Introducción a TensorFlow

Esta guía le permite comenzar a programar en TensorFlow. Antes de utilizar esta guía, [instale TensorFlow](https://www.tensorflow.org/install/index) . Para aprovechar al máximo esta guía, debe saber lo siguiente:

* Cómo programar en Python.
* Al menos un poco sobre arreglos.
* Idealmente, algo sobre el aprendizaje automático. Sin embargo, si sabe poco o nada acerca del aprendizaje automático, esta sigue siendo la primera guía que debe leer.

TensorFlow proporciona varias API. La API de nivel más bajo, TensorFlow Core, le proporciona un control de programación completo. Recomendamos TensorFlow Core para los investigadores que aprenden a máquina y otros que requieren niveles finos de control sobre sus modelos. Las API de nivel superior se construyen sobre TensorFlow Core. Estas API de nivel superior suelen ser más fáciles de aprender y usar que TensorFlow Core. Además, las API de nivel superior hacen que las tareas repetitivas sean más fáciles y coherentes entre los diferentes usuarios. Una API de alto nivel como tf.estimator le ayuda a administrar conjuntos de datos, estimadores, entrenamiento e inferencia.

Esta guía comienza con un tutorial sobre TensorFlow Core. Posteriormente, demostraremos cómo implementar el mismo modelo en tf.estimator. Conocer los principios de TensorFlow Core te dará un gran modelo mental de cómo las cosas funcionan internamente cuando usas el API de nivel superior más compacto.

Tensores

La unidad central de datos en TensorFlow es el **tensor** . Un tensor consiste en un conjunto de valores primitivos conformados en una matriz de cualquier número de dimensiones. El **rango de** un tensor es su número de dimensiones. Aquí hay algunos ejemplos de tensores:

3 # a rank 0 tensor; this is a scalar with shape []  
[1., 2., 3.] # a rank 1 tensor; this is a vector with shape [3]  
[[1., 2., 3.], [4., 5., 6.]] # a rank 2 tensor; a matrix with shape [2, 3]  
[[[1., 2., 3.]], [[7., 8., 9.]]] # a rank 3 tensor with shape [2, 1, 3]

TutorFlow Core tutorial

Importación de TensorFlow

La instrucción de importación canónica para los programas TensorFlow es la siguiente:

import tensorflow as tf

Esto le da a Python acceso a todas las clases, métodos y símbolos de TensorFlow. La mayor parte de la documentación asume que usted ya ha hecho esto.

El Gráfico Computacional

Podría pensar que los programas TensorFlow Core consisten en dos secciones discretas:

1. Construir el gráfico computacional.
2. Ejecución del gráfico computacional.

Un **gráfico computacional** es una serie de operaciones TensorFlow dispuestas en un gráfico de nodos. Vamos a construir un gráfico computacional simple. Cada nodo toma cero o más tensores como entradas y produce un tensor como una salida. Un tipo de nodo es una constante. Como todas las constantes de TensorFlow, no toma entradas, y produce un valor que almacena internamente. Podemos crear dos Tensores de punto flotante node1y node2como sigue:

node1 = tf.constant(3.0, dtype=tf.float32)  
node2 = tf.constant(4.0) # also tf.float32 implicitly  
print(node1, node2)

La declaración final de impresión produce

Tensor("Const:0", shape=(), dtype=float32) Tensor("Const\_1:0", shape=(), dtype=float32)

Observe que la impresión de los nodos no genera los valores 3.0y 4.0como es de esperar. En su lugar, son nodos que, cuando se evalúan, producirían 3.0 y 4.0, respectivamente. Para evaluar realmente los nodos, debemos ejecutar el gráfico computacional dentro de una **sesión** . Una sesión encapsula el control y el estado del tiempo de ejecución de TensorFlow.

