ZTM

TENSORFLOW CHEAT SHEET

DANIEL BOURKE

V1.01

HEEELLLOOOOO!

I'm Andrei Neagoie, Founder and Lead Instructor of the Zero To Mastery Academy.

After working as a Senior Software Developer over the years, I now dedicate 100% of my time to teaching others in-demand skills, help them break into the tech industry, and advance their careers.

In only a few years, **over 1,000,000 students** around the world have taken Zero To Mastery courses and many of them are now working at top tier companies like Apple, Google, Amazon, Tesla, IBM, Facebook, and Shopify, just to name a few.

This cheat sheet, created by our TensorFlow instructor (Daniel Bourke) provides you with the key TensorFlow information and concepts that you need to know.

If you want to not only learn TensorFlow but also get the exact steps to build your own projects and get hired as a Machine Learning Engineer, then check out our Career Paths.

Happy Coding!
Andrei

[illegible]

Founder & Lead Instructor, Zero To Mastery

Andrei Neagoie

TensorFlow Cheatsheet

TensorFlow in 10 Seconds

TensorFlow is an open-source machine learning library developed by Google. It's designed to be flexible and efficient for all kinds of machine learning tasks.

It has the big advantage of being able to leverage accelerated computing hardware (GPUs and TPUs) for faster machine learning model training.

This cheatsheet provides a quick reference guide for beginners and expert users, along with commentary to help you understand the code.

It is focused on the Python API of TensorFlow, however, the principles can be applied to other languages.

All techniques and code mentioned in this cheatsheet are referenced via the TensorFlow documentation.

Many of the concepts discussed in this cheatsheet are covered as hands-on coding materials in the Zero To Mastery TensorFlow Bootcamp course. You can get started by:

- 💻 Take the course at the ZTM Academy (the first 3 hours are free!)
- 🦾 Get all of the code on GitHub
- 📖 Read the beautiful online book version of the course at learntensorflow.io

TensorFlow Key Terms

Tensors

Tensors are the fundamental building blocks of TensorFlow and machine learning in general.

They are multi-dimensional arrays that can hold numerical data of various types, such as integers, floats, and booleans.

You can represent almost any kind of data (images, rows and columns, text) as some kind of tensor.

In TensorFlow, tensors are represented as `tf.Tensor` objects, which are immutable and have a specified data type and shape.

You can create a tensor with values `[1, 2, 3]` using:

```
import tensorflow as tf

# Create a simple tensor
tensor_A = tf.constant([1, 2, 3])
```

However, most of the time you won't be creating tensors by hand, as TensorFlow will provide functionality for loading (creating tensors) and manipulating data (finding patterns in tensors).

Variables

Variables are tensors that can be modified during model training.

They are used to represent neural network model parameters, such as weights and biases.

These parameters can then be adjusted to better represent data.

For example, your tensors may start with random values to begin with but during training these random values get changed to be more aligned with ideal values that represent your data.

In TensorFlow, variables are created using the `tf.Variable` class, which initializes the variable with a specified initial value and data type.

Neural Network Layers

Layers are the building blocks of neural networks.

They represent the transformations (in the form of mathematical operations) applied to the input data, and can include operations such as convolution, pooling, and activation functions.

In TensorFlow, layers are represented as `tf.keras.layers.Layer` objects, which can be combined to create complex neural network architectures.

For example, you can create a convolutional layer with:

```
import tensorflow as tf

# Create a 2D convolutional layer with TensorFlow
conv_layer = tf.keras.layers.Conv2D()
```

Neural Network Models

In TensorFlow, you can create neural network models to help solve machine learning problems.

TensorFlow provides building blocks for:

- The model architecture (the layers that make up the model).
- The parameters (the patterns or weights and bias values that the model will learn during training).
- The training process (optimizing a model's parameters to lower a loss function).

In TensorFlow, models are typically created using the `tf.keras.Model` class, which provides a high-level API for building and training models.

Loss Functions

The whole goal of machine learning is to build a model which can make some kind of prediction given a piece of input data.

Loss functions are used to measure the difference between the predicted and true values in a machine learning problem.

The higher the loss value, the more wrong the model.

So the lower the loss value, the better.

A model with a loss value of 0 is either:

- Broken (because there has been a data leak and the model has seen data it was supposed to test on).
- Perfect (it predicts the unseen data based on patterns its learn in the training data perfectly).

Loss functions are used to guide the training process and update the model parameters.

In TensorFlow, loss functions are represented as `tf.keras.losses` objects, which can be customized or chosen from a wide range of built-in options.

Optimizers

Optimizers are used to update the model parameters during training, based on the gradients of the loss function.

They are responsible for finding the optimal values of the model parameters that minimize the loss.

In essence, an optimizer will adjust the weights and bias values in a neural network's layers to make sure the model's patterns better match the input data.

If the optimizer does its job right and the model's learned parameters represent the target data better, the loss value should go down.

In TensorFlow, optimizers are represented as `tf.keras.optimizers` objects, which can be customized or chosen from a wide range of built-in options.

Metrics

Metrics are used to evaluate the performance of a machine learning model during training and validation.

They provide additional information beyond the loss function, such as accuracy or precision.

One defining factor of metrics is that they are generally more human-readable than loss function values.

In TensorFlow, metrics are represented as `tf.keras.metrics` objects, which can be customized or chosen from a wide range of built-in options.

Callbacks

Callbacks are functions that are executed at various stages of training, allowing you to customize and monitor the training process.

They can be used to implement several helpful auxiliary functions to training such as:

- **Early stopping** — stop a model from training if it hasn't improved in a certain amount of steps.
- **Model checkpointing** — save a model's progress at specified points.
- **Learning rate scheduling** — adjust a model's learning rate (to better facilitate learning) at programmatic intervals.

In TensorFlow, callbacks are represented as `tf.keras.callbacks` objects, which can be customized or chosen from a wide range of built-in options.

Preprocessing data with TensorFlow

Preprocessing is the process of preparing the data for use in a machine learning model.

It can include tasks such as normalization, scaling, and feature engineering.

In TensorFlow, preprocessing is typically done using the `tf.data` module, which provides tools for loading, transforming, and batching data or directly via model layers with `tf.keras.layers`.

Data Augmentation

Data augmentation is a technique used in computer vision to increase the size and diversity of the training dataset, by applying transformations to the data, such as rotations, flips, and color shifts.

It can help to prevent **overfitting** (a model learning the patterns of the data *too* well) and improve the generalization of the model.

In TensorFlow, data augmentation can be implemented using various layers from `tf.keras.layers`, which provides a wide range of image transformations, such as:

- `tf.keras.layers.Resizing` — resize an image to a target size.
- `tf.keras.layers.Rescaling` — rescale an image to a target scale.
- `tf.keras.layers.RandomFlip` — randomly flip an image (horizontal or vertical).

