

Next steps

To learn more about classifying structured data, try working with other datasets. To improve accuracy during training and testing your models, think carefully about which features to include in your model and how they should be represented.

Below are some suggestions for datasets:

- [TensorFlow Datasets: MovieLens](https://www.tensorflow.org/datasets/catalog/movie_lens): A set of movie ratings from a movie recommendation service.
- [TensorFlow Datasets: Wine Quality](https://www.tensorflow.org/datasets/catalog/wine_quality): Two datasets related to red and white variants of the Portuguese "Vinho Verde" wine. You can also find the Red Wine Quality dataset on Kaggle (<https://www.kaggle.com/uciml/red-wine-quality-cortez-et-al-2009>).
- [Kaggle: arXiv Dataset](https://www.kaggle.com/Cornell-University/arxiv): A corpus of 1.7 million scholarly articles from arXiv, covering physics, computer science, math, statistics, electrical engineering, quantitative biology, and economics.

Copyright 2019 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Deep Convolutional Generative Adversarial Network



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/generative/dcgan)

(<https://www.tensorflow.org/tutorials/generative/dcgan>)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/generative/dcgan.ipynb)

(<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/generative/dcgan.ipynb>)

This tutorial demonstrates how to generate images of handwritten digits using a [Deep Convolutional Generative Adversarial Network](https://arxiv.org/pdf/1511.06434.pdf) (<https://arxiv.org/pdf/1511.06434.pdf>) (DCGAN). The code is written using the [Keras Sequential API](https://www.tensorflow.org/guide/keras) (<https://www.tensorflow.org/guide/keras>) with a `tf.GradientTape` training loop.

What are GANs?

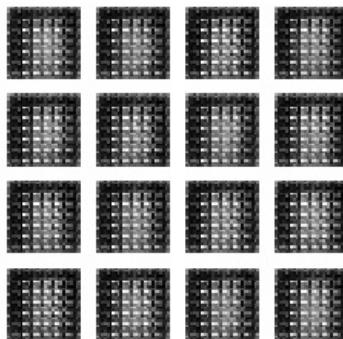
[Generative Adversarial Networks \(https://arxiv.org/abs/1406.2661\)](https://arxiv.org/abs/1406.2661) (GANs) are one of the most interesting ideas in computer science today. Two models are trained simultaneously by an adversarial process. A *generator* ("the artist") learns to create images that look real, while a *discriminator* ("the art critic") learns to tell real images apart from fakes.

[?]

During training, the *generator* progressively becomes better at creating images that look real, while the *discriminator* becomes better at telling them apart. The process reaches equilibrium when the *discriminator* can no longer distinguish real images from fakes.

[?]

This notebook demonstrates this process on the MNIST dataset. The following animation shows a series of images produced by the *generator* as it was trained for 50 epochs. The images begin as random noise, and increasingly resemble hand written digits over time.



To learn more about GANs, see MIT's [Intro to Deep Learning \(http://introtodeeplearning.com/\)](http://introtodeeplearning.com/) course.

Setup

In []:

```
import tensorflow as tf
```

In []:

```
tf.__version__
```

In []:

```
# To generate GIFs
!pip install imageio
!pip install git+https://github.com/tensorflow/docs
```

In []:

```
import glob
import imageio
import matplotlib.pyplot as plt
import numpy as np
import os
import PIL
from tensorflow.keras import layers
import time

from IPython import display
```

Load and prepare the dataset

You will use the MNIST dataset to train the generator and the discriminator. The generator will generate handwritten digits resembling the MNIST data.

In []:

```
(train_images, train_labels), (_, _) = tf.keras.datasets.mnist.load_data()
```

In []:

```
train_images = train_images.reshape(train_images.shape[0], 28, 28, 1).astype('float32')
train_images = (train_images - 127.5) / 127.5 # Normalize the images to [-1, 1]
```

In []:

```
BUFFER_SIZE = 60000
BATCH_SIZE = 256
```

In []:

```
# Batch and shuffle the data
train_dataset = tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
```

Create the models

Both the generator and discriminator are defined using the [Keras Sequential API \(https://www.tensorflow.org/guide/keras#sequential_model\)](https://www.tensorflow.org/guide/keras#sequential_model).

The Generator

The generator uses `tf.keras.layers.Conv2DTranspose` (upsampling) layers to produce an image from a seed (random noise). Start with a `Dense` layer that takes this seed as input, then upsample several times until you reach the desired image size of 28x28x1. Notice the `tf.keras.layers.LeakyReLU` activation for each layer, except the output layer which uses `tanh`.

In []:

```
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((7, 7, 256)))
    assert model.output_shape == (None, 7, 7, 256) # Note: None is the batch size

    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
    assert model.output_shape == (None, 7, 7, 128)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 14, 14, 64)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use_bias=False, activation='tanh'))
    assert model.output_shape == (None, 28, 28, 1)

    return model
```

Use the (as yet untrained) generator to create an image.

In []:

```
generator = make_generator_model()

noise = tf.random.normal([1, 100])
generated_image = generator(noise, training=False)

plt.imshow(generated_image[0, :, :, 0], cmap='gray')
```

The Discriminator

The discriminator is a CNN-based image classifier.

In []:

```
def make_discriminator_model():
    model = tf.keras.Sequential()
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
                           input_shape=[28, 28, 1]))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Flatten())
    model.add(layers.Dense(1))

    return model
```

Use the (as yet untrained) discriminator to classify the generated images as real or fake. The model will be trained to output positive values for real images, and negative values for fake images.

In []:

```
discriminator = make_discriminator_model()
decision = discriminator(generated_image)
print (decision)
```

Define the loss and optimizers

Define loss functions and optimizers for both models.

In []:

```
# This method returns a helper function to compute cross entropy loss
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
```

Discriminator loss

This method quantifies how well the discriminator is able to distinguish real images from fakes. It compares the discriminator's predictions on real images to an array of 1s, and the discriminator's predictions on fake (generated) images to an array of 0s.

In []:

```
def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss
```

Generator loss

The generator's loss quantifies how well it was able to trick the discriminator. Intuitively, if the generator is performing well, the discriminator will classify the fake images as real (or 1). Here, compare the discriminators decisions on the generated images to an array of 1s.

In []:

```
def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)
```

The discriminator and the generator optimizers are different since you will train two networks separately.

In []:

```
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

Save checkpoints

This notebook also demonstrates how to save and restore models, which can be helpful in case a long running training task is interrupted.

In []:

```
checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                                 discriminator_optimizer=discriminator_optimizer,
                                 generator=generator,
                                 discriminator=discriminator)
```

Define the training loop

In []:

```
EPOCHS = 50
noise_dim = 100
num_examples_to_generate = 16

# You will reuse this seed overtime (so it's easier)
# to visualize progress in the animated GIF
seed = tf.random.normal([num_examples_to_generate, noise_dim])
```

The training loop begins with generator receiving a random seed as input. That seed is used to produce an image. The discriminator is then used to classify real images (drawn from the training set) and fakes images (produced by the generator). The loss is calculated for each of these models, and the gradients are used to update the generator and discriminator.

In []:

```
# Notice the use of `tf.function`
# This annotation causes the function to be "compiled".
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)

        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)

        gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
        gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)

        generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
        discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.trainable_variables))
```

In []:

```
def train(dataset, epochs):
    for epoch in range(epochs):
        start = time.time()

        for image_batch in dataset:
            train_step(image_batch)

        # Produce images for the GIF as you go
        display.clear_output(wait=True)
        generate_and_save_images(generator,
                                epoch + 1,
                                seed)

        # Save the model every 15 epochs
        if (epoch + 1) % 15 == 0:
            checkpoint.save(file_prefix = checkpoint_prefix)

        print ('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))

    # Generate after the final epoch
    display.clear_output(wait=True)
    generate_and_save_images(generator,
                            epochs,
                            seed)
```

Generate and save images

```
In [ ]:
```

```
def generate_and_save_images(model, epoch, test_input):
    # Notice `training` is set to False.
    # This is so all layers run in inference mode (batchnorm).
    predictions = model(test_input, training=False)

    fig = plt.figure(figsize=(4, 4))

    for i in range(predictions.shape[0]):
        plt.subplot(4, 4, i+1)
        plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='gray')
        plt.axis('off')

    plt.savefig('image_at_epoch_{:04d}.png'.format(epoch))
    plt.show()
```

Train the model

Call the `train()` method defined above to train the generator and discriminator simultaneously. Note, training GANs can be tricky. It's important that the generator and discriminator do not overpower each other (e.g., that they train at a similar rate).

At the beginning of the training, the generated images look like random noise. As training progresses, the generated digits will look increasingly real. After about 50 epochs, they resemble MNIST digits. This may take about one minute / epoch with the default settings on Colab.

```
In [ ]:
```

```
train(train_dataset, EPOCHS)
```

Restore the latest checkpoint.

```
In [ ]:
```

```
checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))
```

Create a GIF

```
In [ ]:
```

```
# Display a single image using the epoch number
def display_image(epoch_no):
    return PIL.Image.open('image_at_epoch_{:04d}.png'.format(epoch_no))
```

```
In [ ]:
```

```
display_image(EPOCHS)
```

Use `imageio` to create an animated gif using the images saved during training.

```
In [ ]:
```

```
anim_file = 'dcgan.gif'

with imageio.get_writer(anim_file, mode='I') as writer:
    filenames = glob.glob('image*.png')
    filenames = sorted(filenames)
    for filename in filenames:
        image = imageio.imread(filename)
        writer.append_data(image)
    image = imageio.imread(filename)
    writer.append_data(image)
```

```
In [ ]:
```

```
import tensorflow_docs.vis.embed as embed
embed.embed_file(anim_file)
```

Next steps

This tutorial has shown the complete code necessary to write and train a GAN. As a next step, you might like to experiment with a different dataset, for example the Large-scale Celeb Faces Attributes (CelebA) dataset [available on Kaggle](https://www.kaggle.com/jessicali9530/celeba-dataset) (<https://www.kaggle.com/jessicali9530/celeba-dataset>). To learn more about GANs see the [NIPS 2016 Tutorial: Generative Adversarial Networks](https://arxiv.org/abs/1701.00160) (<https://arxiv.org/abs/1701.00160>).

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Intro to Autoencoders



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/generative/autoencoder)

(<https://www.tensorflow.org/tutorials/generative/autoencoder>)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/generative/autoencoder.ipynb)

(<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/generative/autoencoder.ipynb>)

This tutorial introduces autoencoders with three examples: the basics, image denoising, and anomaly detection.

An autoencoder is a special type of neural network that is trained to copy its input to its output. For example, given an image of a handwritten digit, an autoencoder first encodes the image into a lower dimensional latent representation, then decodes the latent representation back to an image. An autoencoder learns to compress the data while minimizing the reconstruction error.

To learn more about autoencoders, please consider reading chapter 14 from [Deep Learning \(<https://www.deeplearningbook.org/>\)](https://www.deeplearningbook.org/) by Ian Goodfellow, Yoshua Bengio, and Aaron Courville.

Import TensorFlow and other libraries

In []:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import tensorflow as tf

from sklearn.metrics import accuracy_score, precision_score, recall_score
from sklearn.model_selection import train_test_split
from tensorflow.keras import layers, losses
from tensorflow.keras.datasets import fashion_mnist
from tensorflow.keras.models import Model
```

Load the dataset

To start, you will train the basic autoencoder using the Fashion MNIST dataset. Each image in this dataset is 28x28 pixels.

In []:

```
(x_train, _), (x_test, _) = fashion_mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.

print(x_train.shape)
print(x_test.shape)
```

First example: Basic autoencoder



Define an autoencoder with two Dense layers: an `encoder`, which compresses the images into a 64 dimensional latent vector, and a `decoder`, that reconstructs the original image from the latent space.

To define your model, use the [Keras Model Subclassing API \(\[https://www.tensorflow.org/guide/keras/custom_layers_and_models\]\(https://www.tensorflow.org/guide/keras/custom_layers_and_models\)\)](https://www.tensorflow.org/guide/keras/custom_layers_and_models).

In []:

```
latent_dim = 64

class Autoencoder(Model):
    def __init__(self, latent_dim):
        super(Autoencoder, self).__init__()
        self.latent_dim = latent_dim
        self.encoder = tf.keras.Sequential([
            layers.Flatten(),
            layers.Dense(latent_dim, activation='relu'),
        ])
        self.decoder = tf.keras.Sequential([
            layers.Dense(784, activation='sigmoid'),
            layers.Reshape((28, 28))
        ])

    def call(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded

autoencoder = Autoencoder(latent_dim)
```

In []:

```
autoencoder.compile(optimizer='adam', loss=losses.MeanSquaredError())
```

Train the model using `x_train` as both the input and the target. The `encoder` will learn to compress the dataset from 784 dimensions to the latent space, and the `decoder` will learn to reconstruct the original images. .

In []:

```
autoencoder.fit(x_train, x_train,
                 epochs=10,
                 shuffle=True,
                 validation_data=(x_test, x_test))
```

Now that the model is trained, let's test it by encoding and decoding images from the test set.

In []:

```
encoded_imgs = autoencoder.encoder(x_test).numpy()
decoded_imgs = autoencoder.decoder(encoded_imgs).numpy()
```

In []:

```
n = 10
plt.figure(figsize=(20, 4))
for i in range(n):
    # display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i])
    plt.title("original")
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i])
    plt.title("reconstructed")
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

Second example: Image denoising



An autoencoder can also be trained to remove noise from images. In the following section, you will create a noisy version of the Fashion MNIST dataset by applying random noise to each image. You will then train an autoencoder using the noisy image as input, and the original image as the target.

Let's reimport the dataset to omit the modifications made earlier.

In []:

```
(x_train, _), (x_test, _) = fashion_mnist.load_data()
```

In []:

```
x_train = x_train.astype('float32') / 255.  
x_test = x_test.astype('float32') / 255.  
  
x_train = x_train[..., tf.newaxis]  
x_test = x_test[..., tf.newaxis]  
  
print(x_train.shape)
```

Adding random noise to the images

In []:

```
noise_factor = 0.2  
x_train_noisy = x_train + noise_factor * tf.random.normal(shape=x_train.shape)  
x_test_noisy = x_test + noise_factor * tf.random.normal(shape=x_test.shape)  
  
x_train_noisy = tf.clip_by_value(x_train_noisy, clip_value_min=0., clip_value_max=1.)  
x_test_noisy = tf.clip_by_value(x_test_noisy, clip_value_min=0., clip_value_max=1.)
```

Plot the noisy images.

In []:

```
n = 10  
plt.figure(figsize=(20, 2))  
for i in range(n):  
    ax = plt.subplot(1, n, i + 1)  
    plt.title("original + noise")  
    plt.imshow(tf.squeeze(x_test_noisy[i]))  
    plt.gray()  
plt.show()
```

Define a convolutional autoencoder

In this example, you will train a convolutional autoencoder using [Conv2D](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D) (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D) layers in the encoder , and [Conv2DTranspose](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2DTranspose) (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2DTranspose) layers in the decoder .

In []:

```
class Denoise(Model):  
    def __init__(self):  
        super(Denoise, self).__init__()  
        self.encoder = tf.keras.Sequential([  
            layers.Input(shape=(28, 28, 1)),  
            layers.Conv2D(16, (3, 3), activation='relu', padding='same', strides=2),  
            layers.Conv2D(8, (3, 3), activation='relu', padding='same', strides=2)])  
  
        self.decoder = tf.keras.Sequential([  
            layers.Conv2DTranspose(8, kernel_size=3, strides=2, activation='relu', padding='same'),  
            layers.Conv2DTranspose(16, kernel_size=3, strides=2, activation='relu', padding='same'),  
            layers.Conv2D(1, kernel_size=(3, 3), activation='sigmoid', padding='same')])  
  
    def call(self, x):  
        encoded = self.encoder(x)  
        decoded = self.decoder(encoded)  
        return decoded  
  
autoencoder = Denoise()
```

In []:

```
autoencoder.compile(optimizer='adam', loss=losses.MeanSquaredError())
```

In []:

```
autoencoder.fit(x_train_noisy, x_train,  
                 epochs=10,  
                 shuffle=True,  
                 validation_data=(x_test_noisy, x_test))
```

Let's take a look at a summary of the encoder. Notice how the images are downsampled from 28x28 to 7x7.

```
In [ ]:
```

```
autoencoder.encoder.summary()
```

The decoder upsamples the images back from 7x7 to 28x28.

```
In [ ]:
```

```
autoencoder.decoder.summary()
```

Plotting both the noisy images and the denoised images produced by the autoencoder.

```
In [ ]:
```

```
encoded_imgs = autoencoder.encoder(x_test_noisy).numpy()
decoded_imgs = autoencoder.decoder(encoded_imgs).numpy()
```

```
In [ ]:
```

```
n = 10
plt.figure(figsize=(20, 4))
for i in range(n):

    # display original + noise
    ax = plt.subplot(2, n, i + 1)
    plt.title("original + noise")
    plt.imshow(tf.squeeze(x_test_noisy[i]))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction
    bx = plt.subplot(2, n, i + n + 1)
    plt.title("reconstructed")
    plt.imshow(tf.squeeze(decoded_imgs[i]))
    plt.gray()
    bx.get_xaxis().set_visible(False)
    bx.get_yaxis().set_visible(False)
plt.show()
```

Third example: Anomaly detection

Overview

In this example, you will train an autoencoder to detect anomalies on the [ECG5000 dataset](http://www.timeseriesclassification.com/description.php?Dataset=ECG5000) (<http://www.timeseriesclassification.com/description.php?Dataset=ECG5000>). This dataset contains 5,000 [Electrocardiograms](https://en.wikipedia.org/wiki/Electrocardiography) (<https://en.wikipedia.org/wiki/Electrocardiography>), each with 140 data points. You will use a simplified version of the dataset, where each example has been labeled either 0 (corresponding to an abnormal rhythm), or 1 (corresponding to a normal rhythm). You are interested in identifying the abnormal rhythms.

Note: This is a labeled dataset, so you could phrase this as a supervised learning problem. The goal of this example is to illustrate anomaly detection concepts you can apply to larger datasets, where you do not have labels available (for example, if you had many thousands of normal rhythms, and only a small number of abnormal rhythms).

How will you detect anomalies using an autoencoder? Recall that an autoencoder is trained to minimize reconstruction error. You will train an autoencoder on the normal rhythms only, then use it to reconstruct all the data. Our hypothesis is that the abnormal rhythms will have higher reconstruction error. You will then classify a rhythm as an anomaly if the reconstruction error surpasses a fixed threshold.

Load ECG data

The dataset you will use is based on one from [timeseriesclassification.com](http://www.timeseriesclassification.com/description.php?Dataset=ECG5000) (<http://www.timeseriesclassification.com/description.php?Dataset=ECG5000>).

```
In [ ]:
```

```
# Download the dataset
dataframe = pd.read_csv('http://storage.googleapis.com/download.tensorflow.org/data/ecg.csv', header=None)
raw_data = dataframe.values
dataframe.head()
```

In []:

```
# The last element contains the labels
labels = raw_data[:, -1]

# The other data points are the electrocardiogram data
data = raw_data[:, 0:-1]

train_data, test_data, train_labels, test_labels = train_test_split(
    data, labels, test_size=0.2, random_state=21
)
```

Normalize the data to [0,1].

In []:

```
min_val = tf.reduce_min(train_data)
max_val = tf.reduce_max(train_data)

train_data = (train_data - min_val) / (max_val - min_val)
test_data = (test_data - min_val) / (max_val - min_val)

train_data = tf.cast(train_data, tf.float32)
test_data = tf.cast(test_data, tf.float32)
```

You will train the autoencoder using only the normal rhythms, which are labeled in this dataset as 1. Separate the normal rhythms from the abnormal rhythms.

In []:

```
train_labels = train_labels.astype(bool)
test_labels = test_labels.astype(bool)

normal_train_data = train_data[train_labels]
normal_test_data = test_data[test_labels]

anomalous_train_data = train_data[~train_labels]
anomalous_test_data = test_data[~test_labels]
```

Plot a normal ECG.

In []:

```
plt.grid()
plt.plot(np.arange(140), normal_train_data[0])
plt.title("A Normal ECG")
plt.show()
```

Plot an anomalous ECG.

In []:

```
plt.grid()
plt.plot(np.arange(140), anomalous_train_data[0])
plt.title("An Anomalous ECG")
plt.show()
```

Build the model

In []:

```
class AnomalyDetector(Model):
    def __init__(self):
        super(AnomalyDetector, self).__init__()
        self.encoder = tf.keras.Sequential([
            layers.Dense(32, activation="relu"),
            layers.Dense(16, activation="relu"),
            layers.Dense(8, activation="relu")])

        self.decoder = tf.keras.Sequential([
            layers.Dense(16, activation="relu"),
            layers.Dense(32, activation="relu"),
            layers.Dense(140, activation="sigmoid")])

    def call(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded

autoencoder = AnomalyDetector()
```

In []:

```
autoencoder.compile(optimizer='adam', loss='mae')
```

Notice that the autoencoder is trained using only the normal ECGs, but is evaluated using the full test set.

In []:

```
history = autoencoder.fit(normal_train_data, normal_train_data,
                           epochs=20,
                           batch_size=512,
                           validation_data=(test_data, test_data),
                           shuffle=True)
```

In []:

```
plt.plot(history.history["loss"], label="Training Loss")
plt.plot(history.history["val_loss"], label="Validation Loss")
plt.legend()
```

You will soon classify an ECG as anomalous if the reconstruction error is greater than one standard deviation from the normal training examples. First, let's plot a normal ECG from the training set, the reconstruction after it's encoded and decoded by the autoencoder, and the reconstruction error.

In []:

```
encoded_data = autoencoder.encoder(normal_test_data).numpy()
decoded_data = autoencoder.decoder(encoded_data).numpy()

plt.plot(normal_test_data[0], 'b')
plt.plot(decoded_data[0], 'r')
plt.fill_between(np.arange(140), decoded_data[0], normal_test_data[0], color='lightcoral')
plt.legend(labels=["Input", "Reconstruction", "Error"])
plt.show()
```

Create a similar plot, this time for an anomalous test example.

In []:

```
encoded_data = autoencoder.encoder(anomalous_test_data).numpy()
decoded_data = autoencoder.decoder(encoded_data).numpy()

plt.plot(anomalous_test_data[0], 'b')
plt.plot(decoded_data[0], 'r')
plt.fill_between(np.arange(140), decoded_data[0], anomalous_test_data[0], color='lightcoral')
plt.legend(labels=["Input", "Reconstruction", "Error"])
plt.show()
```

Detect anomalies

Detect anomalies by calculating whether the reconstruction loss is greater than a fixed threshold. In this tutorial, you will calculate the mean average error for normal examples from the training set, then classify future examples as anomalous if the reconstruction error is higher than one standard deviation from the training set.

Plot the reconstruction error on normal ECGs from the training set

In []:

```
reconstructions = autoencoder.predict(normal_train_data)
train_loss = tf.keras.losses.mae(reconstructions, normal_train_data)

plt.hist(train_loss[None,:], bins=50)
plt.xlabel("Train loss")
plt.ylabel("No of examples")
plt.show()
```

Choose a threshold value that is one standard deviations above the mean.

In []:

```
threshold = np.mean(train_loss) + np.std(train_loss)
print("Threshold: ", threshold)
```

Note: There are other strategies you could use to select a threshold value above which test examples should be classified as anomalous, the correct approach will depend on your dataset. You can learn more with the links at the end of this tutorial.

If you examine the reconstruction error for the anomalous examples in the test set, you'll notice most have greater reconstruction error than the threshold. By varying the threshold, you can adjust the [precision](https://developers.google.com/machine-learning/glossary#precision) (<https://developers.google.com/machine-learning/glossary#precision>) and [recall](https://developers.google.com/machine-learning/glossary#recall) (<https://developers.google.com/machine-learning/glossary#recall>) of your classifier.

In []:

```
reconstructions = autoencoder.predict(anomalous_test_data)
test_loss = tf.keras.losses.mae(reconstructions, anomalous_test_data)

plt.hist(test_loss[None, :], bins=50)
plt.xlabel("Test loss")
plt.ylabel("No of examples")
plt.show()
```

Classify an ECG as an anomaly if the reconstruction error is greater than the threshold.

In []:

```
def predict(model, data, threshold):
    reconstructions = model(data)
    loss = tf.keras.losses.mae(reconstructions, data)
    return tf.math.less(loss, threshold)

def print_stats(predictions, labels):
    print("Accuracy = {}".format(accuracy_score(labels, predictions)))
    print("Precision = {}".format(precision_score(labels, predictions)))
    print("Recall = {}".format(recall_score(labels, predictions)))
```

In []:

```
preds = predict(autoencoder, test_data, threshold)
print_stats(preds, test_labels)
```

Next steps

To learn more about anomaly detection with autoencoders, check out this excellent [interactive example](https://anomagram.fastforwardlabs.com/#/) (<https://anomagram.fastforwardlabs.com/#/>) built with TensorFlow.js by Victor Dibia. For a real-world use case, you can learn how [Airbus Detects Anomalies in ISS Telemetry Data](https://blog.tensorflow.org/2020/04/how-airbus-detects-anomalies-iss-telemetry-data-tfx.html) (<https://blog.tensorflow.org/2020/04/how-airbus-detects-anomalies-iss-telemetry-data-tfx.html>) using TensorFlow. To learn more about the basics, consider reading this [blog post](https://blog.keras.io/building-autoencoders-in-keras.html) (<https://blog.keras.io/building-autoencoders-in-keras.html>) by François Chollet. For more details, check out chapter 14 from [Deep Learning](https://www.deeplearningbook.org/) (<https://www.deeplearningbook.org/>) by Ian Goodfellow, Yoshua Bengio, and Aaron Courville.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

CycleGAN



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/generative/cyclegan)

(<https://www.tensorflow.org/tutorials/generative/cyclegan>)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/generative/cyclegan.ipynb)

(<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/generative/cyclegan.ipynb>)

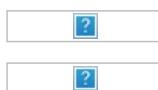
This notebook demonstrates unpaired image to image translation using conditional GAN's, as described in [Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks](https://arxiv.org/abs/1703.10593) (<https://arxiv.org/abs/1703.10593>), also known as CycleGAN. The paper proposes a method that can capture the characteristics of one image domain and figure out how these characteristics could be translated into another image domain, all in the absence of any paired training examples.

This notebook assumes you are familiar with Pix2Pix, which you can learn about in the [Pix2Pix tutorial](https://www.tensorflow.org/tutorials/generative/pix2pix)

(<https://www.tensorflow.org/tutorials/generative/pix2pix>). The code for CycleGAN is similar, the main difference is an additional loss function, and the use of unpaired training data.

CycleGAN uses a cycle consistency loss to enable training without the need for paired data. In other words, it can translate from one domain to another without a one-to-one mapping between the source and target domain.

This opens up the possibility to do a lot of interesting tasks like photo-enhancement, image colorization, style transfer, etc. All you need is the source and the target dataset (which is simply a directory of images).



Set up the input pipeline

Install the [tensorflow_examples](https://github.com/tensorflow/examples) (<https://github.com/tensorflow/examples>) package that enables importing of the generator and the discriminator.

In []:

```
!pip install git+https://github.com/tensorflow/examples.git
```

In []:

```
import tensorflow as tf
```

In []:

```
import tensorflow_datasets as tfds
from tensorflow_examples.models.pix2pix import pix2pix

import os
import time
import matplotlib.pyplot as plt
from IPython.display import clear_output

AUTOTUNE = tf.data.AUTOTUNE
```

Input Pipeline

This tutorial trains a model to translate from images of horses, to images of zebras. You can find this dataset and similar ones [here](https://www.tensorflow.org/datasets/catalog/cycle_gan) (https://www.tensorflow.org/datasets/catalog/cycle_gan).

As mentioned in the [paper](https://arxiv.org/abs/1703.10593) (<https://arxiv.org/abs/1703.10593>), apply random jittering and mirroring to the training dataset. These are some of the image augmentation techniques that avoids overfitting.

This is similar to what was done in [pix2pix](https://www.tensorflow.org/tutorials/generative/pix2pix#load_the_dataset) (https://www.tensorflow.org/tutorials/generative/pix2pix#load_the_dataset)

- In random jittering, the image is resized to 286 x 286 and then randomly cropped to 256 x 256 .
- In random mirroring, the image is randomly flipped horizontally i.e left to right.

In []:

```
dataset, metadata = tfds.load('cycle_gan/horse2zebra',
                             with_info=True, as_supervised=True)

train_horses, train_zebras = dataset['trainA'], dataset['trainB']
test_horses, test_zebras = dataset['testA'], dataset['testB']
```

In []:

```
BUFFER_SIZE = 1000
BATCH_SIZE = 1
IMG_WIDTH = 256
IMG_HEIGHT = 256
```

In []:

```
def random_crop(image):
    cropped_image = tf.image.random_crop(
        image, size=[IMG_HEIGHT, IMG_WIDTH, 3])

    return cropped_image
```

In []:

```
# normalizing the images to [-1, 1]
def normalize(image):
    image = tf.cast(image, tf.float32)
    image = (image / 127.5) - 1
    return image
```

In []:

```
def random_jitter(image):
    # resizing to 286 x 286 x 3
    image = tf.image.resize(image, [286, 286],
                           method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)

    # randomly cropping to 256 x 256 x 3
    image = random_crop(image)

    # random mirroring
    image = tf.image.random_flip_left_right(image)

    return image
```

In []:

```
def preprocess_image_train(image, label):
    image = random_jitter(image)
    image = normalize(image)
    return image
```

In []:

```
def preprocess_image_test(image, label):
    image = normalize(image)
    return image
```

In []:

```
train_horses = train_horses.cache().map(  
    preprocess_image_train, num_parallel_calls=AUTOTUNE).shuffle(  
    BUFFER_SIZE).batch(BATCH_SIZE)  
  
train_zebras = train_zebras.cache().map(  
    preprocess_image_train, num_parallel_calls=AUTOTUNE).shuffle(  
    BUFFER_SIZE).batch(BATCH_SIZE)  
  
test_horses = test_horses.map(  
    preprocess_image_test, num_parallel_calls=AUTOTUNE).cache().shuffle(  
    BUFFER_SIZE).batch(BATCH_SIZE)  
  
test_zebras = test_zebras.map(  
    preprocess_image_test, num_parallel_calls=AUTOTUNE).cache().shuffle(  
    BUFFER_SIZE).batch(BATCH_SIZE)
```

In []:

```
sample_horse = next(iter(train_horses))  
sample_zebra = next(iter(train_zebras))
```

In []:

```
plt.subplot(121)  
plt.title('Horse')  
plt.imshow(sample_horse[0] * 0.5 + 0.5)  
  
plt.subplot(122)  
plt.title('Horse with random jitter')  
plt.imshow(random_jitter(sample_horse[0]) * 0.5 + 0.5)
```

In []:

```
plt.subplot(121)  
plt.title('Zebra')  
plt.imshow(sample_zebra[0] * 0.5 + 0.5)  
  
plt.subplot(122)  
plt.title('Zebra with random jitter')  
plt.imshow(random_jitter(sample_zebra[0]) * 0.5 + 0.5)
```

Import and reuse the Pix2Pix models

Import the generator and the discriminator used in [Pix2Pix](#)

(https://github.com/tensorflow/examples/blob/master/tensorflow_examples/models/pix2pix/pix2pix.py) via the installed [tensorflow_examples](#) (<https://github.com/tensorflow/examples>) package.

The model architecture used in this tutorial is very similar to what was used in [pix2pix](#)

(https://github.com/tensorflow/examples/blob/master/tensorflow_examples/models/pix2pix/pix2pix.py). Some of the differences are:

- Cyclegan uses [instance normalization](https://arxiv.org/abs/1607.08022) (<https://arxiv.org/abs/1607.08022>) instead of [batch normalization](https://arxiv.org/abs/1502.03167) (<https://arxiv.org/abs/1502.03167>).
- The [CycleGAN paper](https://arxiv.org/abs/1703.10593) (<https://arxiv.org/abs/1703.10593>) uses a modified resnet based generator. This tutorial is using a modified unet generator for simplicity.

There are 2 generators (G and F) and 2 discriminators (X and Y) being trained here.

- Generator G learns to transform image X to image Y . ($G: X \rightarrow Y$)
- Generator F learns to transform image Y to image X . ($F: Y \rightarrow X$)
- Discriminator D_X learns to differentiate between image X and generated image X ($F(Y)$).
- Discriminator D_Y learns to differentiate between image Y and generated image Y ($G(X)$).



In []:

```
OUTPUT_CHANNELS = 3  
  
generator_g = pix2pix.unet_generator(OUTPUT_CHANNELS, norm_type='instancenorm')  
generator_f = pix2pix.unet_generator(OUTPUT_CHANNELS, norm_type='instancenorm')  
  
discriminator_x = pix2pix.discriminator(norm_type='instancenorm', target=False)  
discriminator_y = pix2pix.discriminator(norm_type='instancenorm', target=False)
```

In []:

```
to_zebra = generator_g(sample_horse)
to_horse = generator_f(sample_zebra)
plt.figure(figsize=(8, 8))
contrast = 8

imgs = [sample_horse, to_zebra, sample_zebra, to_horse]
title = ['Horse', 'To Zebra', 'Zebra', 'To Horse']

for i in range(len(imgs)):
    plt.subplot(2, 2, i+1)
    plt.title(title[i])
    if i % 2 == 0:
        plt.imshow(imgs[i][0] * 0.5 + 0.5)
    else:
        plt.imshow(imgs[i][0] * 0.5 * contrast + 0.5)
plt.show()
```

In []:

```
plt.figure(figsize=(8, 8))

plt.subplot(121)
plt.title('Is a real zebra?')
plt.imshow(discriminator_y(sample_zebra)[0, ..., -1], cmap='RdBu_r')

plt.subplot(122)
plt.title('Is a real horse?')
plt.imshow(discriminator_x(sample_horse)[0, ..., -1], cmap='RdBu_r')

plt.show()
```

Loss functions

In CycleGAN, there is no paired data to train on, hence there is no guarantee that the input `x` and the target `y` pair are meaningful during training. Thus in order to enforce that the network learns the correct mapping, the authors propose the cycle consistency loss.

The discriminator loss and the generator loss are similar to the ones used in [pix2pix](#) (https://www.tensorflow.org/tutorials/generative/pix2pix#build_the_generator).

In []:

```
LAMBDA = 10
```

In []:

```
loss_obj = tf.keras.losses.BinaryCrossentropy(from_logits=True)
```

In []:

```
def discriminator_loss(real, generated):
    real_loss = loss_obj(tf.ones_like(real), real)

    generated_loss = loss_obj(tf.zeros_like(generated), generated)

    total_disc_loss = real_loss + generated_loss

    return total_disc_loss * 0.5
```

In []:

```
def generator_loss(generated):
    return loss_obj(tf.ones_like(generated), generated)
```

Cycle consistency means the result should be close to the original input. For example, if one translates a sentence from English to French, and then translates it back from French to English, then the resulting sentence should be the same as the original sentence.

In cycle consistency loss,

- Image X is passed via generator G that yields generated image \hat{Y} .
- Generated image \hat{Y} is passed via generator F that yields cycled image \hat{X} .
- Mean absolute error is calculated between X and \hat{X} .

$$\text{forward cycle consistency loss: } X \rightarrow G(X) \rightarrow F(G(X)) \sim \hat{X}$$

$$\text{backward cycle consistency loss: } Y \rightarrow F(Y) \rightarrow G(F(Y)) \sim \hat{Y}$$



In []:

```
def calc_cycle_loss(real_image, cycled_image):  
    loss1 = tf.reduce_mean(tf.abs(real_image - cycled_image))  
  
    return LAMBDA * loss1
```

As shown above, generator G is responsible for translating image X to image Y . Identity loss says that, if you fed image Y to generator G , it should yield the real image Y or something close to image Y .

If you run the zebra-to-horse model on a horse or the horse-to-zebra model on a zebra, it should not modify the image much since the image already contains the target class.

$$\text{Identity loss} = |G(Y) - Y| + |F(X) - X|$$

In []:

```
def identity_loss(real_image, same_image):  
    loss = tf.reduce_mean(tf.abs(real_image - same_image))  
    return LAMBDA * 0.5 * loss
```

Initialize the optimizers for all the generators and the discriminators.

In []:

```
generator_g_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)  
generator_f_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)  
  
discriminator_x_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)  
discriminator_y_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
```

Checkpoints

In []:

```
checkpoint_path = "./checkpoints/train"  
  
ckpt = tf.train.Checkpoint(generator_g=generator_g,  
                            generator_f=generator_f,  
                            discriminator_x=discriminator_x,  
                            discriminator_y=discriminator_y,  
                            generator_g_optimizer=generator_g_optimizer,  
                            generator_f_optimizer=generator_f_optimizer,  
                            discriminator_x_optimizer=discriminator_x_optimizer,  
                            discriminator_y_optimizer=discriminator_y_optimizer)  
  
ckpt_manager = tf.train.CheckpointManager(ckpt, checkpoint_path, max_to_keep=5)  
  
# if a checkpoint exists, restore the latest checkpoint.  
if ckpt_manager.latest_checkpoint:  
    ckpt.restore(ckpt_manager.latest_checkpoint)  
    print ('Latest checkpoint restored!!')
```

Training

Note: This example model is trained for fewer epochs (40) than the paper (200) to keep training time reasonable for this tutorial. Predictions may be less accurate.

