Tensorflow and Pytorch for Speech-to-image Retrieval

Liming Wang
Oct. 29th
Outline

• Speech-to-Image Retrieval
  • Similarity-based Retrieval models
  • Training

• TFLearn implementation for HG-16
  • Overview of Tensorflow APIs
  • Implementation of HG-16

• Pytorch implementation
  • Overview of Pytorch APIs
  • Implementation of HG-16
Speech-to-Image Retrieval

- **Goal**: Find relevant images given a spoken description without any intermediate supervisions.

- **Idea**:
  - **Generative model**: generate image directly from speech?
  - **Retrieval-based model**: Map the speech and image into a same feature space, called the ‘semantic space’?

Cat image

‘a brown cat is lying on the floor’
General Retrieval-based Model

Speech Encoder $a$

Image Encoder $v$

Speech Features $X_1, ..., X_n$

Image Features $I_1, ..., I_n$

Similarity Score
Harwath-Glass (2016) Model

- Retrieval-based model: 30.9% recall@10

Chrupala (2017) Model

- Retrieval-based model: same idea as HG-16 but using recurrent highway network as speech encoder

**Diagram:**
- 1024-d Image Vector
  - Affine Transform
  - VGG 16 (No softmax)
- 1024-d Speech Vector
  - Attention Layer

**Text:**
- ‘a brown cat is lying on the floor’
VGG-16 Feature For Image
Retrieval

- Compute Similarity Score:

\[ S_{ij} = \max\left( \frac{a_i^\top v_j}{\|a_i\| \|v_j\|}, 0 \right) \]

- Find the image with the highest score for a given caption:

\[ \hat{I} = \arg \max_{I_i} S_{ij}(I_i, X_j) \]
Training

• Max-margin Loss:

\[ C(\theta) = \sum_{I=1}^{B} \sum_{j} \max(S_{ij} - S_{ii} + 1, 0) + \max(S_{ji} - S_{ii} + 1, 0) \]

• Sampled Max-margin Loss:

\[ C(\theta) = \sum_{I=1}^{B} \max(S_{i}^{sp} - S_{ii} + 1, 0) + \max(S_{i}^{im} - S_{ii} + 1, 0) \]

• Negative Log Likelihood: Let M(i) be the set of matched image indices,

\[ C(\theta) = \sum_{I=1}^{B} \sum_{j \in M(i)} \log \frac{\exp(S_{ij})}{\sum_{j'} \exp(S_{ij'})} \]
Tensorflow
Implementation of HG-16
Outline

• Computational Graph Basics
• Optimization
• Session
• Visualization of Learning
• Scoped Variable
Computation Graph (CG) of Tensorflow

• **Definition**: A graph with edges as symbolic variables known as *tensors* and nodes as the *operations* of tensors

• `<graph>.as_default()`: Choose a graph as default

• `reset_default_graph()`: Clear all the existing tensors in the graph

Based on: [https://www.tensorflow.org/guide/low_level_intro](https://www.tensorflow.org/guide/low_level_intro)
Tensors

- **Tensor**: Edges that hold numerical values like input features, model parameters, etc.

- **Variable**: tensors with data that holds variable multidimensional data of a single type

- `get_shape()`: Returns a tuple of the tensor size

- Need initialization: e.g., `tf.random_normal()` or `tf.zeros()` or `tf.assign()`

- **Placeholder**: tensors with data that holds the input data; fixed during training

- **None** size: Decided after the data are fed into the graph

Based on: [https://www.tensorflow.org/guide/low_level_intro](https://www.tensorflow.org/guide/low_level_intro)
Operations

- **Operation**: Nodes of the CG that perform symbolic operation on the tensors

- Special cases:
  - **Initializer**: A special op for initialize variables, e.g. `global_variable_initializer()`
  - `tf.nn.<function_name>`: Contain all the essential functions and tensor operations for neural net. More see [https://www.tensorflow.org/api_docs/python/tf/nn](https://www.tensorflow.org/api_docs/python/tf/nn)
  - `minimize()`: Optimizer operation for computing gradients