Transfer Learning

Transfer learning is a technique used to leverage the knowledge learned from pre-trained models, in order to improve the performance of a new model on a related task.

For example, you could take a model trained to predict on images of cars and adjust it to be able to predict on trucks (from one domain to another similar but slightly different domain).

The patterns learned from predicting on cars can often help with predicting on trucks.

It involves freezing some or all of the layers in the pre-trained model, and training only the new layers on the new data.

In TensorFlow, transfer learning can be implemented using the `tf.keras.applications` module, which provides a wide range of pre-trained models on various tasks.

In general, **when approaching a new machine learning problem, it is advised to try and use transfer learning wherever possible**.

The Zero To Mastery TensorFlow Bootcamp course covers transfer learning in sections 04, 05, and 06.

Checkpointing

Model checkpointing is the process of saving the model parameters during training, so that training can be resumed from the last saved checkpoint in case of interruption or failure.

In TensorFlow, checkpointing can be implemented using the `tf.keras.callbacks.ModelCheckpoint` callback, which saves the model weights specified steps.

TensorFlow Fundamental Building Blocks

The following section goes through some of the fundamental building blocks of TensorFlow with code examples.

Many of these are covered extensively in the Zero To Mastery TensorFlow Bootcamp course in sections 00, 01 and 02.

1. Installing TensorFlow

There are several ways to install TensorFlow, one way is to use pip in your terminal:

```
pip install tensorflow
```

For more ways to install TensorFlow such as via a Docker container, see the TensorFlow install page.

2. Importing TensorFlow

The common way to import TensorFlow with Python is to use the abbreviation `tf`:

```
import tensorflow as tf
```

3. Basic Tensor Operations

Tensor operations are used to create, manipulate or analyze tensors. These operations covered in the Zero To Mastery TensorFlow Bootcamp course section 00.

Create Tensors

Tensors are the basic building blocks in TensorFlow. They are used to represent some kind of data in numerical form.

You can create tensors using `tf.constant`:

```
scalar = tf.constant(42)
vector = tf.constant([1, 2, 3])
matrix = tf.constant([[1, 2], [3, 4]])
```

Note: As mentioned before, TensorFlow will do most of the tensor creation behind the scenes for you. In a typical machine learning workflow, it is rare to create tensors by hand.

Shape, Rank, and Size

There are a few important attributes to tensors such as:

- `tf.shape` — What are the dimensions/shape of the tensor?
- `tf.rank` — How many dimensions are in the tensor?
- `tf.size` — How many elements are in the tensor?

```
# What are the dimenions of the tensor? (e.g. [224, 224, 3] -> [height, width, color_channels] for an image)
tensor_shape = tf.shape(tensor)

# How many dimensions are in the tensor? (e.g. [224, 224, 3] = 3 dimensions)
tensor_rank = tf.rank(tensor)

# How many elements are in the tensor? (e.g. 224*224*3 = 150528)
tensor_size = tf.size(tensor)
```

Note: The shape, rank and size of a tensor will vary widely depending on the data you're working with. For example, 2D images generally take on the shape of `[height, width, color_channels]` and text could have the shape `[tokens, embedding_dimension]` where a token is a representation of a word/letter and embedding dimension is a numerical learnable tensor representing that token.

Reshape Tensors

One of the main issues in machine learning and deep learning and AI in general is making sure your tensor shapes line up.

This is due to the requirements of matrix multiplication.

If necessary, you can change the shape of a tensor using `tf.reshape`:

```
reshaped_tensor = tf.reshape(tensor_to_reshape, [new_shape])
```

You can also transpose a tensor (switch its dimensions) via `tf.transpose`:

```
transposed_tensor = tf.transpose(tensor_to_transpose)
```

Basic Math Operations

TensorFlow includes options for performing element-wise addition, subtraction, multiplication, and division:

```
addition = tf.add(a, b)
subtraction = tf.subtract(a, b)
multiplication = tf.multiply(a, b)
division = tf.divide(a, b)
```

Note: These are equivalent to their single operator equivalents in Python, for example, `a + b == tf.add(a, b)`.

4. Variables and GradientTape

Create Variables

Variable tensors are mutable (changable) tensors that store weights and biases in neural networks.

These are updated during neural network training to better represent the data a model is training on.

They can be created with `tf.Variable`:

```
variable = tf.Variable(initial_value)
```

GradientTape

You can use `tf.GradientTape` to record operations on variables for automatic differentiation.

This is helpful for creating your own training loops with TensorFlow:

```
with tf.GradientTape() as tape:
    # Operations involving variables
    gradients = tape.gradient(loss, [list_of_variables])
```

5. Creating Neural Network Layers and Models

A neural network model is a collection of layers.

A layer is a defined set of computations which takes a given tensor as input and produces another tensor as output.

For example, a simple layer could just add 1 to all the elements of an input tensor.

The important point is that a layer manipulates (performs a mathematical operation on) an input tensor in *some* way to produce an output tensor.

Combine a series of layers together and you have a model.

The term “deep learning” comes from the stacking of large numbers of layers on top of eachother (deep in this sense is a synonym for large).

The best way to stack layers together to find patterns in data is constantly changing.

This is why techniques such as **transfer learning** are helpful because they allow you to leverage what has worked for someone else’s similar problem and tailor it to your own.

Layers

Create layers using the `tf.keras.layers` module.

Examples include Dense (fully connected) and Conv2D (convolutional) layers:

```
# Create a dense (or fully connected) layer with TensorFlow
dense_layer = tf.keras.layers.Dense(units,
                                    activation=tf.keras.activations.relu)

# Create a 2D convolutional layer with TensorFlow
conv2d_layer = tf.keras.layers.Conv2D(filters,
                                      kernel_size,
                                      activation=tf.keras.activations.relu)
```

And there are many more pre-built layer types available in the TensorFlow documentation.

Neural Network Models in TensorFlow

The most straightforward way in TensorFlow to a neural network model us using the `tf.keras.Sequential` API, which allows you to stack layers in a way that the computation will be performed sequentially (one layer after the other):

```
# Create a basic neural network model with TensorFlow
model = tf.keras.Sequential([
    tf.keras.layers.InputLayer(input_shape),
    # Add layers here
    # Computation will happen layer by layer sequentially
])
```

Compile Model

Before we start training the model on data we’ve got to compile it model with an optimizer, loss function, and metric.

All three of these are customizable depending on the project you’re working on.

