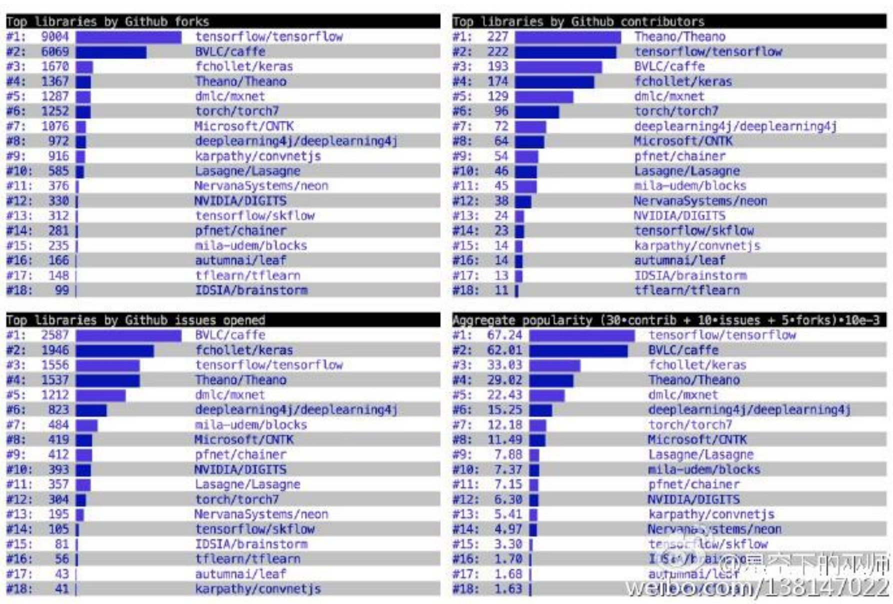
Introduction to tensorflow and keras

Yu-Heng Hsieh

11/14



Tensorflow

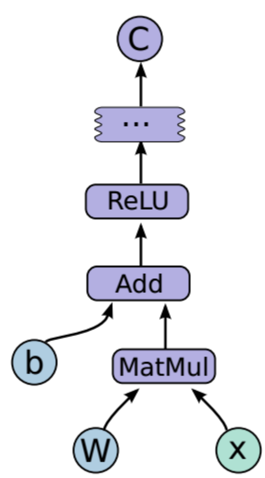
- Google's attempt to put the power of Deep Learning into the hands of developers around the world.
- TensorFlow is an open source software library for numerical computation using data flow graphs.
- Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them.
- Single API that supports one to more CPU and GPU on various platforms

Tensorflow

- we typically use libraries like Numpy that do expensive operations such as matrix multiplication outside Python, using highly efficient code implemented in another language.
- There can still be a lot of overhead from switching back to Python every operation
- TensorFlow also does its heavy lifting outside Python, but it takes things a step further to avoid this overhead.
- TensorFlow lets us describe a graph of interacting operations that run entirely outside Python.

Computational graph

- The tensorflow core program consisting of two operations
  - Building the computational graph
  - Running the computational graph
- A **computational graph** is a series of TensorFlow operations arranged into a graph of nodes.
- Each node takes zero or more tensors as inputs and produces a tensor as an output



Constant

- One type of node is a constant, which takes no input and output the value it stores internally.

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

- The output of the above code

```
Tensor("Const:0", shape=(), dtype=float32)
Tensor("Const_1:0", shape=(), dtype=float32)
```

- Printing the node does not print 3 or 4
- These are nodes that, when evaluated, outputs 3 or 4

Session

- To evaluate the nodes, we need to run the computation graph in a session

```
sess = tf.Session()
print(sess.run([node1, node2]))
[3.0 4.0]
```

- We can also build a more complicated computation.

```
node3 = tf.add(node1, node2)
print("node3:", node3)
print("sess.run(node3):", sess.run(node3))
Output:
node3: Tensor("Add:0", shape=(), dtype=float32)
sess.run(node3): 7.0
```

PlaceHolder

- Previous examples provide constant results (Boring!)
- A graph can be parameterized to accept external inputs, known as **placeholders**.
- A **placeholder** is a promise to provide a value later.

```
a = tf.placeholder(tf.float32)
b = tf.placeholder(tf.float32)
adder_node = a + b
```

- Works like a lambda function

```
print(sess.run(adder_node, {a: 3, b: 4.5}))
print(sess.run(adder_node, {a: [1, 3], b: [2, 4]}))
7.5
[ 3.  7.]
```