Based on: [https://www.tensorflow.org/guide/low_level_intro](https://www.tensorflow.org/guide/low_level_intro)
TFLearn: A Lightweight High-Level API on TF

- **data_preprocessing.py & data_augmentation.py**

- **layers/**: Contains all the essential type of layers in a neural network. More about it later. See also [http://tflearn.org/layers/](http://tflearn.org/layers/)

- **Estimators.py**: Contains a high-level function called `regression()` for optimization

- **Objectives.py**: Contains an extensible list of loss function classes

- **metrics.py**: Contains an extensible list of evaluation metric

- **models/**: Contains two main type of model: *DNN* vs *SequenceGenerator*, handles all the training, evaluating and model saving/loading.
Harwath-Glass (2016) Model

- Retrieval-based model: 30.9% recall@10

Data Preprocessing Example: Mean-Variance Normalization

```python
def _samplewise_zero_center(self, batch):
    for i in range(len(batch)):
        batch[i] -= np.mean(batch[i], axis=0)
    return batch

def _samplewise_stdnorm(self, batch):
    for i in range(len(batch)):
        batch[i] /= (np.std(batch[i], axis=0) + EPSILON)
    return batch
```

• Under the hood (data_preprocessing.py):
Convolutional Layers

In `tflearn/layers/conv.py`:

```python
self.conv1 = tflearn.conv_2d(sp_feat, 64, [1, 5],
   padding='valid',
   activation='relu',
   regularizer='L2',
   scope=self.scope+''/conv1'',
   reuse=reuse)
```

```python
self.pool1 = tflearn.max_pool_2d(self.conv1, [1, 2],
   name='pool1')
```

- In `tflearn/layers/conv.py`:

```python
def conv_2d(incoming, nb_filter, filter_size, strides=1, padding='same',
activation='linear', bias=True, weights_init='uniform_scaling',
bias_init='zeros', regularizer=None, weight_decay=0.001,
trainable=True, restore=True, reuse=False, scope=None,
name="Conv2D"):
```
HG-16 (TFLearn)

• Speech Encoder:

```python
self.conv1 = tflearn.conv_2d(sp_feat, 64, [1, 5],
                          padding='valid',
                          activation='relu',
                          regularizer='L2',
                          scope=self.scope+'/conv1',
                          reuse=reuse)

self.pool1 = tflearn.max_pool_2d(self.conv1, [1, 2],
                               name='pool1')

self.conv2 = tflearn.conv_2d(self.pool1, 64, [1, 5],
                          padding='valid',
                          activation='relu',
                          regularizer='L2',
                          scope=self.scope+'/conv2',
                          reuse=reuse)

self.pool2 = tflearn.max_pool_2d(self.conv1, [1, 2],
                               name='pool2')

a_vec = tflearn.fully_connected(self.pool2, 512,
                               regularizer='L2',
                               scope=self.scope+'/fc1',
                               reuse=reuse)
```
Fully Connected Layers

In `tflearn/layers/core.py`:

```python
a_vec = tflearn.fully_connected(self.pool2, 512,
                          regularizer='L2',
                          scope=self.scope+:'/fc1',
                          reuse=reuse)
```

- In `tflearn/layers/core.py`:

```python
def fully_connected(incoming, n_units, activation='linear', bias=True,
                    weights_init='truncated_normal', bias_init='zeros',
                    regularizer=None, weight_decay=0.001, trainable=True,
                    restore=True, reuse=False, scope=None,
                    name="FullyConnected"):
```

Fully-Connected Layer in layers/core.py

```python
inference = incoming
# If input is not 2d, flatten it.
if len(input_shape) > 2:
    inference = tf.reshape(inference, [-1, n_inputs])

inference = tf.matmul(inference, W)
if b: inference = tf.nn.bias_add(inference, b)
if activation:
    if isinstance(activation, str):
        inference = activations.get(activation)(inference)
    elif hasattr(activation, '__call__'):
        inference = activation(inference)
    else:
        raise ValueError("Invalid Activation.")
```

- The 3rd line is useful, e.g. when passing outputs from conv layer as inputs.
HG-16 (TFLearn)