- **Loss function** = measures how wrong the model is (the higher the loss, the more wrong the model, so lower is better). Common loss values include `tf.keras.losses.CategoricalCrossentropy` for multi-class classification problems and `tf.keras.losses.BinaryCrossentropy` for binary classification problems.
- **Optimizer** = tries to adjust the models parameters to lower the loss value. Common optimizers include `tf.keras.optimizers.SGD` (Stochastic Gradient Descent) and `tf.keras.optimizers.Adam`. Each optimizer comes with generally good default settings, however, of the most important values to set is the `learning_rate` hyperparameter.
- **Metric** = human readable value to interpret how the model is going. Metrics can be defined via `tf.keras.metrics` such as `tf.keras.metrics.Accuracy` or via a list of strings such as `['accuracy']`.

```
# Compile model before training
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
              loss=tf.keras.losses.CategoricalCrossentropy(),
              metrics=['accuracy'])
```

Train Model

You can train your TensorFlow models using the `tf.keras.Model.fit` method and passing it the appropriate hyperparameters.

```
# Fit the model to a set of data
model.fit(x=X_train, # data to find patterns in
          y=y_train, # labels for x
          epochs=1, # how many times to go over the whole dataset
          batch_size=32, # number of samples to look at each time
          validation_data=(X_val, # data to validate the learned patterns
                           y_val))
```

Evaluate Model

Once you've trained a model, you can evaluate its performance on unseen data with `tf.keras.Model.evaluate`:

```
# Evaluate a model on unseen data
loss, accuracy = model.evaluate(x=X_test,
                                y=y_test)
```

Save and Load a Whole TensorFlow Model

There are two main ways to save a whole TensorFlow model:

- `SavedModel` format — default format for saving a model in TensorFlow, saves a complete TensorFlow program including trained parameters and does not require the original model building code to run. Useful for sharing models across TensorFlow.js, TFLite, TensorFlow Serving and more.
- `HDF5` format — a more widely used data standard, however, does not contain as much information as the `SavedModel` format.

You can read the pros and cons of each of these in the TensorFlow documentation on Save and load Keras models.

```
# Save model to SavedModel format (default)
model.save('model_save_path')

# Load model back in from SavedModel format
loaded_model = tf.keras.models.load_model('model_save_path')
```

Saving and Loading Model Weights

In addition to saving and loading the entire model, you can save and load only the model weights.

```
# Save only the model weights
model.save_weights('model_weights')
```

Once the weights are saved, you can load them back in:

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

# Load in the saved weights
model.load_weights('model_weights')
```

This can be useful if you'd like to stop training and resume it later in the same place where you left off.

6. Data Preprocessing

There are several ways to preprocess data with TensorFlow and Keras.

You can use the Keras layers API to create standalone preprocessing pipelines or preprocessing code that becomes part of a `SavedModel`.

You can see a full list of preprocessing options in the TensorFlow preprocesisng layers documentation.

Preprocessing layers include layers for:

- Text preprocessing such as `tf.keras.layers.TextVectorization` (turns text into numbers).
- Numerical features preprocessing such as `tf.keras.layers.Normalization`.
- Categorical features preprocessing such as `tf.keras.layers.CategoryEncoding`.
- Image preprocessing such as `tf.keras.layers.Resizing` to resize a batch of images to a target size.

You can also use `tf.keras.layers` to perform image data augmentation during training:

```
# Create a series of data augmentation steps as layers
# (these could be used in a model)
data_augmentation_steps = tf.keras.Sequential(
    layers=[tf.keras.layers.RandomCrop(224, 224),
            tf.keras.layers.RandomFlip(),
            tf.keras.layers.RandomHeight(0.2)]
)
```

7. Transfer Learning

Transfer learning is a technique to leverage a pre-trained model for a new task with similar data.

For example, leveraging a model trained a diverse range of images (such as ImageNet, a collection of 1000 different classes of images) and tailoring it to predict specifically on your own task of predicting on dogs.

Or a model trained on large amounts of text data (e.g. all of Wikipedia, all Reddit and more) and customizing it to predict whether the text of an insurance claim was fraud or not.

Note: Transfer Learning is covered in-depth in the Zero To Mastery TensorFlow Bootcamp course sections 04, 05, 06.

Load Image Pretrained Models with TensorFlow

You can find many image-based pretrained models in the `tf.keras.applications` API.

For example, to load a pre-trained TensorFlow model (e.g., MobileNetV2, a fast computer vision model which can run on mobile devices) without the top classification layer:

```
pretrained_model = tf.keras.applications.MobileNetV2(
    input_shape=input_shape, # define input shape of your data
    include_top=False, # customize top layer to your own problem
    weights='imagenet') # model weights have been pretrained on ImageNet
```

Load Text and Other Pretrained Models with TensorFlow Hub

There are many more models across different domains that can be loaded from TensorFlow Hub or tfhub.dev.

These include models for images, text, video, audio and more.

You can also load the models via the TensorFlow Python API or JavaScript API.

For example, the LEALLA (Lightweight language-agnostic sentence embedding model supporting 109 languages) is great for encoding text into numbers and then finding patterns in those numbers such as similarity matching (e.g. matching two pieces of text which are similar semantically rather than just string matching).

You can load it with:

```
import tensorflow_hub as hub
import tensorflow as tf
import numpy as np

# Several sizes of model are available
encoder = hub.KerasLayer("https://tfhub.dev/google/LEALLA/LEALLA-small/1")
# encoder = hub.KerasLayer("https://tfhub.dev/google/LEALLA/LEALLA-base/1")
# encoder = hub.KerasLayer("https://tfhub.dev/google/LEALLA/LEALLA-large/1")
# Works for different languages
english_sentences = tf.constant(["dog", "Puppies are nice.", "I enjoy taking long walks along the beach with my dog."])
italian_sentences = tf.constant(["cane", "I cuccioli sono carini.", "Mi piace fare lunghe passeggiate lungo la spiaggia con il mio cane."])
japanese_sentences = tf.constant(["犬", "子犬はいいです", "私は犬と一緒にビーチを散歩するのが好きです"])

# Turn target texts into numbers
english_embeds = encoder(english_sentences)
japanese_embeds = encoder(japanese_sentences)
italian_embeds = encoder(italian_sentences)

# English-Italian similarity (how similar are the texts)
print(np.matmul(english_embeds, np.transpose(italian_embeds)))

# English-Japanese similarity (how similar are the texts)
print(np.matmul(english_embeds, np.transpose(japanese_embeds)))

# Italian-Japanese similarity (how similar are the texts)
print(np.matmul(italian_embeds, np.transpose(japanese_embeds)))
```

Freeze Pretrained Model Layers

To preserve the patterns and weights learned by a pretrained model (to reuse them for your own problem), you can *freeze* them to stop them being updated during training:

```
# Set layers in the pretrained_model to not update during training
pretrained_model.trainable = False
```

Add Custom Layers and Train

With a frozen pretrained model, you can add your own custom layers on top:

```
model = tf.keras.Sequential([
    pretrained_model, # frozen base model

    # Custom top layers take outputs from pretrained model and tailor to your
    # own problem
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(num_classes, activation='softmax')
])

model.compile(...)
model.fit(...)
```

8. Creating more advanced models with the Functional API

The TensorFlow `tf.keras.Sequential` API is a quick way to make a model which processes inputs sequentially (layer by layer).

Where as the TensorFlow Functional API offers a far more flexible way to create models.

For example, with the Functional API, you can create models with:

- Multiple inputs (e.g images and text rather than just images).
- Non-linear steps (e.g. different connections between layers rather than just sequential).
- Shared layers (e.g. share layers between various streams of data).

The following is an example of creating a one hidden layer model via the Functional API:

```
# Simple one hidden layer model with the Functional API
inputs = tf.keras.Input(shape=input_shape)
x = tf.keras.layers.Dense(units, activation='relu')(inputs) # notice the inputs passed on the outside of the layer
outputs = tf.keras.layers.Dense(num_classes, activation='softmax')(x)

# Combine the inputs and the outputs into a model (TensorFlow works out how they're connected)
model = tf.keras.Model(inputs=inputs, outputs=outputs)
```

Multi-Input and Multi-Output Models with the Functional API

One of the big advantages of the Functional API is being able to create models which can handle multiple inputs and produce multiple outputs.

The following is an example of a model that takes two input sources, performs a computation on them with a shared layer and then outputs two outputs:

```
# Setup two input layers
input1 = tf.keras.Input(shape=input_shape1)
input2 = tf.keras.Input(shape=input_shape2)

# Create a shared layer to compute on both inputs
shared_layer = tf.keras.layers.Dense(units, activation='relu')
x1 = shared_layer(input1)
x2 = shared_layer(input2)

# Pass outputs of shared layer to multiple output layers
output1 = tf.keras.layers.Dense(num_classes1, activation='softmax')(x1)
output2 = tf.keras.layers.Dense(num_classes2, activation='softmax')(x2)

# Create a model with multi-inputs and multi-outputs
model = tf.keras.Model(inputs=[input1, input2], # take multiple inputs
                       outputs=[output1, output2]) # output multiple outputs
```

The Functional API allows you to get as creative as you'd like connecting inputs, intermediate layers and output layers.

Note: Multi-input models with the TensorFlow Functional API are covered in the Zero To Mastery TensorFlow Bootcamp course section 09.

9. Callbacks

Callbacks are functions that are executed at various stages of training, allowing you to customize and monitor the training process.

You can access callbacks via the `tf.keras.callbacks` API.

Some examples of popular and useful callbacks include:

EarlyStopping

EarlyStopping stops a model from training when a defined metric has stoped improving.

For example, if you wanted to stop your model from training if the validation loss (represented with the string `'val_loss'`) hadn't improved for 5 epochs, you could write the following:

```
# Stop model training after 5 epochs of validation loss not improving
early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss',
                                                  patience=5)
```

ModelCheckpoint

ModalCheckpoint saves your models weights/training progress at defined intervals.

For example, if you'd like to save your model at the end of every epoch, and only save it if the model is the *best* performing model so far according to the validation loss, you could write:

```
# Create a checkpoint callback to save the best model every epoch (if it's better than the previously saved model)
checkpoint = tf.keras.callbacks.ModelCheckpoint(filepath='model-{epoch:02d}',
                                                 save_best_only=True,
                                                 monitor='val_loss')
```

Learning Rate Scheduler

One of the most important hyperparameters when training a model is the learning rate.

A typical workflow is that the longer training goes on, the smaller the learning rate should get.

For example, you could define a schedule (via a Python function) to lower the learning rate by 10x every 10 epochs.

And then use the `tf.keras.callbacks.LearningRateScheduler` callback to implement the schedule during training:

```
def lr_schedule(epoch, lr):
    """
    Reduce the learning rate by 10x every 10 epochs
    """
    if epoch % 10 == 0:
        return lr * 0.1
    return lr

lr_callback = tf.keras.callbacks.LearningRateScheduler(lr_schedule)
```

Using Callbacks in Training

To use callbacks, pass a list of callback instances to the `fit` method with the `callbacks` parameter:

```
model.fit(x=x_train,
          y=y_train,
          epochs=10,
          batch_size=32,
          validation_data=(x_val, y_val),
          callbacks=[early_stopping, # pass in a list of callbacks to use
                     checkpoint,
                     lr_callback])
```

10. Custom Layers with TensorFlow

TensorFlow has plenty of in-built layers available via `tf.keras.layers`.

These will get you through 95% of your model creation.

However, sometimes you may want to create your own custom layer.

To do so, you can subclass the `tf.keras.layers.Layer` class (the class from which all other layers are built on) and implement:

1. `__init__` — create all objects which are required for the layer (these are generally input-independent).
2. A `build()` method — setup all input-dependent variables for the layer.
3. A `call()` method — setup all steps in the forward computation (when you call the layer, what should it do to the inputs?).

For example, let's create a layer which just adds 1 to all variables on the input:

```
import tensorflow as tf

# Create all zeros tensor as input
x = tf.zeros(10)

# Create a layer which just adds 1 to all inputs
class MyCustomLayer(tf.keras.layers.Layer):
    def __init__(self):
        super(MyCustomLayer, self).__init__()
        # Initialize layer attributes

    def build(self, input_shape):
        # Create layer weights (our layer doesn't use these)
        self.kernel = self.add_weight("kernel")

    def call(self, inputs):
        # Define the forward pass (what happens when you call the layer?)
        return inputs + 1 # add 1 to all inputs

# Create and use the layer
custom_layer = MyCustomLayer()
custom_layer(x)

>>> <tf.Tensor: shape=(10,), dtype=float32, numpy=array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=float32)>
```

For more on creating custom layers, see the TensorFlow tutorial on Custom layers.

11. Custom models via Model Subclassing

Once you've created custom layers, you can stack them together using the Sequential API or the Functional API.

However, you may also want to combine them via your own custom model class.

You create your own custom model by subclassing `tf.keras.Model`.

A `tf.keras.Model` is just like a `tf.keras.layers.Layer` but with added functionality such as the `fit()`, `evaluate()` and `save()` methods.

Just like when creating a layer, you must define the `call()` method which defines the computation that happens when you call `fit()` on the model:

```
# Create a custom model
class MyCustomModel(tf.keras.Model):
    def __init__(self, ...):
        super(MyCustomModel, self).__init__()
        # Initialize and create layers

    # call() defines what happens when you call fit() on the model
    def call(self, inputs, training=False):
        # Define the forward pass
        return ...

