CS 534 Machine Learning

Project 3, due Monday, April 16

Chenxi Cai
Yifei Ren
Qingfeng (Kee) Wang

(Notice, all links (table of contents, figures references, table references etc.) are clickable! You can download this PDF and open it using Preview or Adobe Reader to enable this convenient feature.)

Contents

1 Perceptron Implementation

1.1 Analysis

For perception, the prediction is: $g(x) = \sum w_i x_i$ (in which x_i contains the bais term!) and predicts 1 if $g(x) \geq 0$, or -1 if $g(x) < 0$. So in order to predict result, we need to train weights first. Weight for each feature is trained row by row. If misclassified, weights will be updated which is:

$$\begin{bmatrix} w_1 \\ w_2 \\ bias \end{bmatrix}_{i+1} = \begin{bmatrix} w_1 \\ w_2 \\ bias \end{bmatrix}_i + learning\_rate * y_i * \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix}_i$$

where $learning_rate = 1$ is given, $\mathbf{w}$ is the weight array of size (nfeature + 1), y_i is a scalar of data `y[i]`, $\mathbf{x}$ is a array of size (nfeature + 1). Obviously this is going to be an iterative method.

Initial setup: initial bias are 1, initial weights are 1.

1.2 Result and Discussion

Finial weights are $[3.15, 3.17]$ for feature x_1 and x_2. Final bias is -3. The prediction result is shown in Figure. **??**.

Interestingly, it is possible to reduce the testing error rate to 0 by setting initial bias to other values other than 1, for example, 10 or -1. Probably that is because they converge at a different local optimal value.

2 Adaboost Implementation

2.1 Analysis

The final prediction is $G(x) = sign[\sum_{m=1}^{M} \alpha_m G_m(x)]$, where M is the number of iterations (weak classifiers), and each iteration corresponds to a classifier $G_m(x)$ with a weight α_m.

For each classifier, α can be obtained simply from

$$\alpha = \log(\frac{1 - err_m}{err_m})$$

(Not sure about the base of log, I use e), where

$$err_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i},$$

and where N is the number of points (samples).

$I(y_i \neq G_m(x_i))$ is a function for m-th classifier. It is a 1D array with length number of points. For each entry, it equals to 1 if G_m classifier predicts point wrong or 0 if predicts correctly (Not sure about this one, need check with TA). So, the higher success rate, the higher weight α which makes perfect sense.

Here w_i is the weight that initialized as $w_i = \frac{1}{N}$, for each point, and updates as

$$w_i = w_i e^{\alpha_m I(y_i \neq G_m(x_i))},$$

or explicitly:

$$\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}_{i+1} = \begin{bmatrix} w_1 e^{\alpha_m I(y_i \neq G_m(x_1))} \\ w_2 e^{\alpha_m I(y_i \neq G_m(x_2))} \\ \vdots \\ w_N e^{\alpha_m I(y_i \neq G_m(x_N))} \end{bmatrix}_{i+1}$$

so the weight of the point i w_i does not change if predicts correctly (since $w_i e^{\alpha_m I(y_i \neq G_m(x_1))} = w_i e^0 = w_i$) and will update otherwise (since $w_i e^{\alpha_m I(y_i \neq G_m(x_1))} = w_i e^{\alpha_m}$).

Finally, $G_m(x)$ is a cutoff on one of the axes x=xi or y = yi, depending on if the cutoff gives the lowest error rate.

2.2 Results and discussion

Error rate with different iterations are listed in Table. **??**. A more detailed study is shown in plot Figure. **??**, which can be seen that the error rate goes down as the number of iteration goes up. However the error rate oscillates.

The error rate finally converges at error rate 10.0%. Also, the boundaries are shown in Figure. **??** to Figure. **??** . It is speculated that the error rate should have been a little bit lower, but I don't have time to debug due to the short of time. Moreover, the boundary already seems to be reasonable.

