Deep Learning for Computer Vision with TensorFlow

Hanock Kwak
2017-08-24
Seoul National University
Preliminary

• Machine Learning
• Deep Learning
• Linear Algebra
• Python (numpy)
Throughout the Slides

• Please put following codes to run our sample codes.

```python
import numpy as np
import tensorflow as tf
```

• All codes are written in python 3.x and TensorFlow 1.x.
• We tested codes in Jupyter Notebook.
What is TensorFlow?
What is TensorFlow?

• TensorFlow was originally developed by researchers and engineers working on the Google Brain Team.

• TensorFlow is an open source software library for numerical computation using data flow graphs.

• It deploys computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API.
TensorFlow Architecture

• Core in C++
  • Very low overhead
• Different front ends for specifying/driving the computation
  • Python and C++ today, easy to add more

https://www.slideshare.net/JenAman/large-scale-deep-learning-with-tensorflow
Graphs in TensorFlow

- Computation is a **dataflow graph**.
- A variable is defined as a **symbol**.

```python
a = tf.Variable(3)
b = tf.Variable(2)
c = tf.Variable(1)
x = a*b
y = x + c
```
Device Placement

• A variable or operator can be pinned to a particular device.

# Pin a variable to CPU.
with tf.device("/cpu:0"):  
a = tf.Variable(3)  
b = tf.Variable(2)  
x = a*b

# Pin a variable to GPU.
with tf.device("/gpu:0"):  
c = tf.Variable(1)  
y = x + c
Distributed Systems of GPUs and CPUs
TensorFlow in Distributed Systems

TensorFlow in Distributed Systems cont.

Image Model Training Time

Precision @ 1

2.6 hours vs. 79.3 hours (30.5X)

50 GPUs
10 GPUS
1 GPU

https://www.slideshare.net/JenAman/large-scale-deep-learning-with-tensorflow
Partial Flow

- TensorFlow executes a **subgraph** of the whole graph.
- We do not need “e” and “d” to compute “f”.

Graph Optimizations

- Common Subexpression Elimination
- Controlling Data Communication and Memory Usage
- Asynchronous Kernels
- Optimized Libraries for Kernel Implementations
  - BLAS, cuBLAS, GPU, cuda-convnet, cuDNN
- Lossy Compression
  - $32 \to 16 \to 32$-bit conversion
What is Tensor?
Tensor

- A tensor is a **multidimensional data array**.

<table>
<thead>
<tr>
<th>Order</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Scalar</td>
<td>Vector</td>
<td>Matrix</td>
<td>Cube?</td>
</tr>
<tr>
<td>100</td>
<td>[5, 3, 7, ..., 10]</td>
<td>[1, 2]</td>
<td>[1, 2]</td>
<td></td>
</tr>
</tbody>
</table>
Shape of Tensor

- **List of dimensions** for each order.
- Shape = [4, 5, 2]

```python
V = tf.Variable(tf.zeros([4, 5, 2]))
```
Reshape

- Reshapes the tensor.

\[
V = \text{tf.Variable}(\text{tf.zeros}([4, 5, 2]))
\]

\[
W = \text{tf.reshape}(V, [4, 10])
\]
Transpose

• Transposes tensors.
• Permutes the dimensions.

```python
d = np.arange(2*3*4)
x = tf.Variable(d)
x = tf.reshape(x, [2, 3, 4])
y1 = tf.transpose(x, [0, 2, 1])
y2 = tf.transpose(x, [2, 0, 1])
y3 = tf.transpose(x, [1, 2, 0])

print(y1.get_shape())  # (2, 4, 3)
print(y2.get_shape())  # (4, 2, 3)
print(y3.get_shape())  # (3, 4, 2)
```
Concatenation

- Concatenate two or more tensors.

**tf.concat**

```python
concat(
    values,  # tensor t1 with shape [2, 3]
    axis,    # tensor t2 with shape [2, 3]
    name='concat'
)
```

```python
t3 = tf.concat([t1, t2], 0)  # ==> [4, 3]
t4 = tf.concat([t1, t2], 1)  # ==> [2, 6]
```
Reduce Operations

- Computes an operation over elements across dimensions of a tensor.
  - `tf.reduce_sum(...)`, `tf.reduce_prod(...)`, `tf.reduce_max(...)`, `tf.reduce_min(...)`

```python
# 'x' is [[1, 1, 1],
#        [1, 1, 1]]

# tf.reduce_sum(x) # ==> 6
# tf.reduce_sum(x, 0) # ==> [2, 2, 2]
# tf.reduce_sum(x, 1) # ==> [3, 3]
# tf.reduce_sum(x, 1, keep_dims=True) # ==> [[3], [3]]
# tf.reduce_sum(x, [0, 1]) # ==> 6
```
Matrix Multiplication