```
In [ ]:
```

```
EPOCHS = 40
```

```
In [ ]:
```

```
def generate_images(model, test_input):
    prediction = model(test_input)

    plt.figure(figsize=(12, 12))

    display_list = [test_input[0], prediction[0]]
    title = ['Input Image', 'Predicted Image']

    for i in range(2):
        plt.subplot(1, 2, i+1)
        plt.title(title[i])
        # getting the pixel values between [0, 1] to plot it.
        plt.imshow(display_list[i] * 0.5 + 0.5)
        plt.axis('off')
    plt.show()
```

Even though the training loop looks complicated, it consists of four basic steps:

- Get the predictions.
- Calculate the loss.
- Calculate the gradients using backpropagation.
- Apply the gradients to the optimizer.

In []:

In []:

```
for epoch in range(EPOCHS):
    start = time.time()

    n = 0
    for image_x, image_y in tf.data.Dataset.zip((train_horses, train_zebras)):
        train_step(image_x, image_y)
        if n % 10 == 0:
            print ('.', end='')
        n += 1

    clear_output(wait=True)
    # Using a consistent image (sample_horse) so that the progress of the model
    # is clearly visible.
    generate_images(generator_g, sample_horse)

    if (epoch + 1) % 5 == 0:
        ckpt_save_path = ckpt_manager.save()
        print ('Saving checkpoint for epoch {} at {}'.format(epoch+1,
                                                               ckpt_save_path))

    print ('Time taken for epoch {} is {} sec\n'.format(epoch + 1,
                                                       time.time()-start))
```

Generate using test dataset

In []:

```
# Run the trained model on the test dataset
for inp in test_horses.take(5):
    generate_images(generator_g, inp)
```

Next steps

This tutorial has shown how to implement CycleGAN starting from the generator and discriminator implemented in the [Pix2Pix](#) (<https://www.tensorflow.org/tutorials/generative/pix2pix>) tutorial. As a next step, you could try using a different dataset from [TensorFlow Datasets](#) (https://www.tensorflow.org/datasets/catalog/cycle_gan).

You could also train for a larger number of epochs to improve the results, or you could implement the modified ResNet generator used in the [paper](#) (<https://arxiv.org/abs/1703.10593>) instead of the U-Net generator used here.

Copyright 2019 The TensorFlow Authors.

Licensed under the Apache License, Version 2.0 (the "License");

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

pix2pix: Image-to-image translation with a conditional GAN



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/generative/pix2pix)

(<https://www.tensorflow.org/tutorials/generative/pix2pix>)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/generative/pix2pix.ipynb)

(<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/generative/pix2pix.ipynb>)

This tutorial demonstrates how to build and train a conditional generative adversarial network (cGAN) called pix2pix that learns a mapping from input images to output images, as described in [Image-to-image translation with conditional adversarial networks \(<https://arxiv.org/abs/1611.07004>\)](https://arxiv.org/abs/1611.07004) by Isola et al. (2017). pix2pix is not application specific—it can be applied to a wide range of tasks, including synthesizing photos from label maps, generating colorized photos from black and white images, turning Google Maps photos into aerial images, and even transforming sketches into photos.

In this example, your network will generate images of building facades using the [CMP Facade Database \(<http://cmp.felk.cvut.cz/~tylecr1/facade/>\)](http://cmp.felk.cvut.cz/~tylecr1/facade/) provided by the [Center for Machine Perception \(<http://cmp.felk.cvut.cz/>\)](http://cmp.felk.cvut.cz/) at the [Czech Technical University in Prague \(<https://www.cvut.cz/>\)](https://www.cvut.cz/). To keep it short, you will use a preprocessed copy ([\(<https://people.eecs.berkeley.edu/~tinghuiz/projects/pix2pix/datasets/>\)](https://people.eecs.berkeley.edu/~tinghuiz/projects/pix2pix/datasets/)) of this dataset created by the pix2pix authors.

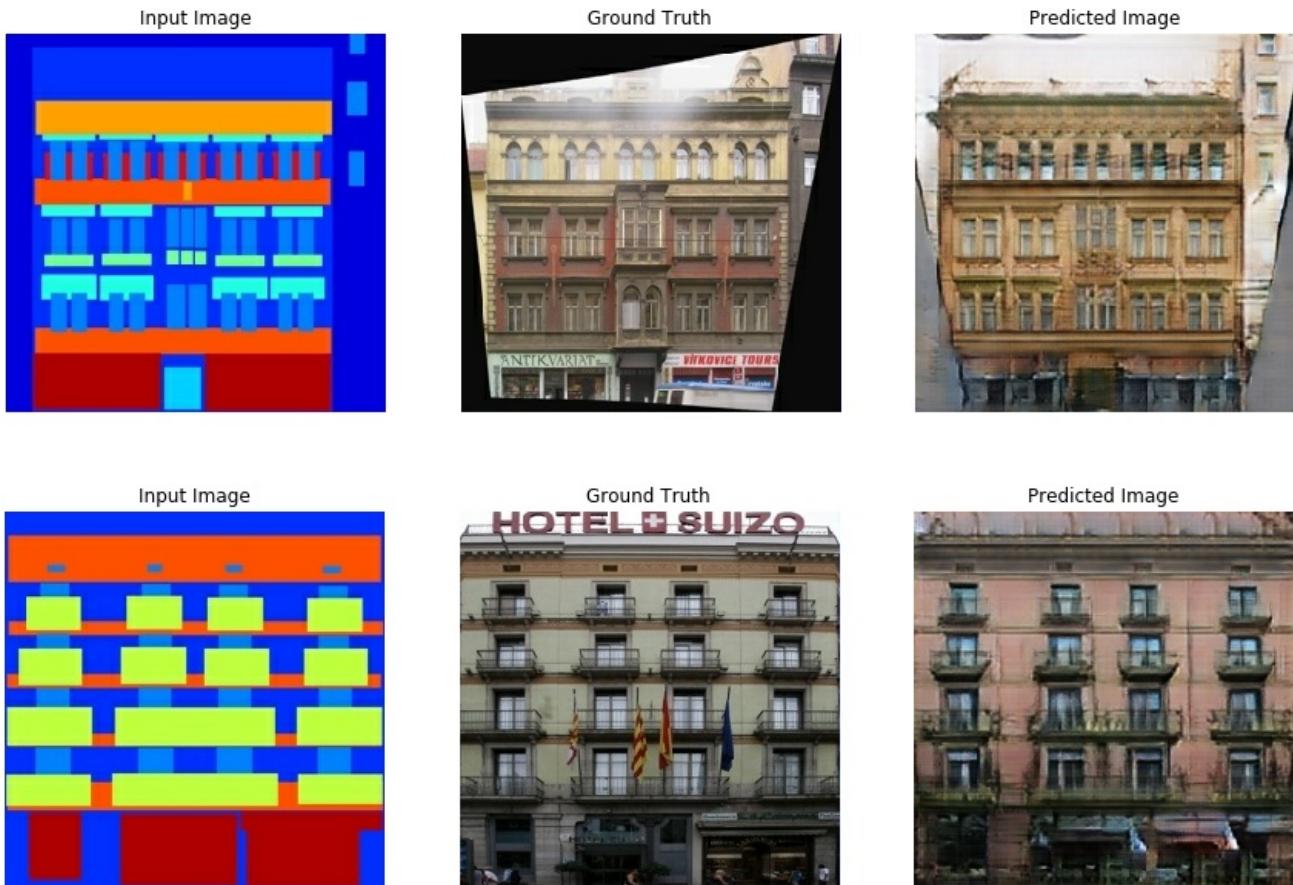
In the pix2pix cGAN, you condition on input images and generate corresponding output images. cGANs were first proposed in [Conditional Generative Adversarial Nets \(<https://arxiv.org/abs/1411.1784>\)](https://arxiv.org/abs/1411.1784) (Mirza and Osindero, 2014)

The architecture of your network will contain:

- A generator with a [U-Net \(<https://arxiv.org/abs/1505.04597>\)](https://arxiv.org/abs/1505.04597)-based architecture.
- A discriminator represented by a convolutional PatchGAN classifier (proposed in the [pix2pix paper \(<https://arxiv.org/abs/1611.07004>\)](https://arxiv.org/abs/1611.07004)).

Note that each epoch can take around 15 seconds on a single V100 GPU.

Below are some examples of the output generated by the pix2pix cGAN after training for 200 epochs on the facades dataset (80k steps).



Import TensorFlow and other libraries

In []:

```
import tensorflow as tf

import os
import pathlib
import time
import datetime

from matplotlib import pyplot as plt
from IPython import display
```

Load the dataset

Download the CMP Facade Database data (30MB). Additional datasets are available in the same format [here](#) ([\(<http://efrosgans.eecs.berkeley.edu/pix2pix/datasets/>\)](http://efrosgans.eecs.berkeley.edu/pix2pix/datasets/)). In Colab you can select other datasets from the drop-down menu. Note that some of the other datasets are significantly larger (edges2handbags is 8GB).

```
In [ ]:
```

```
dataset_name = "facades" #@param ["cityscapes", "edges2handbags", "edges2shoes", "facades", "maps", "night2day"]
```

```
In [ ]:
```

```
_URL = f'http://efrosgans.eecs.berkeley.edu/pix2pix/datasets/{dataset_name}.tar.gz'

path_to_zip = tf.keras.utils.get_file(
    fname=f'{dataset_name}.tar.gz',
    origin=_URL,
    extract=True)

path_to_zip = pathlib.Path(path_to_zip)

PATH = path_to_zip.parent/dataset_name
```

```
In [ ]:
```

```
list(PATH.parent.iterdir())
```

Each original image is of size 256 x 512 containing two 256 x 256 images:

```
In [ ]:
```

```
sample_image = tf.io.read_file(str(PATH / 'train/1.jpg'))
sample_image = tf.io.decode_jpeg(sample_image)
print(sample_image.shape)
```

```
In [ ]:
```

```
plt.figure()
plt.imshow(sample_image)
```

You need to separate real building facade images from the architecture label images—all of which will be of size 256 x 256 .

Define a function that loads image files and outputs two image tensors:

```
In [ ]:
```

```
def load(image_file):
    # Read and decode an image file to a uint8 tensor
    image = tf.io.read_file(image_file)
    image = tf.io.decode_jpeg(image)

    # Split each image tensor into two tensors:
    # - one with a real building facade image
    # - one with an architecture label image
    w = tf.shape(image)[1]
    w = w // 2
    input_image = image[:, w:, :]
    real_image = image[:, :w, :]

    # Convert both images to float32 tensors
    input_image = tf.cast(input_image, tf.float32)
    real_image = tf.cast(real_image, tf.float32)

    return input_image, real_image
```

Plot a sample of the input (architecture label image) and real (building facade photo) images:

```
In [ ]:
```

```
inp, re = load(str(PATH / 'train/100.jpg'))
# Casting to int for matplotlib to display the images
plt.figure()
plt.imshow(inp / 255.0)
plt.figure()
plt.imshow(re / 255.0)
```

As described in the [pix2pix paper](https://arxiv.org/abs/1611.07004) (<https://arxiv.org/abs/1611.07004>), you need to apply random jittering and mirroring to preprocess the training set.

Define several functions that:

1. Resize each 256 x 256 image to a larger height and width— 286 x 286 .
2. Randomly crop it back to 256 x 256 .
3. Randomly flip the image horizontally i.e. left to right (random mirroring).
4. Normalize the images to the [-1, 1] range.

In []:

```
# The facade training set consist of 400 images
BUFFER_SIZE = 400
# The batch size of 1 produced better results for the U-Net in the original pix2pix experiment
BATCH_SIZE = 1
# Each image is 256x256 in size
IMG_WIDTH = 256
IMG_HEIGHT = 256
```

In []:

```
def resize(input_image, real_image, height, width):
    input_image = tf.image.resize(input_image, [height, width],
                                  method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)
    real_image = tf.image.resize(real_image, [height, width],
                                 method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)

    return input_image, real_image
```

In []:

```
def random_crop(input_image, real_image):
    stacked_image = tf.stack([input_image, real_image], axis=0)
    cropped_image = tf.image.random_crop(
        stacked_image, size=[2, IMG_HEIGHT, IMG_WIDTH, 3])

    return cropped_image[0], cropped_image[1]
```

In []:

```
# Normalizing the images to [-1, 1]
def normalize(input_image, real_image):
    input_image = (input_image / 127.5) - 1
    real_image = (real_image / 127.5) - 1

    return input_image, real_image
```

In []:

```
@tf.function()
def random_jitter(input_image, real_image):
    # Resizing to 286x286
    input_image, real_image = resize(input_image, real_image, 286, 286)

    # Random cropping back to 256x256
    input_image, real_image = random_crop(input_image, real_image)

    if tf.random.uniform() > 0.5:
        # Random mirroring
        input_image = tf.image.flip_left_right(input_image)
        real_image = tf.image.flip_left_right(real_image)

    return input_image, real_image
```

You can inspect some of the preprocessed output:

In []:

```
plt.figure(figsize=(6, 6))
for i in range(4):
    rj_inp, rj_re = random_jitter(inp, re)
    plt.subplot(2, 2, i + 1)
    plt.imshow(rj_inp / 255.0)
    plt.axis('off')
plt.show()
```

Having checked that the loading and preprocessing works, let's define a couple of helper functions that load and preprocess the training and test sets:

```
In [ ]:
```

```
def load_image_train(image_file):
    input_image, real_image = load(image_file)
    input_image, real_image = random_jitter(input_image, real_image)
    input_image, real_image = normalize(input_image, real_image)

    return input_image, real_image
```

```
In [ ]:
```

```
def load_image_test(image_file):
    input_image, real_image = load(image_file)
    input_image, real_image = resize(input_image, real_image,
                                    IMG_HEIGHT, IMG_WIDTH)
    input_image, real_image = normalize(input_image, real_image)

    return input_image, real_image
```

Build an input pipeline with `tf.data`

```
In [ ]:
```

```
train_dataset = tf.data.Dataset.list_files(str(PATH / 'train/*.jpg'))
train_dataset = train_dataset.map(load_image_train,
                                 num_parallel_calls=tf.data.AUTOTUNE)
train_dataset = train_dataset.shuffle(BUFFER_SIZE)
train_dataset = train_dataset.batch(BATCH_SIZE)
```

```
In [ ]:
```

```
try:
    test_dataset = tf.data.Dataset.list_files(str(PATH / 'test/*.jpg'))
except tf.errors.InvalidArgumentError:
    test_dataset = tf.data.Dataset.list_files(str(PATH / 'val/*.jpg'))
test_dataset = test_dataset.map(load_image_test)
test_dataset = test_dataset.batch(BATCH_SIZE)
```

Build the generator

The generator of your pix2pix cGAN is a *modified U-Net* (<https://arxiv.org/abs/1505.04597>). A U-Net consists of an encoder (downsampler) and decoder (upsampler). (You can find out more about it in the [Image segmentation](https://www.tensorflow.org/tutorials/images/segmentation) (<https://www.tensorflow.org/tutorials/images/segmentation>) tutorial and on the [U-Net project website](https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/) (<https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/>).

- Each block in the encoder is: Convolution -> Batch normalization -> Leaky ReLU
- Each block in the decoder is: Transposed convolution -> Batch normalization -> Dropout (applied to the first 3 blocks) -> ReLU
- There are skip connections between the encoder and decoder (as in the U-Net).

Define the downsampler (encoder):

```
In [ ]:
```

```
OUTPUT_CHANNELS = 3
```

```
In [ ]:
```

```
def downsample(filters, size, apply_batchnorm=True):
    initializer = tf.random_normal_initializer(0., 0.02)

    result = tf.keras.Sequential()
    result.add(
        tf.keras.layers.Conv2D(filters, size, strides=2, padding='same',
                              kernel_initializer=initializer, use_bias=False))

    if apply_batchnorm:
        result.add(tf.keras.layers.BatchNormalization())

    result.add(tf.keras.layers.LeakyReLU())

    return result
```

```
In [ ]:
```

```
down_model = downsample(3, 4)
down_result = down_model(tf.expand_dims(inp, 0))
print (down_result.shape)
```

Define the upsampler (decoder):

In []:

```
def upsample(filters, size, apply_dropout=False):
    initializer = tf.random_normal_initializer(0., 0.02)

    result = tf.keras.Sequential()
    result.add(
        tf.keras.layers.Conv2DTranspose(filters, size, strides=2,
                                       padding='same',
                                       kernel_initializer=initializer,
                                       use_bias=False))

    result.add(tf.keras.layers.BatchNormalization())

    if apply_dropout:
        result.add(tf.keras.layers.Dropout(0.5))

    result.add(tf.keras.layers.ReLU())

    return result
```

In []:

```
up_model = upsample(3, 4)
up_result = up_model(down_result)
print (up_result.shape)
```

Define the generator with the downampler and the upsampler:

In []:

```
def Generator():
    inputs = tf.keras.layers.Input(shape=[256, 256, 3])

    down_stack = [
        downsample(64, 4, apply_batchnorm=False), # (batch_size, 128, 128, 64)
        downsample(128, 4), # (batch_size, 64, 64, 128)
        downsample(256, 4), # (batch_size, 32, 32, 256)
        downsample(512, 4), # (batch_size, 16, 16, 512)
        downsample(512, 4), # (batch_size, 8, 8, 512)
        downsample(512, 4), # (batch_size, 4, 4, 512)
        downsample(512, 4), # (batch_size, 2, 2, 512)
        downsample(512, 4), # (batch_size, 1, 1, 512)
    ]

    up_stack = [
        upsample(512, 4, apply_dropout=True), # (batch_size, 2, 2, 1024)
        upsample(512, 4, apply_dropout=True), # (batch_size, 4, 4, 1024)
        upsample(512, 4, apply_dropout=True), # (batch_size, 8, 8, 1024)
        upsample(512, 4), # (batch_size, 16, 16, 1024)
        upsample(256, 4), # (batch_size, 32, 32, 512)
        upsample(128, 4), # (batch_size, 64, 64, 256)
        upsample(64, 4), # (batch_size, 128, 128, 128)
    ]

    initializer = tf.random_normal_initializer(0., 0.02)
    last = tf.keras.layers.Conv2DTranspose(OUTPUT_CHANNELS, 4,
                                         strides=2,
                                         padding='same',
                                         kernel_initializer=initializer,
                                         activation='tanh') # (batch_size, 256, 256, 3)

    x = inputs

    # Downsampling through the model
    skips = []
    for down in down_stack:
        x = down(x)
        skips.append(x)

    skips = reversed(skips[:-1])

    # Upsampling and establishing the skip connections
    for up, skip in zip(up_stack, skips):
        x = up(x)
        x = tf.keras.layers.concatenate([x, skip])

    x = last(x)

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

Visualize the generator model architecture:

In []:

```
generator = Generator()
tf.keras.utils.plot_model(generator, show_shapes=True, dpi=64)
```

Test the generator:

In []:

```
gen_output = generator(inp[tf.newaxis, ...], training=False)
plt.imshow(gen_output[0, ...])
```

Define the generator loss

GANs learn a loss that adapts to the data, while cGANs learn a structured loss that penalizes a possible structure that differs from the network output and the target image, as described in the [pix2pix paper](https://arxiv.org/abs/1611.07004) (<https://arxiv.org/abs/1611.07004>).

- The generator loss is a sigmoid cross-entropy loss of the generated images and an **array of ones**.
- The pix2pix paper also mentions the L1 loss, which is a MAE (mean absolute error) between the generated image and the target image.
- This allows the generated image to become structurally similar to the target image.
- The formula to calculate the total generator loss is `gan_loss + LAMBDA * l1_loss`, where `LAMBDA = 100`. This value was decided by the authors of the paper.

```
In [ ]:
```

```
LAMBDA = 100
```

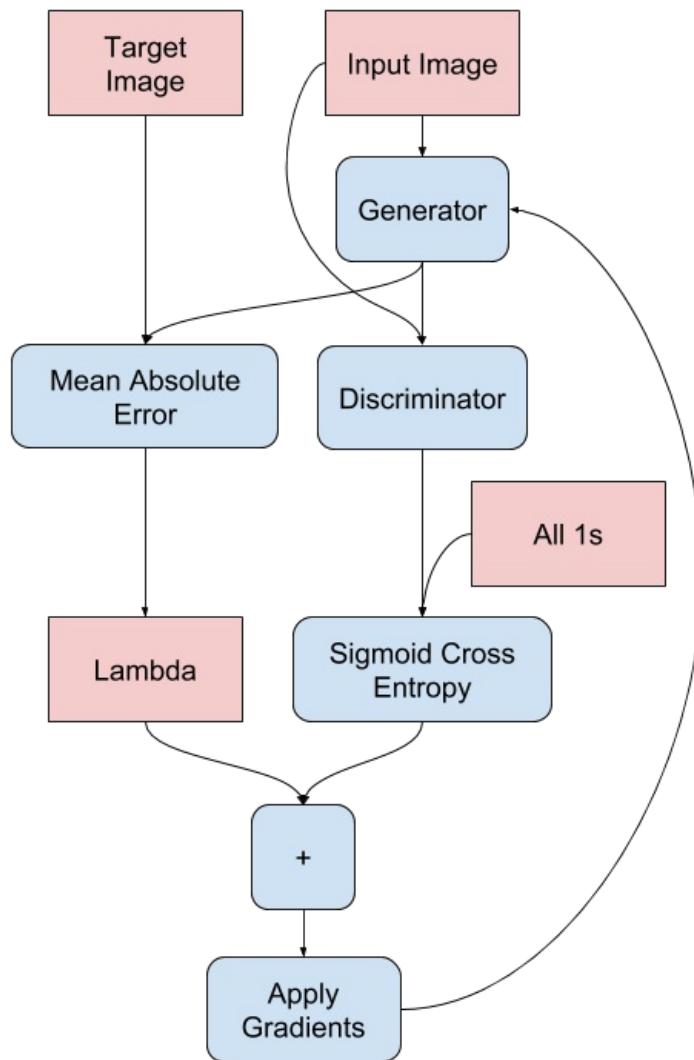
```
In [ ]:
```

```
loss_object = tf.keras.losses.BinaryCrossentropy(from_logits=True)
```

```
In [ ]:
```

```
def generator_loss(disc_generated_output, gen_output, target):  
    gan_loss = loss_object(tf.ones_like(disc_generated_output), disc_generated_output)  
  
    # Mean absolute error  
    l1_loss = tf.reduce_mean(tf.abs(target - gen_output))  
  
    total_gen_loss = gan_loss + (LAMBDA * l1_loss)  
  
    return total_gen_loss, gan_loss, l1_loss
```

The training procedure for the generator is as follows:



Build the discriminator

The discriminator in the pix2pix cGAN is a convolutional PatchGAN classifier—it tries to classify if each image *patch* is real or not real, as described in the [pix2pix paper](https://arxiv.org/abs/1611.07004) (<https://arxiv.org/abs/1611.07004>).

- Each block in the discriminator is: Convolution -> Batch normalization -> Leaky ReLU.
- The shape of the output after the last layer is `(batch_size, 30, 30, 1)`.
- Each 30×30 image patch of the output classifies a 70×70 portion of the input image.
- The discriminator receives 2 inputs:
 - The input image and the target image, which it should classify as real.
 - The input image and the generated image (the output of the generator), which it should classify as fake.
 - Use `tf.concat([inp, tar], axis=-1)` to concatenate these 2 inputs together.

Let's define the discriminator:

In []:

```
def Discriminator():
    initializer = tf.random_normal_initializer(0., 0.02)

    inp = tf.keras.layers.Input(shape=[256, 256, 3], name='input_image')
    tar = tf.keras.layers.Input(shape=[256, 256, 3], name='target_image')

    x = tf.keras.layers.concatenate([inp, tar]) # (batch_size, 256, 256, channels*2)

    down1 = downsample(64, 4, False)(x) # (batch_size, 128, 128, 64)
    down2 = downsample(128, 4)(down1) # (batch_size, 64, 64, 128)
    down3 = downsample(256, 4)(down2) # (batch_size, 32, 32, 256)

    zero_pad1 = tf.keras.layers.ZeroPadding2D()(down3) # (batch_size, 34, 34, 256)
    conv = tf.keras.layers.Conv2D(512, 4, strides=1,
                                kernel_initializer=initializer,
                                use_bias=False)(zero_pad1) # (batch_size, 31, 31, 512)

    batchnorm1 = tf.keras.layers.BatchNormalization()(conv)

    leaky_relu = tf.keras.layers.LeakyReLU()(batchnorm1)

    zero_pad2 = tf.keras.layers.ZeroPadding2D()(leaky_relu) # (batch_size, 33, 33, 512)

    last = tf.keras.layers.Conv2D(1, 4, strides=1,
                                kernel_initializer=initializer)(zero_pad2) # (batch_size, 30, 30, 1)

    return tf.keras.Model(inputs=[inp, tar], outputs=last)
```

Visualize the discriminator model architecture:

In []:

```
discriminator = Discriminator()
tf.keras.utils.plot_model(discriminator, show_shapes=True, dpi=64)
```

Test the discriminator:

In []:

```
disc_out = discriminator([inp[tf.newaxis, ...], gen_output], training=False)
plt.imshow(disc_out[0, ..., -1], vmin=-20, vmax=20, cmap='RdBu_r')
plt.colorbar()
```

Define the discriminator loss

- The `discriminator_loss` function takes 2 inputs: **real images** and **generated images**.
- `real_loss` is a sigmoid cross-entropy loss of the **real images** and an **array of ones**(since these are the **real images**).
- `generated_loss` is a sigmoid cross-entropy loss of the **generated images** and an **array of zeros** (since these are the **fake images**).
- The `total_loss` is the sum of `real_loss` and `generated_loss`.

In []:

```
def discriminator_loss(disc_real_output, disc_generated_output):
    real_loss = loss_object(tf.ones_like(disc_real_output), disc_real_output)

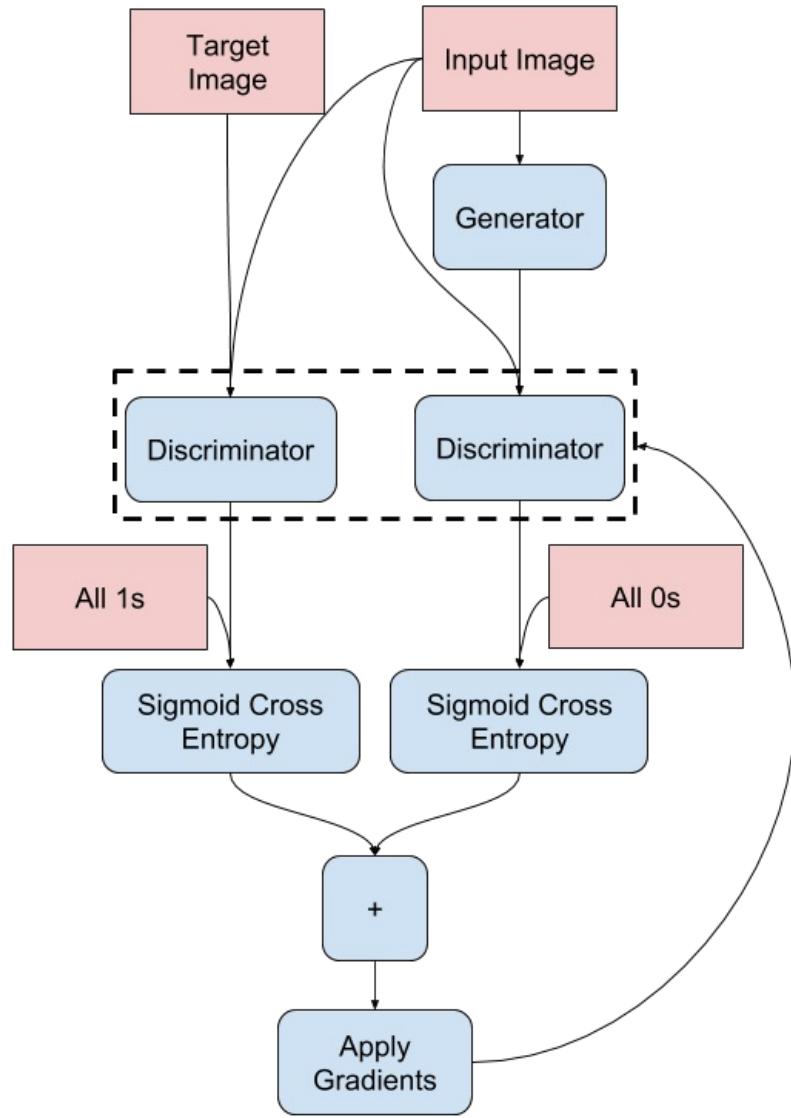
    generated_loss = loss_object(tf.zeros_like(disc_generated_output), disc_generated_output)

    total_disc_loss = real_loss + generated_loss

    return total_disc_loss
```

The training procedure for the discriminator is shown below.

To learn more about the architecture and the hyperparameters you can refer to the [pix2pix paper \(https://arxiv.org/abs/1611.07004\)](https://arxiv.org/abs/1611.07004).



Define the optimizers and a checkpoint-saver

In []:

```
generator_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
discriminator_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
```

In []:

```
checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                                 discriminator_optimizer=discriminator_optimizer,
                                 generator=generator,
                                 discriminator=discriminator)
```

Generate images

Write a function to plot some images during training.

- Pass images from the test set to the generator.
- The generator will then translate the input image into the output.
- The last step is to plot the predictions and *voila!*

Note: The `training=True` is intentional here since you want the batch statistics, while running the model on the test dataset. If you use `training=False`, you get the accumulated statistics learned from the training dataset (which you don't want).

In []:

```
def generate_images(model, test_input, tar):
    prediction = model(test_input, training=True)
    plt.figure(figsize=(15, 15))

    display_list = [test_input[0], tar[0], prediction[0]]
    title = ['Input Image', 'Ground Truth', 'Predicted Image']

    for i in range(3):
        plt.subplot(1, 3, i+1)
        plt.title(title[i])
        # Getting the pixel values in the [0, 1] range to plot.
        plt.imshow(display_list[i] * 0.5 + 0.5)
        plt.axis('off')
    plt.show()
```

Test the function:

In []:

```
for example_input, example_target in test_dataset.take(1):
    generate_images(generator, example_input, example_target)
```

Training

- For each example input generates an output.
- The discriminator receives the `input_image` and the generated image as the first input. The second input is the `input_image` and the `target_image`.
- Next, calculate the generator and the discriminator loss.
- Then, calculate the gradients of loss with respect to both the generator and the discriminator variables(inputs) and apply those to the optimizer.
- Finally, log the losses to TensorBoard.

In []:

```
log_dir="logs/"

summary_writer = tf.summary.create_file_writer(
    log_dir + "fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
```

In []:

```
@tf.function
def train_step(input_image, target, step):
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        gen_output = generator(input_image, training=True)

        disc_real_output = discriminator([input_image, target], training=True)
        disc_generated_output = discriminator([input_image, gen_output], training=True)

        gen_total_loss, gen_gan_loss, gen_l1_loss = generator_loss(disc_generated_output, gen_output, target)
        disc_loss = discriminator_loss(disc_real_output, disc_generated_output)

        generator_gradients = gen_tape.gradient(gen_total_loss,
                                                generator.trainable_variables)
        discriminator_gradients = disc_tape.gradient(disc_loss,
                                                      discriminator.trainable_variables)

        generator_optimizer.apply_gradients(zip(generator_gradients,
                                                generator.trainable_variables))
        discriminator_optimizer.apply_gradients(zip(discriminator_gradients,
                                                   discriminator.trainable_variables))

    with summary_writer.as_default():
        tf.summary.scalar('gen_total_loss', gen_total_loss, step=step//1000)
        tf.summary.scalar('gen_gan_loss', gen_gan_loss, step=step//1000)
        tf.summary.scalar('gen_l1_loss', gen_l1_loss, step=step//1000)
        tf.summary.scalar('disc_loss', disc_loss, step=step//1000)
```

The actual training loop. Since this tutorial can run of more than one dataset, and the datasets vary greatly in size the training loop is setup to work in steps instead of epochs.

- Iterates over the number of steps.
- Every 10 steps print a dot (.).
- Every 1k steps: clear the display and run `generate_images` to show the progress.
- Every 5k steps: save a checkpoint.

```
In [ ]:
```

```
def fit(train_ds, test_ds, steps):
    example_input, example_target = next(iter(test_ds.take(1)))
    start = time.time()

    for step, (input_image, target) in train_ds.repeat().take(steps).enumerate():
        if (step) % 1000 == 0:
            display.clear_output(wait=True)

        if step != 0:
            print(f'Time taken for 1000 steps: {time.time()-start:.2f} sec\n')

        start = time.time()

        generate_images(generator, example_input, example_target)
        print(f'Step: {step//1000}k')

        train_step(input_image, target, step)

    # Training step
    if (step+1) % 10 == 0:
        print('.', end=' ', flush=True)

# Save (checkpoint) the model every 5k steps
if (step + 1) % 5000 == 0:
    checkpoint.save(file_prefix=checkpoint_prefix)
```

This training loop saves logs that you can view in TensorBoard to monitor the training progress.

If you work on a local machine, you would launch a separate TensorBoard process. When working in a notebook, launch the viewer before starting the training to monitor with TensorBoard.

To launch the viewer paste the following into a code-cell:

```
In [ ]:
```

```
%load_ext tensorboard
%tensorboard --logdir {log_dir}
```

Finally, run the training loop:

```
In [ ]:
```

```
fit(train_dataset, test_dataset, steps=40000)
```

If you want to share the TensorBoard results *publicly*, you can upload the logs to [TensorBoard.dev \(https://tensorboard.dev/\)](https://tensorboard.dev/) by copying the following into a code-cell.

Note: This requires a Google account.

```
!tensorboard dev upload --logdir {log_dir}
```

Caution: This command does not terminate. It's designed to continuously upload the results of long-running experiments. Once your data is uploaded you need to stop it using the "interrupt execution" option in your notebook tool.

You can view the [results of a previous run \(https://tensorboard.dev/experiment/lZ0C6FONROaUMfjYkVJqw\)](https://tensorboard.dev/experiment/lZ0C6FONROaUMfjYkVJqw) of this notebook on [TensorBoard.dev \(https://tensorboard.dev/\)](https://tensorboard.dev/).

TensorBoard.dev is a managed experience for hosting, tracking, and sharing ML experiments with everyone.

It can also included inline using an <iframe> :

```
In [ ]:
```

```
display.IFrame(
    src="https://tensorboard.dev/experiment/lZ0C6FONROaUMfjYkVJqw",
    width="100%",
    height="1000px")
```

Interpreting the logs is more subtle when training a GAN (or a cGAN like pix2pix) compared to a simple classification or regression model. Things to look for:

- Check that neither the generator nor the discriminator model has "won". If either the `gen_gan_loss` or the `disc_loss` gets very low, it's an indicator that this model is dominating the other, and you are not successfully training the combined model.
- The value `log(2) = 0.69` is a good reference point for these losses, as it indicates a perplexity of 2 - the discriminator is, on average, equally uncertain about the two options.
- For the `disc_loss`, a value below `0.69` means the discriminator is doing better than random on the combined set of real and generated images.
- For the `gen_gan_loss`, a value below `0.69` means the generator is doing better than random at fooling the discriminator.
- As training progresses, the `gen_l1_loss` should go down.

Restore the latest checkpoint and test the network

In []:

```
!ls {checkpoint_dir}
```

In []:

```
# Restoring the latest checkpoint in checkpoint_dir
checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))
```

Generate some images using the test set

In []:

```
# Run the trained model on a few examples from the test set
for inp, tar in test_dataset.take(5):
    generate_images(generator, inp, tar)
```

Copyright 2020 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Convolutional Variational Autoencoder



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/generative/cvae)

(<https://www.tensorflow.org/tutorials/generative/cvae>)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/generative/cvae.ipynb)

(<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/generative/cvae.ipynb>)

This notebook demonstrates how to train a Variational Autoencoder (VAE) ([1 \(https://arxiv.org/abs/1312.6114\)](https://arxiv.org/abs/1312.6114), [2 \(https://arxiv.org/abs/1401.4082\)](https://arxiv.org/abs/1401.4082)) on the MNIST dataset. A VAE is a probabilistic take on the autoencoder, a model which takes high dimensional input data and compresses it into a smaller representation. Unlike a traditional autoencoder, which maps the input onto a latent vector, a VAE maps the input data into the parameters of a probability distribution, such as the mean and variance of a Gaussian. This approach produces a continuous, structured latent space, which is useful for image generation.



Setup

In []:

```
!pip install tensorflow-probability  
  
# to generate gifs  
!pip install imageio  
!pip install git+https://github.com/tensorflow/docs
```

In [1]:

```
from IPython import display

import glob
import imageio
import matplotlib.pyplot as plt
import numpy as np
import PIL
import tensorflow as tf
import tensorflow_probability as tfp
import time
```

Load the MNIST dataset

Each MNIST image is originally a vector of 784 integers, each of which is between 0-255 and represents the intensity of a pixel. Model each pixel with a Bernoulli distribution in our model, and statically binarize the dataset.

In []:

```
(train_images, _), (test_images, _) = tf.keras.datasets.mnist.load_data()
```

In []:

```
def preprocess_images(images):
    images = images.reshape((images.shape[0], 28, 28, 1)) / 255.
    return np.where(images > .5, 1.0, 0.0).astype('float32')

train_images = preprocess_images(train_images)
test_images = preprocess_images(test_images)
```

In []:

```
train_size = 60000  
batch_size = 32  
test_size = 10000
```

Use `tf.data` to batch and shuffle the data

In []:

```
train_dataset = (tf.data.Dataset.from_tensor_slices(train_images)
                 .shuffle(train_size).batch(batch_size))
test_dataset = (tf.data.Dataset.from_tensor_slices(test_images)
                .shuffle(test_size).batch(batch_size))
```

Define the encoder and decoder networks with `tf.keras.Sequential`

In this VAE example, use two small ConvNets for the encoder and decoder networks. In the literature, these networks are also referred to as inference/recognition and generative models respectively. Use `tf.keras.Sequential` to simplify implementation. Let x and z denote the observation and latent variable respectively in the following descriptions.

Encoder network

This defines the approximate posterior distribution $q(z|x)$, which takes as input an observation and outputs a set of parameters for specifying the conditional distribution of the latent representation z . In this example, simply model the distribution as a diagonal Gaussian, and the network outputs the mean and log-variance parameters of a factorized Gaussian. Output log-variance instead of the variance directly for numerical stability.

Decoder network

This defines the conditional distribution of the observation $p(x|z)$, which takes a latent sample z as input and outputs the parameters for a conditional distribution of the observation. Model the latent distribution prior $p(z)$ as a unit Gaussian.

Reparameterization trick

To generate a sample z for the decoder during training, you can sample from the latent distribution defined by the parameters outputted by the encoder, given an input observation x . However, this sampling operation creates a bottleneck because backpropagation cannot flow through a random node.

To address this, use a reparameterization trick. In our example, you approximate z using the decoder parameters and another parameter ϵ as follows:

$$z = \mu + \sigma \odot \epsilon$$

where μ and σ represent the mean and standard deviation of a Gaussian distribution respectively. They can be derived from the decoder output. The ϵ can be thought of as a random noise used to maintain stochasticity of z . Generate ϵ from a standard normal distribution.

The latent variable z is now generated by a function of μ , σ and ϵ , which would enable the model to backpropagate gradients in the encoder through μ and σ respectively, while maintaining stochasticity through ϵ .

Network architecture

For the encoder network, use two convolutional layers followed by a fully-connected layer. In the decoder network, mirror this architecture by using a fully-connected layer followed by three convolution transpose layers (a.k.a. deconvolutional layers in some contexts). Note, it's common practice to avoid using batch normalization when training VAEs, since the additional stochasticity due to using mini-batches may aggravate instability on top of the stochasticity from sampling.

