北京航空航天大學
BEIHANG UNIVERSITY

Pattern Recognition and Machine Learning

Experiment Report

School of Automation Science and Electrical Engineering

Kun Xiao

14051166

June,2018

Experiment 4 Neural Networks and Back Propagation

1 Introduction

The learning model of Artificial Neural Networks (ANN) (or just a neural network (NN)) is an approach inspired by biological neural systems that perform extraordinarily complex computations in the real world without recourse to explicit quantitative operations. The original inspiration for the technique was from examination of bioelectrical networks in the brain formed by neurons and their synapses. In a neural network model, simple nodes (called variously “neurons” or “units”) are connected together to form a network of nodes, hence the term “neural network”.

Each node has a set of input lines which are analogous to input synapses in a biological neuron. Each node also has an “activation function” that tells the node when to fire, similar to a biological neuron. In its simplest form, this activation function can just be to generate a ‘1’ if the summed input is greater than some value, or a ‘0’ otherwise. Activation functions, however, do not have to be this simple - in fact to create networks that can do useful things, they almost always have to be more complex, for at least some of the nodes in the network. Typically there are at least three layers to a feed-forward network - an input layer, a hidden layer, and an output layer. The input layer does no processing - it is simply where the data vector is fed into the network. The input layer then feeds into the hidden layer. The hidden layer, in turn, feeds into the output layer. The actual processing in the network occurs in the nodes of the hidden layer and the output layer.

2 Principle and Theory

The goal of any supervised learning algorithm is to find a function that best maps a set of inputs to its correct output. An example would be a simple classification task, where the input is an image of an animal, and the correct output would be the name of the animal. For an intuitive example, the first layer of a Neural Network may be responsible for learning the orientations of lines using the inputs from the individual pixels in the image. The second layer may combine the features learned in the first layer and learn to identify simple shapes such as circles. Each higher layer learns more and more abstract features such as those mentioned above that can be used to classify the image. Each layer finds patterns in the layer below it and it is this ability to create internal representations that are independent of outside input that gives multi-layered networks their power. The goal and motivation for developing the back-propagation algorithm was to find a way to train a multi-layered neural network such that it

can learn the appropriate internal representations to allow it to learn any arbitrary mapping of input to output.

Mathematically, a neuron's network function f(x) is defined as a composition of other functions gi(x) which can further be defined as a composition of other functions. This can be conveniently represented as a network structure, with arrows depicting the dependencies between variables. A widely used type of composition is the nonlinear weighted sum, where:

$$\mathrm{f(x)} = (\sum_i w_i g_i(x))$$

where K (commonly referred to as the activation function) is some predefined function, such as the hyperbolic tangent. It will be convenient for the following to refer to a collection of functions gi as simply a vector g = (g1, g2, ..., gn). Back-propagation requires a known, desired output for each input value in order to calculate the loss function gradient. It is therefore usually considered to be a supervised learning method.

The squared error function is:

$$E = \frac{1}{2}(t - y)^2$$

where E is the squared error, t is the target output for a training sample, and y is the actual output of the output neuron. For each neuron j, its output o_j is defined as

$$o_j = \varphi(net_j) = \varphi(\sum_{k=1}^{n} w_{kj} x_k)$$

The input net to a neuron is the weighted sum of outputs ok of previous neurons. If the neuron is in the first layer after the input layer, the o_k of the input layer are simply the inputs x_k to the network. The number of input units to the neuron is n. The variable w_{ij} denotes the weight between neurons i and j.

The activation function φ is in general non-linear and differentiable. A commonly used activation function is the logistic function, e.g.

$$\varphi(z) = \frac{1}{1 + e^{-z}}$$

which has a nice derivative of:

$$\frac{\partial \varphi}{\partial z} = \varphi(1 - \varphi)$$

Calculating the partial derivative of the error with respect to a weight w_{ij} is done using the chain rule twice:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

We can finally yield:

$$\frac{\partial E}{\partial w_{ij}} = \delta_j x_i$$

with

$$\delta_j = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial net_j} = \begin{cases} (o_j - t_j)\varphi(net_j)\left(1 - \varphi(net_j)\right) & if\ j\ is\ an\ output\ neuron \\ \left(\sum_{l \in L} \delta_l w_{jl}\right)\varphi(net_j)\left(1 - \varphi(net_j)\right) & if\ j\ is\ an\ inner\ neuron \end{cases}$$

3 Objective

The goals of the experiment are as follows:

(1) To understand how to build a neural network for a classification problem.

