Tabla de contenido

Introducción 1.1

Tutoriales 1.2

- TensorFlow Mecánica 101 1.2.1

API 1.3

- Python API 1.3.1
  - Framework 1.3.1.1

Comenzando 1.4

- Uso Básico 1.4.1

Comenzando

Vamos a prepararnos para comenzar con TensorFlow!

Pero antes de comenzar, echemos un vistazo a como se ve el código de TensorFlow con la API de Python, para que tengas una idea hacia adonde vamos.

Aquí tenemos un pequeño programa en Python que representa algunos datos en dos dimensiones, con los cuales calcularemos la línea con el mejor ajuste.

```
import tensorflow as tf
import numpy as np

# Crear 100 puntos falsos x, y  en NumPy, y = x * 0.1 + 0.3
x_data = np.random.rand(100).astype(np.float32)
y_data = x_data * 0.1 + 0.3

# Tratar de encontrar valores para W y b que puedan calcular y_data = W * x_data + b

# (Sabemos que W debe de ser 0.1 y b 0.3, pero Tensorflow intentará descubrirlo por no
sotros.)
W = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
b = tf.Variable(tf.zeros([1]))
y = W * x_data + b

# Minimizar los errores de la media cuadrática.

loss = tf.reduce_mean(tf.square(y - y_data))
optimizer = tf.train.GradientDescentOptimizer(0.5)
train = optimizer.minimize(loss)

# Antes de comenzar, inicializar las variables. 'Corramos' esto primero.

init = tf.initialize_all_variables()

# Lanzamos el grafo
sess = tf.Session()
sess.run(init)

# Ajuste de línea.
for step in xrange(201):
    sess.run(train)
    if step % 20 == 0:
        print(step, sess.run(W), sess.run(b))

# Aprende que el mejor ajuste es W: [0.1], b: [0.3]
```

La primera parte de este código crea un grafo de flujo de datos. TensorFlow no ejecuta ningún cómputo hasta que se crea la sesión y la función run es llamada.

Para alimentar aún mas nuestro apetito, te sugerimos que revises como luce un problema clásico de machine learning en TensorFlow. En la tierra de las redes neuronales el problema clásico más "clásico" es la clasificación de dígitos escritos a mano MNIST. Ofrecemos dos introducciones aquí, una para los novatos en machine learning, y otra para profesionales. Si ya has entrenado docenas de modelos MNIST en otros paquetes de software, por favor toma la píldora roja. Si nunca has escuchado acerca de MNIST, definitivamente toma la píldora azul. Pero si consideras ser un punto intermedio se sugerimos probar primero la azul y después la roja.

Si ya estas seguro que deseas aprender e instalar TensorFlow puedes saltarte esto y seguir adelante. No te preocupes, aún asi tendrás oportunidad de ver MNIST -- también usaremos MNIST como ejemplo en un tutorial técnico en donde desarrollaremos las características de

TensorFlow.

Siguientes pasos recomendados

- Descarga e instalación
- Uso básico
- 101 Mecánicas de TensorFlow
- Jugar con TensorFlow en tu navegador

Tutoriales

TensorFlow Mecánica 101

Código: tensorflow / ejemplos / tutoriales / mnist /

El objetivo de este tutorial es mostrar cómo usar TensorFlow para entrenar y evaluar una sencilla red neuronal de tipo Feedforward que clasifica números escritos de forma manual, usando la base de datos MNIST. El público objetivo de este tutorial son aquellos que tengan experciencia en el uso de máquinas de aprendizaje y que esten interesados en utilizar TensorFlow.

En terminos generales estos tutoriales no pretenden enseñar sobre máquinas de aprendizaje.

Por favor asegúrese de haber seguido correctamente las instrucciones para instalar TensorFlow .

Archivos de este tutorial

Este tutorial hace referencia a los siguientes archivos:

Archivo	Propósito
mnist.py	El código para desarrollar un modelo MNIST plenamente conectado (fully-connected MNIST)
fully_connected_feed.py	El código para entrenar el modelo MNIST con el conjunto de datos descargados usando un diccionario de alimentación.

