Computer Vision 2: Exercise 3 (7.5.2019)

Training a convolutional neural network for classification

We will train a convolutional neural network (CNN) for image classification. We will learn about

- Stochastic gradient descent,
- Training, validation, and testing data,
- Setting up convolutional layers and fully connected layers in Tensorflow.

Background: Stochastic gradient descent (SGD)

Recall that a neural network can be viewed as a function f that takes an input sample x_i and, depending on the parameters θ, produces a prediction $\hat{y}_i = f(x_i; \theta)$. For training, we are given a dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$ of n inputs x_i and desired outputs y_i that we should match. How well the network's predictions match the groudn truth are determined by a loss function $L(\mathcal{D}, \theta)$ that depends on the data and the parameters. Typically, we also have a per-sample loss function $L_i(y_i, \hat{y}_i)$ that measures how good the prediction is for the ith sample[1]. Then, we define the overall loss by averaging:

$$L(\mathcal{D}, \theta) = \frac{1}{n} \sum_{i=1}^{n} L_i(y_i, f(x_i; \theta)). \tag{1}$$

Gradient descent updates the parameters at iteration k by

$$\theta_{k+1} = \theta_k - \alpha \left. \frac{\partial}{\partial \theta} L(\mathcal{D}; \theta) \right|_{\theta_k},$$

where $\alpha > 0$ is the learning rate and $\frac{\partial}{\partial \theta} L(\mathcal{D}; \theta)\big|_{\theta_k}$ is the partial derivative evaluated at θ_k.

Sometimes our dataset can have a very large number of samples, i.e., n is very large. It may be impractical to calculate the loss for all of the samples as in Eq. (1). This is the motivation for using *stochastic* gradient descent. It is called stochastic because it evaluates the partial derivative of the loss by using a random sample of training data instead of the entire dataset. In practice, the dataset $\mathcal{D}$ is split into non-overallping *batches* of data, sometimes also called mini-batches.

More formally, let $\mathcal{I} = \{1, 2, \ldots, n\}$ denote the set of indices of all samples in the entire data set $\mathcal{D}$. Formally, the batch index sets $\mathcal{B}_j$ are subsets that form a partition of $\mathcal{I}$ – that is, the union $\bigcup_j \mathcal{B}_j = \mathcal{I}$ covers the index set entirely, furthermore, for $j \neq h$, $\mathcal{B}_j \cap \mathcal{B}_h = \emptyset$, so each sample is in exactly one batch, and each $\mathcal{B}_j$ is non-empty. Figure 1 illustrates what this means.

Given a batch $\mathcal{B}_j$, stochastic gradient descent approximates the loss function by

$$L(\mathcal{D}, \theta) \approx \frac{1}{|\mathcal{B}_j|} \sum_{i \in \mathcal{B}_j} L_i(y_i, f(x_i; \theta)),$$

where $|\mathcal{B}_j|$ is the cardinality, i.e., the number of elements of the set $\mathcal{B}_j$. The per-sample losses are averaged only over the samples in the current batch. The update rule is the same as above. Once we have applied the update rule once on every batch, we say that we have performed one *epoch* of training, indicating we have processed the entire training data set once. Typically, networks are trained for multiple epochs.

[1] For example, in the previous exercise we used $L_i(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2$ to get the squared error.

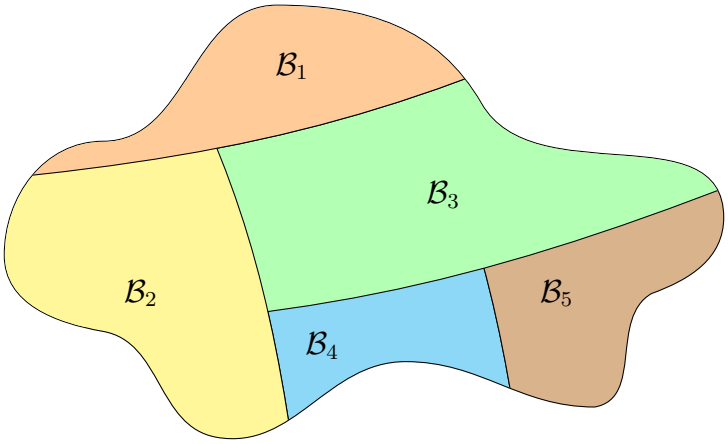

Figure 1: An example of an index set $\mathcal{I}$ partitioned into five non-empty batches $\mathcal{B}_j$. The union of all the batches forms the entire index set, and none of the batches have overlap with any other batch.

Training validation, and testing data

A CNN with a sufficiently large capacity can overfit the training data reaching a loss of 0. However, it does not mean that it will then work perfectly with unseen data. A portion of the data is often reserved for validation and not used in training the CNN. Monitoring the loss on this *validation dataset* provides information on the *generalization capability* of the network, or, how well the CNN can be expected to work on unseen data from the same domain.

We can split a partion of our training dataset $\mathcal{D}$ to act as a validation set. The validation set is not used for training by SGD, but we periodically evaluate the average loss on it to see if the CNN has started to overfit our training data (low loss on training data, increasing loss on validation data).

