Computer Vision 2: Exercise 4 (21.5.2019)

Transfer learning, saving and loading model parameters

We will learn about

- transfer learning
- saving and loading models to resume training
- (extra task) `tf.data`

Transfer learning

Transfer learning refers to taking trained parameters from a network trained on a source task, and using them to initialize the parameters of a network solving another target task. This can be very powerful, as it allows one to use parameters learned on huge datasets with millions of images, and apply them to a problem where one might only have a very limited amount of data. Fine tuning is a related idea, that means learning or tuning the transferred weights based on data from the target task.

The technical mechanism to implement transfer learning is simple: just obtain the parameters of the source network, and use them as initial values for the parameters of the target network.

VGG-16. The VGG-16 network is one of the most popular convolutional neural networks (CNN) today. We investigate the VGG-16D variant, whose structure is shown in Table 1. The left column shows the blocks, or larger structural elements in the network. The center column shows the individual layers. The right column indicates for the first block the shapes of the parameter arrays, which describe the weights of the convolution kernels and the bias vectors. Note that the values are only filled in for the block `conv1`. All max pooling layers share the same parameters as `pool1`. Note that the pooling layer does not have any learnable parameters.

- Download the file `vgg16-conv-weights.npz` and examine its contents using the following code snippet.

```python
weights = np.load('vgg16-conv-weights.npz')
for k in weights.keys():
  print(k + ' shape: {:s}'.format(weights[k].shape))
```

- Make sure you know how these values relate to the layers shown in Table 1.
- Notice that you can access the actual parameter array for example like this: `weights['conv1_1_W']`

Note that in the file `vgg16-fc-weights.npz` there are the weights for the fully connected (FC) layers of VGG-16D. We will not need them in this exercise, but you may find it interesting nonetheless.

Transfer learning We will take the blocks from conv1 through to pool2, initialize their weights to their trained VGG16 values, and add our own fully connected layer which we will learn the parameters for. Use the CIFAR-10 dataset from the previous exercise.

1. Create placeholders for the inputs and labels as in the previous exercise.
2. VGG16 expects an input where the mean image RGB value of the ImageNet dataset is subtracted. Add a preprocessing block that does this as follows:

Table 1: VGG-16D.

Block	Layer	Parameters
conv1	conv3-64 conv3-64	weights: [3,3,3,64], bias: [64] weights: [3,3,64,64], bias: [64]
pool1	maxpool	2 x 2, stride 2
conv2	conv3-128 conv3-128	⋮
pool2	maxpool	
conv3	conv3-256 conv3-256 conv3-256	
pool3	maxpool	
conv4	conv3-512 conv3-512 conv3-512	
pool4	maxpool	
conv5	conv3-512 conv3-512 conv3-512	
pool5	maxpool	
fc6	FC-4096	
fc7	FC-4096	
fc8	FC-1000	
	softmax	

```
with tf.name_scope('preprocess') as scope:
  imgs = tf.image.convert_image_dtype(img_placeholder, tf.float32
   ) * 255.0
  mean = tf.constant([123.68, 116.779, 103.939], dtype=tf.float32
   , shape=[1, 1, 1, 3], name='img_mean')
  imgs_normalized = imgs - mean
```

We use `tf.name_scope` to separate the modules. This will also result in clearer Tensorboard visualizations, where scopes can be expanded for close-up views and closed to get an overall picture.