In []:

```
class CVAE(tf.keras.Model):
    """Convolutional variational autoencoder."""

    def __init__(self, latent_dim):
        super(CVAE, self).__init__()
        self.latent_dim = latent_dim
        self.encoder = tf.keras.Sequential(
            [
                tf.keras.layers.InputLayer(input_shape=(28, 28, 1)),
                tf.keras.layers.Conv2D(
                    filters=32, kernel_size=3, strides=(2, 2), activation='relu'),
                tf.keras.layers.Conv2D(
                    filters=64, kernel_size=3, strides=(2, 2), activation='relu'),
                tf.keras.layers.Flatten(),
                # No activation
                tf.keras.layers.Dense(latent_dim + latent_dim),
            ]
        )

        self.decoder = tf.keras.Sequential(
            [
                tf.keras.layers.InputLayer(input_shape=(latent_dim,)),
                tf.keras.layers.Dense(units=7*7*32, activation=tf.nn.relu),
                tf.keras.layers.Reshape(target_shape=(7, 7, 32)),
                tf.keras.layers.Conv2DTranspose(
                    filters=64, kernel_size=3, strides=2, padding='same',
                    activation='relu'),
                tf.keras.layers.Conv2DTranspose(
                    filters=32, kernel_size=3, strides=2, padding='same',
                    activation='relu'),
                # No activation
                tf.keras.layers.Conv2DTranspose(
                    filters=1, kernel_size=3, strides=1, padding='same'),
            ]
        )

    @tf.function
    def sample(self, eps=None):
        if eps is None:
            eps = tf.random.normal(shape=(100, self.latent_dim))
        return self.decode(eps, apply_sigmoid=True)

    def encode(self, x):
        mean, logvar = tf.split(self.encoder(x), num_or_size_splits=2, axis=1)
        return mean, logvar

    def reparameterize(self, mean, logvar):
        eps = tf.random.normal(shape=mean.shape)
        return eps * tf.exp(logvar * .5) + mean

    def decode(self, z, apply_sigmoid=False):
        logits = self.decoder(z)
        if apply_sigmoid:
            probs = tf.sigmoid(logits)
            return probs
        return logits
```

Define the loss function and the optimizer

VAEs train by maximizing the evidence lower bound (ELBO) on the marginal log-likelihood:

$$\log p(x) \geq \text{ELBO} = \mathbb{E}_{q(z|x)} \left[\log \frac{p(x,z)}{q(z|x)} \right].$$

In practice, optimize the single sample Monte Carlo estimate of this expectation:

$$\log p(x|z) + \log q(z) - \log q(z|x),$$

where z is sampled from $q(z|x)$.

Note: You could also analytically compute the KL term, but here you incorporate all three terms in the Monte Carlo estimator for simplicity.

In []:

```
optimizer = tf.keras.optimizers.Adam(1e-4)

def log_normal_pdf(sample, mean, logvar, raxis=1):
    log2pi = tf.math.log(2. * np.pi)
    return tf.reduce_sum(
        -.5 * ((sample - mean) ** 2. * tf.exp(-logvar) + logvar + log2pi),
        axis=raxis)

def compute_loss(model, x):
    mean, logvar = model.encode(x)
    z = model.reparameterize(mean, logvar)
    x_logit = model.decode(z)
    cross_ent = tf.nn.sigmoid_cross_entropy_with_logits(logits=x_logit, labels=x)
    logpx_z = -tf.reduce_sum(cross_ent, axis=[1, 2, 3])
    logpz = log_normal_pdf(z, 0., 0.)
    logqz_x = log_normal_pdf(z, mean, logvar)
    return -tf.reduce_mean(logpx_z + logpz - logqz_x)

@tf.function
def train_step(model, x, optimizer):
    """Executes one training step and returns the loss.

    This function computes the loss and gradients, and uses the latter to
    update the model's parameters.
    """
    with tf.GradientTape() as tape:
        loss = compute_loss(model, x)
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
```

Training

- Start by iterating over the dataset
- During each iteration, pass the image to the encoder to obtain a set of mean and log-variance parameters of the approximate posterior $q(z|x)$
- then apply the *reparameterization trick* to sample from $q(z|x)$
- Finally, pass the reparameterized samples to the decoder to obtain the logits of the generative distribution $p(x|z)$
- Note: Since you use the dataset loaded by keras with 60k datapoints in the training set and 10k datapoints in the test set, our resulting ELBO on the test set is slightly higher than reported results in the literature which uses dynamic binarization of Larochelle's MNIST.

Generating images

- After training, it is time to generate some images
- Start by sampling a set of latent vectors from the unit Gaussian prior distribution $p(z)$
- The generator will then convert the latent sample z to logits of the observation, giving a distribution $p(x|z)$
- Here, plot the probabilities of Bernoulli distributions

In []:

```
epochs = 10
# set the dimensionality of the latent space to a plane for visualization later
latent_dim = 2
num_examples_to_generate = 16

# keeping the random vector constant for generation (prediction) so
# it will be easier to see the improvement.
random_vector_for_generation = tf.random.normal(
    shape=[num_examples_to_generate, latent_dim])
model = CVAE(latent_dim)
```

In []:

```
def generate_and_save_images(model, epoch, test_sample):
    mean, logvar = model.encode(test_sample)
    z = model.reparameterize(mean, logvar)
    predictions = model.sample(z)
    fig = plt.figure(figsize=(4, 4))

    for i in range(predictions.shape[0]):
        plt.subplot(4, 4, i + 1)
        plt.imshow(predictions[i, :, :, 0], cmap='gray')
        plt.axis('off')

    # tight_layout minimizes the overlap between 2 sub-plots
    plt.savefig('image_at_epoch_{:04d}.png'.format(epoch))
    plt.show()
```

In []:

```
# Pick a sample of the test set for generating output images
assert batch_size >= num_examples_to_generate
for test_batch in test_dataset.take(1):
    test_sample = test_batch[0:num_examples_to_generate, :, :, :]
```

In []:

```
generate_and_save_images(model, 0, test_sample)

for epoch in range(1, epochs + 1):
    start_time = time.time()
    for train_x in train_dataset:
        train_step(model, train_x, optimizer)
    end_time = time.time()

    loss = tf.keras.metrics.Mean()
    for test_x in test_dataset:
        loss(compute_loss(model, test_x))
    elbo = -loss.result()
    display.clear_output(wait=False)
    print('Epoch: {}, Test set ELBO: {}, time elapse for current epoch: {}'
          .format(epoch, elbo, end_time - start_time))
    generate_and_save_images(model, epoch, test_sample)
```

Display a generated image from the last training epoch

In []:

```
def display_image(epoch_no):
    return PIL.Image.open('image_at_epoch_{:04d}.png'.format(epoch_no))
```

In []:

```
plt.imshow(display_image(epoch))
plt.axis('off') # Display images
```

Display an animated GIF of all the saved images

In []:

```
anim_file = 'cvae.gif'

with imageio.get_writer(anim_file, mode='I') as writer:
    filenames = glob.glob('image*.png')
    filenames = sorted(filenames)
    for filename in filenames:
        image = imageio.imread(filename)
        writer.append_data(image)
    image = imageio.imread(filename)
    writer.append_data(image)
```

In []:

```
import tensorflow_docs.vis.embed as embed
embed.embed_file(anim_file)
```

Display a 2D manifold of digits from the latent space

Running the code below will show a continuous distribution of the different digit classes, with each digit morphing into another across the 2D latent space. Use [TensorFlow Probability](https://www.tensorflow.org/probability) (<https://www.tensorflow.org/probability>) to generate a standard normal distribution for the latent space.

In []:

```
def plot_latent_images(model, n, digit_size=28):
    """Plots n x n digit images decoded from the latent space."""

    norm = tfp.distributions.Normal(0, 1)
    grid_x = norm.quantile(np.linspace(0.05, 0.95, n))
    grid_y = norm.quantile(np.linspace(0.05, 0.95, n))
    image_width = digit_size*n
    image_height = image_width
    image = np.zeros((image_height, image_width))

    for i, yi in enumerate(grid_x):
        for j, xi in enumerate(grid_y):
            z = np.array([[xi, yi]])
            x_decoded = model.sample(z)
            digit = tf.reshape(x_decoded[0], (digit_size, digit_size))
            image[i * digit_size: (i + 1) * digit_size,
                  j * digit_size: (j + 1) * digit_size] = digit.numpy()

    plt.figure(figsize=(10, 10))
    plt.imshow(image, cmap='Greys_r')
    plt.axis('Off')
    plt.show()
```

In []:

```
plot_latent_images(model, 20)
```

Next steps

This tutorial has demonstrated how to implement a convolutional variational autoencoder using TensorFlow.

As a next step, you could try to improve the model output by increasing the network size. For instance, you could try setting the `filter` parameters for each of the `Conv2D` and `Conv2DTranspose` layers to 512. Note that in order to generate the final 2D latent image plot, you would need to keep `latent_dim` to 2. Also, the training time would increase as the network size increases.

You could also try implementing a VAE using a different dataset, such as CIFAR-10.

VAEs can be implemented in several different styles and of varying complexity. You can find additional implementations in the following sources:

- [Variational AutoEncoder \(keras.io\)](https://keras.io/examples/generative/vae/) (<https://keras.io/examples/generative/vae/>)
- [VAE example from "Writing custom layers and models" guide \(tensorflow.org\)](https://www.tensorflow.org/guide/keras/custom_layers_and_models#putting_it_all_together_an_end-to-end_example) (https://www.tensorflow.org/guide/keras/custom_layers_and_models#putting_it_all_together_an_end-to-end_example)
- [TFP Probabilistic Layers: Variational Auto Encoder \(https://www.tensorflow.org/probability/examples/Probabilistic_Layers_VAE\)](https://www.tensorflow.org/probability/examples/Probabilistic_Layers_VAE)

If you'd like to learn more about the details of VAEs, please refer to [An Introduction to Variational Autoencoders](https://arxiv.org/abs/1906.02691) (<https://arxiv.org/abs/1906.02691>).

Copyright 2019 The TensorFlow Authors.

Licensed under the Apache License, Version 2.0 (the "License");

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Adversarial example using FGSM



[View on TensorFlow.org](#)

(https://www.tensorflow.org/tutorials/generative/adversarial_fgsm) (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/generative/adversarial_fgsm)



[Run in Google Colab](#)

This tutorial creates an *adversarial example* using the Fast Gradient Signed Method (FGSM) attack as described in [Explaining and Harnessing Adversarial Examples](#) (<https://arxiv.org/abs/1412.6572>) by Goodfellow *et al.* This was one of the first and most popular attacks to fool a neural network.

What is an adversarial example?

Adversarial examples are specialised inputs created with the purpose of confusing a neural network, resulting in the misclassification of a given input. These notorious inputs are indistinguishable to the human eye, but cause the network to fail to identify the contents of the image. There are several types of such attacks, however, here the focus is on the fast gradient sign method attack, which is a *white box* attack whose goal is to ensure misclassification. A white box attack is where the attacker has complete access to the model being attacked. One of the most famous examples of an adversarial image shown below is taken from the aforementioned paper.



Here, starting with the image of a panda, the attacker adds small perturbations (distortions) to the original image, which results in the model labelling this image as a gibbon, with high confidence. The process of adding these perturbations is explained below.

Fast gradient sign method

The fast gradient sign method works by using the gradients of the neural network to create an adversarial example. For an input image, the method uses the gradients of the loss with respect to the input image to create a new image that maximises the loss. This new image is called the adversarial image. This can be summarised using the following expression:

$$\text{adv_x} = x + \epsilon * \text{sign}(\nabla_x J(\theta, x, y))$$

where

- adv_x : Adversarial image.
- x : Original input image.
- y : Original input label.
- ϵ : Multiplier to ensure the perturbations are small.
- θ : Model parameters.
- J : Loss.

An intriguing property here, is the fact that the gradients are taken with respect to the input image. This is done because the objective is to create an image that maximises the loss. A method to accomplish this is to find how much each pixel in the image contributes to the loss value, and add a perturbation accordingly. This works pretty fast because it is easy to find how each input pixel contributes to the loss by using the chain rule and finding the required gradients. Hence, the gradients are taken with respect to the image. In addition, since the model is no longer being trained (thus the gradient is not taken with respect to the trainable variables, i.e., the model parameters), and so the model parameters remain constant. The only goal is to fool an already trained model.

So let's try and fool a pretrained model. In this tutorial, the model is [MobileNetV2](#)

(https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/applications/MobileNetV2) model, pretrained on [ImageNet](#) (<http://www.image-net.org/>).

In []:

```
import tensorflow as tf
import matplotlib as mpl
import matplotlib.pyplot as plt

mpl.rcParams['figure.figsize'] = (8, 8)
mpl.rcParams['axes.grid'] = False
```

Let's load the pretrained MobileNetV2 model and the ImageNet class names.

In []:

```
pretrained_model = tf.keras.applications.MobileNetV2(include_top=True,
                                                       weights='imagenet')
pretrained_model.trainable = False

# ImageNet labels
decode_predictions = tf.keras.applications.mobilenet_v2.decode_predictions
```

In []:

```
# Helper function to preprocess the image so that it can be inputted in MobileNetV2
def preprocess(image):
    image = tf.cast(image, tf.float32)
    image = tf.image.resize(image, (224, 224))
    image = tf.keras.applications.mobilenet_v2.preprocess_input(image)
    image = image[None, ...]
    return image

# Helper function to extract labels from probability vector
def get_imagenet_label(probs):
    return decode_predictions(probs, top=1)[0][0]
```

Original image

Let's use a sample image of a Labrador Retriever (https://commons.wikimedia.org/wiki/File:YellowLabradorLooking_new.jpg) by Mirko CC-BY-SA 3.0 (<https://creativecommons.org/licenses/by-sa/3.0/>) from Wikimedia Common and create adversarial examples from it. The first step is to preprocess it so that it can be fed as an input to the MobileNetV2 model.

In []:

```
image_path = tf.keras.utils.get_file('YellowLabradorLooking_new.jpg', 'https://storage.googleapis.com/download.tensorflow.org/example_images/YellowLabradorLooking_new.jpg')
image_raw = tf.io.read_file(image_path)
image = tf.image.decode_image(image_raw)

image = preprocess(image)
image_probs = pretrained_model.predict(image)
```

Let's have a look at the image.

In []:

```
plt.figure()
plt.imshow(image[0] * 0.5 + 0.5) # To change [-1, 1] to [0,1]
_, image_class, class_confidence = get_imagenet_label(image_probs)
plt.title('{} : {:.2f}% Confidence'.format(image_class, class_confidence*100))
plt.show()
```

Create the adversarial image

Implementing fast gradient sign method

The first step is to create perturbations which will be used to distort the original image resulting in an adversarial image. As mentioned, for this task, the gradients are taken with respect to the image.

In []:

```
loss_object = tf.keras.losses.CategoricalCrossentropy()

def create_adversarial_pattern(input_image, input_label):
    with tf.GradientTape() as tape:
        tape.watch(input_image)
        prediction = pretrained_model(input_image)
        loss = loss_object(input_label, prediction)

    # Get the gradients of the loss w.r.t to the input image.
    gradient = tape.gradient(loss, input_image)
    # Get the sign of the gradients to create the perturbation
    signed_grad = tf.sign(gradient)
    return signed_grad
```

The resulting perturbations can also be visualised.

In []:

```
# Get the input label of the image.
labrador_retriever_index = 208
label = tf.one_hot(labrador_retriever_index, image_probs.shape[-1])
label = tf.reshape(label, (1, image_probs.shape[-1]))

perturbations = create_adversarial_pattern(image, label)
plt.imshow(perturbations[0] * 0.5 + 0.5); # To change [-1, 1] to [0,1]
```

Let's try this out for different values of epsilon and observe the resultant image. You'll notice that as the value of epsilon is increased, it becomes easier to fool the network. However, this comes as a trade-off which results in the perturbations becoming more identifiable.

In []:

```
def display_images(image, description):
    _, label, confidence = get_imagenet_label(pretrained_model.predict(image))
    plt.figure()
    plt.imshow(image[0]*0.5+0.5)
    plt.title('{} \n {} : {:.2f}% Confidence'.format(description,
                                                       label, confidence*100))
    plt.show()
```

In []:

```
epsilons = [0, 0.01, 0.1, 0.15]
descriptions = [('Epsilon = {:.3f}'.format(eps) if eps else 'Input')
                 for eps in epsilons]

for i, eps in enumerate(epsilons):
    adv_x = image + eps*perturbations
    adv_x = tf.clip_by_value(adv_x, -1, 1)
    display_images(adv_x, descriptions[i])
```

Next steps

Now that you know about adversarial attacks, try this out on different datasets and different architectures. You may also create and train your own model, and then attempt to fool it using the same method. You can also try and see how the confidence in predictions vary as you change epsilon.

Though powerful, the attack shown in this tutorial was just the start of research into adversarial attacks, and there have been multiple papers creating more powerful attacks since then. In addition to adversarial attacks, research has also led to the creation of defenses, which aims at creating robust machine learning models. You may review this [survey paper \(<https://arxiv.org/abs/1810.00069>\)](https://arxiv.org/abs/1810.00069) for a comprehensive list of adversarial attacks and defences.

For many more implementations of adversarial attacks and defenses, you may want to see the adversarial example library [CleverHans](https://github.com/tensorflow/cleverhans) (<https://github.com/tensorflow/cleverhans>).

Copyright 2019 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

DeepDream



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/generative/deepdream)

(<https://www.tensorflow.org/tutorials/generative/deepdream>)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/generative/deepdream.ipynb)

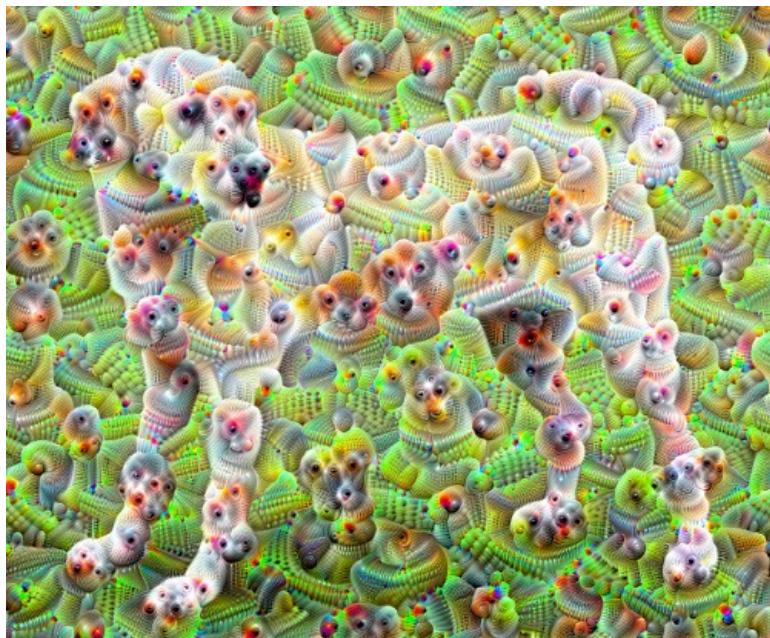
(<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/generative/deepdream.ipynb>)

This tutorial contains a minimal implementation of DeepDream, as described in this [blog post](https://ai.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html) (<https://ai.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html>) by Alexander Mordvintsev.

DeepDream is an experiment that visualizes the patterns learned by a neural network. Similar to when a child watches clouds and tries to interpret random shapes, DeepDream over-interprets and enhances the patterns it sees in an image.

It does so by forwarding an image through the network, then calculating the gradient of the image with respect to the activations of a particular layer. The image is then modified to increase these activations, enhancing the patterns seen by the network, and resulting in a dream-like image. This process was dubbed "Inceptionism" (a reference to [InceptionNet](https://arxiv.org/pdf/1409.4842.pdf) (<https://arxiv.org/pdf/1409.4842.pdf>), and the [movie](https://en.wikipedia.org/wiki/Inception) (<https://en.wikipedia.org/wiki/Inception>) Inception).

Let's demonstrate how you can make a neural network "dream" and enhance the surreal patterns it sees in an image.



In []:

```
import tensorflow as tf
```

In []:

```
import numpy as np
import matplotlib as mpl
import IPython.display as display
import PIL.Image
```

Choose an image to dream-ify

For this tutorial, let's use an image of a [labrador](https://commons.wikimedia.org/wiki/File:YellowLabradorLooking_new.jpg) (https://commons.wikimedia.org/wiki/File:YellowLabradorLooking_new.jpg).

In []:

```
url = 'https://storage.googleapis.com/download.tensorflow.org/example_images/YellowLabradorLooking_new.jpg'
```

In []:

```
# Download an image and read it into a NumPy array.
def download(url, max_dim=None):
    name = url.split('/')[-1]
    image_path = tf.keras.utils.get_file(name, origin=url)
    img = PIL.Image.open(image_path)
    if max_dim:
        img.thumbnail((max_dim, max_dim))
    return np.array(img)

# Normalize an image
def deprocess(img):
    img = 255*(img + 1.0)/2.0
    return tf.cast(img, tf.uint8)

# Display an image
def show(img):
    display.display(PIL.Image.fromarray(np.array(img)))

# Downsizing the image makes it easier to work with.
original_img = download(url, max_dim=500)
show(original_img)
display.display(display.HTML('Image cc-by: <a "href=https://commons.wikimedia.org/wiki/File:Felis_catus-cat_on_snaw.jpg">Von.grzanka</a>'))
```

Prepare the feature extraction model

Download and prepare a pre-trained image classification model. You will use [InceptionV3](https://keras.io/api/applications/inceptionv3/) (<https://keras.io/api/applications/inceptionv3/>) which is similar to the model originally used in DeepDream. Note that any [pre-trained model](https://keras.io/api/applications/#available-models) (<https://keras.io/api/applications/#available-models>) will work, although you will have to adjust the layer names below if you change this.

In []:

```
base_model = tf.keras.applications.InceptionV3(include_top=False, weights='imagenet')
```

The idea in DeepDream is to choose a layer (or layers) and maximize the "loss" in a way that the image increasingly "excites" the layers. The complexity of the features incorporated depends on layers chosen by you, i.e, lower layers produce strokes or simple patterns, while deeper layers give sophisticated features in images, or even whole objects.

The InceptionV3 architecture is quite large (for a graph of the model architecture see TensorFlow's [research repo](https://github.com/tensorflow/models/tree/master/research/slim) (<https://github.com/tensorflow/models/tree/master/research/slim>)). For DeepDream, the layers of interest are those where the convolutions are concatenated. There are 11 of these layers in InceptionV3, named 'mixed0' through 'mixed10'. Using different layers will result in different dream-like images. Deeper layers respond to higher-level features (such as eyes and faces), while earlier layers respond to simpler features (such as edges, shapes, and textures). Feel free to experiment with the layers selected below, but keep in mind that deeper layers (those with a higher index) will take longer to train on since the gradient computation is deeper.

In []:

```
# Maximize the activations of these layers
names = ['mixed3', 'mixed5']
layers = [base_model.get_layer(name).output for name in names]

# Create the feature extraction model
dream_model = tf.keras.Model(inputs=base_model.input, outputs=layers)
```

Calculate loss

The loss is the sum of the activations in the chosen layers. The loss is normalized at each layer so the contribution from larger layers does not outweigh smaller layers. Normally, loss is a quantity you wish to minimize via gradient descent. In DeepDream, you will maximize this loss via gradient ascent.

In []:

```
def calc_loss(img, model):
    # Pass forward the image through the model to retrieve the activations.
    # Converts the image into a batch of size 1.
    img_batch = tf.expand_dims(img, axis=0)
    layer_activations = model(img_batch)
    if len(layer_activations) == 1:
        layer_activations = [layer_activations]

    losses = []
    for act in layer_activations:
        loss = tf.math.reduce_mean(act)
        losses.append(loss)

    return tf.reduce_sum(losses)
```

Gradient ascent

Once you have calculated the loss for the chosen layers, all that is left is to calculate the gradients with respect to the image, and add them to the original image.

Adding the gradients to the image enhances the patterns seen by the network. At each step, you will have created an image that increasingly excites the activations of certain layers in the network.

The method that does this, below, is wrapped in a `tf.function` for performance. It uses an `input_signature` to ensure that the function is not retraced for different image sizes or `steps / step_size` values. See the [Concrete functions guide \(../guide/function.ipynb\)](#) for details.

In []:

```
class DeepDream(tf.Module):
    def __init__(self, model):
        self.model = model

    @tf.function(
        input_signature=(
            tf.TensorSpec(shape=[None, None, 3], dtype=tf.float32),
            tf.TensorSpec(shape=[], dtype=tf.int32),
            tf.TensorSpec(shape=[], dtype=tf.float32),))
    def __call__(self, img, steps, step_size):
        print("Tracing")
        loss = tf.constant(0.0)
        for n in tf.range(steps):
            with tf.GradientTape() as tape:
                # This needs gradients relative to `img`
                # `GradientTape` only watches `tf.Variable`s by default
                tape.watch(img)
                loss = calc_loss(img, self.model)

            # Calculate the gradient of the loss with respect to the pixels of the input image.
            gradients = tape.gradient(loss, img)

            # Normalize the gradients.
            gradients /= tf.math.reduce_std(gradients) + 1e-8

            # In gradient ascent, the "loss" is maximized so that the input image increasingly "excites" the layers.
            # You can update the image by directly adding the gradients (because they're the same shape!)
            img = img + gradients*step_size
            img = tf.clip_by_value(img, -1, 1)

        return loss, img
```

In []:

```
deepdream = DeepDream(dream_model)
```

Main Loop

In []:

```
def run_deep_dream_simple(img, steps=100, step_size=0.01):
    # Convert from uint8 to the range expected by the model.
    img = tf.keras.applications.inception_v3.preprocess_input(img)
    img = tf.convert_to_tensor(img)
    step_size = tf.convert_to_tensor(step_size)
    steps_remaining = steps
    step = 0
    while steps_remaining:
        if steps_remaining>100:
            run_steps = tf.constant(100)
        else:
            run_steps = tf.constant(steps_remaining)
        steps_remaining -= run_steps
        step += run_steps

        loss, img = deepdream(img, run_steps, tf.constant(step_size))

        display.clear_output(wait=True)
        show(deprocess(img))
        print ("Step {}, loss {}".format(step, loss))

    result = deprocess(img)
    display.clear_output(wait=True)
    show(result)

return result
```

In []:

```
dream_img = run_deep_dream_simple(img=original_img,
                                   steps=100, step_size=0.01)
```

Taking it up an octave

Pretty good, but there are a few issues with this first attempt:

1. The output is noisy (this could be addressed with a `tf.image.total_variation` loss).
2. The image is low resolution.
3. The patterns appear like they're all happening at the same granularity.

One approach that addresses all these problems is applying gradient ascent at different scales. This will allow patterns generated at smaller scales to be incorporated into patterns at higher scales and filled in with additional detail.

To do this you can perform the previous gradient ascent approach, then increase the size of the image (which is referred to as an octave), and repeat this process for multiple octaves.

In []:

```
import time
start = time.time()

OCTAVE_SCALE = 1.30

img = tf.constant(np.array(original_img))
base_shape = tf.shape(img)[:-1]
float_base_shape = tf.cast(base_shape, tf.float32)

for n in range(-2, 3):
    new_shape = tf.cast(float_base_shape*(OCTAVE_SCALE**n), tf.int32)

    img = tf.image.resize(img, new_shape).numpy()

    img = run_deep_dream_simple(img=img, steps=50, step_size=0.01)

display.clear_output(wait=True)
img = tf.image.resize(img, base_shape)
img = tf.image.convert_image_dtype(img/255.0, dtype=tf.uint8)
show(img)

end = time.time()
end-start
```

Optional: Scaling up with tiles

One thing to consider is that as the image increases in size, so will the time and memory necessary to perform the gradient calculation. The above octave implementation will not work on very large images, or many octaves.

To avoid this issue you can split the image into tiles and compute the gradient for each tile.

Applying random shifts to the image before each tiled computation prevents tile seams from appearing.

Start by implementing the random shift:

In []:

```
def random_roll(img, maxroll):
    # Randomly shift the image to avoid tiled boundaries.
    shift = tf.random.uniform(shape=[2], minval=-maxroll, maxval=maxroll, dtype=tf.int32)
    img_rolled = tf.roll(img, shift=shift, axis=[0,1])
    return shift, img_rolled
```

In []:

```
shift, img_rolled = random_roll(np.array(original_img), 512)
show(img_rolled)
```

Here is a tiled equivalent of the `deeprdream` function defined earlier:

In []:

```
class TiledGradients(tf.Module):
    def __init__(self, model):
        self.model = model

    @tf.function(
        input_signature=(
            tf.TensorSpec(shape=[None, None, 3], dtype=tf.float32),
            tf.TensorSpec(shape=[2], dtype=tf.int32),
            tf.TensorSpec(shape=[], dtype=tf.int32),))
    def __call__(self, img, img_size, tile_size=512):
        shift, img_rolled = random_roll(img, tile_size)

        # Initialize the image gradients to zero.
        gradients = tf.zeros_like(img_rolled)

        # Skip the last tile, unless there's only one tile.
        xs = tf.range(0, img_size[1], tile_size)[:-1]
        if not tf.cast(len(xs), bool):
            xs = tf.constant([0])
        ys = tf.range(0, img_size[0], tile_size)[:-1]
        if not tf.cast(len(ys), bool):
            ys = tf.constant([0])

        for x in xs:
            for y in ys:
                # Calculate the gradients for this tile.
                with tf.GradientTape() as tape:
                    # This needs gradients relative to `img_rolled`.
                    # `GradientTape` only watches `tf.Variable`s by default.
                    tape.watch(img_rolled)

                    # Extract a tile out of the image.
                    img_tile = img_rolled[y:y+tile_size, x:x+tile_size]
                    loss = calc_loss(img_tile, self.model)

                # Update the image gradients for this tile.
                gradients = gradients + tape.gradient(loss, img_rolled)

        # Undo the random shift applied to the image and its gradients.
        gradients = tf.roll(gradients, shift=-shift, axis=[0,1])

        # Normalize the gradients.
        gradients /= tf.math.reduce_std(gradients) + 1e-8

    return gradients
```

In []:

```
get_tiled_gradients = TiledGradients(dream_model)
```

Putting this together gives a scalable, octave-aware deepdream implementation:

In []:

```
def run_deep_dream_with_octaves(img, steps_per_octave=100, step_size=0.01,
                                 octaves=range(-2,3), octave_scale=1.3):
    base_shape = tf.shape(img)
    img = tf.keras.utils.img_to_array(img)
    img = tf.keras.applications.inception_v3.preprocess_input(img)

    initial_shape = img.shape[:-1]
    img = tf.image.resize(img, initial_shape)
    for octave in octaves:
        # Scale the image based on the octave
        new_size = tf.cast(tf.convert_to_tensor(base_shape[:-1]), tf.float32)*(octave_scale**octave)
        new_size = tf.cast(new_size, tf.int32)
        img = tf.image.resize(img, new_size)

        for step in range(steps_per_octave):
            gradients = get_tiled_gradients(img, new_size)
            img = img + gradients*step_size
            img = tf.clip_by_value(img, -1, 1)

            if step % 10 == 0:
                display.clear_output(wait=True)
                show(deprocess(img))
                print ("Octave {}, Step {}".format(octave, step))

    result = deprocess(img)
    return result
```

In []:

```
img = run_deep_dream_with_octaves(img=original_img, step_size=0.01)

display.clear_output(wait=True)
img = tf.image.resize(img, base_shape)
img = tf.image.convert_image_dtype(img/255.0, dtype=tf.uint8)
show(img)
```

Much better! Play around with the number of octaves, octave scale, and activated layers to change how your DeepDream-ed image looks.

Readers might also be interested in [TensorFlow Lucid](https://github.com/tensorflow/lucid) (<https://github.com/tensorflow/lucid>) which expands on ideas introduced in this tutorial to visualize and interpret neural networks.

Copyright 2018 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Neural style transfer



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/generative/style_transfer)

(https://www.tensorflow.org/tutorials/generative/style_transfer)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/generative/style_transfer.ipynb)

(https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/generative/style_transfer.ipynb)

This tutorial uses deep learning to compose one image in the style of another image (ever wish you could paint like Picasso or Van Gogh?). This is known as *neural style transfer* and the technique is outlined in [A Neural Algorithm of Artistic Style](https://arxiv.org/abs/1508.06576) (<https://arxiv.org/abs/1508.06576>) (Gatys et al.).

Note: This tutorial demonstrates the original style-transfer algorithm. It optimizes the image content to a particular style. Modern approaches train a model to generate the stylized image directly (similar to [cyclegan](#) ([cyclegan.ipynb](#))). This approach is much faster (up to 1000x).

For a simple application of style transfer check out this [tutorial](https://www.tensorflow.org/hub/tutorials/tf2_arbitrary_image_stylization) (https://www.tensorflow.org/hub/tutorials/tf2_arbitrary_image_stylization) to learn more about how to use the pretrained [Arbitrary Image Stylization model](https://tfhub.dev/google/magenta/arbitrary-image-stylization-v1-256/2) (<https://tfhub.dev/google/magenta/arbitrary-image-stylization-v1-256/2>) from [TensorFlow Hub](https://tfhub.dev) (<https://tfhub.dev>) or how to use a style transfer model with [TensorFlow Lite](https://www.tensorflow.org/lite/models/style_transfer/overview) (https://www.tensorflow.org/lite/models/style_transfer/overview).

Neural style transfer is an optimization technique used to take two images—a *content* image and a *style reference* image (such as an artwork by a famous painter)—and blend them together so the output image looks like the content image, but “painted” in the style of the style reference image.

This is implemented by optimizing the output image to match the content statistics of the content image and the style statistics of the style reference image. These statistics are extracted from the images using a convolutional network.

For example, let's take an image of this dog and Wassily Kandinsky's Composition 7:



[Yellow Labrador Looking](https://commons.wikimedia.org/wiki/File:YellowLabradorLooking_new.jpg) (https://commons.wikimedia.org/wiki/File:YellowLabradorLooking_new.jpg), from Wikimedia Commons by [Elf](https://en.wikipedia.org/wiki/User:Elf) (<https://en.wikipedia.org/wiki/User:Elf>). License [CC BY-SA 3.0](https://creativecommons.org/licenses/by-sa/3.0/deed.en) (<https://creativecommons.org/licenses/by-sa/3.0/deed.en>)



Now how would it look like if Kandinsky decided to paint the picture of this Dog exclusively with this style? Something like this?



Setup

Import and configure modules

In []:

```
import os
import tensorflow as tf
# Load compressed models from tensorflow_hub
os.environ['TFHUB_MODEL_LOAD_FORMAT'] = 'COMPRESSED'
```

In []:

```
import IPython.display as display

import matplotlib.pyplot as plt
import matplotlib as mpl
mpl.rcParams['figure.figsize'] = (12, 12)
mpl.rcParams['axes.grid'] = False

import numpy as np
import PIL.Image
import time
import functools
```

In []:

```
def tensor_to_image(tensor):
    tensor = tensor*255
    tensor = np.array(tensor, dtype=np.uint8)
    if np.ndim(tensor)>3:
        assert tensor.shape[0] == 1
        tensor = tensor[0]
    return PIL.Image.fromarray(tensor)
```

Download images and choose a style image and a content image:

In []:

```
content_path = tf.keras.utils.get_file('YellowLabradorLooking_new.jpg', 'https://storage.googleapis.com/download.tensorflow.org/example_images/YellowLabradorLooking_new.jpg')
style_path = tf.keras.utils.get_file('kandinsky5.jpg','https://storage.googleapis.com/download.tensorflow.org/example_images/Vassily_Kandinsky%2C_1913_-_Composition_7.jpg')
```

Visualize the input

Define a function to load an image and limit its maximum dimension to 512 pixels.

In []:

```
def load_img(path_to_img):
    max_dim = 512
    img = tf.io.read_file(path_to_img)
    img = tf.image.decode_image(img, channels=3)
    img = tf.image.convert_image_dtype(img, tf.float32)

    shape = tf.cast(tf.shape(img)[:-1], tf.float32)
    long_dim = max(shape)
    scale = max_dim / long_dim

    new_shape = tf.cast(shape * scale, tf.int32)

    img = tf.image.resize(img, new_shape)
    img = img[tf.newaxis, :]
    return img
```

Create a simple function to display an image:

In []:

```
def imshow(image, title=None):
    if len(image.shape) > 3:
        image = tf.squeeze(image, axis=0)

    plt.imshow(image)
    if title:
        plt.title(title)
```

In []:

```
content_image = load_img(content_path)
style_image = load_img(style_path)

plt.subplot(1, 2, 1)
imshow(content_image, 'Content Image')

plt.subplot(1, 2, 2)
imshow(style_image, 'Style Image')
```

Fast Style Transfer using TF-Hub

This tutorial demonstrates the original style-transfer algorithm, which optimizes the image content to a particular style. Before getting into the details, let's see how the [TensorFlow Hub model \(<https://tfhub.dev/google/magenta/arbitrary-image-stylization-v1-256/2>\)](https://tfhub.dev/google/magenta/arbitrary-image-stylization-v1-256/2) does this:

In []:

```
import tensorflow_hub as hub
hub_model = hub.load('https://tfhub.dev/google/magenta/arbitrary-image-stylization-v1-256/2')
stylized_image = hub_model(tf.constant(content_image), tf.constant(style_image))[0]
tensor_to_image(stylized_image)
```

Define content and style representations

Use the intermediate layers of the model to get the *content* and *style* representations of the image. Starting from the network's input layer, the first few layer activations represent low-level features like edges and textures. As you step through the network, the final few layers represent higher-level features —object parts like *wheels* or *eyes*. In this case, you are using the VGG19 network architecture, a pretrained image classification network. These intermediate layers are necessary to define the representation of content and style from the images. For an input image, try to match the corresponding style and content target representations at these intermediate layers.

Load a [VGG19 \(<https://keras.io/api/applications/vgg/#vgg19-function>\)](https://keras.io/api/applications/vgg/#vgg19-function) and test run it on our image to ensure it's used correctly:

In []:

```
x = tf.keras.applications.vgg19.preprocess_input(content_image*255)
x = tf.image.resize(x, (224, 224))
vgg = tf.keras.applications.VGG19(include_top=True, weights='imagenet')
prediction_probabilities = vgg(x)
prediction_probabilities.shape
```

In []:

```
predicted_top_5 = tf.keras.applications.vgg19.decode_predictions(prediction_probabilities.numpy())[0]
[(class_name, prob) for (number, class_name, prob) in predicted_top_5]
```

Now load a VGG19 without the classification head, and list the layer names

In []:

```
vgg = tf.keras.applications.VGG19(include_top=False, weights='imagenet')

print()
for layer in vgg.layers:
    print(layer.name)
```

Choose intermediate layers from the network to represent the style and content of the image:

In []:

```
content_layers = ['block5_conv2']

style_layers = ['block1_conv1',
                'block2_conv1',
                'block3_conv1',
                'block4_conv1',
                'block5_conv1']

num_content_layers = len(content_layers)
num_style_layers = len(style_layers)
```

Intermediate layers for style and content

So why do these intermediate outputs within our pretrained image classification network allow us to define style and content representations?

At a high level, in order for a network to perform image classification (which this network has been trained to do), it must understand the image. This requires taking the raw image as input pixels and building an internal representation that converts the raw image pixels into a complex understanding of the features present within the image.

This is also a reason why convolutional neural networks are able to generalize well: they're able to capture the invariances and defining features within classes (e.g. cats vs. dogs) that are agnostic to background noise and other nuisances. Thus, somewhere between where the raw image is fed into the model and the output classification label, the model serves as a complex feature extractor. By accessing intermediate layers of the model, you're able to describe the content and style of input images.