Solo es necesario ejecutar el programa **fully_connected_feed.py** para comenzar el entrenamiento:

```
python fully_connected_feed.py
```

Preparación de los Datos

MNIST es un problema clásico del aprendizaje de máquina. Consiste en análizar las imágenes de los números escritos a mano de tamaño 28x28 píxeles y que estan en una escala de grises para determinar qué dígito representa la imagen, esto incluye todos los números desdee el cero hasta nueve.

Para obtener más información, consulte las página ...

Descargas

Al inicio del método **run_training()**, la función input_data.read_data_sets() se encarga de que los datos para el entrenamiento hayan sido descargados a su carpeta local y despues los descomprime para devolver un diccionario de instancias tipo **DataSet**.

```
data_sets = input_data.read_data_sets(FLAGS.train_dir, FLAGS.fake_data)
```

NOTA: El indicador **fake_data** se utiliza con el fun de realizar pruebas unitarias y puede ser omitido por el lector.

Datos	Propósito
data_sets.train	55000 imagenes y nombres para el entrenamiento principal
data_sets.validation	5000 imagenes y nombres para validar periodicamente la precisión del entrenamiento
data_sets.test	10000 imagenes y nombres, para las pruebas finales de cuan preciso es el entrenamiento

Para obtener más información acerca de los datos, por favor lea el tutorial sobre descargas.

Entradas y marcadores de posición

La función placeholder_inputs() crea dos operaciones tf.placeholder que definen como son las entradas, incluyendo el tamaño (batch_size), esto es para el resto de la gráfica y es con la que se poblaran los ejemplos.

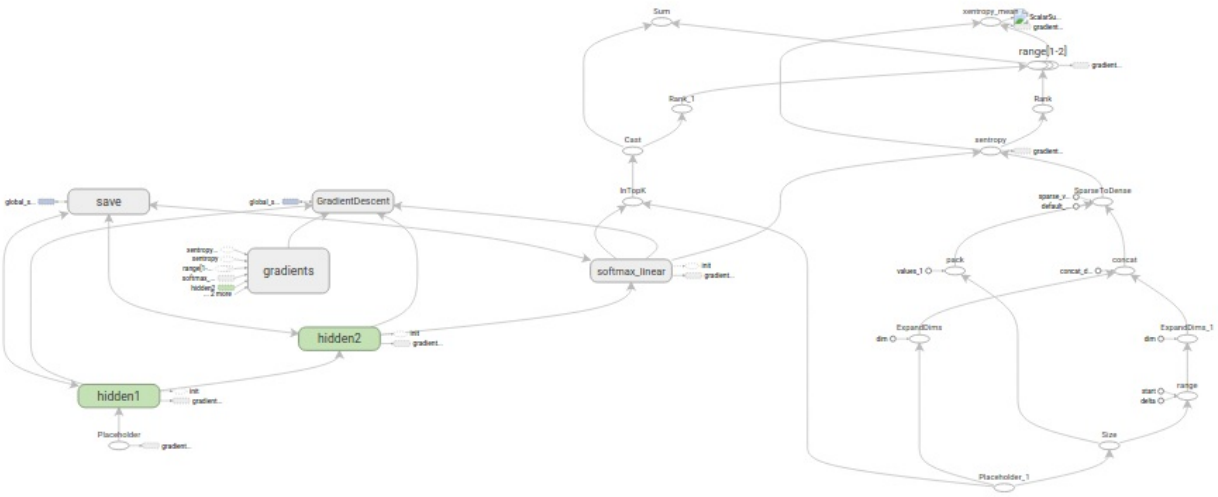
```
images_placeholder = tf.placeholder(tf.float32, shape=(batch_size, mnist.IMAGE_PIXELS)
)
labels_placeholder = tf.placeholder(tf.int32, shape=(batch_size)
```

Mas abajo en el cada ciclo del loop de entrenamiento, cada imagen se separa de su etiqueta para ajustar el batch_size, luego se asocia un marcador de posición y el resultado se pasa como valor dentro parametro **feed_dict** en la función **sess.run()**.

