Cognitive Computer Vision: Exercise 3

Convolutional neural networks for classification

In the previous exercise, we solved a regression problem using a neural network with a single fully-connected layer. In a *classification* problem, we try to predict which label a given input sample has, or, in other words, to which *class* the sample belongs to. We build a convolutional neural network (CNN) to classify images of handwritten digits from 0 to 9 (Figure 1). In a CNN, we learn the filter kernels as opposed to defining them by hand as in Exercise 1.

Figure 1: Examples of handwritten digits.

The MNIST dataset

The MNIST database contains examples of handwritten digits 0-9, formatted as 28x28-pixel monochrome images. The following code snippet downloads the dataset and prints out information about the array sizes.

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

mnist = tf.contrib.learn.datasets.load_dataset("mnist")
train_data = mnist.train.images # Returns np.array
train_labels = np.asarray(mnist.train.labels, dtype=np.int32)
eval_data = mnist.test.images # Returns np.array
eval_labels = np.asarray(mnist.test.labels, dtype=np.int32)

print('Training data shape: ' + str(train_data.shape))
print('Training labels shape: ' + str(train_labels.shape))
print('Evaluation data shape: ' + str(eval_data.shape))
print('Evaluation labels shape: ' + str(eval_labels.shape))
```

Examine the arrays `train_data`, `train_labels`, `eval_data`, and `eval_labels`.

- How many training examples are there? How many evaluation examples are there?
- Why is the size of the second dimension of `train_data` and `eval_data` 784?
- Draw the first training image by using similarly to Exercise 1:

  ```python
  import matplotlib.pyplot as plt
  imgplot = plt.imshow(x, cmap='gray')
  plt.show()
  ```

Which digit is in the image? What is the corresponding label? Hints:

- A row of a 2-dimensional array `x` can be accessed by `x[i,:]` where i denotes the row to access.
- Use `y = np.reshape(y, (28,28))` to reshape the image before drawing.

Setting up a CNN

Figure 2 shows the CNN we will use in this exercise. It takes as input a batch of images, each 28-by-28 pixels in size with a single channel (grayscale). As output, the CNN produces a vector of 10 unscaled probability values, or so called logits. Set up a CNN as shown in Figure 2. Use the code in Listing 1 as a starting point.

- CONV layers: use `tf.layers.conv2d` with `padding="same"`.
- POOL layers: use `tf.layers.max_pooling2d`.
- FC layers: use `tf.layers.dense`.
- Important: you must flatten the output tensor of POOL2 before feeding it into the FC1 layer! See line 29 in Listing 1.

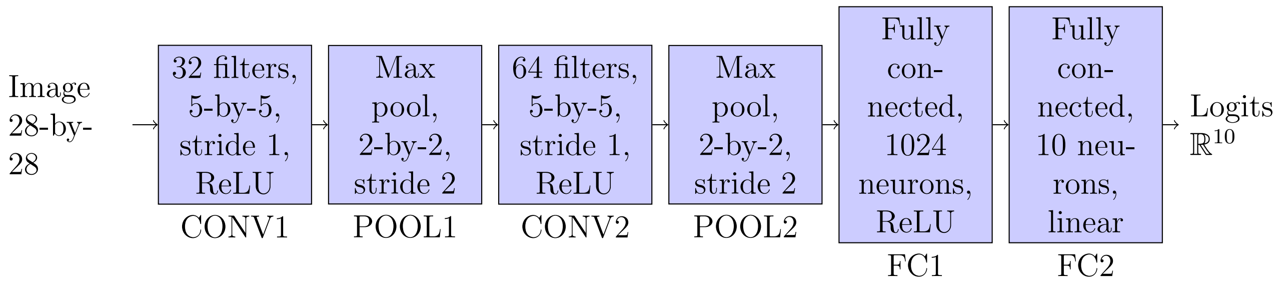
Image 28-by-28
32 filters, 5-by-5, stride 1, ReLU
CONV1
Max pool, 2-by-2, stride 2
POOL1
64 filters, 5-by-5, stride 1, ReLU
CONV2
Max pool, 2-by-2, stride 2
POOL2
Fully connected, 1024 neurons, ReLU
FC1
Fully connected, 10 neurons, linear
FC2
Logits $\mathbb{R}^{10}$

Figure 2: CNN for digit classification.