• Matrix multiplication with two tensors of order 2.

```python
# 2-D tensor `a`
a = tf.constant([1, 2, 3, 4, 5, 6], shape=[2, 3])  # => [[1. 2. 3.]  
                                             
                                             [4. 5. 6.]]

# 2-D tensor `b`
b = tf.constant([7, 8, 9, 10, 11, 12], shape=[3, 2])  # => [[7. 8.]  
                                             
                                             [9. 10.]
                                             [11. 12.]]

c = tf.matmul(a, b)  # => [[58 64]  
                          [139 154]]
```
Broadcasting

- Broadcasting is the process of making arrays with different shapes have compatible shapes for arithmetic operations.
  - This is similar to that of numpy
- Adding a vector to a matrix.

\[
\begin{pmatrix}
1 & 2 & 3 \\
4 & 5 & 6
\end{pmatrix} + \begin{pmatrix}
7 & 8 & 9 \\
4 & 5 & 6
\end{pmatrix} = \begin{pmatrix}
8 & 10 & 12 \\
11 & 13 & 15
\end{pmatrix}
\]

- Adding a scalar to a matrix

\[
\begin{pmatrix}
1 & 2 & 3 \\
4 & 5 & 6
\end{pmatrix} + 7 = \begin{pmatrix}
8 & 9 & 10 \\
11 & 12 & 13
\end{pmatrix}
\]
Gradients

• Constructs symbolic partial derivatives.

```python
# Build a graph.
x = tf.placeholder(tf.float32, shape=())
y = x*x + tf.sin(x)
g = tf.gradients(y, x)  # 2*x + cos(x)

# Launch the graph in a session.
sess = tf.Session()

# Evaluate the tensor `g`.
print(sess.run(g, {x:0.0}))  # 1.0
print(sess.run(g, {x:np.pi}))  # 5.2831855
```
Variables, Graph, and Session
Variables

- Variables are in-memory buffers containing tensors.
- All variables have names.
  - If you do not give a name, then unique name will be automatically assigned.

```python
# Various ways to create variables.
x = tf.Variable(tf.zeros([200]), name="x")
y = tf.Variable([[1, 0], [0, 1]]) # identity matrix
z = tf.constant(6.0) # this is also a variable that does not change!
learning_rate = tf.Variable(0.01, trainable=False) # not trainable!
```
Initialization of Variables and Session

• Variables initializer must be called before other ops in your model can be run.

• A session encapsulates the control and state of the TensorFlow runtime.

• A graph is created and allocated in memory when the session is created.

```python
# Add an op to initialize the variables.
init_op = tf.global_variables_initializer()

# Later, when launching the model with tf.Session() as sess:
# Run the init operation.
sess.run(init_op)

# Use the model
...```
sess.run()

- Runs operations and evaluates tensors.
- You may feed values to specific variables in the graph.

```python
# Build a graph.
a = tf.constant(5.0)
b = tf.constant(6.0)
c = a * b

# Launch the graph in a session.
sess = tf.Session()

# Evaluate the tensor `c`.
print(sess.run(c))  # 30.0
print(sess.run(c, {b:3.0}))  # 15.0
print(sess.run(c, {a:1.0, b:2.0}))  # 2.0
print(sess.run(c, {c:100.0}))  # 100.0
```
# Placeholders

- Inserts a placeholder for a variable that will be always fed.
- Pass type and shape for the placeholders.

```python
# Build a graph.
a = tf.placeholder(tf.float32, shape=())  # scalar tensor
b = tf.constant(6.0)
c = a * b

# Launch the graph in a session.
sess = tf.Session()

# Evaluate the tensor `c`.
print(sess.run(c))  # error !
print(sess.run(c, {b:3.0}))  # error !
print(sess.run(c, {a:2.0}))  # 12.0
```
Variable Update

• Variables can be updated through assign(...) function.

```python
# Build a graph.
x = tf.Variable(100)
assign_op = x.assign(x - 1)

# Launch the graph in a session.
sess = tf.Session()

# Run assign_op
sess.run(tf.global_variables_initializer())
print(sess.run(assign_op)) # 99
print(sess.run(assign_op)) # 98
print(sess.run(assign_op)) # 97
```
Problems with Variables

- Sometimes we want to reuse same set of variables.
- Whenever Variable is called it only creates new variable.
- How can we reuse same variable?