(2) To understand how the back-propagation algorithm is used for training a given a neural network.

(3) To understand the limitation of the neural network model (e.g., the local minimum).

(4) To understand how to use back-propagation in Autoencoder.

4 Contents and Procedures

该实验对两个数据集进行了训练，分别是 Iris 和 MINIST，二者都是多分类任务。前者的数据量较小，分类较简单；而后者本质是二维图像数据，一维化排序后也可以处理，但显然后者的数据量远大于前者。本实验对于前者，使用了具有六个隐含节点的单隐含层 MLP；对于后者，则尝试了不同方法，不含隐含层的感受器，含一个全连接隐含层、含两个全连接隐含层的 MLP 以及含两个卷积隐含层和一个全连接隐含层的 CNN，并对不同学习率对训练结果的影响进行了探究。

本实验开发环境为 Ubuntu 14.04+ Python 3.4+Tensorflow 1.3.0+Cuda 8.0+CuDNN 6

IRIS 分类程序：

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
import tensorflow as tf
import numpy as np

IRIS_TRAINING = "iris_training.csv"
IRIS_TEST = "iris_test.csv"

# 读取训练集和测试集
training_set = tf.contrib.learn.datasets.base.load_csv_with_header(
```

```
    filename=IRIS_TRAINING,
    target_dtype=np.int,
    features_dtype=np.float32)
test_set = tf.contrib.learn.datasets.base.load_csv_with_header(
    filename=IRIS_TEST,
    target_dtype=np.int,
    features_dtype=np.float32)

# 特征
feature_columns = [tf.contrib.layers.real_valued_column("", dimension=4)]

# 构建 MLP，单隐含层，有三个隐含节点
classifier = tf.contrib.learn.DNNClassifier(feature_columns=feature_columns,
                                            hidden_units=[3],
                                            n_classes=3,
                                            optimizer=tf.train.AdamOptimizer(0.1))

# 拟合模型，迭代 20000 步
classifier.fit(x=training_set.data, y=training_set.target, steps=20000)

# 计算精度
accuracy_score = classifier.evaluate(x=test_set.data, y=test_set.target)["accuracy"]

print('Accuracy: {0:f}'.format(accuracy_score))

# 预测新样本的类别
new_samples = np.array([[6.4, 3.2, 4.5, 1.5], [5.8, 3.1, 5.0, 1.7]], dtype=np.float32)
y = list(classifier.predict(new_samples, as_iterable=True))
print('Predictions: {}'.format(str(y)))
```

使用 0.05 初始学习率的 Adam 优化器，测试准确率为 100% 但当把学习率调高到 0.3 时，训练结果就很不稳定，测试准确率有时是 1，有时只有 33.3%甚至更低。可见学习率对训练效果有很大的影响。

由于 Iris 数据量较小，不适合用复杂的神经网络训练。因此对于层数的研究，我使用的是 MNIST 数据。

MNIST 分类程序（DNN）：

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets("MNIST_data", one_hot=True)
sess = tf.InteractiveSession()

in_units = 784
h1_units = 300
h2_units = 100
W1 = tf.Variable(tf.truncated_normal([in_units, h1_units], stddev=0.1))
```

```
b1 = tf.Variable(tf.zeros([h1_units]))
W2 = tf.Variable(tf.truncated_normal([h1_units, h2_units], stddev=0.1))
b2 = tf.Variable(tf.zeros([h2_units]))
W3 = tf.Variable(tf.zeros([h2_units, 10]))
b3 = tf.Variable(tf.zeros([10]))

x = tf.placeholder(tf.float32, [None, in_units])
keep_prob = tf.placeholder(tf.float32)

hidden1 = tf.nn.relu(tf.matmul(x, W1) + b1)
hidden1_drop = tf.nn.dropout(hidden1, keep_prob)

hidden2 = tf.nn.relu(tf.matmul(hidden1_drop, W2) + b2)
hidden2_drop = tf.nn.dropout(hidden2, keep_prob)

y = tf.nn.softmax(tf.matmul(hidden2_drop, W3) + b3)

y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean((-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1])))
train_step = tf.train.AdagradOptimizer(0.05).minimize(cross_entropy)

correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

tf.global_variables_initializer().run()
for i in range(10000):
    batch_xs, batch_ys = mnist.train.next_batch(32)
    train_step.run({x: batch_xs, y_: batch_ys, keep_prob: 0.75})
    train_accuracy = accuracy.eval(feed_dict={x: batch_xs, y_: batch_ys, keep_prob: 1})
    train_loss=cross_entropy.eval(feed_dict={x: batch_xs, y_: batch_ys, keep_prob: 1})
    print("step %d, train accuracy %g ,%g" % (i, train_accuracy,train_loss))

print(accuracy.eval({x: mnist.train.images, y_: mnist.train.labels, keep_prob: 1.0}))
```

