Lakshmi Chamala

DL-2

Introduction:

To implement a content arrangement and portraying it utilizing tensor stream for the characterization performed over convolution neural systems.Finally,requested to perform content characterization with CNN over another informational index that isn't already utilized as a part of the class.

Objective:

The main goal of this assignment is to learn the concepts like

Text classification

Convolution of neural networks

Work flow in tensor board

Changing the hyper parameters to compare the results.

Approaches:

The approach for the assignment can be defined as simple steps given below:

- Importing Data from Data set
- Assigning X and Y Placeholders
- Variable Weights and Bias Collection
- Construction of prediction model
- Optimize model for less errors
- Train model for the training data
- Compare the prediction and actual model variables
- Compute the accuracy of the model
- Change hyper parameters to get the results

Parameters:

Parameters used here are

Below are the parameters set for the assignment:

ALLOW_SOFT_PLACEMENT=True

BATCH_SIZE=64

CHECKPOINT_EVERY=100

DEV_SAMPLE_PERCENTAGE=0.1

DROPOUT_KEEP_PROB=0.5

EMBEDDING_DIM=128

EVALUATE_EVERY=100

FILTER_SIZES=3,4,5

L2_REG_LAMBDA=0.0

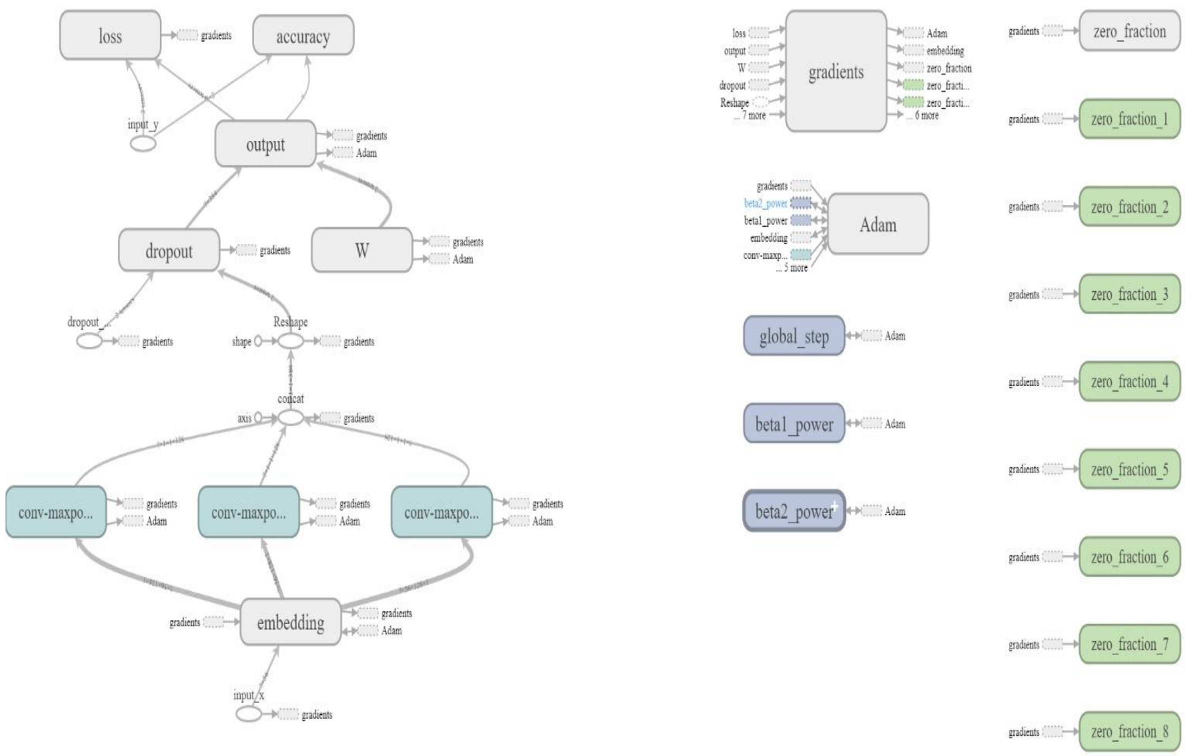
LOG_DEVICE_PLACEMENT=False NUM_CHECKPOINTS=5

NUM_EPOCHS=200

NUM_FILTERS=128

WORK FLOW:

The work flow diagram for the CNN is



Dataset:

Kaggle consumer finance data set

The dataset contains of complaints given by everyone represented in rows.
The CSV file contains 17 columns listing the reasons for the complaints from about 5 lakh customers to the Financial sector.
The figure below shows the preview of the dataset obtained from the website.