El código siguiente crea un Sessionobjeto y luego invoca su runmétodo para ejecutar suficiente del gráfico computacional para evaluar node1y node2. Ejecutando el gráfico computacional en una sesión de la siguiente manera:

sess = tf.Session()  
print(sess.run([node1, node2]))

vemos los valores esperados de 3.0 y 4.0:

[3.0, 4.0]

Podemos construir cálculos más complicados combinando Tensornodos con operaciones (las operaciones también son nodos). Por ejemplo, podemos agregar nuestros dos nodos constantes y producir un nuevo gráfico de la siguiente manera:

node3 = tf.add(node1, node2)  
print("node3:", node3)  
print("sess.run(node3):", sess.run(node3))

Las dos últimas instrucciones de impresión producen

node3: Tensor("Add:0", shape=(), dtype=float32)  
sess.run(node3): 7.0

TensorFlow proporciona una utilidad llamada TensorBoard que puede mostrar una imagen del gráfico computacional. Aquí hay una captura de pantalla que muestra cómo TensorBoard visualiza el gráfico:



Tal como está, esta gráfica no es especialmente interesante porque siempre produce un resultado constante. Un gráfico puede parametrizarse para aceptar entradas externas, conocidas como **marcadores de posición** . Un **marcador de posición** es una promesa de proporcionar un valor más adelante.

a = tf.placeholder(tf.float32)  
b = tf.placeholder(tf.float32)  
adder\_node = a + b  # + provides a shortcut for tf.add(a, b)

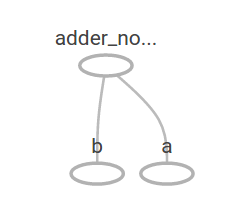
Las tres líneas anteriores son un poco como una función o una lambda en la que definimos dos parámetros de entrada (ayb) y luego una operación sobre ellos. Podemos evaluar este gráfico con múltiples entradas utilizando el argumento feed\_dict al [método run](https://www.tensorflow.org/api_docs/python/tf/Session#run) para alimentar valores concretos a los marcadores de posición:

print(sess.run(adder\_node, {a: 3, b: 4.5}))  
print(sess.run(adder\_node, {a: [1, 3], b: [2, 4]}))

resultando en la salida

7.5  
[ 3.  7.]

En TensorBoard, la gráfica se ve así:



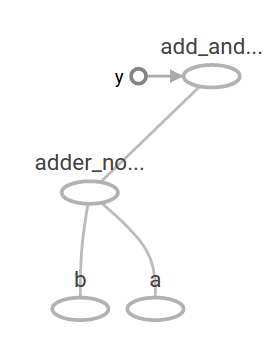
Podemos hacer que el gráfico computacional sea más complejo añadiendo otra operación. Por ejemplo,

add\_and\_triple = adder\_node \* 3.  
print(sess.run(add\_and\_triple, {a: 3, b: 4.5}))

produce la salida

22.5

El gráfico computacional anterior se vería de la siguiente manera en TensorBoard:



En el aprendizaje de máquina, normalmente queremos un modelo que puede tomar entradas arbitrarias, como la anterior. Para que el modelo pueda ser entrenado, debemos ser capaces de modificar el gráfico para obtener nuevas salidas con la misma entrada. **Las variables** nos permiten agregar parámetros entrenables a un gráfico. Se construyen con un tipo y un valor inicial:

W = tf.Variable([.3], dtype=tf.float32)  
b = tf.Variable([-.3], dtype=tf.float32)  
x = tf.placeholder(tf.float32)  
linear\_model = W \* x + b

Las constantes se inicializan cuando llamas tf.constant, y su valor nunca puede cambiar. Por el contrario, las variables no se inicializan cuando se llama tf.Variable. Para inicializar todas las variables en un programa TensorFlow, debe llamar explícitamente a una operación especial de la siguiente manera:

init = tf.global\_variables\_initializer()  
sess.run(init)

Es importante darse cuenta de que inites un identificador para el subgráfico TensorFlow que inicializa todas las variables globales. Hasta que llamamos sess.run, las variables no se inicializan.

Dado que xes un marcador de posición, podemos evaluar linear\_modelpara varios valores de formaxsimultánea de la siguiente manera:

print(sess.run(linear\_model, {x: [1, 2, 3, 4]}))

para producir la salida

[ 0.          0.30000001  0.60000002  0.90000004]

Hemos creado un modelo, pero no sabemos lo bueno que es todavía. Para evaluar el modelo en los datos de entrenamiento, necesitamos un ymarcador de posición para proporcionar los valores deseados, y necesitamos escribir una función de pérdida.

