

HƯỚNG DẪN CÀI ĐẶT MẠNG TÍCH CHẬP SỬ DỤNG FRAMEWORK TENSORFLOW

Nguyễn Vinh Tiệp và các cộng sự VietAI

Ngày 24 tháng 5 năm 2018

Tóm tắt nội dung

Chương này hướng dẫn cách sử dụng framework Tensorflow để cài đặt một Mạng tích chập (Convolutional Neural Network - CNN). Các bước chính để tiến hành bao gồm: cài đặt lớp *LeNet* để xây dựng kiến trúc mạng cho trước, cài đặt lớp *Dataset* để quản lý tập dữ liệu. Để hiểu rõ hơn bản chất và vai trò của các siêu tham số trong mạng CNN, chúng tôi đề xuất một số giả thuyết và thiết kế thí nghiệm nhằm trả lời các giả thuyết này.

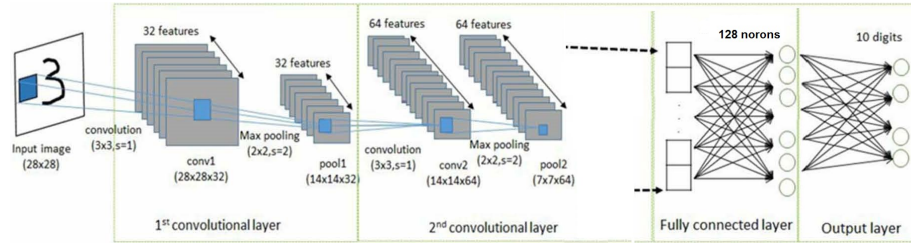
1 Giới thiệu kiến trúc mạng LeNet

LeNet là một trong những Mạng tích chập đầu tiên được đề xuất để giải quyết bài toán nhận dạng chữ số viết tay (Handwritten Digit Recognition) [1]. Phiên bản gốc có kiến trúc tương đối đơn giản. Kế thừa các kết quả nghiên cứu mới nhất, mạng LeNet vẫn cố định về mặt độ sâu (số lớp ẩn) nhưng có một số thay đổi như:

- Hàm kích hoạt (Activation Function): sử dụng hàm *ReLU* (Rectified Linear Unit) thay vì sử dụng các hàm phi tuyến quen thuộc như *sigmoid*, *tanh*.
- Pooling: Sử dụng phương pháp *max pooling* thay vì *average pooling*.
- Dropout: Trong quá trình huấn luyện, loại bỏ bớt một số neuron một cách ngẫu nhiên để giảm bớt hiệu ứng overfitting.

2 Cài đặt lớp LeNet

Kiến trúc của mạng LeNet mở rộng được mô tả như trên Hình 1. Để tiến hành cài đặt sử dụng tensorflow, ta tạo class LeNet với các phương thức chính như: hàm khởi tạo *init*, hàm tạo các lớp tích chập *conv_layers*, hàm tạo các lớp kết nối đầy đủ *fc_layers* và hàm huấn luyện *train*.



Hình 1: Kiến trúc mạng LeNet mở rộng.

Đối với hàm khởi tạo, ta tiến hành khởi tạo các biến placeholder để xác định bộ khung kích thước dữ liệu đầu vào và đầu ra của mạng CNN. Đầu vào của hệ thống được lưu trong thuộc tính *self.X* có kích thước là $[None, 28, 28, 1]$. Trong đó *None* thể hiện số lượng dữ liệu đưa vào mạng là không biết trước. Ảnh đầu vào có kích thước 28×28 với 1 kênh màu thể hiện mức xám. Đồng thời, chúng ta khởi tạo tham số khác như *self.keep_prob* cho phương pháp Dropout, *self.log* để chuyển đổi chế độ ghi hay không ghi log, *self.sess* để lưu giữ đối tượng *tf.Session*. Sau đó ta tiến hành khởi tạo kiến trúc mạng và định nghĩa xác suất phân lớp chữ số dựa trên hàm *Softmax*.

```

1 def __init__(self, weights=None, sess=None, log=True):
2     self.X = tf.placeholder(tf.float32, [None, 28, 28, 1], name='X')
3     self.keep_prob = tf.placeholder(tf.float32, name='keep_prob')
4     self.log = log
5     self.sess = sess
6
7     self.conv_layers()
8     self.fc_layers()
9
10    self.probs = tf.nn.softmax(self.logits, name='softmax')
```

Đối với các hàm tạo lớp tích chập và tạo các lớp kết nối đầy đủ, ta dựa trên các mô tả sau:

Lớp input: chứa dữ liệu ảnh đầu vào với kích thước được chuẩn hóa 28×28 . Ảnh này chỉ có một kênh màu (ảnh mức xám - grayscale) nên kích thước ảnh khi khai báo trên tensorflow là $[28 \times 28 \times 1]$. Thực tế khi huấn luyện hoặc test trên nhiều ảnh khi đó dữ liệu truyền vào sẽ là một tensor 4D có kích thước: $[batch_size \times 28 \times 28 \times 1]$. Trong đó, *batch_size* là số lượng ảnh đưa vào mạng CNN để huấn luyện hoặc test.

```

1 # Input:
2 images = self.X
```

Lớp tích chập thứ 1: bao gồm các phép biến đổi được thực hiện tuần tự như sau: tích chập, cổng phi tuyến ReLU và max pooling. Trong đó biến đổi tích chập được thực hiện trên 32 bộ lọc (filter) kích thước 3×3 trên các ảnh 1 chiều. Do đó tham số cho phần này sẽ là một biến *kernel* kiểu tensor 4D kích thước $[3 \times 3 \times 1 \times 32]$. Đối với bước thực hiện max pooling, ta tiến hành trên

của số kích thước 2×2 đồng thời làm giảm kích thước của bản đồ đặc trưng (feature map) còn một nửa nên tham số *strides* = 2. Kết thúc bước này ta sẽ được một tensor kích thước $[14 \times 14 \times 32]$, đồng thời cũng là đầu vào của Lớp tích chập thứ 2.

```

1 # Layer 1: Conv
2 with tf.name_scope('conv1') as scope:
3     kernel = tf.Variable(tf.random_normal([3, 3, 1, 32], dtype=tf.
4         float32, stddev=1e-1, name='weights'))
5     conv = tf.nn.conv2d(images, kernel, [1, 1, 1, 1], padding='SAME
6         ')
7     biases = tf.Variable(tf.constant(0.0, shape=[32], dtype=tf.
8         float32), trainable=True, name='biases')
9     out = tf.nn.bias_add(conv, biases)
10    self.conv1 = tf.nn.relu(out, name=scope)
11    self.parameters += [kernel, biases]
12
13 # Layer 1: Pooling
14 self.pool1 = tf.nn.max_pool(self.conv1,
15     ksize=[1, 2, 2, 1],
16     strides=[1, 2, 2, 1],
17     padding='SAME',
18     name='pool1')

```

Lớp tích chập thứ 2: tương tự như lớp tích chập thứ 1, cũng bao gồm các phép biến đổi được thực hiện tuần tự: tích chập, cộng phi tuyến và max pooling. Riêng đối với phép tích chập được thực hiện trên 64 bộ filter 3×3 trên tập hợp 32 feature map. Do đó tham số cho phần này là một biến kiểu tensor 4D kích thước $[3 \times 3 \times 32 \times 64]$. Đầu ra của bước này là một tensor kích thước $[7 \times 7 \times 64]$ do được thực hiện giảm một nửa kích thước đặc trưng.

```

1 # Layer 2: Conv
2 with tf.name_scope('conv2') as scope:
3     kernel = tf.Variable(tf.random_normal([3, 3, 32, 64], dtype=tf.
4         float32, stddev=1e-1, name='weights'))
5     conv = tf.nn.conv2d(self.pool1, kernel, [1, 1, 1, 1], padding='
6         SAME')
7     biases = tf.Variable(tf.constant(0.0, shape=[64], dtype=tf.
8         float32), trainable=True, name='biases')
9     out = tf.nn.bias_add(conv, biases)
10    self.conv2 = tf.nn.relu(out, name=scope)
11    self.parameters += [kernel, biases]
12
13 # Layer 2: Pooling
14 self.pool2 = tf.nn.max_pool(self.conv2,
15     ksize=[1, 2, 2, 1],
16     strides=[1, 2, 2, 1],
17     padding='SAME',
18     name='pool2')

```

Lớp kết nối đầy đủ thứ 1: từ kết quả tính toán ở bước cuối cùng của lớp tích chập thứ 2, feature map được trải ra thành một vector kích thước $[1 \times \text{shape}]$, trong đó *shape* là tổng số phần tử của tensor đầu ra của bước tích chập thứ 2. Cụ thể trong trường hợp này $\text{shape} = 7 * 7 * 64 = 3136$. Với dữ liệu đầu vào này, ta thực hiện tuần tự các biến đổi như sau: nhân ma trận trọng số

$fc1w$, cộng vector $fc1b$ và cổng phi tuyến ReLU. Số lượng neuron đầu ra của bước này bao gồm 128 neuron, do đó các tham số trọng số $fc1w$ và bias $fc1b$ có kích thước lần lượt là $[3136 \times 128]$ và $[1 \times 128]$. Để giảm khả năng overfitting, ta sử dụng phương pháp Dropout với tham số $self.keep_prob$.