Configuration:

Python 3.6.4 is used and the main part of the code is developed using PyCharm shell software.

Evaluation & discussion:

The code snippets are provided below evaluating the performance of text classification on Kaggle Consumer Finance Complaint Dataset.

cnn.py CNN model class is created which defines the training model for the text classification. The model is set by initializing input X & Y using place holders.

```
import tensorflow as tf
import numpy as np


class TextCNN(object):
    """
    A CNN for text classification.
    Uses an embedding layer, followed by a convolutional, max-pooling and softmax layer.
    """
    def __init__(
      self, sequence_length, num_classes, vocab_size,
      embedding_size, filter_sizes, num_filters, l2_reg_lambda=0.0):

        # Placeholders for input, output and dropout
        self.input_x = tf.placeholder(tf.int32, [None, sequence_length], name="input_x")
        self.input_y = tf.placeholder(tf.float32, [None, num_classes], name="input_y")
        self.dropout_keep_prob = tf.placeholder(tf.float32, name="dropout_keep_prob")

        # Keeping track of l2 regularization loss (optional)
        l2_loss = tf.constant(0.0)

        # Embedding layer
        with tf.device('/cpu:0'), tf.name_scope("embedding"):
            self.W = tf.Variable(
                tf.random_uniform([vocab_size, embedding_size], -1.0, 1.0),
                name="W")
            self.embedded_chars = tf.nn.embedding_lookup(self.W, self.input_x)
            self.embedded_chars_expanded = tf.expand_dims(self.embedded_chars, -1)
```

```python
# Create a convolution + maxpool layer for each filter size
pooled_outputs = []
for i, filter_size in enumerate(filter_sizes):
    with tf.name_scope("conv-maxpool-%s" % filter_size):
        # Convolution Layer
        filter_shape = [filter_size, embedding_size, 1, num_filters]
        W = tf.Variable(tf.truncated_normal(filter_shape, stddev=0.1), name="W")
        b = tf.Variable(tf.constant(0.1, shape=[num_filters]), name="b")
        conv = tf.nn.conv2d(
            self.embedded_chars_expanded,
            W,
            strides=[1, 1, 1, 1],
            padding="VALID",
            name="conv")
        # Apply nonlinearity
        h = tf.nn.relu(tf.nn.bias_add(conv, b), name="relu")
        # Maxpooling over the outputs
        pooled = tf.nn.max_pool(
            h,
            ksize=[1, sequence_length - filter_size + 1, 1, 1],
            strides=[1, 1, 1, 1],
            padding='VALID',
            name="pool")
        pooled_outputs.append(pooled)
```

```python
# Combine all the pooled features
num_filters_total = num_filters * len(filter_sizes)
self.h_pool = tf.concat(pooled_outputs, 3)
self.h_pool_flat = tf.reshape(self.h_pool, [-1, num_filters_total])

# Add dropout
with tf.name_scope("dropout"):
    self.h_drop = tf.nn.dropout(self.h_pool_flat, self.dropout_keep_prob)

# Final (unnormalized) scores and predictions
with tf.name_scope("output"):
    W = tf.get_variable(
        "W",
        shape=[num_filters_total, num_classes],
        initializer=tf.contrib.layers.xavier_initializer())
    b = tf.Variable(tf.constant(0.1, shape=[num_classes]), name="b")
    l2_loss += tf.nn.l2_loss(W)
    l2_loss += tf.nn.l2_loss(b)
    self.scores = tf.nn.xw_plus_b(self.h_drop, W, b, name="scores")
    self.predictions = tf.argmax(self.scores, 1, name="predictions")

# Calculate mean cross-entropy loss
with tf.name_scope("loss"):
    losses = tf.nn.softmax_cross_entropy_with_logits(logits=self.scores, labels=self.input_y)
    self.loss = tf.reduce_mean(losses) + l2_reg_lambda * l2_loss

# Accuracy
with tf.name_scope("accuracy"):
    correct_predictions = tf.equal(self.predictions, tf.argmax(self.input_y, 1))
    self.accuracy = tf.reduce_mean(tf.cast(correct_predictions, "float"), name="accuracy")
```

```python
FLAGS = tf.flags.FLAGS
FLAGS._parse_flags()
print("\nParameters:")
for attr, value in sorted(FLAGS.__flags.items()):
    print("{}={}".format(attr.upper(), value))
print("")


# Data Preparation
# ==================================================

# Load data
print("Loading data...")
x_text, y = data_helper.load_data_and_labels(FLAGS.positive_data_file, FLAGS.negative_data_file)

# Build vocabulary
max_document_length = max([len(x.split(" ")) for x in x_text])
vocab_processor = learn.preprocessing.VocabularyProcessor(max_document_length)
x = np.array(list(vocab_processor.fit_transform(x_text)))

# Randomly shuffle data
np.random.seed(10)
shuffle_indices = np.random.permutation(np.arange(len(y)))
x_shuffled = x[shuffle_indices]