```
1  # Placeholders for input images and labels
2  # The first dimension (the batch size) will be determined at runtime.
3  x = tf.placeholder(tf.float32, shape=(None, 784))
4  labels = tf.placeholder(tf.int32, shape=(None,))
5
6  # Before feeding the images to the CNN, reshape them to proper size.
7  # Meaning of -1: infer the actual value from the shape of the input
8  input_layer = tf.reshape(x, [-1, 28, 28, 1])
9
10 # Next: insert CNN layers below!
11 # Why are the layer i/o tensor shapes as they are claimed to be?
12 # Compare the shapes to yours to see if your implementation is correct
13
14 # ** CONV1
15 # Input Tensor Shape: [batch_size, 28, 28, 1]
16 # Output Tensor Shape: [batch_size, 28, 28, 32]
17 # ** POOL1
18 # Input Tensor Shape: [batch_size, 28, 28, 32]
19 # Output Tensor Shape: [batch_size, 14, 14, 32]
20 # ** CONV2
21 # Input Tensor Shape: [batch_size, 14, 14, 32]
```

```
# Output Tensor Shape: [batch_size, 14, 14, 64]
# ** POOL2
# Input Tensor Shape: [batch_size, 14, 14, 64]
# Output Tensor Shape: [batch_size, 7, 7, 64]
# ** Flatten tensor into a batch of vectors
# Input Tensor Shape: [batch_size, 7, 7, 64]
# Output Tensor Shape: [batch_size, 7 * 7 * 64]
pool2_flat = tf.reshape(pool2, [-1, 7 * 7 * 64])
# ** FC1
# Input Tensor Shape: [batch_size, 7 * 7 * 64]
# Output Tensor Shape: [batch_size, 1024]
# ** FC2
# Input Tensor Shape: [batch_size, 1024]
# Output Tensor Shape: [batch_size, 10]

# Define a loss function to optimize.
# This assumes the final layer of your CNN is called logits.
loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=
    logits)

# Accuracy is the fraction of correctly predicted classes.
predictions = tf.argmax(logits, axis=1, output_type=tf.int32)
correct_prediction = tf.equal(labels, predictions)
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

# Define an optimizer and training op
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
train_op = optimizer.minimize(loss)
```

Listing 1: CNN classifier

Before moving on, ensure that the building of the computational graph and thus setting up of your CNN works without errors. Add the code necessary to view the graph in Tensorboard. Ensure the shapes of the tensors match the values above.

Training the CNN

Insert the code from above that loads the dataset at the end of the CNN building code from the previous section. Then add afterwards the following training loop:

```
batchsize = 100
num_training_examples = train_data.shape[0]
num_batches = num_training_examples / batchsize
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())

  for b in range(num_batches):
    starting_index = b*batchsize
    stopping_index = min((b+1)*batchsize, num_training_examples)

    batch_images = train_data[starting_index:stopping_index, :]
    batch_labels = train_labels[starting_index:stopping_index]

    batch_loss, batch_acc, _ = sess.run([loss, accuracy, train_op],
   feed_dict={x: batch_images, labels: batch_labels})
    print('Batch ' + str(b) + ', loss: ' + str(batch_loss) + ',
   accuracy: ' + str(batch_acc))
```

- Run the code and observe the output.
- Experiment with different values of the learning rate.
- Add a scalar summary similar to Exercise 2 to visualize the training loss as function of the number of batches.

Validation

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.

After the training loop, but still inside the `with tf.Session() as sess` block, add the following validation loop that will calculate the average loss and average accuracy on the validation dataset.

```python
num_eval_examples = eval_data.shape[0]
num_eval_batches = num_eval_examples / batchsize
total_eval_loss = 0.0
total_eval_acc = 0.0
for b in range(num_eval_batches):
  starting_index = b*batchsize
  stopping_index = min((b+1)*batchsize, num_eval_examples)

  batch_images = eval_data[starting_index:stopping_index, :]
  batch_labels = eval_labels[starting_index:stopping_index]
  [batch_eval_loss, batch_eval_acc] = sess.run([loss, accuracy],
   feed_dict={x: batch_images, labels: batch_labels})
  total_eval_loss += batch_eval_loss
  total_eval_acc += batch_eval_acc

print('Average loss on validation data: ' + str(total_eval_loss /
   num_eval_batches))
print('Average accuracy on validation data: ' + str(total_eval_acc /
   num_eval_batches))
```

- How is your validation accuracy and loss compared to training?