上面展示的是具有两个全连接隐含层（第一层 300 个节点，第二层 100 个节点）的 MLP，最后靠 Softmax 输出层实现的多分类。分别删一个隐含层、两个隐含层，即得到感知器和单隐含层 MLP。比较三者的测试准确率，感知器的准确率在 92%左右，单隐含层在 98%左右，双隐含层在 98.8%左右（此三者均在使用各种 trick，调到比较好的状态下测得），可见层数对于 MNIST 具有较大的影响。

然而全连接层网络，再怎么调试，也难以使 MNIST 分类准确率达到 99%以上，原因就在于 MNIST 数据其实是图像数据，一维排列后，许多二维图像特征丢失，但是若想直接处理二维图像的话，MLP 的过多节点和过多权值，导致了过高的计算复杂度。而卷积神经网络（CNN）很好地解决了这个问题。 CNN 的要点是局部连接、权值共享与池化层（Pooling）的降采样。局部连接和权值共享降低了参数量，使训练复杂度大大下降，并减轻了过拟合。同时权值共享还赋予了卷及网络对平移的容忍性，而池

化层降采样则进一步降低了输出参数量，并赋予模型对轻度形变的容忍性，提高了模型的泛化能力。

MNIST 分类程序（CNN）：

```
from tensorflow.examples.tutorials.mnist import input_data
import tensorflow as tf

mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
sess = tf.InteractiveSession()


def weight_variable(shape):
  initial = tf.truncated_normal(shape, stddev=0.1)
  return tf.Variable(initial)


def bias_variable(shape):
  initial = tf.constant(0.1, shape=shape)
  return tf.Variable(initial)


def conv2d(x, W):
  return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding="SAME")


def max_pool_2x2(x):
  return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding="SAME")


x = tf.placeholder(tf.float32, [None, 784])
y_ = tf.placeholder(tf.float32, [None, 10])
x_image = tf.reshape(x, [-1, 28, 28, 1])

W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)

W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])
h_pool2_flat = tf.reshape(h_pool2, [-1, 7 * 7 * 64])

h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
```

```
    keep_prob = tf.placeholder(tf.float32)
    h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

    W_fc2 = weight_variable([1024, 10])
    b_fc2 = bias_variable([10])
    y_conv = tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)

    cross_entropy = tf.reduce_mean((-tf.reduce_sum(y_ * tf.log(y_conv),
reduction_indices=[1])))
    train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)

    correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

    tf.global_variables_initializer().run()
    for i in range(20000):
      batch = mnist.train.next_batch(50)
      if i % 100 == 0:
        a = 10
        b = 50
        sum = 0
        if i % 100 == 0:
          train_accuracy = accuracy.eval(feed_dict={x: batch[0], y_: batch[1], keep_prob:
0.5})
          print("step %d, train accuracy %g" % (i, train_accuracy))
      train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

    # print("test accuracy %g" % accuracy.eval(feed_dict={x: mnist, y_: mnist.test.labels,
keep_prob: 1.0}))
    a = 10
    b = 50
    sum = 0
    for i in range(a):
      testSet = mnist.test.next_batch(b)
      c = accuracy.eval(feed_dict={x: testSet[0], y_: testSet[1], keep_prob: 1.0})
      sum += c
    print("test accuracy %g" % (sum / a))
```

该程序使用了两个卷积隐含层和一个全连接隐含层，测试准确度最后为 99.2%，高于两个全连接隐含层的 MLP 的 98.8%

对于人工神经网络与生物神经网络，以现在人们对于脑科学的研究，在很多方面，二者具有相似性。但显然，二者不可能相同，原因很简单，因为人类目前对于脑的认知还很有限，也谈不上完全模仿脑的运作方式。

5 Conclusion

网络的类型、层数、相关参数（如学习率）等等，都对于神经网络的训练效果有非常大的影响。数据集越复杂，层数要越深、节点要越多，学习率、batch size、训练步数等等，要不断地调试，才能取得良好的训练效果。