y_shuffled = y[shuffle_indices]

# Split train/test set
# TODO: This is very crude, should use cross-validation
dev_sample_index = -1 * int(FLAGS.dev_sample_percentage * float(len(y)))
x_train, x_dev = x_shuffled[:dev_sample_index], x_shuffled[dev_sample_index:]
y_train, y_dev = y_shuffled[:dev_sample_index], y_shuffled[dev_sample_index:]


with tf.Graph().as_default():
    session_conf = tf.ConfigProto(
      allow_soft_placement=FLAGS.allow_soft_placement,
      log_device_placement=FLAGS.log_device_placement)
    sess = tf.Session(config=session_conf)
    with sess.as_default():
        cnn = TextCNN(
            sequence_length=x_train.shape[1],
            num_classes=y_train.shape[1],
            vocab_size=len(vocab_processor.vocabulary_),
            embedding_size=FLAGS.embedding_dim,
            filter_sizes=list(map(int, FLAGS.filter_sizes.split(","))),
            num_filters=FLAGS.num_filters,
            l2_reg_lambda=FLAGS.l2_reg_lambda)

        # Define Training procedure
        global_step = tf.Variable(0, name="global_step", trainable=False)
        optimizer = tf.train.AdamOptimizer(1e-3)
        grads_and_vars = optimizer.compute_gradients(cnn.loss)
        train_op = optimizer.apply_gradients(grads_and_vars, global_step=global_step)

        # Keep track of gradient values and sparsity (optional)
        grad_summaries = []
        for g, v in grads_and_vars:
            if g is not None:
                grad_hist_summary = tf.summary.histogram("{}/grad/hist".format(v.name), g)
                sparsity_summary = tf.summary.scalar("{}/grad/sparsity".format(v.name), tf.nn.zero_fraction(g))
                grad_summaries.append(grad_hist_summary)
                grad_summaries.append(sparsity_summary)
        grad_summaries_merged = tf.summary.merge(grad_summaries)
```

```
        # Output directory for models and summaries
        timestamp = str(int(time.time()))
        out_dir = os.path.abspath(os.path.join(os.path.curdir, "runs", timestamp))
        print("Writing to {}\n".format(out_dir))

        # Summaries for loss and accuracy
        loss_summary = tf.summary.scalar("loss", cnn.loss)
        acc_summary = tf.summary.scalar("accuracy", cnn.accuracy)

        # Train Summaries
        train_summary_op = tf.summary.merge([loss_summary, acc_summary, grad_summaries_merged])
        train_summary_dir = os.path.join(out_dir, "summaries", "train")
        train_summary_writer = tf.summary.FileWriter(train_summary_dir, sess.graph)

        # Dev summaries
        dev_summary_op = tf.summary.merge([loss_summary, acc_summary])
        dev_summary_dir = os.path.join(out_dir, "summaries", "dev")
        dev_summary_writer = tf.summary.FileWriter(dev_summary_dir, sess.graph)

        # Checkpoint directory. Tensorflow assumes this directory already exists so we need to create it
        checkpoint_dir = os.path.abspath(os.path.join(out_dir, "checkpoints"))
        checkpoint_prefix = os.path.join(checkpoint_dir, "model")
        if not os.path.exists(checkpoint_dir):
            os.makedirs(checkpoint_dir)
        saver = tf.train.Saver(tf.global_variables(), max_to_keep=FLAGS.num_checkpoints)

        # Write vocabulary
        vocab_processor.save(os.path.join(out_dir, "vocab"))
```

```
    # Initialize all variables
    sess.run(tf.global_variables_initializer())

    def train_step(x_batch, y_batch):
        """
        A single training step
        """
        feed_dict = {
          cnn.input_x: x_batch,
          cnn.input_y: y_batch,
          cnn.dropout_keep_prob: FLAGS.dropout_keep_prob
        }
        _, step, summaries, loss, accuracy = sess.run(
            [train_op, global_step, train_summary_op, cnn.loss, cnn.accuracy],
            feed_dict)
        time_str = datetime.datetime.now().isoformat()
        print("{}: step {}, loss {:g}, acc {:g}".format(time_str, step, loss, accuracy))
        train_summary_writer.add_summary(summaries, step)
```

```python
# Generate batches
batches = data_helper.batch_iter(
    list(zip(x_train, y_train)), FLAGS.batch_size, FLAGS.num_epochs)
# Training loop. For each batch...
for batch in batches:
    x_batch, y_batch = zip(*batch)
    train_step(x_batch, y_batch)
    current_step = tf.train.global_step(sess, global_step)
    if current_step % FLAGS.evaluate_every == 0:
        print("\nEvaluation:")
        dev_step(x_dev, y_dev, writer=dev_summary_writer)
        print("")
    if current_step % FLAGS.checkpoint_every == 0:
        path = saver.save(sess, checkpoint_prefix, global_step=current_step)
        print("Saved model checkpoint to {}\n".format(path))
```