```python
# define function
def f(x):
    b = tf.Variable(tf.random_normal([10], stddev=1.0))
    return x + b

...y1 = f(x1)
y2 = f(x2)  # it use different ‘b’ variable
```
Sharing Variables: tf.get_variable()

- The function `tf.get_variable()` is used to get or create a variable instead of a direct call to `tf.Variable`.

```python
# define function
def f(x):
    b = tf.get_variable('b', [10], initializer=tf.random_normal_initializer())
    return x + b

... with tf.variable_scope("bias") as scope:
    y1 = f(x1)
    scope.reuse_variables()
    y2 = f(x2) # it use same 'b' variable
```
How Does Variable Scope Work?

- Variable scope wraps variables with a namespace.
- Reusing variables is only valid within the scope.

```python
with tf.variable_scope("foo"):
    v = tf.get_variable("v", [1])
assert v.name == "foo/v:0"

with tf.variable_scope("foo"):
    v = tf.get_variable("v", [1])
with tf.variable_scope("foo", reuse=True):
    v1 = tf.get_variable("v", [1])
assert v1 is v
```
with tf.variable_scope("root"):
    # At start, the scope is not reusing.
    assert tf.get_variable_scope().reuse == False
with tf.variable_scope("foo"):
    # Opened a sub-scope, still not reusing.
    assert tf.get_variable_scope().reuse == False
with tf.variable_scope("foo", reuse=True):
    # Explicitly opened a reusing scope.
    assert tf.get_variable_scope().reuse == True
with tf.variable_scope("bar"):
    # Now sub-scope inherits the reuse flag.
    assert tf.get_variable_scope().reuse == True
    # Exited the reusing scope, back to a non-reusing one.
    assert tf.get_variable_scope().reuse == False
Caution: Name Duplication

• Calling \texttt{tf.get\_variable()} twice with same name when reuse is off, invokes error.

\begin{verbatim}
b1 = tf.get_variable('b', [10], initializer=tf.random_normal_initializer())
b2 = tf.get_variable('b', [10], initializer=tf.random_normal_initializer()) # error!
\end{verbatim}

\textbf{ValueError}: Variable \texttt{b} already exists, disallowed. Did you mean to set \texttt{reuse=True} in \texttt{VarScope}? Originally defined at:
Saving Variables

- Call `tf.train.Saver()` to manage all variables in the model.

```python
# Add an op to initialize the variables.
init_op = tf.global_variables_initializer()

# Add ops to save and restore all the variables.
saver = tf.train.Saver()

# Later, launch the model, initialize the variables, do some work, save the # variables to disk.
with tf.Session() as sess:
    sess.run(init_op)
    # Do some work with the model.
    ...
    # Save the variables to disk.
save_path = saver.save(sess, "/tmp/model.ckpt")
print("Model saved in file: %s" % save_path)
```
Restoring Variables

- The same Saver object is used to restore variables.

```python
...  
# Add ops to save and restore all the variables.  
saver = tf.train.Saver()

# Later, launch the model, use the saver to restore variables from disk, and  
# do some work with the model.  
with tf.Session() as sess:
    # Restore variables from disk.  
saver.restore(sess, "/tmp/model.ckpt")
print("Model restored.")
# Do some work with the model
...  
```
Convolutional Neural Network in TensorFlow
Four Main Components in Machine Learning

• Hypothesis space
• Objective function
• Optimization algorithm
• Data
Convolution Operations: conv1d, 2d, 3d

- TensorFlow provides convolution operations.

conv1d

conv2d

conv3d
tf.nn.conv2d()

- Computes a 2-D convolution given 4-D input and filter tensors.
- Input is 4-D tensor.
  - shape=(batch_size, height, width, channels)
- Filter is 4-D tensor.
  - shape=(filter_height, filter_width, in_channels, out_channels)
- Stride is a size of the sliding window for each dimension of input.
tf.nn.conv2d() Padding

• padding = “VALID”
  • Do not use zero padding.
  • Size of filter map shrinks.
    • out_height = ceil((in_height - filter_height + 1) / strides[1])
    • out_width = ceil((in_width - filter_width + 1) / strides[2])

• padding = “SAME”
  • Tries to pad zeros evenly left and right to preserve width and height.
  • If the amount of columns to be added is odd, it will add the extra column to the right.
    • out_height = ceil(in_height / strides[1])
    • out_width = ceil(in_width / strides[2])
import tensorflow as tf
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
%matplotlib inline

# constants
batch_size = 1
img_height = 224
img_width = 224
img_channel = 3

# Build a graph.
x = tf.placeholder(tf.float32, [batch_size, img_height, img_width, img_channel])
w = tf.Variable(tf.random_normal([5, 5, 3, 64], stddev=0.35))
output = tf.nn.conv2d(x, w, strides=[1, 2, 2, 1], padding='SAME')
tf.nn.conv2d() Example cont.