3. Implement all of the layers from conv1 up to and including pool3, and initialize their parameters to their trained VGG16 values. Create a name scope for each layer. Set the variables to be un-trainable by choosing `trainable=False` in the variable constructor, as we do not want to modify them any more. Example:

```
with tf.name_scope('conv1_1') as scope:
  kernel = tf.Variable(initial_value=weights['conv1_1_W'],
   trainable=False, name="weights")
  biases = tf.Variable(initial_value=weights['conv1_1_b'],
   trainable=False, name="biases")
  conv = tf.nn.conv2d(imgs_normalized, kernel, [1, 1, 1, 1],
   padding='SAME')
  out = tf.nn.bias_add(conv, biases)
  act = tf.nn.relu(out, name=scope)
```

4. Add a module that will take the output of `pool3`, flatten it, and then feed it through a fully connected (dense) layer with 10 output units. Use a `tf.variable_scope` as follows:

```
with tf.variable_scope('fc') as scope:
  flat = tf.layers.flatten(pool3)
  logits = tf.layers.dense(flat, units=10)
```

5. Similarly as in the previous exercise, add a cross entropy loss function, an optimizer, and accuracy calculation and the related summaries.

6. Using a large batch size and low learning rate (suggestion: start with batch size 64 and learning rate 10^{-6}), train the network in the same way as in the previous exercise. How does the accuracy compare?

A final remark: the VGG-16 network is trained on ImageNet data, which are 224-by-224 images. But the CIFAR-10 data are 32-by-32 images, there is quite a large mismatch in the scale of the images which will effect how successful transfer learning can be expected to be. In a real application, such factors should be taken into account. This exercise merely illustrates the technical method to apply transfer learning in TF.

Saving and restoring model parameters

When training a CNN, it is useful to be able to save the learned parameters from time to time. Good parameter sets can be recovered later, or training can be resumed after a break. In Tensorflow, we can do this by saving and restoring `tf.Variables`. Here is an example of saving all variables in a model.

```
# First, construct all variables.
# Now add ops to save and restore all the variables.
saver = tf.train.Saver()
with tf.Session() as sess:
  # Train the network here...
  # Then, whenever you want, save the variables into a file
  save_path = saver.save(sess, "/my/path/my-model", global_step=b)
```

Here is an example of restoring all variables in a model.

```
# First, construct all variables as before.
# Now add ops to save and restore all the variables.
saver = tf.train.Saver()
with tf.Session() as sess:
  # Read variables from disk for model saved with global_step=50
  saver.restore(sess, "/my/path/my-model-50")
```

`Saver` only restores all variables **already defined in your model**. This means that to read the checkpoint, we must also **first construct the model with all the variables**. This is not a problem since we know how to construct the computational graph for our model.

1. Modify your code so that it stores a checkpoint.

2. After you have saved a checkpoint, modify your code so that you load the checkpoint and resume training from it.

By default, `Saver` will keep 5 of the most recent checkpoints stored on disk. If you want to keep more checkpoints, you can use the `max_to_keep` argument of the constructor.

Suggestion: save a checkpoint when you run a validation step. But, don't save checkpoints too often. This wastes disk space.

Extra task: `tf.data`

If you finished the other tasks, you can work on this task.

Up to now, we have used `tf.placeholder` to feed in input data. This has a number of inefficiencies, though. The most severe is that data is read from memory only when it is requested, no pre-loading is done. It can improve training speed immensely if we use the CPU cores to preprocess and prepare the data, so that it is immediately ready to be transferred to the GPU when new data is needed. `tf.data` provides an efficient mechanism to do this type of preprocessing via `tf.data.Dataset`. It also makes it easy to modify input pipelines, adding data augmentation or other operations.

1. Study the guide at `https://www.tensorflow.org/guide/datasets` to learn about the basics of `Dataset`
2. Remove the input placeholders from your code, and replace them instead by a dataset + iterator. Here is a snippet to get you started:

```
ds = tf.data.Dataset.from_tensor_slices((train_data, train_label)
ds = ds.batch(batch_size)
iterator = ds.make_one_shot_iterator()
next_images, next_labels = iterator.get_next()
# use next_images and next_labels in your network definition
   instead of placeholders
```

3. The code above will allow you to train for one epoch only. Explore the other opportunities provided by `Dataset`: find out how you can repeat data, how you can shuffle data, and what it means to `prefetch`. The API documentation can be found at `https://www.tensorflow.org/api_docs/python/tf/data/Dataset`