Table 1: For problem 2, Adaboost.

Iteration	3	5	10	20
Error rate(%)	16.67	10.0	10.0	13.3

3 SVM with different kernels

Plots and error rate information for SVM with different kernels are shown in the Figure **??**, Figure **??** and Figure **??**.

It should be noted that the result could be different for each run.

4 Random Forest

For random forest, result is shown in Figure: **??**. Feature importance are ranked in Table. **??**. Notice, results could be difference from each run.

Table 2: Ranking the features according to importance. The ranking and feature number begins with 0.

Ranking	Feature Number	Importance
0	9	0.12
1	4	0.12
2	14	0.09
3	10	0.04
4	0	0.04
5	11	0.04
6	2	0.04
7	5	0.04
8	12	0.04
9	8	0.04
10	7	0.04
11	6	0.04
12	3	0.04
13	1	0.04
14	16	0.04
15	13	0.04
16	15	0.04
17	17	0.04
18	19	0.04
19	18	0.03

5 Gradient Boosting

Error rate for gradient boosting is: 19.33% (and in some runs it become 19.67%!). The ranking of features is presented in Table. **??**.

Table 3: Ranking the features according to importance for gradient boosting. The ranking and feature number begins with 0.

Ranking	Feature Number	Importance
0	9	0.25
1	4	0.24
2	14	0.21
3	1	0.03
4	10	0.03
5	3	0.02
6	7	0.02
7	6	0.02
8	2	0.02
9	18	0.02
10	5	0.02
11	0	0.02
12	8	0.02
13	17	0.01
14	13	0.01
15	11	0.01
16	19	0.01
17	16	0.01
18	15	0.01
19	12	0.00

6 MARS

For MARS, testing error rate is 21.14%.

7 Source codes

7.1 Comments

Finally, codes are enclosed here. It is good to remind the grader that some of the defined functions are not used in the run (especially in Q2).