Lớp kết nối đầy đủ thứ 2: tương tự như lớp thứ 1 nhưng kết quả đầu ra bao gồm 10 neuron tương ứng với 10 chữ số từ 0 đến 9. Hai lớp này được cài đặt như sau:

```

1 # fc1
2 with tf.name_scope('fc1') as scope:
3     shape = int(np.prod(self.pool2.get_shape()[1:]))
4     fc1w = tf.Variable(tf.random_normal([shape, 128], dtype=tf.
5         float32, stddev=1e-1), name='weights')
6     fc1b = tf.Variable(tf.constant(1.0, shape=[128], dtype=tf.
7         float32), trainable=True, name='biases')
8     pool2_flat = tf.reshape(self.pool2, [-1, shape])
9     fc1l = tf.nn.bias_add(tf.matmul(pool2_flat, fc1w), fc1b)
10    self.fc1 = tf.nn.relu(fc1l)
11    self.dropout1 = tf.nn.dropout(self.fc1, keep_prob=self.
12        keep_prob, name='dropout1')
13    self.parameters += [fc1w, fc1b]
14
15 # fc2
16 with tf.name_scope('fc2') as scope:
17     fc2w = tf.Variable(tf.random_normal([128, 10], dtype=tf.float32
18         , stddev=1e-1), name='weights')
19     fc2b = tf.Variable(tf.constant(1.0, shape=[10], dtype=tf.
20         float32), trainable=True, name='biases')
21     fc2l = tf.nn.bias_add(tf.matmul(self.dropout1, fc2w), fc2b)
22     self.logits = tf.nn.relu(fc2l)
23     self.parameters += [fc2w, fc2b]

```

3 Cài đặt lớp Dataset

Để quản lý và thao tác trên tập dữ liệu huấn luyện, ta tạo class *DataSet* với các phương thức chính như: hàm khởi tạo *init*, hàm lấy khối dữ liệu huấn luyện *next_batch* và khối dữ liệu test *next_batch_test*. Đối với hàm khởi tạo, ta tiến hành tải về và load dữ liệu vào bộ nhớ:

```

1 def __init__(self):
2     mnist = tf.contrib.learn.datasets.load_dataset("mnist")
3     self.train_data = mnist.train.images # Returns np.array
4     self.train_labels = np.asarray(mnist.train.labels, dtype=np.
5         int32)
6     self.eval_data = mnist.test.images # Returns np.array
7     self.eval_labels = np.asarray(mnist.test.labels, dtype=np.int32
8         )
9
10    self.curr_training_step = 0
11    self.curr_test_step = 0

```

Kết thúc bước này, đối tượng sẽ quản lý các tập dữ liệu huấn luyện (*self.train_data*, *self.train_labels*) và tập dữ liệu để đánh giá độ chính xác thật sự của hệ thống

(self.eval_data, self.eval_labels). Do kích thước dữ liệu huấn luyện và đánh giá rất lớn nên để sử dụng một cách hiệu quả, ta trang bị các hàm lấy dữ liệu theo từng khối như sau:

```

1 def next_batch(self, batch_size):
2     X_train_bs = self.train_data[self.curr_training_step *
3     batch_size:self.curr_training_step * batch_size + batch_size,:]
4     Y_train_bs = self.train_labels[self.curr_training_step *
5     batch_size:self.curr_training_step * batch_size + batch_size]
6
7     self.curr_training_step = self.curr_training_step + 1
8     self.curr_training_step = self.curr_training_step if (
9     self.curr_training_step * batch_size < self.get_train_set_size
10    ()) else 0
11
12    return (X_train_bs, self.to_one_hot(Y_train_bs))
13
14 def next_batch_test(self, batch_size):
15     X_test_bs = self.eval_data[self.curr_test_step * batch_size:
16     self.curr_test_step * batch_size + batch_size,:]
17     Y_test_bs = self.eval_labels[self.curr_test_step * batch_size:
18     self.curr_test_step * batch_size + batch_size]
19
20     self.curr_test_step = self.curr_test_step + 1
21     self.curr_test_step = self.curr_test_step if (self.
22     curr_test_step * batch_size < self.get_test_set_size()) else 0
23
24    return (X_test_bs, self.to_one_hot(Y_test_bs))