Una función de pérdida mide cuán distante está el modelo actual de los datos proporcionados. Utilizaremos un modelo de pérdida estándar para la regresión lineal, que suma los cuadrados de los deltas entre el modelo actual y los datos proporcionados. linear\_model - ycrea un vector donde cada elemento es el error del ejemplo correspondiente delta. Llamamos tf.squarea cuadrar ese error. Entonces, sumamos todos los errores cuadrados para crear un solo escalar que abstrae el error de todos los ejemplos usando tf.reduce\_sum:

y = tf.placeholder(tf.float32)  
squared\_deltas = tf.square(linear\_model - y)  
loss = tf.reduce\_sum(squared\_deltas)  
print(sess.run(loss, {x: [1, 2, 3, 4], y: [0, -1, -2, -3]}))

produciendo el valor de pérdida

23.66

Podríamos mejorar esto manualmente reasignando los valores de Wy bpara los valores perfectos de -1 y 1. Una variable se inicializa al valor proporcionado tf.Variablepero se puede cambiar usando operaciones como tf.assign. Por ejemplo, W=-1y b=1son los parámetros óptimos para nuestro modelo. Podemos cambiar Wy por bconsiguiente:

fixW = tf.assign(W, [-1.])  
fixb = tf.assign(b, [1.])  
sess.run([fixW, fixb])  
print(sess.run(loss, {x: [1, 2, 3, 4], y: [0, -1, -2, -3]}))

La impresión final muestra que la pérdida es ahora cero.

0.0

Adivinamos los valores "perfectos" de Wy b, pero el punto de aprendizaje de la máquina es encontrar automáticamente los parámetros correctos del modelo. Mostraremos cómo lograr esto en la siguiente sección.

API de tf.train

Una discusión completa sobre el aprendizaje automático está fuera del alcance de este tutorial. Sin embargo, TensorFlow proporciona **optimizadores** que cambian lentamente cada variable con el fin de minimizar la función de pérdida. El optimizador más simple es el **descenso gradiente** . Modifica cada variable de acuerdo con la magnitud de la derivada de pérdida con respecto a esa variable. En general, calcular manualmente las derivas simbólicas es tedioso y propenso a errores. Por lo tanto, TensorFlow puede producir automáticamente derivados dada sólo una descripción del modelo utilizando la función tf.gradients. Por simplicidad, los optimizadores normalmente lo hacen por usted. Por ejemplo,

optimizer = tf.train.GradientDescentOptimizer(0.01)  
train = optimizer.minimize(loss)

sess.run(init) # reset values to incorrect defaults.  
for i in range(1000):  
  sess.run(train, {x: [1, 2, 3, 4], y: [0, -1, -2, -3]})  
  
print(sess.run([W, b]))

resultados en los parámetros finales del modelo:

[array([-0.9999969], dtype=float32), array([ 0.99999082],  
 dtype=float32)]

Ahora hemos hecho aprendizaje de máquina real! Aunque hacer esta simple regresión lineal no requiere mucho código de núcleo TensorFlow, modelos y métodos más complicados para alimentar datos en su modelo requieren más código. Por lo tanto, TensorFlow proporciona abstracciones de nivel superior para patrones, estructuras y funcionalidades comunes. Aprenderemos a usar algunas de estas abstracciones en la siguiente sección.

Programa completo

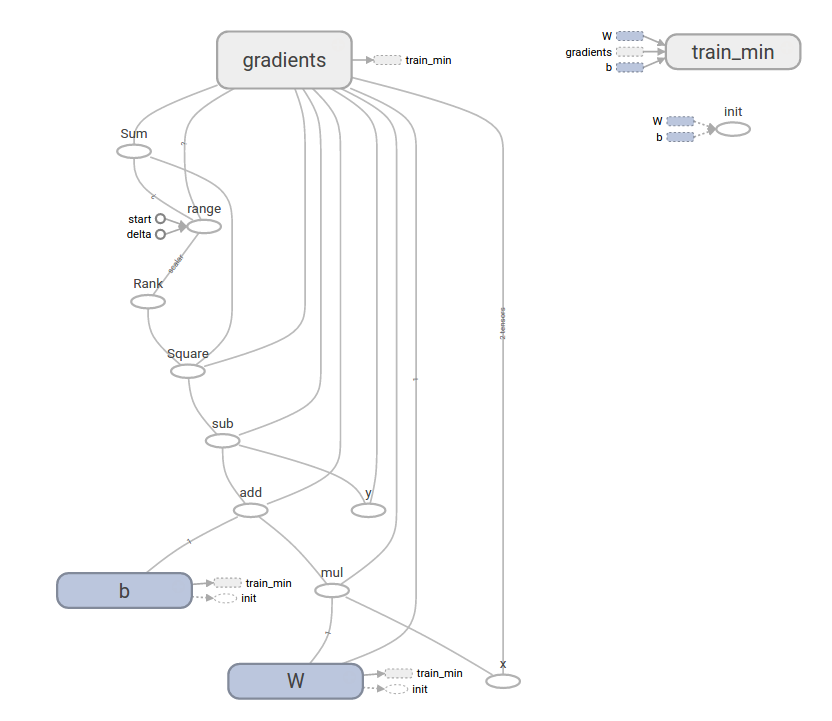
El modelo de regresión lineal manejable completado se muestra aquí:

import tensorflow as tf  
  
# Model parameters  
W = tf.Variable([.3], dtype=tf.float32)  
b = tf.Variable([-.3], dtype=tf.float32)  
# Model input and output  
x = tf.placeholder(tf.float32)  
linear\_model = W \* x + b  
y = tf.placeholder(tf.float32)  
  
# loss  
loss = tf.reduce\_sum(tf.square(linear\_model - y)) # sum of the squares  
# optimizer  
optimizer = tf.train.GradientDescentOptimizer(0.01)  
train = optimizer.minimize(loss)  
  
# training data  
x\_train = [1, 2, 3, 4]  
y\_train = [0, -1, -2, -3]  
# training loop  
init = tf.global\_variables\_initializer()  
sess = tf.Session()  
sess.run(init) # reset values to wrong  
for i in range(1000):  
  sess.run(train, {x: x\_train, y: y\_train})  
  
# evaluate training accuracy  
curr\_W, curr\_b, curr\_loss = sess.run([W, b, loss], {x: x\_train, y: y\_train})  
print("W: %s b: %s loss: %s"%(curr\_W, curr\_b, curr\_loss))

Cuando se ejecuta, produce

W: [-0.9999969] b: [ 0.99999082] loss: 5.69997e-11

Observe que la pérdida es un número muy pequeño (muy cercano a cero). Si ejecuta este programa, su pérdida puede no ser exactamente la misma porque el modelo se inicializa con valores pseudoaleatorios.

Este programa más complicado todavía se puede visualizar en TensorBoard 

tf.estimator

tf.estimator es una biblioteca TensorFlow de alto nivel que simplifica la mecánica del aprendizaje automático, incluyendo lo siguiente:

* ejecución de bucles de entrenamiento
* ejecución de bucles de evaluación
* gestión de conjuntos de datos

tf.estimator define muchos modelos comunes.

Uso básico

Observe cuánto más sencillo es el programa de regresión lineal con tf.estimator:

import tensorflow as tf  
# NumPy is often used to load, manipulate and preprocess data.  
import numpy as np  
  
# Declare list of features. We only have one numeric feature. There are many  
# other types of columns that are more complicated and useful.  
feature\_columns = [tf.feature\_column.numeric\_column("x", shape=[1])]  
  
# An estimator is the front end to invoke training (fitting) and evaluation  
# (inference). There are many predefined types like linear regression,  
# linear classification, and many neural network classifiers and regressors.  
# The following code provides an estimator that does linear regression.  
estimator = tf.estimator.LinearRegressor(feature\_columns=feature\_columns)  
  
# TensorFlow provides many helper methods to read and set up data sets.  
# Here we use two data sets: one for training and one for evaluation  
# We have to tell the function how many batches  
# of data (num\_epochs) we want and how big each batch should be.  
x\_train = np.array([1., 2., 3., 4.])  
y\_train = np.array([0., -1., -2., -3.])  
x\_eval = np.array([2., 5., 8., 1.])  
y\_eval = np.array([-1.01, -4.1, -7, 0.])  
input\_fn = tf.estimator.inputs.numpy\_input\_fn(  
    {"x": x\_train}, y\_train, batch\_size=4, num\_epochs=None, shuffle=True)  
train\_input\_fn = tf.estimator.inputs.numpy\_input\_fn(  
    {"x": x\_train}, y\_train, batch\_size=4, num\_epochs=1000, shuffle=False)  
eval\_input\_fn = tf.estimator.inputs.numpy\_input\_fn(  
    {"x": x\_eval}, y\_eval, batch\_size=4, num\_epochs=1000, shuffle=False)  
  
# We can invoke 1000 training steps by invoking the  method and passing the  
# training data set.  
estimator.train(input\_fn=input\_fn, steps=1000)  
  
# Here we evaluate how well our model did.  
train\_metrics = estimator.evaluate(input\_fn=train\_input\_fn)  
eval\_metrics = estimator.evaluate(input\_fn=eval\_input\_fn)  
print("train metrics: %r"% train\_metrics)  
print("eval metrics: %r"% eval\_metrics)

Cuando se ejecuta, produce

train metrics: {'loss': 1.2712867e-09, 'global\_step': 1000}  
eval metrics: {'loss': 0.0025279333, 'global\_step': 1000}

Observe cómo nuestros datos del eval tienen una pérdida más alta, pero sigue siendo cercano a cero. Eso significa que estamos aprendiendo adecuadamente.