# Use your custom model
custom_model = MyCustomModel(...)
```

12. Custom Loss Functions

The loss function measures how wrong a model's predictions are.

The more wrong a model is, the higher the loss value will be.

TensorFlow has many in-built loss functions available in the `tf.keras.losses` API.

But if you'd like to create your own loss function, you can do so by subclassing `tf.keras.losses.Loss` (this is what all existing loss functions already do).

For example, if we wanted to recreate mean absolute error (MAE), we could do so with:

```
# Create y_true and y_pred values
y_true = tf.ones(10)
y_pred = y_true * 2

# Create custom loss (recreate MAE)
class MeanAbsoluteError(tf.keras.losses.Loss):
  def call(self, y_true, y_pred):
    return tf.reduce_mean(tf.abs(y_pred - y_true))

# Instantiate and use the custom loss value
mae_loss = MeanAbsoluteError()
mae_loss(y_true, y_pred)

>>> <tf.Tensor: shape=(), dtype=float32, numpy=1.0>

# Compile the model with the custom loss
model.compile(optimizer=optimizer,
              loss=mae_loss,
              metrics=metrics)
```

13. Custom Metrics

A metric is a human-readable value to measure how *right* your model is performing.

For example, if a model achieved 99/100 correct predictions, you might say it had an accuracy of 99% (where *accuracy* is the metric of choice).

TensorFlow has a bunch of pre-built metrics for all kinds of different problems available in `tf.keras.metrics`.

However, if you'd like to create your own custom metric, you can do so via subclassing `tf.keras.metrics.Metric` (the class all other existing TensorFlow metrics build on):

```
# Create your own custom metric
class MyCustomMetric(tf.keras.metrics.Metric):
    def __init__(self, ...):
        super(MyCustomMetric, self).__init__()
        # Initialize metric attributes

    def update_state(self, y_true, y_pred, sample_weight=None):
        # Update the metric's state based on y_true and y_pred

    def result(self):
        # Compute and return the final result

    def reset_states(self):
        # Reset the metric's state

# Use your own custom metric
custom_metric = MyCustomMetric(...)
```

14. Custom Optimizers with TensorFlow

TensorFlow has many common and useful optimizers built into `tf.keras.optimizers` such as Adam and SGD (stochastic gradient descent).

But perhaps you'd like to try out creating your own custom optimizer.

To do so, you can subclass `tf.keras.optimizers.Optimizer`.

Creating your own custom optimizer requires overriding the `build()`, `update_step()` and `get_config()` methods:

```
class MyCustomOptimizer(tf.keras.optimizers.Optimizer):
    def __init__(self, ...):
        super(MyCustomOptimizer, self).__init__(...)

    def build(self):
        # Create optimizer-related variables such as momentum in SGD

    def update_step(self):
        # Implement the optimizer's updating logic

    def get_config(self):
        # Return optimizer configuration
```

15. Custom Training Loops with TensorFLow

Instead of calling `model.fit()` on your data, (which is powerful but abstracts away many steps) for more control over your training, you can create your own custom training loops:

```
# Create optimizer and loss function
optimizer = tf.keras.optimizers.Adam(learning_rate)
loss_function = tf.keras.losses.CategoricalCrossentropy()

# Create a custom training loop
for epoch in range(epochs):
    for x_batch, y_batch in train_data:
        with tf.GradientTape() as tape:
            # Forward pass
            y_pred = model(x_batch, training=True)
            # Compute loss
            loss = loss_function(y_batch, y_pred)

        # Compute gradients
        gradients = tape.gradient(loss, model.trainable_variables)

        # Apply gradients
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))
```

This allows for greater flexibility, such as implementing custom learning rate schedules, gradient clipping, or using multiple optimizers.

For more on custom training loops in TensorFlow, see the Writing a training loop from scratch guide in the TensorFlow documenation.

16. Measure your TensorFlow model results with TensorBoard

TensorBoard is a visualization tool that comes with TensorFlow.

It's main use case is to help you monitor model experiments and understand your models better.

It can help you see interactive versions of:

- Training progress
- Model graphs
- Metrics
- Data exploration

One of most common workflows of using TensorBoard is to incorporate it as a callback during model training.

Doing so will save various model training logs to a specified directory for later viewing.

Setup TensorBoard Logging Directory

To use TensorBoard, start by creating a log directory:

```
import os

log_dir = "logs"
if not os.path.exists(log_dir):
    os.makedirs(log_dir)
```

Setup TensorBoard Callback

Once the logging directory is created, a TensorBoard instance can be setup via the `tf.keras.callbacks.TensorBoard` callback.

And then we can use it during training by passing it to the `fit` method:

```
# Create tensorboard callback
tensorboard = tf.keras.callbacks.TensorBoard(log_dir=log_dir
                                             # how often to log metrics
                                             update_freq='epoch')

# Fit the model to the data
model.fit(x=x_train,
          y=y_train,
          epochs=10,
          batch_size=32,
          validation_data=(x_val, y_val),
          callbacks=[tensorboard]) # use tensorboard callback when training
```

Launch TensorBoard

To view TensorBoard, run the following command in your terminal:

```
tensorboard --logdir logs
```

Then, open the provided URL in your web browser.

Or if you're running in a notebook, you can do so by first loading the `%tensorboard` magic extension and then calling it as above (except for using `%tensorboard` instead of `tensorboard`):

```
# Load the TensorBoard notebook extension
%load_ext tensorboard

# View the TensorBoard logs inside of a notebook
%tensorboard --logdir logs
```

Advanced techniques

Note: I use the term "advanced" here loosely. Not because these techniques may take advanced knowledge to know, learn or implement but because they generally come more into play later on in a TensorFlow/machine learning workflow.

17. Regularization

Because neural network models have incredible potential to learn patterns in data, they are prone to **overfitting**.

Overfitting is when a model learns patterns in a training dataset *too well.*

By too well, I mean it basically memorizes the patterns in the training set and doesn't generalize to *unseen* data such as the test dataset.

Such as when a student learns the course materials (training data) very well but fails the final exam (testing data).

There are several techniques to prevent overfitting and these techniques are collectively known as **regularization**:

- Get more training data (more data means more opportunities for a model to learn diverse patterns).
- Add dropout (randomly remove connections in the network).
- Add weight decay or weight regularization (make sure the weights don't get too *close* to the target data).
- Use data augmentation.
- Use batch normalization or layer normalization.

Weight decay/regularization

You can add weight decay/regularization to a particular layer using L1 or L2 regularization via the `tf.keras.regularizers` API (see the TensorFlow documentation for more on each of these), let's try L2 regularization:

```
# Create the regularizer
l2_regularizer = tf.keras.regularizers.l2(l2=0.01)

# Add it to the layer
layer = tf.keras.layers.Dense(units,
                              activation='relu',
                              # Note: kernel is another word for "weight matrix"
                              kernel_regularizer=l2_regularizer)
```

You can also apply regularization via string identifier, such as applying L1 and L2 regularization simultaneously with `'l1_l2'`:

```
# Create layer with L1 and L2 regularization
layer = tf.keras.layers.Dense(units,
                              activation='relu',
                              # Note: kernel is another word for "weight matrix"
                              kernel_regularizer='l1_l2')
```

You can also apply weight decay right into the optimizer using the `weight_decay` parameter such as in `tf.keras.optimizers.AdamW`:

```
adam_with_weight_decay = tf.keras.optimizers.AdamW(learning_rate=0.001,
                                                   weight_decay=0.004)
```

Dropout

Dropout is another common regularization technique which randomly sets a specified fraction of input values to zero.

The thought process here is that if a certain number (usually around 10-20%) of layer weights are set to zero, the remaining weights should be forced to be *better* representations of the data.

Dropout is usually placed after and in between layers and can be created via `tf.keras.layers.Dropout`.