Variable

- In machine learning we will typically want a model that can take arbitrary inputs
- To make the model trainable, we need to be able to modify the graph to get new outputs with the same input.
- **Variables** allow us to add trainable parameters to a graph.
- They are constructed with a type and initial value.

```
W = tf.Variable([.3], dtype=tf.float32)
b = tf.Variable([-.3], dtype=tf.float32)
x = tf.placeholder(tf.float32)
linear_model = W*x + b
```

Variable

- Constants are initialized when you call *tf.constant*, and their value can never change.
- Variables are not initialized when you call tf.Variable.
- To initialize all the variables in a TensorFlow program

```
init = tf.global_variables_initializer()
sess.run(init)
```

- It is important to realize init is a handle to the TensorFlow sub-graph that initializes all the global variables. Until we call sess.run, the variables are uninitialized.

Loss function

- We've created a model, but we don't know how good it is yet. To evaluate the model on training data, we need another placeholder to provide the desired values, and we need to write a loss function.

```
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]}))
23.66
```

Change variable value

- Variable value are variable with tf.assign operation

```
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]}))
```

Model training

- TensorFlow provides **optimizers** that slowly change each variable in order to minimize the loss function.

```
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]))
```

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

Session and Interactive Session

- A Session object encapsulates the environment in which Operation objects are executed, and Tensor objects are evaluated.
- Always remember to close after you finish the task to release resources.
- Use with…as… statement to automatically close.
- Interactive Session is a default session.
- You don’t have to specify the session if using interactive session
- with tf.Session(): works the same as interactive session

Regression with tensorflow

- Classify the MNIST data set 28 x 28 pixel each picture
- Create the input of picture and the correct answer

```
x = tf.placeholder(tf.float32, [None, 784])
y_ = tf.placeholder(tf.float32, [None, 10])
```

- Create the variable for regression, the weight and the bios

```
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
```

- Process the output with softmax

```
y = tf.nn.softmax(tf.matmul(x, W) + b)
```

Regression with tensorflow

- Define the error, using cross entropy

```
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y),
reduction_indices=[1]))
```

- Now that the model has been built and the error is defined, lets train the model

```
train_step =
tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
```

- The above code define the training operation step, to start the real training

```
sess = tf.InteractiveSession()
tf.global_variables_initializer().run()
for _ in range(1000):
  batch_xs, batch_ys = mnist.train.next_batch(100)
  sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

Regression with tensorflow

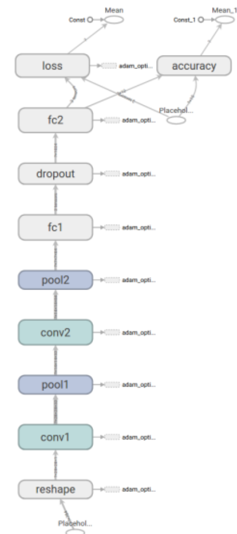
- Finally, we evaluate the result through

```
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

- The accuracy of such model is about 92%, which is actually pretty bad
- How can we improve the accuracy?

Neural Network with tensorflow

- A more complicated CNN model
- Two convolution layer
- Two fully connected layer



Weight Initialization

- NN requires lots of weight and bias
- The weight should be initialize with small noise for symmetry breaking and prevent 0 gradient
- Let’s build a function to avoid repeated initialization

```
def weight_variable(shape):
  initial = tf.truncated_normal(shape, stddev=0.1)
  return tf.Variable(initial)

def bias_variable(shape):
  initial = tf.constant(0.1, shape=shape)
  return tf.Variable(initial)
```

Convolution and Pooling

- Again to avoid repeated declaration of convolution and max pooling, we define the functions.

```
def conv2d(x, W):
  return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max_pool_2x2(x):
  return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                        strides=[1, 2, 2, 1], padding='SAME')
```

- The meaning of stride [batch, x, y, depth]
- The padding can be 'SAME' or 'VALID'

Build the model – First Conv layer

- First convolution + max pooling layer
- Initialize the weight

```
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
```

- [5, 5] is the size of filter, 1 is the number of input channel, 32 is the output channel

```
x_image = tf.reshape(x, [-1, 28, 28, 1])
```

- -1 means to adjust the size to match the need

```
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)
```

- What is the output size?

Build the model – Second Layer

- The second layer is the same as the first one except for the size of the weight and bias

```
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])

h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)
```

- What is the output size now?