- Image Encoder:

```python
v_vec = tflearn.fully_connected(self.vgg_feat, 512,
                               regularizer='L2',
                               scope=self.scope+'/fc2',
                               reuse=reuse)
```

- Similarity Score:

```python
def similarity(a_vec, v_vec):
    a_norm = tf.nn.l2_normalize(a_vec, dim=1)
    v_norm = tf.nn.l2_normalize(v_vec, dim=1)
    return tf.nn.relu(tf.matmul(a_norm, v_norm, transpose_b=True))
```
Outline

• Computational Graph Basics
• Optimization
• Session
• Visualization of Learning
• Scoped Variable
Optimization on CG

- **Optimizer**: Implement gradient-based optimization algorithms like SGD, ADAM, RMSProp, etc.

  - `tf.train.<opt_name>(<learning_rate>)`: Initialize optimizer, e.g. `tf.train.AdamOptimizer(1e-5)`

  - `<optimizer_name>.minimize(<loss>)`: return an operation to update the variables on the graph. Need e.g. `sess.run(<minimize_op>, feed_dict={...}, var_list={...})`

  - Internally compute the gradients of the loss w.r.t each variable in var_list and update each of them

Based on: [https://www.tensorflow.org/guide/low_level_intro](https://www.tensorflow.org/guide/low_level_intro)
Regression Layer

```python
self.reg = tflearn.regression(net, optimizer='adam',
                              metric=accuracy, learning_rate=1e-5,
                              loss='hgl6_hinge_loss')
self.model = tflearn.DNN(self.reg)
```

- Need to modify objectives.py:

```python
def hgl6_hinge_loss(y_pred, y_true):
    with tf.name_scope(None):
        floss = tf.reduce_mean(tf.nn.relu(y_pred - tf.diag(y_pred) + 1))
        bloss = tf.reduce_mean(tf.nn.relu(tf.transpose(y_pred) - tf.diag(y_pred) + 1))
    return floss + bloss
```

- layers/estimator.py:

```python
def regression(incoming, placeholder='default', optimizer='adam',
              loss='categorical_crossentropy', metric='default',
              learning_rate=0.001, dtype=tf.float32, batch_size=64,
              shuffle_batches=True, to_one_hot=False, n_classes=None,
              trainable_vars=None, restore=True, op_name=None,
              validation_monitors=None, validation_batch_size=None, name=None):
```
Training

• Save/Load the trained model:
  • `model.save(<name>)`
  • `model.load(<name>)`

• In models/dnn.py:
  ```python
def fit(self, X_inputs, Y_targets, n_epoch=10, validation_set=None, show_metric=False, batch_size=None, shuffle=None, snapshot_epoch=True, snapshot_step=None, excl_trainops=None, validation_batch_size=None, run_id=None, callbacks=[]):
```
Retrieval

- For simple classifier: `DNN.evaluate(<x>, <y>, <metric>)`
- `recall_at_k()`:

```python
def test(sp_feat, vgg_feat):
    self.s = self.model.predict([[sp_feat, vgg_feat]])

    max_ids, recall = recall_at_k(self.s, k=1)
    print("Test recall is \$4f\$ \%(recall))

def recall_at_k(s, k=1):
    n = s.shape[0]
    max_ids = np.argsort(s, 1)[:, :k]
    recall = 0
    for i in range(n):
        for j in range(k):
            recall += (i == max_ids[i][j])

    return max_ids, recall / n
```
Outline

- Computational Graph Basics
- Optimization
  - Session
- Visualization of Learning
- Scoped Variable
Session

- **Session**: “encapsulates the state of the TensorFlow runtime, and runs TensorFlow operations”. *Everything in tensorflow needs session.run() to work.*

- `run(<variable name>, feed_dict={<placeholder name: value>})`:
  - Evaluate an operation:
    - Initialization: `sess.run(global_variable_initializer())`
    - Update training: `sess.run(<minimize_op>)`
  - Evaluate a variable: `sess.run(a, feed_dict={…})`

Based on: [https://www.tensorflow.org/guide/low_level_intro](https://www.tensorflow.org/guide/low_level_intro)
Load/Save Pretrained Model