7.2 Q1: Implementing Perceptron

For question 1:

```
import tensorflow as tf
import numpy as np


#This is to list all INFO
tf.logging.set_verbosity(tf.logging.INFO)

def cnn_model_fn(features, labels, mode):
  """Model function for CNN. This function is used to be called
     later in main function.
    features     :: the feature in array, size nnumber-by-784.
       Here 784 = 28*28 is the flattened representation of a hand
        written figure.
                    Features is a dict structure, {'x': <tf.
                       Tensor 'fifo_queue_DequeueUpTo:1' shape=(
                       nnumber, 784) dtype=float32>}
                    features['x'] corresponds to the acutal
                       tensor object
    labels       :: values from 0 to 9. Size nnumber-by-1. A
       tensor object
    mode         :: One of three modes: TRAIN, EVAL, PREDICT
  """

  # It is a function that used to train data.

  # Input Layer
  input_layer = tf.reshape(features["x"], [-1, 28, 28, 1])

  # Convolutional Layer #1
  conv1 = tf.layers.conv2d(
      inputs=input_layer,
      filters=32,
      kernel_size=[5, 5],
```

```
        padding="same",
        activation=tf.nn.relu)

    # Pooling Layer #1
    pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2],
        strides=2)

    # Convolutional Layer #2 and Pooling Layer #2
    conv2 = tf.layers.conv2d(
        inputs=pool1,
        filters=64,
        kernel_size=[5, 5],
        padding="same",
        activation=tf.nn.relu)
    pool2 = tf.layers.max_pooling2d(inputs=conv2, pool_size=[2, 2],
        strides=2)

    # Dense Layer
    pool2_flat = tf.reshape(pool2, [-1, 7 * 7 * 64])
    dense = tf.layers.dense(inputs=pool2_flat, units=1024,
       activation=tf.nn.relu)
    dropout = tf.layers.dropout(
        inputs=dense, rate=0.4, training=mode == tf.estimator.
           ModeKeys.TRAIN)

    # Logits Layer
    logits = tf.layers.dense(inputs=dropout, units=10)



    predictions = {
        # Generate predictions (for PREDICT and EVAL mode)
        "classes": tf.argmax(input=logits, axis=1),#Calculate class
            on the fly
        # Add `softmax_tensor` to the graph. It is used for PREDICT
            and by the
        # `logging_hook`.
        "probabilities": tf.nn.softmax(logits, name="softmax_tensor
           ")
    }

    if mode == tf.estimator.ModeKeys.PREDICT:
```

```
    return tf.estimator.EstimatorSpec(mode=mode, predictions=
        predictions)

  # Calculate Loss (for both TRAIN and EVAL modes)
  loss = tf.losses.sparse_softmax_cross_entropy(labels=labels,
      logits=logits)


  # Configure the Training Op (for TRAIN mode)
  if mode == tf.estimator.ModeKeys.TRAIN:
    optimizer = tf.train.GradientDescentOptimizer(learning_rate
        =0.001)#potimized and learning rate can change
    train_op = optimizer.minimize(
        loss=loss,
        global_step=tf.train.get_global_step())
    return tf.estimator.EstimatorSpec(mode=mode, loss=loss,
        train_op=train_op)

  # Add evaluation metrics (for EVAL mode)
  eval_metric_ops = {"accuracy": tf.metrics.accuracy(labels=
      labels, predictions=predictions["classes"])}#predictions is
      a dict and calculate classes on the fly
  return tf.estimator.EstimatorSpec(mode=mode, loss=loss,
      eval_metric_ops=eval_metric_ops)


def main(aa):


    # Load training and eval data
    mnist = tf.contrib.learn.datasets.load_dataset("mnist")

    p = int(len(mnist.train.images)*0.8) #Probably need to
        randomize

    train_data = mnist.train.images[0:p] # Returns np.array
    train_labels = np.asarray(mnist.train.labels, dtype=np.int32)
        [0:p]

    valid_data = mnist.train.images[p:]  # Returns np.array
    valid_labels = np.asarray(mnist.train.labels, dtype=np.int32)
        [p:]
```

```
eval_data = mnist.test.images # Returns np.array
eval_labels = np.asarray(mnist.test.labels, dtype=np.int32)


# Create the Estimator
mnist_classifier = tf.estimator.Estimator(model_fn=
   cnn_model_fn, model_dir="/tmp/mnist_convnet_model")
# Set up logging for predictions
tensors_to_log = {"probabilities": "softmax_tensor"}
logging_hook = tf.train.LoggingTensorHook(tensors=
   tensors_to_log, every_n_iter=1)

# Train the model
train_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": train_data},#First input, convert the numpy array
        data into a dict structure
    y=train_labels,
    batch_size=200,
    num_epochs=None,
    shuffle=True)

valid_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": valid_data},
    y=valid_labels,
    num_epochs=1,
    shuffle=False)
experiment = tf.contrib.learn.Experiment(
    mnist_classifier,
    train_input_fn,
    valid_input_fn,
    train_steps = 5000,
    eval_steps = None,
    train_steps_per_iteration = 500)


#experiment.continuous_train_and_eval()

#The rest come from tutorial
# mnist_classifier.train(
#     input_fn=train_input_fn,
```

```
    #       steps=1,
    #       hooks=[logging_hook])
    #
    #
    # # Evaluate the model and print results
    # eval_input_fn = tf.estimator.inputs.numpy_input_fn(
    #       x={"x": eval_data},
    #       y=eval_labels,
    #       num_epochs=1,
    #       shuffle=False)
    # eval_results = mnist_classifier.evaluate(input_fn=
        eval_input_fn)
    # print(eval_results)


if __name__ == '__main__':
    """Runs whole fitting program automatically"""
    import time

    start_time = time.time()
    tf.app.run()
    print("--- %s seconds ---" % (time.time() - start_time))
```