Build the model –FC layer

- Now the image size is 7 x 7 with 64 channel
- Add a fully connected layer with 1024 neurons

```
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])

h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
```

- Add dropout layer to avoid overfitting

```
keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

Build the model – Readout Layer

- The output layer

```
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])
y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2
```

- Evaluate the error with cross entropy

```
cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y_conv))
```

- Define optimizer

```
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
```

- Calculate the accuracy

```
correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Train the model

```
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  for i in range(20000):
    batch = mnist.train.next_batch(50)
    if i % 100 == 0:
      train_accuracy = accuracy.eval(feed_dict={
          x: batch[0], y_: batch[1], keep_prob: 1.0})
      print('step %d, training accuracy %g' % (i, train_accuracy))
    train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 
0.5})

  print('test accuracy %g' % accuracy.eval(feed_dict={
      x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))
```

Tensorflow

- Tensorflow is powerful and flexible tool
- Can build various kinds of networks
- However, the syntax and logic behind the code might be difficult to understand

Keras

- Keras can be considered a wrapper for Tensorflow and Theano
- It's a highly modulized framework designed for fast building of the networks
- Use Keras if you need a deep learning library that:
  - Allows for easy and fast prototyping (through user friendliness, modularity, and extensibility).
  - Supports both convolutional networks and recurrent networks, as well as combinations of the two.
  - Runs seamlessly on CPU and GPU.

Sequential Model

- The Sequential model is a linear stack of layers.
- Create Sequential model

```
from keras.models import Sequential
from keras.layers import Dense, Activation
model = Sequential([
Dense(32, input_shape=(784,)),
Activation('relu'),
Dense(10),
Activation('softmax'), ])
```

- Another way to add the layer is through *.add()* method

```
model = Sequential()
model.add(Dense(32, input_dim=784))
model.add(Activation('relu'))
```

Sequential Model

- Before training a model, you need to configure the learning process
- The .compile() method build such process with 3 input argument
  - An optimizer: may be “rmsprop”, “adagrad”…
  - A loss function for the model to minimize
  - A list of metrics to evaluate the performance of the mode

```
# For a multi-class classification problem
model.compile(optimizer='rmsprop', loss='categorical_crossentropy',
metrics=['accuracy'])

# For a binary classification problem
model.compile(optimizer='rmsprop', loss='binary_crossentropy',
metrics=['accuracy'])

# For a mean squared error regression problem
model.compile(optimizer='rmsprop', loss='mse')
```

Core layers

- Dense
- Activation
- Dropout
- Flatten
- Reshape

Dense

- The regular densely connected NN layer

```
model = Sequential()
model.add(Dense(32, input_shape=(16,)))
```

- The *input_shape* should be specified in the first layer of a model
- The model now takes the input array of size (*, 16) and output an array of (*,32)

Activation

- The activation layer
  - softmax
  - elu
  - selu
  - softplus
  - relu
  - tanh
  - sigmoid
  - hard_sigmoid
  - linear

Dropout, Flatten, Reshape

- Dropout(rate, noise_shape, seed)
  - Dropout consists in randomly setting a fraction rate of input units to 0 at each update during training time, which helps prevent overfitting.
- Flatten()
  - Flattens the input. Does not affect the batch size.
- Reshape(new_shape)

Other layers

- Convolution layers
  - Conv1D
  - Conv2D
  - SeperableConv2D
  - Conv2DTranspose
  - Conv3D
  - Cropping1-3D
  - ZeroPadding1-3D
- Pooling layers
  - MaxPooling1D
  - MaxPooling2D
  - MaxPooling3D
  - AveragePooling1-3D
  - GlobalMaxPooling1-2D
  - GlobalAveragePooling1-2D

Building a model with Keras

- Let's build a model that is the same as the previous tensorflow model

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, MaxPooling2D
model = Sequential()
# the first convolution layer
model.add(Convolution2D(32, 3, 3, border_mode='same',
                                dim_ordering = 'tf',
                                input_shape=(1,28,28)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2), dim_ordering = 'tf‘, padding
= ‘same’))
```

Building a model with Keras

- The second convolution layer

```
model.add(Convolution2D(64, 3, 3))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2), dim_ordering = 'tf‘, padding
= ‘same’))
```

- The FC layer

```
model.add(Flatten())
model.add(Dense(1024))
model.add(Activation('relu'))
model.add(Dropout(0.5))
```

Building a model with Keras

- The Readout layer

```
model.add(Dense(nb_classes))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

- To train the model

```
model.fit(X, y,
          batch_size=batch_size,
          nb_epoch=nb_epoch,
          shuffle=True,
          verbose = 1)
```

- To predict the testing set

model.predict_classes(test)