• Maintain a list of parameters of the model, then define:

```python
def load_weights(self, weight_file, sess):
    weights = np.load(weight_file)
    keys = sorted(weights.keys())
    for i, k in enumerate(keys):
        print(i, k, np.shape(weights[k]))
        sess.run(self.parameters[i].assign(weights[k]))
```

• Similarly, for saving model:

```python
def save_weights(self, weight_file):
    lscnn.save_weights(weight_file)
    weight_file [string]: save weights in weight_file + ".npz"
    saveable = [ self.sess.run(v) for v in self.pmtrs ]
    np.savez(weight_file, saveable)
```

• Alternative: `tf.train.Saver()`, `tflearn.DNN.save()`

Outline

• Computational Graph Basics
• Optimization
• Session
• Visualization of Learning
• Scoped Variable
Visualizing Learning with Tensorboard

• **Why**: Display Rate of Convergence of loss/metric, visualization of the CG, (gradients, sparsity) etc.

• **How**:
  
  • **Save tensorboard file**:  
    
    \[DNN(\ldots, \text{tensorboard\_dir}=<\text{name}>, \text{tensorboard\_verbose}=3)\]

  • **Invoke tensorboard**:
    
    `tensorboard --logdir=<name>`
Outline

• Computational Graph Basics
• Optimization
• Session
• Visualization of Learning
• Scoped Variable
Scope: Reason why TF can be tricky

- Default scope = for all tensors. What if two identical CNN appears on the CG?

- Prevent name collision by adding a “prefix” (scope):

  ```python
  with tf.scope('cnn1/ '):
      cnn1 = CNN()

  with tf.scope('cnn2/ '):
      cnn2 = CNN()
  ```