Construyendo el gráfico

Despues de haber creado los marcadores de posición de los datos, el gráfico se construye a partir del programa **mnist.py** de acuerdo con el siguiente patrón de tres etapas: **inference()**, **loss()**, and **training()**.

1. **inference()**: construye el gráfico a medida que se vaya requiriendo para ejecutar la red neuronal (con propagación hacia adelante) y realizar las predicciones.
2. **loss()**: agrega al gráfico de inferencia las operaciones requeridas para poder generar la perdida.
3. **training()**: agrega al gráfico de perdida las operaciones requeridas para calcular y aplicar los gradientes.



Inferencia

La función de inferencia construye el gráfico a medida que se vaya necesitando para retornar el tensor que puede contener las predicciones.

Documentación API

TensorFlow tiene APIs disponibles en varios idiomas, tanto para la construcción como para la ejecución de un gráfico en TensorFlow. La API de Python es en la actualidad la más completa y la más fácil de usar, pero la C++ API puede ofrecer algunas ventajas de rendimiento en ejecución de gráficos, y soporta el despliegue en pequeños dispositivos como Android.

Con el tiempo, esperamos que la comunidad TensorFlow desarrolle frontends para lenguages como Go, Java, JavaScript, Lua, R, y tal vez otras opciones.

Con SWIG, es relativamente fácil de desarrollar una interfaz TensorFlow para su lenguaje preferido.

Python API

Contrucción de Gráficos

- add_to_collection
- as_dtype
- bytes
- control_dependencies
- convert_to_tensor
- convert_to_tensor_or_indexed_slices
- device
- Dimension
- DType
- get_collection
- get_collection_ref
- get_default_graph
- get_seed
- Graph
- GraphKeys
- import_graph_def
- load_file_system_library
- load_op_library
- name_scope
- NoGradient
- op_scope
- Operation
- register_tensor_conversion_function
- RegisterGradient
- RegisterShape
- reset_default_graph
- Tensor
- TensorShape

Contrucción de Gráficos

[Menú con vínculos]

Clases y funciones para la construcción de gráficos en TensorFlow.

tf.add_to_collection(name, value)

Contenedor/envoltorio para **Graph.add_to_collection()** utilizando el gràfico por defecto.

Consulte **Graph.add_to_collection()** apra más detalles.

Argumentos:

- **name**: La clave para la colección.Por ejemplo, la clase **Graph Keys** contiene varios nombres estándares para colecciones.
- **value**: El valor para agregar a la colección.

tf.as_dtype(type_value)

Convierte el **type_value** dado a un **DType**

Argumentos:

- **type_value**: Un valor que se puede convertir a un objeto **tf.DType**. Actualmente esto puede ser un objeto **tf.DType**, una enumeración **DataType**, un nombre tipo cadena, o un **numpy.dtype**.

Retorno: Un **DType** correspondiente a **type_value**.

Lanzamiento:

- TypeError: Si **type_value** no puede ser convertido a **DType**.

class tf.bytes

str(object='') -> cadena

Retorna una linda representación de cadena del objeto. Si el argumento es una cadena, el valor de retorno es el mismo objeto.

tf.control_dependencies(control_inputs)

Contenedor/envoltorio para **Graph.control_dependencies()** utilizado en el gráfico por defecto.

Mire Graph.control_dependencies() para más detalles. **Argumentos**:

- **control_inputs**: una lista de objetos **Operation** o **Tensor** los cuales deben ser ejecutados o computadas antes de la ejecución de las operaciones definidas en el contexto.Puede además ser **None** para limpiar el control de dependencias.

tf.convert_to_tensor(value, dtype=None, name=None, as_ref=False)