Build the model

The networks in `tf.keras.applications` are designed so you can easily extract the intermediate layer values using the Keras functional API.

To define a model using the functional API, specify the inputs and outputs:

```
model = Model(inputs, outputs)
```

This following function builds a VGG19 model that returns a list of intermediate layer outputs:

In []:

```
def vgg_layers(layer_names):
    """Creates a vgg model that returns a list of intermediate output values."""
    # Load our model. Load pretrained VGG, trained on imagenet data
    vgg = tf.keras.applications.VGG19(include_top=False, weights='imagenet')
    vgg.trainable = False

    outputs = [vgg.get_layer(name).output for name in layer_names]

    model = tf.keras.Model([vgg.input], outputs)
    return model
```

And to create the model:

In []:

```
style_extractor = vgg_layers(style_layers)
style_outputs = style_extractor(style_image*255)

#Look at the statistics of each layer's output
for name, output in zip(style_layers, style_outputs):
    print(name)
    print("  shape: ", output.numpy().shape)
    print("  min: ", output.numpy().min())
    print("  max: ", output.numpy().max())
    print("  mean: ", output.numpy().mean())
    print()
```

Calculate style

The content of an image is represented by the values of the intermediate feature maps.

It turns out, the style of an image can be described by the means and correlations across the different feature maps. Calculate a Gram matrix that includes this information by taking the outer product of the feature vector with itself at each location, and averaging that outer product over all locations. This Gram matrix can be calculated for a particular layer as:

$$G_{cd}^l = \frac{\sum_{ij} F_{ijc}^l(x) F_{ijd}^l(x)}{IJ}$$

This can be implemented concisely using the `tf.linalg.einsum` function:

In []:

```
def gram_matrix(input_tensor):
    result = tf.linalg.einsum('bijc,bijd->bcd', input_tensor, input_tensor)
    input_shape = tf.shape(input_tensor)
    num_locations = tf.cast(input_shape[1]*input_shape[2], tf.float32)
    return result/(num_locations)
```

Extract style and content

Build a model that returns the style and content tensors.

In []:

```
class StyleContentModel(tf.keras.models.Model):
    def __init__(self, style_layers, content_layers):
        super(StyleContentModel, self).__init__()
        self.vgg = vgg_layers(style_layers + content_layers)
        self.style_layers = style_layers
        self.content_layers = content_layers
        self.num_style_layers = len(style_layers)
        self.vgg.trainable = False

    def call(self, inputs):
        "Expects float input in [0,1]"
        inputs = inputs*255.0
        preprocessed_input = tf.keras.applications.vgg19.preprocess_input(inputs)
        outputs = self.vgg(preprocessed_input)
        style_outputs, content_outputs = (outputs[:self.num_style_layers],
                                          outputs[self.num_style_layers:])

        style_outputs = [gram_matrix(style_output)
                        for style_output in style_outputs]

        content_dict = {content_name: value
                        for content_name, value
                        in zip(self.content_layers, content_outputs)}

        style_dict = {style_name: value
                      for style_name, value
                      in zip(self.style_layers, style_outputs)}

        return {'content': content_dict, 'style': style_dict}
```

When called on an image, this model returns the gram matrix (style) of the `style_layers` and content of the `content_layers`:

In []:

```
extractor = StyleContentModel(style_layers, content_layers)

results = extractor(tf.constant(content_image))

print('Styles:')
for name, output in sorted(results['style'].items()):
    print("  ", name)
    print("    shape: ", output.numpy().shape)
    print("    min: ", output.numpy().min())
    print("    max: ", output.numpy().max())
    print("    mean: ", output.numpy().mean())
    print()

print("Contents:")
for name, output in sorted(results['content'].items()):
    print("  ", name)
    print("    shape: ", output.numpy().shape)
    print("    min: ", output.numpy().min())
    print("    max: ", output.numpy().max())
    print("    mean: ", output.numpy().mean())
```

Run gradient descent

With this style and content extractor, you can now implement the style transfer algorithm. Do this by calculating the mean square error for your image's output relative to each target, then take the weighted sum of these losses.

Set your style and content target values:

In []:

```
style_targets = extractor(style_image)['style']
content_targets = extractor(content_image)['content']
```

Define a `tf.Variable` to contain the image to optimize. To make this quick, initialize it with the content image (the `tf.Variable` must be the same shape as the content image):

In []:

```
image = tf.Variable(content_image)
```

Since this is a float image, define a function to keep the pixel values between 0 and 1:

In []:

```
def clip_0_1(image):
    return tf.clip_by_value(image, clip_value_min=0.0, clip_value_max=1.0)
```

Create an optimizer. The paper recommends LBFGS, but `Adam` works okay, too:

In []:

```
opt = tf.optimizers.Adam(learning_rate=0.02, beta_1=0.99, epsilon=1e-1)
```

To optimize this, use a weighted combination of the two losses to get the total loss:

In []:

```
style_weight=1e-2
content_weight=1e4
```

In []:

```
def style_content_loss(outputs):
    style_outputs = outputs['style']
    content_outputs = outputs['content']
    style_loss = tf.add_n([tf.reduce_mean((style_outputs[name]-style_targets[name])**2)
                           for name in style_outputs.keys()])
    style_loss *= style_weight / num_style_layers

    content_loss = tf.add_n([tf.reduce_mean((content_outputs[name]-content_targets[name])**2)
                           for name in content_outputs.keys()])
    content_loss *= content_weight / num_content_layers
    loss = style_loss + content_loss
    return loss
```

```
Use tf.GradientTape to update the image.
```

```
In [ ]:
```

```
@tf.function()
def train_step(image):
    with tf.GradientTape() as tape:
        outputs = extractor(image)
        loss = style_content_loss(outputs)

    grad = tape.gradient(loss, image)
    opt.apply_gradients([(grad, image)])
    image.assign(clip_0_1(image))
```

Now run a few steps to test:

```
In [ ]:
```

```
train_step(image)
train_step(image)
train_step(image)
tensor_to_image(image)
```

Since it's working, perform a longer optimization:

```
In [ ]:
```

```
import time
start = time.time()

epochs = 10
steps_per_epoch = 100

step = 0
for n in range(epochs):
    for m in range(steps_per_epoch):
        step += 1
        train_step(image)
        print(".", end='', flush=True)
        display.clear_output(wait=True)
        display.display(tensor_to_image(image))
        print("Train step: {}".format(step))

end = time.time()
print("Total time: {:.1f}".format(end-start))
```

Total variation loss

One downside to this basic implementation is that it produces a lot of high frequency artifacts. Decrease these using an explicit regularization term on the high frequency components of the image. In style transfer, this is often called the *total variation loss*:

```
In [ ]:
```

```
def high_pass_x_y(image):
    x_var = image[:, :, 1:, :] - image[:, :, :-1, :]
    y_var = image[:, 1:, :, :] - image[:, :-1, :, :]

    return x_var, y_var
```

```
In [ ]:
```

```
x_deltas, y_deltas = high_pass_x_y(content_image)

plt.figure(figsize=(14, 10))
plt.subplot(2, 2, 1)
imshow(clip_0_1(2*y_deltas+0.5), "Horizontal Deltas: Original")

plt.subplot(2, 2, 2)
imshow(clip_0_1(2*x_deltas+0.5), "Vertical Deltas: Original")

x_deltas, y_deltas = high_pass_x_y(image)

plt.subplot(2, 2, 3)
imshow(clip_0_1(2*y_deltas+0.5), "Horizontal Deltas: Styled")

plt.subplot(2, 2, 4)
imshow(clip_0_1(2*x_deltas+0.5), "Vertical Deltas: Styled")
```

This shows how the high frequency components have increased.

Also, this high frequency component is basically an edge-detector. You can get similar output from the Sobel edge detector, for example:

In []:

```
plt.figure(figsize=(14, 10))

sobel = tf.image.sobel_edges(content_image)
plt.subplot(1, 2, 1)
imshow(clip_0_1(sobel[..., 0]/4+0.5), "Horizontal Sobel-edges")
plt.subplot(1, 2, 2)
imshow(clip_0_1(sobel[..., 1]/4+0.5), "Vertical Sobel-edges")
```

The regularization loss associated with this is the sum of the squares of the values:

In []:

```
def total_variation_loss(image):
    x_deltas, y_deltas = high_pass_x_y(image)
    return tf.reduce_sum(tf.abs(x_deltas)) + tf.reduce_sum(tf.abs(y_deltas))
```

In []:

```
total_variation_loss(image).numpy()
```

That demonstrated what it does. But there's no need to implement it yourself, TensorFlow includes a standard implementation:

In []:

```
tf.image.total_variation(image).numpy()
```

Re-run the optimization

Choose a weight for the `total_variation_loss`:

In []:

```
total_variation_weight=30
```

Now include it in the `train_step` function:

In []:

```
@tf.function()
def train_step(image):
    with tf.GradientTape() as tape:
        outputs = extractor(image)
        loss = style_content_loss(outputs)
        loss += total_variation_weight*tf.image.total_variation(image)

    grad = tape.gradient(loss, image)
    opt.apply_gradients([(grad, image)])
    image.assign(clip_0_1(image))
```

Reinitialize the optimization variable:

In []:

```
image = tf.Variable(content_image)
```

And run the optimization:

In []:

```
import time
start = time.time()

epochs = 10
steps_per_epoch = 100

step = 0
for n in range(epochs):
    for m in range(steps_per_epoch):
        step += 1
        train_step(image)
        print(".", end='', flush=True)
    display.clear_output(wait=True)
    display.display(tensor_to_image(image))
    print("Train step: {}".format(step))

end = time.time()
print("Total time: {:.1f}".format(end-start))
```

Finally, save the result:

In []:

```
file_name = 'stylized-image.png'
tensor_to_image(image).save(file_name)

try:
    from google.colab import files
except ImportError:
    pass
else:
    files.download(file_name)
```

Learn more

This tutorial demonstrates the original style-transfer algorithm. For a simple application of style transfer check out this [tutorial](#) (https://www.tensorflow.org/hub/tutorials/tf2_arbitrary_image_stylization) to learn more about how to use the arbitrary image style transfer model from TensorFlow Hub (<https://tfhub.dev>).

Copyright 2018 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Image captioning with visual attention



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/text/image_captioning)

(https://www.tensorflow.org/tutorials/text/image_captioning)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/text/image_captioning.ipynb)

(https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/text/image_captioning.ipynb)

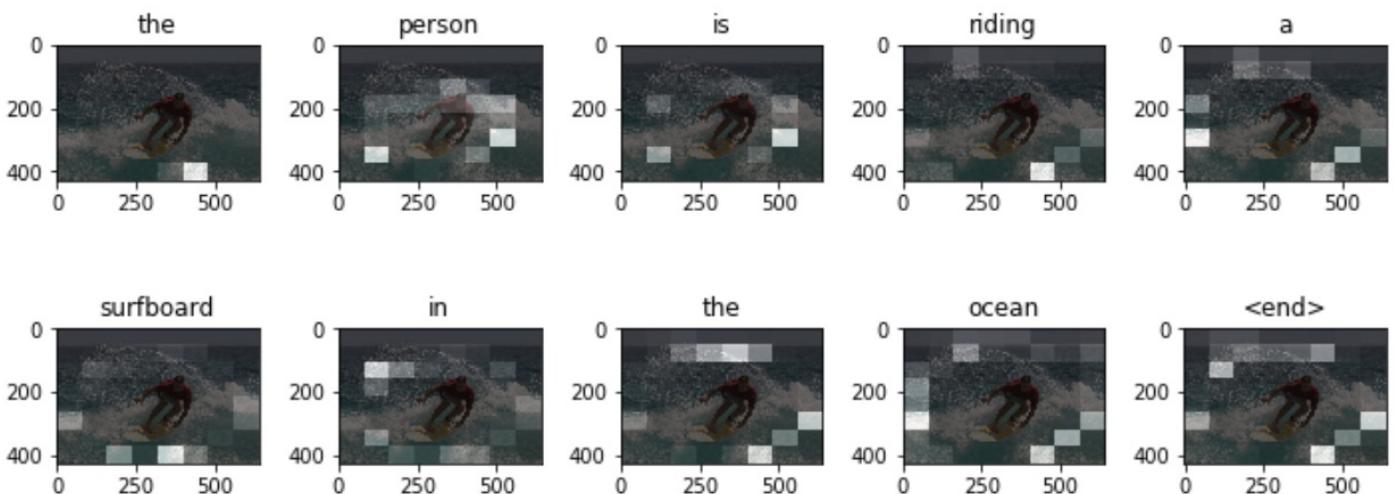
Given an image like the example below, your goal is to generate a caption such as "a surfer riding on a wave".



[Image Source \(\[https://commons.wikimedia.org/wiki/Surfing#/media/File:Surfing_in_Hawaii.jpg\]\(https://commons.wikimedia.org/wiki/Surfing#/media/File:Surfing_in_Hawaii.jpg\)\)](https://commons.wikimedia.org/wiki/Surfing#/media/File:Surfing_in_Hawaii.jpg); License: Public Domain

To accomplish this, you'll use an attention-based model, which enables us to see what parts of the image the model focuses on as it generates a caption.

Prediction Caption: the person is riding a surfboard in the ocean <end>



The model architecture is similar to [Show, Attend and Tell: Neural Image Caption Generation with Visual Attention \(<https://arxiv.org/abs/1502.03044>\)](https://arxiv.org/abs/1502.03044).

This notebook is an end-to-end example. When you run the notebook, it downloads the [MS-COCO \(<http://cocodataset.org/#home>\)](http://cocodataset.org/#home) dataset, preprocesses and caches a subset of images using Inception V3, trains an encoder-decoder model, and generates captions on new images using the trained model.

In this example, you will train a model on a relatively small amount of data—the first 30,000 captions for about 20,000 images (because there are multiple captions per image in the dataset).

In []:

```
import tensorflow as tf

# You'll generate plots of attention in order to see which parts of an image
# your model focuses on during captioning
import matplotlib.pyplot as plt

import collections
import random
import numpy as np
import os
import time
import json
from PIL import Image
```

Download and prepare the MS-COCO dataset

You will use the [MS-COCO dataset \(http://cocodataset.org/#home\)](http://cocodataset.org/#home) to train your model. The dataset contains over 82,000 images, each of which has at least 5 different caption annotations. The code below downloads and extracts the dataset automatically.

Caution: large download ahead. You'll use the training set, which is a 13GB file.

In []:

```
# Download caption annotation files
annotation_folder = '/annotations/'
if not os.path.exists(os.path.abspath('.') + annotation_folder):
    annotation_zip = tf.keras.utils.get_file('captions.zip',
                                              cache_subdir=os.path.abspath('.'),
                                              origin='http://images.cocodataset.org/annotations/annotations_trainval2014.zip',
                                              extract=True)
    annotation_file = os.path.dirname(annotation_zip)+'/annotations/captions_train2014.json'
    os.remove(annotation_zip)

# Download image files
image_folder = '/train2014/'
if not os.path.exists(os.path.abspath('.') + image_folder):
    image_zip = tf.keras.utils.get_file('train2014.zip',
                                         cache_subdir=os.path.abspath('.'),
                                         origin='http://images.cocodataset.org/zips/train2014.zip',
                                         extract=True)
    PATH = os.path.dirname(image_zip) + image_folder
    os.remove(image_zip)
else:
    PATH = os.path.abspath('.') + image_folder
```

Optional: limit the size of the training set

To speed up training for this tutorial, you'll use a subset of 30,000 captions and their corresponding images to train your model. Choosing to use more data would result in improved captioning quality.

In []:

```
with open(annotation_file, 'r') as f:
    annotations = json.load(f)
```

In []:

```
# Group all captions together having the same image ID.
image_path_to_caption = collections.defaultdict(list)
for val in annotations['annotations']:
    caption = f"<start> {val['caption']} <end>"
    image_path = PATH + 'COCO_train2014_' + '%012d.jpg' % (val['image_id'])
    image_path_to_caption[image_path].append(caption)
```

In []:

```
image_paths = list(image_path_to_caption.keys())
random.shuffle(image_paths)

# Select the first 6000 image_paths from the shuffled set.
# Approximately each image id has 5 captions associated with it, so that will
# lead to 30,000 examples.
train_image_paths = image_paths[:6000]
print(len(train_image_paths))
```

In []:

```
train_captions = []
img_name_vector = []

for image_path in train_image_paths:
    caption_list = image_path_to_caption[image_path]
    train_captions.extend(caption_list)
    img_name_vector.extend([image_path] * len(caption_list))
```

In []:

```
print(train_captions[0])
Image.open(img_name_vector[0])
```

Preprocess the images using InceptionV3

Next, you will use InceptionV3 (which is pretrained on Imagenet) to classify each image. You will extract features from the last convolutional layer.

First, you will convert the images into InceptionV3's expected format by:

- Resizing the image to 299px by 299px
- [Preprocess the images \(https://cloud.google.com/tpu/docs/inception-v3-advanced#preprocessing_stage\)](https://cloud.google.com/tpu/docs/inception-v3-advanced#preprocessing_stage) using the [preprocess_input \(https://www.tensorflow.org/api_docs/python/tf/keras/applications/inception_v3/preprocess_input\)](https://www.tensorflow.org/api_docs/python/tf/keras/applications/inception_v3/preprocess_input) method to normalize the image so that it contains pixels in the range of -1 to 1, which matches the format of the images used to train InceptionV3.

In []:

```
def load_image(image_path):  
    img = tf.io.read_file(image_path)  
    img = tf.io.decode_jpeg(img, channels=3)  
    img = tf.keras.layers.Resizing(299, 299)(img)  
    img = tf.keras.applications.inception_v3.preprocess_input(img)  
    return img, image_path
```

Initialize InceptionV3 and load the pretrained Imagenet weights

Now you'll create a tf.keras model where the output layer is the last convolutional layer in the InceptionV3 architecture. The shape of the output of this layer is $8 \times 8 \times 2048$. You use the last convolutional layer because you are using attention in this example. You don't perform this initialization during training because it could become a bottleneck.

- You forward each image through the network and store the resulting vector in a dictionary (image_name --> feature_vector).
- After all the images are passed through the network, you save the dictionary to disk.

In []:

```
image_model = tf.keras.applications.InceptionV3(include_top=False,  
                                                weights='imagenet')  
new_input = image_model.input  
hidden_layer = image_model.layers[-1].output  
  
image_features_extract_model = tf.keras.Model(new_input, hidden_layer)
```

Caching the features extracted from InceptionV3

You will pre-process each image with InceptionV3 and cache the output to disk. Caching the output in RAM would be faster but also memory intensive, requiring $8 \times 8 \times 2048$ floats per image. At the time of writing, this exceeds the memory limitations of Colab (currently 12GB of memory).

Performance could be improved with a more sophisticated caching strategy (for example, by sharding the images to reduce random access disk I/O), but that would require more code.

The caching will take about 10 minutes to run in Colab with a GPU. If you'd like to see a progress bar, you can:

1. Install [tqdm \(https://github.com/tqdm/tqdm\)](https://github.com/tqdm/tqdm):

```
!pip install tqdm
```

2. Import tqdm:

```
from tqdm import tqdm
```

3. Change the following line:

```
for img, path in image_dataset:
```

to:

```
for img, path in tqdm(image_dataset):
```

In []:

```
# Get unique images
encode_train = sorted(set(img_name_vector))

# Feel free to change batch_size according to your system configuration
image_dataset = tf.data.Dataset.from_tensor_slices(encode_train)
image_dataset = image_dataset.map(
    load_image, num_parallel_calls=tf.data.AUTOTUNE).batch(16)

for img, path in image_dataset:
    batch_features = image_features_extract_model(img)
    batch_features = tf.reshape(batch_features,
                                (batch_features.shape[0], -1, batch_features.shape[3]))

    for bf, p in zip(batch_features, path):
        path_of_feature = p.numpy().decode("utf-8")
        np.save(path_of_feature, bf.numpy())
```

Preprocess and tokenize the captions

You will transform the text captions into integer sequences using the [TextVectorization](#) (https://www.tensorflow.org/api_docs/python/tf/keras/layers/TextVectorization) layer, with the following steps:

- Use [adapt](#) (https://www.tensorflow.org/api_docs/python/tf/keras/layers/TextVectorization#adapt) to iterate over all captions, split the captions into words, and compute a vocabulary of the top 5,000 words (to save memory).
- Tokenize all captions by mapping each word to its index in the vocabulary. All output sequences will be padded to length 50.
- Create word-to-index and index-to-word mappings to display results.

In []:

```
caption_dataset = tf.data.Dataset.from_tensor_slices(train_captions)

# We will override the default standardization of TextVectorization to preserve
# "<>" characters, so we preserve the tokens for the <start> and <end>.
def standardize(inputs):
    inputs = tf.strings.lower(inputs)
    return tf.strings.regex_replace(inputs,
                                    r"!\"#$%&\(\)\*\+\.,-/;=?@[\\\\]\^`{|}~", "")

# Max word count for a caption.
max_length = 50
# Use the top 5000 words for a vocabulary.
vocabulary_size = 5000
tokenizer = tf.keras.layers.TextVectorization(
    max_tokens=vocabulary_size,
    standardize=standardize,
    output_sequence_length=max_length)
# Learn the vocabulary from the caption data.
tokenizer.adapt(caption_dataset)
```

In []:

```
# Create the tokenized vectors
cap_vector = caption_dataset.map(lambda x: tokenizer(x))
```

In []:

```
# Create mappings for words to indices and indices to words.
word_to_index = tf.keras.layers.StringLookup(
    mask_token="",
    vocabulary=tokenizer.get_vocabulary())
index_to_word = tf.keras.layers.StringLookup(
    mask_token="",
    vocabulary=tokenizer.get_vocabulary(),
    invert=True)
```

Split the data into training and testing

In []:

```
img_to_cap_vector = collections.defaultdict(list)
for img, cap in zip(img_name_vector, cap_vector):
    img_to_cap_vector[img].append(cap)

# Create training and validation sets using an 80-20 split randomly.
img_keys = list(img_to_cap_vector.keys())
random.shuffle(img_keys)

slice_index = int(len(img_keys)*0.8)
img_name_train_keys, img_name_val_keys = img_keys[:slice_index], img_keys[slice_index:]

img_name_train = []
cap_train = []
for imgt in img_name_train_keys:
    capt_len = len(img_to_cap_vector[imgt])
    img_name_train.extend([imgt] * capt_len)
    cap_train.extend(img_to_cap_vector[imgt])

img_name_val = []
cap_val = []
for imgv in img_name_val_keys:
    capv_len = len(img_to_cap_vector[imgv])
    img_name_val.extend([imgv] * capv_len)
    cap_val.extend(img_to_cap_vector[imgv])
```

In []:

```
len(img_name_train), len(cap_train), len(img_name_val), len(cap_val)
```

Create a tf.data dataset for training

Your images and captions are ready! Next, let's create a `tf.data` dataset to use for training your model.

In []:

```
# Feel free to change these parameters according to your system's configuration

BATCH_SIZE = 64
BUFFER_SIZE = 1000
embedding_dim = 256
units = 512
num_steps = len(img_name_train) // BATCH_SIZE
# Shape of the vector extracted from InceptionV3 is (64, 2048)
# These two variables represent that vector shape
features_shape = 2048
attention_features_shape = 64
```

In []:

```
# Load the numpy files
def map_func(img_name, cap):
    img_tensor = np.load(img_name.decode('utf-8')+'.npy')
    return img_tensor, cap
```

In []:

```
dataset = tf.data.Dataset.from_tensor_slices((img_name_train, cap_train))

# Use map to load the numpy files in parallel
dataset = dataset.map(lambda item1, item2: tf.numpy_function(
    map_func, [item1, item2], [tf.float32, tf.int64]),
    num_parallel_calls=tf.data.AUTOTUNE)

# Shuffle and batch
dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
dataset = dataset.prefetch(buffer_size=tf.data.AUTOTUNE)
```

Model

Fun fact: the decoder below is identical to the one in the example for [Neural Machine Translation with Attention](https://www.tensorflow.org/text/tutorials/nmt_with_attention) (https://www.tensorflow.org/text/tutorials/nmt_with_attention).

The model architecture is inspired by the [Show, Attend and Tell](https://arxiv.org/pdf/1502.03044.pdf) (<https://arxiv.org/pdf/1502.03044.pdf>) paper.

- In this example, you extract the features from the lower convolutional layer of InceptionV3 giving us a vector of shape (8, 8, 2048).
- You squash that to a shape of (64, 2048).
- This vector is then passed through the CNN Encoder (which consists of a single Fully connected layer).
- The RNN (here GRU) attends over the image to predict the next word.

In []:

```
class BahdanauAttention(tf.keras.Model):
    def __init__(self, units):
        super(BahdanauAttention, self).__init__()
        self.W1 = tf.keras.layers.Dense(units)
        self.W2 = tf.keras.layers.Dense(units)
        self.V = tf.keras.layers.Dense(1)

    def call(self, features, hidden):
        # features(CNN_encoder output) shape == (batch_size, 64, embedding_dim)

        # hidden shape == (batch_size, hidden_size)
        # hidden_with_time_axis shape == (batch_size, 1, hidden_size)
        hidden_with_time_axis = tf.expand_dims(hidden, 1)

        # attention_hidden_layer shape == (batch_size, 64, units)
        attention_hidden_layer = (tf.nn.tanh(self.W1(features) +
                                              self.W2(hidden_with_time_axis)))

        # score shape == (batch_size, 64, 1)
        # This gives you an unnormalized score for each image feature.
        score = self.V(attention_hidden_layer)

        # attention_weights shape == (batch_size, 64, 1)
        attention_weights = tf.nn.softmax(score, axis=1)

        # context_vector shape after sum == (batch_size, hidden_size)
        context_vector = attention_weights * features
        context_vector = tf.reduce_sum(context_vector, axis=1)

        return context_vector, attention_weights
```

In []:

```
class CNN_Encoder(tf.keras.Model):
    # Since you have already extracted the features and dumped it
    # This encoder passes those features through a Fully connected layer
    def __init__(self, embedding_dim):
        super(CNN_Encoder, self).__init__()
        # shape after fc == (batch_size, 64, embedding_dim)
        self.fc = tf.keras.layers.Dense(embedding_dim)

    def call(self, x):
        x = self.fc(x)
        x = tf.nn.relu(x)
        return x
```

In []:

```
class RNN_Decoder(tf.keras.Model):
    def __init__(self, embedding_dim, units, vocab_size):
        super(RNN_Decoder, self).__init__()
        self.units = units

        self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
        self.gru = tf.keras.layers.GRU(self.units,
                                      return_sequences=True,
                                      return_state=True,
                                      recurrent_initializer='glorot_uniform')
        self.fc1 = tf.keras.layers.Dense(self.units)
        self.fc2 = tf.keras.layers.Dense(vocab_size)

        self.attention = BahdanauAttention(self.units)

    def call(self, x, features, hidden):
        # defining attention as a separate model
        context_vector, attention_weights = self.attention(features, hidden)

        # x shape after passing through embedding == (batch_size, 1, embedding_dim)
        x = self.embedding(x)

        # x shape after concatenation == (batch_size, 1, embedding_dim + hidden_size)
        x = tf.concat([tf.expand_dims(context_vector, 1), x], axis=-1)

        # passing the concatenated vector to the GRU
        output, state = self.gru(x)

        # shape == (batch_size, max_length, hidden_size)
        x = self.fc1(output)

        # x shape == (batch_size * max_length, hidden_size)
        x = tf.reshape(x, (-1, x.shape[2]))

        # output shape == (batch_size * max_length, vocab)
        x = self.fc2(x)

        return x, state, attention_weights

    def reset_state(self, batch_size):
        return tf.zeros((batch_size, self.units))
```

In []:

```
encoder = CNN_Encoder(embedding_dim)
decoder = RNN_Decoder(embedding_dim, units, tokenizer.vocabulary_size())
```

In []:

```
optimizer = tf.keras.optimizers.Adam()
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True, reduction='none')

def loss_function(real, pred):
    mask = tf.math.logical_not(tf.math.equal(real, 0))
    loss_ = loss_object(real, pred)

    mask = tf.cast(mask, dtype=loss_.dtype)
    loss_*= mask

    return tf.reduce_mean(loss_)
```

Checkpoint

In []:

```
checkpoint_path = "./checkpoints/train"
ckpt = tf.train.Checkpoint(encoder=encoder,
                           decoder=decoder,
                           optimizer=optimizer)
ckpt_manager = tf.train.CheckpointManager(ckpt, checkpoint_path, max_to_keep=5)
```

In []:

```
start_epoch = 0
if ckpt_manager.latest_checkpoint:
    start_epoch = int(ckpt_manager.latest_checkpoint.split('-')[-1])
    # restoring the latest checkpoint in checkpoint_path
    ckpt.restore(ckpt_manager.latest_checkpoint)
```

Training

- You extract the features stored in the respective .npy files and then pass those features through the encoder.
- The encoder output, hidden state(initialized to 0) and the decoder input (which is the start token) is passed to the decoder.
- The decoder returns the predictions and the decoder hidden state.
- The decoder hidden state is then passed back into the model and the predictions are used to calculate the loss.
- Use teacher forcing to decide the next input to the decoder.
- Teacher forcing is the technique where the target word is passed as the next input to the decoder.
- The final step is to calculate the gradients and apply it to the optimizer and backpropagate.

In []:

```
# adding this in a separate cell because if you run the training cell
# many times, the loss_plot array will be reset
loss_plot = []
```

In []:

```
@tf.function
def train_step(img_tensor, target):
    loss = 0

    # initializing the hidden state for each batch
    # because the captions are not related from image to image
    hidden = decoder.reset_state(batch_size=target.shape[0])

    dec_input = tf.expand_dims([word_to_index('<start>')] * target.shape[0], 1)

    with tf.GradientTape() as tape:
        features = encoder(img_tensor)

        for i in range(1, target.shape[1]):
            # passing the features through the decoder
            predictions, hidden, _ = decoder(dec_input, features, hidden)

            loss += loss_function(target[:, i], predictions)

            # using teacher forcing
            dec_input = tf.expand_dims(target[:, i], 1)

    total_loss = (loss / int(target.shape[1]))

    trainable_variables = encoder.trainable_variables + decoder.trainable_variables
    gradients = tape.gradient(loss, trainable_variables)
    optimizer.apply_gradients(zip(gradients, trainable_variables))

    return loss, total_loss
```

In []:

```
EPOCHS = 20

for epoch in range(start_epoch, EPOCHS):
    start = time.time()
    total_loss = 0

    for (batch, (img_tensor, target)) in enumerate(dataset):
        batch_loss, t_loss = train_step(img_tensor, target)
        total_loss += t_loss

        if batch % 100 == 0:
            average_batch_loss = batch_loss.numpy()/int(target.shape[1])
            print(f'Epoch {epoch+1} Batch {batch} Loss {average_batch_loss:.4f}')
    # storing the epoch end loss value to plot later
    loss_plot.append(total_loss / num_steps)

    if epoch % 5 == 0:
        ckpt_manager.save()

    print(f'Epoch {epoch+1} Loss {total_loss/num_steps:.6f}')
    print(f'Time taken for 1 epoch {time.time()-start:.2f} sec\n')
```

In []:

```
plt.plot(loss_plot)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss Plot')
plt.show()
```

Caption!

- The evaluate function is similar to the training loop, except you don't use teacher forcing here. The input to the decoder at each time step is its previous predictions along with the hidden state and the encoder output.
- Stop predicting when the model predicts the end token.
- And store the attention weights for every time step.

In []:

```
def evaluate(image):
    attention_plot = np.zeros((max_length, attention_features_shape))

    hidden = decoder.reset_state(batch_size=1)

    temp_input = tf.expand_dims(load_image(image)[0], 0)
    img_tensor_val = image_features_extract_model(temp_input)
    img_tensor_val = tf.reshape(img_tensor_val, (img_tensor_val.shape[0],
                                                -1,
                                                img_tensor_val.shape[3]))

    features = encoder(img_tensor_val)

    dec_input = tf.expand_dims([word_to_index('<start>')], 0)
    result = []

    for i in range(max_length):
        predictions, hidden, attention_weights = decoder(dec_input,
                                                          features,
                                                          hidden)

        attention_plot[i] = tf.reshape(attention_weights, (-1, )).numpy()

        predicted_id = tf.random.categorical(predictions, 1)[0][0].numpy()
        predicted_word = tf.compat.as_text(index_to_word(predicted_id).numpy())
        result.append(predicted_word)

        if predicted_word == '<end>':
            return result, attention_plot

        dec_input = tf.expand_dims([predicted_id], 0)

    attention_plot = attention_plot[:len(result), :]
    return result, attention_plot
```

In []:

```
def plot_attention(image, result, attention_plot):
    temp_image = np.array(Image.open(image))

    fig = plt.figure(figsize=(10, 10))

    len_result = len(result)
    for i in range(len_result):
        temp_att = np.resize(attention_plot[i], (8, 8))
        grid_size = max(int(np.ceil(len_result/2)), 2)
        ax = fig.add_subplot(grid_size, grid_size, i+1)
        ax.set_title(result[i])
        img = ax.imshow(temp_image)
        ax.imshow(temp_att, cmap='gray', alpha=0.6, extent=img.get_extent())

    plt.tight_layout()
    plt.show()
```

In []:

```
# captions on the validation set
rid = np.random.randint(0, len(img_name_val))
image = img_name_val[rid]
real_caption = ' '.join([tf.compat.as_text(index_to_word(i).numpy())
                        for i in cap_val[rid] if i not in [0]])
result, attention_plot = evaluate(image)

print('Real Caption:', real_caption)
print('Prediction Caption:', ' '.join(result))
plot_attention(image, result, attention_plot)
```

Try it on your own images

For fun, below you're provided a method you can use to caption your own images with the model you've just trained. Keep in mind, it was trained on a relatively small amount of data, and your images may be different from the training data (so be prepared for weird results!)

In []:

```
image_url = 'https://tensorflow.org/images/surf.jpg'
image_extension = image_url[-4:]
image_path = tf.keras.utils.get_file('image'+image_extension, origin=image_url)

result, attention_plot = evaluate(image_path)
print('Prediction Caption:', ' '.join(result))
plot_attention(image_path, result, attention_plot)
# opening the image
Image.open(image_path)
```

Next steps

Congrats! You've just trained an image captioning model with attention. Next, take a look at this example [Neural Machine Translation with Attention](#) (https://www.tensorflow.org/text/tutorials/nmt_with_attention). It uses a similar architecture to translate between Spanish and English sentences. You can also experiment with training the code in this notebook on a different dataset.

Copyright 2020 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/text/word2vec)

(<https://www.tensorflow.org/tutorials/text/word2vec>)



[Run in Google Colab](#)

(<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/text/word2vec.ipynb>)

word2vec

word2vec is not a singular algorithm, rather, it is a family of model architectures and optimizations that can be used to learn word embeddings from large datasets. Embeddings learned through word2vec have proven to be successful on a variety of downstream natural language processing tasks.

Note: This tutorial is based on [Efficient estimation of word representations in vector space \(https://arxiv.org/pdf/1301.3781.pdf\)](https://arxiv.org/pdf/1301.3781.pdf) and [Distributed representations of words and phrases and their compositionality \(https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf\)](https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf). It is not an exact implementation of the papers. Rather, it is intended to illustrate the key ideas.

These papers proposed two methods for learning representations of words:

- **Continuous bag-of-words model:** predicts the middle word based on surrounding context words. The context consists of a few words before and after the current (middle) word. This architecture is called a bag-of-words model as the order of words in the context is not important.
- **Continuous skip-gram model:** predicts words within a certain range before and after the current word in the same sentence. A worked example of this is given below.

You'll use the skip-gram approach in this tutorial. First, you'll explore skip-grams and other concepts using a single sentence for illustration. Next, you'll train your own word2vec model on a small dataset. This tutorial also contains code to export the trained embeddings and visualize them in the [TensorFlow Embedding Projector \(http://projector.tensorflow.org/\)](http://projector.tensorflow.org/).

Skip-gram and negative sampling

While a bag-of-words model predicts a word given the neighboring context, a skip-gram model predicts the context (or neighbors) of a word, given the word itself. The model is trained on skip-grams, which are n-grams that allow tokens to be skipped (see the diagram below for an example). The context of a word can be represented through a set of skip-gram pairs of `(target_word, context_word)` where `context_word` appears in the neighboring context of `target_word`.

Consider the following sentence of eight words:

The wide road shimmered in the hot sun.

The context words for each of the 8 words of this sentence are defined by a window size. The window size determines the span of words on either side of a `target_word` that can be considered a `context_word`. Below is a table of skip-grams for target words based on different window sizes.

Note: For this tutorial, a window size of `n` implies `n` words on each side with a total window span of $2*n+1$ words across a word.

Window Size	Text	Skip-grams
	[The <u>wide</u> road shimmered] in the hot sun.	wide, the wide, road wide, shimmered
2	The [wide road <u>shimmered</u> in the] hot sun.	shimmered, wide shimmered, road shimmered, in shimmered, the
	The wide road shimmered in [the hot <u>sun</u>].	sun, the sun, hot
	[The <u>wide</u> road shimmered in] the hot sun.	wide, the wide, road wide, shimmered wide, in
3	[The wide road <u>shimmered</u> in the hot] sun.	shimmered, the shimmered, wide shimmered, road shimmered, in shimmered, the shimmered, hot
	The wide road shimmered [in the hot <u>sun</u>].	sun, in sun, the sun, hot

The training objective of the skip-gram model is to maximize the probability of predicting context words given the target word. For a sequence of words w_1, w_2, \dots, w_T , the objective can be written as the average log probability

$$\frac{1}{T} \sum_{t=1}^T \sum_{-c \leq j \leq c, j \neq 0} \log p(w_{t+j} | w_t)$$

where c is the size of the training context. The basic skip-gram formulation defines this probability using the softmax function.

$$p(w_O|w_I) = \frac{\exp\left(v'_{w_O}^\top v_{w_I}\right)}{\sum_{w=1}^W \exp\left(v'_{w}^\top v_{w_I}\right)}$$

where v and v' are target and context vector representations of words and W is vocabulary size.

Computing the denominator of this formulation involves performing a full softmax over the entire vocabulary words, which are often large (10^5 - 10^7) terms.

The [noise contrastive estimation \(https://www.tensorflow.org/api_docs/python/tf/nn/nce_loss\)](https://www.tensorflow.org/api_docs/python/tf/nn/nce_loss) (NCE) loss function is an efficient approximation for a full softmax. With an objective to learn word embeddings instead of modeling the word distribution, the NCE loss can be [simplified \(https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf\)](https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf) to use negative sampling.

The simplified negative sampling objective for a target word is to distinguish the context word from `num_ns` negative samples drawn from noise distribution $P_n(w)$ of words. More precisely, an efficient approximation of full softmax over the vocabulary is, for a skip-gram pair, to pose the loss for a target word as a classification problem between the context word and `num_ns` negative samples.

