[Machine Learning Crash Course](https://developers.google.com/machine-learning/crash-course/?authuser=1)

What is (supervised) machine learning? Concisely put, it is the following:

* ML systems learn how to combine input to produce useful predictions on never-before-seen data.

Labels

A **label** is the thing we're predicting—the y variable in simple linear regression. The label could be the future price of wheat, the kind of animal shown in a picture, the meaning of an audio clip, or just about anything.

Features

A **feature** is an input variable—the x variable in simple linear regression. A simple machine learning project might use a single feature, while a more sophisticated machine learning project could use millions of features, specified as:

{

x

1

,

x

2

,

.

.

.

x

N

}

In the spam detector example, the features could include the following:

* words in the email text
* sender's address
* time of day the email was sent
* email contains the phrase "one weird trick."

Examples

An **example** is a particular instance of data, **x**. (We put **x** in boldface to indicate that it is a vector.) We break examples into two categories:

* labeled examples
* unlabeled examples

A **labeled example** includes both feature(s) and the label. That is:

labeled examples: {features, label}: (x, y)

Use labeled examples to **train** the model. In our spam detector example, the labeled examples would be individual emails that users have explicitly marked as "spam" or "not spam."

For example, the following table shows 5 labeled examples from a [data set](https://developers.google.com/machine-learning/crash-course/california-housing-data-description?authuser=1) containing information about housing prices in California:

|  |  |  |  |
| --- | --- | --- | --- |
| housingMedianAge  (feature) | totalRooms  (feature) | totalBedrooms  (feature) | medianHouseValue  (label) |
| 15 | 5612 | 1283 | 66900 |
| 19 | 7650 | 1901 | 80100 |
| 17 | 720 | 174 | 85700 |
| 14 | 1501 | 337 | 73400 |
| 20 | 1454 | 326 | 65500 |

An **unlabeled example** contains features but not the label. That is:

unlabeled examples: {features, ?}: (x, ?)

Here are 3 unlabeled examples from the same housing dataset, which exclude medianHouseValue:

|  |  |  |
| --- | --- | --- |
| housingMedianAge  (feature) | totalRooms  (feature) | totalBedrooms  (feature) |
| 42 | 1686 | 361 |
| 34 | 1226 | 180 |
| 33 | 1077 | 271 |

Once we've trained our model with labeled examples, we use that model to predict the label on unlabeled examples. In the spam detector, unlabeled examples are new emails that humans haven't yet labeled.

Models

A model defines the relationship between features and label. For example, a spam detection model might associate certain features strongly with "spam". Let's highlight two phases of a model's life:

* **Training** means creating or **learning** the model. That is, you show the model labeled examples and enable the model to gradually learn the relationships between features and label.
* **Inference** means applying the trained model to unlabeled examples. That is, you use the trained model to make useful predictions (y'). For example, during inference, you can predict medianHouseValue for new unlabeled examples.

Regression vs. classification

A **regression** model predicts continuous values. For example, regression models make predictions that answer questions like the following:

* What is the value of a house in California?
* What is the probability that a user will click on this ad?

A **classification** model predicts discrete values. For example, classification models make predictions that answer questions like the following:

* Is a given email message spam or not spam?
* Is this an image of a dog, a cat, or a hamster?

Reducing Loss: Gradient Descent

**Estimated Time:** 10 minutes

The iterative approach diagram ([Figure 1](https://developers.google.com/machine-learning/crash-course/reducing-loss/an-iterative-approach#ml-block-diagram)) contained a green hand-wavy box entitled "Compute parameter updates." We'll now replace that algorithmic fairy dust with something more substantial.

Suppose we had the time and the computing resources to calculate the loss for all possible values of w1. For the kind of regression problems we've been examining, the resulting plot of loss vs. w1 will always be convex. In other words, the plot will always be bowl-shaped, kind of like this:

lossvalue of weight wi

**Figure 2. Regression problems yield convex loss vs weight plots.**

Convex problems have only one minimum; that is, only one place where the slope is exactly 0. That minimum is where the loss function converges.

Calculating the loss function for every conceivable value of w1 over the entire data set would be an inefficient way of finding the convergence point. Let's examine a better mechanism—very popular in machine learning—called **gradient descent**.