```python
# Launch the graph in a session.
with tf.Session() as sess:
    tf.global_variables_initializer().run()

    img = np.array(Image.open('test.jpg'))
    plt.imshow(img)
    plt.show()
    img = img.reshape([1, img_height, img_width, img_channel])

    _out = sess.run(output, {x: img})
    print(_out.shape)
    plt.imshow(_out[0, :, :, 0], cmap='gray')
    plt.show()
```

Original image

Gray image from the first channel of the output
Adding Bias After tf.nn.conv2d()

• To enhance representation power of CNN, it is nice to add bias to the output.

```python
# Build a graph.
x = tf.placeholder(tf.float32, [batch_size, img_height, img_width, img_channel])
w = tf.Variable(tf.random_normal([5, 5, 3, 64], stddev=0.35))
b = tf.Variable(tf.random_normal([64], stddev=0.35))
output = tf.nn.conv2d(x, w, strides=[1, 2, 2, 1], padding='SAME') + b
```

Broadcasting addition
Max Pooling

• Performs the max pooling on the input.
• ‘ksize’
  • The size of the window for each dimension of the input tensor.
  • For 2 x 2 pooling, ksize = [1, 2, 2, 1]
• ‘strides’ and ‘padding’ are same as those in the tf.nn.conv2d().
• We can use convolution of stride 2, instead of using max pooling without significant loss of performance.
  • Check “Springenberg, J. T. et al., (2014).”
Max Pooling Example

- Example of 2 x 2 max pooling.

```python
# Build a graph.
x = tf.placeholder(tf.float32, [batch_size, img_height, img_width, img_channel])
w = tf.Variable(tf.random_normal([5, 5, 3, 64], stddev=0.35))
b = tf.Variable(tf.random_normal([64], stddev=0.35))
c = tf.nn.conv2d(x, w, strides=[1, 1, 1, 1], padding='SAME') + b
output = tf.nn.max_pool(c, [1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
```
Activation Functions

• TensorFlow provides most of the popular activation functions.
  • tf.nn.relu, tf.nn.softmax, tf.nn.sigmoid, tf.nn.elu, ...

• Example of using rectified linear function.

```python
# Build a graph.
x = tf.placeholder(tf.float32, [batch_size, img_height, img_width, img_channel])
w = tf.Variable(tf.random_normal([5, 5, 3, 64], stddev=0.35))
b = tf.Variable(tf.random_normal([64], stddev=0.35))
c = tf.nn.conv2d(x, w, strides=[1, 1, 1, 1], padding='SAME') + b
h = tf.nn.relu(c)
output = tf.nn.max_pool(h, [1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
```
Fully Connected (Dense) Layer

• Fully connected (fc) layer can be implemented by calling `tf.matmul()` function.
  • y = tf.matmul(x, W)

• To compute fc layer after convolution operation, we need to reshape 4-D tensor to 2-D tensor.
  • `[batch_size, height, width, channel]`  
    →  `[batch_size, height*width*channel]`
Fully Connected Layer Example

```python
# Build a graph.
x = tf.placeholder(tf.float32, [batch_size, img_height, img_width, img_channel])
w = tf.Variable(tf.random_normal([5, 5, 3, 8], stddev=0.35))
b = tf.Variable(tf.random_normal([8], stddev=0.35))
c = tf.nn.conv2d(x, w, strides=[1, 1, 1, 1], padding='SAME') + b
c = tf.nn.relu(c)
h = tf.nn.max_pool(c, [1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

h = tf.reshape(h, [batch_size, -1])
fc_w = tf.Variable(tf.random_normal([int(h.get_shape()[1]), 10], stddev=0.35))
fc_b = tf.Variable(tf.random_normal([10], stddev=0.35))
output = tf.matmul(h, fc_w) + fc_b
output = tf.nn.softmax(output)
```
TF Layers: High-level API

- The TensorFlow layers module provides a high-level API that makes it easy to construct a neural network.
- No explicit weight (filter) variable creation.
- Includes activation function in one API.