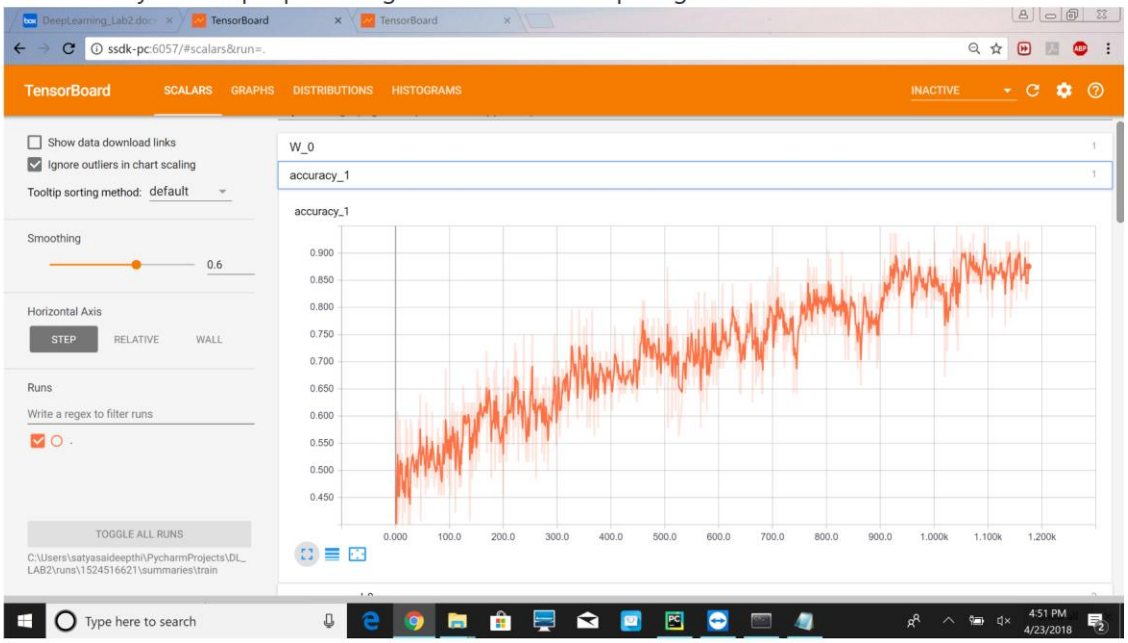
Using data_eval.py file, we would initially tokenize the input data that is considered for evaluation. This would remove the unnecessary content from the dataset and make it ready for evaluation. Now we need to evaluate the effects of changing the parameters on accuracy and loss. Now for this purpose we change the learning rate parameter for this evaluation purpose.