Convertir el **value** dado a un **Tensor**

Esta función convierte objetos Python de varios tipos a objetos **Tensor**. Se aceptan objetos **Tensor**, matrices numpy, listas Python,escalares Python. Por ejemplo:

```
import numpy as np

def my_func(arg):
  arg = tf.convert_to_tensor(arg, dtype=tf.float32)
  return tf.matmul(arg, arg) + arg

# The following calls are equivalent.
value_1 = my_func(tf.constant([[1.0, 2.0], [3.0, 4.0]]))
value_2 = my_func([[1.0, 2.0], [3.0, 4.0]])
value_3 = my_func(np.array([[1.0, 2.0], [3.0, 4.0]], dtype=np.float32))
```

Esta función puede ser útil al componer una nueva operación en Python (como el caso de **mi_func** en el anterior ejemplo). Toda operación constructor Python estándar aplica esta función para cada de uno de los **Tensor-value** ingresados. Los cuales permiten a las operaciones aceptar arreglos numpy, listas Python y escalares además de objetos **Tensor**. **Argumentos**:

- **value**: Un objeto cuyo tipo tiene registrada una función de conversión **Tensor**.
- **dtype**: Tipo de elemento opcional para el tensor de regreso.Si este falta, el tipo se deduce a partir del tipo de **value**.
- **name**: Nombre opcional que se utilizará si se crea un nuevo **Tensor**.
- **as_ref**: "True" si queremos que el resultado como una referencia de tensor. Solo utilizado si se crea un nuevo **Tensor**.

Retorno: Un **Tensor** basado en **value**.

Lanzamiento:

- **TypeError**:Si no se ha registrado la función de conversión de **value**.

- **RuntimeError**: Si una función de conversión registrada devuelve un valor no válido.

Uso Básico

Para usar TensorFlow es necesario entender como TensorFlow:

- Representa cálculos en forma de grafos.
- Ejecuta grafos en el contexto de sesiones.
- Representa los datos en forma de tensors.
- Mantiene el estado con Variables.
- Usa operaciones de alimentación y recuperación para obtener datos dentro y fuera de operaciones arbitrarias.

Resumen

Tensor Flow es un sistema de programación, en el cuál puedes representar cálculos computacionales como grafos. Los Nodos en el grafo son llamados ops (cortos para operaciones)- Un op toma cero o mas Tensors, desempeña cierta operación computacional y produce cero o más Tensors. Un Tensor es un arreglo multidimensional tipificado. Por ejemplo, tu puedes representar una mini-batch de imágenes como un arreglo 4-D de puntos flotantes con dimensiones [dimensiones, altura, anchura, canales].

El cómputo basado en grafos

Los programas de TensorFlow son usualmente estructurados dentro de una fase de construcción, y una fase de ejecución que utiliza una sesión para ejecutar ops en el grafo.

Por ejemplo, es común crear un grafo para representar y entrenar una red neuronal en la fase de construcción, y entonces ejecutar repetidamente un conjunto de ops entrenadas en la fase de ejecución.

TensorFlow puede ser utilizado en programas como: C, C++ y Python. Esto representa una manera mucho más sencilla de utilizar las librerias de Python para unir grafos, como tambien se prevee de un gran conjunto de funciones útiles que no estan disponibles en C y C++.

Las librerías de sesión tienen funcionalidades equivalente paralos tres lenguajes.

Construyendo el grafo

Para construir un grafo comienza con los ops que no necesitan ninguna entrada (ops fuente), como una constante, y pasa su salida a otro ops para realizar el cálculo cómputacional.

El constructor de ops en la librería de Python regresa objetos que se mantienen para la salida de las ops construidas. Usted puede pasarles a otras ops construidas para usarlas como entradas.

La librería de Python para TensorFlow tiene un grafo por default el cual agrega nodos a los ops constructores. El grafo por defecto es suficiente para muchas aplicaciones. Vea la documentación de la clase Graph para como manejar explícitamente múltiples grafos.

```
import tensorflow as tf

# Create a Constant op that produces a 1x2 matrix.  The op is
# added as a node to the default graph.
#
# The value returned by the constructor represents the output
# of the Constant op.
matrix1 = tf.constant([[3., 3.]])

# Create another Constant that produces a 2x1 matrix.
matrix2 = tf.constant([[2.],[2.]])

# Create a Matmul op that takes 'matrix1' and 'matrix2' as inputs.
# The returned value, 'product', represents the result of the matrix
# multiplication.
product = tf.matmul(matrix1, matrix2)
```