```

4 Cài đặt hàm train của lớp đối tượng LeNet

Sau khi đã cài đặt xong lớp *DataSet*, chúng ta tiến hành cài đặt hàm *train* cho lớp đối tượng *LeNet*. Các bước chính để thực hiện việc huấn luyện trên mạng LeNet bao gồm:

- Khởi tạo đối tượng của lớp *DataSet* để thực hiện việc tải và load dataset MNIST vào bộ nhớ.

```

1 # Load dataset for training and testing
2 self.dataset = DataSet()

```

- Định nghĩa kiểu dữ liệu đầu ra *self.Y*, hàm độ lỗi, phương pháp huấn luyện

```

1 # Define size of output
2 self.Y = tf.placeholder(tf.float32, [None, 10], name='Y')
3 # Define cost function
4 self.cost = tf.reduce_mean(tf.nn.
5     softmax_cross_entropy_with_logits(logits=self.logits,
6     labels=self.Y))
7 # Define optimization method
8 self.optimizer = tf.train.AdamOptimizer(learning_rate=
9     learning_rate).minimize(self.cost)

```

- Khởi tạo các biến toàn cục và cục bộ

```

1 self.sess.run(tf.global_variables_initializer())
2 self.sess.run(tf.local_variables_initializer())

```

- Huấn luyện với nhiều vòng khác (epoch) khác nhau

```

1 print('Training...')
2 weights = []
3 # For each epoch, feed training data and perform updating
  parameters
4 for epoch in range(training_epochs):
5     avg_cost = 0
6     # Number of batches = size of training set / batch_size
7     total_batch = int(self.dataset.get_train_set_size() /
8                        batch_size)
9     # For each batch
10    for i in range(total_batch + 1):
11        # Get next batch to feed to the network
12        batch_xs, batch_ys = self.dataset.next_batch(batch_size)
13        feed_dict = {
14            self.X: batch_xs.reshape([batch_xs.shape[0], 28, 28, 1]),
15            self.Y: batch_ys,
16            self.keep_prob: keep_prob
17        }
18
19        weights, summary, c, _ = self.sess.run([self.parameters, self.
20        merged, self.cost, self.optimizer],
21        feed_dict=feed_dict)
22        print('weights: ', weights)
23        print('summary: ', summary),
24        print('cost: ', c)
25        avg_cost += c / total_batch
26
27    if self.log:
28        self.train_writer.add_summary(summary, epoch + 1)
29
30    print('Epoch:', '%02d' % (epoch + 1), 'cost =', '{:.9f}'.
31        format(avg_cost))

```

5 Cài đặt hàm Main

Để sử dụng các lớp đối tượng trên ta cài đặt hàm main như sau:

```

1 def main(unused_argv):
2     sess = tf.Session()
3     lenet = LeNet(sess=sess, weights=None)
4
5     lenet.train(learning_rate=0.001, training_epochs=40, batch_size
6                 =1000, keep_prob=0.7)
7     lenet.evaluate(batch_size=1000, keep_prob=0.7)

```

Để quan sát quá trình huấn luyện, ta theo dõi hàm độ lỗi bằng cách tạo một màn hình tham số dòng lệnh khác. Sau đó vào thư mục hiện hành và khởi động một web service bằng tensorboard: `tensorboard --logdir=train_dir`. Sau đó mở

trình duyệt web vào trang web có cổng 6006 được hiện ra khi gọi câu lệnh trên. Ví dụ như: `http://localhost:6006`.

6 Một số thí nghiệm

Để đánh giá vai trò của các tham số, ta đặt ra các giả thuyết sau:

- Số lượng neuron cho mỗi lớp càng lớn càng tốt?
- Sử dụng cổng phi tuyến *ReLU* cho kết quả tốt hơn so với *sigmoid* và *tanh*.
- Sử dụng phương pháp Dropout cho kết quả tốt hơn so với không sử dụng?

Các bạn hãy thử thay đổi các tham số trong hệ thống, ghi nhận kết quả và đánh giá các giả thuyết trên là đúng hay sai? Ví dụ: Chúng ta có thể nâng số filter của lớp tích chập thứ nhất và thứ hai lần lượt từ 32 lên 64 và 64 lên 128. Sử dụng hàm activation function là *sigmoid* thay vì *ReLU*. Lưu ý rằng, để có kiểm chứng một giả thuyết thì chúng ta cần thiết đặt thí nghiệm một cách công bằng, nghĩa là chỉ thay đổi trên một nhóm các tham số có liên quan, các tham số còn lại phải cố định.

Tài liệu

- [1] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.