A negative sample is defined as a `(target_word, context_word)` pair such that the `context_word` does not appear in the `window_size` neighborhood of the `target_word`. For the example sentence, these are a few potential negative samples (when `window_size` is 2).

```
(hot, shimmered)
(wide, hot)
(wide, sun)
```

In the next section, you'll generate skip-grams and negative samples for a single sentence. You'll also learn about subsampling techniques and train a classification model for positive and negative training examples later in the tutorial.

Setup

In []:

```
import io
import re
import string
import tqdm

import numpy as np

import tensorflow as tf
from tensorflow.keras import layers
```

In []:

```
# Load the TensorBoard notebook extension
%load_ext tensorboard
```

In []:

```
SEED = 42
AUTOTUNE = tf.data.AUTOTUNE
```

Vectorize an example sentence

Consider the following sentence:

The wide road shimmered in the hot sun.

Tokenize the sentence:

In []:

```
sentence = "The wide road shimmered in the hot sun"
tokens = list(sentence.lower().split())
print(len(tokens))
```

Create a vocabulary to save mappings from tokens to integer indices:

In []:

```
vocab, index = {}, 1 # start indexing from 1
vocab['<pad>'] = 0 # add a padding token
for token in tokens:
    if token not in vocab:
        vocab[token] = index
        index += 1
vocab_size = len(vocab)
print(vocab)
```

Create an inverse vocabulary to save mappings from integer indices to tokens:

In []:

```
inverse_vocab = {index: token for token, index in vocab.items()}
print(inverse_vocab)
```

Vectorize your sentence:

In []:

```
example_sequence = [vocab[word] for word in tokens]
print(example_sequence)
```

Generate skip-grams from one sentence

The `tf.keras.preprocessing.sequence` module provides useful functions that simplify data preparation for word2vec. You can use the `tf.keras.preprocessing.sequence.skipgrams` to generate skip-gram pairs from the `example_sequence` with a given `window_size` from tokens in the range `[0, vocab_size]`.

Note: `negative_samples` is set to `0` here, as batching negative samples generated by this function requires a bit of code. You will use another function to perform negative sampling in the next section.

In []:

```
window_size = 2
positive_skip_grams, _ = tf.keras.preprocessing.sequence.skipgrams(
    example_sequence,
    vocabulary_size=vocab_size,
    window_size=window_size,
    negative_samples=0)
print(len(positive_skip_grams))
```

Print a few positive skip-grams:

In []:

```
for target, context in positive_skip_grams[:5]:
    print(f"({target}, {context}): ({inverse_vocab[target]}, {inverse_vocab[context]})")
```

Negative sampling for one skip-gram

The `skipgrams` function returns all positive skip-gram pairs by sliding over a given window span. To produce additional skip-gram pairs that would serve as negative samples for training, you need to sample random words from the vocabulary. Use the `tf.random.log_uniform_candidate_sampler` function to sample `num_ns` number of negative samples for a given target word in a window. You can call the function on one skip-grams's target word and pass the context word as true class to exclude it from being sampled.

Key point: `num_ns` (the number of negative samples per a positive context word) in the `[5, 20]` range is [shown to work](#) (<https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf>) best for smaller datasets, while `num_ns` in the `[2, 5]` range suffices for larger datasets.

In []:

```
# Get target and context words for one positive skip-gram.  
target_word, context_word = positive_skip_grams[0]  
  
# Set the number of negative samples per positive context.  
num_ns = 4  
  
context_class = tf.reshape(tf.constant(context_word, dtype="int64"), (1, 1))  
negative_sampling_candidates, _ = tf.random.log_uniform_candidate_sampler(  
    true_classes=context_class, # class that should be sampled as 'positive'  
    num_true=1, # each positive skip-gram has 1 positive context class  
    num_sampled=num_ns, # number of negative context words to sample  
    unique=True, # all the negative samples should be unique  
    range_max=vocab_size, # pick index of the samples from [0, vocab_size]  
    seed=SEED, # seed for reproducibility  
    name="negative_sampling" # name of this operation  
)  
print(negative_sampling_candidates)  
print([inverse_vocab[index.numpy()] for index in negative_sampling_candidates])
```

Construct one training example

For a given positive (`target_word, context_word`) skip-gram, you now also have `num_ns` negative sampled context words that do not appear in the window size neighborhood of `target_word`. Batch the 1 positive `context_word` and `num_ns` negative context words into one tensor. This produces a set of positive skip-grams (labeled as 1) and negative samples (labeled as 0) for each target word.

In []:

```
# Add a dimension so you can use concatenation (in the next step).  
negative_sampling_candidates = tf.expand_dims(negative_sampling_candidates, 1)  
  
# Concatenate a positive context word with negative sampled words.  
context = tf.concat([context_class, negative_sampling_candidates], 0)  
  
# Label the first context word as `1` (positive) followed by `num_ns` `0`s (negative).  
label = tf.constant([1] + [0]*num_ns, dtype="int64")  
  
# Reshape the target to shape `(1,)` and context and label to `(num_ns+1,)`.  
target = tf.squeeze(target_word)  
context = tf.squeeze(context)  
label = tf.squeeze(label)
```

Check out the context and the corresponding labels for the target word from the skip-gram example above:

In []:

```
print(f"target_index : {target}")  
print(f"target_word : {inverse_vocab[target_word]}")  
print(f"context_indices : {context}")  
print(f"context_words : {[inverse_vocab[c.numpy()] for c in context]}")  
print(f"label : {label}")
```

A tuple of (`target, context, label`) tensors constitutes one training example for training your skip-gram negative sampling word2vec model. Notice that the target is of shape `(1,)` while the context and label are of shape `(1+num_ns,)`

In []:

```
print("target :", target)  
print("context :", context)  
print("label : ", label)
```

Summary

This diagram summarizes the procedure of generating a training example from a sentence:

The wide road shimmered in the hot sun.

`tf.keras.preprocessing.sequence.skipgrams`



(wide, road)	...	(road, shimmered)	(hot, sun)	...	(the, hot)
(2, 3)	...	(3, 4)	(6, 7)	...	(1, 6)

`tf.random.log_uniform_candidate_sampler
(negative_samples = 4)`



(wide, road)	(wide, sun)	(wide, hot)	(wide, temperature)	(wide, code)
(2, 3)	(2, 7)	(2,6)	(2, 23)	(2, 2196)

`concat and add label (pos:1/neg:0)`



(wide, road)	(wide, sun)	(wide, hot)	(wide, temperature)	(wide, code)
(2, 3)	(2, 7)	(2,6)	(2, 23)	(2, 2196)
1	0	0	0	0

`build context words and labels for all vocab words`



Word	Context words					Labels				
2	3	7	6	23	2196	⇒	1	0	0	0
23	12	6	94	17	1085	⇒	1	0	0	0
84	784	11	68	41	453	⇒	1	0	0	0
						⋮				
V	45	598	1	117	43	⇒	1	0	0	0

Notice that the words `temperature` and `code` are not part of the input sentence. They belong to the vocabulary like certain other indices used in the diagram above.

Compile all steps into one function

Skip-gram sampling table

A large dataset means larger vocabulary with higher number of more frequent words such as stopwords. Training examples obtained from sampling commonly occurring words (such as `the`, `is`, `on`) don't add much useful information for the model to learn from. [Mikolov et al.](#) (<https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf>) suggest subsampling of frequent words as a helpful practice to improve embedding quality.

The `tf.keras.preprocessing.sequence.skipgrams` function accepts a sampling table argument to encode probabilities of sampling any token. You can use the `tf.keras.preprocessing.sequence.make_sampling_table` to generate a word-frequency rank based probabilistic sampling table and pass it to the `skipgrams` function. Inspect the sampling probabilities for a `vocab_size` of 10.

In []:

```
sampling_table = tf.keras.preprocessing.sequence.make_sampling_table(size=10)
print(sampling_table)
```

`sampling_table[i]` denotes the probability of sampling the i -th most common word in a dataset. The function assumes a [Zipf's distribution](#) (https://en.wikipedia.org/wiki/Zipf%27s_law) of the word frequencies for sampling.

Key point: The `tf.random.log_uniform_candidate_sampler` already assumes that the vocabulary frequency follows a log-uniform (Zipf's) distribution. Using these distribution weighted sampling also helps approximate the Noise Contrastive Estimation (NCE) loss with simpler loss functions for training a negative sampling objective.

Generate training data

Compile all the steps described above into a function that can be called on a list of vectorized sentences obtained from any text dataset. Notice that the sampling table is built before sampling skip-gram word pairs. You will use this function in the later sections.

In []:

```
# Generates skip-gram pairs with negative sampling for a list of sequences
# (int-encoded sentences) based on window size, number of negative samples
# and vocabulary size.
def generate_training_data(sequences, window_size, num_ns, vocab_size, seed):
    # Elements of each training example are appended to these lists.
    targets, contexts, labels = [], [], []

    # Build the sampling table for `vocab_size` tokens.
    sampling_table = tf.keras.preprocessing.sequence.make_sampling_table(vocab_size)

    # Iterate over all sequences (sentences) in the dataset.
    for sequence in tqdm(sequences):

        # Generate positive skip-gram pairs for a sequence (sentence).
        positive_skip_grams, _ = tf.keras.preprocessing.sequence.skipgrams(
            sequence,
            vocabulary_size=vocab_size,
            sampling_table=sampling_table,
            window_size=window_size,
            negative_samples=0)

        # Iterate over each positive skip-gram pair to produce training examples
        # with a positive context word and negative samples.
        for target_word, context_word in positive_skip_grams:
            context_class = tf.expand_dims(
                tf.constant([context_word], dtype="int64"), 1)
            negative_sampling_candidates, _, _ = tf.random.log_uniform_candidate_sampler(
                true_classes=context_class,
                num_true=1,
                num_sampled=num_ns,
                unique=True,
                range_max=vocab_size,
                seed=SEED,
                name="negative_sampling")

            # Build context and label vectors (for one target word)
            negative_sampling_candidates = tf.expand_dims(
                negative_sampling_candidates, 1)

            context = tf.concat([context_class, negative_sampling_candidates], 0)
            label = tf.constant([1] + [0]*num_ns, dtype="int64")

            # Append each element from the training example to global lists.
            targets.append(target_word)
            contexts.append(context)
            labels.append(label)

    return targets, contexts, labels
```

Prepare training data for word2vec

With an understanding of how to work with one sentence for a skip-gram negative sampling based word2vec model, you can proceed to generate training examples from a larger list of sentences!

Download text corpus

You will use a text file of Shakespeare's writing for this tutorial. Change the following line to run this code on your own data.

In []:

```
path_to_file = tf.keras.utils.get_file('shakespeare.txt', 'https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt')
```

Read the text from the file and print the first few lines:

In []:

```
with open(path_to_file) as f:
    lines = f.read().splitlines()
for line in lines[:20]:
    print(line)
```

Use the non empty lines to construct a `tf.data.TextLineDataset` object for the next steps:

In []:

```
text_ds = tf.data.TextLineDataset(path_to_file).filter(lambda x: tf.cast(tf.strings.length(x), bool))
```

Vectorize sentences from the corpus

You can use the `TextVectorization` layer to vectorize sentences from the corpus. Learn more about using this layer in this [Text classification \(https://www.tensorflow.org/tutorials/keras/text_classification\)](#) tutorial. Notice from the first few sentences above that the text needs to be in one case and punctuation needs to be removed. To do this, define a `custom_standardization` function that can be used in the `TextVectorization` layer.

In []:

```
# Now, create a custom standardization function to lowercase the text and
# remove punctuation.
def custom_standardization(input_data):
    lowercase = tf.strings.lower(input_data)
    return tf.strings.regex_replace(lowercase,
                                    '[%s]' % re.escape(string.punctuation), '')

# Define the vocabulary size and the number of words in a sequence.
vocab_size = 4096
sequence_length = 10

# Use the `TextVectorization` layer to normalize, split, and map strings to
# integers. Set the `output_sequence_length` length to pad all samples to the
# same length.
vectorize_layer = layers.TextVectorization(
    standardize=custom_standardization,
    max_tokens=vocab_size,
    output_mode='int',
    output_sequence_length=sequence_length)
```

Call `TextVectorization.adapt` on the text dataset to create vocabulary.

In []:

```
vectorize_layer.adapt(text_ds.batch(1024))
```

Once the state of the layer has been adapted to represent the text corpus, the vocabulary can be accessed with `TextVectorization.get_vocabulary`. This function returns a list of all vocabulary tokens sorted (descending) by their frequency.

In []:

```
# Save the created vocabulary for reference.
inverse_vocab = vectorize_layer.get_vocabulary()
print(inverse_vocab[:20])
```

The `vectorize_layer` can now be used to generate vectors for each element in the `text_ds` (a `tf.data.Dataset`). Apply `Dataset.batch`, `Dataset.prefetch`, `Dataset.map`, and `Dataset.unbatch`.

In []:

```
# Vectorize the data in text_ds.  
text_vector_ds = text_ds.batch(1024).prefetch(AUTOTUNE).map(vectorize_layer).unbatch()
```

Obtain sequences from the dataset

You now have a `tf.data.Dataset` of integer encoded sentences. To prepare the dataset for training a word2vec model, flatten the dataset into a list of sentence vector sequences. This step is required as you would iterate over each sentence in the dataset to produce positive and negative examples.

Note: Since the `generate_training_data()` defined earlier uses non-TensorFlow Python/NumPy functions, you could also use a `tf.py_function` or `tf.numpy_function` with `tf.data.Dataset.map`.

In []:

```
sequences = list(text_vector_ds.as_numpy_iterator())  
print(len(sequences))
```

Inspect a few examples from `sequences`:

In []:

```
for seq in sequences[:5]:  
    print(f"[seq] => {[inverse_vocab[i] for i in seq]}")
```

Generate training examples from sequences

`sequences` is now a list of int encoded sentences. Just call the `generate_training_data` function defined earlier to generate training examples for the word2vec model. To recap, the function iterates over each word from each sequence to collect positive and negative context words. Length of target, contexts and labels should be the same, representing the total number of training examples.

In []:

```
targets, contexts, labels = generate_training_data(  
    sequences=sequences,  
    window_size=2,  
    num_ns=4,  
    vocab_size=vocab_size,  
    seed=SEED)  
  
targets = np.array(targets)  
contexts = np.array(contexts)[:, :, 0]  
labels = np.array(labels)  
  
print('\n')  
print(f"targets.shape: {targets.shape}")  
print(f"contexts.shape: {contexts.shape}")  
print(f"labels.shape: {labels.shape}")
```

Configure the dataset for performance

To perform efficient batching for the potentially large number of training examples, use the `tf.data.Dataset` API. After this step, you would have a `tf.data.Dataset` object of (`target_word`, `context_word`), (`label`) elements to train your word2vec model!

In []:

```
BATCH_SIZE = 1024  
BUFFER_SIZE = 10000  
dataset = tf.data.Dataset.from_tensor_slices(((targets, contexts), labels))  
dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE, drop_remainder=True)  
print(dataset)
```

Apply `Dataset.cache` and `Dataset.prefetch` to improve performance:

In []:

```
dataset = dataset.cache().prefetch(buffer_size=AUTOTUNE)  
print(dataset)
```

Model and training

The word2vec model can be implemented as a classifier to distinguish between true context words from skip-grams and false context words obtained through negative sampling. You can perform a dot product multiplication between the embeddings of target and context words to obtain predictions for labels and compute the loss function against true labels in the dataset.

Subclassed word2vec model

Use the [Keras Subclassing API \(https://www.tensorflow.org/guide/keras/custom_layers_and_models\)](https://www.tensorflow.org/guide/keras/custom_layers_and_models) to define your word2vec model with the following layers:

- `target_embedding`: A `tf.keras.layers.Embedding` layer, which looks up the embedding of a word when it appears as a target word. The number of parameters in this layer are `(vocab_size * embedding_dim)`.
- `context_embedding`: Another `tf.keras.layers.Embedding` layer, which looks up the embedding of a word when it appears as a context word. The number of parameters in this layer are the same as those in `target_embedding`, i.e. `(vocab_size * embedding_dim)`.
- `dots`: A `tf.keras.layers.Dot` layer that computes the dot product of target and context embeddings from a training pair.
- `flatten`: A `tf.keras.layers.Flatten` layer to flatten the results of `dots` layer into logits.

With the subclassed model, you can define the `call()` function that accepts `(target, context)` pairs which can then be passed into their corresponding embedding layer. Reshape the `context_embedding` to perform a dot product with `target_embedding` and return the flattened result.

Key point: The `target_embedding` and `context_embedding` layers can be shared as well. You could also use a concatenation of both embeddings as the final word2vec embedding.

In []:

```
class Word2Vec(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim):
        super(Word2Vec, self).__init__()
        self.target_embedding = layers.Embedding(vocab_size,
                                                embedding_dim,
                                                input_length=1,
                                                name="w2v_embedding")
        self.context_embedding = layers.Embedding(vocab_size,
                                                embedding_dim,
                                                input_length=num_ns+1)

    def call(self, pair):
        target, context = pair
        # target: (batch, dummy?) # The dummy axis doesn't exist in TF2.7+
        # context: (batch, context)
        if len(target.shape) == 2:
            target = tf.squeeze(target, axis=1)
        # target: (batch,)
        word_emb = self.target_embedding(target)
        # word_emb: (batch, embed)
        context_emb = self.context_embedding(context)
        # context_emb: (batch, context, embed)
        dots = tf.einsum('be,bce->bc', word_emb, context_emb)
        # dots: (batch, context)
        return dots
```

Define loss function and compile model

For simplicity, you can use `tf.keras.losses.CategoricalCrossEntropy` as an alternative to the negative sampling loss. If you would like to write your own custom loss function, you can also do so as follows:

```
def custom_loss(x_logit, y_true):
    return tf.nn.sigmoid_cross_entropy_with_logits(logits=x_logit, labels=y_true)
```

It's time to build your model! Instantiate your word2vec class with an embedding dimension of 128 (you could experiment with different values). Compile the model with the `tf.keras.optimizers.Adam` optimizer.

In []:

```
embedding_dim = 128
word2vec = Word2Vec(vocab_size, embedding_dim)
word2vec.compile(optimizer='adam',
                  loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),
                  metrics=['accuracy'])
```

Also define a callback to log training statistics for Tensorboard:

In []:

```
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir="logs")
```

Train the model on the `dataset` for some number of epochs:

In []:

```
word2vec.fit(dataset, epochs=20, callbacks=[tensorboard_callback])
```

Tensorboard now shows the word2vec model's accuracy and loss:

In []:

```
#docs_infra: no execute
%tensorboard --logdir logs
```

Embedding lookup and analysis

Obtain the weights from the model using `Model.get_layer` and `Layer.get_weights`. The `TextVectorization.get_vocabulary` function provides the vocabulary to build a metadata file with one token per line.

In []:

```
weights = word2vec.get_layer('w2v_embedding').get_weights()[0]
vocab = vectorize_layer.get_vocabulary()
```

Create and save the vectors and metadata files:

In []:

```
out_v = io.open('vectors.tsv', 'w', encoding='utf-8')
out_m = io.open('metadata.tsv', 'w', encoding='utf-8')

for index, word in enumerate(vocab):
    if index == 0:
        continue # skip 0, it's padding.
    vec = weights[index]
    out_v.write('\t'.join([str(x) for x in vec]) + "\n")
    out_m.write(word + "\n")
out_v.close()
out_m.close()
```

Download the `vectors.tsv` and `metadata.tsv` to analyze the obtained embeddings in the [Embedding Projector](https://projector.tensorflow.org/) (<https://projector.tensorflow.org/>):

In []:

```
try:
    from google.colab import files
    files.download('vectors.tsv')
    files.download('metadata.tsv')
except Exception:
    pass
```

Next steps

This tutorial has shown you how to implement a skip-gram word2vec model with negative sampling from scratch and visualize the obtained word embeddings.

- To learn more about word vectors and their mathematical representations, refer to these [notes](https://web.stanford.edu/class/cs224n/readings/cs224n-2019-notes01-wordvecs1.pdf) (<https://web.stanford.edu/class/cs224n/readings/cs224n-2019-notes01-wordvecs1.pdf>).
- To learn more about advanced text processing, read the [Transformer model for language understanding](https://www.tensorflow.org/tutorials/text/transformer) (<https://www.tensorflow.org/tutorials/text/transformer>) tutorial.
- If you're interested in pre-trained embedding models, you may also be interested in [Exploring the TF-Hub CORD-19 Swivel Embeddings](https://www.tensorflow.org/hub/tutorials/cord_19_embeddings_keras) (https://www.tensorflow.org/hub/tutorials/cord_19_embeddings_keras), or the [Multilingual Universal Sentence Encoder](https://www.tensorflow.org/hub/tutorials/cross_lingual_similarity_with_tf_hub_multilingual_universal_encoder) (https://www.tensorflow.org/hub/tutorials/cross_lingual_similarity_with_tf_hub_multilingual_universal_encoder).
- You may also like to train the model on a new dataset (there are many available in [TensorFlow Datasets](https://www.tensorflow.org/datasets) (<https://www.tensorflow.org/datasets>)).

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Integrated gradients



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/interpretability/integrated_gradients)

(https://www.tensorflow.org/tutorials/interpretability/integrated_gradients)



(<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/interpretability/inte>

This tutorial demonstrates how to implement **Integrated Gradients (IG)**, an Explainable AI (https://en.wikipedia.org/wiki/Explainable_artificial_intelligence) technique introduced in the paper [Axiomatic Attribution for Deep Networks](https://arxiv.org/abs/1703.01365) (<https://arxiv.org/abs/1703.01365>). IG aims to explain the relationship between a model's predictions in terms of its features. It has many use cases including understanding feature importances, identifying data skew, and debugging model performance.

IG has become a popular interpretability technique due to its broad applicability to any differentiable model (e.g. images, text, structured data), ease of implementation, theoretical justifications, and computational efficiency relative to alternative approaches that allows it to scale to large networks and feature spaces such as images.

In this tutorial, you will walk through an implementation of IG step-by-step to understand the pixel feature importances of an image classifier. As an example, consider this [image](https://commons.wikimedia.org/wiki/File:San_Francisco_fireboat_showing_off.jpg) (https://commons.wikimedia.org/wiki/File:San_Francisco_fireboat_showing_off.jpg) of a fireboat spraying jets of water. You would classify this image as a fireboat and might highlight the pixels making up the boat and water cannons as being important to your decision. Your model will also classify this image as a fireboat later on in this tutorial; however, does it highlight the same pixels as important when explaining its decision?

In the images below titled "IG Attribution Mask" and "Original + IG Mask Overlay" you can see that your model instead highlights (in purple) the pixels comprising the boat's water cannons and jets of water as being more important than the boat itself to its decision. How will your model generalize to new fireboats? What about fireboats without water jets? Read on to learn more about how IG works and how to apply IG to your models to better understand the relationship between their predictions and underlying features.



Setup

In []:

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
import tensorflow_hub as hub
```

Download a pretrained image classifier from TF-Hub

IG can be applied to any differentiable model. In the spirit of the original paper, you will use a pre-trained version of the same model, Inception V1, which you will download from [TensorFlow Hub](https://tfhub.dev/google-imagenet/inception_v1/classification/4) (https://tfhub.dev/google-imagenet/inception_v1/classification/4).

In []:

```
model = tf.keras.Sequential([
    hub.KerasLayer(
        name='inception_v1',
        handle='https://tfhub.dev/google-imagenet/inception_v1/classification/4',
        trainable=False),
])
model.build([None, 224, 224, 3])
model.summary()
```

From the module page, you need to keep in mind the following about Inception V1:

Inputs: The expected input shape for the model is (None, 224, 224, 3) . This is a dense 4D tensor of dtype float32 and shape (batch_size, height, width, RGB channels) whose elements are RGB color values of pixels normalized to the range [0, 1]. The first element is None to indicate that the model can take any integer batch size.

Outputs: A tf.Tensor of logits in the shape of (batch_size, 1001) . Each row represents the model's predicted score for each of 1,001 classes from ImageNet. For the model's top predicted class index you can use tf.argmax(predictions, axis=-1) . Furthermore, you can also convert the model's logit output to predicted probabilities across all classes using tf.nn.softmax(predictions, axis=-1) to quantify the model's uncertainty as well as explore similar predicted classes for debugging.

In []:

```
def load_imagenet_labels(file_path):
    labels_file = tf.keras.utils.get_file('ImageNetLabels.txt', file_path)
    with open(labels_file) as reader:
        f = reader.read()
        labels = f.splitlines()
    return np.array(labels)
```

In []:

```
imagenet_labels = load_imagenet_labels('https://storage.googleapis.com/download.tensorflow.org/data/ImageNetLabels.txt')
```

Load and preprocess images with tf.image

You will illustrate IG using two images from [Wikimedia Commons](https://commons.wikimedia.org/wiki/Main_Page) (https://commons.wikimedia.org/wiki/Main_Page): a [Fireboat](https://commons.wikimedia.org/wiki/File:San_Francisco_fireboat_showing_off.jpg) (https://commons.wikimedia.org/wiki/File:San_Francisco_fireboat_showing_off.jpg), and a [Giant Panda](https://commons.wikimedia.org/wiki/File:Giant_Panda_2.JPG) (https://commons.wikimedia.org/wiki/File:Giant_Panda_2.JPG).

In []:

```
def read_image(file_name):
    image = tf.io.read_file(file_name)
    image = tf.io.decode_jpeg(image, channels=3)
    image = tf.image.convert_image_dtype(image, tf.float32)
    image = tf.image.resize_with_pad(image, target_height=224, target_width=224)
    return image
```

In []:

```
img_url = {
    'Fireboat': 'http://storage.googleapis.com/download.tensorflow.org/example_images/San_Francisco_fireboat_showing_off.jpg',
    'Giant Panda': 'http://storage.googleapis.com/download.tensorflow.org/example_images/Giant_Panda_2.jpeg',
}

img_paths = {name: tf.keras.utils.get_file(name, url) for (name, url) in img_url.items()}
img_name_tensors = {name: read_image(img_path) for (name, img_path) in img_paths.items()}
```

In []:

```
plt.figure(figsize=(8, 8))
for n, (name, img_tensors) in enumerate(img_name_tensors.items()):
    ax = plt.subplot(1, 2, n+1)
    ax.imshow(img_tensors)
    ax.set_title(name)
    ax.axis('off')
plt.tight_layout()
```

Classify images

Let's start by classifying these images and displaying the top 3 most confident predictions. Following is a utility function to retrieve the top k predicted labels and probabilities.

In []:

```
def top_k_predictions(img, k=3):
    image_batch = tf.expand_dims(img, 0)
    predictions = model(image_batch)
    probs = tf.nn.softmax(predictions, axis=-1)
    top_probs, top_idxs = tf.math.top_k(input=probs, k=k)
    top_labels = imagenet_labels[tuple(top_idxs)]
    return top_labels, top_probs[0]
```

In []:

```
for (name, img_tensor) in img_name_tensors.items():
    plt.imshow(img_tensor)
    plt.title(name, fontweight='bold')
    plt.axis('off')
    plt.show()

pred_label, pred_prob = top_k_predictions(img_tensor)
for label, prob in zip(pred_label, pred_prob):
    print(f'{label}: {prob:0.1%}' )
```

Calculate Integrated Gradients

Your model, Inception V1, is a learned function that describes a mapping between your input feature space, image pixel values, and an output space defined by ImageNet class probability values between 0 and 1. Early interpretability methods for neural networks assigned feature importance scores using gradients, which tell you which pixels have the steepest local relative to your model's prediction at a given point along your model's prediction function. However, gradients only describe *local* changes in your model's prediction function with respect to pixel values and do not fully describe your entire model prediction function. As your model fully "learns" the relationship between the range of an individual pixel and the correct ImageNet class, the gradient for this pixel will *saturate*, meaning become increasingly small and even go to zero. Consider the simple model function below:

In []:

```
def f(x):
    """A simplified model function."""
    return tf.where(x < 0.8, x, 0.8)

def interpolated_path(x):
    """A straight line path."""
    return tf.zeros_like(x)

x = tf.linspace(start=0.0, stop=1.0, num=6)
y = f(x)
```

In []:

```
#@title
fig = plt.figure(figsize=(12, 5))
ax0 = fig.add_subplot(121)
ax0.plot(x, f(x), marker='o')
ax0.set_title('Gradients saturate over F(x)', fontweight='bold')
ax0.text(0.2, 0.5, 'Gradients > 0 = \n x is important')
ax0.text(0.7, 0.85, 'Gradients = 0 \n x not important')
ax0.set_yticks(tf.range(0, 1.5, 0.5))
ax0.set_xticks(tf.range(0, 1.5, 0.5))
ax0.set_ylabel('F(x) - model true class predicted probability')
ax0.set_xlabel('x - (pixel value)')

ax1 = fig.add_subplot(122)
ax1.plot(x, f(x), marker='o')
ax1.plot(x, interpolated_path(x), marker='>')
ax1.set_title('IG intuition', fontweight='bold')
ax1.text(0.25, 0.1, 'Accumulate gradients along path')
ax1.set_ylabel('F(x) - model true class predicted probability')
ax1.set_xlabel('x - (pixel value)')
ax1.set_yticks(tf.range(0, 1.5, 0.5))
ax1.set_xticks(tf.range(0, 1.5, 0.5))
ax1.annotate('Baseline', xy=(0.0, 0.0), xytext=(0.0, 0.2),
            arrowprops=dict(facecolor='black', shrink=0.1))
ax1.annotate('Input', xy=(1.0, 0.0), xytext=(0.95, 0.2),
            arrowprops=dict(facecolor='black', shrink=0.1))
plt.show();
```

- **left:** Your model's gradients for pixel x are positive between 0.0 and 0.8 but go to 0.0 between 0.8 and 1.0. Pixel x clearly has a significant impact on pushing your model toward 80% predicted probability on the true class. *Does it make sense that pixel x 's importance is small or discontinuous?*
- **right:** The intuition behind IG is to accumulate pixel x 's local gradients and attribute its importance as a score for how much it adds or subtracts to your model's overall output class probability. You can break down and compute IG in 3 parts:

1. interpolate small steps along a straight line in the feature space between 0 (a baseline or starting point) and 1 (input pixel's value)
2. compute gradients at each step between your model's predictions with respect to each step
3. approximate the integral between your baseline and input by accumulating (cumulative average) these local gradients.

To reinforce this intuition, you will walk through these 3 parts by applying IG to the example "Fireboat" image below.

Establish a baseline

A baseline is an input image used as a starting point for calculating feature importance. Intuitively, you can think of the baseline's explanatory role as representing the impact of the absence of each pixel on the "Fireboat" prediction to contrast with its impact of each pixel on the "Fireboat" prediction when present in the input image. As a result, the choice of the baseline plays a central role in interpreting and visualizing pixel feature importances. For additional discussion of baseline selection, see the resources in the "Next steps" section at the bottom of this tutorial. Here, you will use a black image whose pixel values are all zero.

Other choices you could experiment with include an all white image, or a random image, which you can create with `tf.random.uniform(shape=(224,224,3), minval=0.0, maxval=1.0)`.

In []:

```
baseline = tf.zeros(shape=(224,224,3))
```

In []:

```
plt.imshow(baseline)
plt.title("Baseline")
plt.axis('off')
plt.show()
```

Unpack formulas into code

The formula for Integrated Gradients is as follows:

$$\text{IntegratedGradients}_i(x) := (x_i - x'_i) \times \int_{\alpha=0}^1 \frac{\partial F(x' + \alpha \times (x - x'))}{\partial x_i} d\alpha$$

where:

i = feature

x = input

x' = baseline

α = interpolation constant to perturb features by

In practice, computing a definite integral is not always numerically possible and can be computationally costly, so you compute the following numerical approximation:

$$\text{IntegratedGrads}_i^{\text{approx}}(x) := (x_i - x'_i) \times \sum_{k=1}^m \frac{\partial F(x' + \frac{k}{m} \times (x - x'))}{\partial x_i} \times \frac{1}{m}$$

where:

i = feature (individual pixel)

x = input (image tensor)

x' = baseline (image tensor)

k = scaled feature perturbation constant

m = number of steps in the Riemann sum approximation of the integral

$(x_i - x'_i)$ = a term for the difference from the baseline. This is necessary to scale the integrated gradients and keep them in terms of the original image. The path from the baseline image to the input is in pixel space. Since with IG you are integrating in a straight line (linear transformation) this ends up being roughly equivalent to the integral term of the derivative of the interpolated image function with respect to α with enough steps. The integral sums each pixel's gradient times the change in the pixel along the path. It's simpler to implement this integration as uniform steps from one image to the other, substituting $x := (x' + \alpha(x - x'))$. So the change of variables gives $dx = (x - x')d\alpha$. The $(x - x')$ term is constant and is factored out of the integral.

Interpolate images

$$\text{interpolate } m \text{ images at } k \text{ intervals}$$

$$\text{IntegratedGrads}_i^{\text{approx}}(x) := (x_i - x'_i) \times \sum_{k=1}^m \frac{\partial F(x' + \frac{k}{m} \times (x - x'))}{\partial x_i} \times \frac{1}{m}$$

First, you will generate a [linear interpolation](https://en.wikipedia.org/wiki/Linear_interpolation) (https://en.wikipedia.org/wiki/Linear_interpolation) between the baseline and the original image. You can think of interpolated images as small steps in the feature space between your baseline and input, represented by α in the original equation.

In []:

```
m_steps=50
alphas = tf.linspace(start=0.0, stop=1.0, num=m_steps+1) # Generate m_steps intervals for integral_approximation() below.
```

In []:

```
def interpolate_images(baseline,
                      image,
                      alphas):
    alphas_x = alphas[:, tf.newaxis, tf.newaxis, tf.newaxis]
    baseline_x = tf.expand_dims(baseline, axis=0)
    input_x = tf.expand_dims(image, axis=0)
    delta = input_x - baseline_x
    images = baseline_x + alphas_x * delta
    return images
```

Let's use the above function to generate interpolated images along a linear path at alpha intervals between a black baseline image and the example "Fireboat" image.

In []:

```
interpolated_images = interpolate_images(
    baseline=baseline,
    image=img_name_tensors['Fireboat'],
    alphas=alphas)
```

Let's visualize the interpolated images. Note: another way of thinking about the α constant is that it is consistently increasing each interpolated image's intensity.

In []:

```
fig = plt.figure(figsize=(20, 20))

i = 0
for alpha, image in zip(alphas[0::10], interpolated_images[0::10]):
    i += 1
    plt.subplot(1, len(alphas[0::10]), i)
    plt.title(f'alpha: {alpha:.1f}')
    plt.imshow(image)
    plt.axis('off')

plt.tight_layout();
```

Compute gradients

Now let's take a look at how to calculate gradients in order to measure the relationship between changes to a feature and changes in the model's predictions. In the case of images, the gradient tells us which pixels have the strongest effect on the models predicted class probabilities.

$$IntegratedGrads_i^{approx}(x) := (x_i - x_i') \times \sum_{k=1}^m \frac{\partial F(\text{interpolated images})}{\partial x_i} \times \frac{1}{m}$$

where:

$F()$ = your model's prediction function

$\frac{\partial F}{\partial x_i}$ = gradient (vector of partial derivatives ∂) of your model F 's prediction function relative to each feature x_i

TensorFlow makes computing gradients easy for you with a `tf.GradientTape` (https://www.tensorflow.org/api_docs/python/tf/GradientTape).

In []:

```
def compute_gradients(images, target_class_idx):
    with tf.GradientTape() as tape:
        tape.watch(images)
        logits = model(images)
        probs = tf.nn.softmax(logits, axis=-1)[:, target_class_idx]
    return tape.gradient(probs, images)
```

Let's compute the gradients for each image along the interpolation path with respect to the correct output. Recall that your model returns a `(1, 1001)` shaped `Tensor` with logits that you convert to predicted probabilities for each class. You need to pass the correct ImageNet target class index to the `compute_gradients` function for your image.

In []:

```
path_gradients = compute_gradients(
    images=interpolated_images,
    target_class_idx=555)
```

Note the output shape of `(n_interpolated_images, img_height, img_width, RGB)`, which gives us the gradient for every pixel of every image along the interpolation path. You can think of these gradients as measuring the change in your model's predictions for each small step in the feature space.

In []:

```
print(path_gradients.shape)
```

Visualizing gradient saturation

Recall that the gradients you just calculated above describe *local* changes to your model's predicted probability of "Fireboat" and can *saturate*.

These concepts are visualized using the gradients you calculated above in the 2 plots below.

In []:

```
pred = model(interpolated_images)
pred_proba = tf.nn.softmax(pred, axis=-1)[:, 555]
```

In []:

```
#@title
plt.figure(figsize=(10, 4))
ax1 = plt.subplot(1, 2, 1)
ax1.plot(alphas, pred_proba)
ax1.set_title('Target class predicted probability over alpha')
ax1.set_ylabel('model p(target class)')
ax1.set_xlabel('alpha')
ax1.set_xlim([0, 1])

ax2 = plt.subplot(1, 2, 2)
# Average across interpolation steps
average_grads = tf.reduce_mean(path_gradients, axis=[1, 2, 3])
# Normalize gradients to 0 to 1 scale. E.g. (x - min(x))/(max(x)-min(x))
average_grads_norm = (average_grads - tf.math.reduce_min(average_grads)) / (tf.math.reduce_max(average_grads) - tf.math.reduce_min(average_grads))
ax2.plot(alphas, average_grads_norm)
ax2.set_title('Average pixel gradients (normalized) over alpha')
ax2.set_ylabel('Average pixel gradients')
ax2.set_xlabel('alpha')
ax2.set_xlim([0, 1]);
```

- **left:** This plot shows how your model's confidence in the "Fireboat" class varies across alphas. Notice how the gradients, or slope of the line, largely flattens or saturates between 0.6 and 1.0 before settling at the final "Fireboat" predicted probability of about 40%.
- **right:** The right plot shows the average gradients magnitudes over alpha more directly. Note how the values sharply approach and even briefly dip below zero. In fact, your model "learns" the most from gradients at lower values of alpha before saturating. Intuitively, you can think of this as your model has learned the pixels e.g. water cannons to make the correct prediction, sending these pixels gradients to zero, but is still quite uncertain and focused on spurious bridge or water jet pixels as the alpha values approach the original input image.

To make sure these important water cannon pixels are reflected as important to the "Fireboat" prediction, you will continue on below to learn how to accumulate these gradients to accurately approximate how each pixel impacts your "Fireboat" predicted probability.

Accumulate gradients (integral approximation)

There are many different ways you can go about computing the numerical approximation of an integral for IG with different tradeoffs in accuracy and convergence across varying functions. A popular class of methods is called [Riemann sums](https://en.wikipedia.org/wiki/Riemann_sum) (https://en.wikipedia.org/wiki/Riemann_sum). Here, you will use the Trapezoidal rule (you can find additional code to explore different approximation methods at the end of this tutorial).

$$\text{IntegratedGrads}_i^{\text{approx}}(x) := (x_i - x_{i-1}) \times \sum_{k=1}^m \text{gradients(interpolated images)} \times \frac{1}{m}$$

From the equation, you can see you are summing over m gradients and dividing by m steps. You can implement the two operations together for part 3 as an *average of the local gradients of m interpolated predictions and input images*.

In []:

```
def integral_approximation(gradients):
    # riemann_trapezoidal
    grads = (gradients[:-1] + gradients[1:]) / tf.constant(2.0)
    integrated_gradients = tf.math.reduce_mean(grads, axis=0)
    return integrated_gradients
```

The `integral_approximation` function takes the gradients of the predicted probability of the target class with respect to the interpolated images between the baseline and the original image.

In []:

```
ig = integral_approximation(  
    gradients=path_gradients)
```

You can confirm averaging across the gradients of m interpolated images returns an integrated gradients tensor with the same shape as the original "Giant Panda" image.

In []:

```
print(ig.shape)
```

Putting it all together

Now you will combine the 3 previous general parts together into an `IntegratedGradients` function and utilize a `@tf.function` (<https://www.tensorflow.org/guide/function>) decorator to compile it into a high performance callable TensorFlow graph. This is implemented as 5 smaller steps below:

$$\text{IntegratedGrads}_i^{\text{approx}}(x) := (x_i - x'_i) \times \sum_{k=1}^m \frac{\partial F(x' + \frac{k}{m} \times (x - x'))}{\partial x_i} \times \frac{1}{m}$$

1. Generate alphas α
2. Generate interpolated images $= (x' + \frac{k}{m} \times (x - x'))$
3. Compute gradients between model F output predictions with respect to input features $= \frac{\partial F(\text{interpolated path inputs})}{\partial x_i}$
4. Integral approximation through averaging gradients $= \sum_{k=1}^m \text{gradients} \times \frac{1}{m}$
5. Scale integrated gradients with respect to original image $= (x_i - x'_i) \times \text{integrated gradients}$. The reason this step is necessary is to make sure that the attribution values accumulated across multiple interpolated images are in the same units and faithfully represent the pixel importances on the original image.

In []:

```
def integrated_gradients(baseline,  
                        image,  
                        target_class_idx,  
                        m_steps=50,  
                        batch_size=32):  
  
    # Generate alphas.  
    alphas = tf.linspace(start=0.0, stop=1.0, num=m_steps+1)  
  
    # Collect gradients.  
    gradient_batches = []  
  
    # Iterate alphas range and batch computation for speed, memory efficiency, and scaling to larger m_steps.  
    for alpha in tf.range(0, len(alphas), batch_size):  
        from_ = alpha  
        to = tf.minimum(from_ + batch_size, len(alphas))  
        alpha_batch = alphas[from_:to]  
  
        gradient_batch = one_batch(baseline, image, alpha_batch, target_class_idx)  
        gradient_batches.append(gradient_batch)  
  
    # Concatenate path gradients together row-wise into single tensor.  
    total_gradients = tf.concat(gradient_batches, axis=0)  
  
    # Integral approximation through averaging gradients.  
    avg_gradients = integral_approximation(gradients=total_gradients)  
  
    # Scale integrated gradients with respect to input.  
    integrated_gradients = (image - baseline) * avg_gradients  
  
    return integrated_gradients
```

In []:

```
@tf.function
def one_batch(baseline, image, alpha_batch, target_class_idx):
    # Generate interpolated inputs between baseline and input.
    interpolated_path_input_batch = interpolate_images(baseline=baseline,
                                                        image=image,
                                                        alphas=alpha_batch)

    # Compute gradients between model outputs and interpolated inputs.
    gradient_batch = compute_gradients(images=interpolated_path_input_batch,
                                         target_class_idx=target_class_idx)
    return gradient_batch
```

In []:

```
ig_attributions = integrated_gradients(baseline=baseline,
                                         image=img_name_tensors['Fireboat'],
                                         target_class_idx=555,
                                         m_steps=240)
```

Again, you can check that the IG feature attributions have the same shape as the input "Fireboat" image.

In []:

```
print(ig_attributions.shape)
```

The paper suggests the number of steps to range between 20 to 300 depending upon the example (although in practice this can be higher in the 1,000s to accurately approximate the integral). You can find additional code to check for the appropriate number of steps in the "Next steps" resources at the end of this tutorial.

Visualize attributions

You are ready to visualize attributions, and overlay them on the original image. The code below sums the absolute values of the integrated gradients across the color channels to produce an attribution mask. This plotting method captures the relative impact of pixels on the model's predictions.

In []:

```
#@title
def plot_img_attributions(baseline,
                           image,
                           target_class_idx,
                           m_steps=50,
                           cmap=None,
                           overlay_alpha=0.4):

    attributions = integrated_gradients(baseline=baseline,
                                         image=image,
                                         target_class_idx=target_class_idx,
                                         m_steps=m_steps)

    # Sum of the attributions across color channels for visualization.
    # The attribution mask shape is a grayscale image with height and width
    # equal to the original image.
    attribution_mask = tf.reduce_sum(tf.math.abs(attributions), axis=-1)

    fig, axs = plt.subplots(nrows=2, ncols=2, squeeze=False, figsize=(8, 8))

    axs[0, 0].set_title('Baseline image')
    axs[0, 0].imshow(baseline)
    axs[0, 0].axis('off')

    axs[0, 1].set_title('Original image')
    axs[0, 1].imshow(image)
    axs[0, 1].axis('off')

    axs[1, 0].set_title('Attribution mask')
    axs[1, 0].imshow(attribution_mask, cmap=cmap)
    axs[1, 0].axis('off')

    axs[1, 1].set_title('Overlay')
    axs[1, 1].imshow(attribution_mask, cmap=cmap)
    axs[1, 1].imshow(image, alpha=overlay_alpha)
    axs[1, 1].axis('off')

    plt.tight_layout()
    return fig
```

Looking at the attributions on the "Fireboat" image, you can see the model identifies the water cannons and spouts as contributing to its correct prediction.

In []:

```
_ = plot_img_attributions(image=img_name_tensors['Fireboat'],
                           baseline=baseline,
                           target_class_idx=555,
                           m_steps=240,
                           cmap=plt.cm.inferno,
                           overlay_alpha=0.4)
```

On the "Giant Panda" image, the attributions highlight the texture, nose, and the fur of the Panda's face.

In []:

```
_ = plot_img_attributions(image=img_name_tensors['Giant Panda'],
                           baseline=baseline,
                           target_class_idx=389,
                           m_steps=55,
                           cmap=plt.cm.viridis,
                           overlay_alpha=0.5)
```

Uses and limitations

Use cases

- Employing techniques like Integrated Gradients before deploying your model can help you develop intuition for how and why it works. Do the features highlighted by this technique match your intuition? If not, that may be indicative of a bug in your model or dataset, or overfitting.

Limitations

- Integrated Gradients provides feature importances on individual examples, however, it does not provide global feature importances across an entire dataset.
- Integrated Gradients provides individual feature importances, but it does not explain feature interactions and combinations.

Next steps

This tutorial presented a basic implementation of Integrated Gradients. As a next step, you can use this notebook to try this technique with different models and images yourself.

For interested readers, there is a lengthier version of this tutorial (which includes code for different baselines, to compute integral approximations, and to determine a sufficient number of steps) which you can find [here \(\[https://github.com/GoogleCloudPlatform/training-data-analyst/tree/master/blogs/integrated_gradients\]\(https://github.com/GoogleCloudPlatform/training-data-analyst/tree/master/blogs/integrated_gradients\)\)](https://github.com/GoogleCloudPlatform/training-data-analyst/tree/master/blogs/integrated_gradients).

To deepen your understanding, check out the paper [Axiomatic Attribution for Deep Networks \(<https://arxiv.org/abs/1703.01365>\)](https://arxiv.org/abs/1703.01365) and [Github repository \(<https://github.com/ankurtaly/Integrated-Gradients>\)](https://github.com/ankurtaly/Integrated-Gradients), which contains an implementation in a previous version of TensorFlow. You can also explore feature attribution, and the impact of different baselines, on [distill.pub \(<https://distill.pub/2020/attribution-baselines/>\)](https://distill.pub/2020/attribution-baselines/).

Interested in incorporating IG into your production machine learning workflows for feature importances, model error analysis, and data skew monitoring? Check out Google Cloud's [Explainable AI \(<https://cloud.google.com/explainable-ai>\)](https://cloud.google.com/explainable-ai) product that supports IG attributions. The Google AI PAIR research group also open-sourced the [What-if tool \(<https://pair-code.github.io/what-if-tool/index.html#about>\)](https://pair-code.github.io/what-if-tool/index.html#about) which can be used for model debugging, including visualizing IG feature attributions.

Copyright 2020 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Load and preprocess images



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/load_data/images)

(https://www.tensorflow.org/tutorials/load_data/images)



[Run in Google Colab](#)

(https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/load_data/images.ipynb)

This tutorial shows how to load and preprocess an image dataset in three ways:

- First, you will use high-level Keras preprocessing utilities (such as `tf.keras.utils.image_dataset_from_directory`) and layers (such as `tf.keras.layers.Rescaling`) to read a directory of images on disk.
- Next, you will write your own input pipeline from scratch [using tf.data \(../guide/data.ipynb\)](#).
- Finally, you will download a dataset from the large [catalog](#) (<https://www.tensorflow.org/datasets/catalog/overview>) available in [TensorFlow Datasets](#) (<https://www.tensorflow.org/datasets>).

Setup

In []:

```
import numpy as np
import os
import PIL
import PIL.Image
import tensorflow as tf
import tensorflow_datasets as tfds
```

In []:

```
print(tf.__version__)
```

Download the flowers dataset

This tutorial uses a dataset of several thousand photos of flowers. The flowers dataset contains five sub-directories, one per class:

```
flowers_photos/
daisy/
dandelion/
roses/
sunflowers/
tulips/
```

Note: all images are licensed CC-BY, creators are listed in the LICENSE.txt file.

In []:

```
import pathlib
dataset_url = "https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz"
data_dir = tf.keras.utils.get_file(origin=dataset_url,
                                    fname='flower_photos',
                                    untar=True)
data_dir = pathlib.Path(data_dir)
```

After downloading (218MB), you should now have a copy of the flower photos available. There are 3,670 total images:

In []:

```
image_count = len(list(data_dir.glob('*/*.jpg')))
print(image_count)
```

Each directory contains images of that type of flower. Here are some roses:

In []:

```
roses = list(data_dir.glob('roses/*'))
PIL.Image.open(str(roses[0]))
```

In []:

```
roses = list(data_dir.glob('roses/*'))
PIL.Image.open(str(roses[1]))
```

Load data using a Keras utility

Let's load these images off disk using the helpful `tf.keras.utils.image_dataset_from_directory` utility.

Create a dataset

Define some parameters for the loader:

In []:

```
batch_size = 32
img_height = 180
img_width = 180
```

It's good practice to use a validation split when developing your model. You will use 80% of the images for training and 20% for validation.

In []:

```
train_ds = tf.keras.utils.image_dataset_from_directory(
    data_dir,
    validation_split=0.2,
    subset="training",
    seed=123,
    image_size=(img_height, img_width),
    batch_size=batch_size)
```

In []:

```
val_ds = tf.keras.utils.image_dataset_from_directory(
    data_dir,
    validation_split=0.2,
    subset="validation",
    seed=123,
    image_size=(img_height, img_width),
    batch_size=batch_size)
```

You can find the class names in the `class_names` attribute on these datasets.

In []:

```
class_names = train_ds.class_names
print(class_names)
```

Visualize the data

Here are the first nine images from the training dataset.

In []:

```
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 10))
for images, labels in train_ds.take(1):
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(images[i].numpy().astype("uint8"))
        plt.title(class_names[labels[i]])
        plt.axis("off")
```

You can train a model using these datasets by passing them to `model.fit` (shown later in this tutorial). If you like, you can also manually iterate over the dataset and retrieve batches of images:

In []:

```
for image_batch, labels_batch in train_ds:
    print(image_batch.shape)
    print(labels_batch.shape)
    break
```

The `image_batch` is a tensor of the shape `(32, 180, 180, 3)`. This is a batch of 32 images of shape `180x180x3` (the last dimension refers to color channels RGB). The `label_batch` is a tensor of the shape `(32,)`, these are corresponding labels to the 32 images.

You can call `.numpy()` on either of these tensors to convert them to a `numpy.ndarray`.

Standardize the data

The RGB channel values are in the `[0, 255]` range. This is not ideal for a neural network; in general you should seek to make your input values small.

Here, you will standardize values to be in the `[0, 1]` range by using `tf.keras.layers.Rescaling`:

In []:

```
normalization_layer = tf.keras.layers.Rescaling(1./255)
```

There are two ways to use this layer. You can apply it to the dataset by calling `Dataset.map`:

In []:

```
normalized_ds = train_ds.map(lambda x, y: (normalization_layer(x), y))
image_batch, labels_batch = next(iter(normalized_ds))
first_image = image_batch[0]
# Notice the pixel values are now in `[0,1]`.
print(np.min(first_image), np.max(first_image))
```

Or, you can include the layer inside your model definition to simplify deployment. You will use the second approach here.

Note: If you would like to scale pixel values to `[-1,1]` you can instead write `tf.keras.layers.Rescaling(1./127.5, offset=-1)`

Note: You previously resized images using the `image_size` argument of `tf.keras.utils.image_dataset_from_directory`. If you want to include the resizing logic in your model as well, you can use the `tf.keras.layers.Resizing` layer.

Configure the dataset for performance

Let's make sure to use buffered prefetching so you can yield data from disk without having I/O become blocking. These are two important methods you should use when loading data:

- `Dataset.cache` keeps the images in memory after they're loaded off disk during the first epoch. This will ensure the dataset does not become a bottleneck while training your model. If your dataset is too large to fit into memory, you can also use this method to create a performant on-disk cache.
- `Dataset.prefetch` overlaps data preprocessing and model execution while training.

Interested readers can learn more about both methods, as well as how to cache data to disk in the *Prefetching* section of the [Better performance with the tf.data API](#) ([../guide/data_performance.ipynb](#)) guide.

In []:

```
AUTOTUNE = tf.data.AUTOTUNE

train_ds = train_ds.cache().prefetch(buffer_size=AUTOTUNE)
val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)
```

Train a model

For completeness, you will show how to train a simple model using the datasets you have just prepared.

The [Sequential](#) (https://www.tensorflow.org/guide/keras/sequential_model) model consists of three convolution blocks (`tf.keras.layers.Conv2D`) with a max pooling layer (`tf.keras.layers.MaxPooling2D`) in each of them. There's a fully-connected layer (`tf.keras.layers.Dense`) with 128 units on top of it that is activated by a ReLU activation function (`'relu'`). This model has not been tuned in any way—the goal is to show you the mechanics using the datasets you just created. To learn more about image classification, visit the [Image classification](#) ([../images/classification.ipynb](#)) tutorial.

```
In [ ]:
```

```
num_classes = 5

model = tf.keras.Sequential([
    tf.keras.layers.Rescaling(1./255),
    tf.keras.layers.Conv2D(32, 3, activation='relu'),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Conv2D(32, 3, activation='relu'),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Conv2D(32, 3, activation='relu'),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(num_classes)
])
```

Choose the `tf.keras.optimizers.Adam` optimizer and `tf.keras.losses.SparseCategoricalCrossentropy` loss function. To view training and validation accuracy for each training epoch, pass the `metrics` argument to `Model.compile`.

```
In [ ]:
```

```
model.compile(
    optimizer='adam',
    loss=tf.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=['accuracy'])
```

Note: You will only train for a few epochs so this tutorial runs quickly.

```
In [ ]:
```

```
model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=3
)
```

Note: You can also write a custom training loop instead of using `Model.fit`. To learn more, visit the [Writing a training loop from scratch](https://www.tensorflow.org/guide/keras/writing_a_training_loop_from_scratch) (https://www.tensorflow.org/guide/keras/writing_a_training_loop_from_scratch) tutorial.

You may notice the validation accuracy is low compared to the training accuracy, indicating your model is overfitting. You can learn more about overfitting and how to reduce it in this [tutorial](https://www.tensorflow.org/tutorials/keras/overfit_and_underfit) (https://www.tensorflow.org/tutorials/keras/overfit_and_underfit).

Using `tf.data` for finer control

The above Keras preprocessing utility—`tf.keras.utils.image_dataset_from_directory`—is a convenient way to create a `tf.data.Dataset` from a directory of images.

For finer grain control, you can write your own input pipeline using `tf.data`. This section shows how to do just that, beginning with the file paths from the TGZ file you downloaded earlier.

```
In [ ]:
```

```
list_ds = tf.data.Dataset.list_files(str(data_dir/'*/*'), shuffle=False)
list_ds = list_ds.shuffle(image_count, reshuffle_each_iteration=False)
```

```
In [ ]:
```

```
for f in list_ds.take(5):
    print(f.numpy())
```

The tree structure of the files can be used to compile a `class_names` list.

```
In [ ]:
```

```
class_names = np.array(sorted([item.name for item in data_dir.glob('*') if item.name != "LICENSE.txt"]))
print(class_names)
```

Split the dataset into training and validation sets:

```
In [ ]:
```

```
val_size = int(image_count * 0.2)
train_ds = list_ds.skip(val_size)
val_ds = list_ds.take(val_size)
```

You can print the length of each dataset as follows:

```
In [ ]:
```

```
print(tf.data.experimental.cardinality(train_ds).numpy())
print(tf.data.experimental.cardinality(val_ds).numpy())
```

Write a short function that converts a file path to an (img, label) pair:

```
In [ ]:
```

```
def get_label(file_path):
    # Convert the path to a list of path components
    parts = tf.strings.split(file_path, os.path.sep)
    # The second to last is the class-directory
    one_hot = parts[-2] == class_names
    # Integer encode the label
    return tf.argmax(one_hot)
```

```
In [ ]:
```

```
def decode_img(img):
    # Convert the compressed string to a 3D uint8 tensor
    img = tf.io.decode_jpeg(img, channels=3)
    # Resize the image to the desired size
    return tf.image.resize(img, [img_height, img_width])
```

```
In [ ]:
```

```
def process_path(file_path):
    label = get_label(file_path)
    # Load the raw data from the file as a string
    img = tf.io.read_file(file_path)
    img = decode_img(img)
    return img, label
```

Use `Dataset.map` to create a dataset of `image, label` pairs:

```
In [ ]:
```

```
# Set `num_parallel_calls` so multiple images are loaded/processed in parallel.
train_ds = train_ds.map(process_path, num_parallel_calls=AUTOTUNE)
val_ds = val_ds.map(process_path, num_parallel_calls=AUTOTUNE)
```

```
In [ ]:
```

```
for image, label in train_ds.take(1):
    print("Image shape: ", image.numpy().shape)
    print("Label: ", label.numpy())
```

Configure dataset for performance

To train a model with this dataset you will want the data:

- To be well shuffled.
- To be batched.
- Batches to be available as soon as possible.

These features can be added using the `tf.data` API. For more details, visit the [Input Pipeline Performance \(../../guide/performance/datasets.ipynb\)](#) guide.

In []:

```
def configure_for_performance(ds):
    ds = ds.cache()
    ds = ds.shuffle(buffer_size=1000)
    ds = ds.batch(batch_size)
    ds = ds.prefetch(buffer_size=AUTOTUNE)
    return ds

train_ds = configure_for_performance(train_ds)
val_ds = configure_for_performance(val_ds)
```

Visualize the data

You can visualize this dataset similarly to the one you created previously:

In []:

```
image_batch, label_batch = next(iter(train_ds))

plt.figure(figsize=(10, 10))
for i in range(9):
    ax = plt.subplot(3, 3, i + 1)
    plt.imshow(image_batch[i].numpy().astype("uint8"))
    label = label_batch[i]
    plt.title(class_names[label])
    plt.axis("off")
```

Continue training the model

You have now manually built a similar `tf.data.Dataset` to the one created by `tf.keras.utils.image_dataset_from_directory` above. You can continue training the model with it. As before, you will train for just a few epochs to keep the running time short.

In []:

```
model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=3
)
```

Using TensorFlow Datasets

So far, this tutorial has focused on loading data off disk. You can also find a dataset to use by exploring the large [catalog](#) (<https://www.tensorflow.org/datasets/catalog/overview>) of easy-to-download datasets at [TensorFlow Datasets](#) (<https://www.tensorflow.org/datasets>).

As you have previously loaded the Flowers dataset off disk, let's now import it with TensorFlow Datasets.

Download the Flowers [dataset](#) (https://www.tensorflow.org/datasets/catalog/tf_flowers) using TensorFlow Datasets:

In []:

```
(train_ds, val_ds, test_ds), metadata = tfds.load(
    'tf_flowers',
    split=['train[:80%]', 'train[80%:90%]', 'train[90%:]'],
    with_info=True,
    as_supervised=True,
)
```

The flowers dataset has five classes:

In []:

```
num_classes = metadata.features['label'].num_classes
print(num_classes)
```

Retrieve an image from the dataset:

In []:

```
get_label_name = metadata.features['label'].int2str  
image, label = next(iter(train_ds))  
_ = plt.imshow(image)  
_ = plt.title(get_label_name(label))
```

As before, remember to batch, shuffle, and configure the training, validation, and test sets for performance:

In []:

```
train_ds = configure_for_performance(train_ds)  
val_ds = configure_for_performance(val_ds)  
test_ds = configure_for_performance(test_ds)
```

You can find a complete example of working with the Flowers dataset and TensorFlow Datasets by visiting the [Data augmentation \(./images/data_augmentation.ipynb\)](#) tutorial.

Next steps

This tutorial showed two ways of loading images off disk. First, you learned how to load and preprocess an image dataset using Keras preprocessing layers and utilities. Next, you learned how to write an input pipeline from scratch using `tf.data`. Finally, you learned how to download a dataset from TensorFlow Datasets.

For your next steps:

- You can learn [how to add data augmentation \(\[https://www.tensorflow.org/tutorials/images/data_augmentation\]\(https://www.tensorflow.org/tutorials/images/data_augmentation\)\)](#).
- To learn more about `tf.data`, you can visit the [tf.data: Build TensorFlow input pipelines \(<https://www.tensorflow.org/guide/data>\)](#) guide.

Copyright 2019 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");  
# you may not use this file except in compliance with the License.  
# You may obtain a copy of the License at  
#  
# https://www.apache.org/licenses/LICENSE-2.0  
#  
# Unless required by applicable law or agreed to in writing, software  
# distributed under the License is distributed on an "AS IS" BASIS,  
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
# See the License for the specific language governing permissions and  
# limitations under the License.
```

TFRecord and tf.train.Example



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/load_data/tfrecord)
(https://www.tensorflow.org/tutorials/load_data/tfrecord)



[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/load_data/tfrecord.ipynb)
(https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/load_data/tfrecord.ipynb)

The TFRecord format is a simple format for storing a sequence of binary records.

[Protocol buffers \(<https://developers.google.com/protocol-buffers/>\)](https://developers.google.com/protocol-buffers/) are a cross-platform, cross-language library for efficient serialization of structured data.

Protocol messages are defined by `.proto` files, these are often the easiest way to understand a message type.

The `tf.train.Example` message (or protobuf) is a flexible message type that represents a `{"string": value}` mapping. It is designed for use with TensorFlow and is used throughout the higher-level APIs such as [TFX \(<https://www.tensorflow.org/tfx/>\)](#).

This notebook demonstrates how to create, parse, and use the `tf.train.Example` message, and then serialize, write, and read `tf.train.Example` messages to and from `.tfrecord` files.

Note: While useful, these structures are optional. There is no need to convert existing code to use TFRecords, unless you are [using `tf.data`](https://www.tensorflow.org/guide/data) (<https://www.tensorflow.org/guide/data>) and reading data is still the bottleneck to training. You can refer to [Better performance with the `tf.data` API](https://www.tensorflow.org/guide/data_performance) (https://www.tensorflow.org/guide/data_performance) for dataset performance tips.

Note: In general, you should shard your data across multiple files so that you can parallelize I/O (within a single host or across multiple hosts). The rule of thumb is to have at least 10 times as many files as there will be hosts reading data. At the same time, each file should be large enough (at least 10 MB+ and ideally 100 MB+) so that you can benefit from I/O prefetching. For example, say you have X GB of data and you plan to train on up to N hosts. Ideally, you should shard the data to $\sim 10^*N$ files, as long as $\sim X/(10^*N)$ is 10 MB+ (and ideally 100 MB+). If it is less than that, you might need to create fewer shards to trade off parallelism benefits and I/O prefetching benefits.

Setup

In []:

```
import tensorflow as tf
import numpy as np
import IPython.display as display
```

tf.train.Example

Data types for `tf.train.Example`

Fundamentally, a `tf.train.Example` is a `{"string": tf.train.Feature}` mapping.

The `tf.train.Feature` message type can accept one of the following three types (See the [.proto file](#) (<https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/example/feature.proto>) for reference). Most other generic types can be coerced into one of these:

1. `tf.train.BytesList` (the following types can be coerced)

- `string`
- `byte`

2. `tf.train.FloatList` (the following types can be coerced)

- `float` (`float32`)
- `double` (`float64`)

3. `tf.train.Int64List` (the following types can be coerced)

- `bool`
- `enum`
- `int32`
- `uint32`
- `int64`
- `uint64`

In order to convert a standard TensorFlow type to a `tf.train.Example`-compatible `tf.train.Feature`, you can use the shortcut functions below. Note that each function takes a scalar input value and returns a `tf.train.Feature` containing one of the three `list` types above:

In []:

```
# The following functions can be used to convert a value to a type compatible
# with tf.train.Example.
```

```
def _bytes_feature(value):
    """Returns a bytes_list from a string / byte."""
    if isinstance(value, type(tf.constant(0))):
        value = value.numpy() # BytesList won't unpack a string from an EagerTensor.
    return tf.train.Feature(bytes_list=tf.train.BytesList(value=[value]))

def _float_feature(value):
    """Returns a float_list from a float / double."""
    return tf.train.Feature(float_list=tf.train.FloatList(value=[value]))

def _int64_feature(value):
    """Returns an int64_list from a bool / enum / int / uint."""
    return tf.train.Feature(int64_list=tf.train.Int64List(value=[value]))
```

Note: To stay simple, this example only uses scalar inputs. The simplest way to handle non-scalar features is to use `tf.io.serialize_tensor` to convert tensors to binary-strings. Strings are scalars in TensorFlow. Use `tf.io.parse_tensor` to convert the binary-string back to a tensor.

Below are some examples of how these functions work. Note the varying input types and the standardized output types. If the input type for a function does not match one of the coercible types stated above, the function will raise an exception (e.g. `_int64_feature(1.0)` will error out because `1.0` is a float—therefore, it should be used with the `_float_feature` function instead):

In []:

```
print(_bytes_feature(b'test_string'))
print(_bytes_feature(u'test_bytes'.encode('utf-8')))

print(_float_feature(np.exp(1)))

print(_int64_feature(True))
print(_int64_feature(1))
```

All proto messages can be serialized to a binary-string using the `.SerializeToString` method:

In []:

```
feature = _float_feature(np.exp(1))

feature.SerializeToString()
```

Creating a `tf.train.Example` message

Suppose you want to create a `tf.train.Example` message from existing data. In practice, the dataset may come from anywhere, but the procedure of creating the `tf.train.Example` message from a single observation will be the same:

1. Within each observation, each value needs to be converted to a `tf.train.Feature` containing one of the 3 compatible types, using one of the functions above.
2. You create a map (dictionary) from the feature name string to the encoded feature value produced in #1.
3. The map produced in step 2 is converted to a [Features message](#) (<https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/example/feature.proto#L85>).

In this notebook, you will create a dataset using NumPy.

This dataset will have 4 features:

- a boolean feature, `False` or `True` with equal probability
- an integer feature uniformly randomly chosen from `[0, 5]`
- a string feature generated from a string table by using the integer feature as an index
- a float feature from a standard normal distribution

Consider a sample consisting of 10,000 independently and identically distributed observations from each of the above distributions:

In []:

```
# The number of observations in the dataset.
n_observations = int(1e4)

# Boolean feature, encoded as False or True.
feature0 = np.random.choice([False, True], n_observations)

# Integer feature, random from 0 to 4.
feature1 = np.random.randint(0, 5, n_observations)

# String feature.
strings = np.array([b'cat', b'dog', b'chicken', b'horse', b'goat'])
feature2 = strings[feature1]

# Float feature, from a standard normal distribution.
feature3 = np.random.randn(n_observations)
```

Each of these features can be coerced into a `tf.train.Example`-compatible type using one of `_bytes_feature`, `_float_feature`, `_int64_feature`. You can then create a `tf.train.Example` message from these encoded features:

In []:

```
def serialize_example(feature0, feature1, feature2, feature3):
    """
    Creates a tf.train.Example message ready to be written to a file.
    """
    # Create a dictionary mapping the feature name to the tf.train.Example-compatible
    # data type.
    feature = {
        'feature0': _int64_feature(feature0),
        'feature1': _int64_feature(feature1),
        'feature2': _bytes_feature(feature2),
        'feature3': _float_feature(feature3),
    }

    # Create a Features message using tf.train.Example.

    example_proto = tf.train.Example(features=tf.train.Features(feature=feature))
    return example_proto.SerializeToString()
```

For example, suppose you have a single observation from the dataset, `[False, 4, bytes('goat'), 0.9876]`. You can create and print the `tf.train.Example` message for this observation using `create_message()`. Each single observation will be written as a `Features` message as per the above. Note that the `tf.train.Example` message (<https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/example/example.proto#L88>) is just a wrapper around the `Features` message:

In []:

```
# This is an example observation from the dataset.

example_observation = []
serialized_example = serialize_example(False, 4, b'goat', 0.9876)
serialized_example
```

To decode the message use the `tf.train.Example.FromString` method.

In []:

```
example_proto = tf.train.Example.FromString(serialized_example)
example_proto
```

TFRecords format details

A TFRecord file contains a sequence of records. The file can only be read sequentially.

Each record contains a byte-string, for the data-payload, plus the data-length, and CRC-32C ([32-bit CRC](https://en.wikipedia.org/wiki/Cyclic_redundancy_check#CRC-32_algorithm) (https://en.wikipedia.org/wiki/Cyclic_redundancy_check#CRC-32_algorithm) using the [Castagnoli polynomial](https://en.wikipedia.org/wiki/Cyclic_redundancy_check#Castagnoli_polynomial) (https://en.wikipedia.org/wiki/Cyclic_redundancy_check#Standards_and_common_use)) hashes for integrity checking.

Each record is stored in the following formats:

```
uint64 length
uint32 masked_crc32_of_length
byte   data[length]
uint32 masked_crc32_of_data
```

The records are concatenated together to produce the file. CRCs are [described here](https://en.wikipedia.org/wiki/Cyclic_redundancy_check) (https://en.wikipedia.org/wiki/Cyclic_redundancy_check), and the mask of a CRC is:

```
masked_crc = ((crc >> 15) | (crc << 17)) + 0xa282ead8ul
```

Note: There is no requirement to use `tf.train.Example` in TFRecord files. `tf.train.Example` is just a method of serializing dictionaries to byte-strings. Any byte-string that can be decoded in TensorFlow could be stored in a TFRecord file. Examples include: lines of text, JSON (using `tf.io.decode_json_example`), encoded image data, or serialized `tf.Tensors` (using `tf.io.serialize_tensor` / `tf.io.parse_tensor`). See the `tf.io` module for more options.

TFRecord files using `tf.data`

The `tf.data` module also provides tools for reading and writing data in TensorFlow.

Writing a TFRecord file

The easiest way to get the data into a dataset is to use the `from_tensor_slices` method.

Applied to an array, it returns a dataset of scalars:

In []:

```
tf.data.Dataset.from_tensor_slices(feature1)
```

Applied to a tuple of arrays, it returns a dataset of tuples:

In []:

```
features_dataset = tf.data.Dataset.from_tensor_slices((feature0, feature1, feature2, feature3))  
features_dataset
```

In []:

```
# Use `take(1)` to only pull one example from the dataset.  
for f0,f1,f2,f3 in features_dataset.take(1):  
    print(f0)  
    print(f1)  
    print(f2)  
    print(f3)
```

Use the `tf.data.Dataset.map` method to apply a function to each element of a `Dataset`.

The mapped function must operate in TensorFlow graph mode—it must operate on and return `tf.Tensors`. A non-tensor function, like `serialize_example`, can be wrapped with `tf.py_function` to make it compatible.

Using `tf.py_function` requires to specify the shape and type information that is otherwise unavailable:

In []:

```
def tf_serialize_example(f0,f1,f2,f3):  
    tf_string = tf.py_function(  
        serialize_example,  
        (f0, f1, f2, f3), # Pass these args to the above function.  
        tf.string) # The return type is `tf.string`.  
    return tf.reshape(tf_string, ()) # The result is a scalar.
```

In []:

```
tf_serialize_example(f0, f1, f2, f3)
```

Apply this function to each element in the dataset:

In []:

```
serialized_features_dataset = features_dataset.map(tf_serialize_example)  
serialized_features_dataset
```

In []:

```
def generator():  
    for features in features_dataset:  
        yield serialize_example(*features)
```

In []:

```
serialized_features_dataset = tf.data.Dataset.from_generator(  
    generator, output_types=tf.string, output_shapes=())
```

In []:

```
serialized_features_dataset
```

And write them to a TFRecord file:

In []:

```
filename = 'test.tfrecord'  
writer = tf.data.experimental.TFRecordWriter(filename)  
writer.write(serialized_features_dataset)
```

Reading a TFRecord file

You can also read the TFRecord file using the `tf.data.TFRecordDataset` class.

More information on consuming TFRecord files using `tf.data` can be found in the [tf.data: Build TensorFlow input pipelines \(`https://www.tensorflow.org/guide/data#consuming_tfrecord_data`\)](https://www.tensorflow.org/guide/data#consuming_tfrecord_data) guide.

Using `TFRecordDataset`s can be useful for standardizing input data and optimizing performance.

In []:

```
filenames = [filename]
raw_dataset = tf.data.TFRecordDataset(filenames)
raw_dataset
```

At this point the dataset contains serialized `tf.train.Example` messages. When iterated over it returns these as scalar string tensors.

Use the `.take` method to only show the first 10 records.

Note: iterating over a `tf.data.Dataset` only works with eager execution enabled.

In []:

```
for raw_record in raw_dataset.take(10):
    print(repr(raw_record))
```

These tensors can be parsed using the function below. Note that the `feature_description` is necessary here because `tf.data.Dataset`s use graph-execution, and need this description to build their shape and type signature:

In []:

```
# Create a description of the features.
feature_description = {
    'feature0': tf.io.FixedLenFeature([], tf.int64, default_value=0),
    'feature1': tf.io.FixedLenFeature([], tf.int64, default_value=0),
    'feature2': tf.io.FixedLenFeature([], tf.string, default_value=''),
    'feature3': tf.io.FixedLenFeature([], tf.float32, default_value=0.0),
}

def _parse_function(example_proto):
    # Parse the input `tf.train.Example` proto using the dictionary above.
    return tf.io.parse_single_example(example_proto, feature_description)
```

Alternatively, use `tf.parse_example` to parse the whole batch at once. Apply this function to each item in the dataset using the `tf.data.Dataset.map` method:

In []:

```
parsed_dataset = raw_dataset.map(_parse_function)
parsed_dataset
```

Use eager execution to display the observations in the dataset. There are 10,000 observations in this dataset, but you will only display the first 10. The data is displayed as a dictionary of features. Each item is a `tf.Tensor`, and the `numpy` element of this tensor displays the value of the feature:

In []:

```
for parsed_record in parsed_dataset.take(10):
    print(repr(parsed_record))
```

Here, the `tf.parse_example` function unpacks the `tf.train.Example` fields into standard tensors.

TFRecord files in Python

The `tf.io` module also contains pure-Python functions for reading and writing TFRecord files.

Writing a TFRecord file

Next, write the 10,000 observations to the file `test.tfrecord`. Each observation is converted to a `tf.train.Example` message, then written to file. You can then verify that the file `test.tfrecord` has been created:

```
In [ ]:
```

```
# Write the `tf.train.Example` observations to the file.
with tf.io.TFRecordWriter(filename) as writer:
    for i in range(n_observations):
        example = serialize_example(feature0[i], feature1[i], feature2[i], feature3[i])
        writer.write(example)
```

```
In [ ]:
```

```
!du -sh {filename}
```

Reading a TFRecord file

These serialized tensors can be easily parsed using `tf.train.Example.ParseFromString`:

```
In [ ]:
```

```
filenames = [filename]
raw_dataset = tf.data.TFRecordDataset(filenames)
raw_dataset
```

```
In [ ]:
```

```
for raw_record in raw_dataset.take(1):
    example = tf.train.Example()
    example.ParseFromString(raw_record.numpy())
    print(example)
```

That returns a `tf.train.Example` proto which is difficult to use as is, but it's fundamentally a representation of a:

```
Dict[str,
      Union[List[float],
            List[int],
            List[str]]]
```

The following code manually converts the `Example` to a dictionary of NumPy arrays, without using TensorFlow Ops. Refer to [the PROTO file](#) (<https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/example/feature.proto>) for details.

```
In [ ]:
```

```
result = {}
# example.features.feature is the dictionary
for key, feature in example.features.feature.items():
    # The values are the Feature objects which contain a `kind` which contains:
    # one of three fields: bytes_list, float_list, int64_list

    kind = feature.WhichOneof('kind')
    result[key] = np.array(getattr(feature, kind).value)

result
```

Walkthrough: Reading and writing image data

This is an end-to-end example of how to read and write image data using TFRecords. Using an image as input data, you will write the data as a TFRecord file, then read the file back and display the image.

This can be useful if, for example, you want to use several models on the same input dataset. Instead of storing the image data raw, it can be preprocessed into the TFRecords format, and that can be used in all further processing and modelling.

First, let's download [this image](https://commons.wikimedia.org/wiki/File:Felis_catus-cat_on_snow.jpg) (https://commons.wikimedia.org/wiki/File:Felis_catus-cat_on_snow.jpg) of a cat in the snow and [this photo](https://upload.wikimedia.org/wikipedia/commons/f/fe/New_East_River_Bridge_from_Brooklyn_det.4a09796u.jpg) (https://upload.wikimedia.org/wikipedia/commons/f/fe/New_East_River_Bridge_from_Brooklyn_det.4a09796u.jpg) of the Williamsburg Bridge, NYC under construction.