Un modelo personalizado

tf.estimatorno te bloquea en sus modelos predefinidos. Supongamos que queremos crear un modelo personalizado que no esté incorporado en TensorFlow. Todavía podemos mantener el alto nivel de abstracción del conjunto de datos, alimentación, formación, etc., de tf.estimator. Por ejemplo, mostraremos cómo implementar nuestro propio modelo equivalente a LinearRegressorusar nuestro conocimiento de la API TensorFlow de nivel inferior.

Para definir un modelo personalizado que funcione tf.estimator, necesitamos usarlotf.estimator.Estimator. tf.estimator.LinearRegressores en realidad una subclase de tf.estimator.Estimator. En lugar de subclasificar Estimator, simplemente proporcionamos Estimatoruna función model\_fnque cuenta tf.estimatorcómo puede evaluar las predicciones, los pasos de entrenamiento y la pérdida. El código es el siguiente:

import numpy as np  
import tensorflow as tf  
  
# Declare list of features, we only have one real-valued feature  
def model\_fn(features, labels, mode):  
  # Build a linear model and predict values  
  W = tf.get\_variable("W", [1], dtype=tf.float64)  
  b = tf.get\_variable("b", [1], dtype=tf.float64)  
  y = W \* features['x'] + b  
  # Loss sub-graph  
  loss = tf.reduce\_sum(tf.square(y - labels))  
  # Training sub-graph  
  global\_step = tf.train.get\_global\_step()  
  optimizer = tf.train.GradientDescentOptimizer(0.01)  
  train = tf.group(optimizer.minimize(loss),  
                   tf.assign\_add(global\_step, 1))  
  # EstimatorSpec connects subgraphs we built to the  
  # appropriate functionality.  
  return tf.estimator.EstimatorSpec(  
      mode=mode,  
      predictions=y,  
      loss=loss,  
      train\_op=train)  
  
estimator = tf.estimator.Estimator(model\_fn=model\_fn)  
# define our data sets  
x\_train = np.array([1., 2., 3., 4.])  
y\_train = np.array([0., -1., -2., -3.])  
x\_eval = np.array([2., 5., 8., 1.])  
y\_eval = np.array([-1.01, -4.1, -7, 0.])  
input\_fn = tf.estimator.inputs.numpy\_input\_fn(  
    {"x": x\_train}, y\_train, batch\_size=4, num\_epochs=None, shuffle=True)  
train\_input\_fn = tf.estimator.inputs.numpy\_input\_fn(  
    {"x": x\_train}, y\_train, batch\_size=4, num\_epochs=1000, shuffle=False)  
eval\_input\_fn = tf.estimator.inputs.numpy\_input\_fn(  
    {"x": x\_eval}, y\_eval, batch\_size=4, num\_epochs=1000, shuffle=False)  
  
# train  
estimator.train(input\_fn=input\_fn, steps=1000)  
# Here we evaluate how well our model did.  
train\_metrics = estimator.evaluate(input\_fn=train\_input\_fn)  
eval\_metrics = estimator.evaluate(input\_fn=eval\_input\_fn)  
print("train metrics: %r"% train\_metrics)  
print("eval metrics: %r"% eval\_metrics)

Cuando se ejecuta, produce

train metrics: {'loss': 1.227995e-11, 'global\_step': 1000}  
eval metrics: {'loss': 0.01010036, 'global\_step': 1000}

Observe cómo el contenido de la model\_fn()función personalizada es muy similar a nuestro bucle de formación de modelo manual de la API de nivel inferior.

Próximos pasos

Ahora usted tiene un conocimiento práctico de los fundamentos de TensorFlow. Tenemos varios tutoriales adicionales que puede consultar para obtener más información. Si usted es un principiante en el aprendizaje de máquina ver [MNIST para principiantes](https://www.tensorflow.org/get_started/mnist/beginners) , de lo contrario ver [Deep MNIST para los expertos](https://www.tensorflow.org/get_started/mnist/pros) .