Sometimes, a test set is also used. This set is supposed to be not used *at all* during training, not even for validation. The purpose of the test set is to evaluate the performance of the network on it *once* after all training has been completed. This is done to avoid implicit overfitting that using a validation set can have: even if you don't train on the validation set, the performance you observe on it can influence your design choices and lead to overfitting.

CNN for classification in Tensorflow

We will train a small CNN for classification of CIFAR-10 data of objects of 10 different classes.

Dataset: CIFAR-10 Follow these steps to prepare the dataset.

- Access the web page at `https://www.cs.toronto.edu/~kriz/cifar.html` and download the Python version of CIFAR-10.
- Download from Moodle the code `ex03_cifar.py` for reading the data. Examine the code to check you understand what the array contents are. **If you get errors running the code,** you may need to use the `python3` version of unpickling the data. Check the CIFAR-10 website at "Dataset layout".

Setting up the network.

1. Set a placeholder with shape `[None, 32, 32, 3]` with datatype `tf.uint8` for the input batch of images. The first dimension denotes the batch size. Note we can use `None` to indicate a dimension can vary at runtime – this is sometimes possible!
2. Set a `tf.int64` placeholder with shape `[None,]` for the labels.

Table 1: CNN specification.

Layer name	Layer specification
conv1_1	3 x 3 kernel, 32 filters, ReLU
conv1_2	3 x 3 kernel, 32 filters, ReLU
pool1	max pooling, size 2 x 2, stride 2
conv2_1	3 x 3 kernel, 64 filters, ReLU
conv2_2	3 x 3 kernel, 64 filters, ReLU
pool2	max pooling, size 2 x 2, stride 2
flatten	flatten previous layer output
fc1	fully connected, 10 outputs, linear activation

3. Preprocessing: convert images to `tf.float32` before feeding them into the first layer of the CNN. Look up the documentation of `tf.image.convert_image_dtype` and use it to do the conversion.

4. Define a network structure as shown in Table 1. The output of one layer is always the input of the next. The input to the first layer is the preprocessed images.

   - Use `tf.layers.conv2d` to define the convolutional layers[2].
   - Use `tf.layers.max_pooling2d` to define the pooling layers.
   - Use `tf.layers.flatten` and `tf.layers.dense` for the flattening and fully connected layers.

5. Define a loss function using `tf.losses.sparse_softmax_cross_entropy`. The `logits` is the output of your FC layer. The cross entropy loss is a standard choice for classification problems.

6. Define a gradient descent minimizer and a minimization op on the loss. Pick a learning rate of at least 10^{-3}.

7. To calculate the prediction accuracy (fraction of correctly predicted samples) use the following code snippet:

```
predictions = tf.argmax(logits, axis=1)
correct_preds = tf.equal(labels, predictions)
accuracy = tf.reduce_mean(tf.cast(correct_preds, tf.float32))
```

8. Add a scalar summary for monitoring the loss and the accuracy.

Training.

> Training a CNN without using GPU acceleration is usually slow. If it takes too much time, use the desktop computers in the classroom or make sure your own computer has a compatible GPU with acceleration enabled.

After you have set up your network, we can start training. Start a `tf.Session` and define a `FileWriter` object. Add a line that runs `sess.run(tf.global_variables_initializer())` before writing the next parts. Then write a training loop. At each loop iteration:

[2]This is a convenience function that internally defines all the variables necessary to hold the parameters. You do not need to manually specify the kernel weights or bias vector.

1. Draw a sample of indices to use at this iteration[3]:

```
idx = np.random.choice(train_data.shape[0], batchsize, replace=
    False)
```

   You can experiment with different values of `batchsize`, but use at least 32.

2. Run the minimizer op, and the loss and accuracy summaries. For `feed_dict`, feed in the training data and labels corresponding to the indices you sampled above!

3. Write the summaries using the `FileWriter`.

4. You can add this to print a message every 100 batches:

```
if batch % 100 == 0:
  print('Batch {:d} done'.format(batch))
```

After the training loop, add a print out for the test performance:

```
test_loss, test_accuracy = sess.run([loss, accuracy], feed_dict={
    images: test_data, labels: test_labels})
print('Test loss: {:f} -- test accuracy: {:f}'.format(test_loss,
    test_accuracy))
```

- Run the training, and open Tensorboard to visualize how it worked. What kind of test accuracy and loss do you get?
- Does it seem like the training loss is still improving? You can run for a longer time (more batches), or try to vary the other parameters (learning rate, batch size).
- Try to tune the parameters so you can improve your test accuracy!

Extra task: validation loss and accuracy

If you finished all the other tasks with time left over, you can work on this additional task.

- Split 20% of the training set into a validation set. You can select the 20% randomly, or from the beginning, or the end of the dataset.
- Add a validation step to your training. Every m iterations[4] of the main training loop, run a validation step that:
  1. Calculates the loss and accuracy on the validation data set, and
  2. Writes a summary containing the validation loss and validation accuracy to an event file.

[3]Note that this method does not guarantee that the kth batch we generate does not have any images that are in any of the preceeding batches $1, 2, \ldots, k-1$. There are ways to guarantee this, but for simplicity we do not cover them here.

[4]It is not reasonable to validate after every batch. It takes time and the result will not have changed much! Prefer to run multiple, even several hundreds, of training batches between each validation.