El grafo por defecto ahora tiene tres nodos: dos constant() ops y uno matmul() op. Para multiplicar las matrices, y obtener el resultado de la multiplicación, usted debe lanzar el grafo firmado en su sesión personal.```

Lanzando el Grafo en una sesión

El lanzamiento requiere la siguiente construcción. Para lanzar el Grafo, cree un objeto Session. Sin argumentos, el constructor de la sesión lanza el grafo por default.

```
# Launch the default graph.
sess = tf.Session()
# To run the matmul op we call the session 'run()' method, passing 'product'
# which represents the output of the matmul op.  This indicates to the call
# that we want to get the output of the matmul op back.
#
# All inputs needed by the op are run automatically by the session.  They
# typically are run in parallel.
#
# The call 'run(product)' thus causes the execution of three ops in the
# graph: the two constants and matmul.
#
# The output of the op is returned in 'result' as a numpy `ndarray` object.
result = sess.run(product)
print(result)
# ==> [[ 12.]]

# Close the Session when we're done.
sess.close()
```

La sesión debe cerrarse para liberar recursos. Tambien puede ingresar a la sesión con un bloque "with". La sessión es cerrada automáticamente al final del bloque "with".

```
with tf.Session() as sess:
  result = sess.run([product])
  print(result)
```

La implementación de TensorFlow traduce la definición de un grafo en operaciones distribuidas ejecutables a lo largo de recursos de cómputo disponibles, como CPU o una de las tarjetas GPU en su computadora. En general no necesita especificar los CPUs o GPUs explícitamente. TensorFlow utiliza el primer GPU, si usted cuenta con uno, para la mayoría de las operaciones como sea posible.

Si usted cuenta con más de un GPU dispinible en su computadora, para usar un GPU más allá del primero usted debe asignarle *ops* explícitamente. Use la instrucción *with...Device* para específicar cual CPU o GPU usar para la operación:

```
with tf.Session() as sess:
  with tf.device("/gpu:1"):
    matrix1 = tf.constant([[3., 3.]])
    matrix2 = tf.constant([[2.],[2.]])
    product = tf.matmul(matrix1, matrix2)
    ...
```

Los dispositivos con especificados como strings. Los dispositivos actualmente soportados son:

- "/cpu:0": The CPU of your machine.
- "/gpu:0": The GPU of your machine, if you have one.
- "/gpu:1": The second GPU of your machine, etc.

Vea Usando_GPUs para más información sobre GPUs y TensorFlow

Lanzando el Grafo en una sesión distribuida

Para crear un cluster de TensorFlow, lance un server de TensorFlow en cada una de las máquinas en el clúster. Cuando usted inicia una sesión en su cliente, usted debe enviar la localización en la red de una de las máquinas en el clúster:

```
with tf.Session("grpc://example.org:2222") as sess:
  # Calls to sess.run(...) will be executed on the cluster.
  ...
```

Esta máquina se convierte en el maestro de la sesión. El maestro distribuye el Grafo a través de otras máquinas en el cluster(workesrs), parecido en la manera en la que la implementación local distribuye el Grafo através de recursos de cómputo dentro de una máquina.

Usted puede usar "with tf.device():" para especificar directamente los workers para partes espcíficas del Grafo:

```
with tf.device("/job:ps/task:0"):
  weights = tf.Variable(...)
  biases = tf.Variable(...)
```

Vea TensorFlow_Distribuido_HowTo para más información sobre sesiones distribuidoas y clusters

Uso interactivo

Los ejemplos de Python en la documentación lanzan el Grafo con una **Session** y usa **Session.run()**