The first stage in gradient descent is to pick a starting value (a starting point) for w1. The starting point doesn't matter much; therefore, many algorithms simply set w1 to 0 or pick a random value. The following figure shows that we've picked a starting point slightly greater than 0:

starting pointlossvalue of weight wi

**Figure 3. A starting point for gradient descent.**

The gradient descent algorithm then calculates the gradient of the loss curve at the starting point. Here in Figure 3, the gradient of loss is equal to the [derivative](https://wikipedia.org/wiki/Differential_calculus#The_derivative) (slope) of the curve, and tells you which way is "warmer" or "colder." When there are multiple weights, the **gradient** is a vector of partial derivatives with respect to the weights.

▾

Click the dropdown arrow to learn more about partial derivatives and gradients.

The math around machine learning is fascinating and we're delighted that you clicked the link to learn more. Please note, however, that TensorFlow handles all the gradient computations for you, so you don't actually have to understand the calculus provided here.

Partial derivatives

A **multivariable function** is a function with more than one argument, such as:

f(x,y)=e2ysin⁡(x)

The **partial derivative**f**with respect to**x, denoted as follows:

∂f∂x

is the derivative of f considered as a function of x alone. To find the following:

∂f∂x

you must hold y constant (so f is now a function of one variable x), and take the regular derivative of f with respect to x. For example, when y is fixed at 1, the preceding function becomes:

f(x)=e2sin⁡(x)

This is just a function of one variable x, whose derivative is:

e2cos⁡(x)

In general, thinking of y as fixed, the partial derivative of f with respect to x is calculated as follows:

∂f∂x(x,y)=e2ycos⁡(x)

Similarly, if we hold x fixed instead, the partial derivative of f with respect to y is:

∂f∂y(x,y)=2e2ysin⁡(x)

Intuitively, a partial derivative tells you how much the function changes when you perturb one variable a bit. In the preceding example:

∂f∂x(0,1)=e2≈7.4

So when you start at (0,1), hold y constant, and move x a little, f changes by about 7.4 times the amount that you changed x.

In machine learning, partial derivatives are mostly used in conjunction with the gradient of a function.

Gradients

The **gradient** of a function, denoted as follows, is the vector of partial derivatives with respect to all of the independent variables:

∇f

For instance, if:

f(x,y)=e2ysin⁡(x)

then:

∇f(x,y)=(∂f∂x(x,y),∂f∂y(x,y))=(e2ycos⁡(x),2e2ysin⁡(x))

Note the following:

|  |  |
| --- | --- |
| ∇f | Points in the direction of greatest increase of the function. |
| −∇f | Points in the direction of greatest decrease of the function. |

The number of dimensions in the vector is equal to the number of variables in the formula for f; in other words, the vector falls within the domain space of the function. For instance, the graph of the following function f(x,y):

f(x,y)=4+(x−2)2+2y2

when viewed in three dimensions with z=f(x,y) looks like a valley with a minimum at (2,0,4):

The gradient of f(x,y) is a two-dimensional vector that tells you in which (x,y) direction to move for the maximum increase in height. Thus, the negative of the gradient moves you in the direction of maximum decrease in height. In other words, the negative of the gradient vector points into the valley.

In machine learning, gradients are used in gradient descent. We often have a loss function of many variables that we are trying to minimize, and we try to do this by following the negative of the gradient of the function.

Note that a gradient is a vector, so it has both of the following characteristics:

* a direction
* a magnitude

The gradient always points in the direction of steepest increase in the loss function. The gradient descent algorithm takes a step in the direction of the negative gradient in order to reduce loss as quickly as possible.

starting pointloss(negative)gradientvalue of weight wi

**Figure 4. Gradient descent relies on negative gradients.**

To determine the next point along the loss function curve, the gradient descent algorithm adds some fraction of the gradient's magnitude to the starting point as shown in the following figure:

starting pointloss(negative)gradientvalue of weight winext point

**Figure 5. A gradient step moves us to the next point on the loss curve.**

The gradient descent then repeats this process, edging ever closer to the minimum.

Reducing Loss: Learning Rate

**Estimated Time:** 5 minutes

As noted, the gradient vector has both a direction and a magnitude. Gradient descent algorithms multiply the gradient by a scalar known as the **learning rate** (also sometimes called **step size**) to determine the next point. For example, if the gradient magnitude is 2.5 and the learning rate is 0.01, then the gradient descent algorithm will pick the next point 0.025 away from the previous point.