```
# Create a dropout layer with a dropout rate of 10%
dropout_layer = tf.keras.layers.Dropout(rate=0.1)
```

18. Hyperparameter Tuning in TensorFlow

Values that you can set yourself, such as the learning rate or number of units in a layer, of a machine learning model are known as **hyperparameters**.

While there are often good defaults for these values, knowing the absolute *best* values is impossible ahead of time.

The practice of finding the best values for these is known as **hyperparameter tuning**.

You can either run multiple experiments adjusting the values by hand.

Or you can write code to do it for you.

The Keras Tuner library allows you to programmtically tune the hyperparameters of your TensorFlow and Keras models.

The Keras Tuner library has four tuners available:

- `RandomSearch`
- `Hyperband`
- `BayesianOptimization`
- `Sklearn`

The documentation explains each of these in more detail, however, here is an example of using `RandomSearch` to tune the hidden units in the first layer and the learning rate of the optimizer:

```
!pip install keras-tuner

import tensorflow as tf
imporkeras_tuner import RandomSearch

# Create dataset
(img_train, label_train), (img_test, label_test) = tf.keras.datasets.fashion_mnist.load_data()

# Normalize dataset
img_train = img_train.astype('float32') / 255.0
img_test = img_test.astype('float32') / 255.0

# Define a model to tune the hyperparameters of
def build_model(hp):
    # Tune the values of the first dense layer between
    # 32 and 512
    units_choice = hp.Int('units',
                          min_value=32,
                          max_value=512,
                          step=32)

    # Define the model
    model = tf.keras.Sequential([
        tf.keras.layers.Flatten(input_shape=(28, 28)), # flatten the inputs
        tf.keras.layers.Dense(units=units_choice, # these will be tuned
                              activation='relu'),
        tf.keras.layers.Dense(10,
                              activation='softmax')
    ])

    # Tune the learning rate to be one of:
    # 0.01, 0.001 or 0.0001
    learning_rate_choices = hp.Choice('learning_rate',
                                      values=[1e-2, 1e-3, 1e-4])
    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate_choices),
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    return model

# Create the tuner with RandomSearch
tuner = RandomSearch(build_model,
                     objective='val_accuracy',
                     max_trials=5)

# Run the hyperparameter search
tuner.search(x=img_train,
             y=label_train,
             epochs=10,
             validation_data=(img_test,
                              label_test))

# Get the best model
best_model = tuner.get_best_models(num_models=1)[0]
```

19. Mixed Precision Training in TensorFlow

The default datatype for computing in TensorFlow is float32.

However, lower-precision datatypes such as float16 can be used to speed up training (less precision means less numbers to compute on) without a significant loss to performance.

If your GPU has a compute capability score of 7.0 or higher, it can benefit from using mixed precision training.

TensorFlow enables automatic mixed precision training (e.g. it will allocate the float16 datatype to layers which are compatible with using it and will default to float32 when necessary) via the `tensorflow.keras.mixed_precision` API:

```
from tensorflow.keras import mixed_precision

# Set the global policy to use mixed precision where possible
policy = mixed_precision.Policy('mixed_float16')
mixed_precision.set_global_policy(policy)

# Note: when using mixed precision training, ensure the last
# layer has float32 output (for numerical stability)
model.add(tf.keras.layers.Activation('softmax', dtype='float32'))

# Train the model as usual and mixed precision will be applied where possible
```

20. Gradient Accumulation in TensorFlow

Gradient accumulation can help when training on large batches that do not fit in memory.

For example, let's say you have a large neural network which performs well but runs into memory issues when using a batch size of 32 (larger batch sizes generally enables better learning opportunities).

With gradient accumulation, you can *simulate* a larger batch size.

For example, you could run with a batch size of 32 but only update the gradients (perform the large computation) every 4 steps.

This effectively means you're using a batch size of 8 (32/4) but getting the benefits of using a batch size of 32.

This code example shows how the forward pass, loss calculation and gradient calculation happens on the whole batch but the *gradient updates* only happen once every 4 steps:

```
# Setup the optimizer and loss function as normal
optimizer = tf.keras.optimizers.Adam(learning_rate)
loss_function = tf.keras.losses.CategoricalCrossentropy()

# Accumulate gradients and only update them every 4 steps
accumulation_steps = 4

# Create a tensor of zeros to store accumulated gradients
accumulated_gradients = [tf.zeros_like(var) for var in model.trainable_variables]

# Perform training loop
for epoch in range(epochs):
    for x_batch, y_batch in train_data:
        with tf.GradientTape() as tape:
            # Forward pass gets performed on the whole batch
            y_pred = model(x_batch, training=True)
            # Compute loss on the whole batch
            loss = loss_function(y_batch, y_pred)

        # Compute gradients on whole batch
        gradients = tape.gradient(loss, model.trainable_variables)
        accumulated_gradients = [acc_grad + grad for acc_grad, grad in zip(accumulated_gradients, gradients)]

        # Only *update* gradients if step threshold is True
        if step % accumulation_steps == 0:
            # Divide accumulated_gradients by number of steps
            accumulated_gradients = accumulated_graidents / accumulation_steps

            # Apply accumulated gradients
            optimizer.apply_gradients(zip(accumulated_gradients, model.trainable_variables))
            # Reset accumulated gradients
            accumulated_gradients = [tf.zeros_like(var) for var in model.trainable_variables]
```

21. Track in-depth model performance with TensorFlow Profiler

The TensorFlow Profiler helps to track the performance of TensorFlow models down to finest the detail.

It helps you understand what operations (ops) are consuming what hardware resources (time and memory wise) and thus allows you to modify various parts which may be taking more time than necessary.

You can get started with the TensorFlow Profiler in several ways via the various profiling APIs:

- Via the TensorBoard Keras Callback, `tf.keras.callbacks.TensorBoard` (shown below)
- Via the `tf.profiler` Function API
- Via the `tf.profiler.experimental.Profile()` context manager

The following is an example of using the TensorFlow Profiler with the TensorBoard Callback:

```
!pip install -U tensorboard_plugin_profile

import tensorflow as tf
from tensorflow.keras.callbacks import TensorBoard

# Set up the log directory
log_dir = "logs"

# Enable the profiler in TensorBoard via callback
tensorboard_callback = TensorBoard(log_dir=log_dir,
                                   # Select which batches to profile
                                   profile_batch=(10, 20))