- Sharing weight: `tf.scope('cnn1', reuse=True)`

- `tf.GraphKeys()`: Store all the variable names in a way according to: [https://www.tensorflow.org/api_docs/python/tf/GraphKeys](https://www.tensorflow.org/api_docs/python/tf/GraphKeys)
Scoped Variable Example: layers/conv.py

```python
inference = tf.nn.conv2d(incoming, W, strides, padding)
if b: inference = tf.nn.bias_add(inference, b)

if activation:
    if isinstance(activation, str):
        inference = activations.get(activation)(inference)
    elif hasattr(activation, '__call__'):
        inference = activation(inference)
    else:
        raise ValueError("Invalid Activation.")

# Track activations.
    tf.add_to_collection(tf.GraphKeys.ACTIVATIONS, inference)

# Add attributes to Tensor to easy access weights.
    inference.scope = scope
    inference.W = W
    inference.b = b

# Track output tensor.
    tf.add_to_collection(tf.GraphKeys.LAYER_TENSOR + '/' + name, inference)
```
Scoped Variable Example 2: VGG 16

```python
class vgg16:
    def __init__(self, imgs, weights=None, sess=None):
        self.imgs = imgs
        self.conv_layers()
        self.fc_layers()
        self.probs = tf.nn.softmax(self.fc3)
        if weights is not None and sess is not None:
            self.load_weights(weights, sess)

    def conv_layers(self):
        self.parameters = []

        # zero-mean input
        with tf.name_scope('preprocess') as scope:
            mean = tf.constant([123.68, 116.779, 103.939], dtype=tf.float32, shape=[1, 1, 1, 3], name='img_mean')
            images = self.imgs - mean

        # conv1_1
        with tf.name_scope('conv1_1') as scope:
            kernel = tf.Variable(tf.truncated_normal([3, 3, 3, 64], dtype=tf.float32,
                                                     stddev=1e-1), name='weights')
            conv = tf.nn.conv2d(images, kernel, [1, 1, 1, 1], padding='SAME')
            biases = tf.Variable(tf.constant(0.0, shape=[64], dtype=tf.float32),
                                 trainable=True, name='biases')
            out = tf.nn.bias_add(conv, biases)
            self.conv1_1 = tf.nn.relu(out, name=scope)
            self.parameters += [kernel, biases]

        # conv1_2
        with tf.name_scope('conv1_2') as scope:
            kernel = tf.Variable(tf.truncated_normal([3, 3, 64, 64], dtype=tf.float32,
                                                     stddev=1e-1), name='weights')
            conv = tf.nn.conv2d(self.conv1_1, kernel, [1, 1, 1, 1], padding='SAME')
            biases = tf.Variable(tf.constant(0.0, shape=[64], dtype=tf.float32),
                                 trainable=True, name='biases')
            out = tf.nn.bias_add(conv, biases)
            self.conv1_2 = tf.nn.relu(out, name=scope)
            self.parameters += [kernel, biases]

        # pool1
        self.pool1 = tf.nn.max_pool(self.conv1_2,
                                    ksize=[1, 2, 2, 1],
                                    `Code from: https://www.cs.toronto.edu/~frossard/post/vgg16/`
Summary

- Define a neural network input with `tf.Placeholder()` and network parameters with `tf.Variable()`; always use `tf.scope()` to avoid name collision and `reuse=True` option to share weights.

- Optimize with `tf.train.<optimizer>(<defaults>).minimize(loss)`.

- Start a `tf.Session();` use `.run()` method for operations like initialization and optimization.

- Use `tflearn.DNN()` to define and manage customized models; hack `tflearn/objectives.py` to define customized loss function.

- Visualize learning with: tensorboard —logdir=<name>
Pytorch Implementation of HG-16
Pytorch vs. TFLearn

- **Speed**: Pytorch is slower on GPU due to its dynamic CG

- **API**: Pytorch (version >0.2.0) is simpler:
  - More diverse in training scheme thanks to dynamic CG;
  - Pre-trained models (see https://modelzoo.co);
  - Allow direct modification on gradients;
  - Sharing operations, broadcasting rules with numpy

- **Model Performance**: Roughly the same for sp2im retrieval
Packages

- **torch**: Contains functions for tensor operations and conversion between data types

- **torch.Tensor**: A class for multidimensional matrix of single type with methods for various tensor operations

- **torch.nn**: Contains an abstract class (nn.Module) to handle general neural network models; various type of neural net layers and functionals; parameter initialization

- **torch.optim**: Various types of optimizers

- **torch.autograd**: Performing and monitoring gradient computations

- **torch.utils**: For data summary, data loading and sampling and pre-trained models
Tensor in Pytorch

- **Name & scope:** Like in TF, but exists only internally
- **Leaf Nodes:** Network parameters in the CG
- **Initialize:**
  - `torch.Tensor(<numpy array>)`
  - `torch.Tensor(<nn.initializer>)`
  - `torch.Tensor(<size>); torch.from_numpy()`
- **Evaluate:** `.numpy()`
- **Size:** `.size()`

For more info: [https://pytorch.org/docs/](https://pytorch.org/docs/)
Autograd

- **Gradients**: Unlike TF, not assigned to every tensor automatically during optimization to freeze certain parameters during training