Para facilitar el uso de ambientes interactivos de Python, como **IPython** usted puede utilizar la clase **InteractiveSession**, y el **Tensor.eval()** y los métodos **Operation.run()**. Esto permitirá mantener una variable para la sesión:

```
# Enter an interactive TensorFlow Session.
import tensorflow as tf
sess = tf.InteractiveSession()

x = tf.Variable([1.0, 2.0])
a = tf.constant([3.0, 3.0])

# Initialize 'x' using the run() method of its initializer op.
x.initializer.run()

# Add an op to subtract 'a' from 'x'.  Run it and print the result
sub = tf.sub(x, a)
print(sub.eval())
# ==> [-2. -1.]

# Close the Session when we're done.
sess.close()
```

Tensors

Los programas de TensorFlow usan una estructura de datos tensor para representar todos los datos -- sólo los tensors pasan por las operaciones del Grafo computacional. Usted puede pensar en un tensor TensorFlow como un arreglo de n-dimensiones o una lista. Un tensor tiene un tipos estático, un rango y una forma. Para aprender más sobre como soporta TensorFlow este concepto, vea Rango,Forma y Tipo.

Variables

Las variables mantienen su estado durante la ejecución del Grafo. Los siguientes ejemplos muestran una variable sirviendo como un simple contador. Vea Variables para más detalles.

```
# Create a Variable, that will be initialized to the scalar value 0.
state = tf.Variable(0, name="counter")

# Create an Op to add one to `state`.

one = tf.constant(1)
new_value = tf.add(state, one)
update = tf.assign(state, new_value)

# Variables must be initialized by running an `init` Op after having
# launched the graph.  We first have to add the `init` Op to the graph.
init_op = tf.initialize_all_variables()

# Launch the graph and run the ops.
with tf.Session() as sess:
  # Run the 'init' op
  sess.run(init_op)
  # Print the initial value of 'state'
  print(sess.run(state))
  # Run the op that updates 'state' and print 'state'.
  for _ in range(3):
    sess.run(update)
    print(sess.run(state))

# output:

# 0
# 1
# 2
# 3
```

La operación **assing()** en este código es una parte de la expresión del Grafo como la operación add(), pero esto no realiza la asignación hasta que se ejecuta la expresión **run()**

Típicamente usted puede representar los parámetros de un modelo estadístico como un conjunto de Variables. Por ejemplo, usted quisiera almacenar los pesos de una red nueronal como un tensor en una variable. Duante el entrenamiento usted actualiza este tensor corriendo un Grafo de entrenamiento repetidamente.

Fetches

Para extraer las salidas de las operaciones, ejecute el Grafo con una llamda run() sobre el objeto Session y pasar en el tensor para recuperarlo. En el ejemplo anterior, nosotros extrajimos el nodo sencillo state, pero también podemos extraer múltiples tensors:

```
input1 = tf.constant([3.0])
input2 = tf.constant([2.0])
input3 = tf.constant([5.0])
intermed = tf.add(input2, input3)
mul = tf.mul(input1, intermed)

with tf.Session() as sess:
  result = sess.run([mul, intermed])
  print(result)

# output:
# [array([ 21.], dtype=float32), array([ 7.], dtype=float32)]
```

Todas las ops necesarios para producir los valores de los tensores requeridos son corridos una vez (no una vez para cada ocasión que son requeridos).

Feeds

Los ejemplos anteriores introduce a los tensors en un Grafo computacional almacenándole en Constantes y Variables. TensorFlow tambien provee un mecanismo de alimentación para ajustar un tensor directamente en una operación en el Grafo.

Un feed temporalmente reemplaza una salida de una operación con un valor del tensor. Usted aliemnta los datos del feed como un argumento de la llamada a run(). El feed es solamente utilizado para ejecutar call en el que es enviado. El caso más común envuelve operaciones especialmente diseñadas para ser operaciones "feed" usando: tf.placeholder() para crearlos:

```
input1 = tf.placeholder(tf.float32)
input2 = tf.placeholder(tf.float32)
output = tf.mul(input1, input2)

with tf.Session() as sess:
  print(sess.run([output], feed_dict={input1:[7.], input2:[2.]}))

# output:
# [array([ 14.], dtype=float32)]
```

La operación placeholer() genera un error si usted no puede soportar un feed para esto. Vea MNIST fully-connected feed tutorial (source code) para código más complejos de Feeds.