Result:

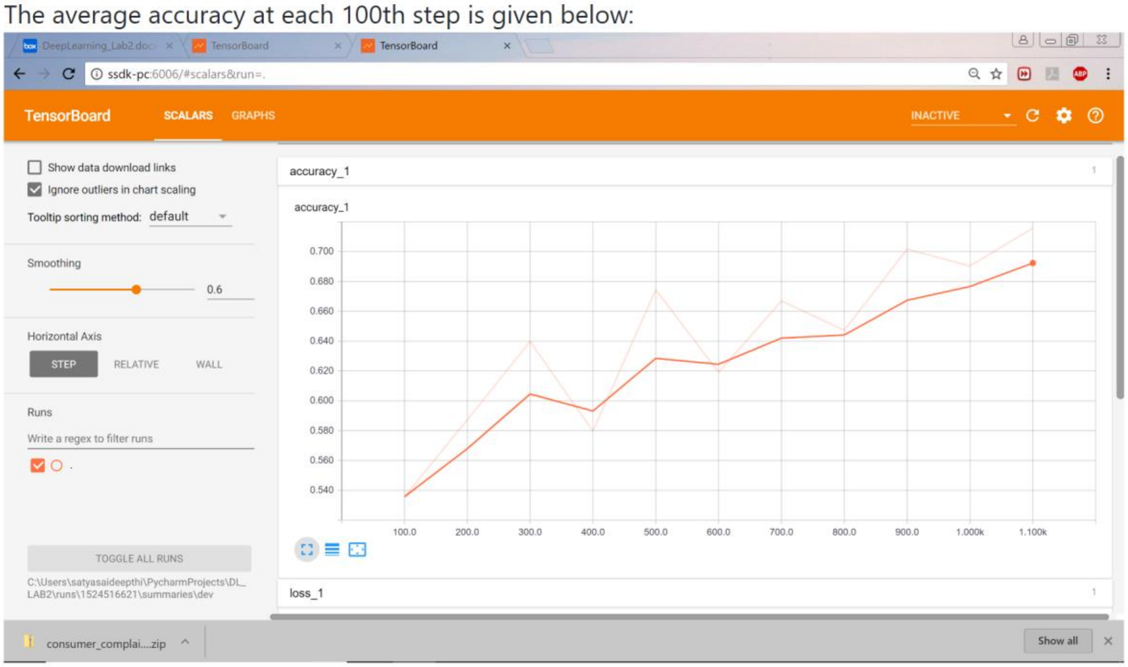
For 1000 steps:

```
2018-04-23T16:54:59.891989: step 101, loss 1.09087, acc 0.59375
2018-04-23T16:55:00.381836: step 102, loss 1.16172, acc 0.625
2018-04-23T16:55:00.845710: step 103, loss 1.25747, acc 0.53125
2018-04-23T16:55:01.375097: step 104, loss 0.988432, acc 0.640625
2018-04-23T16:55:01.969018: step 105, loss 1.35296, acc 0.53125
2018-04-23T16:55:02.447357: step 106, loss 1.07721, acc 0.625
2018-04-23T16:55:02.911188: step 107, loss 1.26589, acc 0.515625
2018-04-23T16:55:03.381021: step 108, loss 1.51506, acc 0.3125
2018-04-23T16:55:03.873373: step 109, loss 1.68325, acc 0.46875
```