```python
# Convolutional Layer #1
cov1 = tf.layers.conv2d(
    inputs=input_layer,
    filters=32,
    kernel_size=[5, 5],
    padding="same",
    activation=tf.nn.relu)

# Pooling Layer #1
pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2], strides=2)
```
Other High-level API

- TF Slim
- TF Learn
- Keras (with TensorFlow backend)
- Tensor2Tensor
Loss Functions

• TensorFlow provides various loss functions.
  • tf.nn.softmax_cross_entropy_with_logits, tf.nn.l2_loss, ...
• TF Layers also provides similar functions starting with tf.losses.
• Example of tf.losses.softmax_cross_entropy.

```python
onehot_labels = tf.one_hot(indices=tf.cast(labels, tf.int32), depth=10)
loss = tf.losses.softmax_cross_entropy(
  onehot_labels=onehot_labels, logits=logits)
```

• Full codes are in https://www.tensorflow.org/tutorials/layers
Optimizers

• TensorFlow provides popular optimizers.
  • Adam, AdaGrad, RMSProp, SGD, ...
• Example of plain gradient descent optimizer.
• Parameters are updated when `sess.run(train_op, ...)` is called.

```python
# optimizer
learning_rate = 0.01
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
train_op = optimizer.minimize(loss)
...
sess.run(train_op, {x: batch_x, y: batch_y})
```
Review of the Batch Normalization

• Normalize the activations of the previous layer.

• Advantages
  • Allows much higher learning rates.
  • Can be less careful about initialization.
  • Faster learning.
  • No need for Dropout.

Input: Values of $x$ over a mini-batch: $B = \{x_1...m\}$; Parameters to be learned: $\gamma$, $\beta$
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$$
\mu_B \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \quad \text{// mini-batch mean}
$$

$$
\sigma_B^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2 \quad \text{// mini-batch variance}
$$

$$
\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad \text{// normalize}
$$

$$
y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \quad \text{// scale and shift}
$$
Batch Normalization

- `tf.nn.batch_normalization()` needs bunch of variables and does not support moving statistics, nor inference mode.
- Use `tf.layers.batch_normalization()`
  - Put `training=False`, when inference mode.
  - It supports moving statistics of the mean and variance.
  - ‘momentum’ determines forget rate of the moving statistics.
• `update_ops` should be called to update statistics of batch normalization.

• In inference mode, the values are normalized by moving statistics.

```python
import tensorflow as tf
import numpy as np

x = tf.placeholder(tf.float32, shape=(3, 1), name='x')
tr = tf.placeholder(tf.bool, shape=())

y = tf.layers.batch_normalization(x, axis=1, momentum=0.9, training=tr)
update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)

with tf.Session() as sess:
    tf.global_variables_initializer().run()
    batch_x = np.arange(8).reshape([8, 1]).astype(np.float32)
    for i in range(10):
        [y, _] = sess.run([y, update_ops], {x: batch_x, tr:True})
        if i == 0:
            print(y.flatten())
            print('a='*50)
        _y = sess.run(y, {x: batch_x, tr:False})
        print(_y.flatten())
```

Residual Connection

• A Residual Network is a neural network architecture which solves the problem of vanishing gradients.
  • Residual connection: $y = f(x) + x$

```python
x = tf.placeholder(tf.float32, [batch_size, img_height, img_width, img_channel])
h = tf.layers.conv2d(x, filters=64, kernel_size=[11, 11], strides=[4, 4], padding="SAME")

# Residual connection
h2 = tf.layers.conv2d(h, filters=64, kernel_size=[3, 3], strides=[1, 1], padding="SAME")
h3 = tf.layers.conv2d(h2, filters=64, kernel_size=[3, 3], strides=[1, 1], padding="SAME")
y = h3 + h
```
Transposed Convolution (Deconvolution)

• The need for transposed convolutions generally arises from the desire to use a transformation going in the opposite direction of a normal convolution.

• `tf.layers.conv2d_transpose()`
Load Pre-trained Models

• There are popular network architectures in TF Slim
  • [https://github.com/tensorflow/models/tree/master/slim](https://github.com/tensorflow/models/tree/master/slim)
  • Inception V1-V4
  • Inception-ResNet-v2
  • ResNet 50/101/152
  • VGG 16/19
  • MobileNet
Thank You
References

• https://www.tensorflow.org
• https://www.slideshare.net/JenAman/large-scale-deep-learning-with-tensorflow
• https://www.slideshare.net/AndrewBabiy2/tensorflow-example-for-ai-ukraine2016
• http://download.tensorflow.org/paper/whitepaper2015.pdf