# Train the model with the TensorBoard callback
model.fit(x=x_train,
          y=y_train,
          epochs=10,
          batch_size=32,
          validation_data=(x_val, y_val),
          callbacks=[tensorboard_callback])
```

After training, the TensorFlow Profiler logs will be available in TensorBoard:

```
# Inside a notebook
%load_ext tensorboard
%tensorboard --logdir logs

# From the terminal
tensorboard --logdir logs
```

22. Distributed Training with TensorFlow

Distributed training involves training TensorFlow models across multiple GPUs, machines or TPUs.

This means you can take your existing code and run it at a larger scale and in turn create larger models and train on larger datasets.

TensorFlow allows for distributed training to take place via the `tf.distribute.Strategy` API.

There are several types of training strategy to use:

- Single machine, single GPU — this is the default strategy and does not require `tf.distribute.Strategy`.
- MirroredStrategy (`tf.distribute.MirroredStrategy`) — this mode supports synchronous distributed training on multiple GPUs on a single machine. Each variable is replicated across each GPU.
- TPUStrategy (`tf.distribute.TPUStrategy`) — this mode allows the running of TensorFlow code on Tensor Processing Units (TPUs). TPUs are Google's custom chips designed for machine learning workflows.
- MultiWorkerMirroredStrategy (`tf.distribute.MultiWorkerMirroredStrategy`) — this mode allows for use of multiple machines with multiple GPUs (e.g. a cluster of 10 machines each with 8 GPUs).
- See more in the TensorFlow distributed training documentation.

Let's see an example of two distributed training options.

MirroredStrategy (one machine with multiple GPUs)

`tf.distribute.MirroredStrategy` is a synchronous training strategy that replicates the model and its variables across multiple GPUs on a single machine, for example, you may have 4 GPUs on a single machine:

```
import tensorflow as tf

# Setup the training strategy
mirrored_strategy = tf.distribute.MirroredStrategy()

# Compile the model within the strategy context manager
with mirrored_strategy.scope():
    # Build and compile your model within the strategy scope
    model = create_model()
    model.compile(optimizer=optimizer, loss=loss, metrics=metrics)

# Train the model as usual
model.fit(x_train, y_train, epochs, batch_size, validation_data=(x_val, y_val))
```

See a full example of the MirroredStrategy in the TensorFlow documentation.

MultiWorkerMirroredStrategy (multiple machines, multiple GPUs)

`tf.distributed.MultiWorkerMirroredStrategy` extends `MirroredStrategy` for synchronous training across multiple machines, each with multiple GPUs:

```
import tensorflow as tf

# Setup the training strategy
multi_worker_mirrored_strategy = tf.distribute.MultiWorkerMirroredStrategy()

# Compile the model within the strategy context manager
with multi_worker_mirrored_strategy.scope():
    # Build and compile your model within the strategy scope
    model = create_model()
    model.compile(optimizer=optimizer, loss=loss, metrics=metrics)

# Train the model as usual
model.fit(x_train, y_train, epochs, batch_size, validation_data=(x_val, y_val))
```

Note: To use `tf.distributed.MultiWorkerMirroredStrategy`, youll need to create a `TF_CONFIG` environment variable on each worker machine with the cluster specification and the current worker's task information. You can see a full guide on this in the TensorFlow documentation on using Multiple workers with Keras.

TensorFlow Extensions and Related Libraries

The following resources are not part of the base TensorFlow API (they require an external library install).

But are very useful add ons for many different machine learning workflows.

23. TensorFlow Datasets

TensorFlow Datasets (TFDS) is a collection of ready-to-use datasets for TensorFlow (and other machine learning frameworks).

It includes hundreds of different ready to download datasets across various modalities from images to text to audio to biology to computer science.

You can get started by installing the `tensorflow-datasets` package:

```
pip install tensorflow-datasets
```

And then load a target dataset such as the MNIST dataset:

```
import tensorflow_datasets as tfds

# Load the MNIST dataset with information about
dataset, info = tfds.load('mnist',
                          with_info=True,
                          as_supervised=True)
```

For a full tutorial on TensorFlow Datasets, see the TensorFlow Datasets tutorial.

And for a full list of datasets available in TensorFlow Datasets, see the dataset catalog.

24. TensorFlow Lite

TensorFlow Lite allows you to convert TensorFlow models into a more efficient format for deployment on mobile and embedded devices (such as an iPhone or microcontroller or Raspberry Pi).

A common workflow is to train models with TensorFlow and Keras via the Python API and then convert the trained model to the TensorFlow Lite format for deployment on a mobile or edge device.

You can do so by first install TensorFlow Lite via the `tflite` package:

```
pip install tflite
```

Then you can either convert a SavedModel (a model you've already trained and saved, the recommended approach) or convert a Keras model to the `.tflite` format:

```
import tensorflow as tf

# Convert the model
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir) # path to the SavedModel directory
tflite_model = converter.convert()

# Save the model.
with open('model.tflite', 'wb') as f:
  f.write(tflite_model)
```

25. TensorFlow Extended (TFX)

TensorFlow Extended (TFX) is an end-to-end platform for deploying production machine learning pipelines, including data loading, preprocessing, validation, model training, and serving.

It can be installed via the `tfx` package:

```
pip install tfx
```

And a machine learning pipeline can be constructed by stringing together TFX components:

```
import tensorflow as tf
import tfx

# A pipeline typically starts by loading data
tfx.components.ExampleGen(...)
```

TensorFlow Serving

TensorFlow Serving is one part of the TFX ecosystem which provides a high-performance serving system for deploying machine learning models.

You can install TensorFlow Serving via:

- Docker images (recommended)
- `apt-get` (the APT package handling utility)

For example, here's a way to serve a demo model via TensorFlow Serving using Docker, the model will create an API for a prediction that returns half plus two of the input (e.g. an input of [1, 2, 3] becomes [2.5, 3.0, 3.5]):

```
# Download the TensorFlow Serving Docker image and repo
docker pull tensorflow/serving

git clone https://github.com/tensorflow/serving
# Location of demo models
TESTDATA="$(pwd)/serving/tensorflow_serving/servables/tensorflow/testdata"

# Start TensorFlow Serving container and open the REST API port
docker run -t --rm -p 8501:8501 \
    -v "$TESTDATA/saved_model_half_plus_two_cpu:/models/half_plus_two" \
    -e MODEL_NAME=half_plus_two \
    tensorflow/serving &

# Query the model using the predict API
curl -d '{"instances": [1.0, 2.0, 5.0]}' \
    -X POST http://localhost:8501/v1/models/half_plus_two:predict