Fetch the images

In []:

```
cat_in_snow = tf.keras.utils.get_file(
    '320px-Felis_catus-cat_on_snow.jpg',
    'https://storage.googleapis.com/download.tensorflow.org/example_images/320px-Felis_catus-cat_on_snow.jpg')

williamsburg_bridge = tf.keras.utils.get_file(
    '194px-New_East_River_Bridge_from_Brooklyn_det.4a09796u.jpg',
    'https://storage.googleapis.com/download.tensorflow.org/example_images/194px-New_East_River_Bridge_from_Brooklyn_det.4a09796u.jpg')
```

In []:

```
display.display(display.Image(filename=cat_in_snow))
display.display(display.HTML('Image cc-by: <a "href=https://commons.wikimedia.org/wiki/File:Felis_catus-cat_on_snow.jpg">Von.grzanka</a>'))
```

In []:

```
display.display(display.Image(filename=williamsburg_bridge))
display.display(display.HTML('<a "href=https://commons.wikimedia.org/wiki/File>New_East_River_Bridge_from_Brooklyn_det.4a09796u.jpg">From Wikimedia</a>'))
```

Write the TFRecord file

As before, encode the features as types compatible with `tf.train.Example`. This stores the raw image string feature, as well as the height, width, depth, and arbitrary `label` feature. The latter is used when you write the file to distinguish between the cat image and the bridge image. Use `0` for the cat image, and `1` for the bridge image:

In []:

```
image_labels = {
    cat_in_snow : 0,
    williamsburg_bridge : 1,
}
```

In []:

```
# This is an example, just using the cat image.
image_string = open(cat_in_snow, 'rb').read()

label = image_labels[cat_in_snow]

# Create a dictionary with features that may be relevant.
def image_example(image_string, label):
    image_shape = tf.io.decode_jpeg(image_string).shape

    feature = {
        'height': _int64_feature(image_shape[0]),
        'width': _int64_feature(image_shape[1]),
        'depth': _int64_feature(image_shape[2]),
        'label': _int64_feature(label),
        'image_raw': _bytes_feature(image_string),
    }

    return tf.train.Example(features=tf.train.Features(feature=feature))

for line in str(image_example(image_string, label)).split('\n')[:15]:
    print(line)
print('...')
```

Notice that all of the features are now stored in the `tf.train.Example` message. Next, functionalize the code above and write the example messages to a file named `images.tfrecords`:

In []:

```
# Write the raw image files to `images.tfrecords`.
# First, process the two images into `tf.train.Example` messages.
# Then, write to a `.tfrecords` file.
record_file = 'images.tfrecords'
with tf.io.TFRecordWriter(record_file) as writer:
    for filename, label in image_labels.items():
        image_string = open(filename, 'rb').read()
        tf_example = image_example(image_string, label)
        writer.write(tf_example.SerializeToString())
```

In []:

```
!du -sh {record_file}
```

Read the TFRecord file

You now have the file—`images.tfrecords`—and can now iterate over the records in it to read back what you wrote. Given that in this example you will only reproduce the image, the only feature you will need is the raw image string. Extract it using the getters described above, namely `example.features.feature['image_raw'].bytes_list.value[0]`. You can also use the labels to determine which record is the cat and which one is the bridge:

In []:

```
raw_image_dataset = tf.data.TFRecordDataset('images.tfrecords')

# Create a dictionary describing the features.
image_feature_description = {
    'height': tf.io.FixedLenFeature([], tf.int64),
    'width': tf.io.FixedLenFeature([], tf.int64),
    'depth': tf.io.FixedLenFeature([], tf.int64),
    'label': tf.io.FixedLenFeature([], tf.int64),
    'image_raw': tf.io.FixedLenFeature([], tf.string),
}

def _parse_image_function(example_proto):
    # Parse the input tf.train.Example proto using the dictionary above.
    return tf.io.parse_single_example(example_proto, image_feature_description)

parsed_image_dataset = raw_image_dataset.map(_parse_image_function)
parsed_image_dataset
```

Recover the images from the TFRecord file:

In []:

```
for image_features in parsed_image_dataset:
    image_raw = image_features['image_raw'].numpy()
    display.display(display.Image(data=image_raw))
```

Copyright 2018 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Load text



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/load_data/text)

(https://www.tensorflow.org/tutorials/load_data/text) (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/load_data/text.ipynb) (https://github.com/tensorflow/tensorflow/blob/master/tensorflow/tutorials/load_data/text.ipynb)



[Run in Google Colab](#)

This tutorial demonstrates two ways to load and preprocess text.

- First, you will use Keras utilities and preprocessing layers. These include `tf.keras.utils.text_dataset_from_directory` to turn data into a `tf.data.Dataset` and `tf.keras.layers.TextVectorization` for data standardization, tokenization, and vectorization. If you are new to TensorFlow, you should start with these.
- Then, you will use lower-level utilities like `tf.data.TextLineDataset` to load text files, and [TensorFlow Text](https://www.tensorflow.org/text) (<https://www.tensorflow.org/text>) APIs, such as `text.UnicodeScriptTokenizer` and `text.case_fold_utf8`, to preprocess the data for finer-grain control.

```
In [ ]:
```

```
!pip install "tensorflow-text==2.8.*"
```

```
In [ ]:
```

```
import collections
import pathlib

import tensorflow as tf

from tensorflow.keras import layers
from tensorflow.keras import losses
from tensorflow.keras import utils
from tensorflow.keras.layers import TextVectorization

import tensorflow_datasets as tfds
import tensorflow_text as tf_text
```

Example 1: Predict the tag for a Stack Overflow question

As a first example, you will download a dataset of programming questions from Stack Overflow. Each question ("How do I sort a dictionary by value?") is labeled with exactly one tag (Python , CSharp , JavaScript , or Java). Your task is to develop a model that predicts the tag for a question. This is an example of multi-class classification—an important and widely applicable kind of machine learning problem.

Download and explore the dataset

Begin by downloading the Stack Overflow dataset using `tf.keras.utils.get_file`, and exploring the directory structure:

```
In [ ]:
```

```
data_url = 'https://storage.googleapis.com/download.tensorflow.org/data/stack_overflow_16k.tar.gz'

dataset_dir = utils.get_file(
    origin=data_url,
    untar=True,
    cache_dir='stack_overflow',
    cache_subdir='')

dataset_dir = pathlib.Path(dataset_dir).parent
```

```
In [ ]:
```

```
list(dataset_dir.iterdir())
```

```
In [ ]:
```

```
train_dir = dataset_dir/'train'
list(train_dir.iterdir())
```

The `train/csharp` , `train/java` , `train/python` and `train/javascript` directories contain many text files, each of which is a Stack Overflow question.

Print an example file and inspect the data:

```
In [ ]:
```

```
sample_file = train_dir/'python/1755.txt'

with open(sample_file) as f:
    print(f.read())
```

Load the dataset

Next, you will load the data off disk and prepare it into a format suitable for training. To do so, you will use the `tf.keras.utils.text_dataset_from_directory` utility to create a labeled `tf.data.Dataset`. If you're new to `tf.data`, it's a powerful collection of tools for building input pipelines. (Learn more in the [tf.data: Build TensorFlow input pipelines \(../../guide/data.ipynb\)](#) guide.)

The `tf.keras.utils.text_dataset_from_directory` API expects a directory structure as follows:

```
train/
...csharp/
.....1.txt
.....2.txt
...java/
.....1.txt
.....2.txt
...javascript/
.....1.txt
.....2.txt
...python/
.....1.txt
.....2.txt
```

When running a machine learning experiment, it is a best practice to divide your dataset into three splits: [training](#) (https://developers.google.com/machine-learning/glossary#training_set), [validation](#) (https://developers.google.com/machine-learning/glossary#validation_set), and [test](#) (<https://developers.google.com/machine-learning/glossary#test-set>).

The Stack Overflow dataset has already been divided into training and test sets, but it lacks a validation set.

Create a validation set using an 80:20 split of the training data by using `tf.keras.utils.text_dataset_from_directory` with `validation_split` set to `0.2` (i.e. 20%):

In []:

```
batch_size = 32
seed = 42

raw_train_ds = utils.text_dataset_from_directory(
    train_dir,
    batch_size=batch_size,
    validation_split=0.2,
    subset='training',
    seed=seed)
```

As the previous cell output suggests, there are 8,000 examples in the training folder, of which you will use 80% (or 6,400) for training. You will learn in a moment that you can train a model by passing a `tf.data.Dataset` directly to `Model.fit`.

First, iterate over the dataset and print out a few examples, to get a feel for the data.

Note: To increase the difficulty of the classification problem, the dataset author replaced occurrences of the words *Python*, *CSharp*, *JavaScript*, or *Java* in the programming question with the word *blank*.

In []:

```
for text_batch, label_batch in raw_train_ds.take(1):
    for i in range(10):
        print("Question: ", text_batch.numpy()[i])
        print("Label: ", label_batch.numpy()[i])
```

The labels are `0`, `1`, `2` or `3`. To check which of these correspond to which string label, you can inspect the `class_names` property on the dataset:

In []:

```
for i, label in enumerate(raw_train_ds.class_names):
    print("Label", i, "corresponds to", label)
```

Next, you will create a validation and a test set using `tf.keras.utils.text_dataset_from_directory`. You will use the remaining 1,600 reviews from the training set for validation.

Note: When using the `validation_split` and `subset` arguments of `tf.keras.utils.text_dataset_from_directory`, make sure to either specify a random seed or pass `shuffle=False`, so that the validation and training splits have no overlap.

```
In [ ]:
```

```
# Create a validation set.  
raw_val_ds = utils.text_dataset_from_directory(  
    train_dir,  
    batch_size=batch_size,  
    validation_split=0.2,  
    subset='validation',  
    seed=seed)
```

```
In [ ]:
```

```
test_dir = dataset_dir/'test'  
  
# Create a test set.  
raw_test_ds = utils.text_dataset_from_directory(  
    test_dir,  
    batch_size=batch_size)
```

Prepare the dataset for training

Next, you will standardize, tokenize, and vectorize the data using the `tf.keras.layers.TextVectorization` layer.

- *Standardization* refers to preprocessing the text, typically to remove punctuation or HTML elements to simplify the dataset.
- *Tokenization* refers to splitting strings into tokens (for example, splitting a sentence into individual words by splitting on whitespace).
- *Vectorization* refers to converting tokens into numbers so they can be fed into a neural network.

All of these tasks can be accomplished with this layer. (You can learn more about each of these in the `tf.keras.layers.TextVectorization` API docs.)

Note that:

- The default standardization converts text to lowercase and removes punctuation (`standardize='lower_and_strip_punctuation'`).
- The default tokenizer splits on whitespace (`split='whitespace'`).
- The default vectorization mode is '`int`' (`output_mode='int'`). This outputs integer indices (one per token). This mode can be used to build models that take word order into account. You can also use other modes—like '`binary`' —to build [bag-of-words](https://developers.google.com/machine-learning/glossary#bag-of-words) (<https://developers.google.com/machine-learning/glossary#bag-of-words>) models.

You will build two models to learn more about standardization, tokenization, and vectorization with `TextVectorization`:

- First, you will use the '`binary`' vectorization mode to build a bag-of-words model.
- Then, you will use the '`int`' mode with a 1D ConvNet.

```
In [ ]:
```

```
VOCAB_SIZE = 10000  
  
binary_vectorize_layer = TextVectorization(  
    max_tokens=VOCAB_SIZE,  
    output_mode='binary')
```

For the '`int`' mode, in addition to maximum vocabulary size, you need to set an explicit maximum sequence length (`MAX_SEQUENCE_LENGTH`), which will cause the layer to pad or truncate sequences to exactly `output_sequence_length` values:

```
In [ ]:
```

```
MAX_SEQUENCE_LENGTH = 250  
  
int_vectorize_layer = TextVectorization(  
    max_tokens=VOCAB_SIZE,  
    output_mode='int',  
    output_sequence_length=MAX_SEQUENCE_LENGTH)
```

Next, call `TextVectorization.adapt` to fit the state of the preprocessing layer to the dataset. This will cause the model to build an index of strings to integers.

Note: It's important to only use your training data when calling `TextVectorization.adapt`, as using the test set would leak information.

```
In [ ]:
```

```
# Make a text-only dataset (without labels), then call `TextVectorization.adapt`.  
train_text = raw_train_ds.map(lambda text, labels: text)  
binary_vectorize_layer.adapt(train_text)  
int_vectorize_layer.adapt(train_text)
```

Print the result of using these layers to preprocess data:

In []:

```
def binary_vectorize_text(text, label):
    text = tf.expand_dims(text, -1)
    return binary_vectorize_layer(text), label
```

In []:

```
def int_vectorize_text(text, label):
    text = tf.expand_dims(text, -1)
    return int_vectorize_layer(text), label
```

In []:

```
# Retrieve a batch (of 32 reviews and labels) from the dataset.
text_batch, label_batch = next(iter(raw_train_ds))
first_question, first_label = text_batch[0], label_batch[0]
print("Question", first_question)
print("Label", first_label)
```

In []:

```
print("'binary' vectorized question:",
      binary_vectorize_text(first_question, first_label)[0])
```

In []:

```
print("'int' vectorized question:",
      int_vectorize_text(first_question, first_label)[0])
```

As shown above, `TextVectorization`'s 'binary' mode returns an array denoting which tokens exist at least once in the input, while the 'int' mode replaces each token by an integer, thus preserving their order.

You can lookup the token (string) that each integer corresponds to by calling `TextVectorization.get_vocabulary` on the layer:

In []:

```
print("1289 ---> ", int_vectorize_layer.get_vocabulary()[1289])
print("313 ---> ", int_vectorize_layer.get_vocabulary()[313])
print("Vocabulary size: {}".format(len(int_vectorize_layer.get_vocabulary())))
```

You are nearly ready to train your model.

As a final preprocessing step, you will apply the `TextVectorization` layers you created earlier to the training, validation, and test sets:

In []:

```
binary_train_ds = raw_train_ds.map(binary_vectorize_text)
binary_val_ds = raw_val_ds.map(binary_vectorize_text)
binary_test_ds = raw_test_ds.map(binary_vectorize_text)

int_train_ds = raw_train_ds.map(int_vectorize_text)
int_val_ds = raw_val_ds.map(int_vectorize_text)
int_test_ds = raw_test_ds.map(int_vectorize_text)
```

Configure the dataset for performance

These are two important methods you should use when loading data to make sure that I/O does not become blocking.

- `Dataset.cache` keeps data in memory after it's loaded off disk. This will ensure the dataset does not become a bottleneck while training your model. If your dataset is too large to fit into memory, you can also use this method to create a performant on-disk cache, which is more efficient to read than many small files.
- `Dataset.prefetch` overlaps data preprocessing and model execution while training.

You can learn more about both methods, as well as how to cache data to disk in the *Prefetching* section of the [Better performance with the `tf.data` API](#) ([../guide/data_performance.ipynb](#)) guide.

In []:

```
AUTOTUNE = tf.data.AUTOTUNE

def configure_dataset(dataset):
    return dataset.cache().prefetch(buffer_size=AUTOTUNE)
```

```
In [ ]:
```

```
binary_train_ds = configure_dataset(binary_train_ds)
binary_val_ds = configure_dataset(binary_val_ds)
binary_test_ds = configure_dataset(binary_test_ds)

int_train_ds = configure_dataset(int_train_ds)
int_val_ds = configure_dataset(int_val_ds)
int_test_ds = configure_dataset(int_test_ds)
```

Train the model

It's time to create your neural network.

For the 'binary' vectorized data, define a simple bag-of-words linear model, then configure and train it:

```
In [ ]:
```

```
binary_model = tf.keras.Sequential([layers.Dense(4)])

binary_model.compile(
    loss=losses.SparseCategoricalCrossentropy(from_logits=True),
    optimizer='adam',
    metrics=['accuracy'])

history = binary_model.fit(
    binary_train_ds, validation_data=binary_val_ds, epochs=10)
```

Next, you will use the 'int' vectorized layer to build a 1D ConvNet:

```
In [ ]:
```

```
def create_model(vocab_size, num_labels):
    model = tf.keras.Sequential([
        layers.Embedding(vocab_size, 64, mask_zero=True),
        layers.Conv1D(64, 5, padding="valid", activation="relu", strides=2),
        layers.GlobalMaxPooling1D(),
        layers.Dense(num_labels)
    ])
    return model
```

```
In [ ]:
```

```
# `vocab_size` is `VOCAB_SIZE + 1` since `0` is used additionally for padding.
int_model = create_model(vocab_size=VOCAB_SIZE + 1, num_labels=4)
int_model.compile(
    loss=losses.SparseCategoricalCrossentropy(from_logits=True),
    optimizer='adam',
    metrics=['accuracy'])
history = int_model.fit(int_train_ds, validation_data=int_val_ds, epochs=5)
```

Compare the two models:

```
In [ ]:
```

```
print("Linear model on binary vectorized data:")
print(binary_model.summary())
```

```
In [ ]:
```

```
print("ConvNet model on int vectorized data:")
print(int_model.summary())
```

Evaluate both models on the test data:

```
In [ ]:
```

```
binary_loss, binary_accuracy = binary_model.evaluate(binary_test_ds)
int_loss, int_accuracy = int_model.evaluate(int_test_ds)

print("Binary model accuracy: {:.2%}".format(binary_accuracy))
print("Int model accuracy: {:.2%}".format(int_accuracy))
```

Note: This example dataset represents a rather simple classification problem. More complex datasets and problems bring out subtle but significant differences in preprocessing strategies and model architectures. Be sure to try out different hyperparameters and epochs to compare various approaches.

Export the model

In the code above, you applied `tf.keras.layers.TextVectorization` to the dataset before feeding text to the model. If you want to make your model capable of processing raw strings (for example, to simplify deploying it), you can include the `TextVectorization` layer inside your model.

To do so, you can create a new model using the weights you have just trained:

In []:

```
export_model = tf.keras.Sequential(
    [binary_vectorize_layer, binary_model,
     layers.Activation('sigmoid')])

export_model.compile(
    loss=losses.SparseCategoricalCrossentropy(from_logits=False),
    optimizer='adam',
    metrics=['accuracy'])

# Test it with `raw_test_ds`, which yields raw strings
loss, accuracy = export_model.evaluate(raw_test_ds)
print("Accuracy: {:.2%}".format(binary_accuracy))
```

Now, your model can take raw strings as input and predict a score for each label using `Model.predict`. Define a function to find the label with the maximum score:

In []:

```
def get_string_labels(predicted_scores_batch):
    predicted_int_labels = tf.argmax(predicted_scores_batch, axis=1)
    predicted_labels = tf.gather(raw_train_ds.class_names, predicted_int_labels)
    return predicted_labels
```

Run inference on new data

In []:

```
inputs = [
    "how do I extract keys from a dict into a list?", # 'python'
    "debug public static void main(string[] args) {...}", # 'java'
]
predicted_scores = export_model.predict(inputs)
predicted_labels = get_string_labels(predicted_scores)
for input, label in zip(inputs, predicted_labels):
    print("Question: ", input)
    print("Predicted label: ", label.numpy())
```

Including the text preprocessing logic inside your model enables you to export a model for production that simplifies deployment, and reduces the potential for [train/test skew](https://developers.google.com/machine-learning/guides/rules-of-ml#training-serving_skew) (https://developers.google.com/machine-learning/guides/rules-of-ml#training-serving_skew).

There is a performance difference to keep in mind when choosing where to apply `tf.keras.layers.TextVectorization`. Using it outside of your model enables you to do asynchronous CPU processing and buffering of your data when training on GPU. So, if you're training your model on the GPU, you probably want to go with this option to get the best performance while developing your model, then switch to including the `TextVectorization` layer inside your model when you're ready to prepare for deployment.

Visit the [Save and load models \(..keras/save_and_load.ipynb\)](#) tutorial to learn more about saving models.

Example 2: Predict the author of Iliad translations

The following provides an example of using `tf.data.TextLineDataset` to load examples from text files, and [TensorFlow Text](#) (<https://www.tensorflow.org/text>) to preprocess the data. You will use three different English translations of the same work, Homer's Iliad, and train a model to identify the translator given a single line of text.

Download and explore the dataset

The texts of the three translations are by:

- [William Cowper](https://en.wikipedia.org/wiki/William_Cowper) (<https://storage.googleapis.com/download.tensorflow.org/data/illiad/cowper.txt>): text
- [Edward, Earl of Derby](https://en.wikipedia.org/wiki/Edward_Smith-Stanley,_14th_Earl_of_Derby) (<https://storage.googleapis.com/download.tensorflow.org/data/illiad/derby.txt>): text
- [Samuel Butler](https://en.wikipedia.org/wiki/Samuel_Butler_%28novelist%29) (<https://storage.googleapis.com/download.tensorflow.org/data/illiad/butler.txt>): text

The text files used in this tutorial have undergone some typical preprocessing tasks like removing document header and footer, line numbers and chapter titles.

Download these lightly munged files locally:

In []:

```
DIRECTORY_URL = 'https://storage.googleapis.com/download.tensorflow.org/data/illiad/'  
FILE_NAMES = ['cowper.txt', 'derby.txt', 'butler.txt']  
  
for name in FILE_NAMES:  
    text_dir = utils.get_file(name, origin= DIRECTORY_URL + name)  
  
parent_dir = pathlib.Path(text_dir).parent  
list(parent_dir.iterdir())
```

Load the dataset

Previously, with `tf.keras.utils.text_dataset_from_directory` all contents of a file were treated as a single example. Here, you will use `tf.data.TextLineDataset`, which is designed to create a `tf.data.Dataset` from a text file where each example is a line of text from the original file. `TextLineDataset` is useful for text data that is primarily line-based (for example, poetry or error logs).

Iterate through these files, loading each one into its own dataset. Each example needs to be individually labeled, so use `Dataset.map` to apply a `labeler` function to each one. This will iterate over every example in the dataset, returning (`example, label`) pairs.

In []:

```
def labeler(example, index):  
    return example, tf.cast(index, tf.int64)
```

In []:

```
labeled_data_sets = []  
  
for i, file_name in enumerate(FILE_NAMES):  
    lines_dataset = tf.data.TextLineDataset(str(parent_dir/file_name))  
    labeled_dataset = lines_dataset.map(lambda ex: labeler(ex, i))  
    labeled_data_sets.append(labeled_dataset)
```

Next, you'll combine these labeled datasets into a single dataset using `Dataset.concatenate`, and shuffle it with `Dataset.shuffle`:

In []:

```
BUFFER_SIZE = 50000  
BATCH_SIZE = 64  
VALIDATION_SIZE = 5000
```

In []:

```
all_labeled_data = labeled_data_sets[0]  
for labeled_dataset in labeled_data_sets[1:]:  
    all_labeled_data = all_labeled_data.concatenate(labeled_dataset)  
  
all_labeled_data = all_labeled_data.shuffle(  
    BUFFER_SIZE, reshuffle_each_iteration=False)
```

Print out a few examples as before. The dataset hasn't been batched yet, hence each entry in `all_labeled_data` corresponds to one data point:

In []:

```
for text, label in all_labeled_data.take(10):  
    print("Sentence: ", text.numpy())  
    print("Label: ", label.numpy())
```

Prepare the dataset for training

Instead of using `tf.keras.layers.TextVectorization` to preprocess the text dataset, you will now use the TensorFlow Text APIs to standardize and tokenize the data, build a vocabulary and use `tf.lookup.StaticVocabularyTable` to map tokens to integers to feed to the model. (Learn more about [TensorFlow Text](https://www.tensorflow.org/text) (<https://www.tensorflow.org/text>)).

Define a function to convert the text to lower-case and tokenize it:

- TensorFlow Text provides various tokenizers. In this example, you will use the `text.UnicodeScriptTokenizer` to tokenize the dataset.
- You will use `Dataset.map` to apply the tokenization to the dataset.

In []:

```
tokenizer = tf_text.UnicodeScriptTokenizer()
```

In []:

```
def tokenize(text, unused_label):
    lower_case = tf_text.case_fold_utf8(text)
    return tokenizer.tokenize(lower_case)
```

In []:

```
tokenized_ds = all_labeled_data.map(tokenize)
```

You can iterate over the dataset and print out a few tokenized examples:

In []:

```
for text_batch in tokenized_ds.take(5):
    print("Tokens: ", text_batch.numpy())
```

Next, you will build a vocabulary by sorting tokens by frequency and keeping the top `VOCAB_SIZE` tokens:

In []:

```
tokenized_ds = configure_dataset(tokenized_ds)

vocab_dict = collections.defaultdict(lambda: 0)
for toks in tokenized_ds.as_numpy_iterator():
    for tok in toks:
        vocab_dict[tok] += 1

vocab = sorted(vocab_dict.items(), key=lambda x: x[1], reverse=True)
vocab = [token for token, count in vocab]
vocab = vocab[:VOCAB_SIZE]
vocab_size = len(vocab)
print("Vocab size: ", vocab_size)
print("First five vocab entries:", vocab[:5])
```

To convert the tokens into integers, use the `vocab` set to create a `tf.lookup.StaticVocabularyTable`. You will map tokens to integers in the range `[2, vocab_size + 2]`. As with the `TextVectorization` layer, `0` is reserved to denote padding and `1` is reserved to denote an out-of-vocabulary (OOV) token.

In []:

```
keys = vocab
values = range(2, len(vocab) + 2) # Reserve `0` for padding, `1` for OOV tokens.

init = tf.lookup.KeyValueTensorInitializer(
    keys, values, key_dtype=tf.string, value_dtype=tf.int64)

num_oov_buckets = 1
vocab_table = tf.lookup.StaticVocabularyTable(init, num_oov_buckets)
```

Finally, define a function to standardize, tokenize and vectorize the dataset using the tokenizer and lookup table:

In []:

```
def preprocess_text(text, label):
    standardized = tf_text.case_fold_utf8(text)
    tokenized = tokenizer.tokenize(standardized)
    vectorized = vocab_table.lookup(tokenized)
    return vectorized, label
```

You can try this on a single example to print the output:

In []:

```
example_text, example_label = next(iter(all_labeled_data))
print("Sentence: ", example_text.numpy())
vectorized_text, example_label = preprocess_text(example_text, example_label)
print("Vectorized sentence: ", vectorized_text.numpy())
```

Now run the preprocess function on the dataset using `Dataset.map`:

In []:

```
all_encoded_data = all_labeled_data.map(preprocess_text)
```

Split the dataset into training and test sets

The Keras `TextVectorization` layer also batches and pads the vectorized data. Padding is required because the examples inside of a batch need to be the same size and shape, but the examples in these datasets are not all the same size—each line of text has a different number of words.

`tf.data.Dataset` supports splitting and padded-batching datasets:

In []:

```
train_data = all_encoded_data.skip(VALIDATION_SIZE).shuffle(BUFFER_SIZE)
validation_data = all_encoded_data.take(VALIDATION_SIZE)
```

In []:

```
train_data = train_data.padded_batch(BATCH_SIZE)
validation_data = validation_data.padded_batch(BATCH_SIZE)
```

Now, `validation_data` and `train_data` are not collections of (`example`, `label`) pairs, but collections of batches. Each batch is a pair of (*many examples, many labels*) represented as arrays.

To illustrate this:

In []:

```
sample_text, sample_labels = next(iter(validation_data))
print("Text batch shape: ", sample_text.shape)
print("Label batch shape: ", sample_labels.shape)
print("First text example: ", sample_text[0])
print("First label example: ", sample_labels[0])
```

Since you use `0` for padding and `1` for out-of-vocabulary (OOV) tokens, the vocabulary size has increased by two:

In []:

```
vocab_size += 2
```

Configure the datasets for better performance as before:

In []:

```
train_data = configure_dataset(train_data)
validation_data = configure_dataset(validation_data)
```

Train the model

You can train a model on this dataset as before:

In []:

```
model = create_model(vocab_size=vocab_size, num_labels=3)

model.compile(
    optimizer='adam',
    loss=losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=['accuracy'])

history = model.fit(train_data, validation_data=validation_data, epochs=3)
```

In []:

```
loss, accuracy = model.evaluate(validation_data)
print("Loss: ", loss)
print("Accuracy: {:.2%}".format(accuracy))
```

Export the model

To make the model capable of taking raw strings as input, you will create a Keras `TextVectorization` layer that performs the same steps as your custom preprocessing function. Since you have already trained a vocabulary, you can use `TextVectorization.set_vocabulary` (instead of `TextVectorization.adapt`), which trains a new vocabulary.

In []:

```
preprocess_layer = TextVectorization(
    max_tokens=vocab_size,
    standardize=tf_text.case_fold_utf8,
    split=tokenizer.tokenize,
    output_mode='int',
    output_sequence_length=MAX_SEQUENCE_LENGTH)

preprocess_layer.set_vocabulary(vocab)
```

In []:

```
export_model = tf.keras.Sequential(
    [preprocess_layer, model,
     layers.Activation('sigmoid')])

export_model.compile(
    loss=losses.SparseCategoricalCrossentropy(from_logits=False),
    optimizer='adam',
    metrics=['accuracy'])
```

In []:

```
# Create a test dataset of raw strings.
test_ds = all_labeled_data.take(VALIDATION_SIZE).batch(BATCH_SIZE)
test_ds = configure_dataset(test_ds)

loss, accuracy = export_model.evaluate(test_ds)

print("Loss: ", loss)
print("Accuracy: {:.2%}".format(accuracy))
```

The loss and accuracy for the model on encoded validation set and the exported model on the raw validation set are the same, as expected.

Run inference on new data

In []:

```
inputs = [
    "Join'd to th' Ionians with their flowing robes,", # Label: 1
    "the allies, and his armour flashed about him so that he seemed to all", # Label: 2
    "And with loud clangor of his arms he fell.", # Label: 0
]

predicted_scores = export_model.predict(inputs)
predicted_labels = tf.argmax(predicted_scores, axis=1)

for input, label in zip(inputs, predicted_labels):
    print("Question: ", input)
    print("Predicted label: ", label.numpy())
```

Download more datasets using TensorFlow Datasets (TFDS)

You can download many more datasets from [TensorFlow Datasets \(<https://www.tensorflow.org/datasets/catalog/overview>\)](https://www.tensorflow.org/datasets/catalog/overview).

In this example, you will use the [IMDB Large Movie Review dataset \(\[https://www.tensorflow.org/datasets/catalog/imdb_reviews\]\(https://www.tensorflow.org/datasets/catalog/imdb_reviews\)\)](https://www.tensorflow.org/datasets/catalog/imdb_reviews) to train a model for sentiment classification:

In []:

```
# Training set.  
train_ds = tfds.load(  
    'imdb_reviews',  
    split='train[:80%]',  
    batch_size=BATCH_SIZE,  
    shuffle_files=True,  
    as_supervised=True)
```

In []:

```
# Validation set.  
val_ds = tfds.load(  
    'imdb_reviews',  
    split='train[80%:]',  
    batch_size=BATCH_SIZE,  
    shuffle_files=True,  
    as_supervised=True)
```

Print a few examples:

In []:

```
for review_batch, label_batch in val_ds.take(1):  
    for i in range(5):  
        print("Review: ", review_batch[i].numpy())  
        print("Label: ", label_batch[i].numpy())
```

You can now preprocess the data and train a model as before.

Note: You will use `tf.keras.losses.BinaryCrossentropy` instead of `tf.keras.losses.SparseCategoricalCrossentropy` for your model, since this is a binary classification problem.

Prepare the dataset for training

In []:

```
vectorize_layer = TextVectorization(  
    max_tokens=VOCAB_SIZE,  
    output_mode='int',  
    output_sequence_length=MAX_SEQUENCE_LENGTH)  
  
# Make a text-only dataset (without labels), then call `TextVectorization.adapt`.  
train_text = train_ds.map(lambda text, labels: text)  
vectorize_layer.adapt(train_text)
```

In []:

```
def vectorize_text(text, label):  
    text = tf.expand_dims(text, -1)  
    return vectorize_layer(text), label
```

In []:

```
train_ds = train_ds.map(vectorize_text)  
val_ds = val_ds.map(vectorize_text)
```

In []:

```
# Configure datasets for performance as before.  
train_ds = configure_dataset(train_ds)  
val_ds = configure_dataset(val_ds)
```

Create, configure and train the model

In []:

```
model = create_model(vocab_size=VOCAB_SIZE + 1, num_labels=1)  
model.summary()
```

In []:

```
model.compile(  
    loss=losses.BinaryCrossentropy(from_logits=True),  
    optimizer='adam',  
    metrics=['accuracy'])
```

In []:

```
history = model.fit(train_ds, validation_data=val_ds, epochs=3)
```

In []:

```
loss, accuracy = model.evaluate(val_ds)  
  
print("Loss: ", loss)  
print("Accuracy: {:.2%}".format(accuracy))
```

Export the model

In []:

```
export_model = tf.keras.Sequential(  
    [vectorize_layer, model,  
     layers.Activation('sigmoid')])  
  
export_model.compile(  
    loss=losses.SparseCategoricalCrossentropy(from_logits=False),  
    optimizer='adam',  
    metrics=['accuracy'])
```

In []:

```
# 0 --> negative review  
# 1 --> positive review  
inputs = [  
    "This is a fantastic movie.",  
    "This is a bad movie.",  
    "This movie was so bad that it was good.",  
    "I will never say yes to watching this movie."  
]  
  
predicted_scores = export_model.predict(inputs)  
predicted_labels = [int(round(x[0])) for x in predicted_scores]  
  
for input, label in zip(inputs, predicted_labels):  
    print("Question: ", input)  
    print("Predicted label: ", label)
```

Conclusion

This tutorial demonstrated several ways to load and preprocess text. As a next step, you can explore additional text preprocessing [TensorFlow Text](https://www.tensorflow.org/text) (<https://www.tensorflow.org/text>) tutorials, such as:

- [BERT Preprocessing with TF Text](https://www.tensorflow.org/text/guide/bert_preprocessing_guide) (https://www.tensorflow.org/text/guide/bert_preprocessing_guide)
- [Tokenizing with TF Text](https://www.tensorflow.org/text/guide/tokenizers) (<https://www.tensorflow.org/text/guide/tokenizers>)
- [Subword tokenizers](https://www.tensorflow.org/text/guide/subwords_tokenizer) (https://www.tensorflow.org/text/guide/subwords_tokenizer)

You can also find new datasets on [TensorFlow Datasets](https://www.tensorflow.org/datasets/catalog/overview) (<https://www.tensorflow.org/datasets/catalog/overview>). And, to learn more about `tf.data`, check out the guide on [building input pipelines](#) ([../guide/data.ipynb](#)).

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Load NumPy data



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/load_data/numpy)

(https://www.tensorflow.org/tutorials/load_data/numpy)



[Run in Google Colab](#)

(https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/load_data/numpy.ipynb) (https://github.com/tensorflow/docs/blob/master/site/en/tutorials/load_data/numpy.ipynb)

This tutorial provides an example of loading data from NumPy arrays into a `tf.data.Dataset`.

This example loads the MNIST dataset from a `.npz` file. However, the source of the NumPy arrays is not important.

Setup

In []:

```
import numpy as np
import tensorflow as tf
```

Load from `.npz` file

In []:

```
DATA_URL = 'https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz'

path = tf.keras.utils.get_file('mnist.npz', DATA_URL)
with np.load(path) as data:
    train_examples = data['x_train']
    train_labels = data['y_train']
    test_examples = data['x_test']
    test_labels = data['y_test']
```

Load NumPy arrays with `tf.data.Dataset`

Assuming you have an array of examples and a corresponding array of labels, pass the two arrays as a tuple into `tf.data.Dataset.from_tensor_slices` to create a `tf.data.Dataset`.

In []:

```
train_dataset = tf.data.Dataset.from_tensor_slices((train_examples, train_labels))
test_dataset = tf.data.Dataset.from_tensor_slices((test_examples, test_labels))
```

Use the datasets

Shuffle and batch the datasets

In []:

```
BATCH_SIZE = 64
SHUFFLE_BUFFER_SIZE = 100

train_dataset = train_dataset.shuffle(SHUFFLE_BUFFER_SIZE).batch(BATCH_SIZE)
test_dataset = test_dataset.batch(BATCH_SIZE)
```

Build and train a model

In []:

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
])

model.compile(optimizer=tf.keras.optimizers.RMSprop(),
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['sparse_categorical_accuracy'])
```

In []:

```
model.fit(train_dataset, epochs=10)
```

In []:

```
model.evaluate(test_dataset)
```

Copyright 2019 The TensorFlow Authors.

Licensed under the Apache License, Version 2.0 (the "License");

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Load a pandas DataFrame



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/load_data/pandas_dataframe)

(https://www.tensorflow.org/tutorials/load_data/pandas_dataframe)

[Run in Gc](#)

(https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/load_data/pandas_dataframe.ipynb)

This tutorial provides examples of how to load [pandas DataFrames](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html) (<https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html>) into TensorFlow.

You will use a small [heart disease dataset](https://archive.ics.uci.edu/ml/datasets/heart+Disease) (<https://archive.ics.uci.edu/ml/datasets/heart+Disease>) provided by the UCI Machine Learning Repository. There are several hundred rows in the CSV. Each row describes a patient, and each column describes an attribute. You will use this information to predict whether a patient has heart disease, which is a binary classification task.

Read data using pandas

In []:

```
import pandas as pd
import tensorflow as tf

SHUFFLE_BUFFER = 500
BATCH_SIZE = 2
```

Download the CSV file containing the heart disease dataset:

In []:

```
csv_file = tf.keras.utils.get_file('heart.csv', 'https://storage.googleapis.com/download.tensorflow.org/data/heart.csv')
```

Read the CSV file using pandas:

In []:

```
df = pd.read_csv(csv_file)
```

This is what the data looks like:

In []:

```
df.head()
```

In []:

```
df.dtypes
```

You will build models to predict the label contained in the `target` column.

In []:

```
target = df.pop('target')
```

A DataFrame as an array

If your data has a uniform datatype, or `dtype`, it's possible to use a pandas DataFrame anywhere you could use a NumPy array. This works because the `pandas.DataFrame` class supports the `__array__` protocol, and TensorFlow's `tf.convert_to_tensor` function accepts objects that support the protocol.

Take the numeric features from the dataset (skip the categorical features for now):

In []:

```
numeric_feature_names = ['age', 'thalach', 'trestbps', 'chol', 'oldpeak']
numeric_features = df[numeric_feature_names]
numeric_features.head()
```

The DataFrame can be converted to a NumPy array using the `DataFrame.values` property or `numpy.array(df)`. To convert it to a tensor, use `tf.convert_to_tensor`:

In []:

```
tf.convert_to_tensor(numeric_features)
```

In general, if an object can be converted to a tensor with `tf.convert_to_tensor` it can be passed anywhere you can pass a `tf.Tensor`.

With Model.fit

A DataFrame, interpreted as a single tensor, can be used directly as an argument to the `Model.fit` method.

Below is an example of training a model on the numeric features of the dataset.

The first step is to normalize the input ranges. Use a `tf.keras.layers.Normalization` layer for that.

To set the layer's mean and standard-deviation before running it be sure to call the `Normalization.adapt` method:

In []:

```
normalizer = tf.keras.layers.Normalization(axis=-1)
normalizer.adapt(numeric_features)
```

Call the layer on the first three rows of the DataFrame to visualize an example of the output from this layer:

In []:

```
normalizer(numeric_features.iloc[:3])
```

Use the normalization layer as the first layer of a simple model:

```
In [ ]:
```

```
def get_basic_model():
    model = tf.keras.Sequential([
        normalizer,
        tf.keras.layers.Dense(10, activation='relu'),
        tf.keras.layers.Dense(10, activation='relu'),
        tf.keras.layers.Dense(1)
    ])

    model.compile(optimizer='adam',
                  loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
                  metrics=['accuracy'])
    return model
```

When you pass the DataFrame as the `x` argument to `Model.fit`, Keras treats the DataFrame as it would a NumPy array:

```
In [ ]:
```

```
model = get_basic_model()
model.fit(numeric_features, target, epochs=15, batch_size=BATCH_SIZE)
```

With `tf.data`

If you want to apply `tf.data` transformations to a DataFrame of a uniform `dtype`, the `Dataset.from_tensor_slices` method will create a dataset that iterates over the rows of the DataFrame. Each row is initially a vector of values. To train a model, you need `(inputs, labels)` pairs, so pass `(features, labels)` and `Dataset.from_tensor_slices` will return the needed pairs of slices:

```
In [ ]:
```

```
numeric_dataset = tf.data.Dataset.from_tensor_slices((numeric_features, target))

for row in numeric_dataset.take(3):
    print(row)
```

```
In [ ]:
```

```
numeric_batches = numeric_dataset.shuffle(1000).batch(BATCH_SIZE)

model = get_basic_model()
model.fit(numeric_batches, epochs=15)
```

A DataFrame as a dictionary

When you start dealing with heterogeneous data, it is no longer possible to treat the DataFrame as if it were a single array. TensorFlow tensors require that all elements have the same `dtype`.

So, in this case, you need to start treating it as a dictionary of columns, where each column has a uniform `dtype`. A DataFrame is a lot like a dictionary of arrays, so typically all you need to do is cast the DataFrame to a Python dict. Many important TensorFlow APIs support (nested-)dictionaries of arrays as inputs.

`tf.data` input pipelines handle this quite well. All `tf.data` operations handle dictionaries and tuples automatically. So, to make a dataset of dictionary-examples from a DataFrame, just cast it to a dict before slicing it with `Dataset.from_tensor_slices`:

```
In [ ]:
```

```
numeric_dict_ds = tf.data.Dataset.from_tensor_slices((dict(numeric_features), target))
```

Here are the first three examples from that dataset:

```
In [ ]:
```

```
for row in numeric_dict_ds.take(3):
    print(row)
```

Dictionaries with Keras

Typically, Keras models and layers expect a single input tensor, but these classes can accept and return nested structures of dictionaries, tuples and tensors. These structures are known as "nests" (refer to the `tf.nest` module for details).

There are two equivalent ways you can write a Keras model that accepts a dictionary as input.

1. The Model-subclass style

You write a subclass of `tf.keras.Model` (or `tf.keras.Layer`). You directly handle the inputs, and create the outputs:

In []:

```
def stack_dict(inputs, fun=tf.stack):
    values = []
    for key in sorted(inputs.keys()):
        values.append(tf.cast(inputs[key], tf.float32))

    return fun(values, axis=-1)
```

In []:

```
#@title
class MyModel(tf.keras.Model):
    def __init__(self):
        # Create all the internal layers in init.
        super().__init__(self)

        self.normalizer = tf.keras.layers.Normalization(axis=-1)

        self.seq = tf.keras.Sequential([
            self.normalizer,
            tf.keras.layers.Dense(10, activation='relu'),
            tf.keras.layers.Dense(10, activation='relu'),
            tf.keras.layers.Dense(1)
        ])

    def adapt(self, inputs):
        # Stack the inputs and `adapt` the normalization layer.
        inputs = stack_dict(inputs)
        self.normalizer.adapt(inputs)

    def call(self, inputs):
        # Stack the inputs
        inputs = stack_dict(inputs)
        # Run them through all the layers.
        result = self.seq(inputs)

    return result

model = MyModel()

model.adapt(dict(numeric_features))

model.compile(optimizer='adam',
              loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              metrics=['accuracy'],
              run_eagerly=True)
```

This model can accept either a dictionary of columns or a dataset of dictionary-elements for training:

In []:

```
model.fit(dict(numeric_features), target, epochs=5, batch_size=BATCH_SIZE)
```

In []:

```
numeric_dict_batches = numeric_dict_ds.shuffle(SHUFFLE_BUFFER).batch(BATCH_SIZE)
model.fit(numeric_dict_batches, epochs=5)
```

Here are the predictions for the first three examples:

In []:

```
model.predict(dict(numeric_features.iloc[:3]))
```

2. The Keras functional style

```
In [ ]:
```

```
inputs = {}
for name, column in numeric_features.items():
    inputs[name] = tf.keras.Input(
        shape=(1,), name=name, dtype=tf.float32)

inputs
```

```
In [ ]:
```

```
x = stack_dict(inputs, fun=tf.concat)

normalizer = tf.keras.layers.Normalization(axis=-1)
normalizer.adapt(stack_dict(dict(numeric_features)))

x = normalizer(x)
x = tf.keras.layers.Dense(10, activation='relu')(x)
x = tf.keras.layers.Dense(10, activation='relu')(x)
x = tf.keras.layers.Dense(1)(x)

model = tf.keras.Model(inputs, x)

model.compile(optimizer='adam',
              loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              metrics=['accuracy'],
              run_eagerly=True)
```

```
In [ ]:
```

```
tf.keras.utils.plot_model(model, rankdir="LR", show_shapes=True)
```

You can train the functional model the same way as the model subclass:

```
In [ ]:
```

```
model.fit(dict(numeric_features), target, epochs=5, batch_size=BATCH_SIZE)
```

```
In [ ]:
```

```
numeric_dict_batches = numeric_dict_ds.shuffle(SHUFFLE_BUFFER).batch(BATCH_SIZE)
model.fit(numeric_dict_batches, epochs=5)
```

Full example

If you're passing a heterogeneous DataFrame to Keras, each column may need unique preprocessing. You could do this preprocessing directly in the DataFrame, but for a model to work correctly, inputs always need to be preprocessed the same way. So, the best approach is to build the preprocessing into the model. [Keras preprocessing layers \(\[https://www.tensorflow.org/guide/keras/preprocessing_layers\]\(https://www.tensorflow.org/guide/keras/preprocessing_layers\)\)](https://www.tensorflow.org/guide/keras/preprocessing_layers) cover many common tasks.

Build the preprocessing head

In this dataset some of the "integer" features in the raw data are actually Categorical indices. These indices are not really ordered numeric values (refer to the [the dataset description \(<https://archive.ics.uci.edu/ml/datasets/heart+Disease>\)](https://archive.ics.uci.edu/ml/datasets/heart+Disease) for details). Because these are unordered they are inappropriate to feed directly to the model; the model would interpret them as being ordered. To use these inputs you'll need to encode them, either as one-hot vectors or embedding vectors. The same applies to string-categorical features.

Note: If you have many features that need identical preprocessing it's more efficient to concatenate them together before applying the preprocessing.

Binary features on the other hand do not generally need to be encoded or normalized.

Start by creating a list of the features that fall into each group:

```
In [ ]:
```

```
binary_feature_names = ['sex', 'fbs', 'exang']
```

```
In [ ]:
```

```
categorical_feature_names = ['cp', 'restecg', 'slope', 'thal', 'ca']
```

The next step is to build a preprocessing model that will apply appropriate preprocessing to each input and concatenate the results.

This section uses the [Keras Functional API \(<https://www.tensorflow.org/guide/keras/functional>\)](https://www.tensorflow.org/guide/keras/functional) to implement the preprocessing. You start by creating one `tf.keras.Input` for each column of the dataframe:

In []:

```
inputs = {}
for name, column in df.items():
    if type(column[0]) == str:
        dtype = tf.string
    elif (name in categorical_feature_names or
          name in binary_feature_names):
        dtype = tf.int64
    else:
        dtype = tf.float32

    inputs[name] = tf.keras.Input(shape=(), name=name, dtype=dtype)
```

In []:

```
inputs
```

For each input you'll apply some transformations using Keras layers and TensorFlow ops. Each feature starts as a batch of scalars (`shape=(batch,)`). The output for each should be a batch of `tf.float32` vectors (`shape=(batch, n)`). The last step will concatenate all those vectors together.

Binary inputs

Since the binary inputs don't need any preprocessing, just add the vector axis, cast them to `float32` and add them to the list of preprocessed inputs:

In []:

```
preprocessed = []

for name in binary_feature_names:
    inp = inputs[name]
    inp = inp[:, tf.newaxis]
    float_value = tf.cast(inp, tf.float32)
    preprocessed.append(float_value)

preprocessed
```

Numeric inputs

Like in the earlier section you'll want to run these numeric inputs through a `tf.keras.layers.Normalization` layer before using them. The difference is that this time they're input as a dict. The code below collects the numeric features from the DataFrame, stacks them together and passes those to the `Normalization.adapt` method.

In []:

```
normalizer = tf.keras.layers.Normalization(axis=-1)
normalizer.adapt(stack_dict(dict(numeric_features)))
```

The code below stacks the numeric features and runs them through the normalization layer.

In []:

```
numeric_inputs = {}
for name in numeric_feature_names:
    numeric_inputs[name]=inputs[name]

numeric_inputs = stack_dict(numeric_inputs)
numeric_normalized = normalizer(numeric_inputs)

preprocessed.append(numeric_normalized)

preprocessed
```

Categorical features

To use categorical features you'll first need to encode them into either binary vectors or embeddings. Since these features only contain a small number of categories, convert the inputs directly to one-hot vectors using the `output_mode='one_hot'` option, supported by both the `tf.keras.layers.StringLookup` and `tf.keras.layers.IntegerLookup` layers.

Here is an example of how these layers work:

In []:

```
vocab = ['a', 'b', 'c']
lookup = tf.keras.layers.StringLookup(vocabulary=vocab, output_mode='one_hot')
lookup(['c', 'a', 'a', 'b', 'zzz'])
```

In []:

```
vocab = [1,4,7,99]
lookup = tf.keras.layers.IntegerLookup(vocabulary=vocab, output_mode='one_hot')

lookup([-1,4,1])
```

To determine the vocabulary for each input, create a layer to convert that vocabulary to a one-hot vector:

In []:

```
for name in categorical_feature_names:
    vocab = sorted(set(df[name]))
    print(f'name: {name}')
    print(f'vocab: {vocab}\n')

    if type(vocab[0]) is str:
        lookup = tf.keras.layers.StringLookup(vocabulary=vocab, output_mode='one_hot')
    else:
        lookup = tf.keras.layers.IntegerLookup(vocabulary=vocab, output_mode='one_hot')

    x = inputs[name][:, tf.newaxis]
    x = lookup(x)
    preprocessed.append(x)
```

Assemble the preprocessing head

At this point `preprocessed` is just a Python list of all the preprocessing results, each result has a shape of `(batch_size, depth)`:

In []:

```
preprocessed
```

Concatenate all the preprocessed features along the `depth` axis, so each dictionary-example is converted into a single vector. The vector contains categorical features, numeric features, and categorical one-hot features:

In []:

```
preprocessed_result = tf.concat(preprocessed, axis=-1)
preprocessed_result
```

Now create a model out of that calculation so it can be reused:

In []:

```
preprocessor = tf.keras.Model(inputs, preprocessed_result)
```

In []:

```
tf.keras.utils.plot_model(preprocessor, rankdir="LR", show_shapes=True)
```

To test the preprocessor, use the [DataFrame.iloc](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.iloc.html) (<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.iloc.html>) accessor to slice the first example from the DataFrame. Then convert it to a dictionary and pass the dictionary to the preprocessor. The result is a single vector containing the binary features, normalized numeric features and the one-hot categorical features, in that order:

In []:

```
preprocessor(dict(df.iloc[:1]))
```

Create and train a model

Now build the main body of the model. Use the same configuration as in the previous example: A couple of `Dense` rectified-linear layers and a `Dense(1)` output layer for the classification.

In []:

```
body = tf.keras.Sequential([
    tf.keras.layers.Dense(10, activation='relu'),
    tf.keras.layers.Dense(10, activation='relu'),
    tf.keras.layers.Dense(1)
])
```

Now put the two pieces together using the Keras functional API.

In []:

```
inputs
```

In []:

```
x = preprocessor(inputs)
x
```

In []:

```
result = body(x)
result
```

In []:

```
model = tf.keras.Model(inputs, result)

model.compile(optimizer='adam',
              loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

This model expects a dictionary of inputs. The simplest way to pass it the data is to convert the DataFrame to a dict and pass that dict as the `x` argument to `Model.fit`:

In []:

```
history = model.fit(dict(df), target, epochs=5, batch_size=BATCH_SIZE)
```

Using `tf.data` works as well:

In []:

```
ds = tf.data.Dataset.from_tensor_slices((
    dict(df),
    target
))

ds = ds.batch(BATCH_SIZE)
```

In []:

```
import pprint

for x, y in ds.take(1):
    pprint.pprint(x)
    print()
    print(y)
```

In []:

```
history = model.fit(ds, epochs=5)
```

Copyright 2019 The TensorFlow Authors.

In []:

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Load CSV data



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/load_data/csv)

(https://www.tensorflow.org/tutorials/load_data/csv)



[Run in Google Colab](#)

(https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/load_data/csv.ipynb)

(https://github.com/tensorflow/tensorflow/blob/r1.14/tensorflow/tutorials/load_data/csv.ipynb)

This tutorial provides examples of how to use CSV data with TensorFlow.

There are two main parts to this:

1. **Loading the data off disk**
2. **Pre-processing it into a form suitable for training.**

This tutorial focuses on the loading, and gives some quick examples of preprocessing. For a tutorial that focuses on the preprocessing aspect see the [preprocessing layers guide](https://www.tensorflow.org/guide/keras/preprocessing_layers#quick_recipes) (https://www.tensorflow.org/guide/keras/preprocessing_layers#quick_recipes) and [tutorial](https://www.tensorflow.org/tutorials/structured_data/preprocessing_layers) (https://www.tensorflow.org/tutorials/structured_data/preprocessing_layers).

Setup

In []:

```
import pandas as pd
import numpy as np

# Make numpy values easier to read.
np.set_printoptions(precision=3, suppress=True)

import tensorflow as tf
from tensorflow.keras import layers
```

In memory data

For any small CSV dataset the simplest way to train a TensorFlow model on it is to load it into memory as a pandas Dataframe or a NumPy array.

A relatively simple example is the [abalone dataset](https://archive.ics.uci.edu/ml/datasets/abalone) (<https://archive.ics.uci.edu/ml/datasets/abalone>).

- The dataset is small.
- All the input features are all limited-range floating point values.

Here is how to download the data into a [Pandas DataFrame](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html) (<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>):

In []:

```
abalone_train = pd.read_csv(
    "https://storage.googleapis.com/download.tensorflow.org/data/abalone_train.csv",
    names=["Length", "Diameter", "Height", "Whole weight", "Shucked weight",
           "Viscera weight", "Shell weight", "Age"])

abalone_train.head()
```

The dataset contains a set of measurements of [abalone](https://en.wikipedia.org/wiki/Abalone) (<https://en.wikipedia.org/wiki/Abalone>), a type of sea snail.



"Abalone shell" (<https://www.flickr.com/photos/thenickster/16641048623/>) (by [Nicki Dugan Pogue](https://www.flickr.com/photos/thenickster/) (<https://www.flickr.com/photos/thenickster/>), CC BY-SA 2.0)

The nominal task for this dataset is to predict the age from the other measurements, so separate the features and labels for training:

In []:

```
abalone_features = abalone_train.copy()  
abalone_labels = abalone_features.pop('Age')
```

For this dataset you will treat all features identically. Pack the features into a single NumPy array.:

In []:

```
abalone_features = np.array(abalone_features)  
abalone_features
```

Next make a regression model predict the age. Since there is only a single input tensor, a `keras.Sequential` model is sufficient here.

In []:

```
abalone_model = tf.keras.Sequential([  
    layers.Dense(64),  
    layers.Dense(1)  
])  
  
abalone_model.compile(loss = tf.keras.losses.MeanSquaredError(),  
                      optimizer = tf.optimizers.Adam())
```

To train that model, pass the features and labels to `Model.fit` :

In []:

```
abalone_model.fit(abalone_features, abalone_labels, epochs=10)
```

You have just seen the most basic way to train a model using CSV data. Next, you will learn how to apply preprocessing to normalize numeric columns.

Basic preprocessing

It's good practice to normalize the inputs to your model. The Keras preprocessing layers provide a convenient way to build this normalization into your model.

The layer will precompute the mean and variance of each column, and use these to normalize the data.

First you create the layer:

In []:

```
normalize = layers.Normalization()
```

Then you use the `Normalization.adapt()` method to adapt the normalization layer to your data.

Note: Only use your training data to `.adapt()` preprocessing layers. Do not use your validation or test data.

In []:

```
normalize.adapt(abalone_features)
```

Then use the normalization layer in your model:

In []:

```
norm_abalone_model = tf.keras.Sequential([
    normalize,
    layers.Dense(64),
    layers.Dense(1)
])

norm_abalone_model.compile(loss = tf.losses.MeanSquaredError(),
                            optimizer = tf.optimizers.Adam())

norm_abalone_model.fit(abalone_features, abalone_labels, epochs=10)
```

Mixed data types

The "Titanic" dataset contains information about the passengers on the Titanic. The nominal task on this dataset is to predict who survived.



Image from Wikimedia (https://commons.wikimedia.org/wiki/File:RMS_Titanic_3.jpg)

The raw data can easily be loaded as a Pandas `DataFrame`, but is not immediately usable as input to a TensorFlow model.

In []:

```
titanic = pd.read_csv("https://storage.googleapis.com/tf-datasets/titanic/train.csv")
titanic.head()
```

In []:

```
titanic_features = titanic.copy()
titanic_labels = titanic_features.pop('survived')
```

Because of the different data types and ranges you can't simply stack the features into NumPy array and pass it to a `keras.Sequential` model. Each column needs to be handled individually.

As one option, you could preprocess your data offline (using any tool you like) to convert categorical columns to numeric columns, then pass the processed output to your TensorFlow model. The disadvantage to that approach is that if you save and export your model the preprocessing is not saved with it. The Keras preprocessing layers avoid this problem because they're part of the model.

In this example, you'll build a model that implements the preprocessing logic using [Keras functional API](https://www.tensorflow.org/guide/keras/functional) (<https://www.tensorflow.org/guide/keras/functional>). You could also do it by [subclassing](https://www.tensorflow.org/guide/keras/custom_layers_and_models) (https://www.tensorflow.org/guide/keras/custom_layers_and_models).

The functional API operates on "symbolic" tensors. Normal "eager" tensors have a value. In contrast these "symbolic" tensors do not. Instead they keep track of which operations are run on them, and build representation of the calculation, that you can run later. Here's a quick example:

In []:

```
# Create a symbolic input
input = tf.keras.Input(shape=(), dtype=tf.float32)

# Perform a calculation using the input
result = 2*input + 1

# the result doesn't have a value
result
```

In []:

```
calc = tf.keras.Model(inputs=input, outputs=result)
```

```
In [ ]:
```

```
print(calc(1).numpy())
print(calc(2).numpy())
```

To build the preprocessing model, start by building a set of symbolic `keras.Input` objects, matching the names and data-types of the CSV columns.

```
In [ ]:
```

```
inputs = {}

for name, column in titanic_features.items():
    dtype = column.dtype
    if dtype == object:
        dtype = tf.string
    else:
        dtype = tf.float32

    inputs[name] = tf.keras.Input(shape=(1,), name=name, dtype=dtype)

inputs
```

The first step in your preprocessing logic is to concatenate the numeric inputs together, and run them through a normalization layer:

```
In [ ]:
```

```
numeric_inputs = {name:input for name,input in inputs.items()
                  if input.dtype==tf.float32}

x = layers.concatenate(list(numeric_inputs.values()))
norm = layers.Normalization()
norm.adapt(np.array(titanic[numeric_inputs.keys()]))
all_numeric_inputs = norm(x)

all_numeric_inputs
```

Collect all the symbolic preprocessing results, to concatenate them later.

```
In [ ]:
```

```
preprocessed_inputs = [all_numeric_inputs]
```

For the string inputs use the `tf.keras.layers.StringLookup` function to map from strings to integer indices in a vocabulary. Next, use `tf.keras.layers.CategoryEncoding` to convert the indexes into `float32` data appropriate for the model.

The default settings for the `tf.keras.layers.CategoryEncoding` layer create a one-hot vector for each input. A `layers.Embedding` would also work. See the [preprocessing layers guide \(https://www.tensorflow.org/guide/keras/preprocessing_layers#quick_recipes\)](https://www.tensorflow.org/guide/keras/preprocessing_layers#quick_recipes) and [tutorial \(./structured_data/preprocessing_layers.ipynb\)](#) for more on this topic.

```
In [ ]:
```

```
for name, input in inputs.items():
    if input.dtype == tf.float32:
        continue

    lookup = layers.StringLookup(vocabulary=np.unique(titanic_features[name]))
    one_hot = layers.CategoryEncoding(num_tokens=lookup.vocabulary_size())

    x = lookup(input)
    x = one_hot(x)
    preprocessed_inputs.append(x)
```

With the collection of `inputs` and `processed_inputs`, you can concatenate all the preprocessed inputs together, and build a model that handles the preprocessing:

```
In [ ]:
```

```
preprocessed_inputs_cat = layers.concatenate(preprocessed_inputs)

titanic_preprocessing = tf.keras.Model(inputs, preprocessed_inputs_cat)

tf.keras.utils.plot_model(model = titanic_preprocessing , rankdir="LR", dpi=72, show_shapes=True)
```

This `model` just contains the input preprocessing. You can run it to see what it does to your data. Keras models don't automatically convert Pandas `DataFrames` because it's not clear if it should be converted to one tensor or to a dictionary of tensors. So convert it to a dictionary of tensors:

```
In [ ]:
```

```
titanic_features_dict = {name: np.array(value)
                         for name, value in titanic_features.items()}
```

Slice out the first training example and pass it to this preprocessing model, you see the numeric features and string one-hots all concatenated together:

```
In [ ]:
```

```
features_dict = {name:values[:1] for name, values in titanic_features_dict.items()}
titanic_preprocessing(features_dict)
```

Now build the model on top of this:

```
In [ ]:
```

```
def titanic_model(preprocessing_head, inputs):
    body = tf.keras.Sequential([
        layers.Dense(64),
        layers.Dense(1)
    ])

    preprocessed_inputs = preprocessing_head(inputs)
    result = body(preprocessed_inputs)
    model = tf.keras.Model(inputs, result)

    model.compile(loss=tf.losses.BinaryCrossentropy(from_logits=True),
                  optimizer=tf.optimizers.Adam())
    return model

titanic_model = titanic_model(titanic_preprocessing, inputs)
```

When you train the model, pass the dictionary of features as `x`, and the label as `y`.

```
In [ ]:
```

```
titanic_model.fit(x=titanic_features_dict, y=titanic_labels, epochs=10)
```

Since the preprocessing is part of the model, you can save the model and reload it somewhere else and get identical results:

```
In [ ]:
```

```
titanic_model.save('test')
reloaded = tf.keras.models.load_model('test')
```

```
In [ ]:
```

```
features_dict = {name:values[:1] for name, values in titanic_features_dict.items()}

before = titanic_model(features_dict)
after = reloaded(features_dict)
assert (before-after)<1e-3
print(before)
print(after)
```

Using `tf.data`

In the previous section you relied on the model's built-in data shuffling and batching while training the model.

If you need more control over the input data pipeline or need to use data that doesn't easily fit into memory: use `tf.data`.

For more examples see the [tf.data guide \(../../guide/data.ipynb\)](#).

On in memory data

As a first example of applying `tf.data` to CSV data consider the following code to manually slice up the dictionary of features from the previous section. For each index, it takes that index for each feature:

```
In [ ]:
```

```
import itertools

def slices(features):
    for i in itertools.count():
        # For each feature take index `i`
        example = {name:values[i] for name, values in features.items()}
        yield example
```

Run this and print the first example:

```
In [ ]:
```

```
for example in slices(titanic_features_dict):
    for name, value in example.items():
        print(f"{name:19s}: {value}")
    break
```

The most basic `tf.data.Dataset` in memory data loader is the `Dataset.from_tensor_slices` constructor. This returns a `tf.data.Dataset` that implements a generalized version of the above `slices` function, in TensorFlow.

```
In [ ]:
```

```
features_ds = tf.data.Dataset.from_tensor_slices(titanic_features_dict)
```

You can iterate over a `tf.data.Dataset` like any other python iterable:

```
In [ ]:
```

```
for example in features_ds:
    for name, value in example.items():
        print(f"{name:19s}: {value}")
    break
```

The `from_tensor_slices` function can handle any structure of nested dictionaries or tuples. The following code makes a dataset of (`features_dict`, `labels`) pairs:

```
In [ ]:
```

```
titanic_ds = tf.data.Dataset.from_tensor_slices((titanic_features_dict, titanic_labels))
```

To train a model using this `Dataset`, you'll need to at least `shuffle` and `batch` the data.

```
In [ ]:
```

```
titanic_batches = titanic_ds.shuffle(len(titanic_labels)).batch(32)
```

Instead of passing `features` and `labels` to `Model.fit`, you pass the dataset:

```
In [ ]:
```

```
titanic_model.fit(titanic_batches, epochs=5)
```

From a single file

So far this tutorial has worked with in-memory data. `tf.data` is a highly scalable toolkit for building data pipelines, and provides a few functions for dealing loading CSV files.

```
In [ ]:
```

```
titanic_file_path = tf.keras.utils.get_file("train.csv", "https://storage.googleapis.com/tf-datasets/titanic/train.csv")
```

Now read the CSV data from the file and create a `tf.data.Dataset`.

(For the full documentation, see `tf.data.experimental.make_csv_dataset`)

In []:

```
titanic_csv_ds = tf.data.experimental.make_csv_dataset(  
    titanic_file_path,  
    batch_size=5, # Artificially small to make examples easier to show.  
    label_name='survived',  
    num_epochs=1,  
    ignore_errors=True,)
```

This function includes many convenient features so the data is easy to work with. This includes:

- Using the column headers as dictionary keys.
- Automatically determining the type of each column.

In []:

```
for batch, label in titanic_csv_ds.take(1):  
    for key, value in batch.items():  
        print(f"{key}: {value}")  
    print()  
print(f"label: {label}")
```

Note: if you run the above cell twice it will produce different results. The default settings for `make_csv_dataset` include `shuffle_buffer_size=1000`, which is more than sufficient for this small dataset, but may not be for a real-world dataset.

It can also decompress the data on the fly. Here's a gzipped CSV file containing the [metro interstate traffic dataset](#) (<https://archive.ics.uci.edu/ml/datasets/Metro+Interstate+Traffic+Volume>)



Image from Wikimedia (<https://commons.wikimedia.org/wiki/File:Trafficjam.jpg>)

In []:

```
traffic_volume_csv_gz = tf.keras.utils.get_file(  
    'Metro_Interstate_Traffic_Volume.csv.gz',  
    "https://archive.ics.uci.edu/ml/machine-learning-databases/00492/Metro_Interstate_Traffic_Volume.csv.gz",  
    cache_dir='.', cache_subdir='traffic')
```

Set the `compression_type` argument to read directly from the compressed file:

In []:

```
traffic_volume_csv_gz_ds = tf.data.experimental.make_csv_dataset(  
    traffic_volume_csv_gz,  
    batch_size=256,  
    label_name='traffic_volume',  
    num_epochs=1,  
    compression_type="GZIP")  
  
for batch, label in traffic_volume_csv_gz_ds.take(1):  
    for key, value in batch.items():  
        print(f"{key}: {value[:5]}")  
    print()  
print(f"label: {label[:5]})
```

Note: If you need to parse those date-time strings in the `tf.data` pipeline you can use `tfa.text.parse_time`.

Caching

There is some overhead to parsing the csv data. For small models this can be the bottleneck in training.

Depending on your use case it may be a good idea to use `Dataset.cache` or `data.experimental.snapshot` so that the csv data is only parsed on the first epoch.

The main difference between the `cache` and `snapshot` methods is that `cache` files can only be used by the TensorFlow process that created them, but `snapshot` files can be read by other processes.

For example, iterating over the `traffic_volume_csv_gz_ds` 20 times, takes ~15 seconds without caching, or ~2s with caching.

In []:

```
%%time
for i, (batch, label) in enumerate(traffic_volume_csv_gz_ds.repeat(20)):
    if i % 40 == 0:
        print('.', end='')
print()
```

Note: `Dataset.cache` stores the data from the first epoch and replays it in order. So using `.cache` disables any shuffles earlier in the pipeline. Below the `.shuffle` is added back in after `.cache`.

In []:

```
%%time
caching = traffic_volume_csv_gz_ds.cache().shuffle(1000)

for i, (batch, label) in enumerate(caching.shuffle(1000).repeat(20)):
    if i % 40 == 0:
        print('.', end='')
print()
```

Note: `snapshot` files are meant for *temporary* storage of a dataset while in use. This is *not* a format for long term storage. The file format is considered an internal detail, and not guaranteed between TensorFlow versions.

In []:

```
%%time
snapshot = tf.data.experimental.snapshot('titanic.tfsnap')
snapshotting = traffic_volume_csv_gz_ds.apply(snapshot).shuffle(1000)

for i, (batch, label) in enumerate(snapshotting.shuffle(1000).repeat(20)):
    if i % 40 == 0:
        print('.', end='')
print()
```

If your data loading is slowed by loading csv files, and `cache` and `snapshot` are insufficient for your use case, consider re-encoding your data into a more streamlined format.

Multiple files

All the examples so far in this section could easily be done without `tf.data`. One place where `tf.data` can really simplify things is when dealing with collections of files.

For example, the [character font images](https://archive.ics.uci.edu/ml/datasets/Character+Font+Images) (<https://archive.ics.uci.edu/ml/datasets/Character+Font+Images>) dataset is distributed as a collection of csv files, one per font.



Image by [Willi Heidelbach](https://pixabay.com/users/wilhei-883152/?utm_source=link-attribution&utm_medium=referral&utm_campaign=image&utm_content=705667) (https://pixabay.com/users/wilhei-883152/?utm_source=link-attribution&utm_medium=referral&utm_campaign=image&utm_content=705667) from [Pixabay](https://pixabay.com/?utm_source=link-attribution&utm_medium=referral&utm_campaign=image&utm_content=705667) (https://pixabay.com/?utm_source=link-attribution&utm_medium=referral&utm_campaign=image&utm_content=705667)

Download the dataset, and have a look at the files inside:

In []:

```
fonts_zip = tf.keras.utils.get_file(
    'fonts.zip', "https://archive.ics.uci.edu/ml/machine-learning-databases/00417/fonts.zip",
    cache_dir='.', cache_subdir='fonts',
    extract=True)
```

In []:

```
import pathlib
font_csvs = sorted(str(p) for p in pathlib.Path('fonts').glob('*.*csv'))
font_csvs[:10]
```

In []:

```
len(font_csvs)
```

When dealing with a bunch of files you can pass a glob-style `file_pattern` to the `experimental.make_csv_dataset` function. The order of the files is shuffled each iteration.

Use the `num_parallel_reads` argument to set how many files are read in parallel and interleaved together.

In []:

```
fonts_ds = tf.data.experimental.make_csv_dataset(  
    file_pattern = "fonts/*.csv",  
    batch_size=10, num_epochs=1,  
    num_parallel_reads=20,  
    shuffle_buffer_size=10000)
```

These csv files have the images flattened out into a single row. The column names are formatted `r{row}c{column}`. Here's the first batch:

In []:

```
for features in fonts_ds.take(1):  
    for i, (name, value) in enumerate(features.items()):  
        if i>15:  
            break  
        print(f"{name:20s}: {value}")  
print('...')  
print(f"[total: {len(features)} features]")
```

Optional: Packing fields

You probably don't want to work with each pixel in separate columns like this. Before trying to use this dataset be sure to pack the pixels into an image-tensor.

Here is code that parses the column names to build images for each example:

In []:

```
import re  
  
def make_images(features):  
    image = [None]*400  
    new_feats = {}  
  
    for name, value in features.items():  
        match = re.match('r(\d+)c(\d+)', name)  
        if match:  
            image[int(match.group(1))*20+int(match.group(2))] = value  
        else:  
            new_feats[name] = value  
  
    image = tf.stack(image, axis=0)  
    image = tf.reshape(image, [20, 20, -1])  
    new_feats['image'] = image  
  
    return new_feats
```

Apply that function to each batch in the dataset:

In []:

```
fonts_image_ds = fonts_ds.map(make_images)  
  
for features in fonts_image_ds.take(1):  
    break
```

Plot the resulting images:

In []:

```
from matplotlib import pyplot as plt  
  
plt.figure(figsize=(6,6), dpi=120)  
  
for n in range(9):  
    plt.subplot(3,3,n+1)  
    plt.imshow(features['image'][..., n])  
    plt.title(chr(features['m_label'][n]))  
    plt.axis('off')
```

Lower level functions

So far this tutorial has focused on the highest level utilities for reading csv data. There are other two APIs that may be helpful for advanced users if your use-case doesn't fit the basic patterns.

- `tf.io.decode_csv` - a function for parsing lines of text into a list of CSV column tensors.
- `tf.data.experimental.CsvDataset` - a lower level csv dataset constructor.

This section recreates functionality provided by `make_csv_dataset`, to demonstrate how this lower level functionality can be used.

`tf.io.decode_csv`

This function decodes a string, or list of strings into a list of columns.

Unlike `make_csv_dataset` this function does not try to guess column data-types. You specify the column types by providing a list of `record_defaults` containing a value of the correct type, for each column.

To read the Titanic data **as strings** using `decode_csv` you would say:

In []:

```
text = pathlib.Path(titanic_file_path).read_text()
lines = text.split('\n')[1:-1]

all_strings = [str()] * 10
all_strings
```

In []:

```
features = tf.io.decode_csv(lines, record_defaults=all_strings)

for f in features:
    print(f"type: {f.dtype.name}, shape: {f.shape}")
```

To parse them with their actual types, create a list of `record_defaults` of the corresponding types:

In []:

```
print(lines[0])
```

In []:

```
titanic_types = [int(), str(), float(), int(), int(), float(), str(), str(), str(), str()]
titanic_types
```

In []:

```
features = tf.io.decode_csv(lines, record_defaults=titanic_types)

for f in features:
    print(f"type: {f.dtype.name}, shape: {f.shape}")
```

Note: it is more efficient to call `decode_csv` on large batches of lines than on individual lines of csv text.

`tf.data.experimental.CsvDataset`

The `tf.data.experimental.CsvDataset` class provides a minimal CSV Dataset interface without the convenience features of the `make_csv_dataset` function: column header parsing, column type-inference, automatic shuffling, file interleaving.

This constructor follows uses `record_defaults` the same way as `io.parse_csv`:

In []:

```
simple_titanic = tf.data.experimental.CsvDataset(titanic_file_path, record_defaults=titanic_types, header=True)

for example in simple_titanic.take(1):
    print([e.numpy() for e in example])
```

The above code is basically equivalent to:

```
In [ ]:
```

```
def decode_titanic_line(line):
    return tf.io.decode_csv(line, titanic_types)

manual_titanic = (
    # Load the lines of text
    tf.data.TextLineDataset(titanic_file_path)
    # Skip the header row.
    .skip(1)
    # Decode the line.
    .map(decode_titanic_line)
)

for example in manual_titanic.take(1):
    print([e.numpy() for e in example])
```

Multiple files

To parse the fonts dataset using `experimental.CsvDataset`, you first need to determine the column types for the `record_defaults`. Start by inspecting the first row of one file:

```
In [ ]:
```

```
font_line = pathlib.Path(font_csvs[0]).read_text().splitlines()[1]
print(font_line)
```

Only the first two fields are strings, the rest are ints or floats, and you can get the total number of features by counting the commas:

```
In [ ]:
```

```
num_font_features = font_line.count(',')+1
font_column_types = [str(), str()] + [float()]* (num_font_features - 2)
```

The `CsvDataset` constructor can take a list of input files, but reads them sequentially. The first file in the list of CSVs is `AGENCY.csv`:

```
In [ ]:
```

```
font_csvs[0]
```

So when you pass the list of files to `CsvDataset` the records from `AGENCY.csv` are read first:

```
In [ ]:
```

```
simple_font_ds = tf.data.experimental.CsvDataset(
    font_csvs,
    record_defaults=font_column_types,
    header=True)
```

```
In [ ]:
```

```
for row in simple_font_ds.take(10):
    print(row[0].numpy())
```

To interleave multiple files, use `Dataset.interleave`.

Here's an initial dataset that contains the csv file names:

```
In [ ]:
```

```
font_files = tf.data.Dataset.list_files("fonts/*.csv")
```

This shuffles the file names each epoch:

```
In [ ]:
```

```
print('Epoch 1:')
for f in list(font_files)[:5]:
    print("    ", f.numpy())
print('    ...')
print()

print('Epoch 2:')
for f in list(font_files)[:5]:
    print("    ", f.numpy())
print('    ...')
```

The `interleave` method takes a `map_func` that creates a child- `Dataset` for each element of the parent- `Dataset` .

Here, you want to create a `CsvDataset` from each element of the dataset of files:

In []:

```
def make_font_csv_ds(path):
    return tf.data.experimental.CsvDataset(
        path,
        record_defaults=font_column_types,
        header=True)
```

The `Dataset` returned by `interleave` returns elements by cycling over a number of the child- `Dataset` s. Note, below, how the dataset cycles over `cycle_length=3` three font files:

In []:

```
font_rows = font_files.interleave(make_font_csv_ds,
                                   cycle_length=3)
```

In []:

```
fonts_dict = {'font_name':[], 'character':[]}

for row in font_rows.take(10):
    fonts_dict['font_name'].append(row[0].numpy().decode())
    fonts_dict['character'].append(chr(row[2].numpy()))

pd.DataFrame(fonts_dict)
```

Performance

Earlier, it was noted that `io.decode_csv` is more efficient when run on a batch of strings.

It is possible to take advantage of this fact, when using large batch sizes, to improve CSV loading performance (but try [caching](#) first).

With the built-in loader 20, 2048-example batches take about 17s.

In []:

```
BATCH_SIZE=2048
fonts_ds = tf.data.experimental.make_csv_dataset(
    file_pattern = "fonts/*.csv",
    batch_size=BATCH_SIZE, num_epochs=1,
    num_parallel_reads=100)
```

In []:

```
%%time
for i,batch in enumerate(fonts_ds.take(20)):
    print('.',end='')

print()
```

Passing **batches of text lines** to `decode_csv` runs faster, in about 5s:

In []:

```
fonts_files = tf.data.Dataset.list_files("fonts/*.csv")
fonts_lines = fonts_files.interleave(
    lambda fname:tf.data.TextLineDataset(fname).skip(1),
    cycle_length=100).batch(BATCH_SIZE)

fonts_fast = fonts_lines.map(lambda x: tf.io.decode_csv(x, record_defaults=font_column_types))
```

In []:

```
%%time
for i,batch in enumerate(fonts_fast.take(20)):
    print('.',end='')

print()
```

For another example of increasing csv performance by using large batches see the [overfit and underfit tutorial \(../keras/overfit_and_underfit.ipynb\)](#).

This sort of approach may work, but consider other options like `cache` and `snapshot`, or re-encoding your data into a more streamlined format.