- **Automatic Differentiation**:
  - `autograd.Variable(<tensor>, requires_grad=True)`: Wrapper class for a tensor that needs gradients; set `requires_grad=False` to freeze the tensor
  - `<variable>.backward()`: Compute the gradients of the variable w.r.t the leaf nodes using back-propagation
  - Note: Variable class deprecated for torch > 1.0.0, add `requires_grad=True` in torch.Tensor for newer version

For more info: [https://pytorch.org/docs/](https://pytorch.org/docs/)
Optimizer

- **Initialize**: `torch.optim.<optim>(params, defaults)`
  - “params”: e.g. come from `<module>.parameters()`; more generally, a list of parameters that you want to train
  - “defaults”: things like learning rate and weight decay

- **Optimize**: `.step()`
  - `.zero_grads()`: Torch optimizers keep gradients of the previous steps for things like Momentum; zeros them out at the beginning of each iteration

For more info: [https://pytorch.org/docs/](https://pytorch.org/docs/)
**nn.Module**

- **Purpose**: Define customized model as subclass by *class* `<class_name>*(nn.Module)`

- `forward()`: Need overwritten by all subclasses; define the forward pass of the network

- `parameters()`: Returns all the parameters in the module; where scope and name of a tensor is in

- `State_dict()`: Returns a mapping from parameter to its value; save it using `torch.save()`

- `load_state_dict()`: Load a pre-trained model given its state_dict

For more info: [https://pytorch.org/docs/](https://pytorch.org/docs/)
Retriever as nn.Module (2)

class LSCNN2(nn.Module):
    def __init__(self, n_word=1):
        super(LSCNN2, self).__init__()
        self.splen = 100
        self.conv1 = nn.Conv1d(40, 64, 5)
        self.conv2 = nn.Conv1d(64, 64, 5)
        self.pool = nn.MaxPool1d(2)
        self.hidlen = 64 * int((int((self.splen - 4)/2)-4)/2)
        self.fc_a = nn.Linear(self.hidlen, 512)
        self.fc_v = nn.Linear(4096, 512)

    def forward(self, a_feat, v_feat):
        self.a_vec = self.sp_enc(a_feat)
        self.v_vec = self.im_enc(v_feat)
        s = self.similarity(self.a_vec, self.v_vec)
        return s
Encoders

def sp_enc(self, x):
    x = F.relu(self.conv1(x))
    x = self.pool(x)
    x = F.relu(self.conv2(x))
    x = self.pool(x)
    x = x.view(-1, self.hidden)
    x = F.relu(self.fc_a(x))
    return x

def im_enc(self, x):
    return self.fc_v(x)

<tensor>.view(): Flatten the x into 1-D array
Similarity Score

def similarity(self, a, v, mode="relu"):
    # The t() is unnecessary for newer version
    la = a.norm(2, 1).expand(a.size()[1], a.size()[0])
    lv = v.norm(2, 1).expand(v.size()[1], v.size()[0])
    a_norm = a / la.t()
    v_norm = v / lv.t()
    s = a_norm.mm(v_norm.t())
    if mode == "relu":
        return F.relu(s)
    elif mode == "coscos":
        return s ** 2

<tensor>.expand(): Expand x into a given size
Loss and Training

def hinge_loss(self, s):
    s_diag = s.diag().expand(s.size()[0], s.size()[1])
    return torch.mean(F.relu(s - s_diag + 1) + F.relu(s.t() - s_diag + 1))

def sample_hinge_loss(s, s_neg_a, s_neg_v):
    floss = F.relu(s_neg_a.diag() - s.diag() + 1)
    bloss = F.relu(s_neg_v.diag() - s.diag() + 1)
    return torch.mean(floss + bloss)

def train_step(self, optimizer, a_feat, v_feat):
    self.s = self.forward(a_feat, v_feat)
    self.loss = self.hinge_loss(self.s)
    self.loss.backward()
    optimizer.step()
    return self.loss
for t in range(n_epoch):
    rand_ids = [np.argsort(np.random.normal(size=(ntrs[i],))) for i in range(nclass)]
    for j in range(n_batch):
        optimizer.zero_grad()
        a_batch, v_batch = class_based_sampling(a_feats_tr,
                                                v_feats_tr,
                                                batch_size,
                                                rand_ids, j)

        a_batch = Variable(Tensor(a_batch))
        v_batch = Variable(Tensor(v_batch))

        lscnn.train_step(optimizer, a_batch, v_batch)
        running_loss += lscnn.loss.data.numpy()
        nstep += 1

        if (nstep + 1) % 10 == 0:
            _, recall = recall_at_k(lscnn.s.data.numpy())
            print("Epoch %d batch %d : training loss: %4f, recall at 1: %4f" %
                  (t, j, running_loss/nstep, recall))

        running_loss = 0
        nstep = 0
Summary

In Pytorch:

- Define a network as a subclass of `nn.Module`; Use functions in packages like torch.nn and torch.Tensor

- To train a network:
  1. Use `torch.autograd.Variable` as network parameters
  2. Compute the gradients with `<variable>.backward()`
  3. Optimize using `torch.optim.<optimizer>.zero_grads()` before each iteration and `torch.optim.<optimizer>.step()`

- Loading/Save a network by loading/saving its state dictionary
Conclusion

• Speech-to-image retrieval: Finding the mapping between two embedding space encoded from audio and image

• TFLearn: Convenient API for tensorflow, suitable for speech-to-image retrieval

• Pytorch: A more flexible framework to train speech-to-image retrieval models