# Returns => { "predictions": [2.5, 3.0, 4.5] }
```

26. TensorFlow Addons

TensorFlow Addons (TFA) is a collection of community-contributed extensions for TensorFlow, including:

- `tfa.callbacks` for custom callbacks.
- `tfa.image` for custom image transformations.
- `tfa.layers` for custom neural network layers.
- `tfa.optimizers` for custom optimizers.
- `tfa.losses` for custom loss functions.
- And more in the TensorFlow Addons API documentation.

These extensions are in place because machine learning moves so fast it's often hard to incorporate everything in a central library such as TensorFlow, hence the addons.

All addons follow similar API standards as the regular TensorFlow library.

You can get started with TensorFlow Addons by installing the `tensorflow-addons` package:

```
pip install tensorflow-addons
```

And we can create an example addon optimizer such as the `tf.optimizers.CyclicalLearningRate`:

```
import tensorflow_addons as tfa

# Create an optimizer from TensorFlow Addons
tfa.optimizers.CyclicalLearningRate(...)
```

27. TensorFlow Decision Forests

TensorFlow Decision Forests (TF-DF) is a library for training, running and exploring decision forest models, for example, Random Forests, Gradient Boosted Trees.

With TF-DF you can perform classification, regression and ranking tasks.

Decision forest models typically perform exceptionally well on structured data (e.g. rows and columns) where as neural networks typically perform exceptionally on unstructured data (e.g. images, text, video, audio).

You can get started with TensorFlow Decision Forests by installing the `tensorflow_decision_forests` package:

```
# Install TensorFlow Decision Forests
pip3 install tensorflow_decision_forests --upgrade
```

And an end-to-end workflow can be accomplished with the following:

```
import tensorflow_decision_forests as tfdf
import pandas as pd

# Load the dataset in a Pandas dataframe.
train_df = pd.read_csv("project/train.csv")
test_df = pd.read_csv("project/test.csv")

# Convert the dataset into a TensorFlow dataset.
train_ds = tfdf.keras.pd_dataframe_to_tf_dataset(train_df, label="my_label")
test_ds = tfdf.keras.pd_dataframe_to_tf_dataset(test_df, label="my_label")

# Train the model
model = tfdf.keras.RandomForestModel()
model.fit(train_ds)

# Look at the model.
model.summary()

# Evaluate the model.
model.evaluate(test_ds)

# Export to a TensorFlow SavedModel.
# Note: the model is compatible with Yggdrasil Decision Forests.
model.save("project/model")
```

For more, you can find the full codebase for the TensorFlow Decisions Forests library on GitHub.

28. TensorFlow Probability

TensorFlow Probability (TFP) is a library built on top of TensorFlow to allow for combining probabilistic models and deep learning on accelerated hardware such as GPUs and TPUs.

It allows you to create probability distributions such as Bernoulli with `tfp.distributions.Bernoulli` and Normal with `tfp.distributions.Normal`.

It provides tools for performing variational inference, Markov chain and Monte Carlo sampling and more.

You can get started with TensorFlow Probability by installing the `tensorflow-probability` package:

```
pip install tensorflow-probability
```

And to start creating a probability distribution, you can use `tfp.distributions`:

```
import tensorflow_probability as tfp

# Create a Normal distribution
normal_distribution = tfp.distributions.Normal(loc=0., scale=1.).sample(int(100e3))
```

29. TensorFlow Graphics

TensorFlow Graphics is a library for cominbing traditional computer graphics techniques for dealing with 3D objects as well as newer computer vision techniques to enable the training of more advanced computer vision and graphics models.

Combining computer vision and computer graphics techniques provides enables machine learning practitioners to leverage already existing graphical data for machine learning models.

To get started with TensorFlow Graphics, you can install the package `tensorflow-graphics`:

```
pip install tensorflow-graphics
```

And as an example, to compute the triangle areas of a given tensor, you can use:

```
import tensorflow_graphics as tfg

# Compute triangle areas of a given tensor
tfg.geometry.representation.mesh.sampler.triangle_area()
```

30. TensorFlow Cloud

TensorFlow Cloud is a library for training TensorFlow models on Google Cloud AI Platform, enabling seamless scaling of your training tasks.

For example, you could code a model locally on your smaller hardware (e.g. 1 GPU) and verify it works before running the same model with TensorFlow Cloud and scaling it up on Google Cloud with larger amounts of hardware (e.g. 8 GPUs).

To get started with TensorFlow Cloud, you can install the package `tensorflow-cloud`:

```
pip install tensorflow-cloud
```

You can see a full workflow of creating and running a TensorFlow model with TensorFlow Cloud in the TensorFlow documentation.

Otherwise, the worfklow is as follows:

```
import tensorflow_cloud as tfc

# Create model
# ...

# Create training code
# ...

# Run the above code on the cloud (with predefined resources)
tfc.run(distribution_strategy='auto',
        docker_config=..., # define parameters you'd like your code to run with
        ...)
```

31. TensorFlow.js

TensorFlow.js is a library for machine learning in JavaScript, enabling you to train and deploy models directly in web browsers or Node.js environments.

The benefit of deploying a machine learning model right into the browser is that it enables a whole bunch of benefits:

- Data can be computed on in the browser (without being sent to a server), which is good for privacy and latency.

- Predictions in the browser can leverage the persons device hardware and save on server costs.
- You can create a machine learning application with one code stack (e.g. all in JavaScript).

To get started with TensorFlow.js, you can install it via `npm`:

```
npm install @tensorflow/tfjs
```

Otherwise, these are several more options for installing such as via `yarn` or directly in HTML:

```
<html>
  <head>
    <script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"> </script>
  </head>
  <body>
    <h4>Tiny TFJS example<hr/></h4>
    <div id="micro-out-div">Training...</div>
    <script src="./index.js"> </script>
  </body>
</html>
```

Once TensorFlow.js is ready to go, the usage is very similar to the Python API:

```
// Create a simple model.
const model = tf.sequential();
model.add(tf.layers.dense({units: 1, inputShape: [1]}));

// Prepare the model for training: Specify the loss and the optimizer.
model.compile({loss: 'meanSquaredError', optimizer: 'sgd'});

// Generate some synthetic data for training. (y = 2x - 1)
const xs = tf.tensor2d([-1, 0, 1, 2, 3, 4], [6, 1]);
const ys = tf.tensor2d([-3, -1, 1, 3, 5, 7], [6, 1]);

// Train the model using the data.
await model.fit(xs, ys, {epochs: 250});
```

You can see more on TensorFlow.js such as using pretrained models and creating size-optimized models (for running directly in browsers) in the TensorFlow.js documentation.

Summary

As you've seen, the TensorFlow library is vast and covers almost every area of machine learning you can imagine.

By following this cheatsheet, you've now got the resources and ideas to start performing a variety of machine learning tasks with TensorFlow.

You can keep this guide as a reference when working on your own projects.

And don't forget, there's plenty more to explore in the TensorFlow documentation, follow your curiosity and see where it leads you.

Happy machine learning!