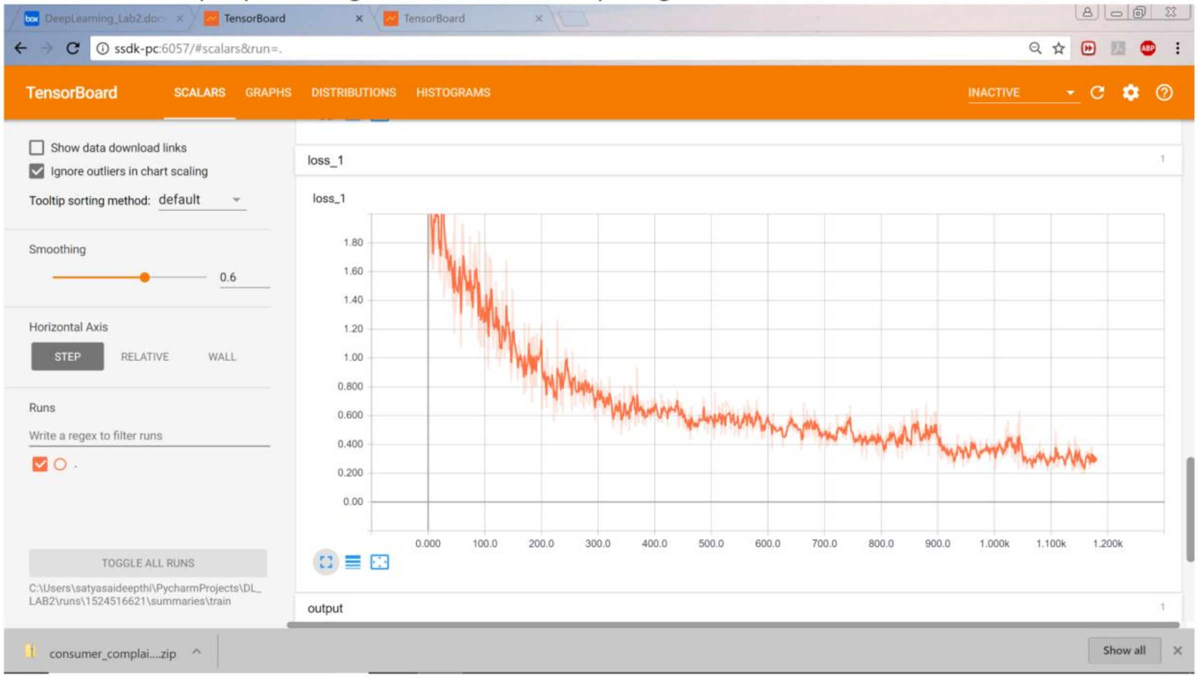
The accuracy for sample percentage = 0.1 for 1000 steps is given below:



The average accuracy at each 100th step is given below:



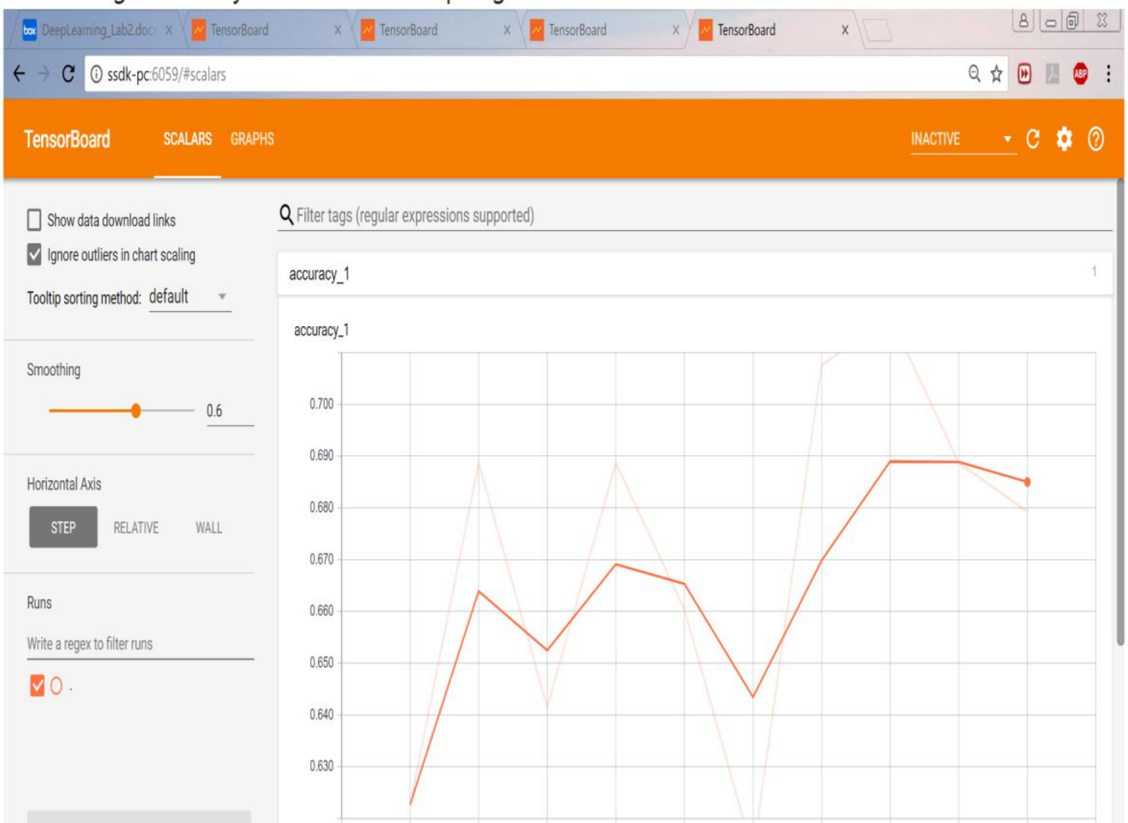
The loss for sample percentage = 0.1 for 1000 steps is given below:



The average loss at each step 100th step is below:

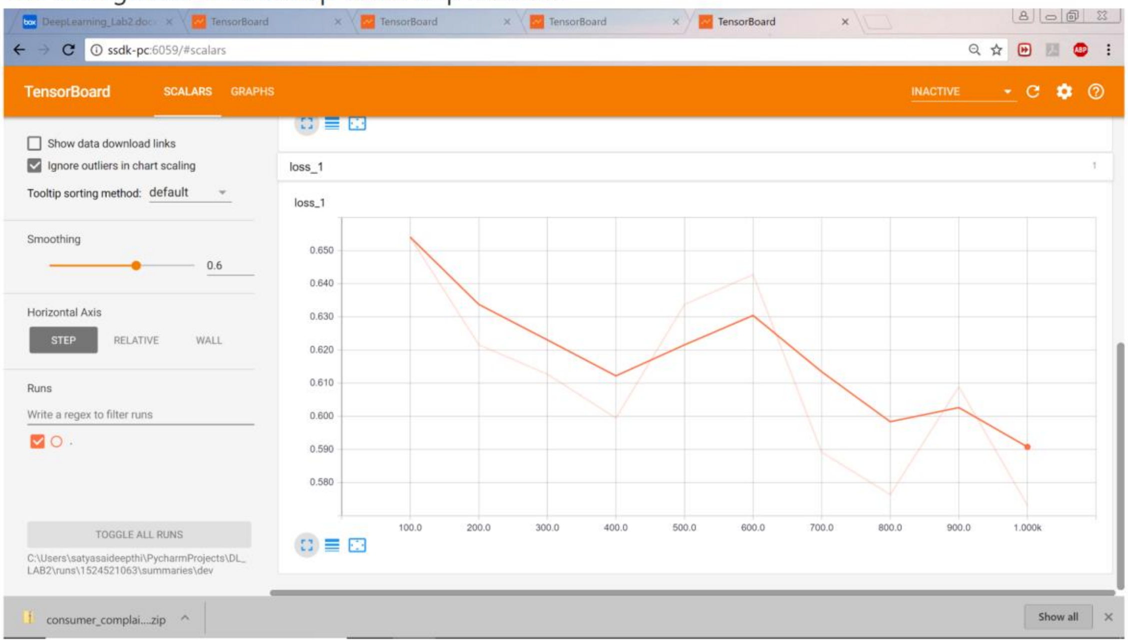
For 0.01

The average accuracy at each 100th step is given below:



The loss for sample percentage = 0.01 for 1000 steps is given below:

The average loss at each step 100th step is below:



Conclusion:

Performing Text classification on Kaggle Consumer Finance Complaints dataset gives the following conclusions:

- By increasing the sample percentage, the accuracy value increases.
- By increasing the sample percentage, the cross-entropy loss value decreases.