**Hyperparameters** are the knobs that programmers tweak in machine learning algorithms. Most machine learning programmers spend a fair amount of time tuning the learning rate. If you pick a learning rate that is too small, learning will take too long:

value of weight wilossSmall learning ratetakes forever!starting point

**Figure 6. Learning rate is too small.**

Conversely, if you specify a learning rate that is too large, the next point will perpetually bounce haphazardly across the bottom of the well like a quantum mechanics experiment gone horribly wrong:

value of weight wilossOvershoots theminimum!starting point

**Figure 7. Learning rate is too large.**

There's a [Goldilocks](https://wikipedia.org/wiki/Goldilocks_principle) learning rate for every regression problem. The Goldilocks value is related to how flat the loss function is. If you know the gradient of the loss function is small then you can safely try a larger learning rate, which compensates for the small gradient and results in a larger step size.

starting pointvalue of weight wilossWe’ll get thereefficiently.

**Figure 8. Learning rate is just right.**

▾

Click the dropdown arrow to learn more about the ideal learning rate.

The ideal learning rate in one-dimension is 1f(x)″ (the inverse of the second derivative of f(x) at x).

The ideal learning rate for 2 or more dimensions is the inverse of the [Hessian](https://wikipedia.org/wiki/Hessian_matrix) (matrix of second partial derivatives).

The story for general convex functions is more complex.

Reducing Loss: Stochastic Gradient Descent

**Estimated Time:** 3 minutes

In gradient descent, a **batch** is the total number of examples you use to calculate the gradient in a single iteration. So far, we've assumed that the batch has been the entire data set. When working at Google scale, data sets often contain billions or even hundreds of billions of examples. Furthermore, Google data sets often contain huge numbers of features. Consequently, a batch can be enormous. A very large batch may cause even a single iteration to take a very long time to compute.

A large data set with randomly sampled examples probably contains redundant data. In fact, redundancy becomes more likely as the batch size grows. Some redundancy can be useful to smooth out noisy gradients, but enormous batches tend not to carry much more predictive value than large batches.

What if we could get the right gradient *on average* for much less computation? By choosing examples at random from our data set, we could estimate (albeit, noisily) a big average from a much smaller one.**Stochastic gradient descent** (**SGD**) takes this idea to the extreme--it uses only a single example (a batch size of 1) per iteration. Given enough iterations, SGD works but is very noisy. The term "stochastic" indicates that the one example comprising each batch is chosen at random.

**Mini-batch stochastic gradient descent** (**mini-batch SGD**) is a compromise between full-batch iteration and SGD. A mini-batch is typically between 10 and 1,000 examples, chosen at random. Mini-batch SGD reduces the amount of noise in SGD but is still more efficient than full-batch.

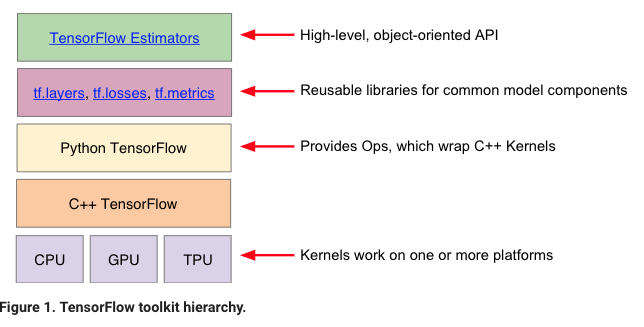
To simplify the explanation, we focused on gradient descent for a single feature. Rest assured that gradient descent also works on feature sets that contain multiple features.

# First Steps with TensorFlow: Toolkit

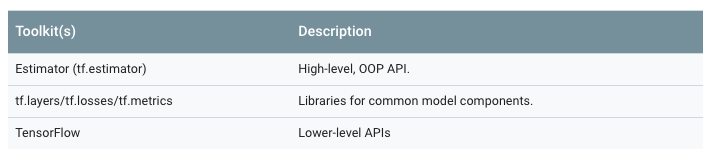
**Estimated Time:** 4 minutes

Tensorflow is a computational framework for building machine learning models. TensorFlow provides a variety of different toolkits that allow you to construct models at your preferred level of abstraction. You can use lower-level APIs to build models by defining a series of mathematical operations. Alternatively, you can use higher-level APIs (like tf.estimator) to specify predefined architectures, such as linear regressors or neural networks.

The following figure shows the current hierarchy of TensorFlow toolkits:

****

The following table summarizes the purposes of the different layers:



TensorFlow consists of the following two components:

* a [graph protocol buffer](https://www.tensorflow.org/extend/tool_developers/#protocol_buffers)
* a runtime that executes the (distributed) graph

These two components are analogous to Python code and the Python interpreter. Just as the Python interpreter is implemented on multiple hardware platforms to run Python code, TensorFlow can run the graph on multiple hardware platforms, including CPU, GPU, and [TPU](https://wikipedia.org/wiki/Tensor_processing_unit).

Which API(s) should you use? You should use the highest level of abstraction that solves the problem. The higher levels of abstraction are easier to use, but are also (by design) less flexible. We recommend you start with the highest-level API first and get everything working. If you need additional flexibility for some special modeling concerns, move one level lower. Note that each level is built using the APIs in lower levels, so dropping down the hierarchy should be reasonably straightforward.

## **tf.estimator API**

We'll use tf.estimator for the majority of exercises in Machine Learning Crash Course. Everything you'll do in the exercises could have been done in lower-level (raw) TensorFlow, but using tf.estimator dramatically lowers the number of lines of code.

tf.estimator is compatible with the scikit-learn API. [Scikit-learn](http://scikit-learn.org/) is an extremely popular open-source ML library in Python, with over 100k users, including many at Google.

Very broadly speaking, here's the pseudocode for a linear classification program implemented in tf.estimator:

import tensorflow as tf  
  
# Set up a linear classifier.  
classifier = tf.estimator.LinearClassifier(feature\_columns)  
  
# Train the model on some example data.  
classifier.train(input\_fn=train\_input\_fn, steps=2000)  
  
# Use it to predict.  
predictions = classifier.predict(input\_fn=predict\_input\_fn)

# First Steps with TensorFlow: Programming Exercises

**Estimated Time:** 55 minutes

As you progress through Machine Learning Crash Course, you'll put the principles and techniques you learn into practice by coding models using tf.estimator, a high-level [TensorFlow](https://www.tensorflow.org/?authuser=1) API.

The programming exercises in Machine Learning Crash Course use a data-analysis platform that combines code, output, and descriptive text into one collaborative document.

Programming exercises run directly in your browser (no setup required!) using the [Colaboratory](https://colab.research.google.com/?authuser=1) platform. Colaboratory is supported on most major browsers, and is most thoroughly tested on desktop versions of Chrome and Firefox. If you'd prefer to download and run the exercises offline, see [these instructions](https://github.com/google/eng-edu/blob/master/ml/cc/README.md#with-docker) for setting up a local environment.

Run the following three exercises in the provided order:

1. [Quick Introduction to pandas](https://colab.research.google.com/notebooks/mlcc/intro_to_pandas.ipynb?utm_source=mlcc&utm_campaign=colab-external&utm_medium=referral&utm_content=pandas-colab&hl=en&authuser=1). pandas is an important library for data analysis and modeling, and is widely used in TensorFlow coding. This tutorial provides all the pandas information you need for this course. If you already know pandas, you can skip this exercise.
2. [First Steps with TensorFlow](https://colab.research.google.com/notebooks/mlcc/first_steps_with_tensor_flow.ipynb?utm_source=mlcc&utm_campaign=colab-external&utm_medium=referral&utm_content=firststeps-colab&hl=en&authuser=1). This exercise explores linear regression.
3. [Synthetic Features and Outliers](https://colab.research.google.com/notebooks/mlcc/synthetic_features_and_outliers.ipynb?utm_source=mlcc&utm_campaign=colab-external&utm_medium=referral&utm_content=syntheticfeatures-colab&hl=en&authuser=1). This exercise explores synthetic features and the effect of input outliers.

## **Common hyperparameters in Machine Learning Crash Course exercises**

Many of the coding exercises contain the following hyperparameters:

* **steps**, which is the total number of training iterations. One step calculates the loss from one batch and uses that value to modify the model's weights once.
* **batch size**, which is the number of examples (chosen at random) for a single step. For example, the batch size for SGD is 1.

The following formula applies:

## **../../../Desktop/Screen%20Shot%202018-06-02%20at%209.36.56%20PMA convenience variable in Machine Learning Crash Course exercises**

The following convenience variable appears in several exercises:

* **periods**, which controls the granularity of reporting. For example, if periods is set to 7 and steps is set to 70, then the exercise will output the loss value every 10 steps (or 7 times). Unlike hyperparameters, we don't expect you to modify the value of periods. Note that modifying periodsdoes not alter what your model learns.

The following formula applies:

